Sparta: Practical Anonymity with Long-Term Resistance to Traffic Analysis

Kyle Fredrickson
UC Santa Cruz
kyfredri@ucsc.edu

Ioannis Demertzis
UC Santa Cruz
idemertz@ucsc.edu

James P. Hughes
UC Santa Cruz
japhughe@ucsc.edu

Darrell D.E. Long
UC Santa Cruz
darrell@ucsc.edu

***Abstract*—Existing metadata-private messaging systems are either non-scalable or vulnerable to long-term traffic analysis. Approaches that mitigate traffic analysis attacks often suffer from unrealistic and unimplementable assumptions or impose system-wide bandwidth restrictions, degrading usability, and performance. In this work, we present a new model for metadata-private communication systems—deferred retrieval—that guarantees traffic analysis resistance under realistic, implementable user assumptions. We introduce Sparta systems, practical and scalable instantiations of deferred retrieval that are distributable, achieve high throughput, and support multiple concurrent conversations without message loss. Specifically, we present three Sparta constructions optimized for different scenarios: (i) low-latency, (ii) high-throughput in shared-memory environments (multi-thread implementations), and (iii) high throughput in shared-nothing (distributed) environments. Our low latency Sparta supports latencies of less than one millisecond, while our high-throughput Sparta can scale to deliver over 700,000 100B messages per second on a single 48-core server.**

1. Introduction

Today's messaging systems, like WhatsApp, iMessage, and Signal, use end-to-end encryption to protect message contents. However, this does not hide metadata—information about who communicates with whom, when, and how much—which remains visible to systems and network observers. Metadata is highly sensitive and valuable; as former NSA general counsel, Stuart Baker, said, "metadata absolutely tells you everything about somebody's life. If you have enough metadata, you don't really need content". For example, if an employee sends encrypted files to a journalist who later exposes company corruption, the employee becomes a prime suspect based on metadata alone. Messaging systems must therefore protect both content and metadata to be truly private.

Metadata privacy has been a topic of significant interest over the past decades. Despite this, Tor [1] remains the only widely used metadata-private communication system. Tor is well-known to be vulnerable to global adversaries capable of observing all network links; however, the lesson of Tor is not that the non-global adversary model is too weak (though it is), but that traffic analysis is a much more pertinent concern [2], [3]. While significant research efforts have been devoted to designing scalable, metadata-private messaging systems secure against global adversaries [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], the problem of traffic analysis has received much less attention.

Traffic Analysis Attacks. Traffic analysis attacks aim to infer protected information, such as who is talking to whom, by identifying correlations in traffic patterns at users. For example, consider a trusted messaging service that never discloses how a message was routed. Such a system would be secure against a global adversary by assumption of trust. However, if we observe Alice sending a message to the service and Bob receiving a message shortly after, this strongly suggests they are communicating. These attacks are statistical and improve with time and additional observations [2], [18], [19], [20].

Mitigating Traffic Analysis Attacks & Limitations. To mitigate traffic analysis attacks, existing systems either broadcast or impose global bandwidth restrictions. Broadcast is robust against traffic analysis attacks since messages are sent to all potential recipients simultaneously, hiding the specific sender-receiver pairs. However, broadcast becomes infeasible as the number of users increases.

Bandwidth restrictions ensure all users' traffic looks identical, eliminating correlations between communicating users. These come in two forms: restrictions on input rates and restrictions on output rates.

Asynchronous systems [13], [21], [22], [23] impose restrictions on output rates, building these restrictions into the system by implementing a set of mailboxes for each user with fixed size, k. Senders deposit messages into these mailboxes, and receivers explicitly fetch to receive their messages. On a fetch, the entire mailbox, *i.e.*, k messages, is downloaded whether or not the mailbox is full. While asynchronous systems allow users to fetch independently of the other users, to prevent traffic analysis, many assume all users will fetch at a fixed rate. [13], [21] Combined with the fixed mailbox size, this results in a globally fixed output rate, denoted k_{out}. While there is often nothing to prevent users from fetching more frequently, these systems face an additional problem. If a user receives more messages than the capacity of the mailbox, messages collide, corrupting them and preventing their correct delivery. Mailbox sizes,

k, are most often 1, [13], [21] effectively prohibiting users from having multiple concurrent conversations.

Synchronous systems, *e.g.*, mixnets [4], [7], [8], [9], [10], [16], [24], [25] and secret sharing based schemes [5], [6], [12], [26], impose restrictions on input rates by making assumptions about user behavior, *i.e.*, all users will send $k = 1$ messages each round—the "one message assumption". At the end of the round after messages are accrued, all messages are randomly permuted and delivered. Because all users are bound by the same round schedule and are required to send equal volumes of messages, this results in a globally fixed input rate, denoted k_{in}. But this assumption is unrealistic and unimplementable. Users cannot be expected to be online to send a message each round, especially in mobile environments where they may lose connectivity, and systems cannot control how users behave. Groove [11] addressed this problem by designing components to enforce that one message per user is submitted each round to its internal mixnet, but still suffers from globally set input rates.

While global bandwidth restrictions like the one message assumption prevent traffic analysis, [27], [28], [29] they lead to significant and previously unacknowledged problems in real deployments. If a user of a synchronous system sends at a higher rate than the round rate, these additional messages will be delayed/deferred to subsequent rounds. By Little's law [30], we know that this outbound message queue will grow without bound, translating to unbounded latency for these highly active users. If, in response to this, we increase the bandwidth rate to be higher than the rate of the most active user, this will result in much higher overhead for low-activity users. Thus, even if all users could realistically meet this assumption, global bandwidth restrictions would lead to significant performance problems.

Our goal is to build practical metadata-private messaging systems that are secure in the long term. Toward this goal, we raise two questions.

1) Can systems provide long-term traffic analysis resistance with practical performance and realistic user assumptions? And if so,
2) Can they be securely and scalably implemented?

In this work. We affirmatively address both questions as we summarize in Fig. 1. In §3.2 we introduce a new model—deferred retrieval—for how metadata-private messaging systems can operate so that traffic patterns at communicating users do not reveal correlations. Deferred retrieval provides traffic analysis resistance under realistic user assumptions, without imposing global bandwidth restrictions, and with practical latencies and overheads in deployed systems. In §4, we introduce Sparta systems, which securely and efficiently support deferred retrieval. These systems are scalable, achieve high throughput, and support multiple concurrent conversations without message loss. Sparta systems implement deferred retrieval using Intel SGX and oblivious algorithms and data structures. Our implementations of Sparta ensure that all operations on the server leak no more than users' traffic patterns, which themselves are secure against traffic analysis due to deferred retrieval.

We present three Sparta constructions that provide the listed system and usability properties optimized for different use cases. Sparta-LL is optimized for low-latency and based on oblivious data structures. It supports latency of less than one millisecond on database sizes of 2^{20}. Sparta-SB is based on sort and can scale to deliver over 700,000 100B messages per second on a single 48-core server with a database of size 2^{20}. Sparta-D is distributable and can scale to database sizes of 2^{23} with less than one second of latency.

Contributions. We offer three core contributions that lead to these results:

1) First, we precisely define traffic analysis resistance. To achieve this, we propose a new framework for describing traffic leakage that is system-independent and can be used to categorize all existing work. This is the first framework that captures long-term statistical leakages and identifies the specific features that enable traffic analysis attacks. Using this, we observe that we can relax existing proposals for leakage without disclosing information that could be used for long-term correlation of communicating users.
2) We propose a new model for metadata private communication systems—deferred retrieval—that achieves traffic analysis resistance with less stringent assumptions on users than prior work. We evaluate the practical costs of this model under real email workloads and find that it is orders of magnitude cheaper than existing proposals for traffic analysis systems and supports sub-minute latencies with low network overhead. This remains true even when we compare our systems with non-optimal parameters to prior works with optimal parameters. To our knowledge, this is the first evaluation of the costs associated with methods for achieving long-term traffic analysis resistance.
3) Finally, we present three implementations of deferred retrieval. Sparta-LL is optimized for low processing latency, Sparta-SB is optimized for high throughput on existing workloads, and Sparta-D is designed to be distributable. These systems impose no global bandwidth restrictions on users, place no assumptions on the number of contacts users can have, and do not suffer from message loss like many prior systems. In our experimental evaluation, we show that our systems achieve high throughput. Our implementations are aligned with the Signal Messaging App, which combines hardware enclaves (Intel SGX) with oblivious algorithms and data structures to implement private contact discovery.

Limitations. The latency and overhead of deferred retrieval is highly dependent on the fetch rates that users choose. Setting these rates is not trivial, as they must be set to

accommodate users' future behavior (prior work neglected this problem by setting rates to, *e.g.*, one message per round). Properly setting these rates in practice would require additional study of users' behavior, e.g., [31], [32]. We address this limitation in §5.1 by evaluating our systems using progressively less information about the optimal rate.

Though deferred retrieval will always outperform global bandwidth restrictions (see §3.2), the degree of its advantage is dependent on characteristics of the workload. Unfortunately, suitable datasets are rare. Enron [33] is the only dataset that has previously been used in the context of anonymity research [34]. Seattle [35] by contrast is a new dataset that we introduce for this purpose.

2. Preliminaries

Trusted Execution Environments/Hardware Enclaves. Our systems are implemented using trusted execution environments/hardware enclaves (*e.g.*, Intel SGX [36], ARM TrustZone [37], AMD Enclave [38]). A hardware enclave resides on an untrusted operating system and offers enhanced security functionalities to ensure confidential computing, including sealing, isolation, and remote attestation. Sealing enables the enclave to encrypt data using its private sealing key. Remote attestation verifies that the enclave has been initialized with the expected code and data. Isolation ensures the secure separation of a portion of the system's memory, known as the Enclave Page Cache (EPC), which stores user data and executable code. Our Sparta implementation targets Intel SGXv2 [39], which introduced flexible and dynamic EPC memory allocation and larger EPC sizes, allowing applications to scale larger. Importantly, hardware enclaves do not hide memory access patterns [40], [41] or control-flow [42], thus it is standard to ensure that the algorithms running in hardware enclaves are oblivious [43], [44], [45], [46], [47], [48], [49], [50].

Obliviousness. An algorithm or data structure is considered oblivious if, for any two sequences of operations of the same size, the resulting memory accesses and code traces are computationally indistinguishable—even assuming the presence of an adversary who can observe all memory accesses and network communications. Intuitively, this means that the execution of an algorithm leaks only the length of the inputs and is independent of the value.

Oblivious RAM (ORAM). ORAM [51], [52], [53], [54], [55], [56] is a compiler that transforms memory access patterns to conceal the original access sequences. This ensures that the observed memory accesses do not reveal any information about the logical accesses. ORAM is defined by two main protocols: $\mathsf{setup}(\lambda, N)$ and $\mathsf{access}(\mathsf{op}, i, v)$. The setup protocol initializes an ORAM of size N with security parameter λ, while access returns the element at index i if $\mathsf{op} = \mathsf{read}$, or writes v at index i and returns a dummy element if $\mathsf{op} = \mathsf{write}$. The access protocol manages the actual memory accesses, ensuring they remain indistinguishable from random, thereby hiding both the operations performed and the index i.

Oblivious Map (OMAP). An OMAP is a privacy-preserving variant of a regular map that conceals the type and content of operations. OMAP is defined by three main protocols: $\mathsf{setup}(\lambda, N)$, $\mathsf{put}(k, v)$, and $\mathsf{get}(k)$. The setup protocol is defined similarly to that for ORAMs. The put protocol inserts a new (k, v) pair into the data structure. The get protocol retrieves the value associated with a key. All sequences of data accesses (get/put) of equal length are indistinguishable (see Wang *et al.* [57] for details).

Oblivious Sort. An oblivious sorting algorithm sorts an array of N elements without revealing any information about the array beyond its length. We use bitonic sort [58], an efficient and well-known oblivious sorting algorithm. The algorithm has a time complexity of $O(N \log^2 N)$ and performs sorting through a series of compare-and-swap operations, making it suitable for parallel execution. We implement the multi-threaded bitonic sort from Ngai *et al.* [49], which is in practice the most efficient choice for a single server. Other oblivious sorting algorithms with $O(N \log N)$ complexity, including the bucket sort from Ngai *et al.* [49], perform worse in a single-server scenario.

Oblivious Compaction. Given an array of N elements, where some elements are tagged with a bit, an oblivious compaction algorithm [46] arranges the elements such that the tagged elements appear at the beginning of the list without disclosing the tags (and any other information about the memory access patterns). Order-preserving compaction maintains the relative order of the tagged elements.

Oblivious Selection. Oblivious selection is a primitive that allows us to select one of two values based on a condition without branching. Oblivious selection of a or b based on a condition c, can be computed as $(!(\mathsf{c}-1)\&\mathsf{a})|((\mathsf{c}-1)\&\mathsf{b})$. We include if statements in our pseudocode, but this is for readability only. In reality we implement these conditionals using oblivious select.

Adversary Model. Like most prior work in the anonymity literature, we focus on a global adversary that can monitor all network links. However, unlike much prior work, we extend the adversary in three important ways. (1) We assume an active adversary that can modify all network traffic and that can participate in the protocol. (2) We assume the adversary can observe traffic for an arbitrarily long time. (3) In alignment with prior works combining obliviousness with hardware enclaves [43], [44], [45], [46], [47], [48], [49], [50], we assume a powerful attacker who can observe network traffic, compromise all server software up to and including privileged software, and control the operating system, but cannot breach the secure processor or access its secret key. This attacker can observe memory accesses, data on the memory bus, in main memory, and code traces. Hardware side-channel attacks [41], [59], [60], [61], [62] (*e.g.*, power consumption analysis) and denial-of-service attacks are out of scope. Techniques to mitigate such attacks from prior works can be integrated alongside our approach.

System	Weak Assumptions/TA Resistance	Correctness	Global Bandwidth Limitation	Throughput
Sabre [21]	✓	✗	Yes	200KB/s
Pung/Seal PIR [22]	✓	✗	Yes	256KB/s
Groove [11]	✓	✓	Yes	3.6MB/s
Sparta (this work)	✓	✓	No	**53MB/s**

Figure 1. Our work, Sparta, is the first system that imposes no system-wide bandwidth restrictions on inbound and outbound traffic while providing long-term traffic analysis resistance under realistic user assumptions. Specifically, users must not fetch based on their actual received traffic. The lack of global bandwidth restrictions implies that in a real deployment, Sparta, operating according to the assumptions of deferred retrieval, will perform significantly better than existing work (see §5.1). Unlike some prior works, Sparta is correct, *i.e.*, it will not drop messages if users receive more than a globally set k messages between fetches. The throughput is provided for context. We arrived at these numbers by taking best-reported numbers in existing works' evaluations independently of the database sizes. The throughput for Sparta is taken from Sparta-SB on the Legacy workload with a database size of 2^{23}, while the throughput for Sabre, Pung, and Groove is on databases of size 2^{15}, 32K, and 2M, respectively.

Performance Metrics. For our systems evaluations, we focus on three metrics: latency, throughput, and network overhead. In this context, latency denotes the time taken to process messages by the system, which we denote l_p, and throughput is the number of messages a system can process per unit time. This latency is one component of the actual latency users can expect when using traffic analysis resistant systems. In this paper, we expand metrics to also include latency due to assumptions and parameters set by the system. To our knowledge, no other work has considered these as part of the costs of their systems or attempted to measure these costs. Formally,

$$L = l_p + l_u + l_s, \quad (1)$$

where in addition to l_p we consider the latency from messages waiting at the user before it can be sent l_u, and l_s, the latency due to the message waiting at the server before they can be delivered. We also consider network overhead, which is the amount of network traffic sent/received per relevant party per unit time. That is, we say, Alice's overhead is 42B/s if she must send this traffic to maintain traffic analysis resistance independently of her actual traffic. These metrics are common in the theoretical literature with trilemmas from Das *et al.* [63] suggesting an inherent tradeoff between traffic analysis resistance, latency, and overhead. As an example, to illustrate the sources of these quantities, consider a synchronous system such as a mixnet [4] that requires users to send exactly one message per round to prevent intersection attacks [18], [27], [34]. The overhead is one message per round. If users wish to send two messages in a round, this second message will be deferred to the next round, increasing latency through l_u. Similarly, if the system sets a longer round parameter to allow all users to participate, this would increase latency through l_s.

3. Traffic Analysis Resistance via Deferred Retrieval

For a messaging system to truly be secure in the long-term, it must (1) operate in such a way that traffic does not leak correlations between communicating users and (2) be implemented in such a way that systems leak only those permitted traffic patterns. In this section, we address the first point. We develop deferred retrieval—a new model for how metadata-private communication systems can operate so that traffic patterns do not leak correlations between communicating users. Toward this end, we first develop a framework for quantifying leakages and reasoning about traffic analysis resistance and then design deferred retrieval.

3.1. Traffic Analysis Resistance

Currently in the literature we do not have a formal, system-independent way to quantify or reason about the security of systems' traffic leakages. We know certain points in the space are insecure against traffic analysis, *e.g.*, Tor [1] has surveys devoted to attacks against it [3], while others, *e.g.*, broadcast are not vulnerable to traffic analysis. In this section we describe the first such framework.

Modeling Traffic. As the model underpinning our framework we propose *communication states*. In communication states we assume that messages are split/padded to a fixed length (*e.g.*, 100B) and re-encrypted to prevent input and output from the system from being linked on features of the traffic. This allows us to exclude content from consideration, assuming a computationally bounded adversary.

Definition 3.1 (Communication State). *A communication state, C, is a set of tuples (s, r, t), where s denotes the sender, r denotes the recipient, and t denotes the time the message was sent.*

One advantage of this model is that it specifies nothing about the system. Earlier work defined batches for synchronous systems in a similar way [64], [65], but because they did not include timing information they (1) could only capture synchronous systems, (2) could not capture susceptibility to intersection attacks, and (3) could not capture *all* traffic that was sent through a system over the long-term.

Traffic Leakage. The goal of metadata-private messaging systems is to reduce the amount of information adversaries learn about communication states. We can formalize this loss of information by specifying leakage functions on communication states in the style of MPC [66], [67], [68] and Searchable Encryption [69], [70], [71], [72], [73], [74], [75], [76], [77], [78]. In general, leakage functions are composed of two subleakages: the leakage on the sender

Family	Leakage	Traffic Analysis Resistant
Onion Routers	$(S_t, R_{t+\epsilon})$	✗
Broadcast	S_t	✓
Synchronous (single round)	(S_t, R)	✓
Synchronous (multi-round)	$(S_t, R_{[t_i,t_{i+1}]})$	✗
Synchronous (one-message in)	$R_{[t_i,t_{i+1}]}$	✓

Figure 2. The leakages of some common families of metadata private messaging systems. Families marked with "✓" are secure against traffic analysis, while those marked with "✗" are not.

side of the communication system and the leakage on the receiver side of the communication system. Though there are many possible leakage functions, we consider the following as they are particularly useful for modeling existing work.

1) $S_t = \{(s,t) | (s,r,t) \in C\}$, the sender and time of every message. This captures that s sent a message at time t, but contains no information about who the message was addressed to.
2) $R_{[t_i,t_{i+1}]} = \{r | (s,r,t) \in C \text{ and } t \in [t_i, t_{i+1}]\}$, all receivers who were sent a message during some interval. This leakage is implemented by synchronous systems that batch and permute messages, thus breaking any link between the time they were sent and the time they were delivered beyond that it was sent during some time interval.
3) $R_{f(t)} = \{(r, f(t)) | (s,r,t) \in C \text{ and } t \leq f(t)\}$, all receivers and some function, $f(t) \geq t$. This models continuous systems that may add some random delay [79] before delivering messages. The condition that $f(t) \geq t$ captures that messages cannot be delivered before they are sent. And finally,
4) $R = \{r | (s,r,t) \in C\}$, the total volume of messages receiver by each user. This contains no timing information per tuple but can be used to compute the **total** number of messages a user received during the lifetime of the system. Importantly, because there is no timing information, an adversary cannot learn any information about the volume of message a user received during a smaller time interval.

Traffic Analysis & Resistance. Using these leakages we can quantify the leakages of existing systems and assumptions and reason about the properties of leakage functions that permit and do not permit traffic analysis attacks. In Fig. 2 we give the leakages of some existing schemes.

For example, the leakage of onion routers (Tor [1]), can be captured as $(S_t, R_{t+\epsilon})$, where $\epsilon > 0$ is some small random delay due to uncertainty in networking. By observing this leakage, we can simply match tuples $(s,t) \in S_t$ and $(r, t+\epsilon) \in R_{t+\epsilon}$ on t to reconstruct the tuple (s,r,t), deanonymizing the connection. This agrees with the observation that Tor is vulnerable to traffic analysis [3].

On the other hand, broadcast's network patterns are determined from S_t alone, since for each $(s,t) \in S_t$ we simply deliver a message to all recipients. Broadcast is not vulnerable to traffic analysis [80]. In fact, with this leakage the only way we could match users would be if there was a discernible correlation in sending behavior, *e.g.*, if every time Alice sends Bob sends (responds) shortly after. Unfortunately, broadcast is not scalable.

Synchronous systems batch and permute requests, breaking the connection between input and output within batches. For a single round then, the timing of the output is independent of the input. We express this leakage as (S_t, R), because only the timing of sent messages and the volume of received messages is leaked. With more than one round, we can observe the output of multiple batches, thus the leakage is $(S_t, R_{[t_i,t_{i+1}]})$. Utilizing changes in the sender and receiver traffic, adversaries can correlate communicating users using intersection/statistical disclosure attacks [27], [34], [81]. Intersection attacks are not possible with a single observation [18] because we cannot observe changes between observations.

The conventional defense to intersection attacks is to assume that all users will participate with a fixed k messages per round (most often $k = 1$ [5], [6], [9], [11], [13], [21]). This fixes the sender side of the leakage function, reducing the leakage to $R_{[t_i,t_{i+1}]}$, and ensures that variations in the recipient traffic cannot be correlated with variations in the sender traffic [27], [82].

Though synchronous systems can be made secure against traffic analysis by assuming that all users send one message in every round, such an assumption is unrealistic [11] and unenforceable, except by banning users that do not participate in each round [83]. Even if it were, we show in §5.1 that achieving this leakage by imposing the same bandwidth restriction globally on all users of the system leads to prohibitively high costs. The leakage function (S_t, R), on the other hand, is attractive for two reasons. First, and most importantly, it is the leakage function of single-round synchronous systems, which are acknowledged to be secure against traffic analysis [28], [84]. Second, it inherently allows users to receive different amounts of traffic and thus does not impose global bandwidth restrictions. Unfortunately, single-round synchronous systems are unsuited to messaging because users cannot respond to messages.

3.2. Deferred Retrieval

Deferred retrieval is a new way for a metadata-private messaging system to operate such that traffic does not leak correlations between communicating users. It is not a particular system, but rather is a class of systems that can be implemented in a variety of ways (see §4) and a variety

```
send(r, m; US, MS)
 1: next ← U(0, 2^l − 1)
 2: rand ← U(0, 2^l − 1)
 3: (head, tail) ← US.update(r, (head, next))
 4: MS.access(write, rand, (r, tail, next, m))
fetch(r, k; US, MS)
 1: (first, last) ← US.get(r)
 2: x = first, M = {}
 3: while |M| < k do
 4:    if x ≠ last then
 5:       (r, curr, next, m) ← MS.access(read, x, ∅)
 6:       x = next
 7:    else
 8:       (−, −, −, m) ← MS.access(read, dummy, ∅)
 9:    end if
10:    M = M ∪ {m}
11: end while
12: US.put(r, (x, last))
13: return M
```

Figure 3. The operations of Sparta-LL. Because US and MS are oblivious data structures, accesses leak nothing beyond their size. We assume for traffic analysis resistance that k is at most a function of (S_t, R).

of trust models. Deferred retrieval is a sub-category of asynchronous systems designed to meet our traffic leakage goal, (S_t, R), under weak assumptions on the behavior of users. In it users push messages into a system's state and later fetch k_i messages. However, it differs from existing asynchronous systems in two important points.

First, prior systems were designed to only fetch globally set k messages (most often $k = 1$). If a user receives less than k messages, their true number of messages will be padded to k to avoid leaking this volume and exposing them to traffic analysis attacks. If a user received more than k messages these would be lost. In deferred retrieval, rather than dropping messages if a user receives more than k messages between fetches, these messages are just deferred to a subsequent fetch following a first in, first out convention—hence the name deferred retrieval. In order to correctly deliver these messages the system must maintain some internal state between deliveries, that is the system is inherently asynchronous.

The second difference is a result of our new traffic leakage, (S_t, R). Prior systems set the fetch rate, determined by k, globally for all users; however, in deferred retrieval, k_i is set per user. In order to meet the leakage (S_t, R) there are two assumptions on user behavior. (1) These k_i must not vary in response to the actual volume of traffic waiting for delivery in the system but must be set based on an estimate of users' *overall* traffic rates. While an exact estimate is impossible without knowledge of future traffic, users often can estimate the order of magnitude of messages they receive in some interval. For example, a user may not know they receive exactly 42 messages per day, but they may know that they typically receive less than 100. (2) Related to the first, users may submit fetch requests on their own schedule so long as the timing of fetch requests does not leak any more information than (S_t, R).

This flexibility allows deferred retrieval to use strictly less overhead than globally set bandwidth restrictions for the same latency. Suppose k_i is the largest number of messages each user receives in some time-frame, L. This implies that if each user fetches k_i messages per L time, no message will be deferred. If we set a global fetch rate such that no message will be deferred, k must be at least $\max_i(k_i)$. The overhead for the system setting k_i per user is less than the overhead for the system setting a global fetch rate, because

$$\sum_i k_i \leq n \max_i(k_i). \quad (2)$$

onymoy This relaxations allows for many attractive usability properties [11], but all reduce to allowing users to vary their download rates. While existing work sets download rates globally, deferred retrieval sets them per user. We denote this bandwidth restriction by k^*_{out} to signify a per user variable number of messages exiting the system. This gives us the flexibility to change rates, so long as these changes do not depend on users' actual received traffic. For example, users on mobile networks can reduce their rate without leaking information about their communication patterns. Users can go offline without becoming vulnerable to traffic analysis—this is not possible in synchronous systems that require users to participate in every round. If they are willing to tolerate additional latency while they are asleep, for example, they can reduce their rates. Conversely, if they have been offline and wish to catch up on their messages they can issue a larger fetch request without compromising security.

Unfortunately, this functionality requires additional primitives over those implemented in existing work. Specifically, systems must enable asynchronicity—messages may be deferred to subsequent fetches, so state must persist between these operations. Relatedly, these extra messages should not be lost, implying variably sized mailboxes. Finally, systems must support efficient fetching and padding to variable numbers of messages. Existing systems do not support this functionality.

4. System Designs

In §3 we designed a model for a system with traffic leakages that are not vulnerable to traffic analysis. In this section we design secure implementations of deferred retrieval, *i.e.* when the system processes requests, it should leak no more than the traffic at users. We implement three separate systems for different use cases. The first, Sparta-LL, is based on oblivious data structures [57] and is optimized for low latency in cases where the number of users is smaller than the message database. The second, Sparta-SB is optimized for high-throughput where the number of users is close to the size of the message database. Finally, Sparta-D, combines ideas from both and achieves high throughput with large message stores due to its distributability.

Sparta systems implement deferred retrieval and thus are all asynchronous, meaning they have internal state that

messages are sent into and fetched from, and support variable k_i fetch sizes. Existing work suffers from scalability problems even without these additional primitives, so we instead turn to hardware enclaves running oblivious algorithms to implement deferred retrieval. Beyond offering strong security, hardware enclaves have the advantage that they can be deployed within single organizations in contrast with systems that rely on distributed trust. This deployment problem has not been resolved [85], and the lack of deployed globally secure systems in the anytrust model casts doubt on its real-world feasibility. In contrast, Signal has successfully deployed hardware enclaves running oblivious algorithms to support oblivious contact discovery for millions of users.

Oblivious Multiqueues (OMQs). Despite the different goals of these systems, the intuition behind how each Sparta system offers security is the same. Each system supports the primitives required to efficiently implement deferred retrieval, *i.e.*, asynchronicity and variably sized mailboxes and fetches. They do this by implementing a data structure we call an *oblivious multiqueue (OMQ)*. OMQs are simply sets of queues, with each queue representing a user's mailbox. OMQs support three functions: $\mathsf{setup}(\lambda, N)$, $\mathsf{push}(i, m)$, and $\mathsf{pop}(i, k)$. The setup function takes a security parameter and a total capacity for the queues. A push operation takes a mailbox identifier, i, an element, m, and inserts m at the tail of queue i. A pop operation takes a mailbox, i, and a value, k, and returns the first k elements from queue i. A multiqueue is oblivious if the execution of push and pop does not depend on the *values* of the inputs, but only the *lengths*, *i.e.*, the execution of a pop for one mailbox should be indistinguishable from a pop for a different mailbox, though the execution can depend on k, the number of elements to return. This guarantees that when executed within a hardware enclave that hides the values of the operands, incoming and outgoing messages cannot be linked based on how the system operates. This functionality can be used to implement deferred retrieval, where a send corresponds to a push and a pop corresponds to a fetch.

4.1. Sparta — Low Latency

The goal of Sparta-LL is to implement a low-latency data structure for storing and retrieving messages. Sparta-LL is composed of two components: a user store and a message store. The user store is implemented as an oblivious map [43] and relates user identifiers to the head and tail of items in the message store. The message store is implemented as a non-recursive ORAM (*i.e.*, PathORAM [51]) and stores queue nodes, with each node storing a message and a pointer to the next node in the ORAM, inspired by Wang *et al.* [57]. On a send, the location of the tail of the recipient is looked up in the user store, and the message is written at that location in the message store. On a fetch, the user looks up the head of their queue in the message store and follows the pointer in each message node to the next message node.

Detailed Description. In Fig. 3 we give the pseudocode for these routines. In a send, we take in a recipient and a message and precompute the address for the next send request so that this can be stored in the message node. We then make a request to the user store to get the position for the current message and update it with the new tail value. Finally, we write back the message node with pointer to the next message to the message store.

In a fetch we take in the recipient and the volume of messages k to read. We first look up the pointer to the head of the queue from the user store, then iterate k times, making an oblivious request to the message store in each iteration, getting the message and the pointer to the next node in the queue. As long as we have not reached the end of the message queue as denoted by last, we continue making real accesses and otherwise make dummy requests to the messages store to avoid leaking the true number of messages the user has in the message store.

Security. In send, the security of the scheme reduces to the primitives we use and the random accesses we make to the message store. Because the user store is oblivious, accesses to it do not reveal anything about the recipient. Similarly, because we write back the node to the message store as a random location only used once, subsequent reads of this block will be independent. In fetch, we first fetch the head of the queue from the user store, then iterate through the first k messages. Note that leaking k is permissible so long as k does not depend on the true volume of messages a user receives. Within the loop, we give an if statement for simplicity but implement this using oblivious select. Thus this conditional leaks no information, because in both cases we make accesses to the message store using fresh randomness from the user store. Because the message store is implemented as an oblivious RAM, these accesses leak no information about the index of the record accessed. Finally, updating the returned messages M occurs independently of the branch taken, leaking no additional information.

Efficiency. In this scheme the cost of accessing the user store is $O(\log^2 N)$, where N is the number of users. This is due to the underlying oblivious map structure. The message store is a non-recursive PathORAM, so the cost of an access is $O(\log M)$ [57] where M is the size of the message store. Because we can assume that the number of users is much less than the number of messages sent, this scheme is more efficient than naive use of a sorted multimap. The total cost of a send then is $O(\log^2 N + \log M)$, while the cost of a fetch is $O(\log^2 N + k \log M)$.

4.2. Sparta — Sort-Based

The above scheme is asymptotically efficient, but it requires that requests all be processed sequentially and thus can be expected to have relatively low-throughput in practice (see Fig. 8). Sparta-SB (sort-based) implements the same oblivious multiqueue functionality but trades asymptotic complexity to achieve much higher throughput by processing batches of requests together. At a high level, this system works by using bitonic sort [58] to group messages by sender and time and then mark the appropriate messages for

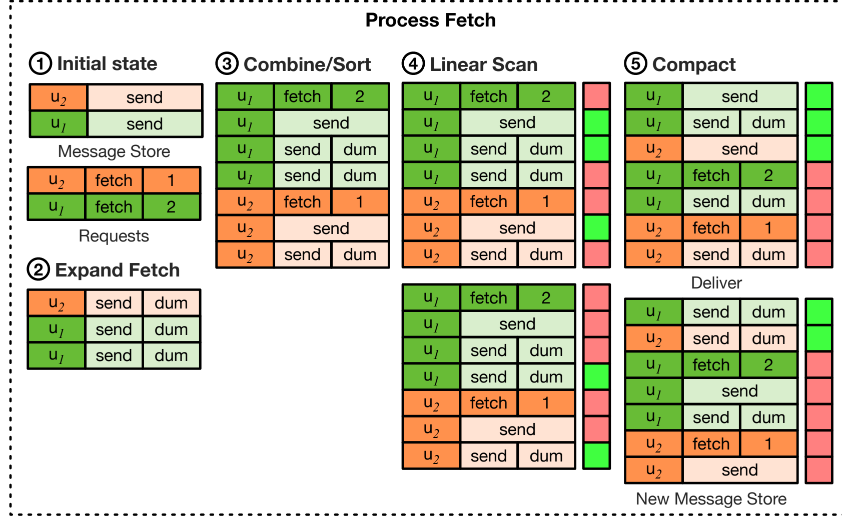


Figure 4. The fetch operation in Sparta-SB. This operation can be performed in an oblivious sort, a linear scan, and two oblivious compactions. Because the volumes of the requests, k_i, are public we simply take the first $\sum k_i$ from the compacted messages for delivery. Because the volume of incoming messages is public we take the first m messages from the new compacted message store as the next state of the message store.

ProcessBatch$(S, R; MS)$

1: Let S be a set of send requests, (r, m). Let R be a set of fetch requests, (r, k), where k is the number of messages.
2: For each (r, k) insert k tuples $(r, dummy)$ into D.
3: Prepend M with R, then append S followed by D.
4: Using an order-preserving oblivious sort, sort M by receiver then request type (fetch $<$ send $<$ dummy).
5: Do a linear scan over M. On each new receivers fetch request, maintain a sum of the k. Then on the first non-fetch, begin marking them until k elements have been marked.
6: Do an order preserving oblivious compact on the marked messages then take the top $\sum k_i$ as the messages to deliver.
7: Oblivious compact on the unmarked, non-fetch tuples, and take these as the new state of the message store.

Figure 5. The ProcessBatch routine of the Sparta Sort-Based solution. At a high-level it sorts message requests by users such that fetches appear first. It then marks the first k elements and filters these out using two compactions. For a visual representation see Fig. 4.

delivery via a linear scan. Afterward we use two oblivious compactions [46] to filter out messages that have been marked for delivery and messages that should be deffered for subsequent fetches (see Figs. 4 and 5).

Detailed Description. The above description leaves out several important details. Suppose a user issues a fetch for more messages than they currently have stored in the database. We must take care to not mark messages meant for other users. Because users could have zero messages, for fetches of size k we must insert k dummy messages. We then sort the message store so that fetches precede sent messages precede dummy messages. This ensures that real messages are fetched before dummy messages. As we scan the database, on each new user identifier we obliviously update the count of the remaining messages to fetch to k, and then mark the first k messages. These conditionals are implemented using oblivious selection. After we complete the scan, messages that ought to be delivered have been marked, and the messages that should be deferred to the new state of the store are not. Notice that because each k_i is assumed to be public, we can compact and take the first $\sum k_i$ messages to deliver. Similarly, because we add dummy messages, the size of the new dummy store will be exactly the size of the old store plus the size of the new send requests.

Security. The security of this scheme reduces to the obliviousness of the primitives and the fact that the number of sends and fetches are public. Because we first obliviously sort and pad with k dummies, we are guaranteed that each user has at least k elements, thus one user's fetch will never mark a message not addressed to them. The linear scan similarly operates over the size of the messages store, which is the sum of the prior sends and fetches. Within the loop, all logic is implemented using oblivious selects, thus our conditionals leak nothing about the contents of requests. Finally, because compaction is oblivious and the size of marked messages will be exactly the sum of k_i, this is oblivious and correct.

Efficiency. This solution is implemented in a single oblivious sort, one linear scan and two oblivious compacts. The asymptotic costs of bitonic sort [58] and our compact rou-

tine [46] are $O(M \log^2 M)$ and $O(M \log M)$ respectively, thus the overall cost of the sort dominates, *i.e.* the cost is $O(M \log^2 M)$ where M is the size of the message store.

4.3. Sparta — Distributed

The above sort-based system achieves high throughput as we see in §5.2, but with each batch we operate over the entire size of the message store, limiting scalability. In Sparta-D we describe a method for distributing this message store into many smaller parts, while maintaining the exact queueing semantics as in the prior systems. Sparta-D has two components: a queue maintainer that maintains per user metadata for queueing and a number of submaps used to store messages. The queue maintainer is similar to the user store in Sparta-LL, but is implemented using sort and compaction to improve throughput. The queue maintainer takes incoming requests and translates them into unique object identifiers. It then obliviously batches and sends requests for these objects to the submaps. The submaps are themselves oblivious, high-throughput maps. In Figs. 6 and 7 we show the operation and the psuedocode of the queue maintainer.

Detailed Description. In Sparta-D the queue maintainer translates requests to unique object identifiers. It does this by maintaining a per user count of the the number of messages read and sent. On send and fetch requests, requests are sorted together so that counters from the user store can be propagated and incremented in the requests during a linear scan of the user store. Sparta-D then takes a hash of the user's identifier and these counters to create a unique object ID for each object, such that the IDs of sends are fresh and the IDs of fetches correspond to the IDs of previous sends. These IDs are unique and randomly distributed, so using Theorem 3 from Snoopy [86] we can obliviously construct equal sub-batches of requests using only the number of submaps and the number of requests, both of which are public. This theorem gives cryptographic upper bounds on the number of unique and randomly distributed IDs that are mapped to individual submaps, thus by padding the size of the subbatches to this upper-bound they reveal no information about their composition. The submaps then return the values associated with the object ID, and Sparta-D computes an oblivious shuffle over the returned values to hide which value came from which submap.

Security. The security of Sparta-D reduces to the oblivious primitives. The sort is oblivious, as is the linear scan to propagate request counters. The linear scan is over a publicly sized data structure. The conditionals within the loop are implemented obliviously, and the compaction to remove requests for the submaps is also oblivious. Because at the end of the construction of the indices all are unique and randomly distributed, we can apply Theorem 3 from Snoopy [86] to obliviously construct the sub-batches. These sub-batches are sent to the submaps, which are implemented as oblivious maps. The final shuffle ensures that delivering the results of the fetches reveals no information about the composition of the sub-batches.

Efficiency. Sparta-D is more efficient than Sparta-SB when the size of the message store is much greater than the number of users. In this case the extra costs of maintaining the queues are outweighed by the fact that the message store is sharded into roughly M/S chunks, where M and S are the size of the message store and the number of submaps. When these submaps are highly distributed, the cost of Sparta-D is dominated by sorting at the queue maintainer. This sort is over the number of users, N, and thus the complexity is $O(N \log^2 N)$.

5. Experimental Evaluation

We demonstrate the efficiency of deferred retrieval in §5.1 and our Sparta systems in §5.2. These experiments[1] give a complete study of the latency (see Eq. 1) and overhead of Sparta under the assumptions of deferred retrieval. In particular, §5.1 quantifies l_u, l_s (the latency due to assumptions for traffic analysis resistance), and overhead under real email workloads, while §5.2 quantifies l_p (the latency due to system processing) and throughput.

Experimental Setup & Implementation. In our experiments we use four datasets. Enron [33] and Seattle [35] (see Appendix A) contain real email metadata, which we use to evaluate the costs of traffic analysis resistance for our system model. Using real user data is critical for evaluating system assumptions and models, because l_u and l_s depend on both the actual sending and receiving rates as well as the assumptions made to prevent traffic analysis.

For our system evaluations we use two synthetic datasets: the Storage workload and the Legacy workload. In the Storage workload we fix the number of users and fetches to 2^{13} and set the size of the message store between 2^{18} and 2^{23}. We cap the sizes of message stores at 2^{23} because of limitations in the scalability of sort. Relatedly, we set the users to 2^{13} to capture situations where users have many messages stored at the server. This would occur if users go offline and/or have many concurrent conversations. The Legacy workload is taken from prior work [5], [13], [14], [21] and assumes a one-to-one correspondence between users, the size of the message store, and the number of fetches issued. In all of these datasets we set the block size to 128B, which, in our Sparta systems, give a message payload of 100B. The remaining bytes in each block are used by the system to process messages.

We run our systems experiments on Microsoft Azure, which provides support for Intel SGXv2 in the `DC48sv3` class of VMs. These VMs have forty-eight cores and 384GB memory. We implement our systems in Rust with $\sim$1600 lines of code using the Fortanix Enclave Development Platform (EDP). We also provide parallel Rust implementations ofoblivious primitives, such as bitonic sort [58] and OR-Compact [46], using constant-time CMOV-based oblivious

1. Code for these experiments can be found at
https://github.com/ucsc-anonymity/sparta-model-evaluation and
https://github.com/ucsc-anonymity/sparta-experiments

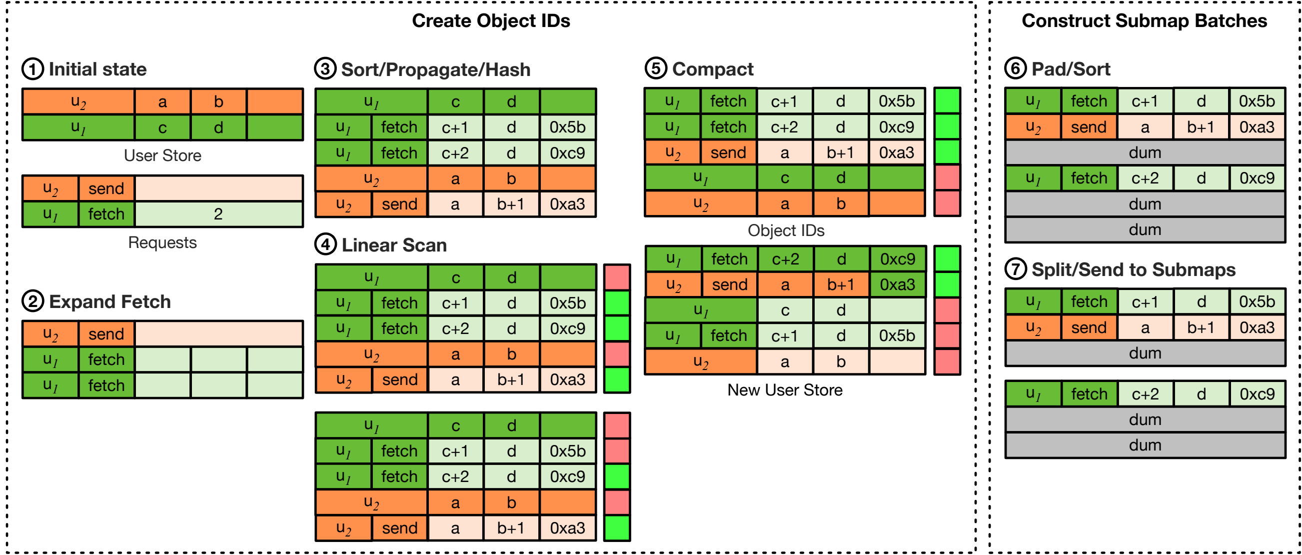


Figure 6. The queue maintainer in Sparta-D. The queue maintainer maintains pointers to the head and tail of each users queue. On a request it constructs random object IDs, then obliviously pads batches of requests and sends them to the appropriate submaps. It then waits for responses from the submaps, shuffles the responses and delivers them to the recipients.

CreateObjectIds$(S, R; US)$

1: Let S be a set of send requests, (r, send, m). Let R be a set of fetch requests, (r, fetch, k), where k is the number of messages to fetch.
2: For each (r, k) expand them to k tuples $(r, \mathsf{fetch}, dummy)$ as R.
3: Take S and R together, and expand them to be of the form $(r, op, b, first, last, key, m)$.
4: Take US and expand it to be of the form $(r, \mathsf{ms}, b, first, last, key, m)$
5: Let M be the R, MS, and S sorted together by (r, op), where $\mathsf{ms} < \mathsf{fetch} < \mathsf{send}$.
6: Let $w_i = 0$, $r_i = 0, k_i = 0, next = M[1].r$
7: **for** $(r, op, first, last, key, m) \in M$ **do**
8: Set $w_i = first$ if $op = \mathsf{ms}$, $w_i + 1$ if $op = \mathsf{send}$, w_i if $op = \mathsf{fetch}$.
9: Set $r_i = \max(last, w_i)$ if $op = \mathsf{ms}$, $r_i + 1$ if $op = \mathsf{fetch}$, r_i if $op = \mathsf{send}$.
10: Set $k_i = key$ if $op = \mathsf{ms}$, $H(r, w_i)$ if $op = \mathsf{send}$, $H(r, r_i)$ if $op = \mathsf{fetch}$.
11: Set $b = 0$ if $next = r$ else 1.
12: Set $next = M[i+1].r$.
13: **end for**
14: Obliviously compact M on b, then take the top n records as US.
15: Obliviously compact M on $op \neq \mathsf{ms}$, then take the top $|S| + |R|$ as the requests.

Figure 7. Creating object IDs is one routine that Sparta-D runs at the load balancer. Object IDs are unique and uniform due to hashing, so we can apply Theorem 3 from Snoopy [86] to obliviously construct batches of requests for the submaps.

swaps as in Sasy *et al.* [46]. We run each experiment ten times, reporting the mean of the measurements.

5.1. Model Evaluation

To quantify the costs of assumptions and bandwidth restrictions to achieve traffic analysis resistance we focus on two metrics: latency, the time between when a message is ready to send and when it is downloaded, and overhead, the amount of network traffic a user is required to send/receive.

Baselines. In the introduction we taxonomize existing work by the bandwidth restrictions they impose. There are two classifications that existing work falls into: (1) global input restrictions (denoted by k_{in}), which contain synchronous systems (*e.g.*, mixnets, DC-net-like systems) [4], [6], [8], [12], [16] and differentially private systems [9], [10], [11], [25], and global output restrictions (denoted by k_{out}), which contain existing asynchronous systems [13], [21], [22]. Input restrictions are typically imposed by assumptions made about user behavior (*i.e.* the one message assumption) and are unimplementable, while output restrictions are typically imposed by system design, *e.g.*, in Pung [22], where users can download a globally fixed k messages per round.

Measuring Latency and Overhead. By considering ideal implementations of systems with zero processing latency

Dataset	Latency (s)	Throughput (fetches/s)
Storage (2^{20})	0.00067	1500
Legacy (2^{20})	0.0011	910

Figure 8. The latency and throughput of Sparta-LL.

$(l_p = 0)$, we can compute latencies and overheads solely due to bandwidth restrictions for traffic analysis resistance. This reduces all systems with a particular bandwidth restriction to a single ideal representation, specified only by their bandwidth limitations. This approach favors prior systems, which incur higher processing costs than Sparta due to their use of more expensive primitives like PIR, FHE, and MPC (i.e., l_p is significantly lower in Sparta).

Given a send/fetch schedule, a number of messages to send/fetch, and the sender, receiver, and timing of messages, we can calculate the time messages enter and exit this ideal system, as well as the amount of traffic per interval. As an example, if we specify that all users must send exactly one message per five second interval, if one user wants to send two messages during that interval, the second will be deferred to the subsequent interval increasing latency. If a user has no messages to send in a particular interval, a dummy message will be filled in, increasing overhead.

Given a maximum latency L, for deferred retrieval (denoted by k^*_{out}), the optimal setting of the traffic rate to meet latency L is k_i, where k_i is the maximum number of messages received in an L length interval for each user. For systems that set global bandwidth limitations (*i.e.*, k_{in} and k_{out}), the optimal setting to guarantee this latency is $k = \max_i(k_i)$. For example, in an ideal system if we know that in one minute, we will not receive more that one hundred messages we can set the download rate to one hundred and never have messages delivered more than one minute after they are sent.

In practice, these k_i are unknown. However, we may have *some* information about k_i. Users may not know their exact max traffic rate, but they may be able to upper-bound the order of magnitude, *i.e.* $k_i < 10^x$ for some x. For example, most users could be confident that they will not receive more than one thousand messages in a minute. By increasing the base of this exponent the estimate of the max rate becomes progressively poorer—using fewer bits of information from the true k_i—making it more realistic that a user will be able to estimate x (an estimation factor equal to one denotes perfect knowledge of the max rate; increasing the estimation factor indicates increased uncertainty).

In order to give the advantage to existing work, we compare these poor estimates of k_i in our work to optimally set k in existing work. We vary this estimation base between 2 and 512. We chose this range as a superset of the range of reasonable user estimations. We set the maximum target latency to be one minute to capture text messaging scenarios where users expect to receive their messages quickly.

Results. We report our results in Fig. 9. The x-axis of these figures represents the estimation factor for our work only,

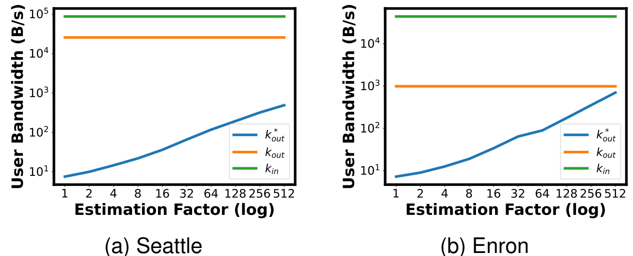
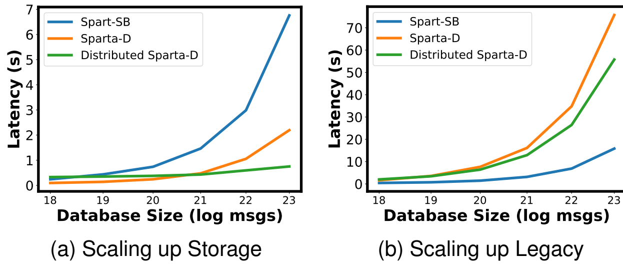


Figure 9. The average bandwidth per user to support sub-minute latency. As user's estimates get worse our system's performance will eventually converge to (but never be worse than) the performance of prior models. In practice, estimates within a factor of 10 are sufficient to reduce network overhead by multiple orders of magnitude.

so as we increase the estimation factor we get worse estimations and worse overhead. Comparing optimal estimations of k for prior work (k_{out}) against optimal estimations of the k_i in our work (k^*_{out}, $x = 1$), in Seattle we observe that our model results in a $3400\times$ reduction in required traffic to support traffic analysis resistance. In absolute terms this means that to support sending short 128B messages with at most 60s of latency and achieving traffic analysis resistance, the average user in our systems will download just ~6B/s, meaning the average user receives at most about one message every 20 seconds. For comparison, streaming music downloads 3KB/s worth of traffic. This is a practical amount of traffic. Prior systems would have required they receive and send at about 26 and 87 KB/s respectively. The gap between k^*_{out} and k_{out} is smaller in Enron because the spread because the average user's traffic rate and the max user's traffic rate is smaller than in Seattle (see Appendix A). Comparing optimal settings of k_i both in our work and prior work shows that our approach results in a $140\times$ reduction in overhead in Enron.

While optimally setting k_i is unrealistic (both for existing work and ours), users of our systems only need coarse estimates of k_i to see significant improvements in overhead compared to prior work. Estimations of the order of magnitude of the true value give us significant improvements over prior work in both datasets. When users can estimate their optimal rate to the nearest power of sixteen, *e.g.*, our work gives a $710\times$ and $29\times$ improvement over optimally set rates in existing work for Seattle and Enron, respectively. Such estimations may be practical for users in the real world.

With worse estimations (x increasing), as expected we see that in both datasets the amount of traffic per user increases and the graphs converge. There is also a significant gap between k_{out} and k_{in}. The reason for this is that sender traffic is more bursty than receiver traffic. This is due to the presence of emails with many recipients in our datasets. We split these into many emails (see Appendix A), following prior work [34]. This results in large bursts of sent messages and consequently increases max traffic rates for senders. We note that as the target latency increases, the gaps between our model and previous models widen. This is because the number of messages sent/delivered per round grows faster



(a) Scaling up Storage (b) Scaling up Legacy

Figure 10. Results of Experiment #1. We observe that Sparta-D gains a significant performance advantage compared with Sparta-SB as the size of the storage workload increases. This is more pronounced in distributed Sparta-D, which has 15 submaps. On the Legacy dataset, which Sparta-SB is optimized for, we observe it gains a large advantage even over distributed Sparta-D.

over all users than it does for individual users as the length of the round grows.

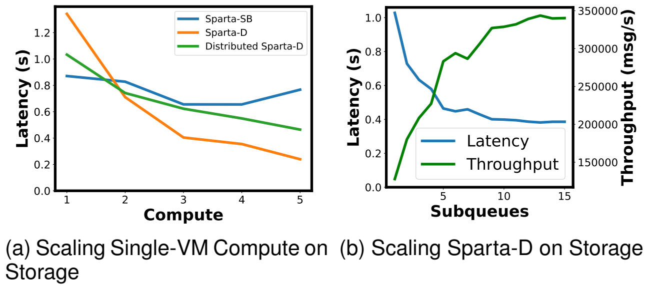
5.2. System Evaluation

Our Sparta systems are instantiations of deferred retrieval, providing practical traffic analysis resistance. We demonstrate how Sparta systems scale using the Storage and Legacy workloads as we increase their sizes (Experiment #1) and as we use more compute resources (Experiment #2).

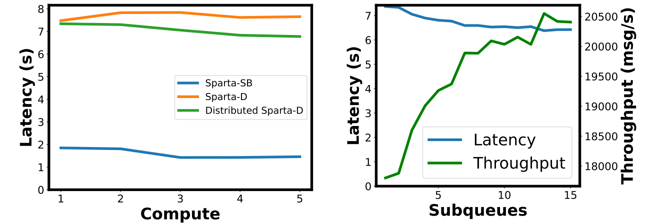
Experiment #1: Scaling the Message Database. In this set of experiments, we vary the number of messages from 2^{18} to 2^{23}. We run two instances of Sparta-D: the first runs a version with the queue maintainer and five submaps running in parallel, each with eight threads; the second runs a version with the queue maintainer and fifteen submaps with forty-eight threads. For Sparta-SB, we allocate a total of forty-eight threads, equal to the resources in the non-distributed Sparta-D. We report our results in Fig 10. We additionally, run Sparta-LL with both workloads for sizes 2^{20} and report results in Fig. 8.

As we expect, Sparta-D enables greater scaling of the message database than Sparta-SB on the Storage workload (see Fig. 10a). In the single machine case Sparta-D and Sparta-SB diverge as the size of the message store increases with Sparta-D performing the same workload $3.2\times$ faster than Sparta-SB. In the case of the distributed Sparta-D as we add resources in the experiment this trend is even more pronounced. A Sparta-D instance with fifteen submaps performs about $15\times$ faster than Sparta-SB and about $5\times$ faster than the single machine Sparta-D instance. In these experiments, especially on Storage, we observe that Sparta-D and distributed Sparta-D initially are slower than Sparta-SB due to the additional work done in the queue maintainer. However, as the size of the database increases, its ability to distribute contributes to better performance.

Sparta-D is not optimized for the Legacy dataset, while Sparta-SB is. As we would expect, for the Legacy dataset, Sparta-SB significantly outperforms even the distributed



(a) Scaling Single-VM Compute on Storage (b) Scaling Sparta-D on Storage



(c) Scaling Single-VM Compute on Legacy (d) Scaling Sparta-D on Legacy

Figure 11. Results of Experiment #2 results. In Figs. 11a and 11d we test our systems on the Storage workload. We observe significant reductions in latency and increases in throughput as we add compute resources to Sparta-D. In Figs. 11c and 11d we observe must smaller benefits in adding resources to Sparta-D when running the Legacy workload. This is expected since Sparta-D is not optimized for this workload. On the other hand Sparta-SB is, and performs very well, processing 2^{20} message in about 1.5 seconds.

Sparta-D (see Fig. 10b). The reason for this, is that in Sparta-D the queue manager is not distributed, so for large numbers of users the work done at the queue maintainer becomes the dominating factor. We note that on a single 48-core machine, Sparta-SB increases throughput by $15\times$ in a database of size 2^{23} compared with a 150-machine Groove cluster [11] operating on a database of size 2M.

Sparta-LL predictably has low latency and throughput, and thus is not suited for use as a general messaging system.

Experiment #2: Scaling Compute. We investigate how Sparta-SB and Sparta-D scale with additional compute resources on Storage and Legacy with fixed size of 2^{20}. In the first experiments, we first measure the effect of adding smaller amounts of resources for Sparta-SB, single-machine Sparta-D, and distributed Sparta-D. For single-machine Sparta-D we vary the number of submaps from one to five with eight threads allocated to each and eight threads allocated to the queue maintainer. To keep parity with single-machine Sparta-D, for Sparta-SB we vary the number of threads from sixteen to forty-eight in increments of eight. For distributed Sparta-D, we allocate additional submaps. In the second experiments, we investigate the scalability of larger instances of distributed Sparta-D on Storage and Legacy, also with size 2^{20}. We vary the number of submaps from one to fifteen. We measure the time to fetch one message per user and report our results in Fig. 11.

We see the results of the first experiments in Figs. 11a and 11c. For the Storage workload, Sparta-D benefits

the most from additional resources in the single machine case, outpacing Sparta-SB by a factor of two. The reason for this is that the workload is relatively small and thus parallelizing the submaps significantly reduces the cost of those operations. In the distributed case, the advantage of distributing this workload is not as apparent because of network latency (see Fig. 10a). In the sort-based version of Sparta compute is not the limiting factor in either workload.

In the second set of experiments, we see a significant improvement to the latency and throughput of distributed Sparta when more machines are added to the storage workload (see Fig. 11b). Past ten submaps, the returns diminish as the queue manager becomes the bottleneck. We see a similar but much diminished trend when running distributed Sparta on the legacy workload (see Fig. 11d). The reason for this is that the bottleneck is the queue manager from the beginning, thus we see only small gains as we distribute the cost among more submaps.

6. Discussion

Our system model—deferred retrieval—reduces the cost of long-term traffic analysis by orders of magnitude with realistic user assumptions. With our Sparta systems we demonstrate that deferred retrieval can be efficiently implemented. This model was enabled by insights derived from our leakage framework, namely that assumptions for traffic analysis resistance can be relaxed and that global bandwidth restrictions are not required for traffic analysis resistance. Instead, traffic at recipients may vary so long as these variations are not dependent on the timing of the input to the system, thereby preserving independence of the input and output of systems.

Though we implement our Sparta systems using Intel SGX and oblivious algorithms, deferred retrieval is not tied to this trust model. One direction of future research is to support the functionality of deferred retrieval in conventional implementation models, *e.g.*, MPC/anytrust, FHE. However, an advantage of our approach is that (1) systems implemented using Intel SGX and oblivious algorithms are performant enough to be deployed in the real world, as demonstrated by Signal, and (2) they can be securely deployed without requiring coordination between several different trust domains. Anytrust systems require non-colluding parties to securely deploy them, which is a significant challenge in practice [85] as evidenced by the lack of deployed systems following this implementation strategy.

As discussed in §5.1, deferred retrieval requires users' download rates to be carefully set to minimize latency and overhead due to queueing at the messaging service. Theoretically, optimally setting rates in deferred retrieval will always lead to better performance than optimally set global bandwidth rates. However, the degree to which deferred retrieval will outperform existing work depends on features of the workloads. This is clear from the results of §5.1, where the gap between our work and prior work is much greater in the Seattle dataset than in the Enron dataset. The features of datasets that lead to better performance in our work are directly related to Eq. 2. From this, we can reason that the greater the difference between the max and average user's rates, the better deferred retrieval will perform compared to global traffic rates. These are exactly the features we observe in Seattle and to a lesser degree in Enron (see Appendix A). To keep the required traffic rates low, workloads should also not be very bursty. Enron is a very sparse dataset when compared with Seattle, thus it is unsurprising that deferred retrieval would perform better in Seattle. We believe that Seattle's denser workload is more representative of messaging applications. Email data is not a perfect analog for the characteristics of messaging systems, unfortunately suitable datasets are scarce. Because of this, another direction for future research is to collect such metadata or construct representative synthetic data.

Setting traffic rates in traffic analysis resistant systems is a fundamental problem because these rates cannot vary based on actual traffic. Though deferred retrieval dramatically reduces the overhead and is feasible for low-bandwidth messaging, high-bandwidth messaging (*i.e.*, if users exchange video) will likely be too expensive. One approach to dealing with this would be to allow users to update their traffic rates at wide intervals (*e.g.*, every week or month). This violates security and theoretically enables traffic analysis attacks over time; however, it is likely possible that allowing some dependence between the input and output will not translate into practical attacks. Making these relaxations in a principled way is an interesting direction for future work.

7. Conclusion

We introduce the first long-term traffic analysis resistant metadata-private communication system with practical performance and realistic assumptions. The system is trivially deployable and can support sub-minute latencies with less network overhead than streaming music. Simultaneously, we introduce the first system-independent framework for quantifying traffic leakage. We designed a new model for how systems can operate that allows them to achieve traffic analysis resistance under reasonable assumptions with orders of magnitude less overhead for the same latency. We build three versions of this functionality on Intel SGX [36]. Our experiments demonstrate that Sparta-SB and Sparta-D are significantly more scalable than existing works.

Acknowledgments

This work was supported by the National Science Foundation under Grant No. CNS-2106259 and Grant No. CNS-1814347, the Kumar Malavalli Endowment at UC Santa Cruz, and the industrial sponsors of the Center for Research in Storage Systems (CRSS) at UC Santa Cruz. We thank the faculty and students in the CRSS for their comments, and especially Apostolos Mavrogiannakis for his invaluable assistance in helping us prepare this paper. We also thank our anonymous reviewers for their valuable feedback and guidance which greatly improved the quality of this paper.

References

[1] R. Dingledine, N. Mathewson, P. F. Syverson *et al.*, “Tor: The Second-Generation Onion Router,” in *Proceedings of the USENIX Security Symposium*, 2004.

[2] T. Wang and I. Goldberg, “Improved Website Fingerprinting on Tor,” in *Proceedings of the ACM Workshop on Privacy in the Electronic Society*, 2013.

[3] I. Karunanayake, N. Ahmed, R. Malaney, R. Islam, and S. K. Jha, “De-anonymisation attacks on Tor: A Survey,” *IEEE Communications Surveys & Tutorials*, 2021.

[4] D. L. Chaum, “Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms,” *Communications of the ACM*, 1981.

[5] 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*, 2015.

[6] I. Abraham, B. Pinkas, and A. Yanai, “Blinder: Scalable, Robust Anonymous Committed Broadcast,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2020.

[7] A. Kwon, D. Lu, and S. Devadas, “XRD: Scalable Messaging System with Cryptographic Privacy,” in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2020.

[8] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford, “Atom: Horizontally Scaling Strong Anonymity,” in *Proceedings of the ACM Symposium on Operating Systems Principles*, 2017.

[9] 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*, 2015.

[10] D. Lazar, Y. Gilad, and N. Zeldovich, “Yodel: Strong Metadata Security for Voice Calls,” in *Proceedings of the ACM Symposium on Operating Systems Principles*, 2019.

[11] L. Barman, M. Kol, D. Lazar, Y. Gilad, and N. Zeldovich, “Groove: Flexible metadata-private messaging,” in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2022.

[12] D. Lu and A. Kate, “RPM: Robust Anonymity at Scale,” in *Proceedings of the Workshop on Privacy Enhancing Technologies*, 2023.

[13] S. Eskandarian, H. Corrigan-Gibbs, M. Zaharia, and D. Boneh, “Express: Lowering the Cost of Metadata-hiding Communication with Cryptographic Privacy,” in *Proceedings of the USENIX Security Symposium*, 2021.

[14] S. Eskandarian and D. Boneh, “Clarion: Anonymous Communication from Multiparty Shuffling Protocols,” in *Proceeding of the Network and Distributed Systems Security Symposium*, 2022.

[15] N. Alexopoulos, A. Kiayias, R. Talviste, and T. Zacharias, “MCMix: Anonymous Messaging via Secure Multiparty Computation,” in *Proceedings of the USENIX Security Symposium*, 2017.

[16] D. Chaum, D. Das, F. Javani, A. Kate, A. Krasnova, J. D. Ruiter, and A. T. Sherman, “cMix: Mixing with Minimal Real-Time Asymmetric Cryptographic Operations,” in *Proceedings of the Applied Cryptography and Network Security Conference*, 2017.

[17] R. Cheng, W. Scott, E. Masserova, I. Zhang, V. Goyal, T. Anderson, A. Krishnamurthy, and B. Parno, “Talek: Private Group Messaging with Hidden Access Patterns,” in *Proceedings of the Annual Computer Security Applications Conference*, 2020.

[18] D. Kesdogan, D. Agrawal, and S. Penz, “Limits of Anonymity in Open Environments,” in *Proceedings of the Workshop on Information Hiding*, 2002.

[19] G. Danezis, “The Traffic Analysis of Continuous-time Mixes,” in *Proceedings of the Workshop on Privacy Enhancing Technologies*, 2004.

[20] S. Oya, C. Troncoso, and F. Pérez-González, “Meet the Family of Statistical Disclosure Attacks,” in *Proceedings of the IEEE Global Conference on Signal and Information Processing*, 2013.

[21] A. Vadapalli, K. Storrier, and R. Henry, “Sabre: Sender-Anonymous Messaging with Fast Audits,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 2022.

[22] S. Angel and S. Setty, “Unobservable Communication over Fully Untrusted Infrastructure,” in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[23] 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 Applied Cryptography and Network Security Conference*, 2004.

[24] D. Lazar and N. Zeldovich, “Alpenhorn: Bootstrapping Secure Communication without Leaking Metadata,” in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[25] D. Lazar, Y. Gilad, and N. Zeldovich, “Karaoke: Distributed Private Messaging Immune to Passive Traffic Analysis,” in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2018.

[26] H. Corrigan-Gibbs and B. Ford, “Dissent: Accountable Anonymous Group Messaging,” in *Proceedings of the ACM Conference on Computer and Communications Security*, 2010.

[27] D. Kesdogan and L. Pimenidis, “The Hitting Set Attack on Anonymity Protocols,” in *Proceedings of the Workshop on Information Hiding*, 2004.

[28] 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*, 2006.

[29] A. Pfitzmann and M. Köhntopp, “Anonymity, Unobservability, and Pseudonymity — A Proposal for Terminology,” in *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, 2001.

[30] J. D. Little, “A Proof for the Queuing Formula: $L = \lambda W$,” *Operations Research*, 1961.

[31] J. R. Tyler and J. C. Tang, “When Can I Expect an Email Response? A Study of Rhythms in Email Usage,” in *Proceedings of the European Conference on Computer Supported Cooperative Work*, 2003.

[32] I. Gamzu, Z. Karnin, Y. Maarek, and D. Wajc, “You Will Get Mail! Predicting the Arrival of Future Email,” in *Proceedings of the Conference on World Wide Web*, 2015.

[33] B. Klimt and Y. Yang, “The Enron Corpus: A New Dataset for Email Classification Research,” in *Proceedings of the European Conference on Machine Learning*, 2004.

[34] S. Oya, C. Troncoso, and F. Pérez-González, “Understanding the Effects of Real-World Behavior in Statistical Disclosure Attacks,” in *Proceedings of the IEEE Workshop on Information Forensics and Security*, 2014.

[35] M. Chapman, “Seattle Email Metadata,” 2018. [Online]. Available: https://www.kaggle.com/datasets/foiachap/seattle-email-metadata

[36] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, “Using Innovative Instructions to Create Trustworthy Software Solutions,” in *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[37] ARM Limited, “ARM TrustZone Technology,” *White Paper*, 2004. [Online]. Available: https://developer.arm.com/documentation/102412/latest

[38] D. Kaplan, J. Powell, and T. Woller, “AMD Memory Encryption,” *White Paper*, 2016.

[39] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, “Intel Software Guard Extensions (Intel SGX) Support for Dynamic Memory Management Inside an Enclave,” in *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy*, 2016.

[40] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling Your Secrets without Page Faults: Stealthy Page Table-based Attacks on Enclaved Execution," in *Proceedings of the USENIX Security Symposium*, 2017.

[41] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software Grand Exposure: SGX Cache Attacks are Practical," in *Proceedings of the USENIX Workshop on Offensive Technologies*, 2017.

[42] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *Proceedings of the USENIX Security Symposium*, 2017.

[43] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An Efficient Oblivious Search Index," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2018.

[44] A. Tinoco, S. Gao, and E. Shi, "EnigMap: External-Memory Oblivious Map for Secure Enclaves," in *Proceedings of the USENIX Security Symposium*, 2023.

[45] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An Oblivious and Encrypted Distributed Analytics Platform," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2017.

[46] S. Sasy, A. Johnson, and I. Goldberg, "Fast Fully Oblivious Compaction and Shuffling," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2022.

[47] S. Sasy *et al.*, "Waks-On/Waks-Off: Fast Oblivious Offline/Online Shuffling and Sorting with Waksman Networks," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2023.

[48] J. G. Chamani, I. Demertzis, D. Papadopoulos, C. Papamanthou, and R. Jalili, "GraphOS: Towards Oblivious Graph Processing," in *Proceedings of the VLDB Endowment*, 2023.

[49] N. Ngai, I. Demertzis, J. G. Chamani, and D. Papadopoulos, "Distributed & Scalable Oblivious Sorting and Shuffling," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2024.

[50] G. Asharov, T. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi, "Bucket Oblivious Sort: An Extremely Simple Oblivious Sort," in *Proceeding of the Symposium on Simplicity in Algorithms*, 2020.

[51] E. Stefanov, M. V. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An Extremely Simple Oblivious RAM Protocol," *Journal of the ACM*, 2018.

[52] G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, and E. Shi, "OptORAMa: Optimal Oblivious RAM," in *Advances in Cryptology—EUROCRYPT*, 2020.

[53] S. Patel, G. Persiano, M. Raykova, and K. Yeo, "PanORAMa: Oblivious RAM with Logarithmic Overhead," in *Proceedings of the IEEE Annual Symposium on Foundations of Computer Science*, 2018.

[54] S. Devadas, M. v. Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs, "Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM," in *Proceedings of the Theory of Cryptography Conference*, 2016.

[55] S. Garg, P. Mohassel, and C. Papamanthou, "TWORAM: Efficient Oblivious RAM in Two Rounds with Applications to Searchable Encryption," in *Proceedings of the Annual Cryptology Conference*, 2016.

[56] X. Wang, H. Chan, and E. Shi, "Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2015.

[57] X. S. Wang, K. Nayak, C. Liu, T. H. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious Data Structures," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2014.

[58] K. E. Batcher, "Sorting Networks and Their Applications," in *Proceedings of the AFIPS Spring Joint Computer Conference*, 1968.

[59] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache Attacks on Intel SGX," in *Proceedings of the European Workshop on Systems Security*, 2017.

[60] M. Hähnel, W. Cui, and M. Peinado, "High-Resolution Side Channels for Untrusted Operating Systems," in *Proceedings of the USENIX Annual Technical Conference*, 2017.

[61] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017.

[62] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017.

[63] D. Das, S. Meiser, E. Mohammadi, and A. Kate, "Anonymity Trilemma: Strong Anonymity, Low Bandwidth Overhead, Low Latency – Choose Two," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2018.

[64] N. Gelernter and A. Herzberg, "On the Limits of Provable Anonymity," in *Proceedings of the ACM Workshop on Privacy in the Electronic Society*, 2013.

[65] A. Hevia and D. Micciancio, "An Indistinguishability-Based Characterization of Anonymous Channels," in *Proceedings on Privacy Enhancing Technologies*, 2008.

[66] R. Canetti, "Universally Composable Security: A New Paradigm for Cryptographic Protocols," in *Proceedings of the IEEE Annual Symposium on Foundations of Computer Science*, 2001.

[67] R. Canetii *et al.*, "Security and Composition of Multiparty Cryptographic Protocols," in *Cryptology*, 2000.

[68] A. C. C. Yao, "How to Generate and Exchange Secrets," in *Proceedings of the IEEE Annual Symposium on Foundations of Computer Science*, 1986.

[69] E. Stefanov, C. Papamanthou, and E. Shi, "Practical Dynamic Searchable Encryption with Small Leakage," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2013.

[70] I. Demertzis and C. Papamanthou, "Fast Searchable Encryption with Tunable Locality," in *Proceedings of the ACM Conference on Management of Data*, 2017.

[71] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre, "SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage," in *Proceedings of the USENIX Security Symposium*, 2020.

[72] I. Demertzis, D. Papadopoulos, and C. Papamanthou, "Searchable Encryption with Optimal Locality: Achieving Sublogarithmic Read Efficiency," in *Procedings of Advances in Cryptology*, 2018.

[73] I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou, "Dynamic Searchable Encryption with Small Client Storage," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2020.

[74] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis, "Practical Private Range Search Revisited," in *Proceedings of the ACM Conference on Management of Data*, 2016.

[75] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, M. Garofalakis, and C. Papamanthou, "Practical Private Range Search in Depth," in *Proceeding of the ACM Transactions on Database Systems*, 2018.

[76] I. Demertzis, R. Talapatra, and C. Papamanthou, "Efficient Searchable Encryption through Compression," in *Proceedings of the VLDB Endowment*, 2018.

[77] J. G. Chamani, D. Papadopoulos, M. Karbasforushan, and I. Demertzis, "Dynamic Searchable Encryption with Optimal Search in the Presence of Deletions," in *Proceedings of the USENIX Security Symposium*, 2022.

[78] P. Mondal, J. G. Chamani, I. Demertzis, and D. Papadopoulos, "I/O-Efficient Dynamic Searchable Encryption meets Forward & Backward Privacy," in *Proceedings of the USENIX Security Symposium*, 2024.

[79] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis, "The Loopix Anonymity System," in *Proceedings of the USENIX Security Symposium*, 2017.

[80] J. F. Raymond, "Traffic Analysis: Protocols, Attacks, Design Issues, and Open Problems," in *Proceedings of the Workshop on Designing Privacy Enhancing Technologies*, 2001.

[81] G. Danezis, "Statistical Disclosure Attacks: Traffic Confirmation in Open Environments," in *Proceedings of the Conference on Information Security*, 2003.

[82] A. Pfitzmann, *Diensteintegrierende Kommunikationsnetze mit Teilnehmerüberprüfbarem Datenschutz*. Springer-Verlag, 2013.

[83] J. Hayes, C. Troncoso, and G. Danezis, "TASP: Towards Anonymity Sets that Persist," in *Proceedings of the ACM Workshop on Privacy in the Electronic Society*, 2016.

[84] M. Edman, F. Sivrikaya, and B. Yener, "A Combinatorial Approach to Measuring Anonymity," in *Proceedings of the IEEE Conference on Intelligence and Security Informatics*, 2007.

[85] E. Dauterman, V. Fang, N. Crooks, and R. A. Popa, "Reflections on Trusting Distributed Trust," in *Proceedings of the ACM Workshop on Hot Topics in Networks*, 2022.

[86] E. Dauterman, V. Fang, I. Demertzis, N. Crooks, and R. A. Popa, "Snoopy: Surpassing the Scalability Bottleneck of Oblivious Storage," in *Proceedings of the ACM Symposium on Operating Systems Principles*, 2021.

	Seattle	Enron
Original	28,573,566	517,401
Split	56,057,732	3,130,272
Senders	54,900,229	3,130,272
Receivers	55,968,045	3,130,272
Time	53,824,943	2,879,512
Final Emails	52,765,722	2,879,512
Final Users	547,631	69,295

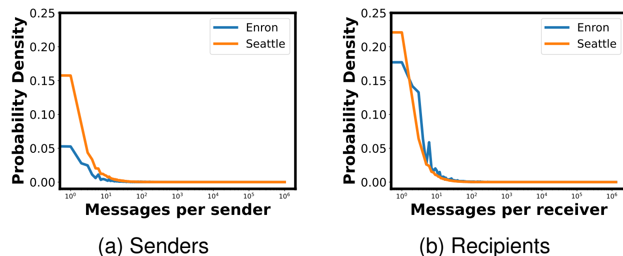
Figure 12. Cleaning Enron and Seattle. Emails with multiple recipients are treated as multiple messages. We filtered the datasets on availability of sender and receiver information. We also excluded messages with timestamps that were outside the timeframe of the datasets. The numbers reported are the number of valid messages after filtering.

Appendix A. Datasets

We use two datasets to measure the costs of assumptions and bandwidth limitations in traffic analysis resistant systems under real workloads. Enron [33] is a very well-used dataset and has been used to evaluate intersection attacks [34] in the context of anonymity research. Enron was made public during litigation against the Enron Corporation and contains internal email. Seattle [35] is a new dataset that was gathered via a Freedom of Information Act request. It contains all of the email sent to and from the Seattle city government from January to March 2017.

To conduct our experiments we need three fields per email: sender, recipient, and timestamp. Following the approach of Oya *et al.* [34], we split messages to many recipients into many messages for one recipient. We then filtered these datasets so that the sender and receiver fields were present and the timestamp of the message was correct. Here by correct we mean that the timestamp is between July 16, 1985 and December 3, 2001 for Enron (this corresponds to the day it was founded to the day it filed for bankruptcy) and from January 1, 2017 and April 1, 2017 for Seattle (this corresponds to the stated time frame of the dataset). In Fig. 12, we break down the number of messages after each step in the cleaning process. Enron is well structured, so we lose no messages due to unavailable sender and receivers, though some are excluded based on timing. Seattle is not as well structured and has numerous missing fields, thus we exclude $\sim 6\%$ of the total messages. After cleaning we are left with 2.9M messages from 69K users in Enron and 53M messages from 550K users in Seattle.

Enron and Seattle are very different datasets. Seattle is far denser than Enron with 53M messages sent in three months compared with 2.9M sent in over fifteen years. The characteristics of users are also very different as we see in Fig. 13. Though in both datasets more users send and receive small numbers of messages, this is more pronounced in Seattle. Seattle also has more extreme outliers in the number of messages individual users send and receive than Enron.



(a) Senders (b) Recipients

Figure 13. The probability density functions of Enron and Seattle. The x-axis is the number of messages a sender/recipient receives, while the y-axis is the probability that a user sends/receives that number of messages. We see from this that Seattle is more skewed than Enron, with more mass concentrated in the low numbers of messages and more extreme outliers.