University of Glasgow | School of Computing Science

Universally Composable Anonymous Broadcast Protocols

Kyla Nel
2576953n

Abstract

Preserving anonymity in online interactions is essential for safeguarding personal privacy, enabling secure communication, and supporting applications such as whistleblowing platforms, anonymous voting systems, and privacy-focused messaging. Anonymous broadcast protocols (ABPs) allow message delivery without revealing the sender's identity. Despite their utility, few ABPs have been proven secure under the Universal Composability (UC) framework, which provides robust security guarantees under concurrent executions.

This project aims to prove the security of an existing ABP under the UC framework. We focus on Riposte, an anonymous messaging system designed to resist traffic-analysis attacks and provide disruption resistance through zero-knowledge proofs. Our work thus far includes a detailed ideal functionality for anonymous broadcast based on Riposte. This research provides a step toward integrating provably secure ABPs into broader privacy-preserving systems.

1 Introduction

Our world is increasingly dominated by surveillance and data tracking. Preserving anonymity in interactions online is crucial not only to restricting surveillance and traffic analysis opportunities, but to other domains like protecting personal privacy in communication and enabling secure and private transactions. Anonymous broadcast protocols (ABPs) are communication protocols designed to protect the anonymity of communicators. These protocols prioritise anonymity, unlikability, and unobservability while establishing secure communication channels. This allows for the successful delivery of the same message to all recipients without revealing the identity of the sender.

ABPs have numerous applications. Chief among these are whistleblowing platforms that allow participants to submit sensitive information anonymously, anonymous voting systems (e-voting), decentralised cryptocurrencies, and privacy-focused messaging apps.

While several ABPs exist which provide varying levels of anonymity and security, they have largely not been proven secure under a robust security framework like the Universal Composability (UC) framework, which preserves security under concurrent protocol executions. Doing so would establish an ABP which could be used in a modular design of larger systems such as e-voting software, while essentially being treated as a "black box".

Case 1: We first examine the case where Server A is corrupted and $\mathcal{S}$ controls the honest Server B. Here $\mathcal{A}$ requests to corrupt Server A in the hybrid world and so S corrupts Server A in the ideal world. Upon receiving (**sid**, **WRITE**, $|M|$, P) from $\mathcal{F}_R^K$ where P is an honest party, $\mathcal{S}$ simulates a **WRITE** request with a dummy all-zero message $M_0 = 0^{|M|}$ on behalf of P. In this case, $\mathcal{S}$ simulates the protocol as written, using the dummy message M_0 to generate e_{ℓ,M_0}. LEAK DATA. Upon receiving (**sid**, **WRITE**, $|M^*|$, P^*) from $\mathcal{A}$ where P^* is a corrupted party, $\mathcal{S}$ reconstructs M^* from the two shares sent to Server A and Server B respectively across $\mathcal{F}_{AEC}(\{A, B\})$. This is made possible by the fact that parties are passively corrupted and so are forced to communicate using $\mathcal{F}_{AEC}$ exclusively, which $\mathcal{S}$ controls. Having reconstructed M^*, $\mathcal{S}$ then sends a WRITE request (**SID**, **WRITE**, M^*, P^*) to the functionality. Upon receiving (**sid**, **BROADCAST**, $< M_1, ..., M_K >$) from $\mathcal{F}_R^K$, $\mathcal{S}$ extracts the subset of honest messages from $< M_1, ..., M_K >$ and randomly assigns the honest messages to honest parties. For each honest party P_i with associated message M_i, $\mathcal{S}$ constructs e_{ℓ,M_i} and creates the share $e_{\ell,M_i} \oplus r$. $\mathcal{S}$ sends r to Server A and e_{ℓ,M_i} to Server B. At this stage, $\mathcal{S}$ must also simulate the WRITE request sent by the party P_K which is associated with M_K, the K-th message added to L_{pend}. This follows the steps outlined above, depending on whether P_K is honest or corrupted.

Case 2: Next, we examine the case where Server B is corrupted and $\mathcal{S}$ controls the honest Server A. In this case, $\mathcal{S}$ follows the same steps as in Case 1, however, $\mathcal{S}$ equivocates by sending any random bitsring r to Server B. After the assignment of honest messages to honest partiews, $\mathcal{S}$ constructs a consistent $e_{\ell,M}$ for every honest message M and sneds $e_{\ell,M} \oplus r$ to Server B.

In both cases, whatever is sent by the corrupted server when the servers combine their states, the results will be statistically indistuinguishable from the result if both servers were honest.

2 Background

There are several different types of ABPs which achieve varying levels of anonymity, security, efficiency, and scalability. Mix networks (mix-nets) send messages through a series of nodes (servers) that shuffle and re-encrypt them [9]. They provide strong anonymity and resistance to traffic analysis but suffer from high latency. Onion routing protocols encrypt messages in layers which are successively decrypted by intermediate nodes [8]. These protocols are common in the wild as they are efficient. However, they are vulnerable to traffic analysis. Dining cryptographers networks (DC-nets) allow clients to create a shared communication channel that hides the identity of the sender [4]. DC-nets provide perfect broadcast anonymity but suffer from scalability and high communication overhead.

Multiple attempts have been made at creating commercial implementations of ABPs. Herbivore [7] and Dissent [6] are both implementations of the DCN protocol. These were early attempts at commercial implementations and both lack the scalability required for commercial deployment. Dissent in Numbers [11] builds on Dissent and shifts from a decentralised setting to a server-client architecture. This improves performance but introduces the problem of handling server failures. The introduction of the server-client architecture also requires at least one trusted server as the anonymity provider. Riposte [5] is another implementation that utilises a server-client architecture. They provide two versions of the model. The first version utilises three servers and requires at least two of them to be honest. This provides scalability but has drawbacks in terms of the security provided. The second version adopts an any-trust model that provides more robust security but comes with higher bandwidth cost.

The Universal Composability (UC) framework is designed to prove robust security for cryptographic protocols [3]. It guarantees that a protocol is secure regardless of how it interacts with other subsystems when it is composed with other UC-secure protocols; for example, when a protocol is executed in complex, adversarial environments. In this framework, security is proven by comparing the real protocol to an ideal functionality (a theoretical model that perfectly performs the task at hand). If the results of the ideal functionality and the protocol are statistically indistinguishable, the protocol is considered to UC-realise the functionality and is deemed secure. This framework implements simulation-based security. It considers a theoretical simulator which can mimic any adversary. If it can be shown that any attack in the real protocol can be captured by the simulator in the ideal world without breaking the properties of the ideal functionality, then the protocol is considered secure.

UC treatment of the anonymous broadcast problem is as yet a relatively unexplored field. Arapinis et al. [1] constructed a universally composable e-voting system using anonymous broadcast. Vyrostko [10] created a universally composable anonymous broadcast channel based on Dining Cryptographers Networks (DC-nets). This construction was limited in that only one message could be sent by only one client every round.

3 Building Blocks

3.1 Authenticated Encrypted Channel

AE channel functionality $\mathcal{F}_{\mathcal{AEC}}(\{A, B\})$

Initialise a list $PendingMsg \leftarrow \emptyset$.
■ Upon receiving (**sid**, **SEND**, M) from P, if P is honest, then:

(1) If $\{A, B\} \backslash \{P\}$ is corrupted, then send (**sid**, **SEND**, M, P) to $\mathcal{S}$.
(2) If $\{A, B\} \setminus \{P\}$ is honest, then
   - Choose a random tag $\xleftarrow{\$} \{0,1\}^{\lambda}$.
   - Add (**tag**, M, P) to $PendingMsg$
   - Send (**sid**, **SEND**, **tag**, $|M|$, P, {A,B} \ {P}) to $\mathcal{S}$.
(3) Upon receiving (**sid**, **ALLOW**, **tag**) from $\mathcal{S}$, if there is a (**tag**, M, P) in $PendingMsg$, then remove (**tag**, M, P) from $PendingMsg$ and send (**sid**,**SEND**,M) to {A,B}\{P}

Figure 1: Anonymous broadcast ideal functionality.

3.2 Non-interactive Zero-knowledge

Non-interactive zero knowledge functionality $\mathcal{F}^{R}_{NIZK}$

(1) **Proof:** On input (prove, **sid**, x, w) from party P: if $\mathcal{R}(x, w) = 1$ then send (prove, P, **sid**, x) to $\mathcal{S}$. Upon receiving (proof, **sid**, π) from $\mathcal{S}$, store (**sid**, x, w, π) and send (proof, **sid**, π) to P.
(2) **Verification:** On input (verify, **sid**, x, π) from a party V: If (**sid**, x, w, π) is stored, then return (verification, **sid**, x, π, $\mathcal{R}(x, w)$) to V. Else, send (verify, V, **sid**, x, π) to $\mathcal{S}$. Upon receiving, (witness, **sid**, w) from $\mathcal{S}$, store (**sid**, x, w, π) and return (verification, **sid**, x, π, $\mathcal{R}(x, w)$) to V.
(3) **Corruption:** When receiving (corrupt, **sid**) from $\mathcal{S}$, mark **sid** as corrupted. If there is a stored tuple (**sid**, x, w, π), then send it to $\mathcal{S}$.

Figure 2: Non-interactive zero knowledge functionality.

3.3 Broadcast

Broadcast functionality $\mathcal{F}_{BC}$

The functionality interacts with an adversary $\mathcal{S}$ and a set $\mathcal{P} = \{P_1, ..., P_n\}$ of parties.

- Upon receiving (**sid**, **BROADCAST**, M) from P_i, send (**sid**, **BROADCAST**, M) to all parties in $\mathcal{P}$ and $\mathcal{S}$.

Figure 3: Broadcast functionality $\mathcal{F}_{BC}$

4 Riposte in a UC Setting

4.1 System Description

For the purposes of this dissertation, we examine a simple construction of Riposte which utilises two servers. Each server, A and B, stores a table with R rows and C columns, where C is the length of messages being sent, which we assume to be constant. Each "row" of the database table represents an index or space to which a message can be written. This database table is initialised to all zeros. When a party wishes to send a message M, they first choose an index ℓ at random, then generate a random R x C bitstring r and construct an R x C vector $e_{\ell,m}$ which contains zeros in each row except row ℓ, which contains M. The party sends r to Server A and $r \oplus e_{\ell,M}$ to Server B. Here $\oplus$ represents bitwise XOR. Upon receiving the write request from the party, both servers first confirm that the party has not sent a valid write request in the current round already. This is easy for ther servers to check because all communication between servers and parties is conducted through an authenticated encrypted channel. If this is the party's first write request, Server A writes r into its database state and Server B writes $r \oplus e_{\ell,M}$ into its database state. We assume the messages being written to the database are elements in some field $\mathbb{F}$ of prime charactersitic p. When a message is written to the database, it is added to the current database state by addition in $\mathbb{F}$.

After processing n write requests, Server A's database state will be:

$$d_A = r_1 + ... + r_n$$

and Server B's database state will be:

$$d_B = (r_1 \oplus e_{\ell,M_1}) + ... + (r_n \oplus e_{\ell,M_n}).$$

Since both bitwise XOR and addition in $\mathbb{F}$ modulo p are associative operations, Server B's database state can be rearranged as follows:

$$d_B = (r_1 + .. + r_n) \oplus (e_{\ell,M_1} + ... + e_{\ell,M_n})$$

which is clearly the same as

$$d_B = d_A \oplus (e_{\ell,M_1} + ... + e_{\ell,M_n}).$$

The servers will continue to accept write requests until a predetermined number of valid requests K have been received. This marks the end of the current round. At the end of the round, the servers combine their local states d_A and d_B to reveal the plaintext database and broadcast this to all parties. The fact that any one party can only send one write request per round protects the system from denial-of-service attacks. Due to the obfuscation of of each message M and its corresponding index ℓ, neither server, nor the other parties, will know which party sent message M or which location ℓ corresponds to M. Thus, write-privacy is maintained by this system. It should be noted that although this system provides anonymous broadcast with an anonymity set of size K, this comes at the cost of a liveness problem in the system. Since a round does not end until K write requests are received, if fewer than K write requests are received, no messages will be broadcast at all, no matter how much time has elapsed in a current round. This problem is unavoidable if we do not want the anonymity set to be smaller than K.

4.1.1 Collision Handling. When a party P_A wants to write the message $M_A \in \mathbb{F}$ to row ℓ in the database, the party actually writes the pair $(M_A, M_A^2) \in \mathbb{F}^2$ into row ℓ. If no collision occurs, i.e. only one client writes their message to row ℓ in the database, recovering the original message is trivial. It is easy to check when this is the case, since the second coordinate will be the square of the first. In the case of a two-way collision, say party P_B also writes their message pair $(M_B, M_B^2) \in \mathbb{F}^2$ to the same row ℓ in the database. Recall that when a message is written to the database, it is added to the current database state by addition in $\mathbb{F}$. Thus, the message pair now contained in row ℓ of the database is

$$S_1 = M_A + M_B (\mathrm{mod} p)$$

and

$$S_2 = M_A^2 + M_B^2 (\mathrm{mod} p).$$

Observing that

$$2S_2 - S_1^2 = (M_A - M_B)^2 (\mathrm{mod} p)$$

we can obtain $M_A - M_B$ by taking a square root modulo p. Using $S_1 = M_A + M_B$we can easily recover M_A and M_B.

This way of handling two-way collisions can be generalised to k-way collisions for $k > 2$. Each party P_i wanting to write their message M_i into the database instead writes $(M_i, M_i^2, ..., M_i^k) \in \mathbb{F}^k$ to its chosen row ℓ. Now, if a k-way collision occurs, we have k equations in k variables which can be solved efficiently to recover all k messages, as long as the characteristic p of $\mathbb{F}$ is greater than k. If we choose k to be the same size as the number of parties interacting with the system, complete collision resistance can be guaranteed, even if every single party wrote a message to the same database row. However, increasing the size of k increases the memory overhead of the system. In essence, increasing k increases the number of columns required in the database table of each server. In practice, with passive adversaries, the probability of all parties writing to the same row in the database tables is negligible. the choice of k is a tradeoff between how much memory overhead can be accommodated and how low the system designer wants the probability of unrecoverable collisions to be.

4.1.2 Disruption Resistance. Some disruption resistance is already provided by preventing any one party from sending more than one write request per round. However, in the above description of the system, there is nothing preventing a party from sending a malformed write request. For example, an adversary controlling several corrupted parties might instruct each of them to construct a vector e_ℓ containing a message in every row (instead of only in one row with zeros everywhere else as an honest party would do). A corrupted party could then generate a random vector r as described above, and send r to Server A and $r \oplus e_\ell$ to Server B. If enough parties are corrupted, this could flood the system with messages.

To address this, a non-interactive zero-knowledge proof is utilised for detecting disruptors. A non-interactive zero-knowledge proof is a cryptographic primitive which allows a prover to convince a verifier of the validity of some information without revealing anything except the legitimacy of the proof, with the help of a pre-established common random string [2]. A non-interactive proof for any NP language L can be defined using a binary relation $\mathcal{R}$ and a tuple of algorithms (K, P, V). The syntax is as follows:

- **Setup:** $\sigma \leftarrow \mathrm{K}(1^n)$ outputs a common random string
- **Prove:** $\pi \leftarrow \mathrm{P}(\sigma, x, w)$ takes a common random string σ, a statement $x \in L$ and a witness w, and outputs a proof π
- **Verify:** $\mathrm{V}(\sigma, x, \pi)$ outputs 1 if it accepts the proof and 0 otherwise

This primitive is applied to allow parties to prove that their write requests are well formed. In this case, the party (the prover) proves to the verifier (the servers) that the vector $e_{\ell,M}$ they are sending only contains a message in one row, and contains zeros everywhere else. To start we define a vector using the obfuscated message the client requests to send $e_{\ell,M}$ and the number of servers in the system s as follows:

$$\sum_{i=0}^{s-1} \overline{b}_i = e_{\ell,M}.$$

The party will perform non-interactive zero-knowledge proofs over a collection of Pedersen commitments. The public parameters for the Pedersen commitments are two generators P and Q, which generate a group $\mathbb{G}$ of prime order q such that the discrete logarithm $\log_Q P$ is not known to anyone. A Pedersen commitment to an element $u \in \mathbb{Z}_q$ with randomness $r \in \mathbb{Z}_q$ is $C(u, r) = (uP + rQ) \in \mathbb{G}$. The party starts by generating Pedersen commitments to each element to the i-th element correctly. Pedersen commitments are homomorphic. The servers can use this property to generate commitments to the sum of the elements of $\overline{b}_i$. The party finally proves in zero knowledge that the sum has the correct value. The details of this process are as follows:

(1) The party generates a vector of Pedersen commitments $\overline{B}_i$ to each element of $\overline{b}_i$ and sends $\overline{B}$ to every server.
(2) The party sends the opening of the commitments to $\overline{B}_i$ to server i. Every server i verifies that $\overline{B}_i$ is a valid commitment to $\overline{b}_i$. All servers reject the write request if this check fails at any server i.

(3) Due to the homomorphic property of the commitments, every server can compute a vector of commitments $\overline{B}_{sum}$ to $\sum_{i=0}^{s-1} \overline{b}_i$.
(4) The party uses a non-interactive zero-knowledge proof to prove to the servers that 1) $\overline{B}_{sum}$ contains commitments to 0 everywhere except at the secret index ℓ and 2) $\overline{B}_{sum}[\ell]$ is a commitment to 1.

If the proof is valid, the party has successfully proves to the servers that their write request is well formed, i.e. that their write request contains only a single message.

4.2 The Riposte Functionality

The ideal functionality we have constructed based on Riposte follows closely to the original model. The pseudocode for this functionality is presented in Figure 4.

Anonymous broadcast functionality $\mathcal{F}_R^K$

Initialise:

(1) a list of pending messages $L_{pend} \leftarrow []$
(2) $status_P \in \{0,1\} \leftarrow 0$ for party P indicating whether P has sent a message in the current round

■ Upon receiving (**sid**, **WRITE**, M) from honest party P or (**sid**, **WRITE**, M, P) from S on behalf of corrupted party P:
If $status_P = 0$, then

(1) set $status_P \leftarrow 1$
(2) append M to L_{pend}
(3) if $|L_{pend}| = K$, then
    (a) order the messages lexicographically as $< M_1, ..., M_K >$
    (b) set $Lpend \leftarrow []$
    (c) set $status_P \leftarrow 0$ for every P
    (d) send (**sid**, **BROADCAST**, $< M_1, ..., M_K >$ to all parties and (**sid**, **BROADCAST**, $< M_1, ...M_K >, P$) to S
(4) else, send (**sid**, **WRITE**, $|M|$, P) to S

Figure 4: Anonymous broadcast ideal functionality. This functionality uses an initially empty list of pending messages L_{pend} and a variable $status_P$ which logs whether a particular client has sent a write request in a given round. When a write request is sent by a client who has not already sent a write request in the current round, that client is marked as having sent a write request in the current round and their message is added to the list of pending messages. If enough write requests have been received, the messages are ordered and published to all parties, and the identity of the last party to send a write request is sent to the simulator S. In this case, the pending message list and the status of all parties are reset to their initial values and a new round is begun. Otherwise, the write request, the length of the message, and the party that sent it is sent to the simulator S.

4.3 The Riposte Protocol

5 Proof

Case 1: We first examine the case where Server A is corrupted and $\mathcal{S}$ controls the honest Server B. Here $\mathcal{A}$ requests to corrupt Server A in the hybrid world and so S corrupts Server A in the ideal world. Upon receiving (**sid**, **WRITE**, $|M|$, P) from $\mathcal{F}_R^K$ where P is an honest party, $\mathcal{S}$ simulates a **WRITE** request with a dummy all-zero message $M_0 = 0...0$ on behalf of P. In this case, $\mathcal{S}$ simulates the protocol as written, using the dummy message M_0 to generate e_{ℓ,M_0}. LEAK DATA. Upon receiving (**sid**, **WRITE**, $|M^*|$, P^*) from $\mathcal{F}_R^K$ where P^* is a corrupted party, $\mathcal{S}$ reconstructs M^* from the two shares sent to Server A and Server B respectively across $\mathcal{F}_{AEC}(\{A, B\})$. This is made possible by the fact that parties are passively corrupted and so are forced to communicate using $\mathcal{F}_{AEC}$ exclusively, which $\mathcal{S}$ controls. Having reconstructed M^*, $\mathcal{S}$ then sends a WRITE request (**SID**, **WRITE**, M^*, P^*) to the functionality. Upon receiving (**sid**, **BROADCAST**, $< M_1, ..., M_K >$) from $\mathcal{F}_R^K$, $\mathcal{S}$ extracts the subset of honest messages from $< M_1, ..., M_K >$ and randomly assigns the honest messages to honest parties. For each honest party P_i with associated message M_i, $\mathcal{S}$ constructs e_{ℓ,M_i} and creates the share $e_{\ell,M_i} \oplus r$. $\mathcal{S}$ sends r to Server A and e_{ℓ,M_i} to Server B. At this stage, $\mathcal{S}$ must also simulate the WRITE request sent by the party P_K which is associated with M_K, the K-th message added to L_{pend}. This follows the steps outlined above, depending on whether P_K is honest or corrupted.
Case 2: Next, we examine the case where Server B is corrupted and $\mathcal{S}$ controls the honest Server A. In this case, $\mathcal{S}$ follows the same steps as in Case 1, however, $\mathcal{S}$ equivocates by sending any random bitsring r to Server B. After the assignment of honest messages to honest parties, $\mathcal{S}$ constructs a consistent $e_{\ell,M}$ for every honest message M and sneds $e_{\ell,M} \oplus r$ to Server B.
In both cases, whatever is sent by the corrupted server when the servers combine their states, the results will be statistically indistuinguishable from the result if both servers were honest.

6 Evaluation

6.1 Experimental Setup

6.2 Experimental Analysis

7 Conclusions

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Acknowledgments

I would like to thank …

Riposte UC Protocol

Variables:

- R - number of rows in each database table
- C - length of messages
- $e_{\ell,M}$ - R x C x 2 tensor containing 0 everywhere except in row l which contains $(M, M^2) \in \mathbb{F}^k$, where M is the message to be sent
- K - message limit in a round

Initialise:

(1) $status_P \in \{0,1\} \leftarrow 0$ for party P indicating whether P has sent a message in the current round
(2) $count \in \mathbb{N} \leftarrow 0$ indicating the number of valid write requests received this round

■ Upon receiving (**sid**, **WRITE**, M) from P
If $status_P = 0$, then

(1) set $status_P \leftarrow 1$
(2) P chooses index $l \xleftarrow{\$} [0, R)$ and generates bitstring $e_{\ell,M}$
(3) P generates random R x C x 2 tensor r
(4) P sends (**prove**, P, $e_{\ell,M}$) to $\mathcal{F}_{ZK}^{R,R'}$
(5) P sends $r \oplus e_{\ell,M}$ to Server B using $\mathcal{F}_{\mathcal{AEC}}(\{A, B\})$
(6) P sends r to Server A using $\mathcal{F}_{\mathcal{AEC}}(\{A, B\})$
(7) $count$ += 1
(8) if $count = K$, then
    (a) set $status_p \leftarrow 0$
    (b) set $count \leftarrow 0$

■ Upon receiving (**sid**, **SEND**, $r \oplus e_l$) from P, if P has not executed a write request in this phase, then Server B executes the following:
Upon receiving (**proof**, l(y)) from $\mathcal{F}_{ZK}^{R,R'}$, if received (**sid**, WRITE, M) from P, or upon receiving (**sid**, WRITE, M) from P, if received (**proof**, l(y)) from $\mathcal{F}_{ZK}^{R,R'}$:

(1) add $r \oplus e_{\ell,M}$ into its database
(2) if $count = K$, then
    (a) send the contents of its database to Server A
    (b) receive the contents of Server A's database
    (c) combine its database with Server A's database using row-by-row bitwise XOR
    (d) detect and resolve collisions as decsribed in 4.1.1
    (e) order messages lexicographically as $M_B =< M_1, ..., M_K >$
    (f) broadcast messages to all parties using $\mathcal{F}_{BC}$

■ Upon receiving (**sid**, **SEND**, r) from P, if P has not executed a write request in this phase, then Server A executes the following:
Upon receiving (**proof**, l(y)) from $\mathcal{F}_{ZK}^{R,R'}$, if received (**sid**, WRITE, M) from P, or upon receiving (**sid**, WRITE, M) from P, if received (**proof**, l(y)) from $\mathcal{F}_{ZK}^{R,R'}$:

(1) add r into its database
(2) if $count = K$, then
    (a) send the contents of its databse to Server B
    (b) receive the contents of Server A's database
    (c) combine database with Server B's database using row-by-row bitwise XOR
    (d) detect and resolve collisions as described in 4.1.1
    (e) order messages lexicographically as $M_A =< M_1, ..., M_K >$
    (f) broadcast messages to all parties using $\mathcal{F}_{BC}$

Figure 5: Anonymous broadcast protocol.

References

[1] Myrto Arapinis, Nikolaos Lamprou, Lenka Mareková, Thomas Zacharias, Léo Ackermann, and Pavlos Georgiou. 2020. E-cclesia: Universally Composable Self-Tallying Elections. Cryptology ePrint Archive, Paper 2020/513. https://eprint.iacr.org/2020/513

[2] Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. 1991. Noninteractive Zero-Knowledge. *SIAM J. Comput.* 20, 6 (1991), 1084–1118. doi:10.1137/0220068

[3] R. Canetti. 2001. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. 136–145. doi:10.1109/SFCS.2001.959888

[4] David Chaum. 1988. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of cryptology* 1 (1988), 65–75.

[5] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. 2015. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 321–338.

[6] Henry Corrigan-Gibbs and Bryan Ford. 2010. Dissent: accountable anonymous group messaging. In *Proceedings of the 17th ACM conference on Computer and communications security*. 340–350.

[7] Sharad Goel, Mark Robson, Milo Polte, and Emin Sirer. 2003. *Herbivore: A scalable and efficient protocol for anonymous communication.* Technical Report. Cornell University.

[8] M.G. Reed, P.F. Syverson, and D.M. Goldschlag. 1998. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications* 16, 4 (1998), 482–494. doi:10.1109/49.668972

[9] Pance Ribarski and Ljupcho Antovski. 2012. Mixnets: Implementation and performance evaluation of decryption and re-encryption types. In *Proceedings of the ITI 2012 34th International Conference on Information Technology Interfaces*. 493–498. doi:10.2498/iti.2012.0432

[10] Filip Vyrostko. 2024. *Creating Provably Secure Anonymous Broadcast Channels.* Master's thesis. University of Glasgow.

[11] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. 2012. Dissent in numbers: Making strong anonymity scale. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 179–182.