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—deferred retrieval—that guarantees traffic analysis resistance under weak, implementable user assumptions, leading to practical deployments. We introduce Sparta systems, practical and scalable instantiations of deferred retrieval that can horizontally scale, 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. For reference, our low-latency Sparta supports latencies of less than 1 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. To ensure true privacy, messaging systems must protect both metadata and content, hiding the connection between senders and recipients.

Metadata-privacy has been a topic of major 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, but if we observe Alice sending a message to the service and Bob receiving a message from the service 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 a globally 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 [13], [21] assume all users will fetch at a fixed rate. 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 the messages. Most often, the mailbox

size k is 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}. This assumption is unrealistic because users cannot always be expected to be online to send a message in every round in mobile environments where they may lose connection. Groove [11] addressed this problem, by designing components to enforce that one message per user is submitted each round to its internal mixnet.

While global bandwidth restrictions, like the one message assumption, prevent traffic analysis [27], [28], they lead to significant and previously unacknowledged problems in a real deployment. If 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 [29], 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 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 user assumptions? And if so,
2) Can they be efficiently implemented to resist global adversaries?

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 weak 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 scale horizontally, achieve high throughput, and support multiple concurrent conversations without message loss. These Sparta systems implement deferred retrieval to resist traffic analysis by adversaries observing the network and Intel SGX using oblivious algorithms and data structures to resist attacks by adversaries observing the system itself.

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 1 millisecond on database sizes of 2^{20}. Sparta-D scales horizontally, and can scale to database sizes of 2^{23} with less than 1 second of latency. Sparta-SB is based on sort and can scale to deliver over 700,000 100B messages per second on a single 48-core server. This throughput is $15\times$ greater than that of the current highest throughput system, Groove [11], which imposes global bandwidth restrictions. Because our systems are built on top of SGX, they are straightforward to deploy securely [30] and can immediately begin protecting metadata in the long term for those who need it.

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 system model—deferred retrieval—that achieves traffic analysis resistance with less stringent assumptions on users. We evaluate the practical costs of this model under real email workloads and find it to be orders of magnitude cheaper than existing proposals for traffic analysis systems, and supports sub-minute latencies with an overhead of less than 400B per minute. This represents a practical network overhead. 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 this model. Sparta-LL is optimized for low processing latency, Sparta-SB is optimized for high-throughput on existing workloads, and Sparta-D is designed to scale horizontally. These systems implement deferred retrieval and 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 and horizontal scalability. Our Sparta implementation, is aligned with the Signal Messaging App[1] (one of the most secure messaging apps), which combines hardware enclaves (Intel SGX) with oblivious computations for private contact discovery.

1. `https://signal.org`

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 addressed 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 that it will is dependent on characteristics of the dataset. 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 found 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 not been tampered with. Isolation ensures the secure separation of a portion of the system's memory, known as the Enclave Page Cache (EPC), which stores both 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 standard to ensure that the algorithms running in hardware enclaves are oblivious [43], [44], [45], [46], [47], [48], [49], [50].

Obliviousness. A memory, algorithm, or data structure is considered oblivious if, for any two sequences of operations of the same size, the resulting access patterns (i.e., the sequence of memory accesses) are computationally indistinguishable to anyone except the client—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). Oblivious RAM (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 access patterns observed in the logical memory do not reveal any information about the access patterns in the original memory. 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 their access patterns.

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 use 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], have worse performance 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 $(!(\mathsf{c}-1)\&\mathsf{a})|((\mathsf{c}-1)\&\mathsf{b})$. In our pseudocode, we include if statements, 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 except onion routers, 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 amount of 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

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 provides long-term traffic analysis resistance under weak user assumptions while imposing no system-wide bandwidth restrictions on in-bound and out-bound 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). Furthermore, 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 on databases of size 2^{15}, 32K, and 2M, respectively.

memory accesses, data on the memory bus, in main memory, as well as 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.

Performance Metrics. For our systems evaluations we focus on two metrics: latency and throughput. 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 even attempted to measure these costs. Formally,

$$L = l_p + l_u + l_s, \tag{1}$$

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 security, 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], [34], [64]. The overhead would be 1 message per the length of the 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 realistically send, 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 new 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 [3] devoted to attacks against it, 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 this model 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.

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 [65], [66], 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.

Family	Leakage	Secure?
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.

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 [67], [68], [69] and Searchable Encryption [70], [71]. In general, leakage functions are composed of two subleakages: the leakage on the sender 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 on its own 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 send and the time they were delivered beyond that it was sent in 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 [72] 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 [73]. 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 $(S_t, R_{[t_i,t_{i+1}]})$. Utilzing changes in the sender and receiver traffic, adversaries can correlate communicating users using intersection/statistical disclosure attacks [34], [64], [74]. 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 [64], [75].

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 [76]. 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 [27], [77]. Second, it 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.

```
send(r, m; US, MS)
 1: nexttail ← U(0, 2^l − 1)
 2: rand ← U(0, 2^l − 1)
 3: (head, tail) ← US.update(r, (head, nexttail))
 4: MS.access(write, rand, (r, tail, nexttail, m))
fetch(r, k; US, MS)
 1: (first, last) ← US.update(r, (last, last))
 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: return M
```

Figure 3. Our ODS solution. Because US and MS are oblivious data structure, accesses leak nothing beyond their size. We assume for traffic analysis resistance that k is at most a function of (S_t, R).

3.2. Deferred Retrieval

In this section we describe deferred retrieval—a new way for a metadata-private messaging system to operate such that traffic does not leak correlations between communicating users. Deferred retrieval 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 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 do not leak any more information than (S_t, R).

This flexibility allows deferred retrieval to use strictly less overhead that globally set bandwidth restrictions for the same latency. Suppose k_i is the largest number of messages received in some latency threshold, L, for each user. This implies that if each user fetches k_i per L unit of 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

$$\sum_i k_i \leq n \max_i(k_i). \tag{2}$$

This relaxation also allows for many attractive usability properties [11], but all reduce to allowing users to vary their download rates, so while existing work set download rates globally, we set 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 cellular networks can reduce their rate without leaking information about their communication patters. 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.

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 sizes mailboxes. And, finally, systems must support efficient fetching and padding to variable numbers of messages. This functionality is not supported in existing systems.

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. 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 for when the number of users is larger 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. And finally, Sparta-D, combines ideas from both and achieve high throughput with large message stores.

Because they implement our theoretical model, the systems we describe here 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 already suffers from scalability problems even without these additional primitives, so we instead turn to hardware enclaves running oblivious algorithms to implement this model. 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 [30], and the lack of deployed globally secure systems in the anytrust model casts doubt on its real-world applicability. In contrast, Signal [78] 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 fetch sizes. 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. A push operation takes a mailbox identifier and an element and inserts the element at the tail of the queue. A pop operation takes a mailbox and a value, k, and returns the top k elements from the specified queue. 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.

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 an ORAM (*e.g.*, 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 message store. On a fetch, the user looks up the head of their queue in the message store, then 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. We first precompute the address for the next send so that this can be stored in the new 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 with the precomputed address of 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 head of the queue from the user store, then then iterate k times making an oblivious request to the message store in each iteration. 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 access we make to the message store. Because the user store is oblivious accesses to it do not reveal anything about the recipient accesses. 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 queue state from the user store. Note that as part of our leakage function we allow k to be leaked so long as k does not depend on the true volume of messages a user receives, thus if we assume this is true, iterating k times does not leak any additional information. Within the loop, we give an if statement for simplicity, but in reality this is implemented 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 record accessed. Finally, updating the returned messages M occurs independently of the value selected and this leaks 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 cost of an access in PathORAM is $O(\log M)$ 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 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

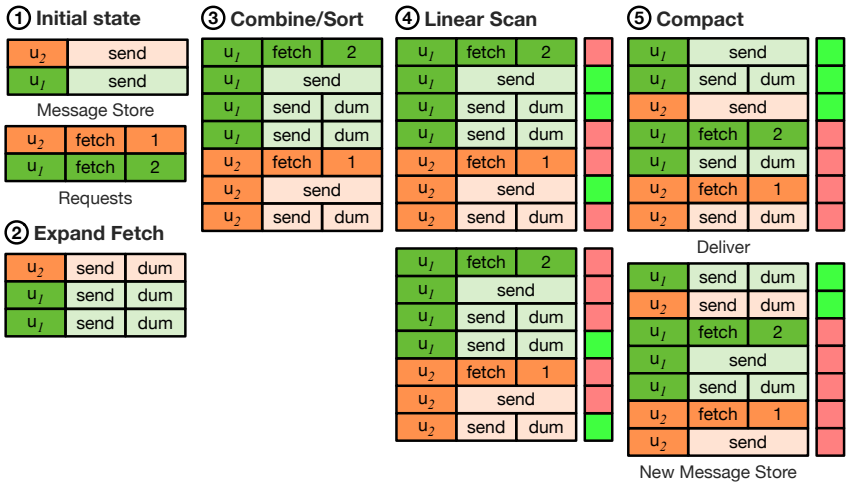


Figure 4. Fetching from Sparta-SB. This operation can be performed in an oblivious sort, a linear scan, and two oblivious compactions. Because the volumes of the requests, k_i, are public we simply take the first $\sum k_i$ from the messages compacted for delivery. Because the volume of incoming messages 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: Let R be the set of read requests, (r, k). For each (r, k) insert write 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 write, 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 fetch tuples, then remove these from M.

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. 5.

(see Fig. 8). In this section we design a scheme that implements the same oblivious multiqueue functionality but trades asymptotic complexity to achieve much higher throughput by instead 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 delivery via a linear scan. Afterward we use two oblivious compactions [46] to filter out messages the have been marked for delivery and the records that should be included in the updated state of the system (see Fig. 5).

Detailed Description. The above description leaves out several important details. In the case that a user issues a fetch for more messages that 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, we must introduce a new type of messages: dummies. For a fetch of size k we must insert k dummy messages. We then sort so that fetches precede sent messages precede dummy messages. This ensures we will always fetch real messages before fetching 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. We then mark the first k messages. These conditionals are implemented using oblivious selection. After we complete the scan, messages that ought to be deliver 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 the obliviousness of the primitives and that the number of sends and fetches are public. Because we first obliviously sort and pad with k dummies, we are guaranteed that each fetch has at least k elements, thus one users' 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 sizes, the number of sends, and the total number of 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. The same is true

of the compaction of the new elements of the message store.
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 routine [46] are $O(M \log^2 M)$ and $O(M \log M)$ respectively, thus the overall 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 our experimental evaluation, but with each batch we operate over the entire size of the message store. In this solution 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 subqueues used to store messages. The queue maintainer is similar to the user store in Sparta-LL, but instead it is implemented using sort and compaction to improve throughput. The queue maintainer is similar to the user store in Sparta-LL, but instead it 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 subqueues. The subqueues are themselves oblivious data structures run in enclaves. In Fig. 6 we give a high-level overview of the architecture of distributed Sparta and in Fig. 7 we give the pseudocode of the queue maintainer.
Detailed Description. In Sparta-D the queue maintainer translate 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 they may be obliviously counted and added to the queueing counts. Sparta-D then takes a hash of the user's identifier and these counts to create a unique object id for each object to store. Using Theorem 3 from Snoopy [79], we can obliviously construct sub-batches of requests using only the number of subqueues and the number of request. This theorem gives cryptographic upper bounds on the number of randomly distributed ids that fall into particular bins, thus the size of the editbatches reveals no information. The subqueues then return the values associated with the object id, Spart-D computes one more sort to reorder the batches and delivers the messages.
Security. The security of Sparta is due to the fact that all operations are based on public information. The sort to group messages is oblivious. The linear scan is over a publicly sized datastructure. The conditionals within the loop are implemented obliviously and the compaction to remove requests for the subqueues is also oblivious. Because at the end of the construction of the indices all are unique, we can apply Theorem 3 from Snoopy [79] to obliviously construct the sub-batches. These sub-batches are send to the oblivious subqueues, thus the operation at the subqueues is also oblivious.
Efficiency. Sparta is more efficient than the sort based solution when the size of the message store is much greater than the number of users. In this case the extra costs of maintaining the distributed queue 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 subqueues. The cost of Sparta-D when it is highly distributed 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 broadcast in §5.1 and our Sparta systems in §5.2. These experiments give a complete study of the latency (see Eq. 1) and overhead of Sparta, under the assumptions of deferred broadcast. 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] contain real email metadata, which we use to evaluate the costs of traffic analysis resistance for our system model. After filtering for the availability of the necessary fields (sender, receiver, time), we have 2.9 million emails from 69,000 users over 15 years in Enron, and 53 million emails from 550,000 users over 3 months in Seattle. Using real user data is critical for model evaluations, because actual user latencies depend on the arrival and departure rates of their message queues. 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 for processing.

We run our systems experiments on Microsoft Azure, which provides support for Intel SGXv2 using the `DC48sv3` class of VMs with 48 cores and 384GB memory. We implement our systems in Rust with ∼1600 lines of code using the Fortanix Enclave Development Platfrom (EDP). We also provide our own parallel Rust implementations of bitonic sort [58] and ORCompact [46], using constant-time CMOV-based oblivious swaps as in [46]. We run each experiment 10 times and report the mean of the measurements. All of our code will be made open source.

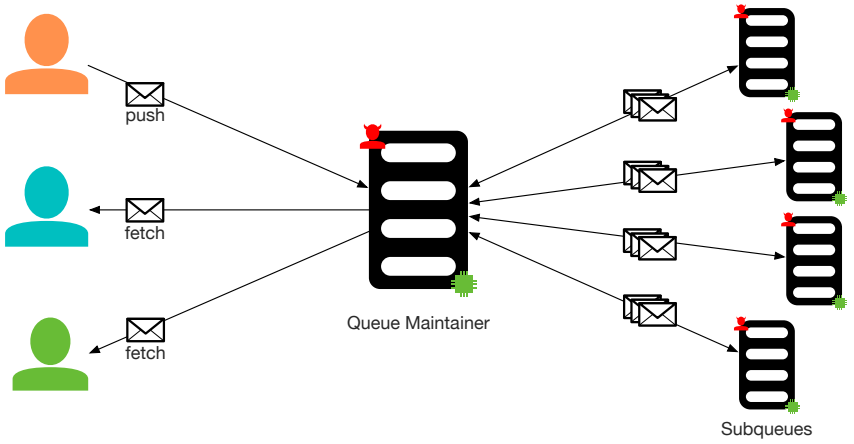


Figure 6. Sparta maintains correct queueing semantic among many smaller subqueues.

CreateObjectIds$(S, R; US)$

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.
For each (r, k) expand them in k write tuples $(r, \mathsf{fetch}, dummy)$ as R.
Take S and R together, and expand them to be of the form $(r, op, b, first, last, key, m)$.
Take US and expand it to be of the form $(r, \mathsf{ms}, b, first, last, key, m)$
Let M be the R, MS, and S sorted together by (r, op), where $\mathsf{ms} < \mathsf{fetch} < \mathsf{send}$.
Let $w_i = 0$, $r_i = 0, k_i = 0, next = M[1].r$
for $(r, op, first, last, key, m) \in M$ **do**
    Set $w_i = first$ if $op = \mathsf{ms}$, $w_i + 1$ if $op = \mathsf{write}$, w_i if $op = \mathsf{read}$.
    Set $r_i = \max(last, w_i)$ if $op = \mathsf{ms}$, $r_i + 1$ if $op = \mathsf{read}$, r_i if $op = \mathsf{write}$.
    Set $k_i = key$ if $op = \mathsf{ms}$ else k_i.
    Set $b = 0$ if $next = r$ else 1.
    Set $next = M[i + 1].r$.
end for
Obliviously compact M on b, then take the top n records as US.
Oblivious compact M on $op \neq \mathsf{ms}$, then take the top $|S| + |R|$ as the requests as M.

Figure 7. This code is one routine within Sparta that translates send and fetch requests into unique requests for objects store in the subqueues. Because these object ids are unique we can apply Theorem 3 from Snoopy [79] to obliviously construct batches of requests for the subqueues.

5.1. Model Evaluation

In this section we quantify the costs of different assumptions and bandwidth restrictions systems impose in order 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. We choose these metrics as there is widely acknowledged to be a trilemma between latency, overhead, and anonymity [63], [80].

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) [5], [6], [8], [12], [16] and differentially private systems [9], [11], 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 ($l_p = 0$), we can compute latencies and overheads solely due to bandwidth restrictions for traffic analysis resistance. This simplifies all systems with a particular bandwidth restriction into a single ideal representation, specified only by bandwidth limitations. This approach favors prior systems, which incur higher processing costs than Sparta due to their use of more expensive primitives like PIR and MPC (i.e., l_p is significantly lower in Sparta).

Given a send/fetch interval, a number of messages to send/fetch, and the sender, receiver, and timing of messages,

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

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

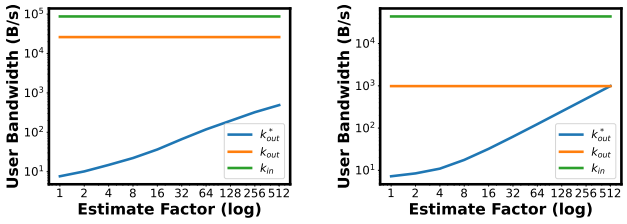
we can calculate the time messages enter and exit this ideal system, as well as the amount of traffic per interval. As an example, if we specify that all users must send exactly one message per five second interval, if one user wants to send two messages during that interval, the second will be deferred 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 systems that set per-user bandwidth limitations, 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, the optimal setting to guarantee this latency is $k = \max_i(k_i)$. For example, if we know that in one minute, we will not receive more that 100 messages we can set the download rate to 100 and never have messages delivered more than one minute after they are sent in an ideal system.

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 relatively confident that they will not receive more than 1000 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 1 denotes perfect knowledge of future behavior; 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, and chose this range as a superset of the reasonable range of user estimations. We set the maximum target latency to be 60s, to capture text messaging scenarios where users expect to receive their messages relatively quickly (within a minute).

Results. To the best of our knowledge we are the first to quantify the costs of different assumptions for traffic analysis, and we report our results in Fig. 9. The x-axis of these figures represents the estimation factor for our work only, 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 implies that to support sending short 128B messages with at most 60s of latency and achieving traffic analysis



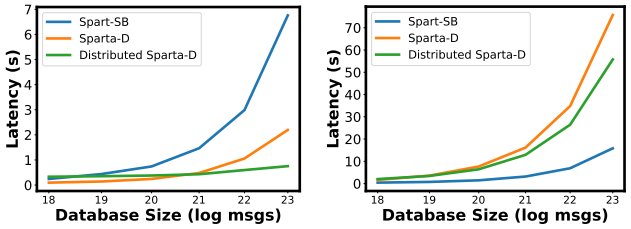
(a) Seattle (b) Enron

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.

resistance, the average user in our systems will download just ∼6B/s, meaning the average user receives at most about 1 message every 20 seconds. This is a practical amount of traffic. For comparison streaming music requires users to download about 3KB/s. Prior systems would have required they receive and send at about 20 and 60 KB/s respectively! In Enron, the gap between our model and prior models is smaller due to Enron's data sparsity—see Fig. 9. Comparing optimal settings of k_i both in our work and prior work shows that our approach results in a $54\times$ reduction in overhead.

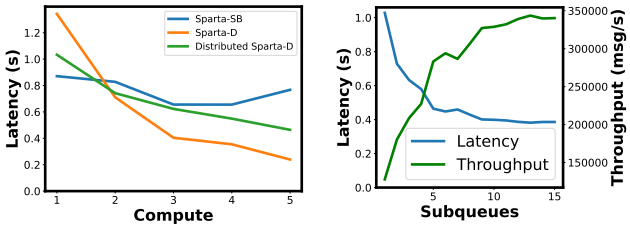
While optimally setting k_i is unrealistic (both for existing work and ours), users of our systems need only the coarsest estimates of k_i to see significant improvements in overhead compared to prior work. Estimations within a factor of a power of 10 of the true value give us orders of magnitude improvements over prior work in both datasets. Specically, when $x = 16$, *i.e.*, the estimate of the true rate is with a factor of 16 for our work, still gives a $713\times$ and $30\times$ improvement over existing work in Seattle and Enron, respectively. Such estimations may be practical for real users to properly set.

As we expect, with worse estimations (x increasing) we see that in both datasets the amount of traffic per user increases significantly and the graphs converge. This occurs when the estimation factor for our system exceeds the maximum number of messages sent/received in an interval in prior systems. 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, thus to have at most 60 seconds of latency we must set our send rates higher to accommodate these bursts. As an example, if a user sends a group email, following prior work [34], we represent this as individual message being sent to each member of the group, resulting in large bursts of messages. Finally, 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 in a given round grows faster over all users than it does for individual users.

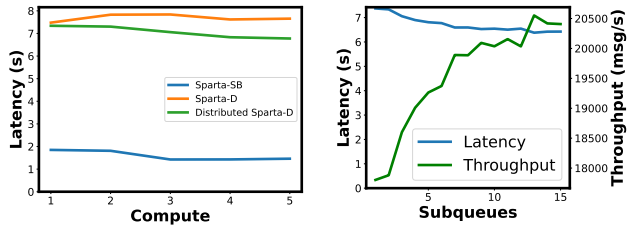


(a) Scaling up Storage (b) Scaling up Legacy

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 subqueues. On the Legacy dataset, which Sparta-SB is optimized for, we observe it gains a large advantage even over distributed Sparta-D.



(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.

5.2. System Evaluation

Our Sparta systems are practical instantiations of our theoretical model providing practical traffic analysis resistance. In this section, we demonstrate how Sparta systems scale using the Storage and Legacy datasets 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 non-distributed version with the queue maintainer and 5 subqueues running in parallel, each with 8 threads; the second runs a distributed version with the queue maintainer and 15 subqueues, each with 48 threads. For Sparta-SB, we allocate a total of 48 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). We see that 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 distributed experiment this trend is even more pronounced. A Sparta-D instance with 15 subqueues 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. The reason for this is that Sparta-D does additional work 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, but Sparta-SB is. As we would expect, for the Legacy dataset, Sparta-SB significantly outperforms even the distributed 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 manager becomes the dominating factor.

Sparta-LL predictably has low latency and throughput. Because general messaging systems must support many concurrent requests, it is not suited to this task.

Experiment #2: Scaling Compute. In this experiment we investigate how Sparta-SB and Sparta-D scale with additional compute resources on Storage and Legacy with fixed size 2^{20}. In the first set of experiments we 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 subqueues from one to five with 8 threads allocated to each and 8 threads allocated to the queue maintainer. For distributed Sparta-D, we allocate additional machines. In order to keep parity with single-machine Sparta-D, for Sparta-SB we vary the number of threads from 16 to 48 in increments of 8. In the second set of 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 subqueues from 1 to 15. 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. We see that for the Storage workload, Sparta-D benefits the most from additional resources in the single machine case, outpacing Sparta-SB by a factor of 2. The reason for this is that the workload is relatively small and thus parallelizing the subqueues significantly reduces

the cost of those operations. In the distributed case, the advantage of distributing this workload is not as apparent due to network latency. In the sort-based version of Sparta compute is not the limiting factor in either workload. We note that on a single 48-core machine, Sparta-SB increases throughput by $15\times$ compared with a 150-machine Groove cluster [11].

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 10 subqueues, 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 subqueues.

6. Discussion

The results of our theory and systems implementations are that we now have traffic analysis resistant systems that are orders of magnitude more efficient than existing work. This is due both to deferred retrieval, which provides traffic analysis resistance under weakened assumptions, and our efficient implementations using Intel SGX. The model that we are proposing is not complicated, so why has it not been done before, especially since it allows us to relax user assumptions? One possible reason for this is that the assumptions of prior work, namely the one message assumption, concealed the significance of the traffic analysis problem in prior synchronous systems. Similarly, because the costs of these assumptions have never been quantitatively measured the lack of realism of the these global bandwidth limitations was not made explicit. Importantly, though we implement our systems using Intel SGX, deferred retrieval is not tied to this trust model, thus one direction of future research is to support the functionality of deferred retrieval in more conventional implementaion models, *e.g.*, MPC, anytrust, FHE. However, an advantage of Intel SGX is that it is trivial to deploy securely [30], as it does not require multiple system operators from different trust domains, as is the case in MPC/anytrust.

As discussed in §5.1, deferred retrieval requires that users' download rates be carefully set to minimize latency and overhead due to queueing at the messaging service. Theoretically, optimally set rates in deferred retrieval will always lead to better performance that optimally set global bandwidth rates. However, the degree to which deferred retrieval will out perform existing work is dependent on features of the workloads. This is clear from the results of §5.1, where the gap between our work and prior work was much greater in the Seattle dataset as compared with the Enron dataset. The features of datasets that lead to better performance in our work is directly related to Eq. 2. From this, we can see that the greater the difference between the max and average user's rates, the better our systems will perform. In order to keep the required traffic rates low, workloads also should 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. While email data is not a perfect analog for the characteristics of messaging systems, unfortunately metadata is difficult to come by, with Seattle and Enron being the only suitable data we could find. Because of this, one direction for future research is to collect such metadata or construct representative synthetic data based on user studies.

7. Conclusion

We introduce the first traffic analysis-resistant metadata private communication system with reasonable 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, and we built three versions of this this functionality on Intel SGX [36]. Our experiments demonstrated that Sparta-SB and Sparta-D are signficantly more scalable than existing works.

References

[1] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The Second-Generation Onion Router," Naval Research Lab Washington DC, Tech. Rep., 2004.

[2] T. Wang and I. Goldberg, "Improved Website Fingerprinting on Tor," in *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, 2013, pp. 201–212.

[3] I. Karunanayake, N. Ahmed, R. Malaney, R. Islam, and S. K. Jha, "De-anonymisation attacks on tor: A survey," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 4, pp. 2324–2350, 2021.

[4] D. L. Chaum, "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms," *Communications of the ACM*, vol. 24, no. 2, pp. 84–90, 1981.

[5] H. Corrigan-Gibbs, D. Boneh, and D. Mazières, "Riposte: An Anonymous Messaging System Handling Millions of Users," in *2015 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 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1233–1252.

[7] A. Kwon, D. Lu, and S. Devadas, "XRD: Scalable Messaging System with Cryptographic Privacy," *arXiv preprint arXiv:1901.04368*, 2019.

[8] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford, "Atom: Horizontally Scaling Strong Anonymity," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 406–422.

[9] J. Van Den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich, "Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis ," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 137–152.

[10] D. Lazar, Y. Gilad, and N. Zeldovich, "Yodel: Strong Metadata Security for Voice Calls," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 211–224.

[11] L. Barman, M. Kol, D. Lazar, Y. Gilad, and N. Zeldovich, "Groove: Flexible {Metadata-Private} messaging," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 735–750.

[12] D. Lu and A. Kate, "Rpm: Robust anonymity at scale," *Cryptology ePrint Archive*, 2022.

[13] S. Eskandarian, H. Corrigan-Gibbs, M. Zaharia, and D. Boneh, "Express: Lowering the Cost of Metadata-hiding Communication with Cryptographic Privacy," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1775–1792.

[14] S. Eskandarian and D. Boneh, "Clarion: Anonymous Communication from Multiparty Shuffling Protocols," in *NDSS Symposium*, 2021.

[15] N. Alexopoulos, A. Kiayias, R. Talviste, and T. Zacharias, "MCMix: Anonymous Messaging via Secure Multiparty Computation," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1217–1234.

[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 *International conference on applied cryptography and network security*. Springer, 2017, pp. 557–578.

[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 *Annual Computer Security Applications Conference*, 2020, pp. 84–99.

[18] D. Kesdogan, D. Agrawal, and S. Penz, "Limits of Anonymity in Open Environments," in *International Workshop on Information Hiding*. Springer, 2002, pp. 53–69.

[19] G. Danezis, "The Traffic Analysis of Continuous-time Mixes," in *International Workshop on Privacy Enhancing Technologies*. Springer, 2004, pp. 35–50.

[20] S. Oya, C. Troncoso, and F. Pérez-González, "Meet the family of statistical disclosure attacks," in *2013 IEEE Global Conference on Signal and Information Processing*. IEEE, 2013, pp. 233–236.

[21] A. Vadapalli, K. Storrier, and R. Henry, "Sabre: Sender-anonymous messaging with fast audits," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1953–1970.

[22] S. Angel and S. Setty, "Unobservable Communication over Fully Untrusted Infrastructure," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 551–569.

[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 *International Conference on Applied Cryptography and Network Security*. Springer, 2004, pp. 16–30.

[24] D. Lazar and N. Zeldovich, "Alpenhorn: Bootstrapping Secure Communication without Leaking Metadata," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 571–586.

[25] D. Lazar, Y. Gilad, and N. Zeldovich, "Karaoke: Distributed Private Messaging Immune to Passive Traffic Analysis," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 711–725.

[26] H. Corrigan-Gibbs and B. Ford, "Dissent: Accountable Anonymous Group Messaging," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 340–350.

[27] D. Kesdogan, D. Agrawal, V. Pham, and D. Rautenbach, "Fundamental limits on the anonymity provided by the mix technique," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006, pp. 14–pp.

[28] A. Pfitzmann and M. Köhntopp, "Anonymity, Unobservability, and Pseudonymity— A Proposal for Terminology," in *Designing privacy enhancing technologies*. Springer, 2001, pp. 1–9.

[29] J. D. Little, "A Proof for the Queuing Formula: $L = \lambda W$," *Operations research*, vol. 9, no. 3, pp. 383–387, 1961.

[30] E. Dauterman, V. Fang, N. Crooks, and R. A. Popa, "Reflections on trusting distributed trust," in *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, 2022, pp. 38–45.

[31] J. R. Tyler and J. C. Tang, "When can i expect an email response? a study of rhythms in email usage," in *ECSCW 2003: Proceedings of the Eighth European Conference on Computer Supported Cooperative Work 14–18 September 2003, Helsinki, Finland*. Springer, 2003, pp. 239–258.

[32] I. Gamzu, Z. Karnin, Y. Maarek, and D. Wajc, "You will get mail! predicting the arrival of future email," in *Proceedings of the 24th International Conference on World Wide Web*, 2015, pp. 1327–1332.

[33] B. Klimt and Y. Yang, "The Enron Corpus: A New Dataset for Email Classification Research," in *European conference on machine learning*. Springer, 2004, pp. 217–226.

[34] S. Oya, C. Troncoso, and F. Pérez-González, "Understanding the effects of real-world behavior in statistical disclosure attacks," in *2014 IEEE International Workshop on Information Forensics and Security (WIFS)*. IEEE, 2014, pp. 72–77.

[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," Intel, Tech. Rep., 2013.

[37] ARM Limited, "Arm trustzone technology," ARM Holdings, Tech. Rep., 2004. [Online]. Available: https://developer.arm.com/documentation/102412/latest

[38] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," AMD, Tech. Rep., 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 *HASP*, 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 *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1041–1056.

[41] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software Grand Exposure: SGX cache attacks are practical," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 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 *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 557–574.

[43] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An Efficient Oblivious Search Index," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 279–296.

[44] A. Tinoco, S. Gao, and E. Shi, "{EnigMap}:{External-Memory} oblivious map for secure enclaves," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4033–4050.

[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 *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 283–298.

[46] S. Sasy, A. Johnson, and I. Goldberg, "Fast fully oblivious compaction and shuffling," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2565–2579.

[47] ——, "Waks-on/waks-off: Fast oblivious offline/online shuffling and sorting with waksman networks," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 3328–3342.

[48] J. G. Chamani, I. Demertzis, D. Papadopoulos, C. Papamanthou, and R. Jalili, "Graphos: Towards oblivious graph processing," *Proceedings of the VLDB Endowment*, vol. 16, no. 13, pp. 4324–4338, 2023.

[49] N. Ngai, I. Demertzis, J. G. Chamani, and D. Papadopoulos, "Distributed & scalable oblivious sorting and shuffling," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 153–153.

[50] G. Asharov, T. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi, "Bucket oblivious sort: An extremely simple oblivious sort," in *Symposium on Simplicity in Algorithms*. SIAM, 2020, pp. 8–14.

[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 (JACM)*, vol. 65, no. 4, pp. 1–26, 2018.

[52] G. Asharov, I. Komargodski, W. Lin, K. Nayak, and E. Shi, "Optorama: Optimal oblivious RAM," *IACR*, 2018.

[53] S. Patel, G. Persiano, M. Raykova, and K. Yeo, "Panorama: Oblivious RAM with logarithmic overhead," in *FOCS 2018*, 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 *Springer 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 *CRYPTO*, 2016.

[56] X. Wang, H. Chan, and E. Shi, "Circuit oram: On tightness of the goldreich-ostrovsky lower bound," in *CCS*, 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 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 215–226.

[58] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, 1968, pp. 307–314.

[59] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx," in *Proceedings of the 10th European Workshop on Systems Security*, 2017, pp. 1–6.

[60] M. Hähnel, W. Cui, and M. Peinado, "{High-Resolution} side channels for untrusted operating systems," in *ATC*, 2017.

[61] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *DIMVA*, 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 *CCS*, 2017.

[63] D. Das, S. Meiser, E. Mohammadi, and A. Kate, "Anonymity Trilemma: Strong Anonymity, Low Bandwidth Overhead, Low Latency–Choose Two," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 108–126.

[64] D. Kesdogan and L. Pimenidis, "The hitting set attack on anonymity protocols," in *International Workshop on Information Hiding*. Springer, 2004, pp. 326–339.

[65] N. Gelernter and A. Herzberg, "On the limits of provable anonymity," in *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, 2013, pp. 225–236.

[66] A. Hevia and D. Micciancio, "An indistinguishability-based characterization of anonymous channels," in *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 2008, pp. 24–43.

[67] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001, pp. 136–145.

[68] ——, "Security and composition of multiparty cryptographic protocols," *Journal of CRYPTOLOGY*, vol. 13, pp. 143–202, 2000.

[69] A. C.-C. Yao, "How to generate and exchange secrets," in *27th annual symposium on foundations of computer science (Sfcs 1986)*. IEEE, 1986, pp. 162–167.

[70] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," *Cryptology ePrint Archive*, 2013.

[71] I. Demertzis and C. Papamanthou, "Fast searchable encryption with tunable locality," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1053–1067.

[72] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis, "The Loopix Anonymity System," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1199–1216.

[73] J.-F. Raymond, "Traffic analysis: Protocols, attacks, design issues, and open problems," in *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*. Springer, 2001, pp. 10–29.

[74] G. Danezis, "Statistical Disclosure Attacks," in *IFIP International Information Security Conference*. Springer, 2003, pp. 421–426.

[75] A. Pfitzmann, *Diensteintegrierende Kommunikationsnetze mit teilnehmerüberpr üfbarem Datenschutz*. Springer-Verlag, 2013, vol. 234, (in German).

[76] J. Hayes, C. Troncoso, and G. Danezis, "Tasp: Towards anonymity sets that persist," in *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society*, 2016, pp. 177–180.

[77] M. Edman, F. Sivrikaya, and B. Yener, "A combinatorial approach to measuring anonymity," in *2007 IEEE Intelligence and Security Informatics*. IEEE, 2007, pp. 356–363.

[78] M. Marlinspike, "Advanced Cryptographic Ratcheting," 2013, [Online; accessed 23-February-2022]. [Online]. Available: https://whispersystems.org/blog/advanced-ratcheting/

[79] 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 SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 655–671.

[80] D. Das, C. Diaz, A. Kiayias, and T. Zacharias, "Are continuous stop-and-go mixnets provably secure?" *Cryptology ePrint Archive*, 2023.