Riposte: An Anonymous Messaging System Handling Millions of Users

Henry Corrigan-Gibbs, Dan Boneh, and David Mazières
Stanford University

April 18, 2017

Abstract

This paper presents Riposte, a new system for anonymous broadcast messaging. Riposte is the first such system, to our knowledge, that simultaneously protects against traffic-analysis attacks, prevents anonymous denial-of-service by malicious clients, and scales to million-user anonymity sets. To achieve these properties, Riposte makes novel use of techniques used in systems for private information retrieval and secure multi-party computation. For latency-tolerant workloads with many more readers than writers (e.g. Twitter, Wikileaks), we demonstrate that a three-server Riposte cluster can build an anonymity set of 2,895,216 users in 32 hours.

1 Introduction

In a world of ubiquitous network surveillance [1–5], prospective whistleblowers face a daunting task. Consider, for example, a government employee who wants to anonymously leak evidence of waste, fraud, or incompetence to the public. The whistleblower could email an investigative reporter directly, but *post hoc* analysis of email server logs could easily reveal the tipster's identity. The whistleblower could contact a reporter via Tor [6] or another low-latency anonymizing proxy [7–10], but this would leave the leaker vulnerable to traffic-analysis attacks [11–13]. The whistleblower could instead use an anonymous messaging system that protects against traffic analysis attacks [14–16], but these systems typically only support relatively small anonymity sets (tens of thousands of users, at most). Protecting whistleblowers in the digital age requires anonymous messaging systems that provide strong security guarantees, but that also scale to very large network sizes.

In this paper, we present a new system that attempts to make traffic-analysis-resistant anonymous broadcast messaging practical at Internet scale. Our system, called Riposte, allows a large number of clients to anonymously post messages to a shared "bulletin board," maintained by a small set of minimally trusted servers. (As few as three non-colluding servers are sufficient). Whistleblowers could use Riposte as a platform for anonymously publishing Tweet- or email-length messages and could combine it with standard public-key encryption to build point-to-point private messaging channels.

While there is an extensive literature on anonymity systems [17, 18], Riposte offers a combination of security and scalability properties unachievable with current designs. To the best of our knowledge, Riposte is the only anonymous messaging system that simultaneously:

1. protects against traffic analysis attacks,
2. prevents malicious clients from anonymously executing denial-of-service attacks, and
3. scales to anonymity set sizes of *millions* of users, for certain latency-tolerant applications.

We achieve these three properties in Riposte by adapting three different techniques from the cryptography and privacy literature. First, we defeat traffic-analysis attacks and protect against malicious servers by using a protocol, inspired by client/server DC-nets [14, 16], in which every participating client sends a fixed-length secret-shared message to the system's servers in every time epoch. Second, we achieve efficient disruption resistance by using a secure multi-party protocol to quickly detect and exclude malformed client requests [19–21]. Third, we achieve scalability by leveraging a specific technique developed in the context of private information retrieval (PIR) to minimize the number of bits each client must upload to each server in every time epoch. The tool we use is called a *distributed point function* [22, 23]. The novel synthesis of these techniques leads to a system that is efficient (in

This is the extended and corrected version of a paper by the same name that appeared at the *IEEE Symposium on Security and Privacy* in May 2015.

terms of bandwidth and computation) and practical, even for large anonymity sets.

Our particular use of private information retrieval (PIR) protocols is unusual: PIR systems [24] allow a client to efficiently read a row from a database, maintained collectively at a set of servers, without revealing to the servers which row it is reading. Riposte achieves scalable anonymous messaging by running a private information retrieval protocol *in reverse*: with reverse PIR, a Riposte client can efficiently *write* into a database maintained at the set of servers without revealing to the servers which row it has written [25].

As we discuss later on, a large Riposte deployment could form the basis for an anonymous Twitter service. Users would "tweet" by using Riposte to anonymously write into a database containing all clients' tweets for a particular time period. In addition, by having read-only users submit "empty" writes to the system, the effective anonymity set can be much larger than the number of writers, with little impact on system performance.

Messaging in Riposte proceeds in regular *time epochs* (e.g., each time epoch could be one hour long). To post a message, the client generates a *write request*, cryptographically splits it into many shares, and sends one share to each of the Riposte servers. A coalition of servers smaller than a certain threshold cannot learn anything about the client's message or write location given its subset of the shares.

The Riposte servers collect write requests until the end of the time epoch, at which time they publish the aggregation of the write requests they received during the epoch. From this information, anyone can recover the set of posts uploaded during the epoch, but the system reveals no information about who posted which message. The identity of the entire set of clients who posted during the interval is known, but no one can link a client to a post. (Thus, each time epoch must be long enough to ensure that a large number of honest clients are able to participate in each epoch.)

In this paper, we describe two Riposte variants, which offer slightly different security properties. The first variant scales to very large network sizes (millions of clients) but requires three servers such that no two of these servers collude. The second variant is more computationally expensive, but provides security even when all but one of the $s > 1$ servers are malicious. Both variants maintain their security properties when network links are actively adversarial, when all but two of the clients are actively malicious, and when the servers are actively malicious (subject to the non-collusion requirement above).

The three-server variant uses a computationally inexpensive multi-party protocol to detect and exclude malformed client requests. (Figure 1 depicts this protocol at a high-level.) The s-server variant uses client-produced zero-knowledge proofs to guarantee the well-formedness of client requests.

Unlike Tor [6] and other low-latency anonymity systems [8, 10, 15, 26], Riposte protects against active traffic analysis attacks by a global network adversary. Prior systems have offered traffic-analysis-resistance only at the cost of scalability:

- Mix-net-based systems [27] require large zero-knowledge proofs of correctness to provide privacy in the face of active attacks by malicious servers [28–32].
- DC-nets-based systems require clients to transfer data *linear* in the size of the anonymity set [14, 16] and rely on expensive zero-knowledge proofs to protect against malicious clients [33, 34].

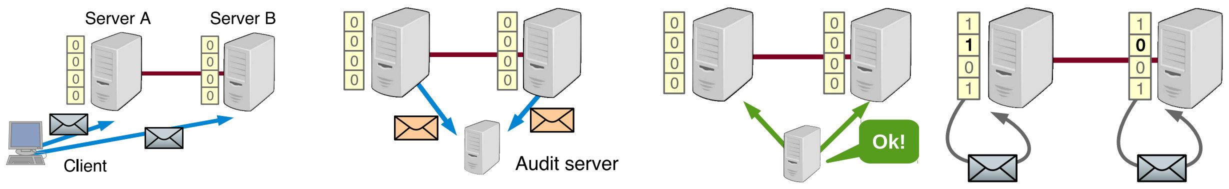
We discuss these systems and other prior work in Section 7.

Experiments. To demonstrate the practicality of Riposte for anonymous broadcast messaging (i.e., anonymous whistleblowing or microblogging), we implemented and evaluated the complete three-server variant of the system. When the servers maintain a database table large enough to fit 65,536 160-byte Tweets, the system can process 32.8 client write requests per second. In Section 6.3, we discuss how to use a table of this size as the basis for very large anonymity sets in read-heavy applications. When using a larger 377 MB database table (over 2.3 million 160-byte Tweets), a Riposte cluster can process 1.4 client write requests per second.

Writing into a 377 MB table requires each client to upload less than 1 MB of data to the servers. In contrast, a two-server DC-net-based system would require each client to upload more than 750 MB of data. More generally, to process a Riposte client request for a table of size L, clients and servers perform only $O(\sqrt{L})$ bytes of data transfer.

The servers' AES-NI encryption throughput limits the rate at which Riposte can process client requests at large table sizes. Thus, the system's capacity to handle client write request scales with the number of available CPU cores. A large Riposte deployment could shard the database table across k machines to achieve a near-k-fold speedup.

We tested the system with anonymity set sizes of up to 2,895,216 clients, with a read-heavy latency-tolerant microblogging workload. To our knowledge, this is the largest anonymity set *ever constructed* in a system defend-



(a) A client submits one share of its write request to each of the two database servers. If the database has length L, each share has length $O(\sqrt{L})$.

(b) The database servers generate blinded "audit request" messages derived from their shares of the write request.

(c) The audit server uses the audit request messages to validate the client's request and returns an "OK" or "Invalid" bit to the database servers.

(d) The servers apply the write request to their local database state. The XOR of the servers' states contains the clients message at the given row.

Figure 1: The process of handling a single client write request. The servers run this process once per client in each time epoch.

ing against traffic analysis attacks. Prior DC-net-based systems scaled to 5,120 clients [16] and prior verifiable-shuffle-based systems scaled to 100,000 clients [29]. In contrast, Riposte scales to millions of clients for certain applications.

Contributions. This paper contributes:

- two new bandwidth-efficient and traffic-analysis-resistant anonymous messaging protocols, obtained by running private information retrieval protocols "in reverse" (Sections 3 and 4),
- a fast method for excluding malformed client requests (Section 5),
- a method to recover from transmission collisions in DC-net-style anonymity systems,
- experimental evaluation of these protocols with anonymity set sizes of up to 2,895,216 users (Section 6).

In Section 2, we introduce our goals, threat model, and security definitions. Section 3 presents the high-level system architecture. Section 4 and Section 5 detail our techniques for achieving bandwidth efficiency and disruption resistance in Riposte. We evaluate the performance of the system in Section 6, survey related work in Section 7, and conclude in Section 8.

2 Goals and Problem Statement

In this section, we summarize the high-level goals of the Riposte system and present our threat model and security definitions.

2.1 System Goals

Riposte implements an anonymous bulletin board using a primitive we call a *write-private database scheme*. Riposte enables clients to write into a shared database, collectively maintained at a small set of servers, without revealing to the servers the location or contents of the write. Conceptually, the database table is just a long fixed-length bitstring divided into fixed-length rows.

To write into the database, a client generates a *write request*. The write request encodes the message to be written and the row index at which the client wants to write. (A single client write request modifies a single database row at a time.) Using cryptographic techniques, the client splits its write request into a number of shares and the client sends one share to each of the servers. By construction of the shares, no coalition of servers smaller than a particular pre-specified threshold can learn the contents of a single client's write request. While the cluster of servers must remain online for the duration of a protocol run, a client need only stay online for long enough to upload its write request to the servers. As soon as the servers receive a write request, they can apply it to to their local state.

The Riposte cluster divides time into a series of epochs. During each time epoch, servers collect many write requests from clients. When the servers agree that the epoch has ended, they combine their shares of the database to reveal the clients' plaintext messages. A particular client's anonymity set consists of all of the honest clients who submitted write requests to the servers during the time epoch. Thus, if 50,000 distinct honest clients submitted write requests during a particular time epoch, each honest client is perfectly anonymous amongst this set of 50,000 clients.

The epoch could be measured in time (e.g., 4 hours), in a number of write requests (e.g., accumulate 10,000 write requests before ending the epoch), or by some more complicated condition (e.g., wait for a write request signed from each of these 150 users identified by a pre-defined list of public keys). The definition of what constitutes an epoch is crucial for security, since a client's anonymity set is only as large as the number of honest clients who submit write requests in the same epoch [35].

When using Riposte as a platform for anonymous microblogging, the rows would be long enough to fit a Tweet (140 bytes) and the number of rows would be some multiple of the number of anticipated users. To anonymously Tweet, a client would use the write-private database scheme to write its message into a random row of the database. After many clients have written to the database, the servers can reveal the clients' plaintext Tweets. The write-privacy of the database scheme prevents eavesdroppers, malicious clients, and coalitions of malicious servers (smaller than a particular threshold) from learning which client posted which message.

2.2 Threat Model

Clients in our system are *completely untrusted*: they may submit maliciously formed write requests to the system and may collude with servers or with arbitrarily many other clients to try to break the security properties of the system.

Servers in our system are trusted for availability. The failure—whether malicious or benign—of any one server renders the database state unrecoverable but *does not* compromise the anonymity of the clients. To protect against benign failures, server maintainers could implement a single "logical" Riposte server with a cluster of many physical servers running a standard state-machine-replication protocol [36, 37].

For each of the cryptographic instantiations of Riposte, there is a threshold parameter t that defines the number of malicious servers that the system can tolerate while still maintaining its security properties. We make no assumptions about the behavior of malicious servers—they can misbehave by publishing their secret keys, by colluding with coalitions of up to t malicious servers and arbitrarily many clients, or by mounting any other sort of attack against the system.

The threshold t depends on the particular cryptographic primitives in use. For our most secure scheme, *all but one* of the servers can collude without compromising client privacy ($t = |\mathsf{Servers}| - 1$). For our most efficient scheme, *no two* servers can collude ($t = 1$).

2.3 Security Goals

The Riposte system implements a *write-private* and *disruption-resistant* database scheme. We describe the correctness and security properties for such a scheme here.

Definition 1 (Correctness)**.** *The scheme is* correct *if, when all servers execute the protocol faithfully, the plaintext state of the database revealed at the end of a protocol run is equal to the result of applying each valid client write requests to an empty database (i.e., a database of all zeros).*

Since we rely on all servers for availability, correctness need only hold when all servers run the protocol correctly.

To be useful as an anonymous bulletin board, the database scheme must be *write-private* and *disruption resistant*. We define these security properties here.

(s,t)-Write Privacy. Intuitively, the system provides (s,t)-*write-privacy* if an adversary's advantage at guessing which honest client wrote into a particular row of the database is negligibly better than random guessing, even when the adversary controls all but two clients and up to t out of s servers (where t is a parameter of the scheme). We define this property in terms of a *privacy game*, given in full in Appendix A.

Definition 2 ((s,t)-Write Privacy)**.** *We say that the protocol provides* (s,t)-write privacy *if the adversary's advantage in the security game of Appendix A is negligible in the (implicit) security parameter.*

Riposte provides a very robust sort of privacy: the adversary can select the messages that the honest clients will send and can send maliciously formed messages that depend on the honest clients' messages. Even then, the adversary still cannot guess which client uploaded which message.

Disruption resistance. The system is *disruption resistant* if an adversary who controls n clients can write into at most n database rows during a single time epoch. A system that lacks disruption resistance might be susceptible to denial-of-service attacks: a malicious client could corrupt every row in the database with a single write request. Even worse, the write privacy of the system might prevent the servers from learning which client was the disruptor. Preventing such attacks is a major focus of prior anonymous messaging schemes [14–16, 34, 38]. Under our threat model, we trust all servers for availability of the system (though not for privacy). Thus, our definition of disruption resistance concerns itself only with clients attempting to disrupt the system—we *do not* try to prevent servers from corrupting the database state.

We formally define *disruption resistance* using the following game, played between a challenger and an adversary. In this game, the challenger plays the role of all of the servers and the adversary plays the role of all clients.

1. The adversary sends n write requests to the challenger (where n is less than or equal to the number of rows in the database).

2. The challenger runs the protocol for a single time epoch, playing the role of the servers. The challenger then combines the servers' database shares to reveal the plaintext output.

The adversary wins the game if the plaintext output contains more than n non-zero rows.

Definition 3 (Disruption Resistance)**.** *We say that the protocol is* disruption resistant *if the probability that the adversary wins the game above is negligible in the (implicit) security parameter.*

2.4 Intersection Attacks

Riposte makes it infeasible for an adversary to determine which client posted which message *within* a particular time epoch. If an adversary can observe traffic patterns *across* many epochs, as the set of online clients changes, the adversary can make statistical inferences about which client is sending which stream of messages [39–41]. These "intersection" or "statistical disclosure" attacks affect many anonymity systems and defending against them is an important, albeit orthogonal, problem [41, 42]. Even so, intersection attacks typically become more difficult to mount as the size of the anonymity set increases, so Riposte's support for very large anonymity sets makes it less vulnerable to these attacks than are many prior systems.

3 System Architecture

As described in the prior section, a Riposte deployment consists of a small number of servers, who maintain the database state, and a large number of clients. To write into the database, a client splits its write request using secret sharing techniques and sends a single share to each of the servers. Each server updates its database state using the client's share. After collecting write requests from many clients, the servers combine their shares to reveal the plaintexts represented by the write requests. The security requirement is that no coalition of t servers can learn which client wrote into which row of the database.

3.1 A First-Attempt Construction: Toy Protocol

As a starting point, we sketch a simple "straw man" construction that demonstrates the techniques behind our scheme. This first-attempt protocol shares some design features with anonymous communication schemes based on client/server DC-nets [14, 16].

In the simple scheme, we have two servers, A and B, and each server stores an L-bit bitstring, initialized to all zeros. We assume for now that the servers *do not collude*—i.e., that one of the two servers is honest. The bitstrings represent shares of the database state and each "row" of the database is a single bit.

Consider a client who wants to write a "1" into row ℓ of the database. To do so, the client generates a random L-bit bitstring r. The client sends r to server A and $r \oplus e_\ell$ to server B, where e_ℓ is an L-bit vector of zeros with a one at index ℓ and $\oplus$ denotes bitwise XOR. Upon receiving the write request from the client, each server XORs the received string into its share of the database.

After processing n write requests, the database state at server A will be:

$$d_A = r_1 \oplus \cdots \oplus r_n$$

and the database at server B will be:

$$\begin{aligned} d_B &= (e_{\ell_1} \oplus \cdots \oplus e_{\ell_n}) \oplus (r_1 \oplus \cdots \oplus r_n) \\ &= (e_{\ell_1} \oplus \cdots \oplus e_{\ell_n}) \oplus d_A \end{aligned}$$

At the end of the time epoch, the servers can reveal the plaintext database by combining their local states d_A and d_B.

The construction generalizes to fields larger than $\mathbb{F}_2$. For example, each "row" of the database could be a k-bit bitstring instead of a single bit. To prevent impersonation, network-tampering, and replay attacks, we use authenticated and encrypted channels with per-message nonces bound to the time epoch identifier.

This protocol satisfies the write-privacy property as long as the two servers do not collude (assuming that the clients and servers deploy the replay attack defenses mentioned above). Indeed, server A can information theoretically simulate its view of a run of the protocol given only $e_{\ell_1} \oplus \cdots \oplus e_{\ell_n}$ as input. A similar argument shows that the protocol is write-private with respect to server B as well.

This first-attempt protocol has two major limitations. The first limitation is that it is not bandwidth-efficient. If millions of clients want to use the system in each time epoch, then the database must be at least millions of bits in length. To flip a *single bit* in the database then, each client must send *millions of bits* to each database, in the form of a write request.

The second limitation is that it is not disruption resistant: a malicious client can corrupt the entire database with a single malformed request. To do so, the malicious client picks random L-bit bitstrings r and r', sends r to server A, and sends r' (instead of $r \oplus e_\ell$) to server B. Thus, a single malicious client can efficiently and anonymously deny service to all honest clients.

Improving bandwidth efficiency and adding disruption resistance are the two core contributions of this work, and we return to them in Sections 4 and 5.

3.2 Collisions

Putting aside the issues of bandwidth efficiency and disruption resistance for the moment, we now discuss the issue of *colliding writes* to the shared database. If clients write into random locations in the database, there is some chance that one client's write request will overwrite a previous client's message. If client A writes message m_A into location ℓ, client B might later write message m_B into the same location ℓ. In this case, row ℓ will contain $m_A \oplus m_B$, and the contents of row ℓ will be unrecoverable.

To address this issue, we set the size of the database table to be large enough to accommodate the expected number of write requests for a given "success rate." For example, the servers can choose a table size that is large enough to accommodate 2^{10} write requests such that 95% of write requests will not be involved in a collision (in expectation). Under these parameters, 5% of the write requests will fail and those clients will have to resubmit their write requests in a future time epoch.

We can determine the appropriate table size by solving a simple "balls and bins" problem. If we throw m balls independently and uniformly at random into n bins, how many bins contain exactly one ball? Here, the m balls represent the write requests and the n bins represent the rows of the database.

Let B_{ij} be the probability that ball i falls into bin j. For all i and j, $\Pr[B_{ij}] = 1/n$. Let $O_i^{(1)}$ be the event that *exactly* one ball falls into bin i. Then

$$\Pr\left[O_i^{(1)}\right] = \frac{m}{n}\left(1-\frac{1}{n}\right)^{m-1}$$

Expanding using the binomial theorem and ignoring low order terms we obtain

$$\Pr\left[O_i^{(1)}\right] \approx \frac{m}{n} - \left(\frac{m}{n}\right)^2 + \frac{1}{2}\left(\frac{m}{n}\right)^3$$

where the approximation ignores terms of order $(m/n)^4$ and $o(1/n)$. Then $n \cdot \Pr[O_i^{(1)}]$ is the expected number of bins with exactly one ball which is the expected number of messages successfully received. Dividing this quantity by m gives the expected success rate so that:

$$\mathrm{E}[\mathrm{SuccessRate}] = \frac{n}{m}\Pr[O_i^{(1)}] \approx 1 - \frac{m}{n} + \frac{1}{2}\left(\frac{m}{n}\right)^2$$

So, if we want an expected success rate of 95% then we need $n \approx 19.5m$. For example, with $m = 2^{10}$ writers, we would use a table of size $n \approx 20,000$.

Handling collisions. We can shrink the table size n by coding the writes so that we can recover from collisions. We show how to handle two-way collisions. That is, when at most two clients write to the same location in the database. Let us assume that the messages being written to the database are elements in some field $\mathbb{F}$ of odd characteristic (say $\mathbb{F} = \mathbb{F}_p$ where $p = 2^{64} - 59$). We replace the XOR operation used in the basic scheme by addition in $\mathbb{F}$.

To recover from a two-way collision we will need to double the size of each cell in the database, but the overall number of cells n will shrink by more than a factor of two.

When a client A wants to write the message $m_A \in \mathbb{F}$ to location ℓ in the database the client will actually write the pair $(m_A, m_A^2) \in \mathbb{F}^2$ into that location. Clearly if no collision occurs at location ℓ then recovering m_A at the end of the epoch is trivial: simply drop the second coordinate (it is easy to test that no collision occurred because the second coordinate is a square of the first). Now, suppose a collision occurs with some client B who also added her own message $(m_B, m_B^2) \in \mathbb{F}^2$ to the same location ℓ (and no other client writes to location ℓ). Then at the end of the epoch the published values are

$$S_1 = m_A + m_B \pmod{p} \quad \text{and} \quad S_2 = m_A^2 + m_B^2 \pmod{p}$$

From these values it is quite easy to recover both m_A and m_B by observing that

$$2S_2 - S_1^2 = (m_A - m_B)^2 \pmod{p}$$

from which we obtain $m_A - m_B$ by taking a square root modulo p (it does not matter which of the two square roots we use—they both lead to the same result). Since $S_1 = m_A + m_B$ is also given it is now easy to recover both m_A and m_B.

Now that we can recover from two-way collisions we can shrink the number of cells n in the table. Let $O_i^{(2)}$ be the event that exactly two balls fell into bin i. Then the expected number of received messages is

$$n\Pr[O_i^{(1)}] + 2n\Pr[O_i^{(2)}] \tag{1}$$

where $\Pr[O_i^{(2)}] = \binom{m}{2}\frac{1}{n^2}\left(1-\frac{1}{n}\right)^{m-2}$. As before, dividing the expected number of received messages (1) by m, expanding using the binomial theorem, and ignoring low order terms gives the expected success rate as:

$$\mathrm{E}[\mathrm{SuccessRate}] \approx 1 - \frac{1}{2}\left(\frac{m}{n}\right)^2 + \frac{1}{3}\left(\frac{m}{n}\right)^3$$

So, if we want an expected success rate of 95% we need a table with $n \approx 2.7m$ cells. This is a far smaller table than before, when we could not handle collisions. In that case we needed $n \approx 19.5m$ which results in much bigger tables, despite each cell being half as big. Shrinking the table reduces the storage and computational burden on the servers.

This two-way collision handling technique generalizes to handle k-way collisions for $k > 2$. To handle k-way collisions, we increase the size of each cell by a factor of k and have each client i write $(m_i, m_i^2, \ldots, m_i^k) \in \mathbb{F}^k$ to its chosen cell. A k-collision gives k equations in k variables that can be efficiently solved to recover all k messages, as long as the characteristic of $\mathbb{F}$ is greater than k [43, 44]. Using $k > 2$ further reduces the table size as the desired success rate approaches one.

The collision handling method described in this section will also improve performance of our full system, which we describe in the next section.

Adversarial collisions. The analysis above assumes that clients behave honestly. Adversarial clients, however, need not write into random rows of the database—i.e., all m balls might not be thrown independently and uniformly at random. A coalition of clients might, for example, try to increase the probability of collisions by writing into the database using some malicious strategy.

By symmetry of writes we can assume that all $\hat{m}$ adversarial clients write to the database before the honest clients do. Now a message from an honest client is properly received at the end of an epoch if it avoids all the cells filled by the malicious clients. We can therefore carry out the honest client analysis above assuming the database contain $n - \hat{m}$ cells instead of n cells. In other words, given a bound $\hat{m}$ on the number of malicious clients we can calculate the required table size n. In practice, if too many collisions are detected at the end of an epoch the servers can adaptively double the size of the table so that the next epoch has fewer collisions.

3.3 Forward Security

Even the first-attempt scheme sketched in Section 3.1 provides *forward security* in the event that all of the servers' secret keys are compromised [45]. To be precise: an adversary could compromise the state and secret keys of *all servers* after the servers have processed n write requests from honest clients, but *before* the time epoch has ended. Even in this case, the adversary will be unable to determine which of the n clients submitted which of the n plaintext messages with a non-negligible advantage over random guessing. (We assume here that clients and servers communicate using encrypted channels which themselves have forward secrecy [46].)

This forward security property means that clients need not trust that $S - t$ servers stay honest forever—just that they are honest at the moment when the client submits its upload request. Being able to weaken the trust assumption about the servers in this way might be valuable in hostile environments, in which an adversary could compromise a server at any time without warning.

Mix-nets do not have this property, since servers must accumulate a set of onion-encrypted messages before shuffling and decrypting them [27]. If an adversary always controls the first mix server and if it can compromise the rest of the mix servers after accumulating a set of ciphertexts, the adversary can de-anonymize all of the system's users. DC-net-based systems that use "blame" protocols to retroactively discover disruptors have a similar weakness [16, 47].

The full Riposte protocol maintains this forward security property.

4 Improving Bandwidth Efficiency with Distributed Point Functions

This section describes how application of private information retrieval techniques can improve the bandwidth efficiency of the first-attempt protocol.

Notation. The symbol $\mathbb{F}$ denotes an arbitrary finite field, $\mathbb{Z}_L$ is the ring of integers modulo L. We use $e_\ell \in \mathbb{F}^L$ to represent a vector that is zero everywhere except at index $\ell \in \mathbb{Z}_L$, where it has value "1." Thus, for $m \in \mathbb{F}$, the vector $m \cdot e_\ell \in \mathbb{F}^L$ is the vector whose value is zero everywhere except at index ℓ, where it has value m. For a finite set S, the notation $x \xleftarrow{R} S$ indicates that the value of x is sampled independently and uniformly at random from S. The element $\mathbf{v}[i]$ is the value of a vector $\mathbf{v}$ at index i. We index vectors starting at zero.

4.1 Definitions

The bandwidth inefficiency of the protocol sketched above comes from the fact that the client must send an L-bit vector to each server to flip a single bit in the logical database. To reduce this $O(L)$ bandwidth overhead, we apply techniques inspired by private information retrieval protocols [22–24].

The problem of private information retrieval (PIR) is essentially the converse of the problem we are interested in here. In PIR, the client must *read* a bit from a replicated database without revealing to the servers the index being read. In our setting, the client must *write* a bit into a replicated database without revealing to the servers the index being written. Ostrovsky and Shoup first made this connection in the context of a "private information storage" protocol [25].

PIR schemes allow the client to split its query to the servers into shares such that (1) a subset of the shares does not leak information about the index of interest, and (2) the length of the query shares is much less than the length of the database. The core building block of many PIR schemes, which we adopt for our purposes, is a *distributed point function*. Although Gilboa and Ishai [23] defined distributed point functions as a primitive only recently, many prior PIR schemes make implicit use the primitive [22, 24]. Our definition of a distributed point function follows that of Gilboa and Ishai, except that we generalize the definition to allow for more than two servers.

First, we define a (non-distributed) point function.

Definition 4 (Point Function)**.** *Fix a positive integer L and a finite field $\mathbb{F}$. For all $\ell \in \mathbb{Z}_L$ and $m \in \mathbb{F}$, the* point function *$P_{\ell,m} : \mathbb{Z}_L \to \mathbb{F}$ is the function such that $P_{\ell,m}(\ell) = m$ and $P_{\ell,m}(\ell') = 0$ for all $\ell \neq \ell'$.*

That is, the point function $P_{\ell,m}$ has the value 0 when evaluated at any input not equal to ℓ and it has the value m when evaluated at ℓ. For example, if $L = 5$ and $\mathbb{F} = \mathbb{F}_2$, the point function $P_{3,1}$ takes on the values $(0,0,0,1,0)$ when evaluated on the values $(0,1,2,3,4)$ (note that we index vectors from zero).

An (s,t)-distributed point function provides a way to distribute a point function $P_{\ell,m}$ amongst s servers such that no coalition of at most t servers learns anything about ℓ or m given their t shares of the function.

Definition 5 (Distributed Point Function (DPF))**.** *Fix a positive integer L and a finite field $\mathbb{F}$. An (s,t)*-distributed point function *consists of a pair of possibly randomized algorithms that implement the following functionalities:*

- $\mathsf{Gen}(\ell, m) \to (k_0, \ldots, k_{s-1})$. *Given an integer $\ell \in \mathbb{Z}_L$ and value $m \in \mathbb{F}$, output a list of s keys.*
- $\mathsf{Eval}(k, \ell') \to m'$. *Given a key k generated using* Gen, *and an index $\ell' \in \mathbb{Z}_L$, return a value $m' \in \mathbb{F}$.*

We define correctness and privacy for a distributed point function as follows:

- **Correctness.** For a collection of s keys generated using $\mathsf{Gen}(\ell, m)$, the sum of the outputs of these keys (generated using Eval) must equal the point function $P_{\ell,m}$. More formally, for all $\ell, \ell' \in \mathbb{Z}_L$ and $m \in \mathbb{F}$:

$$\Pr[(k_0, \ldots, k_{s-1}) \leftarrow \mathsf{Gen}(\ell, m) : \\ \Sigma_{i=0}^{s-1} \mathsf{Eval}(k_i, \ell') = P_{\ell,m}(\ell')] = 1$$

  where the probability is taken over the randomness of the Gen algorithm.
- **Privacy.** Let S be any subset of $\{0, \ldots, s-1\}$ such that $|S| \leq t$. Then for any $\ell \in \mathbb{Z}_L$ and $m \in \mathbb{F}$, let $D_{S,\ell,m}$ denote the distribution of keys $\{(k_i) \mid i \in S\}$ induced by $(k_0, \ldots, k_{s-1}) \leftarrow \mathsf{Gen}(\ell, m)$. We say that an (s,t)-DPF maintains privacy if there exists a p.p.t. algorithm Sim such that the following distributions are computationally indistinguishable:

$$D_{S,\ell,m} \approx_c \mathsf{Sim}(S)$$

  That is, any subset of at most t keys leaks no information about ℓ or m. (We can also strengthen this definition to require statistical or perfect indistinguishability.)

Toy Construction. To make this definition concrete, we first construct a trivial information-theoretically secure $(s, s-1)$-distributed point function with length-L keys. As above, we fix a length L and a finite field $\mathbb{F}$.

- $\mathsf{Gen}(\ell, m) \to (k_0, \ldots, k_{s-1})$. Generate random vectors $k_0, \ldots, k_{s-2} \in \mathbb{F}^L$. Set $k_{s-1} = m \cdot e_\ell - \Sigma_{i=0}^{s-2} k_i$.
- $\mathsf{Eval}(k, \ell') \to m'$. Interpret k as a vector in $\mathbb{F}^L$. Return the value of the vector k at index ℓ'.

The correctness property of this construction follows immediately. Privacy is maintained because the distribution of any collection of $s-1$ keys is independent of ℓ and m.

This toy construction uses length-L keys to distribute a point function with domain $\mathbb{Z}_L$. Later in this section we describe DPF constructions which use much shorter keys.

4.2 Applying Distributed Point Functions for Bandwidth Efficiency

We can now use DPFs to improve the efficiency of the write-private database scheme introduced in Section 3.1. We show that the existence of an (s,t)-DPF with keys of length $|k|$ (along with standard cryptographic assumptions) implies the existence of write-private database scheme using s servers that maintains anonymity in the presence of t malicious servers, such that write requests have length $s|k|$. Any DPF construction with short keys thus immediately implies a bandwidth-efficient write-private database scheme.

The construction is a generalization of the one presented in Section 3.1. We now assume that there are s servers such that no more than t of them collude. Each of the s servers maintains a vector in $\mathbb{F}^L$ as their database state, for some fixed finite field $\mathbb{F}$ and integer L. Each "row" in the database is now an element of $\mathbb{F}$ and the database has L rows.

When the client wants to write a message $m \in \mathbb{F}$ into location $\ell \in \mathbb{Z}_L$ in the database, the client uses an (s,t)-distributed point function to generate a set of s DPF keys:

$$(k_0, \dots, k_{s-1}) \leftarrow \mathsf{Gen}(\ell, m)$$

The client then sends one of the keys to each of the servers. Each server i can then expand the key into a vector $v \in \mathbb{F}^L$ by computing $v(\ell') = \mathsf{Eval}(k_i, \ell')$ for $\ell' = 0, \dots, L-1$. The server then adds this vector v into its database state, using addition in $\mathbb{F}^L$. At the end of the time epoch, all servers combine their database states to reveal the set of client-submitted messages.

Correctness. The correctness of this construction follows directly from the correctness of the DPF. For each of the n write requests submitted by the clients, denote the j-th key in the i-th request as $k_{i,j}$, denote the write location as ℓ_i, and the message being written as m_i. When the servers combine their databases at the end of the epoch, the contents of the final database at row ℓ will be:

$$d_\ell = \sum_{i=0}^{n-1} \sum_{j=0}^{s-1} \mathsf{Eval}(k_{i,j}, \ell) = \sum_{i=0}^{n-1} P_{\ell_i, m_i}(\ell) \quad \in \mathbb{F}$$

In words: as desired, the combined database contains the sum of n point functions—one for each of the write requests.

Anonymity. The anonymity of this construction follows directly from the privacy property of the DPF. Given the plaintext database state d (as defined above), any coalition of t servers can simulate its view of the protocol. By definition of DPF privacy, there exists a simulator Sim, which simulates the distribution of any subset of t DPF keys generated using Gen. The coalition of servers can use this simulator to simulate each of the n write requests it sees during a run of the protocol. Thus, the servers can simulate their view of a protocol run and cannot win the anonymity game with non-negligible advantage.

Efficiency. A client in this scheme sends $|k|$ bits to each server (where k is a DPF key), so the bandwidth efficiency of the scheme depends on the efficiency of the DPF. As we will show later in this section, $|k|$ can be much smaller than the length of the database.

4.3 A Two-Server Scheme Tolerating One Malicious Server

Having established that DPFs with short keys lead to bandwidth-efficient write-private database schemes, we now present one such DPF construction. This construction is a simplification of computational PIR scheme of Chor and Gilboa [22].

This is a $(2,1)$-DPF with keys of length $O(\sqrt{L})$ operating on a domain of size L. This DPF yields a two-server write-private database scheme tolerating one malicious server such that writing into a database of size L requires sending $O(\sqrt{L})$ bits to each server. Gilboa and Ishai [23] construct a $(2,1)$-DPF with even shorter keys ($|k| = \mathsf{polylog}(L)$), but the construction presented here is efficient enough for the database sizes we use in practice. Although the DPF construction works over any field, we describe it here using the binary field $\mathbb{F} = \mathbb{F}_{2^k}$ (the field of k-bit bitstrings) to simplify the exposition.

When $\mathsf{Eval}(k, \ell')$ is run on every integer $\ell' \in \{0, \dots, L-1\}$, its output is a vector of L field elements. The DPF key construction conceptually works by representing this a vector of L field elements as an $x \times y$ matrix, such that $xy \geq L$. The trick that makes the construction work is that the size of the keys needs only to grow with the size of the *sides* of this matrix rather than its *area*. The DPF keys that $\mathsf{Gen}(\ell, m)$ outputs give an efficient way to construct two matrices M_A and M_B that differ only at one cell $\ell = (\ell_x, \ell_y) \in \mathbb{Z}_x \times \mathbb{Z}_y$ (Figure 2).

Fix a binary finite field $\mathbb{F} = \mathbb{F}_{2^k}$, a DPF domain size L, and integers x and y such that $xy \geq L$. (Later in this section, we describe how to choose x and y to minimize the key size.) The construction requires a pseudo-random generator (PRG) G that stretches seeds from some space $\mathbb{S}$ into length-y vectors of elements of $\mathbb{F}$ [48]. So the signature of the PRG is $G : \mathbb{S} \to \mathbb{F}^y$. In practice, an implementation might use AES-128 in counter mode as the pseudo-random generator [49].

The algorithms comprising the DPF are:

- $\mathsf{Gen}(\ell, m) \to (k_A, k_B)$. Compute integers $\ell_x \in \mathbb{Z}_x$ and $\ell_y \in \mathbb{Z}_y$ such that $\ell = \ell_x y + \ell_y$. Sample a random bit-vector $\mathbf{b}_A \xleftarrow{R} \{0,1\}^x$, a random vector of PRG seeds $\mathbf{s}_A \xleftarrow{R} \mathbb{S}^x$, and a single random PRG seed $s^*_{\ell_x} \xleftarrow{R} \mathbb{S}$.
  Given $\mathbf{b}_A$ and $\mathbf{s}_A$, we define $\mathbf{b}_B$ and $\mathbf{s}_B$ as:

$$\begin{aligned} \mathbf{b}_A &= (b_0, \dots, b_{\ell_x}, \dots, b_{x-1}) \\ \mathbf{b}_B &= (b_0, \dots, \bar{b}_{\ell_x}, \dots, b_{x-1}) \\ \mathbf{s}_A &= (s_0, \dots, s_{\ell_x}, \dots, s_{x-1}) \\ \mathbf{s}_B &= (s_0, \dots, s^*_{\ell_x}, \dots, s_{x-1}) \end{aligned}$$

  That is, the vectors $\mathbf{b}_A$ and $\mathbf{b}_B$ (similarly $\mathbf{s}_A$ and $\mathbf{s}_B$) differ only at index ℓ_x.
  Let $m \cdot e_{\ell_y}$ be the vector in $\mathbb{F}^y$ of all zeros except that it has value m at index ℓ_y. Define $\mathbf{v} \leftarrow m \cdot e_{\ell_y} + G(s_{\ell_x}) + G(s^*_{\ell_x})$.
  The output DPF keys are:

$$k_A = (\mathbf{b}_A, \mathbf{s}_A, \mathbf{v}) \qquad k_B = (\mathbf{b}_B, \mathbf{s}_B, \mathbf{v})$$

- $\mathsf{Eval}(k, \ell') \to m'$. Interpret k as a tuple $(\mathbf{b}, \mathbf{s}, \mathbf{v})$. To evaluate the PRF at index ℓ', first write ℓ' as an

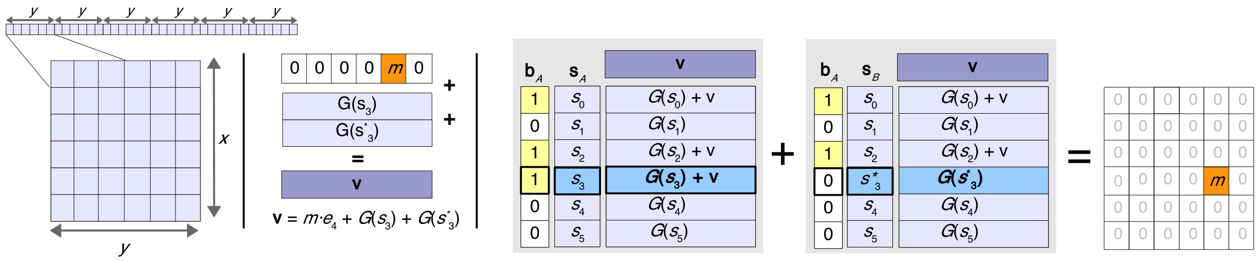


Figure 2: Left: We represent the output of Eval as an $x \times y$ matrix of field elements. Left-center: Construction of the $\mathbf{v}$ vector used in the DPF keys. Right: using the $\mathbf{v}$, $\mathbf{s}$, and $\mathbf{b}$ vectors, Eval expands each of the two keys into an $x \times y$ matrix of field elements. These two matrices sum to zero everywhere except at $(\ell_x, \ell_y) = (3,4)$, where they sum to m.

(ℓ'_x, ℓ'_y) tuple such that $\ell'_x \in \mathbb{Z}_x$, $\ell'_y \in \mathbb{Z}_y$, and $\ell' = \ell'_x y + \ell'_y$. Use the PRG G to stretch the ℓ'_x-th seed of $\mathbf{s}$ into a length-y vector: $\mathbf{g} \leftarrow G(\mathbf{s}[\ell'_x])$. Return $m' \leftarrow (\mathbf{g}[\ell'_y] + \mathbf{b}[\ell'_x]\mathbf{v}[\ell'_y])$.

Figure 2 graphically depicts how Eval stretches the keys into a table of $x \times y$ field elements.

Correctness. We prove correctness of the scheme in Appendix B.

Privacy. The privacy property requires that there exists an efficient simulator that, on input "A" or "B," outputs samples from a distribution that is computationally indistinguishable from the distribution of DPF keys k_A or k_B.

The simulator Sim simulates each component of the DPF key as follows: It samples $\mathbf{b} \xleftarrow{R} \{0,1\}^x$, $\mathbf{s} \xleftarrow{R} \mathbb{S}^x$, and $\mathbf{v} \xleftarrow{R} \mathbb{F}^y$. The simulator returns $(\mathbf{b}, \mathbf{s}, \mathbf{v})$.

We must now argue that the simulator's output distribution is computationally indistinguishable from that induced by the distribution of a single output of Gen. Since the $\mathbf{b}$ and $\mathbf{s}$ vectors outputted by Gen are random, the simulation is perfect. The $\mathbf{v}$ vector outputted by Gen is computationally indistinguishable from random, since it is padded with the output of the PRG seeded with a seed unknown to the holder of the key. An efficient algorithm to distinguish the simulated $\mathbf{v}$ vector from random can then also distinguish the PRG output from random.

Key Size. A key for this DPF scheme consists of: a vector in $\{0,1\}^x$, a vector in $\mathbb{S}^x$, and a vector in $\mathbb{F}^y$. Let α be the number of bits required to represent an element of $\mathbb{S}$ and let β be the number of bits required to represent an element of $\mathbb{F}$. The total length of a key is then:

$$|k| = (1+\alpha)x + \beta y$$

For fixed spaces $\mathbb{S}$ and $\mathbb{F}$, we can find the optimal choices of x and y to minimize the key length. To do so, we solve:

$$\min_{x,y}((1+\alpha)x + \beta y) \quad \text{subject to} \quad xy \geq L$$

and conclude that the optimal values of x and y are:

$$x = c\sqrt{L} \quad \text{and} \quad y = \frac{1}{c}\sqrt{L} \quad \text{where} \quad c = \sqrt{\frac{\beta}{1+\alpha}}.$$

The key size is then $O(\sqrt{L})$.

When using a database table of one million rows in length ($L = 2^{20}$), a row length of 1 KB per row ($\mathbb{F} = \mathbb{F}_{2^{8192}}$), and a PRG seed size of 128 bits (using AES-128, for example) the keys will be roughly 263 KB in length. For these parameters, the keys for the naïve construction (Section 3.1) would be 1 GB in length. Application of efficient DPFs thus yields a 4,000× bandwidth savings in this case.

Computational Efficiency. A second benefit of this scheme is that both the Gen and Eval routines are computationally efficient, since they just require performing finite field additions (i.e., XOR for binary fields) and PRG operations (i.e., computations of the AES function). The construction requires no public-key primitives.

4.4 An s-Server Scheme Tolerating $s-1$ Malicious Servers

The $(2,1)$-DPF scheme described above achieved a key size of $O(\sqrt{L})$ bits using only symmetric-key primitives. The limitation of that construction is that it only maintains privacy when a single key is compromised. In the context of a write-private database scheme, this means that the construction can only maintain anonymity in the presence of a *single* malicious server. It would be much better to have a write-private database scheme with s servers that maintains anonymity in the presence of $s-1$ malicious servers. To achieve this stronger security notion, we need a bandwidth-efficient $(s, s-1)$-distributed point function.

In this section, we construct an $(s, s-1)$-DPF where each key has size $O(\sqrt{L})$. We do so at the cost of requiring more expensive public-key cryptographic operations, instead of the symmetric-key operations used in the prior DPF. While the $(2,1)$-DPF construction above directly follows the work of Chor and Gilboa [22], this $(s,s-1)$-DPF construction is novel, as far as we know. In recent work, Boyle et al. present a $(s,s-1)$-DPF construction using only symmetric-key operations, but this construction exhibits a key size *exponential* in the number of servers s [50].

This construction uses a *seed-homomorphic pseudo-random generator* [51–53], to split the key for the pseudo-random generator G across a collection of s DPF keys.

Definition 6 (Seed-Homomorphic PRG)**.** *A* seed-homomorphic *PRG is a pseudo-random generator G mapping seeds in a group $(\mathbb{S},\oplus)$ to outputs in a group $(\mathbb{G},\otimes)$ with the additional property that for any $s_0,s_1 \in \mathbb{S}$:*

$$G(s_0 \oplus s_1) = G(s_0) \otimes G(s_1)$$

It is possible to construct a simple seed-homomorphic PRG from the decision Diffie-Hellman (DDH) assumption [52,53]. The public parameters for the scheme are list of y generators chosen at random from an order-q group $\mathbb{G}$, in which the DDH problem is hard [54]. For example, if $\mathbb{G}$ is an elliptic curve group [55], then the public parameters will be y points $(P_0,\ldots,P_{y-1}) \in \mathbb{G}^y$. The seed space is $\mathbb{Z}_q$ and the generator outputs vectors in $\mathbb{G}^y$. On input $s \in \mathbb{Z}_q$, the generator outputs $(sP_0,\ldots,sP_{y-1})$. The generator is seed-homomorphic because, for any $s_0,s_1 \in \mathbb{Z}_q$, and for all $i \in \{1,\ldots,y\}$: $s_0P_i + s_1P_i = (s_0+s_1)P_i$.

As in the prior DPF construction, we fix a DPF domain size L, and integers x and y such that $xy \geq L$. The construction requires a seed-homomorphic PRG $G : \mathbb{S} \mapsto \mathbb{G}^y$, for some group $\mathbb{G}$ of prime order q.

For consistency with the prior DPF construction, we will write the group operation in $\mathbb{G}$ using additive notation. Thus, the group operation applied component-wise to vectors $\mathbf{u},\mathbf{v} \in \mathbb{G}^y$ results in the vector $(\mathbf{u}+\mathbf{v}) \in \mathbb{G}^y$. Since $\mathbb{G}$ has order q, $qA = 0$ for all $A \in \mathbb{G}$.

The algorithms comprising the $(s,s-1)$-DPF are:

- $\mathsf{Gen}(\ell,m) \to (k_0,\ldots,k_{s-1})$. Compute integers $\ell_x \in \mathbb{Z}_x$ and $\ell_y \in \mathbb{Z}_y$ such that $\ell = \ell_x y + \ell_y$. Sample random integer-valued vectors $\mathbf{b}_0,\ldots,\mathbf{b}_{s-2} \xleftarrow{R} (\mathbb{Z}_q)^x$, random vectors of PRG seeds $\mathbf{s}_0,\ldots,\mathbf{s}_{s-2} \xleftarrow{R} \mathbb{S}^x$, and a single random PRG seed $s^* \xleftarrow{R} \mathbb{S}$.
  Select $\mathbf{b}_{s-1} \in (\mathbb{Z}_q)^x$ such that $\Sigma_{k=0}^{s-1}\mathbf{b}_k = e_{\ell_x} \pmod q$ and select $\mathbf{s}_{s-1} \in \mathbb{S}^x$ such that $\Sigma_{k=0}^{s-1}\mathbf{s}_k = s^* \cdot e_{\ell_x} \in \mathbb{G}^x$. Define $\mathbf{v} \leftarrow m \cdot e_{\ell_y} - G(s^*)$.
  The DPF key for server $i \in \{0,\ldots,s-1\}$ is $k_i = (\mathbf{b}_i,\mathbf{s}_i,\mathbf{v})$.
- $\mathsf{Eval}(k,\ell') \to m'$. Interpret k as a tuple $(\mathbf{b},\mathbf{s},\mathbf{v})$. To evaluate the PRF at index ℓ', first write ℓ' as an (ℓ'_x,ℓ'_y) tuple such that $\ell'_x \in \mathbb{Z}_x$, $\ell'_y \in \mathbb{Z}_y$, and $\ell' = \ell'_x y + \ell'_y$. Use the PRG G to stretch the ℓ'_x-th seed of $\mathbf{s}$ into a length-y vector: $\mathbf{g} \leftarrow G(\mathbf{s}[\ell'_x])$. Return $m' \leftarrow (\mathbf{g}[\ell'_y] + \mathbf{b}[\ell'_x]\mathbf{v}[\ell'_y])$.

We omit correctness and privacy proofs, since they follow exactly the same structure as those used to prove security of our prior DPF construction. The only difference is that correctness here relies on the fact that G is a seed-homomorphic PRG, rather than a conventional PRG. As in the DPF construction of Section 4.3, the keys here are of length $O(\sqrt{L})$.

Computational Efficiency. The main computational cost of this DPF construction comes from the use of the seed-homomorphic PRG G. *Unlike* a conventional PRG, which can be implemented using AES or another fast block cipher in counter mode, known constructions of seed-homomorphic PRGs require algebraic groups [53] or lattice-based cryptography [51,52].

When instantiating the $(s,s-1)$-DPF with the DDH-based PRG construction in elliptic curve groups, each call to the DPF Eval routine requires an expensive elliptic curve scalar multiplication. Since elliptic curve operations are, per byte, orders of magnitude slower than AES operations, this $(s,s-1)$-DPF will be orders of magnitude slower than the $(2,1)$-DPF. Security against an arbitrary number of malicious servers comes at the cost of computational efficiency, at least for these DPF constructions.

With DPFs, we can now construct a bandwidth-efficient write-private database scheme that tolerates one malicious server (first construction) or $s-1$ out of s malicious servers (second construction).

5 Preventing Disruptors

The first-attempt construction of our write-private database scheme (Section 3.1) had two limitations: (1) client write requests were very large and (2) malicious clients could corrupt the database state by sending malformed write requests. We addressed the first of these two challenges in Section 4. In this section, we address the second challenge.

A client write request in our protocol just consists of a collection of s DPF keys. The client sends one key to each of the s servers. The servers must collectively decide whether the collection of s keys is a valid output of the DPF Gen routine, without revealing any information about the keys themselves.

One way to view the servers' task here is as a secure multi-party computation [20,21]. Each server i's private input is its DPF key k_i. The output of the protocol is a single bit, which determines if the s keys $(k_0, \ldots, k_{s-1})$ are a well-formed collection of DPF keys.

Since we already rely on servers for availability (Section 2.2), we *need not* protect against servers maliciously trying to manipulate the output of the multi-party protocol. Such manipulation could only result in corrupting the database (if a malicious server accepts a write request that it should have rejected) or denying service to an honest client (if a malicious server rejects a write request that it should have accepted). Since both attacks are tantamount to denial of service, we need not consider them.

We *do* care, in contrast, about protecting client privacy against malicious servers. A malicious server participating in the protocol should not gain any additional information about the private inputs of other parties, no matter how it deviates from the protocol specification.

We construct two protocols for checking the validity of client write requests. The first protocol is computationally inexpensive, but requires introducing a third non-colluding party to the two-server scheme. The second protocol requires relatively expensive zero-knowledge proofs [56–59], but it maintains security when all but one of s servers is malicious. Both of these protocols must satisfy the standard notions of soundness, completeness, and zero-knowledge [60].

Since the publication of this paper, Boyle et al. have designed a very efficient protocol for checking the well-formedness of DPF keys [61]. Their checking protocol provides security against semi-honest (i.e., honest but curious) servers and requires only a *constant* amount of server-to-server communication. If it is possible to extend their checking protocol to provide security against fully malicious servers, their new scheme could serve as a more efficient alternative to the protocols described herein.

5.1 Three-Server Protocol

Our first protocol for detecting malformed write requests works with the $(2,1)$-DPF scheme presented in Section 4.3. The protocol uses only hashing and finite field additions, so it is computationally inexpensive. The downside is that it requires introducing a third *audit* server, which must not collude with either of the other two servers. This simple protocol draws inspiration from classical secure multi-party computation protocols [19–21].

As a warm-up to the full checking protocol, we develop a three-server protocol called AlmostEqual that we use as a subroutine to implement the full write-request validation protocol. The AlmostEqual protocol takes place between the client and three servers: database server A, database server B, and an audit server.

Let λ be a security parameter (e.g., $\lambda = 256$). The protocol uses a hash function $H : \{0,1\}^* \to \{0,1\}^\lambda$, which we model as a random oracle [62].

At the start of the protocol, database server A holds a vector $\mathbf{v}_A \in \mathbb{F}^n$, and database server B holds a vector $\mathbf{v}_B \in \mathbb{F}^n$. The audit server takes no private input. The client holds both $\mathbf{v}_A$ and $\mathbf{v}_B$.

The three servers execute a protocol that allows them to confirm that the vectors $\mathbf{v}_A$ and $\mathbf{v}_B$ are equal everywhere except at exactly one index, and such that any one malicious server learns nothing about the index at which these vectors differ, as long as the vectors indeed differ at a single index.

More formally, the servers want to execute this check in such a way that the following properties hold:

- **Completeness.** If all parties are honest, and $\mathbf{v}_A$ and $\mathbf{v}_B$ are well formed, then the database servers almost always accept the vectors as well formed.
- **Soundness.** If $\mathbf{v}_A$ and $\mathbf{v}_B$ do not differ at a single index, and all three servers are honest, then the servers will reject the vectors almost always.
- **Zero knowledge.** If $\mathbf{v}_A$ and $\mathbf{v}_B$ are well formed and the client is honest, then any one actively malicious server can simulate its view of the protocol execution. Furthermore, the simulator does not take as input the index at which $\mathbf{v}_A$ and $\mathbf{v}_B$ differ nor the value of the vectors at that index.

We denote an instance of the three-server protocol as $\mathsf{AlmostEqual}(\mathbf{v}_A, \mathbf{v}_B)$, where the arguments denote the private values that the two database servers take as input. The protocol proceeds as follows:

1. The client sends a PRG seed $\sigma \in \{0,1\}^\lambda$ to both database servers.

2. Servers A and B use a PRG seeded with the seed σ to sample n pseudorandom values $(r_0, \ldots, r_{n-1}) \in \{0,1\}^{\lambda n}$. The servers also use the seed σ to agree upon a pseudorandom "shift" value $f \in \mathbb{Z}_n$.

3. Server A computes the values $m_i \leftarrow H(\mathbf{v}_A[i], r_i)$ for every index $i \in \{0, \ldots, n-1\}$ and sends to the auditor
$$\mathbf{m}_A = (m_f, m_{f+1}, \ldots, m_{n-1}, m_0, \ldots, m_{f-1}).$$
The vector $\mathbf{m}_A$ is a blinded version of server A's input vector $\mathbf{v}_A$. Using a secret random value $\rho \in \{0,1\}^\lambda$ shared with server B (constructed using a coin-flipping protocol [63], for example), server A computes a check value $c_A \leftarrow \sigma \oplus \rho$ and sends c_A to the auditor.

4. Server B repeats Step 2 with $\mathbf{v}_B$.

5. Since the client knows $\mathbf{v}_A$, $\mathbf{v}_B$, and σ, it can compute $\mathbf{m}_A$ and $\mathbf{m}_B$ on its own. The client computes digests $d_A = H(\mathbf{m}_A)$ and $d_B = H(\mathbf{m}_B)$, and sends these digests to the audit server.

6. The audit server returns "1" to servers A and B if and only if:

   - the vectors it receives from the two servers are equal at every index except one,
   - the values c_A and c_B are equal, and
   - the vectors $\mathbf{m}_A$ and $\mathbf{m}_B$ satisfy $d_A = H(\mathbf{m}_A)$ and $d_B = H(\mathbf{m}_B)$, where d_A and d_B are the client-provided digests.

   The auditor returns "0" otherwise.

We include proofs of soundness, correctness, and zero-knowledge for this construction in Appendix C.

The keys for the $(2,1)$-DPF construction have the form

$$k_A = (\mathbf{b}_A, \mathbf{s}_A, \mathbf{v}) \qquad k_B = (\mathbf{b}_B, \mathbf{s}_B, \mathbf{v}).$$

In a correctly formed pair of keys, the $\mathbf{b}$ and $\mathbf{s}$ vectors differ at a single index ℓ_x, and the $\mathbf{v}$ vector is equal to $\mathbf{v} = m \cdot e_{\ell_y} + G(\mathbf{s}_A[\ell_x]) + G(\mathbf{s}_B[\ell_x])$.

To determine whether a pair of keys is correct, server A constructs a test vector $\mathbf{t}_A$ such that $\mathbf{t}_A[i] = \mathbf{b}_A[i] \| \mathbf{s}_A[i]$ for $i \in \{0, \ldots, x-1\}$. (where $\|$ denotes concatenation). Server B constructs a test vector $\mathbf{t}_B$ in the same way and the two servers, along with the client and the auditor, run the protocol $\mathsf{AlmostEqual}(\mathbf{t}_A, \mathbf{t}_B)$. If the output of this protocol is "1," then the servers conclude that their $\mathbf{b}$ and $\mathbf{s}$ vectors differ at a single index, though the protocol does not reveal to the servers which index this is. Otherwise, the servers reject the write request.

Next, the servers must verify that the $\mathbf{v}$ vector is well-formed. To do so, the servers compute another pair of test vectors:

$$\mathbf{u}_A = \sum_{i=0}^{x-1} G(\mathbf{s}_A[i]) \qquad \mathbf{u}_B = \mathbf{v} + \sum_{i=0}^{x-1} G(\mathbf{s}_B[i]).$$

The client and servers run $\mathsf{AlmostEqual}(\mathbf{u}_A, \mathbf{u}_B)$ and accept the write request as valid if it returns "1."

We prove security of this construction in Appendix D.

An important implementation note is that if $m = 0$—that is, if the client writes the string of all zeros into the database—then the $\mathbf{u}$ vectors will not differ at any index and this information is leaked to the auditor. The protocol only provides security if the vectors differ at *exactly one* index. To avoid this information leakage, client requests must be defined such that $m \neq 0$ in every write request. To achieve this, clients could define some special non-zero value to indicate "zero" or could use a padding scheme to ensure that zero values occur with negligible probability.

As a practical matter, the audit server needs to be able to match up the portions of write requests coming from server A with those coming from server B. Riposte achieves this as follows: When the client sends its upload request to server A, the client includes a cryptographic hash of the request it sent to server B (and vice versa). Both servers can use these hashes to derive a common nonce for the request. When the servers send audit requests to the audit server, they include the nonce for the write request in question. The audit server can use the nonce to match every audit request from server A with the corresponding request from server B.

This three-party protocol is very efficient—it only requires $O(\sqrt{L})$ applications of a hash function and $O(\sqrt{L})$ communication from the servers to the auditor. The auditor only performs a simple string comparison, so it needs minimal computational and storage capabilities.

5.2 Zero Knowledge Techniques

Our second technique for detecting disruptors makes use of non-interactive zero-knowledge proofs [58, 59, 64].

We apply zero-knowledge techniques to allow clients to prove the well-formedness of their write requests. This technique works in combination with the $(s, s-1)$-DPF presented in Section 4.4 and maintains client write-privacy when *all but one* of s servers is dishonest.

The keys for the $(s, s-1)$-DPF scheme are tuples $(\mathbf{b}_i, \mathbf{s}_i, \mathbf{v})$ such that:

$$\sum_{i=0}^{s-1} \mathbf{b}_i = e_{\ell_x} \qquad \sum_{i=0}^{s-1} \mathbf{s}_i = s^* \cdot e_{\ell_x} \qquad \mathbf{v} = m \cdot e_{\ell_y} - G(s^*)$$

To prove that its write request was correctly formed, we have the client perform zero-knowledge proofs over collections of Pedersen commitments [65]. The public parameters for the Pedersen commitment scheme consist of a group $\mathbb{G}$ of prime order q and two generators P and Q of $\mathbb{G}$ such that no one knows the discrete logarithm $\log_Q P$. A Pedersen commitment to a message $m \in \mathbb{Z}_q$ with randomness $r \in \mathbb{Z}_q$ is $C(m,r) = (mP + rQ) \in \mathbb{G}$ (writing the group operation additively). Pedersen commitments are *homomorphic*, in that given commitments to m_0 and m_1, it is possible to compute a commitment to $m_0 + m_1$:

$$C(m_0, r_0) + C(m_1, r_1) = C(m_0 + m_1, r_0 + r_1)$$

Here, we assume that the $(s, s-1)$-DPF is instantiated with the DDH-based PRG introduced in Section 4.4 and that the group $\mathbb{G}$ used for the Pedersen commitments is the same order-q group used in the PRG construction.

To execute the proof, the client first generates Pedersen commitments to elements of each of the s DPF keys. Then each server i can verify that the client computed the commitment to the i-th DPF key elements correctly. The servers use the homomorphic property of Pedersen commitments to generate commitments to the *sum* of the elements of the DPF keys. Finally, the client proves in zero knowledge that these sums have the correct values.

The protocols proceed as follows:

1. The client generates vectors of Pedersen commitments $\mathbf{B}_i$ and $\mathbf{S}_i$ committing to each element of $\mathbf{b}_i$ and $\mathbf{s}_i$. The client sends the $\mathbf{B}$ and $\mathbf{S}$ vectors to *every server*.

2. To server i, the client sends the opening of the commitments $\mathbf{B}_i$ and $\mathbf{S}_i$. Each server i verifies that $\mathbf{B}_i$ and $\mathbf{S}_i$ are valid commitments to the $\mathbf{b}_i$ and $\mathbf{s}_i$ vectors in the DPF key. If this check fails at some server i, server i notifies the other servers and all servers reject the write request.

3. Using the homomorphic property of the commitments, each server can compute vectors of commitments $\mathbf{B}_{\text{sum}}$ and $\mathbf{S}_{\text{sum}}$ to the vectors $\Sigma_{i=0}^{s-1}\mathbf{b}_i$ and $\Sigma_{i=0}^{s-1}\mathbf{s}_i$.

4. Using a non-interactive zero-knowledge proof, the client proves to the servers that $\mathbf{B}_{\text{sum}}$ and $\mathbf{S}_{\text{sum}}$ are commitments to zero everywhere except at a single (secret) index ℓ_x, and that $\mathbf{B}_{\text{sum}}[\ell_x]$ is a commitment to one.[1] This proof uses standard witness hiding techniques for discrete-logarithm-based zero knowledge proofs [64, 66]. If the proof is valid, the servers continue to check the $\mathbf{v}$ vector.

This first protocol convinces each server that the $\mathbf{b}$ and $\mathbf{s}$ components of the DPF keys are well formed. Next, the servers check the $\mathbf{v}$ component:

1. For each server i, the client sums up the seed values $\mathbf{s}_i$ it sent to server i: $\sigma_i = \Sigma_{j=0}^{s-1}\mathbf{s}_i[j]$. The client then generates the output of $G(\sigma_k)$ and blinds it:

$$\mathbf{G}_i = (\sigma_i P_1 + r_1 Q, \sigma_i P_2 + r_2 Q, \ldots).$$

2. The client sends the $\mathbf{G}$ values to all servers and the client sends the opening of $\mathbf{G}_i$ to each server i.

3. Each server verifies that the openings are correct, and all servers reject the write request if this check fails at any server.

4. Using the homomorphic property of Pedersen commitments, every server can compute a vector of commitments $\mathbf{G}_{\text{sum}} = (\Sigma_{i=0}^{s-1}\mathbf{G}_i) + \mathbf{v}$. If $\mathbf{v}$ is well formed, then the $\mathbf{G}_{\text{sum}}$ vector contain commitments to zero at every index except one (at which it will contain a commitment to the client's message m).

5. The client uses a non-interactive zero-knowledge proof to convince the servers that the vector of commitments $\mathbf{G}_{\text{sum}}$ contains commitments to zero at all indexes except one. If the proof is valid, the servers accept the write request.

We prove in Appendix E that this protocol satisfies the standard notions of soundness, completeness, and zero-knowledge [60].

6 Experimental Evaluation

To demonstrate that Riposte is a practical platform for traffic-analysis-resistant anonymous messaging, we implemented two variants of the system. The first variant uses the two-server distributed point function (Section 4.3) and uses the three-party protocol (Section 5.1) to prevent malicious clients from corrupting the database. This variant is relatively fast, since it relies primarily on symmetric-key primitives, but requires that no two of the three servers collude. Our results for the first variant *include the cost* of identifying and excluding malicious clients.

The second variant uses the s-server distributed point function (Section 4.4). This variant protects against $s-1$ colluding servers, but relies on expensive public-key operations. We have not implemented the zero-knowledge proofs necessary to prevent disruptors for the s-server protocol (Section 5.2), so the performance numbers represent only an upper bound on the system throughput.

We wrote the prototype in the Go programming language and have published the source code online at `https://bitbucket.org/henrycg/riposte/`. We used the DeterLab network testbed for our experiments [67]. All of the experiments used commodity servers running Ubuntu 14.04 with four-core AES-NI-enabled Intel E3-1260L CPUs and 16 GB of RAM.

Our experimental network topology used between two and ten servers (depending on the protocol variant in use) and eight client nodes. In each of these experiments, the eight client machines used many threads of execution to

[1]Technically, this is a zero-knowledge proof of knowledge which proves that the client *knows* an opening of the commitments to the stated values.

submit write requests to the servers as quickly as possible. In all experiments, the server nodes connected to a common switch via 100 Mbps links, the clients nodes connected to a common switch via 1 Gbps links, and the client and server switches connected via a 1 Gbps link. The round-trip network latency between each pair of nodes was 20 ms. We chose this network topology to limit the bandwidth between the servers to that of a fast WAN, but to leave client bandwidth unlimited so that the small number of client machines could saturate the servers with write requests.

Error bars in the charts indicate the standard deviation of the throughput measurements.

6.1 Three-Server Protocol

A three-server Riposte cluster consists of two database servers and one audit server. The system maintains its security properties as long as *no two* of these three servers collude. We have fully implemented the three-server protocol, including the audit protocol (Section 5.1), so the throughput numbers listed here *include* the cost of detecting and rejecting malicious write requests.

The prototype used AES-128 in counter mode as the pseudo-random generator, Poly1305 as the keyed hash function used in the audit protocol [68], and TLS for link encryption.

Figure 3 shows how many client write requests our Riposte cluster can service per second as the number of 160-byte rows in the database table grows. For a database table of 64 rows, the system handles 751.5 write requests per second. At a table size of 65,536 rows, the system handles 32.8 requests per second. At a table size of 1,048,576 rows, the system handles 2.86 requests per second.

We chose the row length of 160 bytes because it was the smallest multiple of 32 bytes large enough to to contain a 140-byte Tweet. Throughput of the system depends only the total size of the table (number of rows $\times$ row length), so larger row lengths might be preferable for other applications. For example, an anonymous email system using Riposte with 4096-byte rows could handle 2.86 requests per second at a table size of 40,960 rows.

An upper bound on the performance of the system is the speed of the pseudo-random generator used to stretch out the DPF keys to the length of the database table. The dashed line in Figure 3 indicates this upper bound (605 MB/s), as determined using an AES benchmark written in Go. That line indicates the maximum possible throughput we could hope to achieve without aggressive optimization (e.g., writing portions of the code in assembly) or more powerful machines. Migrating the performance-critical portions of our implementation from Go to C (using OpenSSL) might increase the throughput by a factor of as much as 6$\times$, since `openssl speed` reports AES throughput of 3.9 GB/s, compared with the 605 MB/s we obtain with Go's crypto library. At very small table sizes, the speed at which the server can set up TLS connections with the clients limits the overall throughput to roughly 900 requests per second.

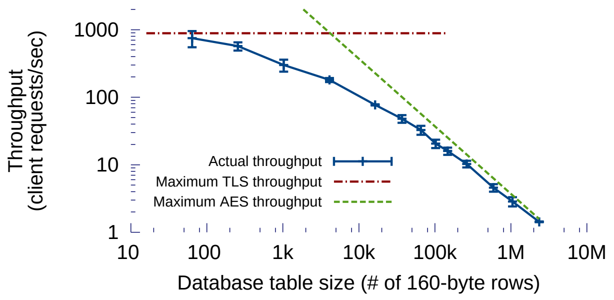


Figure 3: As the database table size grows, the throughput of our system approaches the maximum possible given the AES throughput of our servers.

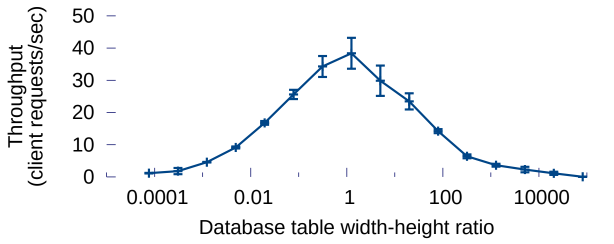


Figure 4: Use of bandwidth-efficient DPFs gives a 768$\times$ speed-up over the naïve constructions, in which a client's request is as large as the database.

Figure 4 demonstrates how the request throughput varies as the width of the table changes, while the number of bytes in the table is held constant at 10 MB. This figure demonstrates the performance advantage of using a bandwidth-efficient $O(\sqrt{L})$ DPF (Section 4) over the naïve DPF (Section 3.1). Using a DPF with optimal table size yields a throughput of 38.4 requests per second. The extreme left and right ends of the figure indicate the performance yielded by the naïve construction, in which making a write request involves sending a $(1 \times L)$-dimension vector to each server. At the far right extreme of the table, performance drops to 0.05 requests per second, so DPFs yield a 768$\times$ speed-up.

Figure 5 indicates the total number of bytes transferred by one of the database servers and by the audit server while processing a single client write request. The dashed

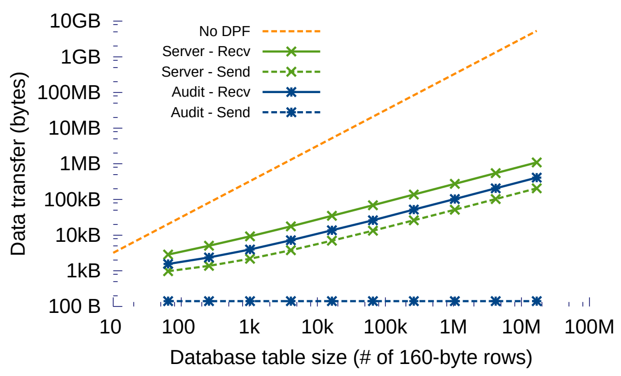


Figure 5: The total client and server data transfer scales sub-linearly with the size of the database.

line at the top of the chart indicates the number of bytes a client would need to send for a single write request if we did not use bandwidth-efficient DPFs (i.e., the dashed line indicates the size of the database table). As the figure demonstrates, the total data transfer in a Riposte cluster scales *sub-linearly* with the database size. When the database table is 2.5 GB in size, the database server transfers only a total of 1.23 MB to process a write request.

6.2 *s*-Server Protocol

In some deployment scenarios, having strong protection against server compromise may be more important than performance or scalability. In these cases, the *s*-server Riposte protocol provides the same basic functionality as the three-server protocol described above, except that it maintains privacy even if $s-1$ out of s servers collude or deviate arbitrarily from the protocol specification. We implemented the basic *s*-server protocol but have not yet implemented the zero-knowledge proofs necessary to prevent malicious clients from corrupting the database state (Section 5.2). These performance figures thus represent an *upper bound* on the *s*-server protocol's performance. Adding the zero-knowledge proofs would require an additional $\Theta(\sqrt{L})$ elliptic curve operations per server in an L-row database. The computational cost of the proofs would almost certainly be dwarfed by the $\Theta(L)$ elliptic curve operations required to update the state of the database table.

The experiments use the DDH-based seed-homomorphic pseudo-random generator described in Section 4.4 and they use the NIST P-256 elliptic curve as the underlying algebraic group. The table row size is fixed at 160 bytes.

Figure 6 demonstrates the performance of an eight-server Riposte cluster as the table size increases. At a table size of 1,024 rows, the cluster can process one re-

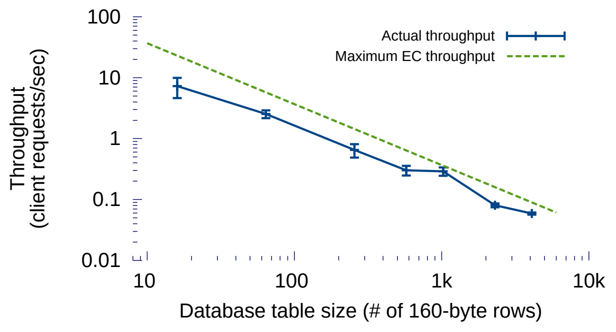


Figure 6: Throughput of an eight-server Riposte cluster using the $(8,7)$-distributed point function.

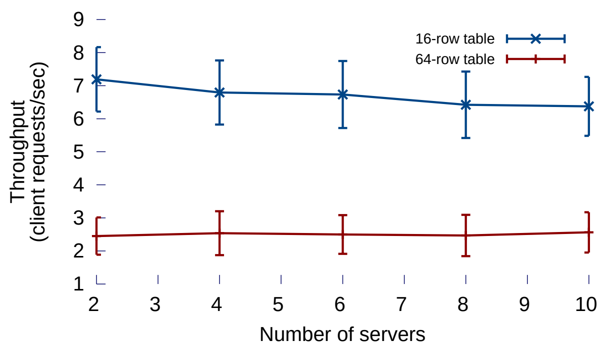


Figure 7: Throughput of Riposte clusters using two different database table sizes as the number of servers varies.

quest every 3.44 seconds. The limiting factor is the rate at which the servers can evaluate the DDH-based pseudo-random generator (PRG), since computing each 32-byte block of PRG output requires a costly elliptic curve scalar multiplication. The dashed line in the figure indicates the maximum throughput obtainable using Go's implementation of P-256 on our servers, which in turn dictates the maximum cluster throughput. Processing a single request with a table size of one million rows would take nearly one hour with this construction, compared to 0.3 seconds in the AES-based three-server protocol.

Figure 7 shows how the throughput of the Riposte cluster changes as the number of servers varies. Since the workload is heavily CPU-bound, the throughput only decreases slightly as the number of servers increases from two to ten.

6.3 Discussion: Whistleblowing and Microblogging with Million-User Anonymity Sets

Whistleblowers, political activists, or others discussing sensitive or controversial issues might benefit from an anonymous microblogging service. A whistleblower, for example, might want to anonymously blog about an instance of bureaucratic corruption in her organization. The utility of such a system depends on the size of the anonymity set it would provide: if a whistleblower is only anonymous amongst a group of ten people, it would be easy for the whistleblower's employer to retaliate against *everyone* in the anonymity set. Mounting this "punish-them-all" attack does not require breaking the anonymity system itself, since the anonymity set is public. As the anonymity set size grows, however, the feasibility of the "punish-them-all" attack quickly tends to zero. At an anonymity set size of 1,000,000 clients, mounting an "punish-them-all" attack would be prohibitively expensive in most situations.

Riposte can handle such large anonymity sets as long as (1) clients are willing to tolerate hours of messaging latency, and (2) only a small fraction of clients writes into the database in each time epoch. Both of these requirements are satisfied in the whistleblowing scenario. First, whistleblowers might not care if the system delays their posts by a few hours. Second, the vast majority of users of a microblogging service (especially in the whistleblowing context) are more likely to *read* posts than write them. To get very large anonymity sets, maintainers of an anonymous microblogging service could take advantage of the large set of "read-only" users to provide anonymity for the relatively small number of "read-write" users.

The client application for such a microblogging service would enable read-write users to generate and submit Riposte write requests to a Riposte cluster running the microblogging service. However, the client application would also allow read-only users to submit an "empty" write request to the Riposte cluster that would always write a random message into the first row of the Riposte database. From the perspective of the servers, a read-only client would be indistinguishable from a read-write client. By leveraging read-only users in this way, we can increase the size of the anonymity set without needing to increase the size of the database table.

To demonstrate that Riposte can support very large anonymity set sizes—albeit with high latency—we configured a cluster of Riposte servers with a 65,536-row database table and left it running for 32 hours. In that period, the system processed a total of 2,895,216 write requests at an average rate of 25.19 requests per second. (To our knowledge, this is the largest anonymity set *ever constructed* in a system that offers protection against traffic analysis attacks.) Using the techniques in Section 3.2, a table of this size could handle 0.3% of users writing at a collision rate of under 5%. Thus, to get an anonymity set of roughly 1,000,000 users with a three-server Riposte cluster and a database table of size $65,536$, the time epoch must be at least 11 hours long.

As of 2013, Twitter reported an average throughput of 5,700 140-byte Tweets per second [69]. That is equivalent roughly 5,000 of our 160-byte messages per second. At a table size of one million messages, our Riposte cluster's end-to-end throughput is 2.86 write requests per second (Figure 3). To handle the same volume of Tweets as Twitter does with anonymity set sizes on the order of hundreds of thousands of clients, we would need to increase the computing power of our cluster by "only" a factor of $\approx$1,750.[2] Since we are using only three servers now, we would need roughly 5,250 servers (split into three non-colluding data centers) to handle the same volume of traffic as Twitter. Furthermore, since the audit server is just doing string comparisons, the system would likely need many fewer audit servers than database servers, so the total number of servers required might be closer to $4,000$.

7 Related Work

Anonymity systems fall into one of two general categories: systems that provide low-latency communication and those that protect against traffic analysis attacks by a global network adversary.

Aqua [8], Crowds [10], LAP [26], ShadowWalker [9], Tarzan [7], and Tor [6] belong to the first category of systems: they provide an anonymous proxy for real-time Web browsing, but they do not protect against an adversary who controls the network, many of the clients, and some of the nodes on a victim's path through the network. Even providing a formal definition of anonymity for low-latency systems is challenging [70] and such definitions typically do not capture the need to protect against timing attacks.

Even so, it would be possible to combine Tor (or another low-latency anonymizing proxy) and Riposte to build a "best of both" anonymity system: clients would submit their write requests to the Riposte servers via the Tor network. In this configuration, even if *all* of the Riposte servers colluded, they could not learn which user wrote which message without also breaking the anonymity of the Tor network.

David Chaum's "cascade" mix networks were one of the first systems devised with the specific goal of defending against traffic-analysis attacks [27]. Since then, there

[2]We assume here that scaling the number of machines by a factor of k increases our throughput by a factor of k. This assumption is reasonable given our workload, since the processing of write requests is an embarrassingly parallel task.

have been a number of mix-net-style systems proposed, many of which explicitly weaken their protections against a near omni-present adversary [71] to improve prospects for practical usability (i.e., for email traffic) [72]. In contrast, Riposte attempts to provide very strong anonymity guarantees at the price of usability for interactive applications.

E-voting systems (also called "verifiable shuffles") achieve the sort of privacy properties that Riposte offers, and some systems even provide stronger voting-specific guarantees (receipt-freeness, proportionality, etc.), though most e-voting systems cannot provide the forward security property that Riposte offers (Section 3.3) [30–32, 73–76].

In a typical e-voting system, voters submit their encrypted ballots to a few trustees, who collectively shuffle and decrypt them. While it is possible to repurpose e-voting systems for anonymous messaging, they typically require expensive zero-knowledge proofs or are inefficient when message sizes are large. Mix-nets that do not use zero-knowledge proofs of correctness typically do not provide privacy in the face of active attacks by a subset of the mix servers.

For example, the verifiable shuffle protocol of Bayer and Groth [29] is one of the most efficient in the literature. Their shuffle implementation, when used with an anonymity set of size N, requires $16N$ group exponentiations per server and data transfer $O(N)$. In addition, messages must be small enough to be encoded in single group elements (a few hundred bytes at most). In contrast, our protocol requires $O(L)$ AES operations and data transfer $O(\sqrt{L})$, where L is the size of the database table. When messages are short and when the writer/reader ratio is high, the Bayer-Groth mix may be faster than our system. In contrast, when messages are long and when the writer/reader ratio is low (i.e., $L \ll O(N)$), our system is faster.

Chaum's Dining Cryptographers network (DC-net) is an information-theoretically secure anonymous broadcast channel [14]. A DC-net provides the same strong anonymity properties as Riposte does, but it requires every user of a DC-net to participate in every run of the protocol. As the number of users grows, this quickly becomes impractical.

The Dissent [16] system introduced the idea of using partially trusted servers to make DC-nets practical in distributed networks. Dissent requires weaker trust assumptions than our three-server protocol does but it requires clients to send $O(L)$ bits to each server per time epoch (compared with our $O(\sqrt{L})$). Also, excluding *a single* disruptor in a 1,000-client deployment takes over an hour. In contrast, Riposte can excludes disruptors as fast as it processes write requests (tens to hundreds per second, depending on the database size). Recent work [33] uses zero-knowledge techniques to speed up disruption resistance in Dissent (building on ideas of Golle and Juels [34]). Unfortunately, these techniques limit the system's end to end-throughput end-to-end throughput to 30 KB/s, compared with Riposte's 450+ MB/s.

Herbivore scales DC-nets by dividing users into many small anonymity sets [15]. Riposte creates a single large anonymity set, and thus enables every client to be anonymous amongst the *entire set* of honest clients.

Our DPF constructions make extensive use of prior work on private information retrieval (PIR) [22–24, 77]. Recent work demonstrates that it is possible to make theoretical PIR fast enough for practical use [78–80]. Function secret sharing [50] generalizes DPFs to allow sharing of more sophisticated functions (rather than just point functions). This more powerful primitive may prove useful for PIR and anonymous messaging applications in the future.

Gertner et al. [81] consider *symmetric* PIR protocols, in which the servers prevent dishonest clients from learning about more than a single row of the database per query. The problem that Gertner et al. consider is, in a way, the dual of the problem we address in Section 5, though their techniques do not appear to apply directly in our setting.

Ostrovsky and Shoup first proposed using PIR protocol as the basis for writing into a database shared across a set of servers [25]. However, Ostrovsky and Shoup considered only the case of a single honest client, who uses the untrusted database servers for private storage. Since *many mutually distrustful clients* use a single Riposte cluster, our protocol must also handle malicious clients.

Pynchon Gate [82] builds a private point-to-point messaging system from mix-nets and PIR. Clients anonymously upload messages to email servers using a traditional mix-net and download messages from the email servers using a PIR protocol. Riposte could replace the mix-nets used in the Pynchon Gate system: clients could anonymously write their messages into the database using Riposte and could privately read incoming messages using PIR.

8 Conclusion and Open Questions

We have presented Riposte, a new system for anonymous messaging. To the best of our knowledge, Riposte is the first system that simultaneously (1) thwarts traffic analysis attacks, (2) prevents malicious clients from anonymously disrupting the system, and (3) enables million-client anonymity set sizes. We achieve these goals through novel application of private information retrieval and secure multiparty computation techniques. We have demonstrated Riposte's practicality by implementing it and evaluating it with anonymity sets of over two million nodes. This work leaves open a number of questions for future work, including:

- Does there exist an $(s, s-1)$-DPF construction for $s > 2$ that uses only symmetric-key operations?
- Are there efficient techniques (i.e., using no public-key primitives) for achieving disruption resistance without the need for a non-colluding audit server?
- Are there DPF constructions that enable processing write requests in amortized time $o(L)$, for a length-L database?

With the design and implementation of Riposte, we have demonstrated that cryptographic techniques can make traffic-analysis-resistant anonymous microblogging and whistleblowing more practical at Internet scale.

Acknowledgements

We thank Elette Boyle and Yuval Ishai for pointing out an error in the AlmostEqual protocol of Section 5.1, and for helpful discussions on how to repair it. We would like to thank Joe Zimmerman and David Wu for helpful discussions about distributed point functions. We would like to thank Stephen Schwab and the staff of DeterLab for giving us access to their excellent network testbed. This work was supported by NSF, an IARPA project provided via DoI/NBC, a grant from ONR, an NDSEG fellowship, and by a Google faculty scholarship. Opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or IARPA.

References

[1] K. Bennhold, "In Britain, guidelines for spying on lawyers and clients," *New York Times*, p. A6, 7 Nov. 2014.

[2] B. Gellman and A. Soltani, "NSA infiltrates links to Yahoo, Google data centers worldwide, Snowden documents say," *Washington Post*, Oct. 30 2013.

[3] B. Gellman, J. Tate, and A. Soltani, "In NSA-intercepted data, those not targeted far outnumber the foreigners who are," *Washington Post*, 5 Jul. 2014.

[4] V. Goel, "Government push for Yahoo's user data set stage for broad surveillance," *New York Times*, p. B3, 7 Sept. 2014.

[5] E. Nakashima and B. Gellman, "Court gave NSA broad leeway in surveillance, documents show," *Washington Post*, 30 Jun. 2014.

[6] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *USENIX Security Symposium*, Aug. 2004.

[7] M. J. Freedman and R. Morris, "Tarzan: A peer-to-peer anonymizing network layer," in *CCS*. ACM, 2002.

[8] S. Le Blond, D. Choffnes, W. Zhou, P. Druschel, H. Ballani, and P. Francis, "Towards efficient traffic-analysis resistant anonymity networks," in *SIGCOMM*. ACM, 2013.

[9] P. Mittal and N. Borisov, "ShadowWalker: Peer-to-peer anonymous communication using redundant structured topologies," in *CCS*. ACM, November 2009.

[10] M. K. Reiter and A. D. Rubin, "Crowds: Anonymity for Web transactions," *ACM Transactions on Information and System Security*, vol. 1, no. 1, pp. 66–92, 1998.

[11] K. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. Sicker, "Low-resource routing attacks against Tor," in *WPES*. ACM, 2007.

[12] S. J. Murdoch and G. Danezis, "Low-cost traffic analysis of Tor," in *Security and Privacy*. IEEE, 2005.

[13] S. J. Murdoch and P. Zieliński, "Sampled traffic analysis by Internet-exchange-level adversaries," in *PETS*, June 2007.

[14] D. Chaum, "The Dining Cryptographers problem: Unconditional sender and recipient untraceability," *Journal of Cryptology*, pp. 65–75, Jan. 1988.

[15] S. Goel, M. Robson, M. Polte, and E. Sirer, "Herbivore: A scalable and efficient protocol for anonymous communication," Cornell University, Tech. Rep., 2003.

[16] D. I. Wolinsky, H. Corrigan-Gibbs, A. Johnson, and B. Ford, "Dissent in numbers: Making strong anonymity scale," in *10th OSDI*. USENIX, Oct. 2012.

[17] G. Danezis and C. Diaz, "A survey of anonymous communication channels," Technical Report MSR-TR-2008-35, Microsoft Research, Tech. Rep., 2008.

[18] M. Edman and B. Yener, "On anonymity in an electronic society: A survey of anonymous communication systems," *ACM Computing Surveys*, vol. 42, no. 1, p. 5, 2009.

[19] R. Fagin, M. Naor, and P. Winkler, "Comparing information without leaking it," *Communications of the ACM*, vol. 39, no. 5, pp. 77–85, 1996.

[20] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game," in *STOC*. ACM, 1987.

[21] A. C. Yao, "Protocols for secure computations," in *FOCS*. IEEE, 1982.

[22] B. Chor and N. Gilboa, "Computationally private information retrieval," in *STOC*. ACM, 1997.

[23] N. Gilboa and Y. Ishai, "Distributed point functions and their applications," in *EUROCRYPT*, 2014.

[24] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *Journal of the ACM*, vol. 45, no. 6, pp. 965–981, 1998.

[25] R. Ostrovsky and V. Shoup, "Private information storage," in *STOC*, 1997.

[26] H.-C. Hsiao, T.-J. Kim, A. Perrig, A. Yamada, S. C. Nelson, M. Gruteser, and W. Meng, "LAP: Lightweight anonymity and privacy," in *Security and Privacy*. IEEE, May 2012.

[27] D. L. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Communications of the ACM*, vol. 24, no. 2, pp. 84–90, 1981.

[28] B. Adida and D. Wikström, "How to shuffle in public," in *Theory of Cryptography*, 2007.

[29] S. Bayer and J. Groth, "Efficient zero-knowledge argument for correctness of a shuffle," in *EUROCRYPT*, 2012.

[30] J. Furukawa, "Efficient, verifiable shuffle decryption and its requirement of unlinkability," in *PKC*, 2004.

[31] J. Groth, "A verifiable secret shuffle of homomorphic encryptions," *Journal of Cryptology*, vol. 23, no. 4, pp. 546–579, 2010.

[32] C. A. Neff, "A verifiable secret shuffle and its application to e-voting," in *CCS*. ACM, 2001.

[33] H. Corrigan-Gibbs, D. I. Wolinsky, and B. Ford, "Proactively accountable anonymous messaging in Verdict," in *USENIX Security Symposium*, 2013.

[34] P. Golle and A. Juels, "Dining cryptographers revisited," in *EUROCRYPT*, 2004.

[35] A. Serjantov, R. Dingledine, and P. Syverson, "From a trickle to a flood: Active attacks on several mix types," in *Information Hiding*, 2003.

[36] B. Liskov and J. Cowling, "Viewstamped replication revisited," MIT CSAIL, Tech. Rep. MIT-CSAIL-TR-2012-021, Jul. 2013.

[37] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *ATC*. USENIX, Jun. 2014.

[38] M. Waidner and B. Pfitzmann, "The Dining Cryptographers in the disco: Unconditional sender and recipient untraceability with computationally secure serviceability," in *EUROCRYPT*, Apr. 1989.

[39] G. Danezis and A. Serjantov, "Statistical disclosure or intersection attacks on anonymity systems," in *Information Hiding Workshop*, May 2004.

[40] D. Kedogan, D. Agrawal, and S. Penz, "Limits of anonymity in open environments," in *Information Hiding*, 2003.

[41] N. Mathewson and R. Dingledine, "Practical traffic analysis: Extending and resisting statistical disclosure," in *Privacy Enhancing Technologies*, 2005.

[42] D. Wolinsky, E. Syta, and B. Ford, "Hang with your buddies to resist intersection attacks," in *CCS*, November 2013.

[43] J. Bos and B. den Boer, "Detection of disrupters in the DC protocol," in *EUROCRYPT*, 1989.

[44] R. T. Chien and W. Frazer, "An application of coding theory to document retrieval," *Information Theory, IEEE Transactions on*, vol. 12, no. 2, pp. 92–96, 1966.

[45] R. Canetti, S. Halevi, and J. Katz, "A forward-secure public-key encryption scheme," in *EUROCRYPT*, 2003.

[46] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and K. T, "RFC7296: Internet key exchange protocol version 2 (IKEv2)," Oct. 2014.

[47] H. Corrigan-Gibbs and B. Ford, "Dissent: Accountable anonymous group messaging," in *CCS*. ACM, October 2010.

[48] J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby, "A pseudorandom generator from any one-way function," *SIAM Journal on Computing*, vol. 28, no. 4, pp. 1364–1396, 1999.

[49] National Institute of Standards and Technology, "Specification for the advanced encryption standard (AES)," Federal Information Processing Standards Publication 197, Nov. 2001.

[50] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing," in *EUROCRYPT*, 2015, pp. 337–367.

[51] A. Banerjee and C. Peikert, "New and improved key-homomorphic pseudorandom functions," in *CRYPTO*, 2014.

[52] D. Boneh, K. Lewi, H. Montgomery, and A. Raghunathan, "Key homomorphic PRFs and their applications," in *CRYPTO*, 2013.

[53] M. Naor, B. Pinkas, and O. Reingold, "Distributed pseudo-random functions and KDCs," in *EUROCRYPT*, 1999.

[54] D. Boneh, "The decision Diffie-Hellman problem," in *Algorithmic Number Theory*, ser. Lecture Notes in Computer Science, J. P. Buhler, Ed. Springer, 1998, vol. 1423, pp. 48–63.

[55] V. S. Miller, "Use of elliptic curves in cryptography," in *CRYPTO*, 1986.

[56] U. Feige, A. Fiat, and A. Shamir, "Zero-knowledge proofs of identity," *Journal of Cryptology*, vol. 1, no. 2, pp. 77–94, 1988.

[57] O. Goldreich, S. Micali, and A. Wigderson, "Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems," *Journal of the ACM*, vol. 38, no. 3, pp. 690–728, 1991.

[58] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM Journal on computing*, vol. 18, no. 1, pp. 186–208, 1989.

[59] C. Rackoff and D. R. Simon, "Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack," in *CRYPTO*, 1992.

[60] J. L. Camenisch, "Group signature schemes and payment systems based on the discrete logarithm problem," Ph.D. dissertation, Swiss Federal Institute of Technology Zürich (ETH Zürich), 1998.

[61] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing: Improvements and extensions," in *CCS*. ACM, 2016, pp. 1292–1303.

[62] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *CCS*. ACM, 1993.

[63] M. Blum, "Coin flipping by telephone: a protocol for solving impossible problems," *ACM SIGACT News*, vol. 15, no. 1, pp. 23–27, 1983.

[64] J. Camenisch and M. Stadler, "Proof systems for general statements about discrete logarithms," Dept. of Computer Science, ETH Zurich, Tech. Rep. 260, Mar. 1997.

[65] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *CRYPTO*, 1992.

[66] R. Cramer, I. Damgård, and B. Schoenmakers, "Proofs of partial knowledge and simplified design of witness hiding protocols," in *CRYPTO*, 1994.

[67] J. Mirkovic and T. Benzel, "Teaching cybersecurity with DeterLab," *Security & Privacy*, vol. 10, no. 1, 2012.

[68] D. J. Bernstein, "The Poly1305-AES message-authentication code," in *Fast Software Encryption*, 2005.

[69] R. Krikorian, "New Tweets per second record, and how!" https://blog.twitter.com/2013/new-tweets-per-second-record-and-how, Aug. 2013.

[70] A. Johnson, "Design and analysis of efficient anonymous-communication protocols," Ph.D. dissertation, Yale University, Dec. 2009.

[71] P. Syverson, "Why I'm not an entropist," in *Security Protocols XVII*, 2013.

[72] G. Danezis, R. Dingledine, and N. Mathewson, "Mixminion: Design of a type III anonymous remailer protocol," in *Security and Privacy*. IEEE, 2003.

[73] B. Adida, "Helios: Web-based open-audit voting." in *USENIX Security Symposium*, vol. 17, 2008.

[74] M. R. Clarkson, S. Chong, and A. C. Myers, “Civitas: A secure voting system,” Cornell University, Tech. Rep. TR 2007-2081, May 2007.

[75] J. Groth and S. Lu, “Verifiable shuffle of large size ciphertexts,” in *PKC*, 2007.

[76] M. O. Rabin and R. L. Rivest, “Efficient end to end verifiable electronic voting employing split value representations,” in *EVOTE 2014*, Aug. 2014.

[77] W. Gasarch, “A survey on private information retrieval,” in *Bulletin of the EATCS*, 2004.

[78] C. Devet and I. Goldberg, “The best of both worlds: Combining information-theoretic and computational pir for communication efficiency,” in *PETS*, July 2014.

[79] I. Goldberg, “Improving the robustness of private information retrieval,” in *Security and Privacy*. IEEE, 2007.

[80] D. Demmler, A. Herzberg, and T. Schneider, “RAID-PIR: Practical multi-server PIR,” in *WPES*, 2014.

[81] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin, “Protecting data privacy in private information retrieval schemes,” in *STOC*, 1998.

[82] L. Sassaman, B. Cohen, and N. Mathewson, “The Pynchon gate: A secure method of pseudonymous mail retrieval,” in *WPES*, November 2005.

A Definition of Write Privacy

An (s,t)-*write-private database scheme* consists of the following three (possibly randomized) algorithms:

$\mathsf{Write}(\ell, m) \to (w^{(0)}, \ldots, w^{(s-1)})$. Clients use the Write functionality to generate the write request queries sent to the s servers. The Write function takes as input a message m (from some finite message space) and an integer ℓ and produces a set of s write requests—one per server.

$\mathsf{Update}(\sigma, w) \to \sigma'$. Servers use the Update functionality to process incoming write requests. The Update function takes as input a server's internal state σ, a write request w, and outputs the updated state of the server σ'.

$\mathsf{Reveal}(\sigma_0, \ldots, \sigma_{s-1}) \to D$. At the end of the time epoch, servers use the Reveal functionality to recover the contents of the database. The Reveal function takes as input the set of states from each of the s servers and produces the plaintext database contents D.

We define the write-privacy property using the following security game, played between the adversary (who statically corrupts up to t servers and all but two clients) and a challenger.

1. In the first step, the adversary performs the following actions:

   - The adversary selects a subset $\mathcal{A}_s \subseteq \{0, \ldots, s-1\}$ of the servers, such that $|\mathcal{A}_s| \leq t$. The set $\mathcal{A}_s$ represents the set of adversarial servers. Let the set $\mathcal{H}_s = \{0, \ldots, s-1\} \setminus \mathcal{A}_s$ represent the set of honest servers.
   - The adversary selects a set of clients $\mathcal{H}_c \subseteq \{0, \ldots, n-1\}$, such that $|\mathcal{H}_c| \geq 2$, representing the set of honest clients. The adversary selects one message-location pair per honest client:
   $$\mathcal{M} = \{(i, m_i, \ell_i) \mid i \in \mathcal{H}_c\}$$

   The adversary sends $\mathcal{A}_s$ and $\mathcal{M}$ to the challenger.

2. In the second step, the challenger responds to the adversary:

   - For each $(i, m_i, \ell_i) \in \mathcal{M}$, the challenger generates a write request:
   $$(w_i^{(0)}, \ldots, w_i^{(s-1)}) \leftarrow \mathsf{Write}(\ell_i, m_i)$$
   The set of shares of the ith write request revealed to the malicious servers is $W_i = \{w_i^{(j)}\}_{j \in \mathcal{A}_S}$.
   In the next steps of the game, the challenger will randomly reorder the honest clients' write requests. The challenger should learn nothing about which client wrote what, despite all the information at its disposal.
   - The challenger then samples a random permutation π over $\{0, \ldots, |\mathcal{H}_c| - 1\}$. The challenger sends the following set of write requests to the adversary, permuted according to π:
   $$\langle W_{\pi(0)}, W_{\pi(1)}, \ldots, W_{\pi(|\mathcal{H}_c|-1)} \rangle$$

3. For each client i in $\{0, \ldots, n-1\} \setminus \mathcal{H}_c$, the adversary computes a write request $(w_i^{(0)}, \ldots, w_i^{(s-1)})$ (possibly according to some malicious strategy) and sends the set of these write requests to the challenger.

4.
   - For each server $j \in \mathcal{H}_s$, the challenger computes the server's final state σ_j by running the Update functionality on each of the n client write requests in order. Let $S = \{(j, \sigma_j) \mid j \in \mathcal{H}_s\}$ be the set of states of the honest servers.
   - The challenger samples a bit $b \xleftarrow{R} \{0,1\}$. If $b = 0$, the challenger send (S, π) to the adversary. Otherwise, the challenger samples a fresh permutation π^* on $\mathcal{H}_c$ and sets (S, π^*) to the adversary.

5. The adversary makes a guess b' for the value of b.

The adversary wins the game if $b = b'$. We define the adversary's advantage as $|\Pr[b = b'] - 1/2|$. The scheme maintains (s,t)-write privacy if no efficient adversary wins the game with non-negligible advantage (in the implicit security parameter).

B Correctness Proof for $(2,1)$-DPF

This appendix proves correctness of the distributed point construction of Section 4.3. For the scheme to be correct, it must be that, for $(k_A, k_B) \leftarrow \mathsf{Gen}(\ell, m)$, for all $\ell' \in \mathbb{Z}_L$:

$$\mathsf{Eval}(k_A, \ell') + \mathsf{Eval}(k_B, \ell') = P_{\ell,m}(\ell').$$

Let (ℓ_x, ℓ_y) be the tuple in $\mathbb{Z}_x \times \mathbb{Z}_y$ representing location ℓ and let (ℓ'_x, ℓ'_y) be the tuple representing ℓ'. Let:

$$m'_A \leftarrow \mathsf{Eval}(k_A, \ell') \qquad m'_B \leftarrow \mathsf{Eval}(k_B, \ell').$$

We use a case analysis to show that the left-hand side of the equation above equals $P_{\ell,m}$ for all ℓ':

Case I: $\ell_x \neq \ell'_x$. When $\ell_x \neq \ell'_x$, the seeds $\mathbf{s}_A[\ell'_x]$ and $\mathbf{s}_B[\ell'_x]$ are equal, so $\mathbf{g}_A = \mathbf{g}_B$. Similarly $\mathbf{b}_A[\ell'_x] = \mathbf{b}_B[\ell'_x]$. The output m'_A will be $\mathbf{g}_A[\ell'_y] + \mathbf{b}_A[\ell'_x]\mathbf{v}[\ell'_y]$, The output m'_B will be identical to m'_A. Since the field is a binary field, adding a value to itself results in the zero element, so the sum $m'_A + m'_B$ will be zero as desired.

Case II: $\ell_x = \ell'_x$ and $\ell_y \neq \ell'_y$. When $\ell_x = \ell'_x$, the seeds $\mathbf{s}_A[\ell'_x]$ and $\mathbf{s}_B[\ell'_x]$ are *not* equal, so $\mathbf{g}_A \neq \mathbf{g}_B$. Similarly $\mathbf{b}_A[\ell'_x] \neq \mathbf{b}_B[\ell'_x]$. When $\ell_y \neq \ell'_y$, $\mathbf{v}[\ell'_y] = \mathbf{g}_A[\ell'_y] + \mathbf{g}_B[\ell'_y]$. Assume $\mathbf{b}_A[\ell'_x] = 0$ (an analogous argument applies when $\mathbf{b}_A[\ell'_x] = 1$), then:

$$\mathbf{v}[\ell'_y] = (m \cdot e_{\ell_y})[\ell'_y] + \mathbf{g}_A[\ell'_y] + \mathbf{g}_B[\ell'_y].$$

The sum $m'_A + m'_B$ will then be:

$$m'_A + m'_B = \mathbf{g}_A[\ell'_y] + \mathbf{g}_B[\ell'_y] + \mathbf{v}[\ell'_y] = 0.$$

Case III: $\ell_x = \ell'_x$ and $\ell_y = \ell'_y$. This is the same as Case II, except that $(m \cdot e_{\ell_y})[\ell'_y] = m$ when $\ell_y = \ell'_y$, so the sum $m'_A + m'_B = m$, as desired.

C Proof Sketches for the AlmostEqual Protocol

This appendix proves security of the AlmostEqual protocol of Section 5.1.

Completeness. We must show that if the vectors $\mathbf{v}_A$ and $\mathbf{v}_B$ differ in exactly one position, the audit server will output "1" with overwhelming probability.

The audit server checks three things: (1) that $\mathbf{m}_A$ and $\mathbf{m}_B$ differ at a single index, (2) that the check values c_A and c_B are equal, and (3) that the digests d_A and d_B match the digests of the vectors the database servers sent to the audit server. This second and third tests will always pass if all parties are honest.

The first test fails with some small probability: since the audit server only outputs "1" if *exactly* one element of the test vectors is equal, if there is a collision in the hash function, the protocol may return an incorrect result. The probability of this event is negligible in the security parameter λ.

Soundness. To demonstrate soundness, it is sufficient to show that it is infeasible for a cheating client to produce values $(\hat{\mathbf{v}}_A, \hat{\mathbf{v}}_B, \hat{\sigma}_A, \hat{\sigma}_B, \hat{d}_A, \hat{d}_B)$ such that $\hat{\mathbf{v}}_A$ and $\hat{\mathbf{v}}_B$ do *not* differ at exactly one index, and yet the three honest servers accept these vectors as almost equal.

Define $(\hat{\mathbf{m}}_A, \hat{c}_A)$ and $(\hat{\mathbf{m}}_B, \hat{c}_B)$ to be the values that the honest database servers send to the audit server the database servers take the values $(\hat{\mathbf{v}}_A, \hat{\sigma}_A)$ and $(\hat{\mathbf{v}}_B, \hat{\sigma}_B)$ as input. the malicious client.

First, note that for the cheating client to succeed, it must be that $\hat{d}_A = H(\hat{\mathbf{m}}_A)$ or $\hat{d}_B = H(\hat{\mathbf{m}}_B)$. So the choice of $\hat{d}_A$ and $\hat{d}_B$ are fixed by the client's choice of the other values.

Next, observe that if a cheating client submits values $\hat{\sigma}_A \neq \hat{\sigma}_B$ to the servers, then $\hat{c}_A \neq \hat{c}_B$ with probability 1. This holds because we require soundness to hold only when all three servers are honest, so both database servers will generate c_A and c_B using a common blinding value ρ. So any successfully cheating client must submit a request with $\hat{c}_A = \hat{c}_B$.

To cause the servers to accept, the client must then find values $(\hat{\mathbf{v}}_A, \hat{\sigma}_A)$ and $(\hat{\mathbf{v}}_B, \hat{\sigma}_B)$ such that, $\sigma = \hat{\sigma}_A = \hat{\sigma}_B$ and, for all but one $i \in \{0, \ldots, n-1\}$, $\hat{\mathbf{m}}_A[i] = \hat{\mathbf{m}}_B[i]$. If $\hat{\mathbf{v}}_A = \hat{\mathbf{v}}_B$ everywhere, then the probability of this event is zero: the vectors $\hat{\mathbf{m}}_A$ and $\hat{\mathbf{m}}_B$ will always be the same and the servers will always reject.

Consider instead that $\hat{\mathbf{v}}_A$ and $\hat{\mathbf{v}}_B$ differ at two or more indices and yet the servers accept the vectors. In this case, $\hat{\mathbf{v}}_A$ and $\hat{\mathbf{v}}_B$ differ at least at indices i_1 and i_2 and yet $\hat{\mathbf{m}}_A$ and $\hat{\mathbf{m}}_B$ differ at only one index. In this case, at some

$i^* \in \{i_1, i_2\}$, we have that

$$\hat{\mathbf{v}}_A[i^*] \neq \hat{\mathbf{v}}_B[i^*] \quad \text{and} \quad H(\hat{\mathbf{v}}_A[i^*], \hat{r}_{i^*}) = H(\hat{\mathbf{v}}_B[i^*], \hat{r}_{i^*}).$$

If the client can find such vectors $\hat{\mathbf{v}}_A$ and $\hat{\mathbf{v}}_B$, then the client can find a collision in H. Any adversary that violates the soundness property of the protocol with non-negligible probability can then find collisions in H with non-negligible probability, which is infeasible.

Zero Knowledge. To show the zero-knowledge property we must show that (1) each database server can simulate its view of the protocol run and (2) the audit server can simulate its view as well. Each simulator takes as input all public parameters of the system plus each server's private input. Each simulator must produce faithful simulations whenever the other parties are honest.

Showing that the audit server can simulate its view of the protocol run is straightforward: the audit server receives values $(\mathbf{m}_A, c_A, d_A)$ and $(\mathbf{m}_B, c_B, d_B)$ from the database servers and honest client.

Whenever the vectors differ at exactly one position the audit server can simulate its view of the protocol. To do so, the simulator picks length-n vectors $\mathbf{m}_A$ and $\mathbf{m}_B$ of random elements in the range of the hash function H subject to the constraint that the vectors are equal everywhere except at a random index $i' \in \mathbb{Z}_n$. The simulator outputs the two vectors as the vectors received from servers A and B. The simulator computes the digests $d_A = H(\mathbf{m}_A)$ and $d_B = H(\mathbf{m}_B)$. The simulator sets $c_A = c_B \xleftarrow{R} \{0,1\}^\lambda$.

The simulation of $\mathbf{m}_A$ and $\mathbf{m}_B$ is perfect, since these values are independently random values, as long as all of the values $(r_0, \dots, r_{n-1})$ generated from the seed σ are distinct. Since the servers sample each r_i from $\{0,1\}^\lambda$, the probability of a collision is negligible. The digests d_A and d_B are constructed exactly as in the real protocol run. The values c_A and c_B that the auditor sees in the real protocol are equal values masked by a random λ-bit string. The simulation of c_A and c_B is then perfect.

We now must show that each database server can simulate its view as well. Since the role of the two database servers is symmetric, we need only produce a simulator for server A.

In the real protocol run, the database server interacts with the honest client and honest audit server as follows:

1. The honest client sends σ to the database server.
2. The database server produces a (possibly malformed) test vector $\hat{\mathbf{m}}_A$ and check value $\hat{c}_A$, and sends these values to the audit server.
3. The audit server returns a bit $\hat{\beta}$ to the database server.

We must produce a simulator $S = (S_1, S_2)$ that simulates the first and third flows of this protocol. The simulator takes all of the public parameters of the system as implicit input. The simulator also takes the vector $\mathbf{v}_A$ and the shared random value ρ as input. The simulation proceeds as follows:

- $(\sigma, \mathsf{state}) \leftarrow S_1(\mathbf{v}_A, \rho)$. Simulator S_1 produces a random PRG seed $\sigma \in \{0,1\}^\lambda$. The simulator computes $\mathbf{m}_A$ using $\mathbf{v}_A$ and σ, as in the real protocol. The simulator computes $c_A \leftarrow \rho \oplus \sigma$ and $d_A \leftarrow H(\mathbf{m}_A)$. The simulator outputs $\mathbf{v}_A$ as the output of the first flow, and outputs the state as: $\mathsf{state} = (d_A, c_A)$.
- $\hat{\beta} \leftarrow S_2(\mathsf{state} = (d_A, c_A), \hat{\mathbf{m}}_A, \hat{c}_A)$. The simulator computes $\hat{d_A} = H(\hat{\mathbf{m}}_A)$ and outputs $\hat{\beta} = 1$ if

  $$d_A = \hat{d_A} \quad \text{and} \quad \hat{c}_A = c_A.$$

  The simulator outputs $\hat{\beta} = 0$ otherwise.

We now must argue that the simulation is accurate. The simulation of σ is perfect, since this is just a random λ-bit value in the real protocol.

The distribution of $\hat{\beta}$ is statistically close to the distribution in the real protocol. Whenever $c_A \neq \hat{c}_A$, in the real protocol, the audit server will return 0, as in the simulation. Whenever $d_A \neq H(\hat{\mathbf{m}}_A)$, in the real protocol, the audit server will return 0, as in the simulation. So the only time when the simulation and real protocol diverge is when $d_A = H(\hat{\mathbf{m}}_A)$ but $\hat{\mathbf{m}}_A \neq \mathbf{m}_A$. The probability (over the randomness of server A and the random oracle) that A outputs such a vector $\hat{\mathbf{m}}_A$ after making a polynomial number of random-oracle queries is negligible in the security parameter. Thus, the simulation is near perfect.

D Security Proof Sketches for the Three-Server Protocol

This appendix contains the security proofs for the three-server protocol for detecting malicious client requests (Section 5.1).

Completeness. If the pair of keys is well-formed then the $\mathbf{b}_A$ and $\mathbf{b}_B$ vectors (also the $\mathbf{s}_A$ and $\mathbf{s}_B$ vectors) are equal at every index $i \neq \ell_x$ and they differ at index $i = \ell_x$. Even in the negligibly unlikely event that the random seed chosen at $\mathbf{s}_A[\ell_x]$ is equal to the random seed chosen at $\mathbf{s}_B[\ell_x]$, the test vectors $\mathbf{t}_A$ and $\mathbf{t}_B$ will still differ because $\mathbf{b}_A[\ell_x] \neq \mathbf{b}_B[\ell_x]$. Thus, a correct pair of $\mathbf{b}$ and $\mathbf{s}$ vectors will pass the first $\mathsf{AlmostEqual}$ check.

The second AlmostEqual check is more subtle. If the $\mathbf{v}$ vector is well formed then, letting ℓ_x be the index where the $\mathbf{s}$ vectors differ, we have:

$$\begin{aligned}\mathbf{u}_B &= \left(\sum_{i=0}^{x-1} G(\mathbf{s}_A[i])\right) + G(\mathbf{s}_A[\ell_x]) + G(\mathbf{s}_B[\ell_x]) + \mathbf{v} \\ &= \mathbf{u}_A + G(\mathbf{s}_A[\ell_x]) + G(\mathbf{s}_B[\ell_x]) + \mathbf{v} \\ &= \mathbf{u}_A + m \cdot e_{\ell_y}\end{aligned}$$

If $\mathbf{v}$ is well-formed, then two test vectors $\mathbf{u}_A$ and $\mathbf{u}_B$ differ only at index ℓ_y.

Soundness. To show soundness, we must bound the probability that the audit server will output "1" when the servers take a malformed pair of DPF keys as input. If the $\mathbf{b}$ and $\mathbf{s}$ vectors are not equal everywhere except at one index, the soundness of the AlmostEqual protocol implies that the audit server will return "0" with overwhelming probability when invoked the first time.

Now, given that the $\mathbf{s}$ vectors differ at one index, we can demonstrate that if the $\mathbf{u}$ vectors pass the second AlmostEqual check, then $\mathbf{v}$ is also well formed. Let ℓ_x be the index at which the $\mathbf{s}$ vectors differ. Write the values of the $\mathbf{s}$ vectors at index ℓ_x as s_A^* and s_B^*. Then, by construction:

$$\begin{aligned}\mathbf{u}_A &= \left(\sum_{i \neq \ell_x}^{x-1} G(\mathbf{s}_A[i])\right) + G(s_A^*) \\ \mathbf{u}_B &= \left(\sum_{i \neq \ell_x}^{x-1} G(\mathbf{s}_B[i])\right) + G(s_B^*) + \mathbf{v}\end{aligned}$$

The first term of these two expressions are equal (because the $\mathbf{s}$ vectors are equal almost everywhere). Thus, to violate the soundness property, an adversary must construct a tuple $(s_A^*, s_B^*, \mathbf{v})$ such that the vectors $G(s_A^*)$ and $(G(s_B^*) + \mathbf{v})$ differ at exactly one index *and* such that $\mathbf{v} \neq G(s_A^*) + G(s_B^*) + m \cdot e_\ell$. This is a contradiction, however, since if $G(s_A^*)$ and $(G(s_B^*) + \mathbf{v})$ differ at exactly one index, then:

$$m \cdot e_{\ell_y} = G(s_A^*) + [(G(s_B^*) + \mathbf{v})]$$

for some ℓ_y and m, by definition of $m \cdot e_{\ell_y}$.

Zero Knowledge. The audit server can simulate its view of a successful run of the protocol (one in which the input keys are well-formed) by invoking the AlmostEqual simulator twice.

E Security Proof Sketches for the Zero-Knowledge Protocol

Completeness. Completeness for the first half of the protocol, which checks the form of the $\mathbf{B}$ and $\mathbf{S}$ vectors, follows directly from the construction of those vectors.

The one slightly subtle step comes in Step 5 of the second half of the protocol. For the protocol to be complete, it must be that $\mathbf{G}_{\text{sum}}$ is zero at every index except one. This is true because:

$$\mathbf{G}_{\text{sum}} = (\Sigma_{i=0}^{s-1} \mathbf{G}_i) + \mathbf{v} = G(s^*) + m \cdot e_{\ell_y} - G(s^*) = m \cdot e_{\ell_y}$$

Soundness. The soundness of the non-interactive zero-knowledge proof in the first half of the protocol guarantees that the $\mathbf{B}$ vectors sum to e_{ℓ_x} and that the $\mathbf{s}$ vectors sum to $s^* \cdot e_{\ell_x}$ for some values $\ell_x \in \mathbb{Z}_x$ and $s^* \in \mathbb{S}$.

We must now argue that the probability that all servers accept an invalid write request in the second half of the protocol is negligible. The soundness property of the underlying zero-knowledge proof used in Step 5 implies that the vector $\mathbf{G}_{\text{sum}}$ contains commitments to zero at all indices except one. A client who violates the soundness property produces a vector $\mathbf{v}$ and seed value s^* such that $(\Sigma_{i=0}^{s-1} \mathbf{G}_i) + \mathbf{v} = m \cdot e_{\ell_y}$ for some values $\ell_y \in \mathbb{Z}_y$ and $m \in \mathbb{G}$, and that $\mathbf{v} \neq m \cdot e_{\ell_y} - G(s^*)$. This is a contradiction, however, since $(\Sigma_{i=0}^{s-1} \mathbf{G}_i) = G(s^*)$, by the first half of the protocol, and so:

$$(\Sigma_{i=0}^{s-1} \mathbf{G}_i) + \mathbf{v} = m \cdot e_{\ell_y} = G(s^*) + \mathbf{v}$$

Finally, we conclude that $\mathbf{v} = m \cdot e_{\ell_y} - G(s^*)$.

Zero Knowledge. The servers can simulate every message they receive during a run of the protocol. In particular, they see only Pedersen commitments, which are statistically hiding, and non-interactive zero-knowledge proofs, which are simulatable in the random-oracle model [62].