Kerblam — Anonymous Messaging System Protecting Both Senders and Recipients

An anonymous post-office without mailboxes

Yanxue Jia, *Purdue University*
Debajyoti Das, *KU Leuven*
Wenhao Zhang, *Northwestern University*
Aniket Kate, *Purdue University / Supra Research*

***Abstract*—While popular messaging apps already offer end-to-end confidentially, end-to-end metadata privacy is still far from being practical. Although several meta-data hiding systems have been developed and some like Tor have been popular, the proposed solutions lack in one or more aspects: the Tor network is prone to easy low-resourced attacks, and most others solely focus on anonymity for senders or receivers but do not both. Some recent solutions do consider end-to-end anonymity, however, they put significant restrictions on how users use the system. Particularly, the receivers must stay online or trust online servers that receive messages on behalf of receivers.**

This work presents a scalable end-to-end anonymity messaging system, Kerblam, that overcomes the mentioned issues and restrictions. It stems from a key observation that combining the recently-emerged oblivious message retrieval (OMR) primitive with oblivious shuffling can offer the desired end-to-end anonymity without severely restricting the number of messages a sender may send or a receiver may receive. We build our solution using two non-colluding servers and recent OMR protocol HomeRun and a compatible oblivious shuffle protocol. We then extend our solution to allow larger messages by employing a novel two-server distributed oblivious RAM technique, called ORAM^{-}. Our performance analysis demonstrates that with the increase in the number and size of messages, the performance improvement brought by ORAM^{-} becomes higher. Specifically, for 2^{20} messages of size 1 KB, our scheme only needs 5.577 s to transmit a message.

1. Introduction

Many popular communication applications, such as Signal and WhatsApp, employ end-to-end encryption to safeguard message content between parties. However, communication metadata—such as who communicated with whom and when [1]—can still lead to severe consequences.

Extensive research (e.g., [2], [3], [4], [5], [6]) has already been conducted to protect communication metadata. Specifically, communication metadata could be abstracted as the linkability between a sender and a message, and/or the linkability between a message and a receiver. Many prior works either break the linkability between the message and the sender [7], [8], [9], or between the message and the receiver [10], but not both. In this work, we aim to break the linkability on both sides to protect the privacy of the sender as well as the receiver, and we call this unlinkability End-to-End Unlinkability (EE-UL), which is the same as "Both-Side Message Unlinkability" defined in [11].

There are *mailbox-based* systems [3], [12], [13], [14] that attempt to hide the link between the sender-receiver pair in an active group of clients who send messages in batches. They either require the sender and the receiver to trust each other (weaker security) and/or an expensive dialing protocol [15] to set up the mailboxes. Such requirements limit the application scenarios where those systems can be used, especially for asynchronous communications (when the sender wants to send messages without any synchronization or setup with the corresponding receiver or to a receiver who is not active at the time).

There are a few designs that attempt to protect the privacy of both the sender and the receiver by mixing the messages. In this way, both the sender-message linkability and message-receiver linkability are broken. However, they suffer from various drawbacks:

- Many application scenarios (e.g., messaging, payments) require to support *asynchronous retrieval*, which means that a sender can send a message to a receiver even if the receiver is not currently online, and the receiver can retrieve the message whenever it comes online. However, the current approaches cannot naturally support it. To solve this issue, Loopix [16] and AnonPOP [17] rely on online servers (called Gateways in the Nym deployment [18] of Loopix) to receive messages on the behalf of receivers. Such strategies require additional trust assumption on such Gateways and still can become vulnerable to intersection and traffic-analysis attacks [19].
- Tor [2], with its onion service functionality, can provide protection for both the sender and receiver side. It is extremely popular for its low latency and low bandwidth overhead. However, there have been a large number of works (e.g., [20], [21], [22], [23]) that successfully attacked Tor using low resources. Moreover, Tor cannot support asynchronous message retrieval.

Application Scenarios of End-to-End Unlinkability (EE-UL). There are many scenarios where the privacy of both the sender and the receiver is equally important. Specifically, as mentioned in [24], "anonymous social media platforms have gained popularity as havens for individuals seeking to express themselves freely without fear of judgment

or exposure." On anonymous social media platforms, each user has an anonymous account (namely, an address). Users can communicate with each other without leaking their real-world identifications. Breaking linkability only either on the sender or on the receiver side cannot satisfy the privacy requirements of such anonymous social media platforms.

Another important application scenario is achieving privacy for transactions on Blockchains. With the property of EE-UL the payment recipient could provide only a wallet address to protect their real identity. At the same time, the sender can benefit from hiding their identity, even from the wallet owner, and avoiding revealing association with the wallet or the possibility of being profiled/blackmailed.

1.1. Our Contribution

In this work, we design an anonymous messaging system, called Kerblam, which achieves the End-to-End Unlinkability (EE-UL), while avoiding the above drawbacks. We summarize our contributions below.

End-to-End Anonymity without Limitations. We design an anonymous messaging system, called Kerblam, that enjoys stronger anonymity, protecting the privacy of both sender and receiver. Kerblam is useful for scenarios where the sender and the receiver not only desire to remain anonymous to others but also from each other. At the same time, we depart from batch-mixing and round-based designs. As a benefit, we can easily allow sending and receiving messages asynchronously. We formally analyze the security of our scheme against global passive adversaries, as well as protect against relevant active attacks.

Highly Scalable Protocol. We observe that combining a fully-fledged oblivious message retrieval (OMR) protocol and oblivious shuffling protocol can obtain end-to-end anonymity without limitations. However, it is not scalable enough to support long messages (e.g., of size 1 KB). To solve this problem, we introduce a new variant of ORAM, called ORAM^-, which is a weaker version of standard ORAM but is enough for communication scenarios. By introducing ORAM^-, we improve the performance for 2^{20} messages of size 1 KB by a factor of 7 (please refer to Table 3 for more details). When there are 2^{20} messages stored by the servers, and each message is of size 1 KB stored, the wall-clock time cost of transmitting a message is 38.942 s using a single thread and 5.577 s using 16 threads.

Novel Variant of ORAM: ORAM^-. Our design of ORAM^- could be of independent interest to scale other designs. We provide a general construction of ORAM^- based on tree-based ORAM, and it is particularly suitable for metadata-private message transmission systems. Compared to the standard ORAM, the main differences are (1) "write" and "read" operations are allowed to be distinguished, (2) block ID can be randomly chosen from a large space, and (3) the total number of blocks is allowed to vary. These adaptations allow significant performance for Kerblam (c.f. Table 3). We give a new formal definition for ORAM^- and show that our construction is provably secure under our definition.

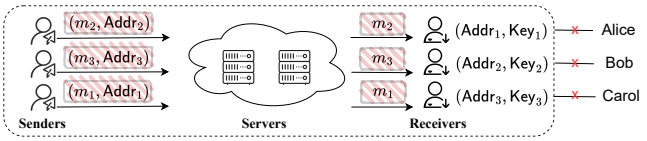


Figure 1: System Overview. We have two servers. Each receiver is associated with an address/key pair, and the address will not leak the receiver's real identity. When a sender wants to send a message m_i to the receiver associated with address Addr_i. The sender sends (m_i, Addr_i) in an "encrypted" version to the two servers; the receiver uses Key_i to retrieve the message m_i without leaking $(m_i, (\mathsf{Addr}_i, \mathsf{key}_i))$.

1.2. Related Work

Based on recent works [1], [11], anonymity notions can be categorized into two main (non-exclusive) types: (1) privacy for senders, and (2) privacy for receivers. Different protocols can provide either or both of these properties with different anonymity levels; Our design achieves both, and we call it End-to-End Unlinkability. Below we discuss different design paradigms of anonymous messaging systems and the kinds of anonymity guarantees they can provide.

Mixnets. Mixnet-based designs [7], [9], [25], [26] can achieve sender anonymity for anonymous broadcast by shuffling the messages through several layers of mixing nodes.

Some mixnets [13], [27], [28] deviate from anonymous-broadcast setting and introduce *mailboxes* where the sender-receiver pair of a message needs to agree on a shared secret. Although, these systems attempt to hide the link between the sender and the receiver from a third-party observer, the sender and receiver need to trust each other or need to use an expensive *dialing* protocol to set up mailboxes. This additional trust requirement makes the anonymity guarantee strictly weaker than either sender anonymity or receiver anonymity [11]. These designs implement mixing in batches, and that adds several restrictions: (1) a sender can only send messages to online receivers, (2) the anonymity set is limited to a batch.

To achieve receiver-side privacy, and to allow the receivers to retrieve messages *asynchronously*, the Nym [18] deployment of Loopix [16] introduces Gateway servers: A Gateway server can collect the messages on behalf of the receivers, and the receiver can retrieve messages when they come online. They additionally employ cover traffic and rate-limiting for download/retrieval to obfuscate the volume metadata. Unfortunately, without the employment of a strong Oblivious Message Retrieval (OMR) protocol, the system is vulnerable to intersection and traffic-analysis attacks [19] with its current deployment parameters. Provable mixing guarantees for Loopix-like systems only exist for sender anonymity and would require really expensive latency overheads [29].

MPC-based Designs. There are MPC-based designs [6], [14], [30], [31] that realize some version of secure shuffle or *private-writing* to achieve guarantees similar to sender anonymity. Express [3] additionally employs *mailboxes*,

however, deviates from the requirements of batch-processing and the sender-receiver trust requirement. Express can only hide the link between the sender and the mailbox, but does not hide who reads from the mailbox: in that sense, Express only achieves sender anonymity.

Other designs like HomeRun [10], Private Signaling (two-server version) [32] realize Oblivious Message Retrieval (OMR) to achieve strong privacy for receivers, but they cannot protect the privacy for senders.

Our design Kerblam employs MPC in a two-server setup: in terms of system setup, Kerblam is identical to Express [3], HomeRun [10], and Private Signaling (two servers) [32]. However, in contrast with other MPC-based designs, it achieves both sender anonymity and receiver anonymity, where the sender and the receiver might not even trust each other, by combining secure shuffle and OMR techniques.

Other Not-so-related Anonymous Messaging Systems. Tor [2] is really popular for its low latency and low bandwidth overhead; and can provide anonymity for both sides with its *onion service* functionality. However, such low-latency and low-bandwidth networks can achieve only weak anonymity guarantees [33], [34], and there are many demonstrated attacks [20], [21], [22], [23] against Tor.

Dining-cryptographer network (DC-net) based systems [8], [35], [36] can achieve strong sender anonymity for anonymous broadcast, however, they cannot provide privacy for the receiver-side.

PW-Panda and SW-Panda [37] achieve properties somewhat similar to our ORAM^{-}. However, in PW-Panda, the creator of a message cannot restrict who reads the message. On the other hand, in SW-Panda, a client can access only their own data, unless there are some explicit key-sharing mechanisms among clients. Because of such limitations, and its use of expensive primitives like fully-homomorphic encryptions, it is not straightforward to adopt such systems to design end-to-end anonymous messaging systems.

Organization. We give the technical overview in Section 2, including the system setting and key ideas. In Section 3, we recollect the building blocks used in our work. Then in Section 4, we give the formal description of our scheme. The security analysis and performance evaluation can be found in Section 5 and Section 6 respectively. Section 7 provides the concluding remarks.

2. Technical Overview

2.1. System Setting

As shown in Figure 1, there are two servers that assist the users with communications. Each receiver is associated with an address and the corresponding secret key, and we assume that the potential senders can obtain the address without knowing the real-world identity of the receiver. For example, the address could be an account on a social media platform, and other users on the social media platform can find the address but do not know the identity behind the address.

Function Goals. When a sender wants to send a message m_i to a receiver associated with Addr_i, the sender sends an "encrypted version" of (m_i, Addr_i) to the two servers. Then the receiver sends a retrieval request to the two servers, proving the possession of the corresponding secret key Key_i and without leaking Addr_i. Finally, the two servers send back the message m_i in an "encrypted" form to the receiver. We only assume that the total number of unretrieved messages is up to a predetermined value N, but do not limit the number of messages waiting to be retrieved by a certain receiver.

Security Goals. Our work aims to protect the privacy of both senders and receivers. This means that even though the sender colludes with one of the servers, they cannot find which party retrieved the message, and vice versa. The formal security definition is given in functionality $\mathcal{F}_{anon}$ (see Figure 12).

Our protocol is suitable for situations where the sender and receiver do not know each other and do not want to reveal their identities to each other. "Address" is the identity in our system, and sending a message to a receiver refers to sending the message to an address. We assume that the address will not leak the real-world identity, and the adversary can recognize the entity sending messages to the two servers, e.g., via IP addresses. We do not protect who is sending or retrieving messages, but we do protect the linkability between messages and entities.

Trust Assumption. We assume that the two servers are non-colluding and will not have detectable malicious behaviors, as in the previous work [3], [10], [32]. We explain how we defend against malicious behaviors in Section 5.3.2. We assume senders and receivers can be malicious and can collude with one of the two servers.

2.2. Key Idea

Our work starts with a key observation that combining an oblivious message retrieval (OMR) and an oblivious shuffling can obtain an end-to-end anonymous messaging system. However, a direct combination has poor scalability and cannot support long messages on large scales. To address this issue, we introduce a new variant of ORAM, called ORAM^{-}, and use distributed ORAM^{-} to achieve scalability. Next, we will first introduce the key observation and explain how to achieve scalability.

The previous work either does not support asynchronous retrieval or relies on mailbox-based methods that limit the number of messages a receiver can receive. The recent research [10], [38], [39] on Oblivious Message Retrieval (OMR) has gradually overcome these limitations. Moreover, we observe that OMR can be changed to achieve end-to-end anonymity while keeping the merits.

OMR was proposed to allow the receiver to retrieve her transactions submitted on blockchains without leaking which messages are retrieved. Therefore, OMR protects the receiver's privacy. It can also be used for messaging systems without blockchains, by allowing the senders directly to send messages to the server(s). However, since the receiver knows

which message is retrieved and the server(s) knows who is the sender of each message, if the receiver colludes with the server or one of the servers, they can collaboratively find who sent the message. Therefore, the key to further protecting the sender's privacy is to guarantee that the server(s) does not know the sender of each message.

To this end, we obliviously shuffle (i.e., the permutation is not known by anyone, and please see Figure 5 for more details) the list(s) stored on the server(s), before providing information to the receiver. In this way, for a retrieved message, the server(s) cannot link it to its sender. The OMR works [10], [38], [39] all can be changed to achieve end-to-end anonymity through the way. However, to date, there are still no efficient single-server oblivious shuffling protocols, and the single-server OMRs [38], [39] are also not efficient enough. Therefore, we choose HomeRun [10], the two-server OMR scheme, as a basis to achieve our end-to-end anonymous messaging system.

2.3. Basic Construction

Next, we first briefly recall HomeRun and then give our basic construction.

Recall HomeRun. The design framework is shown in Figure 2. Each server maintains a list consisting of pairs, with each pair containing a label and a message. The label is used to identify whether a message is pertinent to the receiver requesting retrieval. The flow of HomeRun is as follows.

- Sending:
  - The sender that intends to send the message m, generates labels L and L' according to the address of the receiver, and sends (L, m) and (L', m) to Server_1 and Server_2, respectively;
  - The two servers append (L, m) and (L', m) to their own lists, respectively.
- Retrieval:
  - The receiver generates two labels L_r and L'_r using the secret key of her address, and sends L_r and L'_r to Server_1 and Server_2, respectively;
  - The two servers interact with each other to generate two bit-vectors, $\vec{b}^{(1)}$ and $\vec{b}^{(2)}$, from which the receiver can recover the pertinent indexes (if $\vec{b}^{(1)}[i] + \vec{b}^{(2)}[i] = 1$, i is a pertinent index);
  - The receiver uses the pertinent indexes to retrieve the pertinent messages from the two servers through Private Information Retrieval (PIR).

HomeRun does not consider the privacy of senders and does not hide the content of messages from the servers (as the messages are published on the blockchain[1]). Next, we will explain how to achieve EE-UL while protecting the content of messages based on HomeRun.

Our Basic Scheme. To further hide the content of each message, the sender splits the message m into two shares $[m]_1$ and $[m]_2$ such that $m = [m]_1 + [m]_2$. Then, the sender sends $(L, [m]_1)$ and $(L', [m]_2)$ to the two servers, respectively. In this case, each server cannot learn the message m, and the lists maintained by the two servers are changed to those shown in Figure 3a. In our protocol (see Section 4), the labels L and L' are instantiated by address shares and associated strings. For simplicity, we just use the concept of label here.

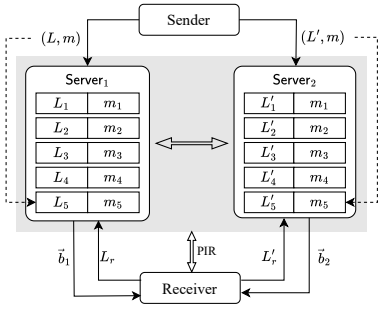
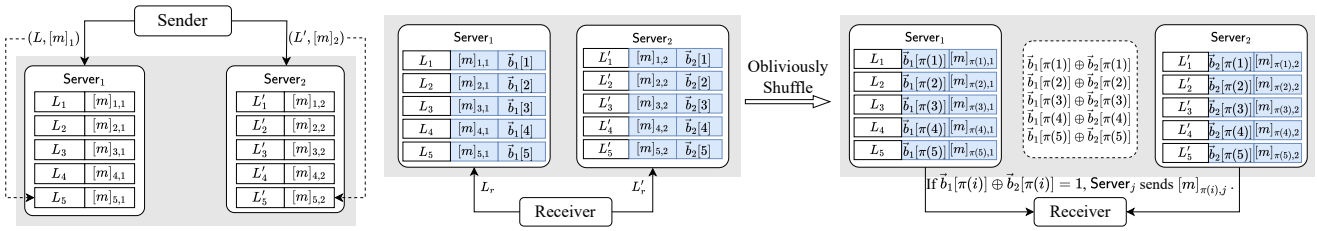


Figure 2: Design Framework of HomeRun [10]. For each message, each server stores a label share. Then, the receiver sends request label shares L_r and L'_r to the two servers respectively. The two servers compute $\vec{b}_1$ and $\vec{b}_2$ and send back the two-bit vectors to the receiver. The receiver recovers the pertinent indexes using $\vec{b}_1$ and $\vec{b}_2$, and then uses these indexes to retrieve the messages via PIR.

Then, we leverage the oblivious shuffling $\mathcal{F}_{\mathsf{Shuffle}}$ (see Figure 5) to protect the privacy of the sender. Specifically, as shown in Figure 3b, after receiving the labels L_r and L'_r from the receiver, respectively, the two servers first generate the two-bit vectors, $\vec{b}_1$ and $\vec{b}_2$, respectively, as in HomeRun. Instead of sending the bit vectors to the receiver, the two servers obliviously shuffle the bit vectors and the message vectors using the same permutation that is not known by anyone. Then, the two servers can interact with each other to recover the pertinent indexes. Note that, different from HomeRun, the two servers here do not know the original indexes, so the two servers can open the pertinent indexes by themselves. Finally, the two servers send the corresponding shares of the pertinent messages to the receiver, without using PIR.

It is easy to see that after the obliviously shuffling, neither the receiver nor the servers can determine which sender provided the pertinent messages, and thus the privacy of the sender is protected. On the other hand, the privacy of the receiver is still protected, as the sender and the servers cannot know which original messages are retrieved by the receiver. Note that we are not intending to protect the number of messages retrieved by a receiver. In addition, different from the existing works in Type 3, we mix a message among all the messages maintained by the two servers, rather than only the messages in a round. The efficiency of our work benefits from the recent work [40] by Peceny et al. This work designed a new oblivious shuffling protocol with high performance (please see Table 1 for more details).

1. The messages on blockchain may also be some ciphertexts.



(a) Sending (b) Retrieval

Figure 3: Basic Scheme for Short Messages. The two servers store the label shares and message shares. After receiving a request (including L_r or L'_r) from a receiver, the two servers collaboratively compute vectors $\vec{b}_1$ and $\vec{b}_2$. Then, the two servers obliviously shuffle the message share list and bit list using the same permutation π. At last, the two servers recover $\vec{b}_1[\pi(i)] + \vec{b}_2[\pi(i)]$, if $\vec{b}_1[\pi(i)] + \vec{b}_2[\pi(i)] = 1$, the server sends the corresponding message share $[m]_{\pi(i),j}$ to the receiver.

However, the communication and computation complexity of the oblivious shuffling designed by Peceny et al. [40] is $O(n\ell)$. Therefore, if the message size ℓ and the number of messages n are both large (e.g., $\ell = 1$ KB and $n = 2^{20}$), the performance of this basic scheme will drastically decrease. To solve this problem, we additionally introduce a variant of ORAM, such that the basic scheme is only used for providing indexes, and messages are maintained by ORAM^-.

2.4. Scaling via a New Variant of ORAM

As mentioned before, the above basic scheme is not efficient enough for a significant number of long messages. To avoid the communication and computation cost of $O(n\ell)$ for each retrieval, we leverage a variant of ORAM structure to maintain the messages. Specifically, we change the message shares in Figure 3 to index shares, and then the two servers use the index shares to retrieve the corresponding messages from a variant of ORAM. In this way, the oblivious shuffling only involves the index, which is much shorter than messages. Next, we will detail the variant of the ORAM structure.

In cloud storage scenarios [41], [42], [43], a client stores their data on a remote server and will access their data later from the server. To hide the access pattern of the client, ORAM was proposed. Therefore, originally, ORAM was used for a client who owns the data and does not have the motivation to destroy the data, and the access pattern includes an arbitrary number of read and write operations. Please see Definition 1 and Definition 2 for more details of ORAM. Compared with the original ORAM, the variant needed in this work has the following differences:

- (D1): Senders and receivers are ordinary users in the system, and thus may misbehave. Therefore, we need an ORAM structure that can support malicious clients.
- (D2): There are many senders and receivers that need to access the messages. Therefore, we need an ORAM structure that can support multiple clients.
- (D3): We are considering an anonymous messaging system where it is not necessary to hide whether access involves sending or retrieving data from the servers. Sending and retrieval correspond to the write operation and the read operation in ORAM, respectively. Therefore, we do not need the ORAM structure to hide the type of access.
- (D4): Our anonymous messaging system only considers that a sender sends a message to a receiver, and thus a message can only be read (i.e., retrieved) once after it is written (i.e., sent) into the ORAM structure. Therefore, the access pattern for a block in the ORAM structure is the sequence $\{\mathsf{write}, \mathsf{read}\}$. Combined with the above D3, we know that the variant of ORAM just needs to break the linkability between the write operation and the read operation.

Why Choose ORAM in Secure Computation Setting. ORAM in the malicious multi-client setting was first considered and formalized by Maffei et al. [44]. Later, Chen et al. [45] significantly improved the prior work by letting two servers emulate all operations that were previously taken by the client; therefore, during an operation, a client only needs to secretly share the corresponding index with these two servers, and no malicious behavior can be performed by a client. In this work, we use the same idea to support multiple malicious clients. This idea essentially follows the concept of ORAM in the secure computation setting which is first studied by Gordon et al. [46]. We refer to Section 3.1 for more details.

One approach towards realizing ORAM in the secure computation setting, denoted as ORAM-SC, is to design an ORAM scheme with a lower circuit complexity on the client side and then realize it by letting servers jointly emulate all client operations. Among all ORAM schemes including hierarchical ORAM [47], square-root ORAM [47] and tree-based ORAM [48], [49], Circuit ORAM, a tree-based ORAM, proposed by Wang et al. [50] shows an optimal circuit complexity both asymptotically and practically. A recent generation of work for ORAM-SC [51], [52], [53] derives from the function secret sharing [54], [55], and uses a list to store data. These protocols have low round and communication complexity but a linear computational complexity. As a result, these protocols show a better practical performance in some specific settings even though their asymptotic complexity is worse than Circuit ORAM.

Next, we will explore the application of the tree-based ORAM-SC within our anonymous messaging system. In

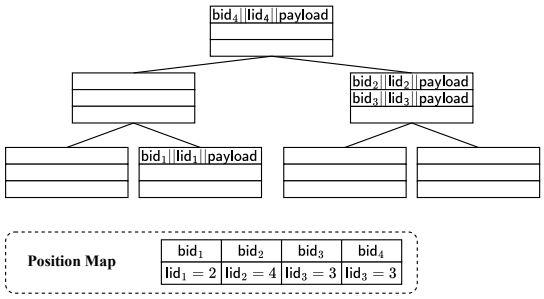


Figure 4: Tree-based ORAM

Appendix A, we also explain why we do not choose the list-based ORAM-SC in our system. In a secure computation setting, the ORAM structure (e.g., the tree and the list) would be secretly shared between the two servers. Later, for simplicity in our explanation, we just use the original structure, but please keep in mind that the data structure is actually stored in a secret sharing manner, and all the operations are performed in a secure computation way.

Use Tree-based ORAM-SC**.** In the original ORAM (no matter tree-based ORAM or list-based ORAM), each block is associated with a block ID , denoted as bid, and there is a fixed set B containing n block IDs. Each access needs to specify a block ID in the set B. In our anonymous communication system, the sender needs to insert his message into the ORAM structure, i.e., write a message into a block. Since we aim to utilize the ORAM structure to store up to n unretrieved messages, while allowing senders to independently select their own block IDs from the set B, it is highly likely that multiple senders will choose the same block ID. Consequently, their messages will not be correctly written into the ORAM. To solve this issue, we allow the senders to choose block IDs from a larger set $\tilde{B}$. Specifically, when we aim to store 2^{20} unretrieved messages in the ORAM, we can set the bit-length of each block ID as $\ell_{\mathsf{bid}} = 80$ and the set $\tilde{B}$ contains 2^{80} block IDs. Then, according to the birthday paradox probability, the collision probability is $1 - e^{-\frac{n^2}{2 \cdot 2^{\ell_{\mathsf{bid}}}}} < 2^{-40}$.

While we solved logical collisions above, we also need to solve physical collisions. In other words, although the probability that multiple senders choose the same block ID is negligible, the number of blocks maintained by the tree-based ORAM is much smaller than $2^{\ell_{\mathsf{bid}}}$, and thus there may be physical collisions. Next, we first recall the data structure in the tree-based ORAM, and then explain how to leverage it to eliminate physical collisions.

We show the data structure of the tree-based ORAM in Figure 4, assuming that there are 4 blocks indexed by $\{\mathsf{bid}_1, \mathsf{bid}_2, \mathsf{bid}_3, \mathsf{bid}_4\}$. In the tree-based ORAM, each block is also associated with a path identified by a leaf ID lid. For a block, after being accessed, the block would be associated with a new randomly chosen path and put into the root node[2]. Later, the block would be evicted from the root node to the leaf node through the new path. When using the tree-based ORAM in our anonymous messaging system, for each new message msg, we need to assign a block ID bid_m and a path lid_m for it and then write $\mathsf{bid}_m||\mathsf{lid}_m||\mathsf{msg}$ into the root node as a block. The block ID can be chosen by the sender using the above way. As for the path lid_m, according to the security analysis for the original tree-based ORAM, we only need to ensure that the paths are selected uniformly at random, without needing to ensure that each block corresponds to a different path (e.g., in Figure 4, the blocks indexed by bid_3 and bid_4 share the same path indexed by $\mathsf{lid}_3 = \mathsf{lid}_4 = 3$). Therefore, we do not need to consider the collision problem for the selection of paths. However, to guarantee that paths are selected uniformly at random, we cannot allow senders to choose paths, as senders may behave maliciously. Instead, we require the two servers to securely choose a random path, with neither server knowing the path. Based on the analysis above, when using tree-based ORAM, there is no need to consider any additional collision issues beyond block IDs, and thus will not incur additional costs compared to the original tree-based ORAM.

Simplify Tree-based ORAM-SC**.** At this point, we have chosen the tree-based ORAM-SC to support for malicious multiple clients, corresponding to D1 and D2. Furthermore, due to D3, D4 and our basic scheme for providing indexes, we can further simplify the tree-based ORAM-SC, and obtain our variant ORAM$^-$.

First, we do not need to hide the type of access (according to D3), and thus we allow different treatments for "read" and "write" operations. In the original ORAM-SC, regardless of whether the access is "write" or "read", the servers need to perform the following three steps:

- Step-1: Retrieve the queried block and empty the corresponding position;
- Step-2: Add the queried block into the root node (if the access is "write", write the new payload to the block);
- Step-3: Perform an eviction process.

Second, in our anonymous messaging system, we assume that a message is sent from a sender to a receiver, and thus the access pattern for a block is just a sequence $\{\mathsf{write}, \mathsf{read}\}$ (according to D4). Therefore, the sending operation in our anonymous messaging system only involves inserting the new message into the ORAM structure, without needing to re-write an existing block. Therefore, we only need to perform Steps 2 and 3 for the writing. In addition, after a message is retrieved (i.e., "read"), we only need to set the corresponding block as a dummy block. In other words, we only need to perform the above Step-1 for reading.

Furthermore, in the original ORAM-SC, the position map needs to be maintained by a recursive process such that the client-side storage cost is just $O(1)$. In our anonymous messaging system, the lists containing block IDs and path IDs are actually the position map, and the clients do not need to maintain them. Therefore, we do not need a recursive process to query the position map.

2. Here, we ignore the stash for simplicity.

3. Preliminary

3.1. Oblivious RAM

Encryption techniques can protect the content of data outsourced to an untrusted server, but cannot hide the access pattern, which also leaks sensitive information about the client. Oblivious RAM (ORAM) was introduced by Goldreich and Ostrovsky [47] to hide the client's access pattern from the untrusted server. Informally, given two equal-length sequences of operations (including read and write) on ORAM, the server cannot distinguish the two sequences according to the physical access sequences. Next, we give the definition provided by [56].

Definition 1 (ORAM). *An* $\mathsf{ORAM} = (\mathsf{Init}, \mathsf{Query})$ *includes the following two polynomial-time algorithms:*

- $O \leftarrow \mathsf{Init}(A, N)$*: Initialize an ORAM object* O *containing an array* A *of length* N *whose elements are from some space* $\mathcal{M}$*.*
- $m' \leftarrow \mathsf{Query}(O, i, m)$*: If* $m = \perp$*, this is a read query and returns the value* m' *indexed by* $i \in [N]$*; if* $m \neq \perp$*, this is a write query and sets the value indexed by* i *to* m *and returns* $m' = \perp$*.*

Definition 2 (Secure ORAM). *An ORAM is secure if it satisfies the following correctness and obliviousness.*

Correctness. *When a read is performed on index* i*, the result equals the value that was last written to index* i*, or if a write has never been performed on index* i*, it returns the initial value of index* i*,* $A[i]$*.*

Obliviousness. *For any initial arrays* A *and* A' *of length* N *and any sequence of queries* $\{(i_1, m_1), \cdots, (i_t, m_t)\}$ *and* $\{(i'_1, m'_1), \cdots, (i'_t, m'_t)\}$ *where* $i_j, i'_j \in [N]$ *and* $m_j, m'_j \in \mathcal{M} \cup \{\perp\}$*, the following equation is satisfied.*

$$\mathsf{Acc}\begin{pmatrix} O \leftarrow \mathsf{Init}(A, N), \\ \mathsf{Query}(O, i_1, m_1), \\ \cdots \\ \mathsf{Query}(O, i_t, m_t) \end{pmatrix} \approx \mathsf{Acc}\begin{pmatrix} O' \leftarrow \mathsf{Init}(A', N), \\ \mathsf{Query}(O', i'_1, m'_1), \\ \cdots \\ \mathsf{Query}(O', i'_t, m'_t) \end{pmatrix}$$

where $\mathsf{Acc}()$ *is the sequence of physical memory accesses when executing the input algorithms, and* $\approx$ *refers to computational, statistical, or perfectly indistinguishability.*

Tree-based ORAM. Our work mainly relies on tree-based ORAM ([48], [49], [50], [57]). Therefore, we briefly recall tree-based ORAM here.

In tree-based ORAM, N blocks are organized into a binary tree of height $L = \log N$, and each node is a bucket containing Z blocks. Each block is of the form $\{\mathsf{bid}||\mathsf{lid}||\mathsf{data}\}$ where bid is a block identifier and lid is a leaf identifier specifying the path on which the block resides, and data is the payload of the block.

Each block is associated with a path identified by the leaf id lid and will be evicted through the path. The mapping relation between blocks and paths is stored in the position map, which is held by the client. According to [50], tree-based ORAMs can be summarized into Algorithm 1, and Evict() is the key difference between tree-based ORAM schemes. Specifically, for access to a block indexed by bid, the client first retrieves the corresponding lid from the position map (line 2). Then, the client fetches all blocks on the path from the server, and reads and removes the block from the path (line 3). Then, the client reassigns a new path for this block and updates the position map (line 4). If this is a "read" operation, the client obtains the data stored in the block, otherwise, the client updates the data field in this block. Finally, the client adds the accessed block into the stash and executes the eviction process (lines 8-9), writing the fetched blocks back into the tree.

Algorithm 1 Tree-based ORAM: Access(op) // where op=("read", bid) or op=("write", bid, data^*)

```
1:  procedure ACCESS(op)
2:      lid = PositionMap[bid]                          ▷ Obtain the leaf id
3:      {bid||lid||data} = ReadAndRm(bid, lid)          ▷ Read and remove the block
4:      PositionMap[bid] = UniformRandom(0, ··· , N−1)  ▷ Assign a new path for the block
5:      if op is "read" then
6:          data* = data
7:      end if
8:      stash.add({bid||PositionMap[bid]||data*})       ▷ Add the accessed block to the stash
9:      Evict()
10:     Return data
11: end procedure
```

ORAM in Secure Computation. The storage of the server and the client is secret-shared between the parties and all the operations are accomplished over secure computation. Specifically, for tree-based ORAM in two-party secure computation, the tree and position map are secret-shared between the two parties (i.e., two servers in our work). For an access, each party obtains shares [bid] and [data^*], and the operations shown in Algorithm 1 are implemented using secure computation protocols.

3.2. Oblivious Shuffling

The two parties input two vectors $\vec{x} = (x_1, \cdots, x_n)$ and $\vec{y} = (y_1, \cdots, y_n)$, respectively. Oblivious shuffling permutes the vector $(x_1 + y_1, \cdots, x_n + y_n)$ to $\vec{z} = (x_\pi(1) + y_\pi(1), \cdots, x_\pi(n) + y_\pi(n))$ with a permutation π not known by P_0 and P_1. Then, each element z_i in $\vec{z}$ is reshared to $z_{i,0}$ and $z_{i,1}$, such that $z_{i,0} + z_{i,1} = z_i$. Finally, P_0 and P_1 obtain $\vec{z}_0 = (z_{1,0}, \cdots, z_{n,0})$ and $\vec{z}_1 = (z_{1,1}, \cdots, z_{n,1})$, respectively. We use the state-of-the-art design by Peceny et al. [40] to instantiate oblivious shuffling.

3.3. Private Equality Test

Through the Private Equality Test (PET), two parties with items x and y can obtain bits b_0 and b_1, respectively, such that if $x = y$, $b_0 \oplus b_1 = 1$, otherwise, $b_0 \oplus b_1 = 0$.

Functionality $\mathcal{F}_{\mathsf{Shuffle}}$

Parameters:
- Two parties: P_0 and P_1;

Functionality:
1. Wait for input $\vec{x} = (x_1, \cdots, x_n)$ from P_0;
2. Wait for input $\vec{y} = (y_1, \cdots, y_n)$ from P_1
3. Choose a random permutation π, compute $\vec{z} = (x_{\pi(1)} + y_{\pi(1)}, \cdots, x_{\pi(n)} + y_{\pi(n)})$, and reshare $\vec{z}$ as $\vec{z}_0$ and $\vec{z}_1$, such that $\vec{z} = \vec{z}_0 + \vec{z}_1$;
4. Send $\vec{z}_0$ and $\vec{z}_1$ to P_0 and P_1, respectively.

Figure 5: Oblivious Shuffling Ideal Functionality

Functionality $\mathcal{F}_{\mathsf{PET}}$

Parameters:
- Two parties: P_0 and P_1;

Functionality:
1. Wait for input $x \in \{0,1\}^{\ell_1}$ from P_0;
2. Wait for input $y \in \{0,1\}^{\ell_1}$ from P_1;
3. Give output b_0 and b_1 to P_0 and P_1, respectively, where b_0 and b_1 are boolean shares of b, where $b = 1$ if $x = y$ and $b = 0$ if $x \neq y$.

Figure 6: Private Equality Testing Ideal Functionality

We show the formal definition in Figure 6. The performance of PET directly affects the latency of our protocol, so we implemented it using the scheme in [58], which enjoys efficient online performance.

4. Protocol Design

In this section, we will give a detailed description of our protocol. Next, we first provide the basic protocol and then leverage a variant of ORAM to support long messages.

4.1. Basic Protocol

In the initialization phase (see Figure 7), Server_1 (resp. Server_2) initializes two empty vectors $\vec{M}_1$ (resp. $\vec{M}_2$) and $\vec{X}_1$ (resp. $\vec{X}_2$). Later, $\vec{M}_j$ ($j \in \{1,2\}$) will be used to store the messages and $\vec{X}_j$ ($j \in \{1,2\}$) will be used to store the information about addresses. Each recipient Rcv_i uses $\mathsf{Addr}_{\mathsf{Rcv}_i} = g^{k_{\mathsf{Rcv}_i}}$ as her address to receive messages, and k_{Rcv_i} is the corresponding secret key. We assume that the potential sender can obtain the address $\mathsf{Addr}_{\mathsf{Rcv}_i}$ in some way, such as through a public website or secure peer-to-peer communications.

In the sending phase (see Figure 8), we assume that a sender wants to send a message m to a recipient associated with an address $\mathsf{Addr}_{\mathsf{Rcv}_i}$. The sender first generates a one-time address $A = \mathsf{Addr}^r_{\mathsf{Rcv}_i}$ and an associated string $R = g^r$. Then, the sender splits the one-time address A into two shares A_1 and A_2, such that $A = A_1 \cdot A_2$, and shares the message m into two shares $[m]_1$ and $[m]_2$, such that $m = [m]_1 + [m]_2$. After preparing these, the sender sends $((A_j, R), [m]_j)$ to Server_j, where $j \in \{1,2\}$. Once receiving the sending request, Server_j appends $[m]_j$ to $\vec{M}_j$ and (A_j, R) to $\vec{X}_j$.

Basic Protocol $\Pi_{\mathsf{Initialize}}$

There are two servers, Server_1 and Server_2, and multiple clients acting as senders or recipients.

Initialization:

Each Server_j ($j \in \{1,2\}$) does the following:

(1) Initialize two empty vectors $\vec{M}_j = \emptyset$ and $\vec{X}_j = \emptyset$ (for retrieval);

Each client acting as a recipient Rcv_i does the following:

(2) Randomly choose $k_{\mathsf{Rcv}_i} \leftarrow \mathbb{Z}_p$, and generate an address $\mathsf{Addr}_{\mathsf{Rcv}_i} = g^{k_{\mathsf{Rcv}_i}} \in \mathbb{G}$; ($\mathbb{G}$ is a cyclic group with prime order p and generator g.)

(3) For each potential sender Sdr_j, send $\mathsf{Addr}_{\mathsf{Rcv}_i}$ to Sdr_j.

Figure 7: Basic protocol for initialization.

Basic Protocol Π_{Send}

Sender: sending information to the two servers:

To send a message m to a recipient Rcv_i associated with address $\mathsf{Addr}_{\mathsf{Rcv}_i}$, the sender does the following:

(1) Randomly choose $r \xleftarrow{\$} \mathbb{Z}_p$, and generate $A = \mathsf{Addr}^r_{\mathsf{Rcv}_i}$ and $R = g^r$;

(2) Randomly split A into A_1 and A_2, such that $A = A_1 \cdot A_2$;

(3) Generate the shares $([m]_1, [m]_2)$;

(4) Send $((A_1, R), [m]_1)$ to Server_1 and $((A_2, R), [m]_2)$ to Server_2.

Servers: preparing for retrievals:

(5) The two servers check if $((A_j, R), [m]_j)$ is well-formed for $j \in \{1,2\}$, if so, continue, otherwise ignore it.

(6) Server_j appends $[m]_j$ to $\vec{M}_j$ and (A_j, R) to $\vec{X}_j$.

Figure 8: Basic protocol for sending.

In the retrieval phase (see Figure 9), the recipient first randomly splits the secret key k_{Rcv_i} into two shares $k_{\mathsf{Rcv}_i,1}$ and $k_{\mathsf{Rcv}_i,2}$, and then sends the two shares to the two senders, respectively. Obviously, if a one-time address (A_k, R_k) corresponds to the secret key k_{Rcv_i}, then $A_k = R_k^{k_{\mathsf{Rcv}_i}} = A_{k,1} \cdot A_{k,2}$ where $A_{k,1}$ and $A_{k,2}$ are the shares of A_k. Therefore, we have $A_{k,1} \cdot A_{k,2} = R_k^{k_{\mathsf{Rcv}_i,1}} \cdot R_k^{k_{\mathsf{Rcv}_i,2}}$, i.e., $A_{k,1}/R_k^{k_{\mathsf{Rcv}_i,1}} = R_k^{k_{\mathsf{Rcv}_i,2}}/A_{k,2}$. Assuming that there are n unretrieved messages, for each $k \in [n]$, the two servers compute $a_{k,1} = H_1(A_{k,1}/R_k^{k_{\mathsf{Rcv}_i,1}})$ and $a_{k,2} = H_1(R_k^{k_{\mathsf{Rcv}_i,2}}/A_{k,2})$, respectively. For each pair $(a_{k,1}, a_{k,2})$, the two servers invoke $\mathcal{F}_{\mathsf{PET}}$ and obtain the shares of equality test result $b_{k,1}$ and $b_{k,2}$ respectively. Then, Server_j concatenates $b_{k,j}$ with the corresponding message share $\vec{M}_j[k]$ to obtain $l_{k,j} = b_{k,j} || \vec{M}_j[k]$. We denote the vector $(l_{1,j}, l_{2,j}, \cdots, l_{n,j})$ as $\vec{l}_j$. Then, the two servers invoke $\mathcal{F}_{\mathsf{Shuffle}}$ with inputs $\vec{l}_1$ and $\vec{l}_2$ and obtain outputs $\vec{g}_1$ and $\vec{g}_2$, respectively. According to the definition of $\mathcal{F}_{\mathsf{Shuffle}}$, $\vec{g}_1[k] + \vec{g}_2[k] = \vec{l}_1[\pi(k)] + \vec{l}_2[\pi(k)]$ for any $k \in [n]$ where the permutation π is not known to the two servers. We denote each $\vec{g}_j[k]$ as $b'_{k,j} || [m]'_{k,j}$. After collaboratively opening

Basic Protocol Π_{Retrieve}

A recipient holding $(\mathsf{Addr}_{\mathsf{Rcv}_i}, k_{\mathsf{Rcv}_i})$ wants to retrieve the messages pertinent to her.

Recipient:

1) Randomly split k_{Rcv_i} into $k_{\mathsf{Rcv}_i,1}$ and $k_{\mathsf{Rcv}_i,2}$ such that $k_{\mathsf{Rcv}_i} = k_{\mathsf{Rcv}_i,1} + k_{\mathsf{Rcv}_i,2}$;
2) Send $k_{\mathsf{Rcv}_i,1}$ to Server_1 and $k_{\mathsf{Rcv}_i,2}$ to Server_2;
3) For each $k \in [n]$, once receiving $[m]'_{k,1}$ and $[m]'_{k,2}$ from the two servers, recover $m = [m]'_{k,1} + [m]'_{k,2}$.

Servers:

1) The two servers do the following for each $k \in [n]$ ($j \in \{1,2\}$) where $|\vec{X}_1| = |\vec{X}_2| = |\vec{M}_1| = |\vec{M}_2| = n$:
   a) Server_j retrieves $\vec{X}_j[k] = (A_{k,j}, R_k)$;
   b) Server_1 computes $a_{k,1} = H_1(A_{k,1}/R_k^{k_{\mathsf{Rcv}_i,1}})$ and Server_2 computes $a_{k,2} = H_1(R_k^{k_{\mathsf{Rcv}_i,2}}/A_{k,2})$, where H_1 is a hash function: $\mathbb{G} \to \{0,1\}^{\ell_3}$;
   c) Server_1 and Server_2 invoke $\mathcal{F}_{\mathsf{PET}}$ with inputs $a_{k,1}$ and $a_{k,2}$, and obtain $b_{k,1}$ and $b_{k,2}$ respectively;
   d) Server_j generates $l_{k,j} = b_{k,j}||\vec{M}_j[k]$ (we denote $\vec{l}_j = (l_{1,j}, l_{2,j}, \cdots, l_{n,j})$);
2) Server_1 and Server_2 invoke $\mathcal{F}_{\mathsf{Shuffle}}$ with inputs $\vec{l}_1$ and $\vec{l}_2$, respectively, and obtain outputs $\vec{g}_1$ and $\vec{g}_2$, where $\vec{g}_1[k] + \vec{g}_2[k] = \vec{l}_1[\pi(k)] + \vec{l}_2[\pi(k)]$ for any $k \in [n]$;
3) For each $k \in [n]$:
   a) Server_j retrieves $\vec{g}_j[k] = b'_{k,j}||[m]'_{k,j}$;
   b) Server_1 and Server_2 collaboratively open $b'_k = b'_{k,1} + b'_{k,2}$;
   c) If $b'_k = 1$, Server_j sends $[m]'_{k,j}$ to the recipient.

Figure 9: Basic protocol for retrieval.

$b'_k = b'_{k,1} + b'_{k,2}$, Server_j sends $[m]'_{k,j}$ to the recipient if $b'_k = 1$.

4.2. Scaling With ORAM

Our basic protocol is not friendly for long messages, as it needs to shuffle all the messages. If the length of each message is too large, the performance would not be practical. To solve this problem, we maintain the messages using a variant of ORAM called ORAM^-, instead of putting the messages into the above list. Specifically, the basic scheme described in Section 4.1 is used to provide short indexes, and these indexes can be used to retrieve the corresponding messages from ORAM^-. Additionally, ORAM^- is maintained by the two servers in a secure computation way.

Next, we first give the construction of ORAM^- and then give our scaling protocol by using ORAM^-. See Section 5.1 for the definition and security analysis of ORAM^-.

Construction of ORAM^-. Compared to classical ORAMs, our ORAM^- enjoys the following features:

- (1) The block bid is randomly chosen;
- (2) An active block can only be written once and read once;
- (3) Each block is not initialized with an bid. When writing, an empty block would be assigned with a specific bid and the data would be written into the block; after reading, the corresponding block would be reset as an inactive block. When using tree-based ORAM, we do not need to evict after reading;
- (4) It is not necessary to hide the operation type.

Next, we construct ORAM^- in Algorithm 2 based on tree-based ORAM (see Algorithm 1). Specifically, different from tree-based ORAM, ORAM^- does not have a position map, and the inputs of Read and Write directly include the leaf id lid. For a Read operation, ORAM^- only needs to return and remove the data stored in the block indexed by bid from the path indexed by lid. For a Write operation, ORAM^- adds the block containing $\mathsf{bid}||\mathsf{lid}||\mathsf{data}^*$ to stash and execute the eviction process.

Algorithm 2 ORAM^-

```
1: procedure READ(bid, lid)
2:     {bid||lid||data} = ReadAndRm(bid, lid)      ▷ Read and remove the block
3:     Return data
4: end procedure
5: procedure WRITE(bid, lid, data*)
6:     stash.add({bid||lid||data*})      ▷ Add the accessed block to the stash
7:     Evict()
8: end procedure
```

Scaling Protocol. Next, we incorporate the above ORAM^- in secure computation setting into the basic protocol to construct a scalable protocol. The initialization phase is the same as that in the basic protocol (i.e., Figure 7). Therefore, we only give the details about the sending and retrieval phases next, and the differences from the basic protocol are marked with underlines.

In the sending phase (see Figure 10), compared to the basic protocol, we need to generate a block id bid and leaf id lid for each message. Later, the message m will be written into and retrieved from the block indexed by bid. To guarantee the correctness of ORAM^-, the block id should be unique and the leaf id should be randomly chosen. Since the sender may be malicious, we cannot assume that the sender generates bid and lid correctly. Moreover, when retrieving, the two servers can know which path identified by a leaf id is retrieved. If the leaf id is chosen by the sender, the sender can learn some information about the recipient by colluding with one of the servers. Therefore, we require the two servers to randomly generate the shares of block id and leaf id, such that none of the servers and sender knows $(\mathsf{bid}, \mathsf{lid})$ and $(\mathsf{bid}, \mathsf{lid})$ are random.

After generating $([\mathsf{bid}]_j, [\mathsf{lid}]_j)$, Server_j appends $[\mathsf{bid}]_j||[\mathsf{lid}]_j$, rather than $[m]_j$, to $\vec{M}_j$. Finally, the two servers collaboratively execute ORAM^-.WRITE with the shares $([\mathsf{bid}]_1, [\mathsf{lid}]_1, [m]_1)$ and $([\mathsf{bid}]_2, [\mathsf{lid}]_2, [m]_2)$ as input.

In the retrieval phase (see Figure 11), different from the basic scheme where the two servers directly retrieve the message shares from $\vec{M}_1$ and $\vec{M}_2$, the two servers here first obtain the shares of pertinent block id and leaf id and then collaboratively execute ORAM^-.READ to obtain the message shares $[m]_{k,1}$ and $[m]_{k,2}$.

Protocol Π_{Send}

Sender: sending information to the two servers:
The same as in the basic protocol (see Figure 8).
Servers: preparing for retrievals:
The two servers check if $((A_j, R), [m]_j)$ it is well-formed for $j \in \{1, 2\}$, if so, continue, otherwise ignore it.
1) Server_j randomly chooses a block id share $[\mathsf{bid}]_j \xleftarrow{\$} \{0,1\}^{\ell_1}$ and a leaf id share $[\mathsf{lid}]_j \xleftarrow{\$} \{0,1\}^{\ell_2}$;
2) Server_j append $[\mathsf{bid}]_j || [\mathsf{lid}]_j$ to $\vec{M}_j$ and (A_j, R) to $\vec{X}_j$;
3) The two servers execute ORAM^-.WRITE with input $([\mathsf{bid}]_1, [\mathsf{lid}]_1, [m]_1)$ and $([\mathsf{bid}]_2, [\mathsf{lid}]_2, [m]_2)$.

Figure 10: Scalable protocol for sending.

Protocol Π_{Retrieve}

A recipient holding $(\mathsf{Addr}_{\mathsf{Rcv}_i}, k_{\mathsf{Rcv}_i})$ wants to retrieve the messages pertinent to her.
Recipient:
The same as in the basic protocol (see Figure 9).
Servers:
1) The same as in the basic protocol (see Figure 9);
2) The same as in the basic protocol (see Figure 9);
3) For each $k \in [n]$:
   a) Server_j retrieves $\vec{g}_j[k] = b'_{k,j} || [\mathsf{bid}]'_{k,j} || [\mathsf{lid}]'_{k,j}$;
   b) Server_1 and Server_2 collaboratively open $b'_k = b'_{k,1} \oplus b'_{k,2}$;
   c) If $b'_k = 1$, Server_1 and Server_2 invoke ORAM^-.READ with inputs $[\mathsf{bid}]'_{k,1} || [\mathsf{lid}]'_{k,1}$ and $[\mathsf{bid}]'_{k,2} || [\mathsf{lid}]'_{k,2}$, respectively, and obtain outputs $[m]_{k,1}$ and $[m]_{k,2}$, respectively;
   d) Server_j sends $[m]_{k,j}$ to the recipient.

Figure 11: Scalable protocol for retrieval.

4.3. Protecting Volume Information by Adding Dummy Messages

Our protocol explained in Sections 4.1 and 4.2 leaks volume information to the adversary: how many messages are sent by a sender, and how many messages are retrieved by a recipient. Different strategies exist in the literature to protect against such volume information leakage. Protocols that inherently employ rounds can use batching [3], [8], [13], [27] techniques, where every user sends exactly the same number of messages in a batch, and each recipient retrieves the exact same number of messages after the batch is processed.

On the other hand, protocols that do not employ such batching methods utilize heuristic techniques based on dummy messages [7], [16]. Our cryptographic construction also departs from the restrictions of rounds and batches and aims to achieve what a trusted third-party anonymizer could guarantee. Therefore we recommend techniques based on dummy messages in order to hide such volume information. The clients can add dummy messages to hide their overall rate of sending and receiving messages, and the amount of dummy messages can be chosen based on the desired amount of privacy. A thorough analysis of such a strategy is out of the scope of this work, and we refer to existing works [7], [16] from literature for the quantification of dummy message rate vs. desired privacy.

5. Security Analysis

5.1. Security of ORAM^-

ORAM^- is a variant of ORAM, where a block is activated by a "write" operation and will be released after a "read" operation, and the type of operation does not need to be hidden. We provide the definition of our variant of ORAM ORAM^- adopted from the definition in [56]. Note that we do not consider ORAM^- in the secure computation setting here. As in previous works, the security of ORAM^- in the secure computation setting relies on the security of ORAM^- and the underlying secure computation techniques.

Definition 3 (ORAM for Transmission)**.** *A* $\mathsf{ORAM}^- = (\mathsf{Init}, \mathsf{Write}, \mathsf{Read})$ *includes the following three polynomial-time algorithms:*

- $(\mathsf{St}, \mathsf{Idx}) \leftarrow \mathsf{Init}(N, p)$*: According to the predefined maximum number of stored messages* N *and the predefined collision probability* p*, generate an initialized state* St *and an index space* Idx*. The collision probability refers to the probability that multiple stored messages correspond to one index.*
- $\mathsf{St}' \leftarrow \mathsf{Write}(\mathsf{St}, i, m)$*: Write a message* m *that associates with an index* $i \in \mathsf{Idx}$ *into the current state* St*, and the state* St *is updated to* St'*.*
- $(\mathsf{St}', m) \leftarrow \mathsf{Read}(\mathsf{St}, i)$*: If the index* i *has been written before, read the message* m *that associates with the index* $i \in \mathsf{Idx}$ *from the current state* St*, otherwise, read* $m = \perp$*. Additionally, update the state* St *to* St'*.*

Definition 4 (Secure ORAM for Transmission)**.** *A* ORAM^- *is secure if it satisfies the following correctness and obliviousness.*

Correctness. *For an operation* $(\mathsf{St}', m_r) \leftarrow \mathsf{Read}(\mathsf{St}, i)$*, if the last operation about the index* i *is* $\mathsf{St}' \leftarrow \mathsf{Write}(\mathsf{St}, i, m_w)$*, then* $m_r = m_w$*, otherwise (there are no previous operations on the index* i*, or the last operation about the index* i *is also a "read" operation),* $m_r = \perp$*.*

Obliviousness. *For any two sequences of operations* $\{\mathsf{Op}_1(\mathsf{St}_1, i_1, m_1), \mathsf{Op}_2(\mathsf{St}_2, i_2, m_2), \cdots, \mathsf{Op}_t(\mathsf{St}_t, i_t, m_t)\}$ *and* $\{\mathsf{Op}'_1(\mathsf{St}'_1, i'_1, m'_1), \mathsf{Op}'_2(\mathsf{St}'_2, i'_2, m'_2), \cdots, \mathsf{Op}'_t(\mathsf{St}'_t, i'_t, m'_t)\}$ *chosen by the adversary, that satisfy the following conditions:*

- Op *and* Op' *are in* $\{\mathsf{Write}, \mathsf{Read}\}$*;*
- *If the operation is* Read*, the corresponding* m_k *or* m'_k *is* $\perp$*;*
- $\mathsf{Op}_k = \mathsf{Op}'_k$*.*

We have

$$\mathsf{Acc}\left(\begin{array}{c}(\mathsf{St}_1, \mathsf{Idx}) \leftarrow \mathsf{Init}(N, p),\\ (\mathsf{St}_2, m_1/\perp) \leftarrow \mathsf{Op}(\mathsf{St}_1, i_1, m_1),\\ \cdots\\ (\mathsf{St}_t, m_t/\perp) \leftarrow \mathsf{Op}(\mathsf{St}_t, i_t, m_t)\end{array}\right)$$

$$\approx \quad \mathsf{Acc}\left(\begin{array}{c}(\mathsf{St}'_1, \mathsf{Idx}') \leftarrow \mathsf{Init}(N, p),\\ (\mathsf{St}'_2, m'_1/\perp) \leftarrow \mathsf{Op}(\mathsf{St}'_1, i'_1, m'_1),\\ \cdots\\ (\mathsf{St}'_{t+1}, m'_t/\perp) \leftarrow \mathsf{Op}(\mathsf{St}'_t, i'_t, m'_t)\end{array}\right)$$

Where $\mathsf{Acc}()$ *is the sequence of physical memory accesses when executing the input algorithms, and* $\approx$ *refers to computational, statistical, or perfectly indistinguishability.*

Theorem 1. *If the underlying tree-based ORAM is a secure ORAM, then our construction shown in Algorithm 2 is a secure* ORAM^-.

Proof. Next, we will analyze the correctness and obliviousness of our construction, respectively.

Correctness. The correctness is obviously guaranteed by the correctness of the underlying tree-based ORAM. For a "read" operation, it just releases an active block, and thus will not incur extra overflow. A "write" operation in ORAM^- is equivalent to writing a message to an empty block in tree-based ORAM. Therefore, as long as the number of stored messages is less than the upper bound N, a "write" operation will also not incur extra overflow.

As for the collisions of indexes, we can use the birthday paradox probability to set the bit-length of each block ID as ℓ_{bid}, given the predefined maximum number of stored messages N.

Obliviousness. If there is an adversary $\mathcal{A}$ that can distinguish two sequences of physical memory accesses generated by two sequences of operations run by ORAM^-. Then, we can construct another adversary $\mathcal{B}$ that can distinguish two sequences of physical memory accesses generated by two sequences of operations run by ORAM. Due to space limitation, we defer the construction of the adversary $\mathcal{B}$ to Appendix B. □

5.2. Security Against Passive Adversaries

5.2.1. Anonymity Definition. We focus on both sender anonymity and recipient anonymity for our protocol. With that goal, we define an ideal functionality $\mathcal{F}_{anon}$ that can capture both properties. Our $\mathcal{F}_{anon}$ functionality, however, does not hide the volume information corresponding to the sender: if a specific sender sends too many (or too few) messages, that volume information is leaked to the adversary. Even a trusted third-party anonymizer that shuffles all the messages after receiving every new message would leak such volume information, unless dummy messages are added by the senders. We discuss how to hide such volume information based on dummy messages in Section 4.3. We present the ideal functionality $\mathcal{F}_{anon}$ in Fig. 12.

Functionality $\mathcal{F}_{anon}$

1: The functionality maintains a set $\mathcal{T}$ which contains the messages and the corresponding sender and recipient. Each element in the set is stored as a tuple $(\mathsf{msg}, \mathtt{sender}, \mathtt{recipient})$.
2: The functionality also maintains a set $\mathcal{R}$ that keeps track of the messages that are retrieved.
3: **Sending a message** (Send)**:**
- Upon receiving $(\mathsf{Send}, \mathsf{Addr}_{\mathsf{Rcv}_j}, \mathsf{msg})$ from an honest sender u_i:
  - store the tuple $(u_i, \mathsf{Addr}_{\mathsf{Rcv}_j}, \mathsf{msg})$ in $\mathcal{T}$;
  - send $(\mathsf{Send}, u_i, \_, \_)$ to the simulator $\mathcal{S}$.
- Upon receiving $(\mathsf{Send}, u_i, \mathsf{Addr}_{\mathsf{Rcv}_j}, \mathsf{msg})$ from $\mathcal{S}$:
  - if sender u_i is a corrupted party, store the tuple $(u_i, \mathsf{Addr}_{\mathsf{Rcv}_j}, \mathsf{msg})$ in $\mathcal{T}$;
  - if u_i is an honest party, return `Invalid command` to $\mathcal{S}$.

4: **Retrieving messages** (Retrieve)**:**
- Upon receiving $(\mathsf{Retrieve}, \mathsf{Rcv}_j)$ from the simulator $\mathcal{S}$ for a corrupted recipient Rcv_j:
  - select all tuples $t = (\mathsf{msg}, \mathtt{sender}, \mathtt{recipient}) \in \mathcal{T}$ where $t.\mathtt{recipient} = \mathsf{Addr}_{\mathsf{Rcv}_j}$.
  - Collect the msg field of those tuples in a list; shuffle the list, and send the shuffled list to $\mathcal{S}$.
  - Remove those tuples from $\mathcal{T}$.
- Upon receiving $(\mathsf{Retrieve}, \_)$ from an honest Rcv_j:
  - select all tuples $t = (\mathsf{msg}, \mathtt{sender}, \mathtt{recipient}) \in \mathcal{T}$ where $t.\mathtt{recipient} = \mathsf{Addr}_{\mathsf{Rcv}_j}$.
  - If there are k such tuples, send the msg field of those tuples to Rcv_j as a list; remove those k entries from $\mathcal{T}$.
  - Send $(\mathsf{Retrieve}, \mathsf{Rcv}_j, k)$ to the simulator $\mathcal{S}$.
  - Remove those k entries from $\mathcal{T}$.

Figure 12: An ideal functionality capturing both sender and recipient anonymity.

Theorem 2. *Assuming the hardness of the Discrete-logarithm problem, computational security and correctness of* ORAM^-*, and at least one of the servers is honest, our protocol* $\Pi = (\Pi_{\mathsf{Initialize}}, \Pi_{\mathsf{Send}}, \Pi_{\mathsf{Retrieve}})$ *UC-realizes the ideal functionality* $\mathcal{F}_{anon}$ *in the* $\{\mathcal{F}_{\mathsf{PET}}, \mathcal{F}_{\mathsf{Shuffle}}, \mathcal{F}_{\mathsf{SC-ORAM}^-}\}$*-hybrid world.*

$\mathcal{F}_{\mathsf{SC-ORAM}^-}$ denotes the ideal functionality that captures running our ORAM^- in the secure computation setting as explained in Section 2.4. We present the definition of $\mathcal{F}_{\mathsf{SC-ORAM}^-}$ in Figure 14 in Appendix C. It takes input from both servers, runs ORAM^- locally without leaking the inputs, and returns the output.

To prove that our protocol UC-realizes the $\mathcal{F}_{anon}$ functionality, we show that there exists a simulator $\mathcal{S}_{\mathtt{full}}$ interacting with $\mathcal{F}_{anon}$ functionality that generates a transcript that is indistinguishable from the transcript generated by the real-world adversary $\mathcal{A}$ in the protocol Π. We present the description of the simulator (Figs. 15 and 16) and the full proof in Appendix C.

5.3. Defending Against Malicious Behaviors

The security arguments presented in Section 5.2 assume that the adversary is honest-but-curious — the compromised clients and server still follow protocol faithfully. However, malicious clients (or the malicious server) could choose to disrupt the protocol. Below we discuss the relevant disruptive behaviors and the defenses against them.

5.3.1. Malicious Senders. If a malicious client (sender) sends invalid A_j shares such that the intended recipient is invalid, it will only accumulate invalid messages in the servers' storage. Such invalid messages will not impact the functionality or security of the system, except for slowing it down over time. To avoid such slowdown, the two servers can periodically run any private-membership-tests techniques (e.g., vector commitments [59]). However, a malicious sender can still overwhelm a specific recipient by sending a lot of messages; such attacks could be prevented by rate-limiting by the servers.

Targeted attacks where a (malicious) sender tries to disrupt/modify the messages sent by others are automatically eliminated in our design: the addresses for the blocks in ORAM^{-} are chosen from a large address space so that the collision probability is negligible (the bid and lid values are randomly chosen by the servers).

5.3.2. Malicious Server Colluding With Receivers. In our scheme, we require the sender to generate a one-time address $\mathsf{Addr}^{r}_{\mathsf{Rcv}}$ for each sending, which is used to defend this malicious behavior. Without the one-time address, for an address Addr, the two servers would hold A_1 and A_2, respectively, such that $A_1 \cdot A_2 = \mathsf{Addr}$. We assume that a malicious receiver colludes with Server_2. Then, the malicious receiver not knowing the secret key of Addr randomly generates a secret key share k, and sends k to Server_1. Server_1 will compute $a_1 = H_1(A_1/g^k)$. Since the receiver colludes with Server_2, Server_2 can compute $a_2 = H_2(\mathsf{Addr}/(A_2 \cdot g^k))$. We can see that a_1 must be equal to a_2, allowing the receiver to get the corresponding message (even though the malicious receiver cannot obtain the plain texts of these messages, the receiver can learn how many messages are sent to the address). By using the one-time address, the malicious receiver cannot know the actual address used for sending messages, and thus cannot retrieve the corresponding messages. Note that the colluding server can still just change the bits output by PETs to allow the receiver to get more messages. However, they cannot get more information from the behavior; the messages can be ciphertexts and they still cannot know the sender or the receiver of the messages.

HomeRun also discussed using one-time addresses to address this issue. However, their solution is not suitable for our work, since Server_1 in HomeRun can learn the one-time address belonging to a receiver. Then, the sender can collude with the server to know the receiver, as the one-time address is generated by the sender. In our scheme, the servers cannot know the one-time address.

5.3.3. Tampering with the stored data. Same as prior works [3], [14], to prevent the servers from tampering with the messages, we require the sender to include a MAC for each message. However, the malicious server might selectively modify messages and verify on the recipient side which messages fail MAC-verification (especially when colluding with the receiver). To address that issue, we utilize the *Blind MAC verification* technique using Beaver triples, introduced in Clarion [14]. The beaver triples are generated as part of a preprocessing phase, and our servers executes the Blind MAC verification before every retrieval operation. For a detailed description of the verification technique we refer to [14, Section V.B].

6. Performance Evaluation

Implementation. We implement Kerblam in C++, and we will make the code public upon acceptance of the paper. The cyclic group $\mathbb{G}$ is realized using the elliptic curve secp256r1 provided in OpenSSL [60]. The oblivious message retrieval part is implemented using the code of HomeRun [10]. The Oblivious Shuffling is implemented using the code of [40][3]. We implement ORAM^{-} based on circuit ORAM [50] whose code is provided in [61].

6.1. Experimental Evaluation

We aim to answer the following questions from our experimental evaluations:

- While trying to achieve stronger anonymity guarantees (End-to-end unlinkability), do we introduce impractical overheads for the system?
- How much performance improvement do we gain from the scaling technique based on ORAM^{-}?

Experimental Environment. Our experiments are conducted on machines equipped `Intel(R) Core(TM) i9-14900K` with 24 cores and 128GB of RAM, running Ubuntu. We evaluate Kerblam in LAN network with 10Gbps bandwidth and 0.08 ms RTT.

6.1.1. Benchmarks. First, we present the benchmarks for the operations (equality test, shuffling, and ORAM^{-} read and write) run on a single thread on the servers. We show the benchmarks in Table 1.

The basic protocol (without ORAM^{-} optimization) only includes "Preparation", "Equality Test", "Shuffling". We can see that for short messages (16B), the overhead for the "Shuffling" part is reasonable (less than 1.3 seconds even for 2^{20} messages), and the equality test is the dominant part in the performance (around 8.6 seconds for 2^{20} messages). However, for long messages of 1KB, the overhead for shuffle drastically increases with the number of messages, and becomes the dominant part among all steps. For 2^{20} messages this step takes more than 244 seconds, which makes the base protocol less practical for long messages.

3. We received the source code by contacting the authors over emails.

	Prep.	Equality Test		Shuffling		ORAM$^-$ (1 KB)	
		Offline	Online	16 B	1 KB	Write	Read
2^{14}	0.448	0.413	0.138	0.031	1.226	0.276	0.058
2^{16}	1.789	1.674	0.522	0.071	5.445	0.3	0.063
2^{18}	7.098	7.218	2.144	0.309	28.457	0.331	0.068
2^{20}	28.628	28.665	8.608	1.264	244.7	0.369	0.073

TABLE 1: The breakdown runtime (seconds) of Kerblam in a single thread, when the number of messages stored by the two servers $n \in \{2^{14}, 2^{16}, 2^{18}, 2^{20}\}$. For basic protocol (see Figure 9), step 1-a and step 1-b correspond to "Preparation"; step 1-c and step 1-d correspond to "Equality Test"; steps 2-3 correspond to "Shuffling". For salable sending protocol (see Figure 10), step 3 corresponds to "ORAM$^-$ Write". For the salable sending protocol (see Figure 11), step 2 corresponds to "Shuffling 16 B" and step 3 corresponds to "ORAM$^-$ Read". Note that the performance of the "Equality Test" phase is almost the performance of HomeRun.

As we can see, the time taken for ORAM$^-$ operations grows very slowly with the number of messages. So, for short (16B) and small number (2^{14}) of messages, ORAM$^-$ does not add much improvements. However, for long messages of 1KB, ORAM$^-$ adds significant benefits in terms of performance. In our scalable protocol with ORAM$^-$, shuffling is performed on short indexes, and the actual messages are stored by ORAM$^-$. So we only need to use "Shuffling (16 B)". In this case, we can see that the "Preparation" phase and "Equality Test" phase dominate the performance.

Fortunately, the two phases are highly parallelizable. We show the performance of these two phases after parallelization in Table 2. We can see that when the number of threads T is 4, the performance improvement is nearly $4\times$ for "preparation" and "equality test online, and there is a significant performance improvement for "equality test offline". When $T = 16$, the performance improvement is not exactly proportional to T, however, there is still a significant improvement.

	Prep.		Equality Test			
	T=4	T=16	T=4		T=16	
			Offline	Online	Offline	Online
2^{14}	0.132	0.066	0.161	0.061	0.117	0.035
2^{16}	0.477	0.189	0.718	0.175	0.525	0.094
2^{18}	1.885	0.724	2.985	0.626	2.178	0.241
2^{20}	7.456	2.836	11.878	2.468	8.396	1.035

TABLE 2: Parallelization of "Preparation" and "Equality Test" using $T = \{4, 16\}$ threads.

6.1.2. End-to-end Performance. Now we measure the end-to-end performance of our basic protocol as well as when we optimize with ORAM$^-$. We present the latency (i.e., the sum of sending runtime and retrieval runtime) of our basic and scalable scheme in Table 3.

We can confirm that when the message size is 16 B, the performance of our basic scheme is adequate, especially with a relatively small number of messages (2^{14}); for $n = 2^{20}$, the latency is 38.5 seconds. However, when the message size increases to 1 KB, the latency is 281.936 seconds, which is not suitable enough for practical use. After introducing ORAM$^-$, the latency can be reduced to 38.942 second for 2^{20} messages of size 1 KB, using a single thread. Further, with 16 threads, the latency goes down to only 5.577 seconds.

	Our basic scheme (T=1)		Our scalable scheme (1KB)		
	16 B	1 KB	T=1	T=4	T=16
2^{14}	0.617	1.812	0.951	0.558	0.466
2^{16}	2.382	7.756	2.745	1.086	0.717
2^{18}	9.551	37.699	9.95	3.219	1.673
2^{20}	38.5	281.936	38.942	11.63	5.577

TABLE 3: The latency (seconds) of our basic scheme and scalable scheme. The number of messages stored by the two servers $n \in \{2^{14}, 2^{16}, 2^{18}, 2^{20}\}$. The basic scheme is evaluated for message sizes of 16 B and 1 KB, in a single thread. The scalable scheme is evaluated for the message size of 1 KB, using $T = 1, 4, 16$ threads.

Comparisons. We do not make direct comparisons with other protocols since our anonymity guarantee is not comparable with existing protocols. Additionally, a significant component of our performance optimization stems from ORAM$^-$, which could arguably be utilized by other existing systems as well. However, we still want to claim that our performance is comparable to existing protocols, despite achieving stronger anonymity. Our performance is comparable to that of Express [3] that needs 15 seconds latency with 2^{20} messages and 16 threads. The performance of HomeRun [10] is almost the same as the "Equality Test" part shown in Table 1. It can be seen that the main additional overhead in our scheme compared to HomeRun is introduced by the "Preparation" part, which is used to defend the collusion between receivers and one server (as explained in Section 5.3.2). Whereas, HomeRun did not consider this attack in their evaluation.

7. Conclusion

In this work, we have designed an end-to-end anonymous messaging system, Kerblam, that can protect the privacy of both senders and receivers. In addition, we support asynchronous retrieval, allowing receivers to go offline at any time without losing messages. Our design is based on a key observation that combining oblivious message retrieval and oblivious shuffling can obtain end-to-end anonymity without compromising on the communication functionalities. However, the performance of direct combination is not enough for large-scale long messages. To improve scalability, we have proposed a novel variant of ORAM, called ORAM$^-$. By introducing ORAM$^-$, our Kerblam can transmit a 1 KB message in about 5.577 seconds when there are a total of 2^{20} unretrieved messages in the system.

References

[1] S. Sasy and I. Goldberg, "Sok: Metadata-protecting communication systems," *Proceedings on Privacy Enhancing Technologies*, 2024.

[2] R. Dingledine, N. Mathewson, and P. F. Syverson, "Tor: The second-generation onion router," in *USENIX Security 2004*, M. Blaze, Ed. USENIX Association, Aug. 2004, pp. 303–320.

[3] S. Eskandarian, H. Corrigan-Gibbs, M. Zaharia, and D. Boneh, "Express: Lowering the cost of metadata-hiding communication with cryptographic privacy," in *USENIX Security 2021*, M. Bailey and R. Greenstadt, Eds. USENIX Association, Aug. 2021, pp. 1775–1792.

[4] A. Vadapalli, K. Storrier, and R. Henry, "Sabre: Sender-anonymous messaging with fast audits," in *2022 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2022, pp. 1953–1970.

[5] D. Chaum, "The dining cryptographers problem: Unconditional sender and recipient untraceability," *Journal of Cryptology*, vol. 1, no. 1, pp. 65–75, Jan. 1988.

[6] I. Abraham, B. Pinkas, and A. Yanai, "Blinder - scalable, robust anonymous committed broadcast," in *ACM CCS 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds. ACM Press, Nov. 2020, pp. 1233–1252.

[7] D. Das, S. Meiser, E. Mohammadi, and A. Kate, "Divide and funnel: A scaling technique for mix-networks," in *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*, 2024, pp. 49–64.

[8] D. Das, E. Mangipudi, and A. Kate, "Organ: Organizational anonymity with low latency," *Proceedings on Privacy Enhancing Technologies*, vol. 2022, pp. 582–605, 07 2022.

[9] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford, "Atom: Horizontally scaling strong anonymity," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, 2017, p. 406–422.

[10] Y. Jia, V. Madathil, and A. Kate, "HomeRun: High-efficiency oblivious message retrieval, unrestricted," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[11] C. Kuhn, M. Beck, S. Schiffner, E. A. Jorswieck, and T. Strufe, "On privacy notions in anonymous communication," *PoPETs*, vol. 2019, no. 2, pp. 105–125, Apr. 2019.

[12] S. Angel and S. Setty, "Unobservable communication over fully untrusted infrastructure," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 551–569.

[13] D. Lazar, Y. Gilad, and N. Zeldovich, "Karaoke: Distributed private messaging immune to passive traffic analysis," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, Oct. 2018, pp. 711–725.

[14] S. Eskandarian and D. Boneh, "Clarion: Anonymous communication from multiparty shuffling protocols," *Cryptology ePrint Archive*, 2021.

[15] D. Lazar and N. Zeldovich, "Alpenhorn: Bootstrapping secure communication without leaking metadata," in *12th {USENIX} Security Symposium ({USENIX} Security 12)*, 2016.

[16] A. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis, "The loopix anonymity system," in *Proc. 26th USENIX Security Symposium*, 2017.

[17] N. Gelernter, A. Herzberg, and H. Leibowitz, "Two cents for strong anonymity: The anonymous post-office protocol," in *CANS 17*, ser. LNCS, S. Capkun and S. S. M. Chow, Eds., vol. 11261. Springer, Heidelberg, Nov. / Dec. 2017, pp. 390–412.

[18] C. Diaz, H. Halpin, and A. Kiayias, "The Nym Network," https://nymtech.net/nym-whitepaper.pdf, February 2021.

[19] L. Oldenburg, M. Juarez, E. Argones Rúa, and C. Diaz, "Mixmatch: Flow matching for mixnet traffic," *Proceedings on Privacy Enhancing Technologies*, vol. 2024, pp. 276–294, 04 2024.

[20] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg, "Effective attacks and provable defenses for website fingerprinting," in *USENIX Security 2014*, K. Fu and J. Jung, Eds. USENIX Association, Aug. 2014, pp. 143–157.

[21] A. Panchenko, F. Lanze, J. Pennekamp, T. Engel, A. Zinnen, M. Henze, and K. Wehrle, "Website fingerprinting at internet scale," in *NDSS 2016*. The Internet Society, Feb. 2016.

[22] A. Kwon, M. AlSabah, D. Lazar, M. Dacier, and S. Devadas, "Circuit fingerprinting attacks: Passive deanonymization of tor hidden services," in *USENIX Security 2015*, J. Jung and T. Holz, Eds. USENIX Association, Aug. 2015, pp. 287–302.

[23] Z. Luo, A. Bhat, K. Nayak, and A. Kate, "Attacking and improving the tor directory protocol," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.

[24] Harvx1010. (2024) 5 best anonymous social media platforms of 2024: Embracing privacy and expression in the digital age. [Online]. Available: https://at-harvx1010.medium.com/5-best-anonymous-social-media-platforms-of-2024-embracing-privacy-and-expression-in-the-digital-b63cc30acf4c

[25] S. Langowski, S. Servan-Schreiber, and S. Devadas, "Trellis: Robust and scalable metadata-private anonymous broadcast," Cryptology ePrint Archive, Paper 2022/1548, 2022.

[26] M. Ando, A. Lysyanskaya, and E. Upfal, "Practical and Provably Secure Onion Routing," in *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, pp. 144:1–144:14.

[27] A. Kwon, D. Lu, and S. Devadas, "XRD: Scalable messaging system with cryptographic privacy," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

[28] D. Lazar, Y. Gilad, and N. Zeldovich, "Yodel: Strong metadata security for voice calls," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19, 2019, p. 211–224.

[29] D. Das, C. Diaz, A. Kiayias, and T. Zacharias, "Are continuous stop-and-go mixnets provably secure?" *Proc. Priv. Enhancing Technol.*, vol. 2024, pp. 665–683, 2024.

[30] H. Corrigan-Gibbs, D. Boneh, and D. Mazières, "Riposte: An anonymous messaging system handling millions of users," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 321–338.

[31] D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, A. Kate, and A. Miller, "Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 887–903.

[32] V. Madathil, A. Scafuro, I. A. Seres, O. Shlomovits, and D. Varlakov, "Private signaling," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[33] D. Das, S. Meiser, E. Mohammadi, and A. Kate, "Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency - choose two," in *2018 IEEE Symposium on Security and Privacy (SP)*. San Francisco, California, USA: IEEE Computer Society, 2018, pp. 108–126.

[34] D. Das, S. Meiser, E. Mohammadi, and A. Kate, "Comprehensive anonymity trilemma: User coordination is not enough," *Proceedings on Privacy Enhancing Technologies*, vol. 2020, no. 3, pp. 356–383, 2020.

[35] T. Ruffing, P. Moreno-Sanchez, and A. Kate, "P2P Mixing and Unlinkable Bitcoin Transactions," in *NDSS17*, 2017.

[36] L. Barman, I. Dacosta, M. Zamani, E. Zhai, A. Pyrgelis, B. Ford, J. Feigenbaum, and J.-P. Hubaux, "Prifi: Low-latency anonymity for organizational networks," *Proceedings on Privacy Enhancing Technologies*, vol. 2020, pp. 24–47, 10 2020.

[37] A. Hamlin, R. Ostrovsky, M. Weiss, and D. Wichs, "Private anonymous data access," in *Advances in Cryptology – EUROCRYPT 2019*, Y. Ishai and V. Rijmen, Eds. Springer International Publishing, 2019, pp. 244–273.

[38] G. Beck, J. Len, I. Miers, and M. Green, "Fuzzy message detection," in *ACM CCS 2021*, G. Vigna and E. Shi, Eds. ACM Press, Nov. 2021, pp. 1507–1528.

[39] Z. Liu and E. Tromer, "Oblivious message retrieval," in *CRYPTO 2022, Part I*, ser. LNCS, Y. Dodis and T. Shrimpton, Eds., vol. 13507. Springer, Heidelberg, Aug. 2022, pp. 753–783.

[40] S. Peceny, S. Raghuraman, P. Rindal, and H. Shah, "Efficient permutation correlations and batched random access for two-party computation," Cryptology ePrint Archive, Paper 2024/547, 2024. [Online]. Available: https://eprint.iacr.org/2024/547

[41] E. Stefanov and E. Shi, "ObliviStore: High performance oblivious cloud storage," in *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2013, pp. 253–267.

[42] M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious ram simulation," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2011, pp. 576–587.

[43] E. Stefanov and E. Shi, "Multi-cloud oblivious storage," in *ACM CCS 2013*, A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM Press, Nov. 2013, pp. 247–258.

[44] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder, "Maliciously secure multi-client ORAM," in *ACNS 17*, ser. LNCS, D. Gollmann, A. Miyaji, and H. Kikuchi, Eds., vol. 10355. Springer, Heidelberg, Jul. 2017, pp. 645–664.

[45] W. Chen and R. A. Popa, "Metal: A metadata-hiding file-sharing system," in *NDSS 2020*. The Internet Society, Feb. 2020.

[46] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis, "Secure two-party computation in sublinear (amortized) time," in *ACM CCS 2012*, T. Yu, G. Danezis, and V. D. Gligor, Eds. ACM Press, Oct. 2012, pp. 513–524.

[47] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[48] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O((\log N)^3)$ worst-case cost," in *ASIACRYPT 2011*, ser. LNCS, D. H. Lee and X. Wang, Eds., vol. 7073. Springer, Heidelberg, Dec. 2011, pp. 197–214.

[49] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," in *ACM CCS 2013*, A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM Press, Nov. 2013, pp. 299–310.

[50] X. Wang, T.-H. H. Chan, and E. Shi, "Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound," in *ACM CCS 2015*, I. Ray, N. Li, and C. Kruegel, Eds. ACM Press, Oct. 2015, pp. 850–861.

[51] J. Doerner and a. shelat, "Scaling ORAM for secure computation," in *ACM CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM Press, Oct. / Nov. 2017, pp. 523–535.

[52] A. Vadapalli, R. Henry, and I. Goldberg, "Duoram: A {Bandwidth-Efficient} distributed {ORAM} for 2-and 3-party computation," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 3907–3924.

[53] W. Zhang, X. Guo, K. Yang, R. Zhu, Y. Yu, and X. Wang, "Efficient actively secure DPF and RAM-based 2PC with one-bit leakage," in *2024 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2024, pp. 561–577.

[54] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing," in *EUROCRYPT 2015, Part II*, ser. LNCS, E. Oswald and M. Fischlin, Eds., vol. 9057. Springer, Heidelberg, Apr. 2015, pp. 337–367.

[55] ——, "Function secret sharing: Improvements and extensions," in *ACM CCS 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM Press, Oct. 2016, pp. 1292–1303.

[56] B. H. Falk, D. Noble, and R. Ostrovsky, "Alibi: A flaw in cuckoo-hashing based hierarchical ORAM schemes and a solution," in *EUROCRYPT 2021, Part III*, ser. LNCS, A. Canteaut and F.-X. Standaert, Eds., vol. 12698. Springer, Heidelberg, Oct. 2021, pp. 338–369.

[57] X. S. Wang, Y. Huang, T.-H. H. Chan, a. shelat, and E. Shi, "SCORAM: Oblivious RAM for secure computation," in *ACM CCS 2014*, G.-J. Ahn, M. Yung, and N. Li, Eds. ACM Press, Nov. 2014, pp. 191–202.

[58] N. Chandran, D. Gupta, and A. Shah, "Circuit-PSI with linear complexity via relaxed batch OPPRF," *PoPETs*, vol. 2022, no. 1, pp. 353–372, Jan. 2022.

[59] D. Catalano and D. Fiore, "Using linearly-homomorphic encryption to evaluate degree-2 functions on encrypted data," in *ACM CCS 2015*, I. Ray, N. Li, and C. Kruegel, Eds. ACM Press, Oct. 2015, pp. 1518–1529.

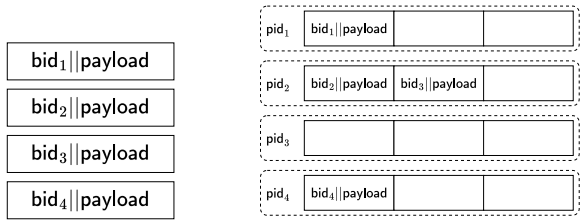
[60] "Openssl," 2024, https://www.openssl.org/source/gitrepo.html.

[61] "Floram: Gitlab repository," 2024. [Online]. Available: https://gitlab.com/neucrypt/floram

[62] R. Motwani and P. Raghavan, *Randomized algorithms*. Cambridge university press, 1995.

Appendix A. Why Not Choose List-based ORAM

From the analysis in Section 2.4, we have known that when using tree-based ORAM, there is no need to consider any additional collision issues beyond block IDs. However, this is not true for list-based ORAM.



(a) Original version (b) Expanded version

Figure 13: List-based ORAM

The data structure of the list-based ORAM is shown in Figure 13a. Unlike the tree-based ORAM where a path can contain multiple real blocks and numerous dummy blocks facilitate the movement of real blocks along corresponding paths, list-based ORAM allocates only one position for each real block, as illustrated in Figure 13a. Therefore, even if the senders can choose the block IDs from a large set to avoid collisions, the number of physical positions is still just n, which will lead to collisions with overwhelming probability. To solve the problem, we need to expand the capacity of each position to allow each position to contain multiple blocks, as shown in Figure 13b. In this way, the positions in the expanded list-based ORAM are equivalent to the paths in the tree-based ORAM. Therefore, even if two senders choose the same position, their messages can still be written into ORAM successfully. To access a block in the expanded list-based ORAM, in addition to the block ID bid, a position ID pid also needs to be provided. Then, the servers first find the position according to pid and then search for the block according to bid in this position. A natural question arises: how many blocks are needed in a position to guarantee that the failure probability is negligible?

Obviously, this is a balls-into-bins problem, and we can use the following inequality [62]

$$\Pr[\exists \text{ position with } \geq \rho \text{ items}] \leq q\left[\sum_{i=\rho}^{n}\binom{n}{i}\cdot\left(\frac{1}{q}\right)^{i}\cdot\left(1-\frac{1}{q}\right)^{n-i}\right]$$

to set the maximum number ρ of blocks in a position, such that no position will contain more than ρ blocks except with probability $2^{-\lambda}$, when the number of positions is q and the number of total real blocks is n. Specifically, to maintain 2^{20} unretrieved messages and achieve at most 2^{-40} failure probability, we can set $\rho = 20$, which will lead to a significant performance decline.

Appendix B. Deferred Proofs about ORAM^{-}

B.1. Security Proof for Obliviousness

Theorem. If the underlying tree-based ORAM is a secure ORAM, then our construction shown in Algorithm 2 is a secure ORAM^{-}.

Proof. We have proved the correctness in Section 5.1, next we prove obliviousness.

Obliviousness. If there is an adversary $\mathcal{A}$ that can distinguish two sequences of physical memory accesses generated by two sequences of operations run by ORAM^{-}. Then, we can construct another adversary $\mathcal{B}$ that can distinguish two sequences of physical memory accesses generated by two sequences of operations run by ORAM. Next, we will construct the adversary $\mathcal{B}$.

The adversary $\mathcal{B}$ maintains two tables $\mathcal{T}^{(0)}$ and $\mathcal{T}^{(1)}$ to store the linkability between an index for ORAM and an index for ORAM^{-}, and a bit indicating if an index for ORAM corresponds to a valid message. Specifically, each entry of the tables is a tuple $(id_{\mathsf{ORAM}}, id_{\mathsf{ORAM}^{-}}, b)$ where $b = 1$ indicates that there is a message associated with id_{ORAM}. The adversary $\mathcal{B}$ initializes the tables $\mathcal{T}^{(0)}$ and $\mathcal{T}^{(1)}$ by storing $(id_{\mathsf{ORAM}}, 0, 0)$ for each index id_{ORAM} of ORAM.

Once receiving two operations $(\mathsf{Op}_k^{(0)}, i_k^{(0)}, m_k^{(0)})\}$ and $(\mathsf{Op}_k^{(1)}, i_k^{(1)}, m_k^{(1)})\}$ from the adversary $\mathcal{A}$, the adversary $\mathcal{B}$ generates another two operations $(\tilde{\mathsf{Op}}_k^{(0)}, \tilde{i}_k^{(0)}, \tilde{m}_k^{(0)})$ and $(\tilde{\mathsf{Op}}_k^{(1)}, \tilde{i}_k^{(1)}, \tilde{m}_k^{(1)})$, based on the following rule.

- If $\mathsf{Op}_k^{(0)} = \mathsf{Op}_k^{(1)} = \mathsf{Write}$:
  - The adversary $\mathcal{B}$ randomly chooses an index $\tilde{i}_k^{(0)}$ (resp. an index $\tilde{i}_k^{(1)}$) whose tuple is $(\tilde{i}_k^{(0)}, 0, 0)$ (resp. $(\tilde{i}_k^{(1)}, 0, 0)$) in the table $\mathcal{T}^{(0)}$ (resp. $\mathcal{T}^{(1)}$), and changes the tuple to $(\tilde{i}_k^{(0)}, i_k^{(0)}, 1)$ (resp. $(\tilde{i}_k^{(1)}, i_k^{(1)}, 1)$);
  - The adversary $\mathcal{B}$ sets $\tilde{\mathsf{Op}}_k^{(0)} = \tilde{\mathsf{Op}}_k^{(1)} = \mathsf{Write}$;
  - The adversary $\mathcal{B}$ sets $\tilde{m}_k^{(0)} = m_k^{(0)}$ and $\tilde{m}_k^{(1)} = m_k^{(1)}$.
- If $\mathsf{Op}_k^{(0)} = \mathsf{Op}_k^{(1)} = \mathsf{Read}$:
  - If there is a tuple $(id^{(j)}, i_k^{(j)}, 1)$ in $\mathcal{T}^{(j)}$:
    - The adversary $\mathcal{B}$ sets $\tilde{i}_k^{(j)} = id^{(j)}$, and changes the tuple to $(id^{(j)}, 0, 0)$;
    - The adversary $\mathcal{B}$ sets $\tilde{\mathsf{Op}}_k^{(j)} = \mathsf{Write}$;
    - The adversary $\mathcal{B}$ sets $\tilde{m}_k^{(j)} = 0$.
  - If there is not a tuple $(id^{(j)}, i_k^{(j)}, 1)$ in $\mathcal{T}^{(j)}$:
    - The adversary $\mathcal{B}$ randomly chooses $\tilde{i}_k^{(j)}$ that is not in the index space of ORAM;
    - The adversary $\mathcal{B}$ sets $\tilde{\mathsf{Op}}_k^{(j)} = \mathsf{Write}$;
    - The adversary $\mathcal{B}$ sets $\tilde{m}_k^{(j)} = 0$.

Then, the adversary $\mathcal{B}$ sends $(\tilde{\mathsf{Op}}_k^{(0)}, \tilde{i}_k^{(0)}, \tilde{m}_k^{(0)})$ and $(\tilde{\mathsf{Op}}_k^{(1)}, \tilde{i}_k^{(1)}, \tilde{m}_k^{(1)})$ to the challenger of ORAM. The challenger will randomly choose a bit c, and run $(\tilde{\mathsf{Op}}_k^{(c)}, \tilde{i}_k^{(c)}, \tilde{m}_k^{(c)})$ on ORAM to generate the physical memory access sequence AccSeq and send back AccSeq to the adversary $\mathcal{B}$. Then, the adversary $\mathcal{B}$ can construct another physical memory access sequence AccSeq' according to AccSeq, and sends AccSeq' to the adversary $\mathcal{A}$.

Next, we explain that how the adversary $\mathcal{B}$ can construct AccSeq' for ORAM^{-} according to AccSeq.

For a Write operation on ORAM^{-}, the physical memory access includes: (1) read the stash and a path for eviction, and (2) write blocks into the stash and the path after performing writing and eviction. The physical memory access for ORAM is exactly the same.

For a Read operation on ORAM^{-}, the physical memory access includes: (1) read the stash and a corresponding path, and (2) write blocks into the stash and the path after releasing the corresponding block. In ORAM, although there is an eviction process, the physical memory access is exactly the same. If there is no active block for the index, both ORAM and ORAM^{-} do not generate the physical memory access.

Finally, the adversary $\mathcal{A}$ outputs a bit c', and then the adversary $\mathcal{B}$ forwards c' to the challenger. Since $\mathsf{AccSeq} = \mathsf{AccSeq}'$, if the adversary $\mathcal{A}$ can distinguish the sequences for ORAM^{-} with non-negligible probability, the adversary $\mathcal{B}$ can distinguish the sequences for ORAM with the same probability. □

Appendix C. Deferred Proofs about Protocols

C.1. Ideal Functionality

C.2. Security Proof for the Complete Protocol

Theorem. Assuming the hardness of Discrete-logarithm problem, computational security and correctness of ORAM^{-}, and at least one of the servers is honest, our protocol $\Pi = (\Pi_{\mathsf{Initialize}}, \Pi_{\mathsf{Send}}, \Pi_{\mathsf{Retrieve}})$ UC-realizes the ideal functionality $\mathcal{F}_{anon}$ in the $\{\mathcal{F}_{\mathsf{PET}}, \mathcal{F}_{\mathsf{Shuffle}}, \mathcal{F}_{\mathsf{SC-ORAM}^{-}}\}$-hybrid world.

Proof. To prove that our protocol UC-realizes the $\mathcal{F}_{anon}$ functionality, we show that there exists a simulator $\mathcal{S}_{\mathtt{full}}$

Functionality $\mathcal{F}_{\mathsf{SC-ORAM}^-}$

1: Upon receiving $((\llbracket \mathsf{bid} \rrbracket_j, \llbracket \mathsf{lid} \rrbracket_j)_{j\in 1,2}, \text{READ})$ from the servers:
   (a) Run ORAM^-.READ with inputs $(\llbracket \mathsf{bid} \rrbracket_1 + \llbracket \mathsf{bid} \rrbracket_2$, $\llbracket \mathsf{lid} \rrbracket_1 + \llbracket \mathsf{lid} \rrbracket_2$) as the ORAM^- server.
   (b) Return the output and all the transcript generated from step 1.(a) to the servers
2: Upon receiving $((\llbracket \mathsf{bid} \rrbracket_j, \llbracket \mathsf{lid} \rrbracket_j)_{j\in 1,2}, \text{WRITE}, \mathsf{msg})$ from the servers:
   (a) Run ORAM^-.WRITE with inputs $(\llbracket \mathsf{bid} \rrbracket_1 + \llbracket \mathsf{bid} \rrbracket_2$, $\llbracket \mathsf{lid} \rrbracket_1 + \llbracket \mathsf{lid} \rrbracket_2$, msg) as the ORAM^- server.
   (b) Return all the transcript generated from step 2.(a) to the servers.

Figure 14: Ideal functionality realizing ORAM^- in secure computation setting.

interacting with $\mathcal{F}_{anon}$ functionality that generates a transcript that is indistinguishable from the transcript generated by the real-world adversary $\mathcal{A}$ in the protocol Π. We present the description of the simulator in Figs. 15 and 16.

We show that the transcript of the adversary in the real-world and in the simulated world are indistinguishable by presenting hybrids that are indistinguishable for the adversary:

- **Hybrid0**: The real-world protocol.
- **Hybrid1**: This hybrid is same as the previous hybrid, except that the random tapes of the corrupted server and the corrupted clients are chosen by the simulator. Since, the adversary is semi-honest, this is indistinguishable from the previous hybrid.
- **Hybrid2**: This hybrid is identical to the previous one, except that the keypairs for the honest server and the $\mathsf{Addr}_{\mathsf{Rcv}}$ values for the honest recipients are generated by the simulator. Any communication from these parties will be intercepted by the simulator, and simulator will generate random communication to replace the original communications. These hybrids are indistinguishable since one of the servers is honest, and based on the hardness of DL-problem.
- **Hybrid3**: This hybrid is identical to the previous hybrid, except that the simulator invokes $\mathcal{F}_{\mathsf{SC-ORAM}^-}$ to execute ORAM^-.WRITE and ORAM^-.READ operations using the manufactured inputs as in Figs. 15 and 16. Based on the computational security of our ORAM^- scheme (see Theorem 2), the accesses with manufactured input are indistinguishable from the access pattern of the original protocol; moreover, $\mathcal{F}_{\mathsf{SC-ORAM}^-}$ guarantees that the original input is never revealed, and our simulator $\mathcal{S}$ ensures that the volume information is preserved. Therefore, the hybrids are indistinguishable.

□

Simulator $\mathcal{S}_{\mathtt{full}}$

Simulating $\Pi_{\mathsf{Initialize}}$:

1) w.l.o.g. assuming Server_2 is corrupted.
2) On behalf of honest Server_1 generate $(\mathsf{pk}_1, \mathsf{sk}_1)$, and broadcast $(\mathtt{PK}, \mathsf{pk}_1)$. (Note that our simulator $\mathcal{S}_s$ knows the key pairs of the honest server.)
3) The simulator also chooses the random tape for Server_2.
4) On behalf of each honest recipient Rcv_i, sample $k_{R_i} \leftarrow \mathbb{Z}_p$ and compute $\mathsf{Addr}_{R_i} = g^{k_{R_i}}$ and broadcast $(\mathtt{ADDR}, \mathsf{Addr}_{R_i})$.
5) Wait for the adversary $\mathcal{A}$ to send $(\mathtt{PK}, \mathsf{pk}_2)$ on behalf of Server_2 and $(\mathtt{ADDR}, \mathsf{Addr}_{R_i})$ on behalf of each corrupted recipient.

Simulating Π_{Send}:

1) Upon receiving $(\mathsf{Send}, u_i, \_\_, \_\_)$ from $\mathcal{F}_{anon}$ functionality (for an honest sender u_i): The simulator generates the shares for the servers by running the code of the client, however, without knowing the actual message or the recipient. And that requires the following modifications.
   a) Generate $[m]_1 + [m]_2 = 0$, $r \xleftarrow{\$} \mathbb{Z}_p$, $R = g^r$, and $A = (\mathsf{Addr}_{\mathsf{Rcv}})^r$, for a randomly chosen recipient Rcv; further, randomly split A into A_1 and A_2 such that $A_1 \cdot A_2 = A$.
   b) Now run the remaining client code of Π_{Send} to send the shares $((\llbracket \mathsf{bid} \rrbracket_j, \llbracket \mathsf{lid} \rrbracket_j, [m]_j), (A_j, R))$ to the respective servers. (Note that, the communications to Server_1 are only for the purpose of generating transcript for the real-world adversary $\mathcal{A}$, any output generated by Server_1 is ignored by $\mathcal{S}$.)
   c) Additionally, store the tuple $([m]_1, \llbracket \mathsf{bid} \rrbracket_1, \llbracket \mathsf{lid} \rrbracket_1, A_1, R, r, 0)$ in a local table $\mathcal{T}_1$, and $([m]_2, \llbracket \mathsf{bid} \rrbracket_2, \llbracket \mathsf{lid} \rrbracket_2, A_2)$ in another local table $\mathcal{T}_2$.
2) Upon receiving $((((\llbracket \mathsf{bid} \rrbracket_j, \llbracket \mathsf{lid} \rrbracket_j, [m]_j), (A_j, R))_{j\in 1,2}, \mathsf{msg}, \mathsf{Addr}_{\mathsf{Rcv}})$ from $\mathcal{A}$ (for a corrupted sender u_i):
   a) Store the tuple $([m]_1, \llbracket \mathsf{bid} \rrbracket_1, \llbracket \mathsf{lid} \rrbracket_1 A_1, R, r, \mathsf{msg})$ in the local table $\mathcal{T}_1$, and $([m]_2, \llbracket \mathsf{bid} \rrbracket_2, \llbracket \mathsf{lid} \rrbracket_2, A_2)$ in another local table $\mathcal{T}_2$.
   b) Send $(\mathtt{SEND}, u_i, \mathsf{Addr}_{\mathsf{Rcv}}, \mathsf{msg})$ to the ideal functionality $\mathcal{F}_{anon}$
3) **Preparing for retrievals:**
   a) Run the server part of Π_{Send} protocol on behalf of Server_1 (see Fig. 10).
   b) In the last step, invoke $\mathcal{F}_{\mathsf{SC-ORAM}^-}$ to execute ORAM^-.WRITE with input $(\llbracket \mathsf{bid} \rrbracket_1, \llbracket \mathsf{lid} \rrbracket_1, [m]_1)$ together with Server_2.

Figure 15: Simulating Send when the servers use ORAM^-

Simulator $\mathcal{S}_{\texttt{full}}$

Simulating Π_{Retrieve}:

1) Upon receiving $(\mathsf{Retrieve}, k_{\mathsf{Rcv}_j,1}, k_{\mathsf{Rcv}_j,2}, \mathsf{Rcv}_j)$ from $\mathcal{A}$ where Rcv_j is corrupted:
   a) Send $(\mathsf{Retrieve}, \mathsf{Rcv}_j)$ to $\mathcal{F}_{anon}$. Let $\mathcal{M}$ be the response from $\mathcal{F}_{anon}$.
   b) Run the code of the two servers locally using $\mathcal{T}_1$ and $\mathcal{T}_2$ to generate the output vectors B_1 and B_2 corresponding to the servers. Suppose, B_1 and B_2 collectively yields the set of messages $\mathcal{M}'$ for $\mathsf{Addr}_{\mathsf{Rcv}_j}$.
   c) For each element m in $\mathcal{M} \setminus \mathcal{M}'$: find a tuple $T = ([m]_1, [\mathsf{bid}]_1, [\mathsf{lid}]_1, A_1, R, r, \mathsf{msg})$ in $\mathcal{T}_1$ such that $\mathsf{msg} = 0$ and $A_1 \cdot A_2 \neq (\mathsf{Addr}_{\mathsf{Rcv}_j})^r$. Update $A_1 = (\mathsf{Addr}_{\mathsf{Rcv}_j})^r / A_2$, update $[m]_1 = \mathsf{msg} - [m]_2$, and set $\mathsf{msg} = m$.
   d) For each element m in $\mathcal{M}' \setminus \mathcal{M}$: find the tuple $T = ([m]_1, [\mathsf{bid}]_1, [\mathsf{lid}]_1, A_1, R, r, \mathsf{msg})$ in $\mathcal{T}_1$ such that $\mathsf{msg} = 0$ and $A_1 \cdot A2 = (\mathsf{Addr}_{\mathsf{Rcv}_j})^r$. Update $A_1 = (\mathsf{Addr}_{\mathsf{Rcv}_w})^r / A_2$, for a randomly chosen recipient $\mathsf{Rcv}_w \neq \mathsf{Rcv}_j$.
   e) Run the code for Server_1 (see Fig. 11) again with updated $\mathcal{T}_1$ as part of the protocol with Server_2, in order to generate new output vector $\vec{g}'_1$, and wait for $\mathcal{A}$ to generate $\vec{g}'_2$ on behalf of Server_2.
   f) For each $g = b'_{k,2}||[\mathsf{bid}]'_{k,2}||[\mathsf{lid}]'_{k,2} \in \vec{g}'_1$: if $(b'_{k,2} \oplus b'_{k,1}) = 1$, invoke $\mathcal{F}_{\mathsf{SC-ORAM}^-}$ to execute ORAM^-.READ with input $([\mathsf{bid}]'_{k,1}||[\mathsf{lid}]'_{k,1}, [\mathsf{bid}]'_{k,2}||[\mathsf{lid}]'_{k,2})$ acting as Server_1.

2) Upon receiving $(\mathsf{Retrieve}, \mathsf{Rcv}_j, k)$ from $\mathcal{F}_{anon}$:
   a) Generate A_1 and A_2 such that $A1 \cdot A2 = (\mathsf{Addr}_{\mathsf{Rcv}_j})^r$ for $r \xleftarrow{\$} \mathbb{Z}_p$.
   b) Run the code of the two servers locally using $\mathcal{T}_1$ and $\mathcal{T}_2$ to generate the output vectors B_1 and B_2 corresponding to the servers. Suppose, B_1 and B_2 collectively yields k' messages $\mathsf{Addr}_{\mathsf{Rcv}_j}$.
   c) If $k' > k$, find $k' - k$ tuples in $\mathcal{T}_1$ such that, for each of them, $\mathsf{msg} = 0$ and $A_1 \cdot A2 = (\mathsf{Addr}_{\mathsf{Rcv}_j})^r$. Update $A_1 = (\mathsf{Addr}_{\mathsf{Rcv}_w})^r / A_2$, for a randomly chosen recipient $\mathsf{Rcv}_w \neq \mathsf{Rcv}_j$.
   d) If $k' < k$, find $k' - k$ tuples in $\mathcal{T}_1$ such that, for each of them, $\mathsf{msg} = 0$ and $A_1 \cdot A_2 \neq (\mathsf{Addr}_{\mathsf{Rcv}_j})^r$. Update $A_1 = (\mathsf{Addr}_{\mathsf{Rcv}_j})^r / A_2$, and set $\mathsf{msg} = 1$.
   e) Run the code for Server_1 (see Fig. 11) again with updated $\mathcal{T}_1$ as part of the protocol with Server_2, in order to generate transcript for the adversary. (Note that the honest recipient has already received the actual output from $\mathcal{F}_{anon}$ and can ignore any output produced by the simulator.)

Figure 16: Simulating Retrieve when the servers use ORAM^- for performance improvement