Abstract

We review the two-round statically-secure MPC protocol of [3], and intuitively motivate its construction from smaller building blocks. Then we consider the problems in making this protocol adaptively-secure, and present a proposed solution that slightly modifies the protocol to overcome these problems. In the process, we introduce the notion of “Deniable Obfuscation”, as an extension of Deniable Encryption. We give a construction of Deniable Obfuscation as a modification of the Deniable Encryption scheme in [5], which can also be seen as a corollary of a compiler in [2].

Contents

1 Preliminaries 2
  1.1 Multi-Party Computation .............................................. 2
  1.2 Round-Complexity ...................................................... 2

2 Achieving Two-Rounds: MPC in the Head 2
  2.1 Semi-Honest .............................................................. 3
  2.2 Collapsing Rounds ...................................................... 3
  2.3 Full Construction ....................................................... 3
  2.4 Simulator and Security ................................................ 4
  2.5 Extension: Low-Communication ...................................... 4

3 Achieving Adaptive Security 5
  3.1 Equivocal and Extractable Commitments ............................. 5
  3.2 Deniable Obfuscation .................................................... 5
  3.3 Proposed Construction ................................................ 6
  3.4 Simulator and Proof of Security ...................................... 6

4 References 8

*Completed as a final project for CS 276 at University of California, Berkeley, under Prof. Sanjam Garg.
1 Preliminaries

1.1 Multi-Party Computation

The goal of multi-party computation (MPC) is for a group of parties (who do not trust each other) to compute some function $f$ of their private inputs. (For example, in a voting system, $f$ could be the majority function).

We would like MPC protocols to be secure, in the sense that parties do not reveal anything more about their private inputs (eg, their votes) than could be learnt from the function output (eg, the election winner).

This is formalized by the Real/Ideal World simulation model. We will use the MPC model of broadcast channels, where in each round parties can publicly broadcast messages to everyone else. In the Real World, an adversary $A$ watches the execution of a MPC protocol $\Pi$, seeing all public messages, as well as the private state of any parties he has corrupted (for example, a group of voters could be colluding, and share their entire states). In the Ideal World, all parties have access a trusted third-party: the ideal functionality $F$ (the “function in-the-sky”). Each party privately hands its input to $F$ (unseen by other parties), and $F$ computes $f(\cdot)$ of the private inputs, and hands the result back to the parties. The protocol $\Pi$ is statically-secure if for all real-world adversaries $A$, there exists an ideal-world simulator $S_A$, which can simulate (for all private inputs) a real-world execution of $\Pi$ (with $A$) using only the ideal functionality $F$. That is, the real-world execution and ideal-world simulation are computationally indistinguishable. The ideal-world simulator $S$ has access to the ideal functionality, and to the internal states of any parties $A$ decides to corrupt. In practice, $S$ is often constructed by running $A$ “in its belly”, and feeding it simulated messages. If $S$ ensures that the view of the adversary $A$ in the real-world vs. in the “belly of $S$” is indistinguishable, then $\Pi$ is statically secure – after all, the adversary can’t tell if it’s in the real-world or in the simulated ideal world. Note that $A$’s view includes his private coins, the CRS, corrupted party state and coins, and public messages. The protocol $\Pi$ is adaptively-secure if $A$ is allowed to adaptively choose which parties to corrupt in the middle of the protocol (eg, he could wait for the first round of messages, then decide to corrupt someone). Naturally, the ideal-world simulator $S$ is given access to the state (and identity) of corrupted parties only when $A$ decides to corrupt them.

1.2 Round-Complexity

Secure MPC cannot be achieved in fewer than two rounds: If there was a one-round MPC protocol, then the final output would be a function $g(m_1, m_2, \ldots)$ of the first-round messages $\{m_i\}$ broadcast by each party $i$. But then Party 1 would have access to $m_2, m_3, \ldots$, and could evaluate $g(x, m_2, m_3, \ldots)$ for any $x$ of his choosing (the “residual function” attack). So secure MPC requires at least two rounds (with the second round dependent on the messages of the first round, to prevent the above attack).

2 Achieving Two-Rounds: MPC in the Head

The protocol introduced by [3] uses indistinguishably obfuscation to achieve two-round statically secure MPC for general functions (in the common-reference-string model). In fact, they extend their protocol to make communication independent of the complexity of the function being computed (depending only on the input/output size). We describe their scheme here.

Their construction is actually a compiler, which converts any arbitrary MPC protocol $\Pi$ into an equivalent two-round protocol. The key idea is, each party broadcasts obfuscations of their next-message function $\pi$ in the original MPC protocol (which their private input hardcoded). Then all parties can run the original MPC, except instead of actually communicating with each other, they just simulate other parties “in their heads” (using the provided obfuscations).
The protocol in [3] compiles a arbitrary-round, statically-secure, semi-honest MPC protocol $\pi$ into a two-round, statically-secure, maliciously-secure protocol $\Pi$.

2.1 Semi-Honesty

To understand the full protocol, we will first consider how to achieve malicious-security from semi-honest security. This is in fact a standard transformation (as given in [4], and explained in [?]), which works by requiring proofs that parties are acting honestly. We need to ensure that parties use uniformly random private coins, and that they honestly compute messages based on their private inputs, and coins:

- **Input Commitment phase.** Parties commit to their private inputs.
- **Coin Generation phase.** Parties execute secure coin-flipping, to generate private coins that are uniform (if at least one honest party exists). Unlike public coin-flipping, here parties only know commitments of others’ private coins (so the coins are hidden, but everyone is bound to their coins).
- **Protocol Emulation phase.** Parties execute the MPC protocol $\pi$, attaching to each message NIZK proofs that: there exist private-input and private-coins consistent with both their input/coin commitments, and their computed message.

2.2 Collapsing Rounds

Now we will assume semi-honesty (for now) and just try to collapse multi-round $\to$ two-round. Even with VBB obfuscation, we cannot simply broadcast all the obfuscations in the first round, since then malicious parties could evaluate the “residual function” (with honest parties inputs fixed) on any non-honest inputs of their choice.

That is, we need to ensure that parties are “honest and not-too-curious”, and don’t try evaluating the obfuscations on other inputs. We fix this in the same way we enforced semi-honesty: We make parties commit to their inputs in the first round, then ensure that obfuscations sent in the second round can only be evaluated on inputs consistent with the commitments.

Roughly:

- **Round 1.** Parties commit to their inputs, and to the randomness that they will use for the underlying MPC protocol.
- **Round 2.** Parties broadcast obfuscations of their “next-message” functions, with their inputs and randomness hardcoded. These obfuscations require NIZK proofs that their inputs are consistent with the commitments of Round 1, and the transcript so-far. If so, they output the next-message, along with a NIZK that it was computed honestly (to be used in the next round).

Notice that by construction, the messages in Round 1 fix the result of the rest of the protocol (Round 2, and the transcript computed by obfuscations) – since they only accept valid inputs. Further, we may expect this to naturally extend to achieving both static-security and round-collapse simultaneously, since they both involve phases of input-commitment and NIZK-proofs-of-honesty. This will be true, and essentially this same construction will work with $iO$ instead of VBB.

2.3 Full Construction

The full construction from [3] is included verbatim in figures at the end. It proceeds roughly as:

**Round 1.** Parties commit to their private-inputs and some randomness, using CCA-secure encryptions. Party $P_i$ commits to randomness $r_{i,j}$ for all parties $j$ (including himself). Later, $P_i$ will use $\bigoplus_k r_{k,i}$ as random
coins for the underlying protocol $\pi$. So each party influences the random coins used by others (and hence the coins will be fair with at least one honest party).

**Round 2.**

- $P_i$ “opens” the commitments to the randomness $r_{i,j}$ for $j \neq i$ (keeping his own $r_{i,i}$ unopened). Actually, instead of fully opening the commitments, $P_i$ reveals $r_{i,j}$, then gives a NIZK $\gamma_{i,j}$ that this was the value committed (without revealing the actual randomness used in the commitment).

- Let $\pi$ be a $t$-round protocol. Each player $P_i$ outputs $t$ obfuscations $iO_1, \ldots, iO_{t,i}$ of its next-message function for each round (with input $x_i$ and randomness $r_{i,i}$ hardcoded). These obfuscations take in the $r_{i,j}$ values, proofs $\gamma_{i,j}$ they are consistent with commitments, the current transcript-so-far $M$, and proofs $\Phi$ the transcripts were generated correctly (consistent with committed private inputs, randomness, and previous-transcripts). They output the next-message $\pi_i(\cdot)$, as well as a NIZK that it was generated correctly.

**Evaluation.** All parties independently evaluate the protocol $\pi$ “in their heads”, using the provided obfuscations.

Perhaps the only surprising thing is that in Round 2, parties provide NIZK proofs that their commitments are valid, without fully opening the randomness. This will be necessary in the proof of static-security, so the simulator can lie about the commitments (as we will see).

### 2.4 Simulator and Security

We briefly describe the simulator; the full treatment is in Appendix B, C of [3].

**Simulator.**

- **Setup.** The simulator generates the CRS $\sigma$ according to the NIZK simulator (keeping the trapdoor that will allow it to generate simulated proofs). It also generates the $(pk, sk)$ pair of a CCA-secure public-key encryption system, putting $pk$ in the reference-string. It will later use $sk$ to decrypt commitments by the adversary.

- **Round 1.** Simulate honest party input/randomness commitments with just encryptions of the zero-string. Also decrypt the adversary’s commitments using $sk$.

- **Round 2.** Pass the extracted inputs of the adversary to the ideal functionality, receiving the output $f(\cdot)$. Use the simulator $S_\pi$ of the underlying semi-honest MPC protocol $\pi$ to generate a simulated transcript consistent with this output. Note, this transcript specifies all public messages, as well as the private-coins of corrupted parties (call these $s_j$). The simulator then generates obfuscations of honest-party next-message-functions by simply hardcoding the simulated messages from the transcript (that is, the obfuscations have a fixed-output hardcoded). It forces the adversary to use coins $s_j$ for corrupted party $P_j$ by sampling $r_{i,j}$ accordingly. (The simulator committed to zero-strings in Round 1, but it can use the NIZK trapdoor to lie about the randomness here).

### 2.5 Extension: Low-Communication

The authors also noted that their construction can be used to achieve two-round MPC with communication-complexity independent of the function being computed. Roughly, this uses Multi-Key Fully-Homomorphic-Encryption: the players broadcast their encrypted inputs, locally compute the encrypted output, then execute an MPC protocol for the “joint-decoding” function.
3 Achieving Adaptive Security

We will try to extend the construction of [3] to achieve adaptive security. Now the adversary can decide to corrupt a party at some intermediate point in the protocol. An ideal-world simulator must be able to prepare the state of newly-corrupted parties, such that it appears indistinguishable from the real world.

The Problem: If we try to use the statically-secure construction directly, we run into two immediate problems in simulation. Recall, in Round 1 our simulator committed to the zero-string on behalf of honest parties. If the adversary later corrupts an honest party (after Round 1), we will be unable to produce randomness to convince him we committed to our actual input, and not zeros (since commitments are binding). We have this same problem with the \( iO \) obfuscations output in Round 2. Although the \( \text{fixedOutput} \) obfuscations are indistinguishable from actual next-message functions if we don’t know the randomness used in obfuscation, we still cannot produce randomness to convince \( A \) that an fixed-output obfuscation is actually an honest obfuscation.

3.1 Equivocal and Extractable Commitments

An Equivocal Commitment scheme is essentially a commitment scheme where the CRS can be set in two modes: binding mode, and non-binding mode. When the CRS is in binding mode, commitments are perfectly binding. But in non-binding mode, commitments are not binding at all – they can be opened ("equivocated") to anything (if we know some trapdoor information). Further, the binding and non-binding setups are indistinguishable. This will allow the simulator to lie about commitments. An Equivocal and Extractable Commitment scheme adds the additional ability for extracting the plaintext of commitments, when the CRS is non-binding (if we know some trapdoor). This is a standard existing construction, for example from [1].

3.2 Deniable Obfuscation

We will need a "deniable" variant of indistinguishably obfuscation, which we construct based off the deniable encryption scheme of [5]. Normal encryption schemes have the "coercability" weakness, in that an external party may coerce the sender into revealing his message (by demanding he reveal the randomness used in encrypting). Deniable Encryption schemes fix this problem, by allowing an arbitrary cyphertext to be “explained” as an encryption of any arbitrary message. That is, the sender (or in fact, anyone) can always produce randomness such that an arbitrary message is encrypted to his cyphertext. Intuitively, this works by augmenting a normal PKE scheme, such that for most randomness it encrypts normally, but for some sparse set of randomness, it activates a "backdoor" that can encrypt to anything. We will need a similar construction, but for obfuscations, such that we can explain an obfuscation of one program as an obfuscation of another (functionally-equivalent) program. In fact, we can simply use the construction of [5], except output obfuscations instead of encryptions.

For ease of proof, we will use the result of [2], which notices that arbitrary programs can be compiled into “explainable” versions as follows: Any algorithm \( \text{Alg} \) can be compiled into PPT algorithms \( \text{DenAlg}, \text{Explain} \) such that:

- The functionality of \( \text{Alg} \) is preserved in \( \text{DenAlg} \) (with high probability, over the random coins used by \( \text{DenAlg} \)).
- For any valid \((\text{input}, \text{output})\) pair, we can run \( r^* \leftarrow \text{Explain}(\text{input}, \text{output}) \) to get randomness such that \( \text{output} = \text{DenAlg}(\text{input}; r^*) \).

More formally, the second condition says that the following two distributions are computationally indistinguishable, for all \( \text{input} \) (and \( \text{DenAlg}, \text{Explain} \) public):
\[ r \xleftarrow{\$} \quad \text{output} \leftarrow \text{DenAlg(input; } r \text{)} \quad \{\text{output, } r\} \]

Now, let us take \( \text{Alg} = iO \), and consider two functionally-equivalent programs \( C_1, C_2 \). For \( \text{input} = C_1 \), the distribution on the right becomes:

\[ r \xleftarrow{\$} \quad \text{output} \leftarrow \text{DeniO}(C_1; r) \quad r^* \leftarrow \text{Explain(input, output)} \quad \{\text{output, } r^*\} \]

Then by the security of \( iO \) (and since DeniO behaves like \( iO \) with overwhelming probability), this is indistinguishable from \( \text{output} \leftarrow \text{DeniO}(C_2; r) \). So this construction of “deniable obfuscation” lets us claim that an obfuscation of \( C_1 \) is actually an obfuscation of \( C_2 \).

### 3.3 Proposed Construction

Here we aim to compile an arbitrary-round, adaptively-secure, semi-honest protocol \( \pi \) into a two-round, adaptively-secure, maliciously-secure protocol \( \Pi \). Note, we now require the underlying protocol to be adaptively-secure. Our protocol \( \Pi \) is exactly the same as the statically-secure construction from [3], with two changes:

- In Round 1, use Equivocal and Extractable Commitments to commit to inputs/randomness. Then in Round 2, we can simply send openings instead of NIZK proofs of commitments.
- In Round 2, send “deniable obfuscations” (as constructed above) instead of normal indistinguishably obfuscations.

Note, it seems tempting to use Deniable Encryptions for commitments in Round 1, but this fails because Deniable Encryptions are never perfectly binding (which we need, so the transcripts are fixed). Equivocal commitments lets us achieve perfect binding in the real-world, but non-binding in the simulation.

### 3.4 Simulator and Proof of Security

**Simulator.** This is similar to the statically-secure simulator, with changes for adaptive corruptions (in particular, with how the underlying simulator is used). \( S \) is the simulator for our construction, and let \( S_\pi \) be the simulator for the underlying semi-honest MPC protocol \( \pi \).

- **Setup.** The simulator generates the CRS \( (\sigma, \sigma') \) by running the NIZK simulator, and the setup for the equivocal commitment scheme (in non-binding mode). Keep the trapdoor information for both, allowing the simulator to create fake proofs, equivocate & extract commitments.

- **Round 1.** Simulate honest party input/randomness commitments with just commitments of the zero-string. Also extract the adversary’s commitments using the trapdoor.

- **Corruptions during/after Round 1.** Equivocate on the commitments to produce randomness consistent with committing the true input.

- **Round 2.**
  
  Pass the extracted inputs of the parties currently adversary-controlled to the ideal functionality, receiving the output \( f(\cdot) \). Use the simulator \( S_\pi \) of the underlying semi-honest MPC protocol \( \pi \) to generate a simulated transcript consistent with this output. Note, this transcript specifies all public messages, as
well as the private-coins of currently-corrupted parties (call these coins $s_j$). The simulator forces the adversary to use coins $s_j$ for currently-corrupted party $P_j$ by sampling $r_{i,j}$ accordingly, then “opening” the zero-commitments to $r_{i,j}$ (using the equivocation trapdoor). It then generates obfuscations of honest-party next-message-functions by simply hardcoding the simulated messages $m_{i,j}$ from the transcript (that is, the obfuscations have a fixed-output hardcoded).

- **Corruptions during/after Round 2.** Deny on the obfuscations, to produce randomness that is consistent with honest obfuscations (ie, not of fixed-output). Then, for corruptions that occur after all the $\{r_{i,j}\}$ broadcasts, we ask the underlying simulator $S_\pi$ for random coins $s_j$ consistent with a corruption of part $P_j$ at this stage (eg, we ask it for coins corresponding to a new corruption at the end of protocol $\pi$, consistent with its previous simulated messages $m_{i,j}$). The simulator $S$ gives $S_\pi$ the corrupted party’s input at the time of corruption, as well as access to the ideal functionality. Then, the simulator $S$ sets $r_{i,j}$ such that $\bigoplus_i r_{i,j} = s_j$, and equivocates on the Round-1 commitment to $r_{j,j}$.

**Proof.** By hybrid argument (following the original proof for the static case in [3]).

- $H_1$: Honest strategy. The simulator has access to all the private inputs $\{x_i\}$, and uses them to execute the protocol honestly. The CRS is generated honestly, as in the real world (with commitments in binding-mode). And the randomness $r_{i,j}$ of honest-parties is sampled uniformly, in the setup phase.

- $H_2$: Use the trapdoor from the commitment setup to extract the inputs/randomness of corrupted parties in Round 1.

- $H_3$: In Round 2, generate deniable obfuscations on behalf of honest parties by using the known or extracted inputs/randomness of all parties to pre-compute the outputs of the obfuscations. Since the commitments are perfectly binding, and the obfuscations check NIZKs at every stage, the Round 1 commitments fix the transcript of the obfuscations: they will output messages $m_{i,j}$ and proofs $\phi_{i,j}$ on the unique input consistent with the commitments, or output $\bot$ otherwise. Therefore, they are indistinguishable from obfuscations of fixedOutput with $(m_{i,j}, \phi_{i,j})$ hardcoded, and indistinguishably follows from $iO$. Now parties’ private inputs/randomness are no longer directly encoded in the obfuscations.

For corruptions in Round 2 (after obfuscations have been sent), deny on the obfuscations to explain them as honest obfuscations (not fixedOutput).

- $H_4$: Generate the NIZK CRS and the hardcoded NIZK proofs $\phi_{i,j}$ (used in the obfuscations) using the NIZK simulator.

- $H_5$: In Round 1, the simulator commits to the zero-string on behalf of the honest parties, instead of committing to their inputs/randomness. Then in Round 2, the simulator equivocates to open commitments as the true randomness $r_{i,j}$. Corruptions at any point after Round 1 commitments are handled by the same equivocation. This is indistinguishable by the polynomial equivocability of the commitment scheme (ie, even with access to polynomially many equivocal commitments and their openings, commitments are still computationally hiding).

Note: At this point, the simulator only uses the private inputs $\{x_i\}$ of honest parties to compute fixed outputs of the obfuscations. However, the simulator still requires $\{x_i\}$, since the honest-party messages in $\pi$ will depend on the randomness used by the corrupted parties. We will fix this in the next hybrid.

- $H_6$: At the point when party $j$ is corrupted, sample coins $s_j$ randomly. Then in Round 2, sample $r_{i,j}$ for all remaining honest parties randomly, subject to $\bigoplus_i r_{i,j} = s_j$ for all corrupted parties $P_j$. Here, we are considering parties corrupted at the time (ie, before all honest Round 2 openings are sent). As long as at least one uncorrupted party remains at this time, this step is possible (if not, the simulator has nothing else to do anyway). This results in an identical distribution as $H_5$.

- $H_7$: For new corruptions of party $j$ after all honest Round 2 messages, set $r_{j,j}$ such that $\bigoplus_i r_{i,j} = s_j$. As usual, continue to equivocate the Round 1 commitment (in which the simulator committed to zeros) as arising from $r_{j,j}$. Notice that in previous hybrids, we could have equivalently sampled $r_{j,j}$ when the
corruption occurs, instead of in the setup phase. So $H_7$ simply changes the sampling order, resulting in an identical distribution.

Note: Now the randomness used by eventually-corrupted parties (in executing the underlying protocol $\pi$) is effectively controlled by the simulator. Further, the simulator can sample coins $s_j$ for corrupted parties to use at the time of the corruption.

- $H_8$: Now our simulator is sufficiently powerful to invoke the simulator $S_\pi$ of the underlying adaptively-secure semi-honest protocol $\pi$: First it uses $S_\pi$ (with oracle access to the ideal functionality) to generate a fake transcript $m_{i,j}$, and coins $s_j$ consistent with the transcript for all statically-corrupted parties. Then, as parties are adaptively corrupted it calls on $S_\pi$ to adaptively generate private coins consistent with the transcript, which it forces onto the adversary as usual.

4 References


\footnote{There are some details/justification missing here (TODO). For example, the simulator of the underlying protocol may depend on the adversary for the underlying protocol. We use a semi-honest adversary in this case. Also, the reduction to using the simulator needs to be done carefully, using the fact that it will still be hard to produce valid NIZK proofs. Note, the simulator of [3] performs similar steps.}
Protocol Π

Protocol Π uses an Indistinguishability Obfuscator $\mathcal{O}$, a NIZK proof system $(K,P,V)$, a CCA-secure PKE scheme $(\text{Gen, Enc, Dec})$ with perfect correctness and an $n$-party semi-honest MPC protocol $\pi$.

**Private Inputs:** Party $P_i$ for $i \in [n]$, receives its input $x_i$.

**Common Reference String:** Let $\sigma \leftarrow K(\lambda)$ and $(pk, \cdot) \leftarrow \text{Gen}(\lambda)$ and then output $(\sigma, pk)$ as the common reference string.

**Round 1:** Each party $P_i$ proceeds as:
- $c_i = \text{Enc}(i\|x_i)$ and,
- $\forall j \in [n]$, sample randomness $r_{i,j} \in \{0,1\}^\ell$ and generate $d_{i,j} = \text{Enc}(i\|r_{i,j})$. (Here $\ell$ is the length of the maximum number of random coins needed by any party in $\pi$.)

It then sends $Z_i = \{c_i, \{d_{i,j}\}_{j \in [n]}\}$ to every other party.

**Round 2:** $P_i$ generates:
- For every $j \in [n], \ j \neq i$ generate $\gamma_{i,j}$ as the NIZK proof under $\sigma$ for the NP-statement:
  $$\{\exists \rho_{r_{i,j}} \mid d_{i,j} = \text{Enc}(i\|r_{i,j}; \rho_{r_{i,j}})\}.$$  

- A sequence of obfuscations $(i\mathcal{O}_{i,1}, \ldots, i\mathcal{O}_{i,t})$ where $i\mathcal{O}_{i,j}$ is the obfuscation of the program $\text{Prog}_{i,j}$.

- It sends $(\{r_{i,j}, \gamma_{i,j}\}_{j \in [n], j \neq i}, \{i\mathcal{O}_{i,j}\}_{j \in [t]})$ to every other party.

**Evaluation (MPC in the Head):** For each $j \in [t]$ proceed as follows:
- For each $i \in [n]$, evaluate the obfuscation $i\mathcal{O}_{i,j}$ of program $\text{Prog}_{i,j}$ on input $(R, \Gamma, M_{j-1}, \Phi_{j-1})$ where
  $$R = \begin{pmatrix} r_{1,1} & \cdots & r_{1,n} \\ \vdots & \ddots & \vdots \\ r_{n,1} & \cdots & r_{n,n} \end{pmatrix}, \quad \Gamma = \begin{pmatrix} \gamma_{1,1} & \cdots & \gamma_{1,n} \\ \vdots & \ddots & \vdots \\ \gamma_{n,1} & \cdots & \gamma_{n,n} \end{pmatrix}$$
  $$M_{j-1} = \begin{pmatrix} m_{1,1} & m_{1,2} & \cdots & m_{1,n} \\ m_{1,2} & m_{2,2} & \cdots & m_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{1,n-1} & m_{2,n-1} & \cdots & m_{n,n-1} \end{pmatrix}, \quad \Phi = \begin{pmatrix} \phi_{1,1} & \phi_{2,1} & \cdots & \phi_{n,1} \\ \phi_{1,2} & \phi_{2,2} & \cdots & \phi_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{1,n-1} & \phi_{2,n-1} & \cdots & \phi_{n,n-1} \end{pmatrix}$$
- And obtain, $m_{1,j}, \ldots, m_{n,j}$ and $\phi_{1,j}, \ldots, \phi_{n,j}$.

Finally each party $P_i$ outputs $m_{i,t}$.

Figure 1: Two-Round MPC protocol, from [3]
Program $\text{Prog}_{i,j}$ takes as input $(R, \Gamma, M_{i-1}, \Phi_{j-1})$ where $\Gamma$ and $\Phi$ are defined. Specifically, it proceeds as follows:

- $\forall p, q \in [n]$ such that $p \neq q$ check that $\gamma_{p,q}$ is an accepting proof under $\sigma$ for the NP-statement:
  \[
  \{ \exists r_{p,q} \mid d_{p,q} = \text{Enc}(p|r_{p,q}, \rho_{r_{p,q}}) \}.
  \]

- $\forall p \in [n], q \in [j-1]$ check that $\phi_{p,q}$ is an accepting proof for the NP-statement
  \[
  \{ \exists (x_p, r_{p,p}, \rho_{x_p}, \rho_{r_{p,p}}) \mid (c_p = \text{Enc}(p|x_p; \rho_{x_p}) \land d_{p,p} = \text{Enc}(p|r_{p,p}, \rho_{r_{p,p}}) \land m_{p,q} = \pi_p(x_p, \oplus_{k \in [n]} r_{k,p}, M_{q-1}) \}.
  \]

- If the checks above fail, output $\bot$. Otherwise, if $\text{flag} = 0$ then output $(\pi_i(x_i, \oplus_{j \in [n]} r_{j,i}, M_{j-1}), \phi_{i,j})$ where $\phi_{i,j}$ is the proof for the NP-statement: (under some fixed randomness)
  \[
  \{ \exists (x_i, r_{i,i}, \rho_{x_i}, \rho_{r_{i,i}}) \mid (c_i = \text{Enc}(i|x_i; \rho_{x_i}) \land d_{i,i} = \text{Enc}(i|r_{i,i}, \rho_{r_{i,i}}) \land m_{i,j} = \pi_i(x_i, \oplus_{j \in [n]} r_{j,i}, M_{j-1}) \}.
  \]

Otherwise, output $\text{fixedOutput}$.

Figure 2: Obfuscated Programs in the Protocol, from [3]