Universally Composable Anonymous Broadcast Protocols

Kyla Nel
2576953n

Abstract

1 Introduction

2 Proof

3 Background

Perhaps you want to cite the seminal paper of Turing [3], or prior [2] and concurrent [1] work.

4 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. 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:

$$d_A = r_1 + \ldots + r_n$$

and Server B's database state will be:

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

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

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

which is clearly the same as

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

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. However, this does create 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 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

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

and

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

Observing that

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

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 collision for $k > 2$. Each party P_i wanting to write their message M_i into the database instead writes $(M_i, M_i^2, \ldots, M_i^k) \in \mathbb{F}^k$ 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.

4.2 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. Zero knowledge techniques are applied to allow parties to prove that their write requests are well formed.

5 Functionality & Protocol

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 1: Anonymous broadcast ideal functionality.

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 3: Anonymous broadcast ideal functionality.

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

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 4: Non-interactive zero knowledge functionality.

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

6 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

Riposte UC Protocol

Variables:

- R - number of rows in each database table
- C - length of messages
- $e_{\ell,M}$ - R x C x 2 bitstring 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_l
3. P generates random R x C x 2 bitstring 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:

1. add $r \oplus e_{\ell,M}$ into its database
2. if $count = K$, then
3. (a) add $r \oplus e_l$ into its database
   (b) if $count = K$, then
      (i) combine database with Server A's database
      (ii) resolve collisions
      (iii) order messages lexicographically as $M_B =< M_1, ..., M_K >$
      (iv) broadcast messages to all parties

Upon receiving (**sid**, WRITE, M) from P, if received (**proof**, l(y)) from $\mathcal{F}_{ZK}^{R,R'}$:

1. add $r \oplus e_l$ into its database
2. if $count = K$, then
   (a) combine database with Server A's database
   (b) check for collisions
   (c) resolve collisions
   (d) order messages lexicographically as $M_B =< M_1, ..., M_K >$
   (e) broadcast messages to all parties

■ 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:

1. add r into its database
2. if $count = K$, then
   (a) combine database with Server B's database
   (b) resolve collisions
   (c) order messages lexicographically as $M_A =< M_1, ..., M_K >$
   (d) broadcast messages to all parties

Upon receiving (**sid**, WRITE, M) from P, if received (**proof**, l(y)) from $\mathcal{F}_{ZK}^{R,R'}$:

1. XOR r into its database
2. if $count = K$, then
   (a) combine database with Server B's database
   (b) resolve collisions
   (c) order messages lexicographically as $M_A =< M_1, ..., M_K >$
   (d) broadcast messages to all parties

Figure 2: Anonymous broadcast protocol.

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.

7 Evaluation

7.1 Experimental Setup

7.2 Experimental Analysis

8 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 . . .

References

[1] Alonzo Church. 1936. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics* 58, 2 (1936), 345–363. http://www.jstor.org/stable/2371045

[2] Kurt Gödel. 1931. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik* 38–38, 1 (Dec. 1931), 173—-198. doi:10.1007/bf01700692

[3] Alan M. Turing. 1937. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* s2-42, 1 (1937), 230–265. doi:10.1112/plms/s2-42.1.230