Remise: Authorized Anonymous Communication Systems

Rohan Ravi
IIT Kanpur
rohanra@cse.iitk.ac.in

Paritosh Shukla
IIT Kanpur
paritoshs24@cse.iitk.ac.in

Adithya Vadapalli
IIT Kanpur
avadapalli@cse.iitk.ac.in

Abstract

We present Remise, a two-server authorized anonymous communication system built on Distributed Oblivious RAM (DORAM). Remise supports two modes of operation: (i) an anonymous bulletin board, where messages are publicly revealed at the end of each epoch without linking senders to messages, and (ii) anonymous communication channels, where messages remain secret-shared, and writers selectively grant and revoke read access to chosen readers. In both modes, the two servers execute read and write operations without learning which indices are accessed, which authorization tokens are used, or which relationships exist between writers and readers, assuming at least one honest server. A central contribution of Remise is a lightweight and efficient access-control mechanism. Authorization proofs are maintained in secret-shared form across the two servers, enabling oblivious verification while preventing leakage even under client–server collusion. Unlike prior DPF-based systems, Remise provides built-in auditing by having the semi-honest servers generate standard-basis vector shares internally, eliminating the need for server-side DPF validity checks. We implement a prototype of Remise and evaluate it under realistic network conditions. Our experiments show 80× improvement in online server time when compared to PACL (Spectrum) (IEEE S&P 2023) for databases of size 2^{24}.

1 Introduction

The birth of the Internet was accompanied by great optimism [21]. It was envisioned as a democratizing force—one that would enable free exchange of ideas, universal access to information, and participation independent of geography or social status. While this vision has been realized in many respects, the modern Internet also exposes users to pervasive surveillance, manipulation, and censorship [25]. As a result, users increasingly feel compelled to self-censor and communicate with caution. In this context, *anonymous communication systems* offer a compelling response. By decoupling identities from messages, such systems protect users from surveillance and retaliation and play a critical role in preserving free expression. They are particularly valuable for journalists, whistleblowers, activists, and ordinary users operating in adversarial environments.

1.1 Anonymous Communication Systems

Sasy and Goldberg [28] classify anonymous communication systems into (i) sender-unlinkable broadcast systems (SUBS), (ii) sender-unlinkable messaging systems (SUMS), (iii) relationship-unobservable systems (RUS), and (iv) communication-unobservable systems (CUS). In this work, we focus on the first three categories. In a SUBS, users anonymously post messages to a public bulletin board that is typically secret-shared across multiple servers and reconstructed at fixed epochs. While the set of messages and the anonymity set of broadcasters are public, the link between a message and its author is hidden; cover traffic further prevents distinguishing real messages from dummy ones. Conceptually, SUBS resemble an anonymous microblogging platform. In a SUMS, receivers own mailboxes identified by secret mailbox IDs that are shared out of band with intended senders. Senders remain anonymous, only authorized senders can write to a mailbox, and receivers obtain only messages addressed to them, though receiver anonymity is not provided. This model is commonly used for confidential disclosures, such as journalist–source communication (e.g., SecureDrop [15]). RUS extend SUMS by additionally ensuring that even a global adversary cannot link senders to receivers.

Sender Anonymous Communication Channels: This work introduces a new type of RUS that has not been studied hitherto, namely, Sender Anonymous Communication

Channels. In this model, a set of senders posts messages on a secret message board shared by two parties. However, the parties never reconstruct the messages to post them to the public. Instead, writers give read access to readers of their choice. Furthermore, the writers can revoke the access whenever they wish to do so. We could also think about this as an anonymous *Substack* or *Blog*. The subscribers (of the *Substack* or *Blog*) are the readers who have been granted read access by the authors. The authors (the writers) are granted write access by the service provider (the Remise servers). The writers have "ownership" of that space in the sense that they can write messages of their choice in the space, allow only the people of their choice to read messages from that space, and can remove read access for any reader at any point.

1.2 Review of State-of-the-Art

Remise follows the lineage of Anonymous Communication Systems like Riposte, Express, Sabre, and Spectrum, which are based on the so-called Distributed Point Functions (DPFs). We will go into the details of DPFs in Appendix B; however, for now, it suffices to regard a DPF as a functional, succinct representation of secret shares of a 1-hot vector. The servers use shares of a 1-hot vector produced by DPFs to perform read and write operations on a database without revealing which element was accessed. Performing a write operation can be done by adding shares of the 1-hot vector component-wise to the shares of the bulletin board that the servers are holding. Such systems typically need to perform two checks before allowing a write request to be executed: (i) the DPFs are valid, i.e., they will expand to the shares of a 1-hot vector *(audit)*, and (ii) the user is allowed to access at the index at which they are attempting to do so *(authorization)*.

Riposte: Corrigan-Gibbs, Boneh, and Mazières, presented Riposte [6], a sender-anonymous broadcasting system that works in the bulletin board model. In Riposte, the writers (or broadcasters) submit write requests, creating a DPF at a random location and sending it to the servers. The servers reconstruct and publish all the messages that were received during each epoch. A problem with users writing at random locations on the bulletin board is that a broadcaster could overwrite a message written by another broadcaster. To reduce the probability of such a write-collision to 5%, the size of the Riposte servers is 19.5 times the number of messages to be posted [1]. Riposte runs an audit protocol to ascertain that the write requests submitted are well-formed DPFs. One major weakness of Riposte is that it uses $O(\sqrt{n})$- sized DPFs, where n is the size of the database. This is because the structure of these DPFs make their auditing protocol efficient.

Express: In Express [14], each receiver is assigned a mailbox together with a $\lambda = 128$-bit mailbox ID by the servers. Receivers share this mailbox ID with intended senders, who write messages by generating DPFs targeting the corresponding ID. Although senders generate DPFs over a domain of size 2^{128}, the Express servers evaluate them only at registered mailbox IDs. Express improves upon Riposte by using logarithmic-sized DPFs together with SNIPs for auditing. However, auditing requires the servers to evaluate each submitted DPF at all registered mailbox IDs, incurring work linear in the number of mailboxes. As a result, Express is vulnerable to DoS attacks in which malicious clients submit malformed DPFs that force expensive evaluations before rejection. Moreover, Express disallows server–client collusion, since servers know all registered mailbox IDs and can thereby derive write authorization for any mailbox.

Sabre: The work by Vadapalli, Storrier, and Henry, Sabre [33] extends the work by Express and prevents DoS attacks by presenting $O(\lg n)$ auditing. For authorization, Sabre uses a mailbox ID verification using pseudorandom functions. If a writer is authorized to write at the $\mathtt{i}^{*\text{th}}$ location, the writer should know $\mathtt{PRF}(\mathtt{k},\mathtt{i}^*)$ while only the servers know the secret key, $\mathtt{k}$. The writer submits the additive shares of $\mathtt{PRF}(\mathtt{k},\mathtt{i}^*)$ along with the shares of $\mathtt{i}^*$ to the Sabre servers. Then, the Sabre servers use MPC to compute the shares of $\mathtt{PRF}(\mathtt{k},\mathtt{i}^*)$, and check for equality. However, this requires the servers to compute a PRF in a secure Multiparty Computation (MPC), and therefore, the underlying cipher chosen needs to have a low multiplicative depth. Like Express, Sabre also disallows collusion between a server and a user, as the servers can calculate the authorization proof, $\mathtt{PRF}(\mathtt{k},\mathtt{i})$, for all index $\mathtt{i}$, by virtue of knowing the key $\mathtt{k}$.

Spectrum: Spectrum [24] is an anonymous broadcasting system introduced by Newman, Servan-Schreiber, and Devadas with a low number of broadcasters. Each broadcaster in Spectrum is allotted a "channel" in the database, where they are authorized to write. Spectrum presents a blind access control mechanism which allows the servers to perform auditing and authorization over the DPFs in the same protocol. In addition, Spectrum has a Blame-Game protocol that protects against malicious Spectrum servers from rejecting valid write requests. Unlike Express and Sabre, Spectrum allows for server-writer collusion under the discrete log hardness assumption.

PACL: Unlike prior work on anonymous communication, PACL [29] addresses access control for Function Secret Sharing, which can also be applied to DPF-based communication systems. It builds on the Spectrum method for blind access control and introduces more effi-

[1] Riposte mitigates collisions by sending (M, M^2), enabling recovery from two-way collisions. A board of size $2.7n$ yields a 5% collision rate, where n is the expected number of messages per epoch.

Table 1: Comparison of Remise with prior systems. N = database size, L = number of registered writers, λ = security parameter. "Auth Secret Hiding" indicates whether per-index authorization material is hidden from individual servers. "Auth Update Privacy" indicates whether authorization state can be updated or revoked without revealing to the servers. For Riposte, $N > L$, while for all other systems $N = L$. Riposte does not have an authorization mechanism. * Express authorization overlaps with database write. ** Spectrum and PACL have unified audit and authorization mechanism. *** Clients send secret shares of index (instead of DPF keys), requiring $\log L$ bits.

System	Client Req. Size	Server Audit Work	Server Auth Work	Auth Secret Hiding	Auth Update Privacy
Riposte	$O(\sqrt{N})$	$O(N)$	-	-	-
Express	$O(\lambda^2)$	$O(L)$	$O(L)^*$	×	×
Sabre	$O(\log L)$	$O(\log L)$	$O(1)$	×	×
Spectrum	$O(\log L)$	$O(L)$	$O(L)^{**}$	✓ (DL-Hard)	×
Spectrum + PACL	$O(\log L)$	$O(L)$	$O(L)^{**}$	✓ (DL-Hard)	×
Remise	$O(\log L)^{***}$	-	$O(L)$	✓ (Secret-shared)	✓

cient techniques. In the basic setup, servers hold a vector of verification keys $\Lambda = \{g^{\alpha_1}, \ldots, g^{\alpha_n}\}$, where secret key α_i authorizes access to index i (DPF generation at i). The DPF is used to locally select (multiplicative) secret shares of $g^{[\alpha_i]}$,enabling verification through additive shares of a proof $[\Pi]_k \leftarrow [-\alpha_i]_k$. The servers then check that $\left(g^{[\alpha_i]_k}\right) \cdot g^{[\Pi]_k}$ yields additive shares of 1. This essentially generalizes the Spectrum protocol to multiple channels, allowing simultaneous auditing and authorization. PACL improves efficiency by replacing exponent-based checks with a *Schnorr proof over secret shares*, which proves knowledge of the discrete logarithm of an additively shared element. A second construction, the *inclusion predicate*, supports multiple access keys per index. Here, servers store an $n \times \ell$ matrix of verification keys and use one DPF to select the i^{th} row and another to pick the correct column via correction terms, yielding the required shares for the audit.

We compare Remise and prior systems in Table 1. We only consider variants that use 2 main servers (for example, Spectrum can be instantiated with more than 2 servers).

1.3 System Overview

Remise is a two-server anonymous communication system that enforces authorization for both read and write operations while providing anonymity. The servers verify authorization without learning the accessed index, the message or the authorization proof. The system supports two instantiations: a public bulletin board model (RemiseBB) and a communication channels model (RemiseCC). In RemiseBB, authorized writers anonymously post messages to pre-assigned slots in the bulletin board. This bulletin board is periodically reconstructed and published. All write requests made by writers (actual or cover traffic), look indistinguishable to the servers or an adversary observing the network. As a result, the link between the writers and the published messages is severed. In RemiseCC, the messages posted by the writers are not published. Instead, writers can selectively grant and revoke read access to chosen readers and only an authorized reader can read from a given index. RemiseCC allows writers to update (grant or revoke) read permissions. The RemiseCC servers remain oblivious to both the identities of authorized readers and any changes in the authorization state. Here, all read and write requests (actual and cover traffic) are indistinguishable to the servers and the network, which hides writer-reader link.

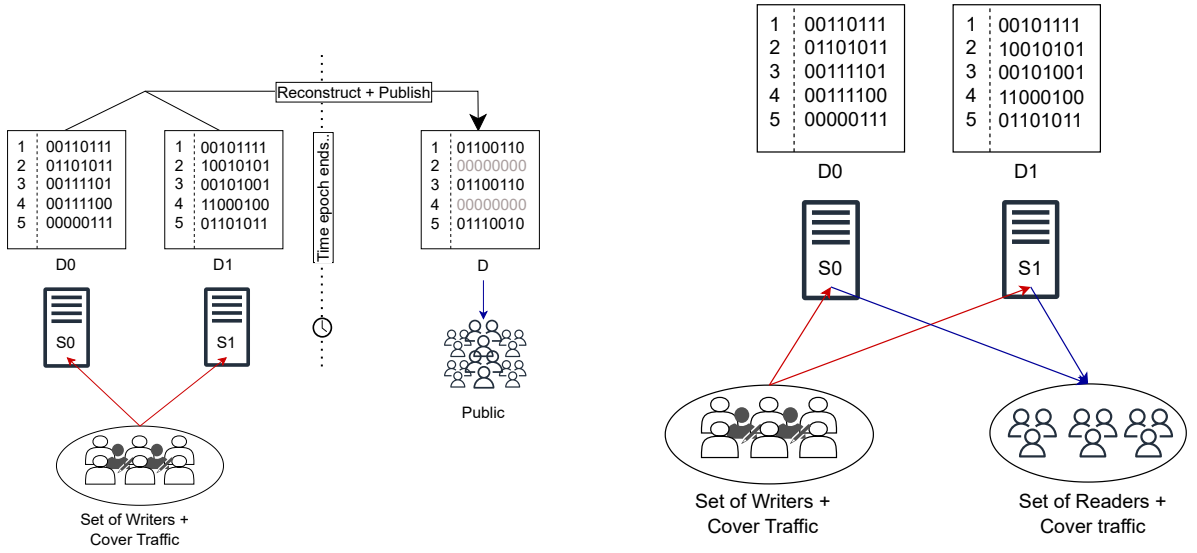
Figure 1 gives a brief overview of RemiseBB and RemiseCC and its interaction with users.

Contributions We make the following contributions: (i) We introduce a DORAM-based authorization mechanism for message boards that provides built-in auditability, (ii) We design communication channels in which writers can grant read permissions to readers without servers learning which reader received access, and (iii) We support revocation of access privileges while hiding from the servers whose permissions were modified.

1.3.1 Main ideas in realizing Remise

Distributed ORAM for hidden accesses: Remise is a Distributed Oblivious Random Access Memory (DORAM) based Anonymous Communication System. While we defer the detailed discussion of DORAMs to Section 2.3, it suffices for now to know that DORAMs enable multiple parties, each holding secret shares of a memory and an index, to perform an oblivious (write or read) access to the memory at the index. Remise servers act like DORAM parties and perform operations on the secret shared database and secret shared inputs.

Built-in Auditing: In previous systems that use



(a) RemiseBB: Remise instantiated in the anonymous bulletin board model. To any adversary or servers, all write requests from the set of (Writers + Cover Traffic) look indistinguishable. Writers write to allocated space, which the servers reconstruct and publish after every time epoch.

(b) RemiseCC: Remise instantiated in the anonymous communication channels model. Authorized writers post messages that can only be accessed by authorized readers. All requests from the set of (Writers + Cover traffic) is indistinguishable. Similarly, all read requests made from the set of (Readers + Cover traffic) is indistinguishable. The messages are hidden from any other entity and writer-reader link is hidden.

Figure 1: Remise system overview

DPFs [6, 14, 24, 33] to perform an access on the secret shared databases held by the servers, the client provides the DPF keys for the servers to evaluate. Consequently, the servers have to perform an *audit* check to ensure that the DPF keys evaluate to valid shares of a 1-hot vector, otherwise malformed DPFs could corrupt the entire database. A consequence of using DORAMs (like Floram [10] and Duoram [32], which Remise does) is that the shares of the standard-basis vector are generated by the servers themselves, thereby offloading this computation from potentially malicious clients to the servers.

Resistance to Client–Server Collusion: In Remise, authorization proofs are stored as secret shares across servers, ensuring that authorization secrets remain hidden as long as at least one server is semi-honest, even under client–server collusion. In contrast, Express and Sabre store authorization material in the clear, allowing colluding servers to leak authorization secrets. While Spectrum (and PACL) prevent this leakage under computational assumptions, Remise provides information-theoretic security.

Authorization Revocability Remise (as RemiseCC) supports authorization revocability by allowing writers to obliviously update read authorization tokens associated with messages they authored. This functionality is essential in anonymous communication settings where a writer controls read access for a dynamic set of readers. Writers may grant access to new readers or revoke access from previously authorized readers, while preserving unlinkability between writers, readers, and database indices. As a consequence of authorization proofs being maintained in secret-shared form, Remise enables these updates to be performed via oblivious DORAM update operations, without exposing which authorization state was modified. In contrast, while Spectrum (and PACL) hide the secret key used for authorization, any modification to the vector of verification keys is directly visible to the servers, precluding oblivious authorization updates.

2 Background

2.1 Secret Shares

Secret sharing is a cryptographic mechanism for distributing a secret among multiple parties such that any subset of fewer than t parties learns no information about the secret, while any subset of t or more parties can reconstruct it. The parameter t is called the *threshold* of the secret-

sharing scheme. Secret-sharing schemes come in many flavors. We denote additive secret sharing of a value x modulo 2^r by x^{AS}. When using (k,k) additive sharing among k parties, the individual shares are denoted by $x_0^{\text{AS}},\ldots,x_{k-1}^{\text{AS}}$, and satisfy $\sum_{i=0}^{k-1} x_i^{\text{AS}} \equiv x \pmod{2^r}$. We use b^{BS} to denote a Boolean share, and x^{GS} to denote a generic (secret) sharing of x. An alternative way to secret-share an index $\mathtt{i}^*$ is via shares of a *one-hot vector*, i.e., a vector that is zero everywhere except at the position corresponding to $\mathtt{i}^*$. Shares of one-hot vectors are particularly useful in sender-anonymous communication. Distributed Point Functions (DPFs) provide a succinct mechanism for secret-sharing standard-basis vectors. In particular, $(2,2)$-DPFs allow two parties to share a standard-basis vector. DPFs form a core building block of the Distributed ORAM constructions used in Remise. Since DPFs are used only as a black box in our construction, we defer their formal definition to Appendix B. For our purposes, it suffices to note that DPFs enable sharing a standard-basis vector of length n with $O(\log n)$ communication.

2.2 Secure Multiparty Computation

Secure Multiparty Computation (MPC) is a cryptographic technique that allows two or more parties to compute a function on secret inputs. A somewhat more formal definition of MPC is as follows:

Definition 1 (Secure Multi-Party Computation)**.** *In a multi-party computation (MPC) protocol, parties* $\mathtt{S}_1,\ldots,\mathtt{S}_n$ *holding private inputs* $x_1,\ldots,x_n$ *jointly compute a function* $f(x_1,\ldots,x_n)$ *such that no party learns anything beyond what can be learned from the output.*

Measuring the performance of MPC: There are three main metrics to evaluate the performance of an MPC protocol, namely, (i) local computation, (ii) bandwidth, and (iii) round complexity. Local computation like the name suggests is the total amount of computational work the parties in the MPC protocol do. The bandwidth consumption is total packet size that needs to be communicated throughout the protocol. The round complexity is the number of Internet latencies that need to be spent to execute the protocol. In the following we will present some of the MPC protocols that will be used in the paper.

Multiplication: We denote the multiplication operation as $z^{\text{GS}} \leftarrow \textsc{Mult}(x^{\text{GS}}, y^{\text{GS}})$. Du-Atallah [12] is a variant of the classic Beaver's method [11] to do multiplication on secret shares, which we use in this paper.

EQZ Protocol: The *Equal to Zero* checks if the given value is equal to zero. We denote operation is $b^{\text{AS}} \leftarrow \textsc{Eqz}(x^{\text{AS}})$. The two parties who hold the shares of a variable x, who run the Eqz protocol. The protocol results in them getting shares of a bit b, such that $b=1$ if $x=0$, else $b=0$. Our implementation uses the EQZ protocol from Damgård et al [9].

The Eqzand Multiplication protocols are described in Appendix C.

2.3 Oblivious Memory Access

Distributed Oblivious Random Access Memory (DORAM). DORAM is a special case of Secure Multiparty Computation (MPC) in which the computed functionality is a memory access. In this work, we consider two-party DORAMs.

Let $\mathtt{D} \in \mathbb{F}^n$ be a database and $\mathtt{i}^* \in [n]$ be an index. Two servers $\mathtt{S}_0$ and $\mathtt{S}_1$ hold additive shares $\mathtt{D}_0^{\text{AS}}, \mathtt{D}_1^{\text{AS}}$ of $\mathtt{D}$, and additive shares $\mathtt{i}^{*\text{AS}}_0, \mathtt{i}^{*\text{AS}}_1$ of $\mathtt{i}^*$. A DORAM read protocol outputs $\mathsf{read}_0, \mathsf{read}_1$ such that $\mathsf{read}_0 + \mathsf{read}_1 = \mathtt{D}[\mathtt{i}^*]$, which we denote by $\mathtt{D}^{\text{AS}}[\mathtt{i}^{*\text{AS}}]$. Given additive shares $\mathtt{M}^{\text{AS}}$ of a value $\mathtt{M}$, a DORAM write produces updated database shares $\mathtt{D}'^{\text{AS}}$ satisfying $\mathtt{D}'[\mathtt{i}] = \mathtt{D}[\mathtt{i}]$ for all $\mathtt{i} \neq \mathtt{i}^*$, and $\mathtt{D}'[\mathtt{i}^*] = \mathtt{D}[\mathtt{i}^*] + \mathtt{M}$.

Read-Only and Write-Only Instantiations. We also consider restricted instantiations of DORAM supporting only a single operation.

In a *Distributed Oblivious Read-Only Memory* (DOROM), the database is fixed after setup. An access to a read-only database at a secret-shared index $\mathtt{i}^{*\text{AS}}$ is denoted by $\mathtt{D}^{\text{RO}}[\mathtt{i}^{*\text{AS}}]$. Both servers hold an encrypted copy of the database: for all $\mathtt{i} \in [n]$, $\mathtt{D}[\mathtt{i}] + \mathtt{PRF}(\mathtt{k}_0, \mathtt{i}) + \mathtt{PRF}(\mathtt{k}_1, \mathtt{i})$, where $\mathtt{S}_0$ holds key $\mathtt{k}_0$ and $\mathtt{S}_1$ holds $\mathtt{k}_1$. A read operation is implemented via PIR on the encrypted database, followed by an MPC step to obtain additive shares of $\mathtt{D}[\mathtt{i}^*]$.

In a *Distributed Oblivious Write-Only Memory* (DOWOM), the servers hold additive shares of the database (notice that this is different from how it is stored in DOROM!). An update to a write-only database at a secret-shared index $\mathtt{i}^{*\text{AS}}$ by a value $\mathtt{M}^{\text{AS}}$ is denoted by $\mathtt{D}^{\text{WO}}[\mathtt{i}^{*\text{AS}}] \leftarrow \mathtt{D}^{\text{WO}}[\mathtt{i}^{*\text{AS}}] + \mathtt{M}^{\text{AS}}$. Such an update is implemented by locally adding additive shares of $\mathbf{e}_{\mathtt{i}^*} \cdot \mathtt{M}$ to the database shares. It is easy to switch from the memory layout for DOWOM to DOROM. Parties choose a PRF key $\mathtt{k}_b$ and exchange $\mathtt{D}_b^{\text{AS}} + \mathtt{PRF}(\mathtt{k}_b)$ to compute $\mathtt{D}_0^{\text{AS}} + \mathtt{PRF}(\mathtt{k}_0) + \mathtt{D}_0^{\text{AS}} + \mathtt{PRF}(\mathtt{k}_0)$.

DOROM and DOWOM are naturally more efficient than a full DORAM. This is because supporting both read and update operations necessitates either switching between distinct memory layouts, as in Floram [10], or maintaining additional auxiliary state such as blinds and blind shares, as in Duoram [32].

In this paper, we instantiate DOROM and DOWOM using the protocols in Floram due to Doerner and shelat (see §3 in the Floram paper [10]). When we need a DORAM (that supports both read and write operations), we use 2P-Duoram [32].

Preprocessing in DORAMs All the standard-basis vector shares needed in DORAMs (including read only and write only) are produced using DPFs. It is important to note that we produce the DPFs in a preprocessing phase (before we know where to write and what to write) and adjust them to the correct index and value in the online phase with constant communication. For more details we refer the reader to §4.1 in the Duoram paper [32].

3 Authorized Messaging

Remise operates with two non-colluding servers, $\mathtt{S}_0$ and $\mathtt{S}_1$, and can be instantiated in two models: (i) the *Anonymous Bulletin Board* model, denoted as RemiseBB, and (ii) the *Anonymous Communication Channels* model, denoted as RemiseCC. Two main types of users interact with Remise, namely, (i) writers, and (ii) readers. A subset of the users also send *zero* messages. Such messages are called cover traffic and serve the purpose of increasing the size of the anonymity set. For the Remise servers, cover messages are information-theoretically indistinguishable from real write requests.

Setup Phase The functioning of Remise (both RemiseBB and RemiseCC) begins with a setup phase, during which writers obtain authorization to write at specific indices, while the servers remain oblivious to which index was assigned to which writer. In particular, the setup ensures that none of the Remise servers learn (i) which users (from the superset of users including cover traffic and actual writers) are authorized to write messages, or (ii) which indices belong to which writers.

RemiseBB In RemiseBB, writers post messages on a shared bulletin board held by the servers which are reconstructed and published after every epoch. A writer submits a write request by sending shares of the message, shares of the index, and shares of the write authorization proof. The servers accept the request if the proof corresponds to the index (of course, without learning either the proof or the index). Authorized indices in RemiseBB allow for preventing write-collisions as writers are only allowed to write at their assigned index. It also capitalizes on the asymmetry of real-world broadcasting, similar to what Spectrum does, where there are fewer writers than total users in a system. Thus while keeping the server computation only be influenced by the number of real writers, we can still derive anonymity from all users without reserving extra space.

RemiseCC In contrast,in RemiseCC, while writers post messages similar to RemiseBB, the two servers never reconstruct the message database. Instead, the writers grant reading rights to specific readers of their choice, and this information is conveyed to the RemiseCC servers obliviously. Similar to write authorization proofs, the servers in RemiseCC also hold shares of read authorization proofs and the writer has control over the values of the read authorization proofs for the index they own. Readers submit read requests by sending shares of the index and shares of the read authorization proofs to the servers (e.g., $\mathtt{i}^*$) in order to retrieve a message. Upon receiving a read request, the RemiseCC servers verify (obliviously) whether the reader has permission to read from $\mathtt{i}^*$. If the permission is not granted, the read request is dropped.

Notation Throughout, for any value, array, or database D, we denote its additive sharing between servers $\mathtt{S}_0$ and $\mathtt{S}_1$ by $\mathrm{D} = \mathrm{D}_0^{\mathrm{AS}} + \mathrm{D}_1^{\mathrm{AS}}$, where $\mathtt{S}_b$ holds $\mathrm{D}_b^{\mathrm{AS}}$ for $b \in \{0,1\}$. If D is an array or database, then for every index $\mathtt{i}$, $\mathrm{D}[\mathtt{i}] = \mathrm{D}_0^{\mathrm{AS}}[\mathtt{i}] + \mathrm{D}_1^{\mathrm{AS}}[\mathtt{i}]$. We use D to denote both the bulletin board in RemiseBB and the message board in RemiseCC. Servers $\mathtt{S}_0$ and $\mathtt{S}_1$ hold additive shares of D, as well as PRF-encrypted versions of the write-authorization proof database, denoted by PROOFDB. For each index $\mathtt{i}$, the entry $\textsc{ProofDB}[\mathtt{i}]$ encodes the write-authorization proof required to write to the $\mathtt{i}^{\text{th}}$ location of D. The database PROOFDB is read-only. In RemiseCC, the servers additionally hold additive shares of the read-authorization proof database, denoted by $\mathcal{T}$. We use the terms *token* and *read-authorization proof* interchangeably. Each entry $\mathcal{T}[\mathtt{i}]$ consists of a list of m tokens, where possession of any one token suffices to prove authorization to read the contents of $\mathrm{D}[\mathtt{i}]$. During the setup phase, servers $\mathtt{S}_0$ and $\mathtt{S}_1$ hold additive shares of the setup database SETUPDB, to which writers post their public keys. At the end of setup, each writer learns its assigned indices via a published database denoted by ACCESSDB.

Threat Model The Remise clients (writers, readers and cover traffic) are completely untrusted. They may deviate arbitrarily from the protocol, form collusion with other clients or with malicious servers. For metadata privacy, Remise requires at least one server to be honest. For integrity, Remise requires both the servers to execute the protocols as specified. We assume a global network adversary that observers all communication between clients and servers. We assume the existence of pseudorandom generators and pseudorandom functions for any randomness and key generation used in the protocols. We assume the existence of secure channels between each client and each server, providing confidentiality, integrity and replay protection for communicated messages.

3.1 Remise APIs

Remise exposes a set of APIs that the users use to interact with the system in both RemiseBB and RemiseCC. In this section, we describe each API by first explaining its purpose and behaviour, and then stating an informal security claim that it is designed to achieve.

3.1.1 Writer Registration: The Setup Phase

Overview Remise (both RemiseBB and RemiseCC) provides a registration API that writers invoke during a one-time setup phase to obtain authorization to write to the database D. Each registered writer is assigned a database index together with a corresponding write authorization proof, which must be presented with every write request. The setup phase hides from the servers both the set of registered writers and the indices assigned to them: while the servers may observe a superset of users submitting requests (including cover traffic), they cannot distinguish registered writers from non-writers.

Description The setup phase of Remise begins with a *writer registration phase*, which follows a bootstrapping mechanism similar to that of Spectrum [24] and relies on an anonymous broadcast system such as Riposte [6].

The setup phase begins with writer registration. Here, servers $\mathtt{S}_0$ and $\mathtt{S}_1$ hold additive shares of an anonymous bulletin board SETUPDB. Writers express their intent to reserve database indices by anonymously writing their public keys to random locations in SETUPDB using a Riposte-style write. A writer may reserve multiple indices by writing multiple public keys to different locations. The servers accept writes to SETUPDB for a fixed time window, after which all further requests are dropped, marking the end of the writer registration phase.

Once the writer registration phase ends, the servers reconstruct SETUPDB and recover the set of public keys written at random locations. Each public key $\mathtt{pk}$ appearing in SETUPDB corresponds to a writer requesting write authorization. For each such $\mathtt{pk}$, the servers deterministically assign a unique database index $\mathtt{i}^*$ using a fixed, agreed-upon ordering. The servers jointly initialize the write-authorization proof database PROOFDB, for which they hold additive shares. To generate a write authorization proof for index $\mathtt{i}^*$ without revealing it to either server, $\mathtt{S}_0$ and $\mathtt{S}_1$ sample random strings ℓ_0 and ℓ_1 such that $\ell_0 + \ell_1 = \ell$, where ℓ is the authorization proof for $\mathtt{i}^*$. They locally update their shares by setting $\text{PROOFDB}_0[\mathtt{i}^*] \leftarrow \ell_0$ and $\text{PROOFDB}_1[\mathtt{i}^*] \leftarrow \ell_1$. After initializing all entries, the servers convert their additive shares of PROOFDB into encrypted form. Each server $\mathtt{S}_b$ applies a PRF under its secret key to encrypt its share $\text{PROOFDB}_b^{\text{AS}}$ and exchanges the resulting encrypted shares with the other server. Both servers then reconstruct the database, obtaining a doubly encrypted, read-only proof database denoted by $\text{PROOFDB}^{\text{RO}}$. Finally, for each writer with public key $\mathtt{pk}$, each server encrypts its share of the authorization proof together with the assigned index $\mathtt{i}^*$ under $\mathtt{pk}$. These ciphertexts are written to a separate database ACCESSDB at the same position from which $\mathtt{pk}$ was read in SETUPDB. The published database ACCESSDB concludes the setup phase, enabling each writer to decrypt and learn its assigned index and corresponding write authorization proof.

Figure 2 illustrates the setup phase in Remise.

Security Claim An adversary controlling all but one server and a strict subset of writers (excluding at least two honest writers), and observing the network cannot determine which indices and write authorization proofs are assigned to an honest writer, beyond what is inferred from the published ACCESSDB. Definition 3 formalizes this notion.

3.1.2 Write Operation in Remise

Overview In both RemiseBB and RemiseCC, registered writers are allowed to submit messages at their assigned locations. The system checks each submission and, if permitted, stores it in the message database. In the RemiseBB variant, a bulletin board is published at the end of each epoch, whereas in RemiseCC, no bulletin board is published. Users willing to provide cover traffic may also send write requests by sending *zero* messages.

Description A user submits a *write* request to servers $\mathtt{S}_0$ and $\mathtt{S}_1$ by sending $(\mathtt{i}^{*\text{AS}}, \Delta\mathtt{M}^{\text{AS}}, \omega^{\text{AS}})$, which are additive shares of the target index, the value update, and the corresponding write authorization proof. Here, $\mathtt{i}^*$ is the write index and $\Delta\mathtt{M} = \mathtt{M}' - \mathtt{M}$, where $\mathtt{M}'$ is the new message and $\mathtt{M}$ is the current value of $\mathrm{D}[\mathtt{i}^*]$.[2] Since only the authorized writer may modify $\mathrm{D}[\mathtt{i}^*]$, we assume the writer knows $\mathtt{M}$. Remise permits any user, authorized or not, to submit a *zero* write with $\Delta\mathtt{M} = 0$ at any index as cover traffic; the servers cannot distinguish such requests from authorized non-zero writes, and higher cover traffic increases the anonymity set. The servers hold additive shares of the write authorization proof database PROOFDB, where $\text{PROOFDB}[\mathtt{i}]$ stores the authorization proof for index $\mathtt{i}$. Upon receiving a write request, $\mathtt{S}_0$ and $\mathtt{S}_1$ perform a DOROM read on PROOFDB using shares of $\mathtt{i}^*$ to obtain shares of the expected authorization proof. Since PROOFDB is read-only, this operation is more efficient. The servers then use MPC to check whether either the retrieved authorization proof matches the writer-supplied proof or the request corresponds to cover traffic; in other words, they verify that $(\omega = \text{PROOFDB}[\mathtt{i}^*]) \vee (\Delta\mathtt{M} = 0)$. If the check succeeds, the write is applied. In RemiseBB, the message database

[2] Writes are implemented as DORAM updates, which add secret shares to the existing database shares.

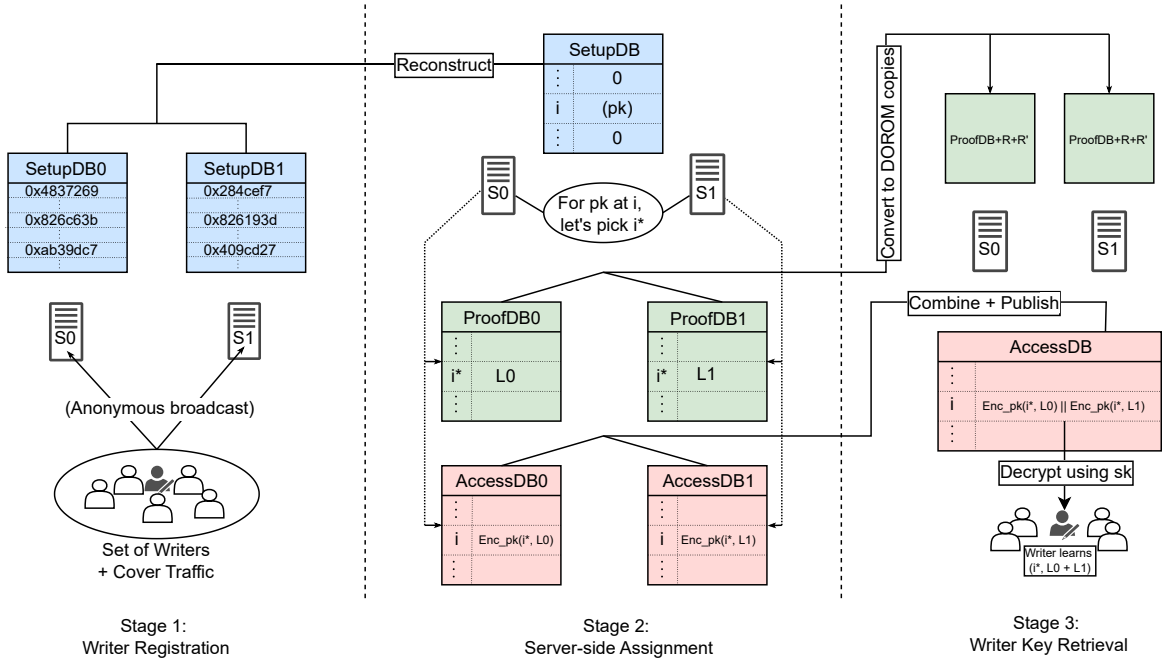


Figure 2: Remise setup phase: In Stage 1: writers post their public keys using an anonymous broadcast system like Riposte. In Stage 2: servers get the value of public key and assign an index $\mathtt{i}^*$. They also sample their respective shares of the write authorization proof ($L0$ and $L1$), so that no single server knows the actual value of the proof. In Stage 3: the servers combine their respective encrypted shares and publish it. They also convert PROOFDB into DOROM copies.

is write-only: servers batch write requests (including cover traffic) into fixed epochs and publish $\mathrm{D}_0^{\mathrm{AS}} + \mathrm{D}_1^{\mathrm{AS}}$ at the end of each epoch. In contrast, RemiseCC supports read access to the secret-shared database; consequently, writes are performed using a full DORAM write. The message database is not published, and writers may grant read access to their messages to selected readers.

Security Claim An adversary controlling all but one server, any subset of clients, and observing the network cannot, with more than negligible advantage, link a write access by an honest user to a particular database index or message, beyond what is implied by the set of participating users. Additionally, only a writer who knows the correct write authorization proof for an index (for $\mathtt{i}$, it is $\textsc{ProofDB}[\mathtt{i}]$) can cause a non-zero message to be written at that index. All other write requests leave the database unchanged. Definition 4 in Appendix D formalizes this notion.

3.1.3 Updating Read Authorization in RemiseCC

Overview Writers can decide who can read the messages they write. They do this by updating the values of the read authorization proofs for the locations they are assigned to write at. Writers can then communicate values of the proofs to new readers, using some out-of-band method. This step ensures that message access remains controlled and private.

Description Granting read access to an index $\mathtt{i}^*$ in RemiseCC consists of two steps: (i) issuing the reader a read-authorization token (a λ-bit string), and (ii) obliviously informing the servers that possession of this token grants read access to $\mathtt{i}^*$.

A writer may share the token with a reader either via an initial in-person exchange or by using an expensive dialing protocol as in Express [14].[3] After distributing the token, the writer must inform the servers that possession of the token $\mathrm{t}_{\mathtt{i}^*}$ authorizes reading index $\mathtt{i}^*$.

Servers $\mathtt{S}_0$ and $\mathtt{S}_1$ hold additive shares of a token database $\mathcal{T}$. Each entry $\mathcal{T}[\mathtt{i}]$ consists of a list of m tokens, where possession of any one token suffices to read $\mathrm{D}[\mathtt{i}]$. To add or revoke read access at index $\mathtt{i}^*$, the writer generates a fresh list of m tokens and sends additive shares of this list to the servers; revocation is achieved by replacing tokens in the list.

Before updating $\mathcal{T}$, the servers verify that the requester is authorized to write to index $\mathtt{i}^*$ by checking the supplied write-authorization proof, exactly as in the write operation (Section 3.1.2). If the check succeeds, $\mathtt{S}_0$ and

[3] While one may ask why messages themselves are not shared this way, note that in-person contact or dialing is required only once.

S_1 update their shares of $\mathcal{T}$ via a DORAM write.

To support cover traffic, RemiseCC designates a public-update index i_{upd} whose write-authorization proof is public. Any user may submit updates to $\mathcal{T}[i_{upd}]$, ensuring that valid token updates do not reveal the set of writers.

Security Claim Only a writer who has the value PROOFDB$[i^*]$ can make changes to $\mathcal{T}[i^*]$ when all servers act honestly. Also an adversary cannot with more than negligible advantage, determine which row (if any) of $\mathcal{T}$ was updated, after a valid request. This notion is formalized in Definition 6 in Appendix D.

3.1.4 Read Operation in RemiseCC

Overview Readers who have been granted read access can request to retrieve messages using the read authorization proofs provided by the writer. The servers checks their authorization, and if valid, provides the reader with the message content.

Description A reader submits a *read* request by sending (i^{*AS}, t^{AS}), consisting of additive shares of the target index and a read-authorization token received from the writer. Upon receiving the request, the RemiseCC servers perform a DORAM read on the token database at index i^* to obtain shares of the corresponding m tokens. They then use MPC to verify that the supplied token matches at least one entry in this list. If the check succeeds, the servers carry out a DORAM read on the message database; otherwise, the request is dropped.

To hide the set of readers, RemiseCC supports cover traffic for read operations. Specifically, the servers reserve a public-read index i_{read} whose read-authorization tokens are publicly known and whose write-authorization proof is undisclosed, preventing updates to $\mathcal{T}[i_{read}]$. Any user may submit cover read requests to i_{read} using one of the public tokens. The index i_{read} is never assigned to any writer during the setup phase.

Security Claim An adversary controlling all but one server, any subset of clients, and the network cannot, with more than negligible advantage, link a read access by an honest user to a particular database index. Additionally, only a reader who holds one of the m tokens in $\mathcal{T}[i^*]$ can read $D[i^*]$. Definition 5 in Appendix D formalizes this notion.

4 Remise: The Details

4.1 Writing in Remise

Protocol 1 describes the process of writing in Remise. w has full control over the value $D[i^*]$ and thus we assume it knows the value M before making write request. Before any write is made $D[i] = 0$ for all index i. Line 3 checks if the payload value (ΔM), is zero. If ΔM is zero, $z^{AS} \leftarrow 1^{AS}$, otherwise $z^{AS} \leftarrow 0^{AS}$. Line 4 does a DOROM read into the proof database to get shares of the actual proof of write authorization. Line 6 computes an MPC OR of two check values, z and auth, to ensure either a zero message is provided or the authorization passed. In RemiseBB, Line 9 is a write-only memory access, while in RemiseCC, it is a DORAM write.

Protocol 1 WRITE IN REMISE: Writer w wants to write M' at $i^{*\text{th}}$ index of the database D. w knows the current value of $D[i^*]$ as M and holds ω as write authorization proof. S_0 and S_1 hold shares of D and PROOFDB. At the end of the protocol, if the write authorization passes, S_0 and S_1 will end up with new shares of D, such that $D[i^*] \leftarrow M'$

1: w calculates $\Delta M \leftarrow M' - M$ // new val. - curr. val.
2: For each $b \in \{0,1\}$, w sends $(i^{*AS}_b, \Delta M^{AS}_b, \omega^{AS}_b)$ to S_b.
// Servers obliviously check for zero message
3: $z^{AS} \leftarrow \text{EQZ}(\Delta M^{AS})$
// Servers do DOROM read on the proofs database
4: $\mathcal{F}^{AS} \leftarrow \text{PROOFDB}^{RO}[i^{*AS}]$
5: $\text{auth}^{AS} \leftarrow \text{EQZ}(\mathcal{F}^{AS} - \omega^{AS})$
// MPC OR operation
6: $\text{audit}^{AS} \leftarrow \text{auth}^{AS} \vee z^{AS}$
// Servers reconstruct audit
7: $\text{audit} \leftarrow \text{REC}(\text{audit}^{AS})$
8: **if** audit = 1 **then**
9: $\quad D^{AS}[i^{*AS}] \leftarrow D^{AS}[i^{*AS}] + \Delta M^{AS}$ // $D[i^*] = M$
10: **else**
11: $\quad$ **return** // Reject the request and perform no write
12: **end if**

4.2 Providing Read Authorization

Recall that providing read access has two main parts. Firstly, the writer has to do an out-of-band communication to give the reader a new read authorization token. Secondly, the writer updates the token database $\mathcal{T}$ at the RemiseCC servers, to add the new read authorization token. Protocol 2 describes the protocol that allows a writer to update the values of read authorization tokens for an index. A writer can use the same protocol to revoke access as well. The writer may replace the value of an existing token, as a result of which, any reader holding that token will no longer be able to make authorized reads. The RemiseCC servers have to first ascertain, that the writer updating the tokens for an index, is indeed the author of the messages at that index (say i^*). Lines 4 to 6 authenticate the writer. Once the writer is authenticated, in Line 8 $\mathcal{T}$ is updated via a DORAM update operation, such that the list of tokens at index i^* holds new token

values that the writer wants. Cover traffic is provided by having a reserved index $\mathtt{i}_{\text{upd}}$, whose write authorization proof is made public, at the end of the setup phase. Any user willing to provide cover traffic can make updates to $\mathtt{i}_{\text{upd}}$ using the same protocol.

Protocol 2 UPDATE READ AUTHORIZATION IN REMISECC: Writer w has write authorization to index $\mathtt{i}^*$ and knows the current list of m tokens at $\mathcal{T}[\,\mathtt{i}^*\,]$ denoted by $R_{\mathtt{i}^*}$. $\mathtt{S}_0$ and $\mathtt{S}_1$ hold shares of a token database $\mathcal{T}$. w wants to write new list $R'_{\mathtt{i}^*}$ at $\mathcal{T}[\,\mathtt{i}^*\,]$.

// w prepares new tokens for $\mathtt{i}^*$.
1: w constructs $R'_{\mathtt{i}^*} = \{\mathsf{t}_1, \ldots, \mathsf{t}_m\}$.
// w computes the element-wise difference
2: $\Delta R_{\mathtt{i}^*} \leftarrow R'_{\mathtt{i}^*} - R_{\mathtt{i}^*}$.
3: For each $b \in \{0,1\}$, w sends $(\mathtt{i}^{*\mathrm{AS}}, \Delta R_{\mathtt{i}^*}{}^{\mathrm{AS}}, \omega^{\mathrm{AS}})$
// Servers check writer's authorization.
4: $\mathcal{F}^{\mathrm{AS}} \leftarrow \textsc{ProofDB}^{\mathrm{AS}}[\mathtt{i}^{*\mathrm{AS}}]$
5: $\text{check}^{\mathrm{AS}} \leftarrow \textsc{Eqz}(\mathcal{F}^{\mathrm{AS}} - \omega^{\mathrm{AS}})$
6: $\text{check} \leftarrow \textsc{Rec}(\text{check}^{\mathrm{AS}})$
7: **if** check = 1 **then**
// Authorized: update token list at $\mathtt{i}^*$.
8: $\quad \mathcal{T}^{\mathrm{AS}}[\mathtt{i}^*] \leftarrow \mathcal{T}^{\mathrm{AS}}[\mathtt{i}^*] + \Delta R_{\mathtt{i}^*}{}^{\mathrm{AS}}$
9: **else**
10: $\quad$ **return** // Reject the request
11: **end if**
// Cover traffic: any user may run the same protocol with $\mathtt{i}^* = \mathtt{i}_{\text{upd}}$, using the public write-proof for $\mathtt{i}_{\text{upd}}$.

4.3 Reading in RemiseCC

Protocol 3 describes the protocol to read in RemiseCC. We first remark that the notion of reading in RemiseBB does not exist since that is a broadcasting system, where the messages are published. Lines 2 to 11 authenticate the reader. To authenticate, the servers obviously check that the reader knows at least one of the read tokens associated with the index they are trying to access. The read tokens of a special index, $\mathtt{i}_{\text{read}}$, are made public. Readers providing cover traffic simply send read requests at $\mathtt{i}_{\text{read}}$ using one of the known tokens. The write proof for $\mathtt{i}_{\text{read}}$, however, must be hidden by the servers. This is to prevent any malicious user from updating $\mathcal{T}[\mathtt{i}_{\text{read}}]$ using Protocol 2.

4.4 RemiseBB (Sabre): Authorization w/o DORAMs

If we relax our threat model to disallow collusion between a Remise server and a writer, we can achieve authorization without relying on DORAMs by borrowing Sabre's authorization check. Let $F : \{0,1\}^\lambda \times I \to \{0,1\}^\lambda$ be a PRF. The authorization token for an index i is defined as $\tau_i = F_{\mathtt{k}}(i)$, where the PRF key $\mathtt{k}$ is known to the two servers. A client authorized to access index $\mathtt{i}^*$ is given the token $\tau_{\mathtt{i}^*} = F_{\mathtt{k}}(\mathtt{i}^*)$ during setup. To make an access, the client submits additive shares of the target index $\mathtt{i}^{*\mathrm{AS}}_0, \mathtt{i}^{*\mathrm{AS}}_1$, along with additive shares of the token $\tau_0^{\mathrm{AS}}, \tau_1^{\mathrm{AS}}$. Using their shares of $\mathtt{i}^*$, the servers jointly evaluate the PRF under MPC to compute $\tau^\star = F_{\mathtt{k}}(\mathtt{i}^*)$, and then perform an MPC equality check to verify that $\tau^\star = \tau_{\mathtt{i}^*}$. If the check succeeds, the access is authorized. The authorization check has constant computational complexity and communication complexity. As in Sabre, collusion between a server and a client is out of scope: since the servers know the PRF key, they can compute valid authorization tokens for any index, and therefore such collusion trivially enables unauthorized access. Unlike Sabre, RemiseBB (Sabre) does not need to perform an audit to check the validity of the DPFs. We remark that com-

Protocol 3 READ IN REMISECC: A reader r wants to read content $\mathrm{D}[\,\mathtt{i}^*\,]$ using read authorization token t. Servers hold shares of D and $\mathcal{T}$. At the end of the protocol, r reads $\mathrm{D}[\mathtt{i}^*]$ if $\mathsf{t} \in \mathcal{T}[\,\mathtt{i}^*\,]$.

// Reader sends shares of index and its read authorization token to the servers
// Readers providing cover traffic would send same request for the reserved index $\mathtt{i}_{\text{read}}$
1: For each $b \in \{0,1\}$, r sends $(\mathtt{i}^{*\mathrm{AS}}_b, \mathsf{t}_b^{\mathrm{AS}})$ to $\mathtt{S}_b$
// Servers do a DORAM read to get shares of the list of tokens for $\mathtt{i}^*$
2: $T^{\mathrm{AS}} \leftarrow \mathcal{T}^{\mathrm{AS}}[\mathtt{i}^{*\mathrm{AS}}]$
3: Parse $T^{\mathrm{AS}} = \{\mathsf{t}_1{}^{\mathrm{AS}}, \ldots, \mathsf{t}_m{}^{\mathrm{AS}}\}$
// Servers calculate additive shares of the check value for each of the token in the list
4: **for all** $\mathsf{t}_i \in T$ **do**
5: $\quad \text{check}_i{}^{\mathrm{AS}} \leftarrow \mathsf{t}_i{}^{\mathrm{AS}} - \mathsf{t}^{\mathrm{AS}}$
6: **end for**
// Servers check if any token matches
7: **for** $i = 1$ to m **do**
8: $\quad z_i{}^{\mathrm{AS}} \leftarrow \textsc{Eqz}(\text{check}_i{}^{\mathrm{AS}})$
9: **end for**
10: $\text{auth}^{\mathrm{AS}} \leftarrow \bigvee_{i=1}^m z_i{}^{\mathrm{AS}}$
11: $\text{auth} \leftarrow \textsc{Rec}(\text{auth}^{\mathrm{AS}})$ // servers reconstruct
12: **if** auth = 0 **then**
// Allow read operation
13: $\quad \text{read}^{\mathrm{AS}} \leftarrow \mathrm{D}^{\mathrm{AS}}[\,\mathtt{i}^{*\mathrm{AS}}\,]$
14: $\quad \mathtt{S}_0$ and $\mathtt{S}_1$ send $\text{read}_0^{\mathrm{AS}}$ and $\text{read}_1^{\mathrm{AS}}$ to the reader.
// Reconstruction of the shares
15: $\quad$ The reader computes: $\text{read} \leftarrow \text{read}_0^{\mathrm{AS}} + \text{read}_1^{\mathrm{AS}}$
16: **else**
// Drop the read request
17: $\quad$ **return**
18: **end if**

bining RemiseCC with Sabre does not yield an efficient construction. To preserve writer–reader unlinkability, the PRF key must be hidden from the servers and generated by the writer. As a result, the servers would need to obliviously select the correct key during authorization, which incurs significant overhead. Consequently, the DORAM-based authorization mechanism presented earlier is more efficient.

We end this section by remarking that throughout all our protocols, although the outcome of the audit (i.e., whether it succeeds or fails) is revealed to the servers at the end of the protocol, this leakage can be eliminated by having the servers perform a no-operation when the audit fails. Similarly, the distinction between read and write requests can be hidden using the same technique. We defer further discussion of these extensions to Appendix E.

5 Implementation and Evaluation

Analytical Evaluation Before presenting the experimental evaluation, we discuss the asymptotic costs. Table 2 describes the asymptotic costs in realizing Remise. DPF evaluation in MPC accounts for the preprocessing cost. Notice that the costs in RemiseCC have factor of m, the number of read tokens per database index.

Implementation We implemented a proof-of-concept prototype[4]. All experiments were conducted on a server equipped with two *Intel® Xeon® Gold 6430* processors, 251 GiB of RAM, and running a 64-bit *Linux* operating system. Network latency and bandwidth were emulated using `tc netem`, and all experiments were run on an otherwise idle system.

5.1 Comparisons with Comparators

In this section, we compare Remise with the various anonymous communication systems in PACL's codebase[5], namely Spectrum and Express.

5.1.1 RemiseBB vs. Spectrum (PACL)

In this section, we compare RemiseBB with Spectrum (PACL), i.e., Spectrum integrated with PACL [29]. Both systems operate under a similar threat model that allows client–server collusion. Figure 3a compares the wall-clock time for servers to *check the validity* of a single write request in RemiseBB and Spectrum (PACL) under three network latencies: 10 ms, 30 ms, and 60 ms. We do not include a comparison with plain Spectrum, since integrating PACL improves its performance by nearly 40×. For smaller database sizes, Spectrum (PACL) outperforms the *total* cost (preprocessing plus online) of RemiseBB. This behavior is expected. The preprocessing phase of RemiseBB —specifically, the MPC-based generation of shares of a standard-basis vector—requires $O(\log n)$ rounds of communication. In contrast, Spectrum (PACL) relies on the evaluation of a verifiable DPF, which, while computationally more expensive than a standard DPF, avoids this additional round complexity. As the database size increases, computational overhead begins to dominate latency-induced costs. This trend is reflected in our results: at lower latencies, the crossover point at which RemiseBB outperforms Spectrum (PACL) occurs at smaller database sizes. For a database of size 2^{24}, RemiseBB achieves a 5× improvement over Spectrum (PACL), and this improvement increases to approximately 11× at size 2^{26}. We were unable to execute the Spectrum (PACL) implementation beyond 2^{26}; however, extrapolating from existing measurements, the performance gap is expected to further favor RemiseBB at larger scales. Finally, when considering only the online phase, the gains are substantially larger. For example, at a database size of 2^{24}, RemiseBB provides nearly an 80× speedup over Spectrum (PACL).

5.1.2 RemiseBB (Sabre) vs. Express (PACL)

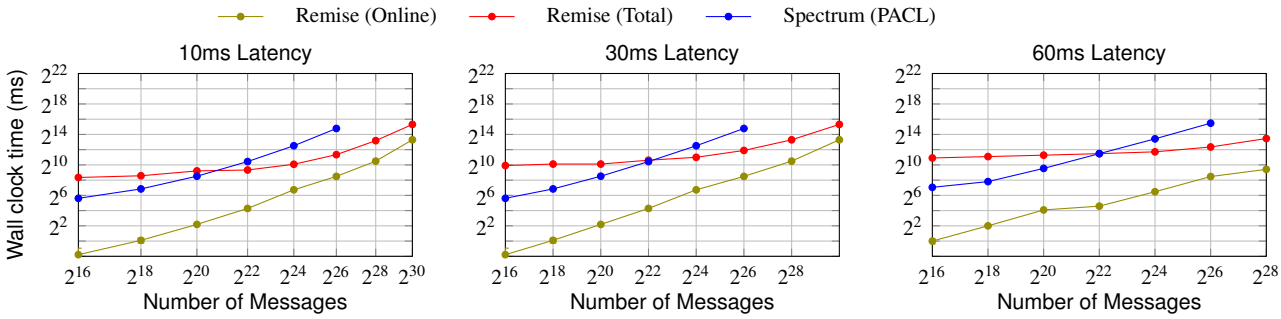
Figure 3b compares the the wall-clock time for servers to *check the validity* of a single write request in RemiseBB (Sabre) with Express (PACL) under three traffic mixes: (i) 100% authorized writes, (ii) 50% unauthorized writes, and (iii) 100% unauthorized writes. Recall that both RemiseBB (Sabre) and Express (PACL) operate under a threat model in which server–client collusion is disallowed. We do not plot the online time for RemiseBB (Sabre), as it would be similar to the plot for 100% unauthorized writes. The difference in total time comes from the amount of DPFs generated in the preprocessing phase. The main observation from Figure 3b is that the fraction of unauthorized writes has a significant impact on the running time of RemiseBB (Sabre). Since the authorization check in RemiseBB (Sabre) takes $O(1)$ time, unauthorized requests can be rejected early, well before any expensive cryptographic processing. In contrast, Express (PACL) must perform a full DPF evaluation before authorization can be verified. When all writes are authorized, RemiseBB (Sabre) outperforms Express (PACL) by approximately 8×. When 50% of the writes are unauthorized, this gap widens to 16×. In the extreme case where all writes are unauthorized, RemiseBB (Sabre) is approximately 18,000× faster than Express (PACL). Notably, the running time of Express (PACL) remains essentially unchanged across these settings, as it incurs the

[4] An anonymized implementation is available at: https://github.com/anonymoususer99946996/anonymous

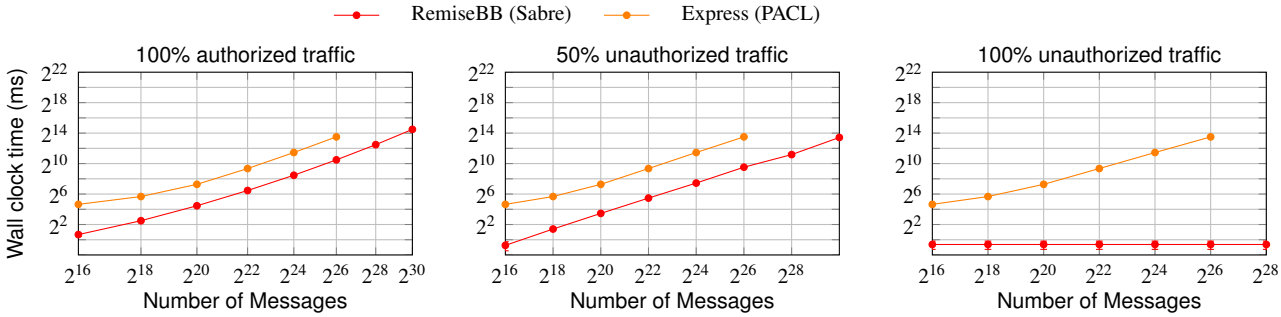
[5] Code from https://github.com/sachaservan/pacl/tree/main/bench-anon

Table 2: Cost Comparison of Remise Variants. Cost mentioned in gray is the preprocessing cost.

Operation	Local Computation	Bandwidth	Rounds
Remise Write	$O(n) + O(n)$	$O(\lg n) + O(1)$	$O(\lg n) + O(1)$
RemiseCC Read	$O(n) + O(n \cdot m)$	$O(\lg n) + O(m)$	$O(\lg n) + O(1)$
RemiseCC Update Authorization	$O(n) + O(n \cdot m)$	$O(\lg n) + O(m)$	$O(\lg n) + O(1)$



(a) RemiseBB vs. Spectrum (PACL); experiments for Spectrum (PACL) were limited to size 2^{26} due to infrastructure constraints.



(b) RemiseBB (Sabre) vs. Express (PACL); experiments for Express (PACL) were limited to size 2^{26} due to infrastructure constraints.

Figure 3: Performance comparison of a write operation in RemiseBB with Spectrum (PACL) for different latencies.

same cost to process both authorized and unauthorized requests.

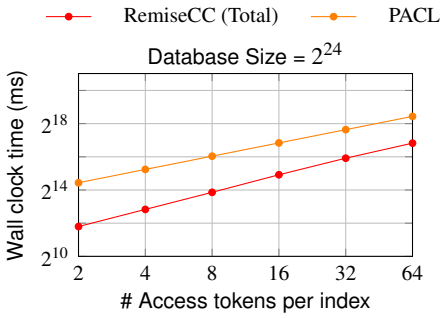


Figure 4: Comparing RemiseCC and PACL's read authorization.

5.1.3 RemiseCC vs. PACL-Inclusion Predicate

In Figure 4, we evaluate the time taken by the servers to perform authorization checks on read requests in RemiseCC. We compare against PACL's inclusion predicate access control, as it provides the same functionality of having multiple read authorization tokens per index. The experiment is done on a database of size 2^{24} while varying the number access tokens per index. For 64 tokens per index, RemiseCC is $3\times$ faster than PACL's inclusion predicate. We do not plot the online time for RemiseCC as it is close to the total time because we use a 2 party Duoram, which uses a Computationally symmetric PIR (CSPIR) [32].

6 Related Work

DC-nets based Systems: Anonymous communication systems often use dining cryptographer networks (DC-nets) to achieve sender-recipient unobservability, where only one participant sends a real message per round while others send cover traffic. DC-nets, however, suffer from high communication overhead and poor scalability. They are also vulnerable to disruption, as any malicious participant can corrupt a broadcast by sending an invalid share. Dissent [7] addresses this with a costly blame protocol to detect disruptors but cannot prevent disruptions. Verdict [8] improves on this by requiring participants to submit proofs of correctness, enabling verifiable DC-nets that can prevent malformed contributions.

PIR-based Systems: Pynchon Gate [27] uses private information retrieval (PIR) to privately fetch messages, making it suitable for pseudonymous mail retrieval. Pung [1] extends this with a PIR-based key-value store: each round, a client privately retrieves a single message, using a key derived from a shared secret and receiving the encrypted value. Talek [5] supports private group messaging in a publish-subscribe model, functionally similar to RemiseCC. A writer communicates with multiple readers via a private log indexed using a blocked cuckoo hash table [26], with clients applying a PRF to a secret log handle to determine pseudorandom locations. Distributed-PIR [30] aims to reduce server computation in private messaging systems by offloading computation to the client, at the cost of increasing client overhead.

Mix-Net Based Systems: Mix-net systems obfuscate message sources, enabling anonymous messaging. In Riffle [19], servers perform verifiable shuffles of messages in each round and forward them, relying on the *anytrust* model—anonymity is preserved as long as one server is honest. Cmix [4] and Trellis [22] improves efficiency by using a precomputation phase. Atom [18] uses multiple server groups that shuffle messages iteratively. Security holds if each group has at least one honest server. XRD [20] assigns each client a mailbox and ensures uniform traffic by using loopback messages for non-communicating users. Communication occurs via shared chains between user pairs, hiding communication patterns.

Differential Privacy based Systems: Vuvuzela [34] pioneered this approach, where users communicate via shared drop locations derived from a shared key. Karaoke [23] builds on Vuvuzela with an efficient noise verification mechanism. Groove [2] further improves DP-based systems by adding user flexibility.

ORAM-based Systems: Myco [17] introduces ORAM-based metadata-private messaging system using an asymmetric two server model. Here, the servers act like an ORAM-like client and server. Myco achieves polylogarithmic per-access server work, outperforming PIR and DPF-based systems. Myco does not provide server enforced access control: any client can read messages from any location but must trial-decrypt each entry. While Remise presents a *Distributed*-ORAM based communication system, it aims to improve access control mechanisms on DPF-based communication systems, where servers enforce which locations a read/write access can be made. Any unauthorized access in Remise is rejected by the servers.

7 Conclusion

Remise enforces that only authorized clients can read from or write to the database. Importantly, the Remise servers perform access control checks without learning (i) the content of the messages written, (ii) which messages are being read, or (iii) any linkage between reads and writes. Furthermore, Remise can be easily extended to i) hide even whether an authorization check succeeded, adding an additional layer of access privacy, ii) hide whether a read or write operation was performed, and iii) achieve robustness.

Ethical Considerations

Stakeholder Identification and Impacts. We structure this section as a stakeholder-based analysis, distinguishing between *direct stakeholders*—those who design, implement, or deploy Remise —and *indirect stakeholders*—those affected by systems built using Remise. Direct stakeholders include the research team and organizations that deploy anonymous communication services based on Remise. Indirect stakeholders include end users who rely on such systems for private and anonymous communication, journalists and whistleblowers who may depend on strong sender anonymity, regulators and auditors responsible for oversight, and adversaries who may attempt to disrupt or misuse anonymous communication infrastructures.

Research Team. This work involves no human subjects, no physical experiments, and no collection of real-world private data. All evaluation is conducted using synthetic workloads and controlled experimental settings. We explicitly acknowledge and respect the intellectual contributions of all authors, and authorship reflects substantive technical involvement.

Service Providers and System Deployers. Organizations deploying Remise (e.g., operators of anonymous bulletin boards or private communication platforms) may benefit from improved efficiency, fast auditing, and strong

authorization privacy. At the same time, deployment requires careful engineering, cryptographic review, and parameter selection. Misconfiguration or incomplete implementation of the protocols—particularly those involving authorization proofs, preprocessing, or blame mechanisms—could lead to service disruption, denial of access for honest users, or weakened anonymity guarantees. To mitigate these risks, we document system assumptions, protocol tradeoffs, and performance characteristics in detail.

End Users and Society. *Beneficence.* Remise is designed to strengthen privacy and freedom of expression by enabling authorized anonymous communication without revealing access patterns, sender–receiver relationships, or authorization state. The system is particularly relevant for high-risk users such as journalists, whistleblowers, activists, and individuals operating under surveillance.

Respect for Persons. We carefully cite prior work in anonymous communication, DPF-based systems, and DORAMs to ensure accurate attribution and to avoid misrepresentation of related techniques. The design of Remise explicitly avoids requiring users to reveal identities, long-term metadata, or sensitive personal information to the servers.

Justice. The protocols underlying Remise rely only on standard cryptographic primitives and do not require specialized hardware or proprietary infrastructure. Our goal is to keep the core techniques accessible to researchers and system designers worldwide, reducing barriers to entry and enabling independent verification and reuse.

Respect for Law and Public Interest. We recognize that anonymous communication systems may raise concerns for regulators and law-enforcement agencies. Remise is designed to operate within the bounds of applicable law, assuming lawful deployment and governance by service providers. The system includes explicit mechanisms—such as authorization controls and blame protocols—to prevent unauthorized writes, mitigate denial-of-service attacks, and detect malicious behavior by clients or servers. These features are intended to balance anonymity with accountability at the system level, without enabling surveillance of honest users.

Publication and Research Impact. This work contributes new techniques for combining DORAMs, authorization, and anonymous communication, and may influence future research on privacy-preserving systems. To reduce misuse or misinterpretation, we document key design decisions and tradeoffs, including the efficiency benefits of read-only and write-only memory instantiations, the assumption of at least one honest server, and the limitations of the threat model.

Potential Harms and Dual-Use Risks. We acknowledge that anonymous communication systems can be misused by malicious actors to conceal unlawful activities. Additionally, immature or unreviewed deployments of Remise could harm honest users by causing loss of availability, incorrect blame attribution, or weakened anonymity. We therefore strongly encourage responsible deployment, thorough security auditing, and integration with appropriate legal, organizational, and governance frameworks. We emphasize that Remise is a research prototype and that production deployments should be preceded by careful review.

Decision to Conduct and Publish. This research was initiated to improve the efficiency, robustness, and accountability of anonymous communication systems. We considered the dual-use nature of such systems and concluded that the benefits to privacy, free expression, and secure communication outweigh the potential risks when the system is deployed responsibly. The decision to publish followed internal review, correctness validation, and performance evaluation. We believe that open dissemination of our results promotes transparency, reproducibility, and informed discussion within the research community, consistent with established norms in security and privacy research.

Open Science Our code is available at https://github.com/anonymoususer99946996/anonymous.

Acknowledgments The authors used generative AI-based tools to revise the text, improve flow and correct any typos, grammatical errors, and awkward phrasing.

References

[1] Sebastian Angel and Srinath Setty. Unobservable Communication over Fully Untrusted Infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[2] Ludovic Barman, Moshe Kol, David Lazar, Yossi Gilad, and Nickolai Zeldovich. Groove: Flexible Metadata-Private messaging. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[3] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function Secret Sharing: Improvements and Extensions. In

Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS), pages 1292–1303. ACM, 2016.

[4] David Chaum, Debajyoti Das, Farid Javani, Aniket Kate, Anna Krasnova, Joeri de Ruiter, and Alan T. Sherman. cMix: Mixing with minimal real-time asymmetric cryptographic operations. Cryptology ePrint Archive, Paper 2016/008, 2016.

[5] Raymond Cheng, William Scott, Elisaweta Masserova, Irene Zhang, Vipul Goyal, Thomas Anderson, Arvind Krishnamurthy, and Bryan Parno. Talek: Private Group Messaging with Hidden Access Patterns. In *Proceedings of the 36th Annual Computer Security Applications Conference*, ACSAC '20, page 84–99, 2020.

[6] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An Anonymous Messaging System Handling Millions of Users. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 321–338. IEEE Computer Society, 2015.

[7] Henry Corrigan-Gibbs and Bryan Ford. Dissent: accountable anonymous group messaging. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 340–350. ACM, 2010.

[8] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively accountable anonymous messaging in verdict. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 147–162. USENIX Association, 2013.

[9] Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. *Proc. IEEE Symp. Security and Privacy*, 2019, 2019.

[10] Jack Doerner and Abhi Shelat. Scaling ORAM for Secure Computation. In *CCS*, pages 523–535. ACM, 2017.

[11] Donald Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO*, pages 420–432, 1991.

[12] Wenliang Du and Mikhail J. Atallah. Protocols for Secure Remote Database Access with Approximate Matching. In *E-Commerce Security and Privacy (Part II)*, Advances in Information Security, Feb 2001.

[13] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function Secret Sharing. In *Advances in Cryptology - EUROCRYPT 2015*, pages 337–367, 2015.

[14] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the Cost of Metadata-hiding Communication with Cryptographic Privacy. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1775–1792. USENIX Association, 2021.

[15] Freedom of the Press Foundation. Securedrop, 2025. Open-source whistleblower submission system. Latest release: 2.12.8 (May 8, 2025).

[16] Niv Gilboa and Yuval Ishai. Distributed Point Functions and Their Applications. In *Advances in Cryptology - EUROCRYPT 2014*, 2014.

[17] Darya Kaviani, Deevashwer Rathee, Bhargav Annem, and Raluca Ada Popa. Myco: Unlocking polylogarithmic accesses in metadata-private messaging. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 4419–4437, 2025.

[18] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 33rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 463–480. ACM, 2017.

[19] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An Efficient Communication System With Strong Anonymity. *Proc. Priv. Enhancing Technol.*, 2016(2), 2016.

[20] Albert Kwon, David Lu, and Srinivas Devadas. XRD: scalable messaging system with cryptographic privacy. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 759–776. USENIX Association, 2020.

[21] Emily B. Laidlaw. *Regulating Speech in Cyberspace: Gatekeepers, Human Rights and Corporate Responsibility*. Cambridge University Press, 2015.

[22] Simon Langowski, Sacha Servan-Schreiber, and Srinivas Devadas. Trellis: Robust and scalable metadata-private anonymous broadcast. Cryptology ePrint Archive, Paper 2022/1548, 2022.

[23] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 711–725, Carlsbad, CA, October 2018. USENIX Association.

[24] Zachary Newman, Sacha Servan-Schreiber, and Srinivas Devadas. Spectrum: High-bandwidth anonymous broadcast. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[25] Jason R. C. Nurse. Cybercrime and you: How criminals attack and the human factors that they seek to exploit, October 2018.

[26] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[27] Len Sassaman, Bram Cohen, and Nick Mathewson. The pynchon gate: a secure method of pseudonymous mail retrieval. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, WPES '05, page 1–9, New York, NY, USA, 2005. Association for Computing Machinery.

[28] Sajin Sasy and Ian Goldberg. Sok: Metadata-protecting communication systems. *Proc. Priv. Enhancing Technol.*, 2024(1):509–524, 2024.

[29] Sacha Servan-Schreiber, Simon Beyzerov, Eli Yablon, and Hyojae Park. Private access control for function secret sharing. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[30] Elkana Tovey, Jonathan Weiss, and Yossi Gilad. Distributed pir: Scaling private messaging via the users' machines. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 1967–1981, New York, NY, USA, 2024. Association for Computing Machinery.

[31] Adithya Vadapalli, Fattaneh Bayatbabolghani, and Ryan Henry. You May Also Like... Privacy: Recommendation Systems Meet PIR. *Proc. Priv. Enhancing Technol.*, 2021(4):30–53, 2021.

[32] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. Duoram: A Bandwidth-Efficient Distributed ORAM for 2- and 3-Party Computation. Cryptology ePrint Archive, Paper 2022/1747, 2022. https://eprint.iacr.org/2022/1747.

[33] Adithya Vadapalli, Kyle Storrier, and Ryan Henry. Sabre: Sender-Anonymous Messaging with Fast Audits. In *IEEE Symposium on Security and Privacy (SP)*, pages 1953–1970, 2022.

[34] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: scalable private messaging resistant to traffic analysis. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 137–152. ACM, 2015.

A Setup Phase Details

In this Appendix we will present the detailed exposition of the Setup Pahse of Remise that we described in Section 3.1.1.

Protocol 4 describes in detail how a writer can register to write to the database D in Remise (both RemiseBB and RemiseCC). In stage 1, writers post their public keys to SETUPDB, which is held in shares by $\mathtt{S}_0$ and $\mathtt{S}_1$. The Remise servers can announce a time window within which writers (and users providing cover traffic) can post to SETUPDB, after which the servers do not accept any requests. Although the protocol shows, w registering for a single index, any writer can reserve multiple indices by making more than one request. The servers can also set a maximum limit to the number of requests made during the setup phase in order to not let a user exhaust server resources. In line 6, the servers jointly assign each posted public key to a unique database index by evaluating an agreed-upon deterministic function ASSIGN on a common counter ctr. The function ASSIGN(ctr) can, for example, enumerate database indices in a fixed order or apply a fixed permutation.

In lines 9-10 each server samples a random string ℓ_b, which are indeed the secret shares of the write authorization proof, as we can see $\text{PROOFDB}_0[\mathtt{i}^*] + \text{PROOFDB}_1[\mathtt{i}^*] = \ell_0 + \ell_1$. This is done so that neither of the servers learn the actual value of the proof. Lines 16-19 shows how the servers convert secret shares of PROOFDB into encrypted copies which is necessary to perform DOROM operations.

B Distributed Point Function

A point function is a function across a domain that evaluates to 0 at every point in the domain, except at one special point where it evaluates to a non-zero value. Observe that the evaluation of the point functions over their entire domain can be viewed as 1-hot vectors. Distributed functions are an efficient way to secret share these 1-hot vectors in a succinct way. Gilboa and Ishai introduced

Protocol 4 SETUP PHASE IN REMISE: A writer w requests write authorization to an index in D. Servers $\mathtt{S}_0$ and $\mathtt{S}_1$ hold secret shares of a setup bulletin board, SETUPDB. A write authorization proof database, PROOFDB, is also secret shared between the two servers. PROOFDB is empty at the start of the protocol.

STAGE 1: Writer Registration

w generates key pair $(\mathtt{pk}, \mathtt{sk})$
w writes pk to a random location i in SETUPDB, i.e. SETUPDB$[\,\mathtt{i}\,] \leftarrow \mathtt{pk}$, using a Riposte-styled write

STAGE 2: Server-Side Assignment

// After the writer-registration window ends
$\mathtt{S}_0$ and $\mathtt{S}_1$ reconstruct SETUPDB
$\mathtt{S}_0$ and $\mathtt{S}_1$ set ctr $\leftarrow 0$
for all $\mathtt{i} \in [N]$ such that SETUPDB$[\,\mathtt{i}\,]$ is valid public key pk **do**
  // Servers compute deterministic assignment of index
  $\mathtt{S}_0$ and $\mathtt{S}_1$ compute $\mathtt{i}^* \leftarrow$ ASSIGN(ctr)
  ctr $\leftarrow$ ctr $+ 1$
  for $b \in \{0,1\}$ **do**
    $\mathtt{S}_b$ samples a random string $\ell_b \xleftarrow{\$} \{0,1\}^\lambda$
    $\mathtt{S}_b$ stores PROOFDB$_b[\mathtt{i}^*] \leftarrow \ell_b$
    $\mathtt{S}_b$ computes $c_b \leftarrow \mathrm{ENC}_{\mathtt{pk}}(\mathtt{i}^* \,||\, \ell_b)$
  end for
  $\mathtt{S}_0$ and $\mathtt{S}_1$ write $(c_0 \,||\, c_1)$ to ACCESSDB$_b[\,\mathtt{i}\,]$
end for
$\mathtt{S}_0$ and $\mathtt{S}_1$ publish ACCESSDB$_b$
// servers convert PROOFDB shares into encrypted copies
For each $b \in \{0,1\}$, $\mathtt{S}_b$ samples key $\mathtt{k}_b$
$\mathtt{S}_b$, constructs vector $R_b[\,\mathtt{i}\,] = \mathtt{PRF}(\mathtt{k}_b, \mathtt{i})$ for all index i
$\mathtt{S}_b$ sends (PROOFDB$_b + R_b$) to $\mathtt{S}_{1-b}$
// $\mathtt{S}_0$, $\mathtt{S}_1$ hold (PROOFDB$_0 + R_0$), (PROOFDB$_1 + R_1$)
Servers calculate and store (PROOFDB $+ R_0 + R_1$) $\leftarrow$ PROOFDB$_0 + R_0 +$ PROOFDB$_1 + R_1$

STAGE 3: Writer Key Retrieval

w retrieves ACCESSDB$[\,\mathtt{i}\,]$
w decrypts using sk to obtain $(\mathtt{i}^*, \ell_0, \ell_1)$
w reconstructs $\ell \leftarrow \ell_0 + \ell_1$ // ℓ is the write authorization proof for $\mathtt{i}^*$

Distributed Point Functions (DPFs) [16], with further improvements by Boyle et al. [3, 13]. In this paper, we only concern ourselves with DPFs that share a point function between exactly two parties. Definition 2 (a restatement of Vadapalli et al. [33, Def 4]) formally defines $(2,2)$-DPFs.

Definition 2. *A* $(2,2)$-distributed point function*, or* $(2,2)$-DPF*, is a pair of PPT algorithms* $(Gen, Eval)$ *defining an infinite family of secret-shared representations of generalized point functions; that is, given (i) a security parameter* $\lambda \in \mathbf{N}$*, (ii) a domain D and range R, and (iii) a distinguished point* $(x,y) \in D \times R$*, we have*

- *If* $(dpf_0, dpf_1) \leftarrow Gen(1^\lambda, D, R;\ x, y)$*, then, for all* $i \in D$*,*

$$Eval(dpf_0, i) + Eval(dpf_1, i) := \begin{cases} y & \text{if } i = x, \text{ and} \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

- *There exists a PPT simulator* $\mathcal{S}$ *such that, for any given domain D, range R, distinguished point* $(x,y) \in D \times R$*, and bit* $b \in \{0,1\}$*, the distribution ensembles:*

  1. $\{\mathcal{S}(1^\lambda, D, R;\ b)\}_{\lambda \in \mathbf{N}}$ *and*
  2. $\{dpf_b \mid (dpf_0, dpf_1) \leftarrow Gen(1^\lambda, D, R;\ x, y)\}_{\lambda \in \mathbf{N}}$

  are computationally indistinguishable.

The dpf_b *output by Gen are called* $(2,2)$-DPF keys*. We note that, the x-coordinate of the distinguished point is the* distinguished input*.*

Evaluating the DPF over the entire domain is done by a function Evalfull. Observe that, Evalfull(dpf_0) and Evalfull(dpf_1) are the shares of a 1-hot vector. For the finer details of DPF construction, we refer the reader to the original DPF paper [3] and some other manuscripts, which use DPFs [31–33].

B.1 Generating DPFs via Secure Computation

Consider the following problem, where two servers, $\mathtt{S}_0$ and $\mathtt{S}_1$ hold additive secret shares of an index $\mathtt{i}^*$ and M. Their goal is to obtain secret shares of a 1-hot vector at $\mathtt{i}^*$ with the non-zero value being M. In this paper, we discuss two main ways to achieve this goal.

Local DPF Computation: This method requires a third trusted party, namely, $\mathtt{S}_2$. $\mathtt{S}_2$ produces DPF pairs $(\mathrm{dpf}_0^{(\alpha)}, \mathrm{dpf}_1^{(\alpha)})$ at a random location α and sends $(\mathrm{dpf}_0^{(\alpha)}, \alpha_0^{\mathrm{AS}})$ to $\mathtt{S}_0$ and $(\mathrm{dpf}_1^{(\alpha)}, \alpha_1^{\mathrm{AS}})$ to $\mathtt{S}_1$. The parties then compute $\alpha - \mathtt{i}^*$ and compute Evalfull$(\mathrm{dpf}_b) >>>$

$(\alpha - \mathtt{i}^*)$, where $>>>$ is a cyclic right shift. The protocol requires: (i) $O(1)$ rounds of communication, (ii) $O(\lg n)$ bandwidth consumption, and (iii) $O(n)$ local computation.

MPC-based DPF Computation: Doerner and shelat [10], presented an MPC algorithm to generate a DPF. Their protocol not only produces the DPF but also generates a full domain evaluation of the DPF. We will not go into the details of the algorithm, but would point to the original paper and recent manuscript by Vadapalli, Henry, and Goldberg [32] for an exposition of the protocol. The procol requires: (i) $O(\lg n)$ rounds of communication, (ii) $O(\lg n)$ bandwidth consumption, and (iii) $O(n)$ local computation.

C MPC Protocols

Multiplication Protocol The Du-Atallah protocol begins with a trusted dealer, sampling random values $(X_0, Y_0, X_1, Y_1, \alpha)$. The trusted dealer sends $(X_0, Y_0, X_0 \cdot Y_1 + \alpha)$ to $\mathtt{S}_0$ and $(X_1, Y_1, X_1 \cdot Y_0 - \alpha)$ to $\mathtt{S}_1$. $\mathtt{S}_0$ and $\mathtt{S}_1$ exchange $x_b + X_b$ and $y_b + Y_b$, where $b \in \{0,1\}$. $\mathtt{S}_0$ and $\mathtt{S}_1$ then compute $z_0 \leftarrow (x_0 \cdot (y_0 + (y_1 + Y_1))) - Y_0 \cdot (x_1 + X_1) + (X_0 \cdot Y_1 + \alpha)$ and $z_1 \leftarrow (x_1 \cdot (y_1 + (y_0 + Y_0))) - Y_1 \cdot (x_0 + X_0) + (X_1 \cdot Y_0 - \alpha)$.

Protocol $\mathsf{EQZ}(\cdot)$ The protocol takes in as input, additive shares of a. It outputs additive shares of b, where $b = (a = 0)$.

- The parties locally sample additive shares ${r_0}^{\mathrm{AS}}, \ldots, {r_{k-1}}^{\mathrm{AS}} \leftarrow \Pi_{\mathsf{RandBit}}()$.
- They compute additive shares of $r \leftarrow \sum_{i=0}^{k-1} 2^i r_i$ as $r^{\mathrm{AS}} \leftarrow \sum_{i=0}^{k-1} 2^i {r_i}^{\mathrm{AS}}$.
- The parties compute $c^{\mathrm{AS}} \leftarrow a^{\mathrm{AS}} + r^{\mathrm{AS}}$ and open c to all parties.
- Let $(c_0, \ldots, c_{k-1})$ denote the binary representation of c.
- The parties locally compute additive shares of $b_2 \leftarrow 1 - \prod_{i=0}^{k-1}(c_i + r_i - 2c_i r_i)$.
- The parties invoke Π_{B2A} on ${b_2}^{\mathrm{AS}}$ to obtain the final additive shares b^{AS}, where Π_{B2A} is Boolean to Arithmetic Conversion.

D Security Analysis

Definition 3 (Setup Anonymity)**.** *Let W denote the set of users who participate in the setup phase. Consider an adversary $\mathcal{A}$ that corrupts all but one server, corrupts any strict subset $W_C \subset W$ of users, and observes the network. At the end of the setup phase, let $\mathrm{View}_{\mathcal{A}}$ denote the joint view of $\mathcal{A}$, consisting of:*

- *the published* ACCESSDB.
- *the internal states of the corrupted servers and the corrupted users in W_C.*

We say that the setup phase satisfies setup anonymity *if there exists a probabilistic polynomial-time simulator $\mathcal{S}$ such that $\mathrm{View}_{\mathcal{A}} \approx_c \mathcal{S}(1^\lambda, \mathrm{ACCESSDB}, W_C)$, where λ is the security parameter. In particular, for any two honest writers $w_0, w_1 \in W \setminus W_C$, the adversary cannot determine which database indices or write-authorization proofs are assigned to w_0 versus w_1, except with negligible advantage over random guessing.*

Simulator Construction (for Definition 3). The simulator $\mathcal{S}$ proceeds as follows to generate a view that is computationally indistinguishable from the real execution:

- To simulate a dishonest server, $\mathcal{S}$ encrypts the indices and authorization-proof shares chosen by the adversary using the public key corresponding to the target index.
- To simulate the view of an honest server, $\mathcal{S}$ samples the corresponding authorization-proof share uniformly at random.
- To simulate dishonest writers, $\mathcal{S}$ decrypts the submitted encrypted indices and proof shares using the writers' private keys, thereby recovering the adversary-chosen inputs.
- For honest writers, $\mathcal{S}$ samples a uniformly random value as the decryption of the encrypted index. Since the adversary does not possess the private keys of honest writers, this simulated value is computationally indistinguishable from the real execution.

Definition 4 (Write Authorization)**.** *A* write request *in Remise is a 3-tuple $(\mathtt{i}^{*AS}, M^{AS}, \omega^{AS})$ of additive shares of the target index $\mathtt{i}^*$, a message M, and a purported write-authorization proof ω. Let $\mathrm{PROOFDB}[\mathtt{i}^*]$ denote the write-authorization proof associated with index $\mathtt{i}^*$. We say that Remise satisfies* write authorization *if the following properties hold.*

Correctness. *For each processed write request $(\mathtt{i}^{*AS}, M^{AS}, \omega^{AS})$:*

- *If $M = 0$, then the request has no effect on D.*
- *If $M \neq 0$ and $\omega = \mathrm{PROOFDB}[\mathtt{i}^*]$, then $D[\mathtt{i}^*] \leftarrow M$.*
- *If $M \neq 0$ and $\omega \neq \mathrm{PROOFDB}[\mathtt{i}^*]$, then the request has no effect on D.*

Privacy. *Consider an adversary $\mathcal{A}$ that corrupts all but one server. Let $View_{\mathcal{A}}$ denote the joint view of $\mathcal{A}$ in an execution of Remise. There exists a probabilistic polynomial-time simulator $\mathcal{S}$ such that, for every write request $(\mathtt{i}^*, M, \omega)$ and every database state $(D, \text{PROOFDB})$,*

$$View_{\mathcal{A}} \approx_c \mathcal{S}(1^\lambda, \mathsf{out}),$$

where λ is the security parameter and $\mathsf{out} \in \{\mathsf{accept}, \mathsf{reject}\}$ indicates whether the write is authorized, i.e., whether $M \neq 0$ and $\omega = \text{PROOFDB}[\mathtt{i}^*]$. In particular, except for the leakage of out*, the adversary learns no additional information about $\mathtt{i}^*$, M, or ω.*

Simulator Construction (for Definition 4). Given input $(1^\lambda, \mathsf{out})$, the simulator $\mathcal{S}$ proceeds as follows:

- **Random sampling.** $\mathcal{S}$ samples the following values independently and uniformly at random: (i) an index $\mathtt{i}^{*\prime} \in [|\mathrm{D}|]$, (ii) an additive share of a message, and (iii) an additive share of a write-authorization proof. Since the corrupted servers only observe uniformly random additive shares, the resulting simulated view is perfectly indistinguishable from the view in a real execution, regardless of whether $\mathsf{out} = \mathsf{accept}$ or $\mathsf{out} = \mathsf{reject}$.
- **First zero-test simulation.** $\mathcal{S}$ invokes the simulator to generate the adversary's view of the first execution of the EQZ protocol.
- **Read-phase simulation.** $\mathcal{S}$ invokes the simulator to generate the adversary's view of the Distributed Oblivious Read protocol.
- **Second zero-test simulation.** $\mathcal{S}$ invokes the simulator to generate the adversary's view of the second execution of the EQZ protocol.
- **Multiplication simulation.** $\mathcal{S}$ invokes the simulator to generate the adversary's view of the multiplication protocol.
- **Conditional update.** If $\mathsf{out} = \mathsf{accept}$, $\mathcal{S}$ invokes the simulator to generate the adversary's view of the Distributed Oblivious Update protocol. Otherwise, if $\mathsf{out} = \mathsf{reject}$, $\mathcal{S}$ aborts.

Definition 5 (Read Authorization)**.** *In RemiseCC, each index $\mathtt{i}^*$ has an associated list $\mathcal{T}[\mathtt{i}^*]$ of m read-authorization tokens. A* read request *is a 2-tuple $(\mathtt{i}^{*AS}, t^{AS})$ of additive shares of the target index $\mathtt{i}^*$ and a purported read-authorization token t.*

We say that RemiseCC satisfies read authorization *if the following properties hold.*

Correctness. *For each processed read request $(\mathtt{i}^{*AS}, t^{AS})$:*

- *If $t \in \mathcal{T}[\mathtt{i}^*]$, then the reader receives $D[\mathtt{i}^*]$.*
- *If $t \notin \mathcal{T}[\mathtt{i}^*]$, then the reader receives no information about $D[\mathtt{i}^*]$.*

Privacy. *Consider an adversary $\mathcal{A}$ that corrupts all but one server. Let $View_{\mathcal{A}}$ denote the joint view of $\mathcal{A}$ in an execution of RemiseCC's Read Authorization. There exists a probabilistic polynomial-time simulator $\mathcal{S}$ such that, for every read request $(\mathtt{i}^*, t)$ and every database state $(D, \mathcal{T})$,*

$$View_{\mathcal{A}} \approx_c \mathcal{S}(1^\lambda, \mathsf{out}),$$

where λ is the security parameter and $\mathsf{out} \in \{\mathsf{accept}, \mathsf{reject}\}$ indicates whether $t \in \mathcal{T}[\mathtt{i}^*]$. In particular, except for the leakage of out*, the adversary learns no additional information about $\mathtt{i}^*$, t, or $D[\mathtt{i}^*]$.*

Simulator Construction (for Definition 5). Given input $(1^\lambda, \mathsf{out})$, the simulator $\mathcal{S}$ proceeds as follows:

- **Random sampling.** $\mathcal{S}$ samples the following values independently and uniformly at random: (i) an index $\mathtt{i}^{*\prime} \in [|\mathrm{D}|]$, and (ii) an additive share of a read--authorization token. Since the corrupted servers only observe uniformly random additive shares, the resulting simulated view is perfectly indistinguishable from the view in a real execution. In particular, the simulated view is independent of $\mathtt{i}^*$, the authorization token t, and the database value $\mathrm{D}[\mathtt{i}^*]$, and depends only on the value of out.
- **Read-phase simulation.** $\mathcal{S}$ invokes the simulator to generate the adversary's view of the Distributed Oblivious Read protocol.
- **Zero-test simulation.** $\mathcal{S}$ invokes the simulator to generate the adversary's view of the EQZ protocol m times.
- **Multiplication simulation.** $\mathcal{S}$ invokes the simulator to generate the adversary's view of the multiplication protocol m times.
- **Conditional update.** If $\mathsf{out} = \mathsf{accept}$, $\mathcal{S}$ runs a simulation of the Distributed Oblivious Read Protocol, if $\mathsf{out} = \mathsf{reject}$, $\mathcal{S}$ aborts.

Definition 6 (Access Update Privacy)**.** *Let RemiseCC be the access-update protocol maintaining a token database $\mathcal{T}$ and a write-authorization database* PROOFDB*. A* read-access update request *is a 3-tuple $(\mathtt{i}^{*AS}, \omega^{AS}, R^{AS})$, where these are additive shares of the target index $\mathtt{i}^*$, a*

purported write-authorization proof ω*, and a new token list* R*, respectively. We say that RemiseCC satisfies* access update correctness and privacy *if the following properties hold.*

Correctness. *Consider any update request denoted as* $(\mathtt{i}^{*AS}, \omega^{AS}, R^{AS})$*:*

- *If* $\omega = \textsc{ProofDB}[\mathtt{i}^*]$*, then* $\mathcal{T}[\mathtt{i}^*] \leftarrow R$*.*
- *Otherwise, the request is rejected and* $\mathcal{T}$ *remains unchanged.*

Privacy. *Consider an adversary* $\mathcal{A}$ *that corrupts all but one server and controls the network. Let* $View_{\mathcal{A}}$ *denote the joint view of* $\mathcal{A}$ *in an execution of RemiseCC, consisting of the internal states of corrupted servers, all messages sent or received by them, and the adversary's randomness. There exists a probabilistic polynomial-time simulator* $\mathcal{S}$ *such that, for every update request* $(\mathtt{i}^*, \omega, R)$ *and every initial database state* $(\mathcal{T}, \textsc{ProofDB})$*,*

$$View_{\mathcal{A}} \approx_c \mathcal{S}(1^{\lambda}, \mathsf{out}),$$

where λ *is the security parameter and* $\mathsf{out} \in \{\mathsf{accept}, \mathsf{reject}\}$ *is the outcome of the authorization check. In particular, the distribution of* $View_{\mathcal{A}}$ *depends only on* out *and the public parameters, and is independent of* $\mathtt{i}^*$*,* ω*, and* R*.*

Simulator Construction (for Definition 6)

- **Random sampling.** $\mathcal{S}$ samples the following values independently and uniformly at random: (i) an index $\mathtt{i}^{*\prime} \in [|\mathrm{D}|]$, and (ii) an additive share of a read--authorization token. Since the corrupted servers only observe uniformly random additive shares, the resulting simulated view is perfectly indistinguishable from the view in a real execution. In particular, the simulated view is independent of $\mathtt{i}^*$, the authorization token t, and the database value $\mathrm{D}[\mathtt{i}^*]$, and depends only on the value of out.
- **Read-phase simulation.** $\mathcal{S}$ invokes the simulator to generate the adversary's view of the Distributed Oblivious Read protocol.
- **Zero-test simulation.** $\mathcal{S}$ invokes the simulator to generate the adversary's view of the EQZ protocol.
- **Conditional update.** If $\mathsf{out} = \mathsf{accept}$, $\mathcal{S}$ invokes the simulator to generate the adversary's view of the Distributed Oblivious Update protocol. Otherwise, if $\mathsf{out} = \mathsf{reject}$, $\mathcal{S}$ aborts.

E Remise Extensions

Hiding whether it is a read or write request Observe that it is not very difficult to construct RemiseCC so that the servers are oblivious to whether it is a read request or a write request. The following changes have to be made: (i) requests are always a 3-tuple $(\mathtt{i}^*, \textsc{proof}^{\mathrm{AS}}, \mathtt{M}^{\mathrm{AS}})$. If it is a read request, the client shares a read authorization token, and if it is a write request, a write proof is shared. Additionally, if it is a read request, the reader sets $\mathtt{M} = 0$, and (ii) any request is processed as both a write and a read request. The servers always return $(\mathrm{ReadAuthCheck}(\textsc{proof})) \cdot \mathrm{D}[\mathtt{i}^*]$ and the write is always $\mathrm{D}[\mathtt{i}^*] + (\mathrm{WriteAuthCheck}(\textsc{proof})) \cdot \mathtt{M}$.

Robust Remise Robustness in Remise can be easily achieved by introducing multiple server pairs. For example, consider $2 \cdot k$ main servers arranged as k pairs: $(\mathtt{S}_0^{(1)}, \mathtt{S}_1^{(1)}), \ldots, (\mathtt{S}_0^{(k)}, \mathtt{S}_1^{(k)})$. If each request is sent to all k pairs, then Remise can tolerate multiple server failures as long as at least one pair has both servers remaining operational.

$(2+1)$-PC Remise The paper considers a 2-party Remise. Remise incurs $O(\lg n)$ rounds in the preprocessing phase. These $O(\lg n)$ are due to the Doerner-shelat MPC DPF generation. However, in RemiseBB if we have an additional third party, the DPF can be produced in the by the third party locally and sent to the two parties incurring just one round of communication. This procedure is described in Appendix B in the paragraph Local DPF Generation. We cannot get this advantage in RemiseCC, since, it uses DORAM that should support both read and write operations and this would still require $O(\lg n)$ rounds even with the third party [32].