CS4501 Cryptographic Protocols Lecture 26: Schnorr Protocols, Coin Tossing Part 2, the GMW Compiler

https://jackdoerner.net/teaching/#2026/Spring/CS4501

Recall: The Zero-Knowledge Functionality

- At a high level, we want a functionality that allows the prover to input some statement, and the verifier receives and indication that the statement is a member of some agreed language L.
- For any NP-Language L, let R be the associated polynomial-time membership-verifier algorithm. That is, $\forall x \in L \;\; \exists w \in \{0,1\}^*$ such that $R_L(x, w) = 1$ and $|w| \in O(\text{poly}(|x|))$, and $\forall x \notin L \;\; \forall w \in \{0,1\}^*$ we have $R_L(x, w) = 0$.
- The prover inputs a statement x and witness w. The verifier learns the statement, and it learns whether or not w witnesses $x \in L$.

Recall: Sigma Protocols

Definition 1: A Sigma Protocol is an IPS with the following 3-message structure:

1. The prover sends a statement x and a commitment message a to the verifier.
2. The verifier samples a challenge e *uniformly* from some set E, and sends it to he prover.
3. The prover sends a response z to the challenge.

Definition 2: A sigma protocol is said to be n-special-sound if there exists an additional PPT algorithm $\mathsf{Reconstruct}$ such that takes as input $x, a, \{e_i, z_i\}_{i\in[n]}$, and achieves the following property:

If V outputs 1 given the transcript (x, a, e_i, z_i) for every $i \in [n]$, and $e_i \neq e_j$ for every $i, j \in [n]$, then $\mathsf{Reconstruct}(x, a, \{e_i, z_i\}_{i\in[n]})$ outputs $\hat{w}$ such that $R(x, \hat{w}) = 1$. Otherwise, $\mathsf{Reconstruct}$ outputs $\perp$.

That is, $\mathsf{Reconstruct}$ outputs a witness given n different accepting proofs with the same first message.

Definition 3: A sigma protocol is *Special Honest Verifier Zero-Knowledge* (SHVZK) if for every semi-honest PPT V^*, there exists a PPT Sim that simulates given a *specific* challenge e in advance. That is, $\{\mathsf{VIEW}_{V^*}\}_{x\in L} \approx_c \{\mathsf{Sim}(x, e) : e \leftarrow E_x\}_{x\in L}$, where E_x is the challenge set associated with x.

What is a protocol compiler?

- A *compiler,* in general, takes a computer program that was designed to run in one environment, and translates it so that it runs in another environment.
- Usually the input is code that runs on some high-level machine model (there may not be any physical machines that actually implement this model, but nevertheless there is a specification for how such machines would behave, hypothetically), and the output is code that runs on a much lower-level machine model, for which we have physical hardware.
- We want compilers to *preserve semantics* through the transformation. That is, the behavior of the program should remain the same, even though the environment in which it runs is different.
- Protocol compilers transform entire protocols (which are just programs that coordinate multiple participants). We similarly expect them to preserve semantics in some sense.
- Usually, rather than the machine model changing, we think of the network setting or adversary changing. The main result of the second half of this class is the *GMW compiler*, which takes protocols with security against semi-honest adversaries, and outputs protocols with security against malicious adversaries.
- We can envision other protocol compilers. Imagine adding starting with a protocol in our model and adding the ability to handle asynchrony, for example.

Recall: A Protocol Compiler for Composable ZK

Theorem 1: Let ℓ be a polynomial and let L be a language with membership-verifier algorithm R. If there exists an $\ell(|x|)$-special-sound SHVZK sigma protocol for L, and there exists an *extractable* commitment scheme, then there exists a protocol in the $\mathcal{F}_{\mathsf{CRS}}$-hybrid model that realizes $\mathcal{F}_{\mathsf{ZK}}^{R}$.

We'll review the protocol compiler, but not the proof. First, some notation. Let

- $(a, s) \leftarrow \mathsf{Prove}_1(x, w)$ and $z \leftarrow \mathsf{Prove}_3(s, x, w, a, e)$ be randomized algorithms that sample the prover's messages in the sigma protocol, and $\mathsf{Verify}(x, a, e, z)$ be the verifier's output algorithm.
- E_x be the challenge set for statement x, of size exactly $\ell(|x|)$. If the sigma protocol has a larger challenge set, then an arbitrary subset of size $\ell(|x|)$ can be chosen. This challenge set is computed by an algorithm $E_x \leftarrow \mathsf{Challenge}(x)$, and it must be independent of a.
- Sim be the SHVZK simulator.
- $(\mathsf{Gen}, \mathsf{Com}, \mathsf{Open}, \mathsf{TGen}, \mathsf{Ext})$ be the extractable commitment scheme.
- $\rho = \ell(|x|) \cdot \kappa$ be a *repetition count*. This is the number of times we need to repeat the proof to achieve negligible soundness error.

Now we are ready to see the compiler. We will refer to the compiled protocol as $\pi_{\mathsf{CompZK}}^{\Sigma}$, where $\Sigma = (\mathsf{Prove}_1, \mathsf{Challenge}, \mathsf{Prove}_3, \mathsf{Verify})$ is the sigma protocol we started with.

Recall: A Protocol Compiler for Composable ZK

receives inputs $1^\kappa, x, w$

if $R(x, w) = 0$, halt immediately without sending messages. Otherwise…

1^κ

k

1^κ

k

receives input 1^κ

x

$\forall i \in [\rho],\ e_i \leftarrow E_x$

$\forall i \in [\rho],\ (c_i^e, d_i^e) \leftarrow \text{Com}_k(e_i)$

$c_1^e, \ldots, c_\rho^e$

$\forall i \in [\rho],\ a_i \leftarrow \text{Prove}_1(x, w)$

$\forall i \in [\rho],\ \forall \hat{e} \in E_x,\ z_{i,\hat{e}} \leftarrow \text{Prove}_3(x, w, a_i, \hat{e})$

$\forall i \in [\rho],\ \forall \hat{e} \in E_x,\ (c_{i,\hat{e}}^z, d_{i,\hat{e}}^z) \leftarrow \text{Com}_k(z_{i,\hat{e}})$

$a_1, \ldots, a_\rho, \left\{ c_{1,\hat{e}}^z, \ldots, c_{\rho,\hat{e}}^z \right\}_{\hat{e} \in E_x}$

$d_1^e, \ldots, d_\rho^e$

$\forall i \in [\rho],\ e_i := \text{Open}_k(c_i^e, d_i^e)$

$d_{1,e_1}^z, \ldots, d_{\rho,e_\rho}^z$

$\forall i \in [\rho],\ z_{i,e_i} := \text{Open}_k(c_{i,e_i}^z, d_{i,e_i}^z)$

outputs x if :

$\forall i \in [\rho],\ \text{Verify}(x, a_i, e_i, z_{i,e_i}) = 1$

Efficiency?

Assuming we start with some circuit C, our final protocol to realize $\mathcal{F}_{\mathsf{ZK}}$ is quite hefty.

After transforming the circuit into an instance of 3-coloring, the basic ZKP is $\Theta(|C|)$-special-sound.

Applying the protocol compiler yields a final protocol that contains $\Theta(|C| \cdot \kappa)$ repetitions of the ZKP, and for each repetition it transmits $\Theta(|C|)$ commitments. If each commitment is $\Omega(\kappa)$ bits, this gives us $\Omega(|C|^2 \cdot \kappa^2)$ bits in total, *even ignoring the cost of the original sigma protocol.*

An even bigger potential source of inefficiency comes from transforming problems into boolean circuits in the first place. Consider the language of elements in a (multiplicative) group, which has a membership checker algorithm R_{DL} such that $R_{\mathsf{DL}}((g, h), x) = 1 \iff g^x = h$.

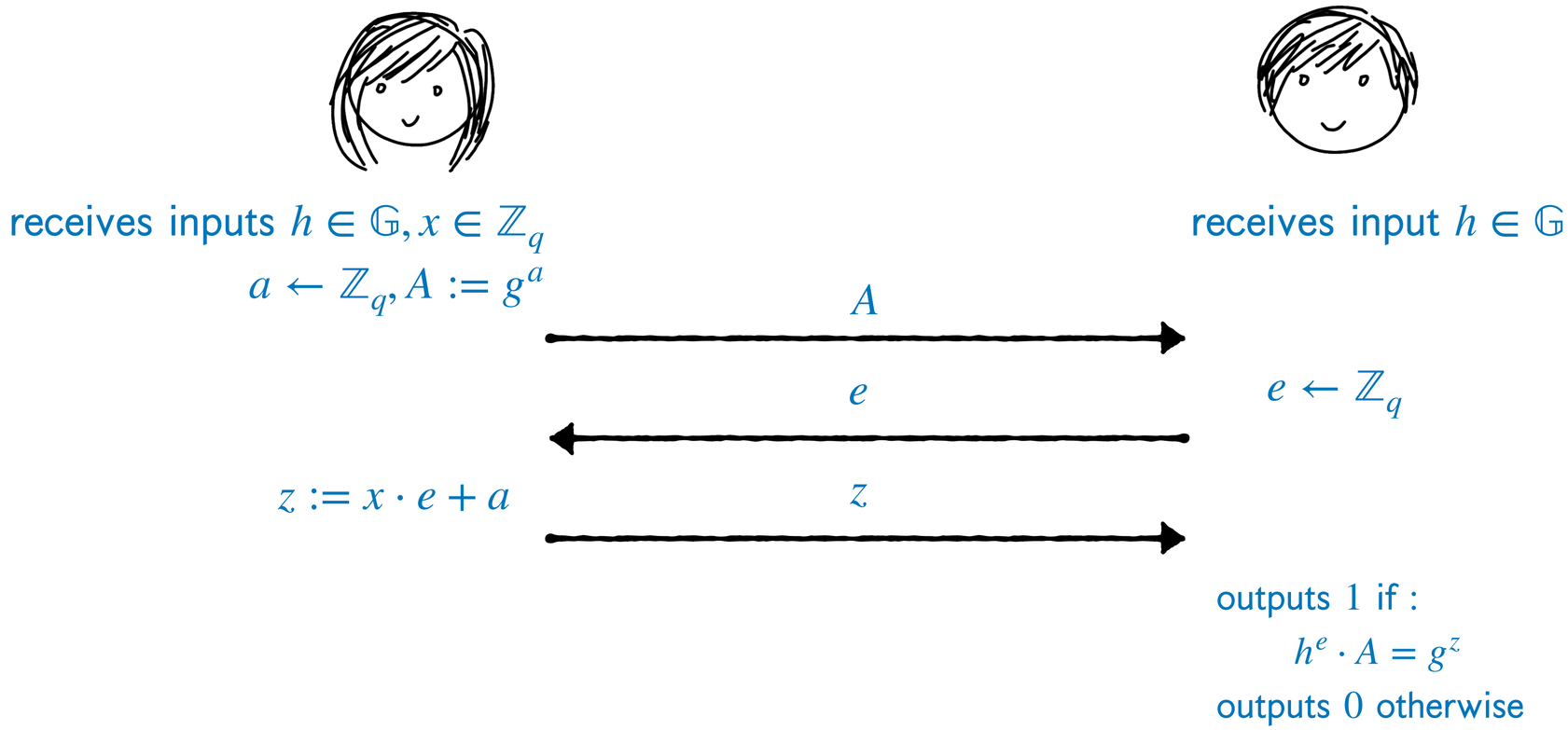
Suppose that the group operation is modular multiplication. If the modulus is ~100 bits, then the naïve circuit to perform a single modular exponentiation would be ~1,000,000 gates or more!

Fortunately, it turns out that for many interesting languages, there are efficient, language-specific sigma protocols that do *not* require us to reduce to graph 3-coloring.

One well-known family of such protocols was introduced by Claus P. Schnorr in 1990. Schnorr's Protocol allow us to prove knowledge of discrete logarithms efficiently, and it can be adapted to prove knowledge of Diffie-Hellman tuples, ElGamal openings, etc.

Def 4: The Schnorr Protocol

- Let $(\mathbb{G}, g, q)$ be the description of a cyclic group such that $q \in \mathbb{N}$ is prime, $|\mathbb{G}| = q$, $|q| = \kappa$, and $\langle g \rangle = \mathbb{G}$ under some operation (written multiplicatively).



Properties of Schnorr

receives inputs $h \in \mathbb{G}, x \in \mathbb{Z}_q$

$a \leftarrow \mathbb{Z}_q, A := g^a$

receives input $h \in \mathbb{G}$

A

$e \leftarrow \mathbb{Z}_q$

e

$z := x \cdot e + a$

z

outputs 1 if :

$h^e \cdot A = g^z$

outputs 0 otherwise

1. ***Correctness:*** observe that if $h = g^x$, then $h^e \cdot A = g^{x \cdot e} \cdot g^a = g^{x \cdot e + a} = g^z$.

2. ***Special Honest Verifier Zero Knowledge:***

   $\mathsf{Sim}(h, e)$ simply computes $A := g^z \cdot h^{-e}$, which is distributed identically to a real transcript, conditioned on a particular value of e.

3. ***2-Special-Soundness:***

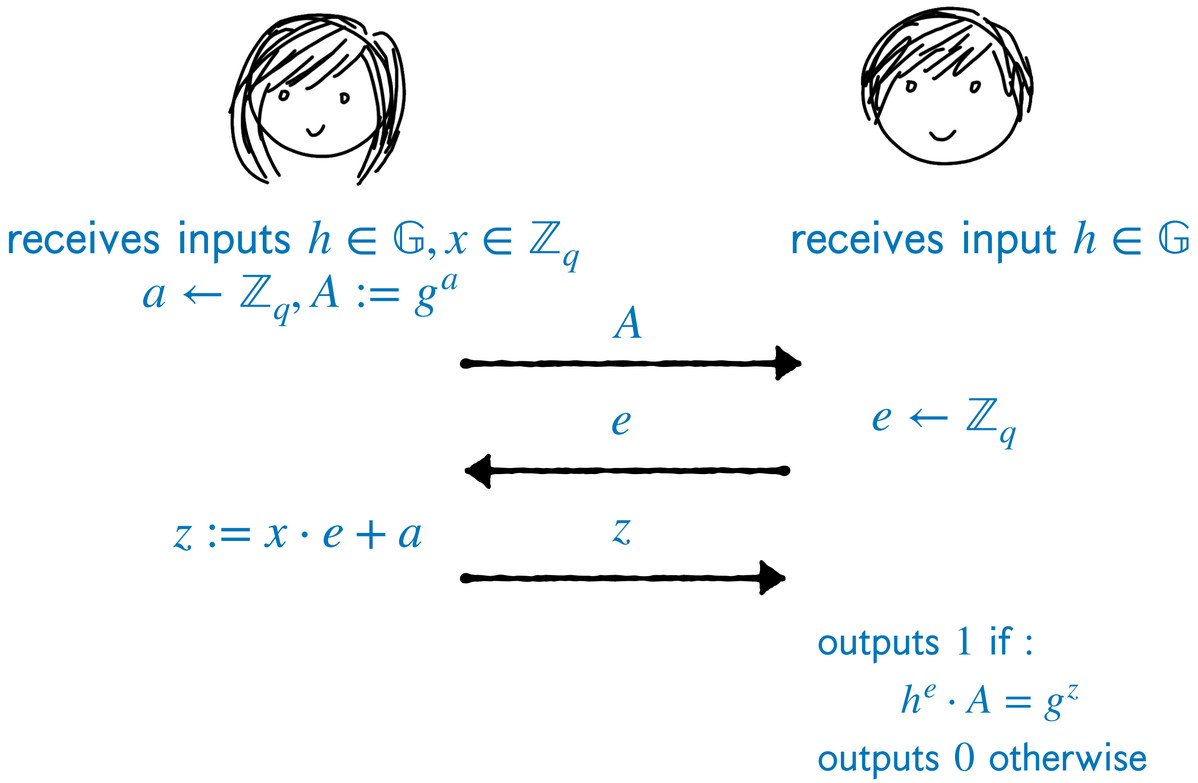
   If we have $(h = g^x, A = g^a, e, e', z, z')$ such that $h^e \cdot A = g^z \wedge h^{e'} \cdot A = g^{z'} \wedge e \neq e'$, then it must be the case that

   $h^e \cdot h^{-e'} = g^z \cdot g^{-z'}$, which implies

   $x \cdot (e - e') = z - z'$, which implies

   $x = (z - z')/(e - e')$, and since the group order is prime and $e - e' \neq 0$, this inverse is guaranteed to exist.

Properties of Schnorr



- What happens when we apply the protocol compiler to this?
- Since the protocol is 2-special sound, we can restrict our challenge space to two values (WLOG, $\{0,1\}$).
- Each repetition has soundness error $1/2$. Thus we exactly κ repetitions to ensure that the total soundness error is $2^{-\kappa}$.
- Thus, the prover sends 2κ commitments in total, and the verifier sends κ, in addition to the costs of κ copies of the underlying protocol (one element of $\mathbb{G}$ and two elements of $\mathbb{Z}_q$, per copy).
- This is *much* more efficient than transforming the group operations into a circuit.
- We can devise similar sigma protocols for *many* relationships related to groups.

Example: Schnorr for ElGamal Commitments

- Let $(\mathbb{G}, g, q)$ be the description of a cyclic group such that $q \in \mathbb{N}$ is prime, $|\mathbb{G}| = q$, $|q| = \kappa$, and $\langle g \rangle = \mathbb{G}$ under some operation (written multiplicatively).

receives inputs $X, C_1, C_2 \in \mathbb{G}, m, r \in \mathbb{Z}_q$ such that $C_1 = g^r \wedge C_2 = X^r \cdot g^m$

receives input $X, C_1, C_2 \in \mathbb{G}$

$a_1 \leftarrow \mathbb{Z}_q, A_1 := g^{a_1}$

$a_2 \leftarrow \mathbb{Z}_q, A_2 := X^{a_1} \cdot g^{a_2}$

A_1, A_2 →

e ←

$e \leftarrow \mathbb{Z}_q$

$z_1 := r \cdot e + a_1$

$z_2 := m \cdot e + a_2$

z_1, z_2 →

outputs 1 if :

$C_1^e \cdot A_1 = g^{z_1} \wedge C_2^e \cdot A_2 = g^{z_2} \cdot X^{z_1}$

outputs 0 otherwise

Several lectures ago, we introduced the idea of commitments, and then the idea of zero knowledge, because we couldn't figure out how to simulate Blum's Coin Tossing protocol.

Now let's return to that problem and see if we can make progress.

Property-Based Coin Tossing

Definition 5. Let $\kappa, n, t \in \mathbb{N}$ such that $t < n$, and let $P_1, \ldots, P_n$ be a group of parties that are connected by synchronous, authenticated channels. Let π be a protocol in which every P_i produces an output $y_i \in \{0,1\}$. We say that π is a (t, δ)-secure *Coin Tossing* protocol if the following conditions hold when $\mathcal{A}$ maliciously corrupts up to t parties:

1. **Consistency:** All honest parties output the same bit y with probability all-but-negligible in κ.
2. **Bias-Freeness:** $\left|\Pr[y = 0] - 1/2\right| \leq \delta(\kappa)$. We call δ the *bias*.

Unfortunately, in Lecture 23 we proved:

Cleve's Theorem: For every $\rho \in \mathbb{N}$, every 2-party, ρ-round, $(1,\delta)$-secure coin-tossing protocol in the plain model must $\delta \in \Omega(1/\rho)$, even against fail-stop adversaries. (This can be generalized to any $n \in \mathbb{N}$ and $t \geq n/2$, even if we use $\mathcal{F}_{\text{PKI}}$ or $\mathcal{F}_{\text{BC}}$).

The best we can hope for in the dishonest-majority setting is a weaker notion with *abort*:

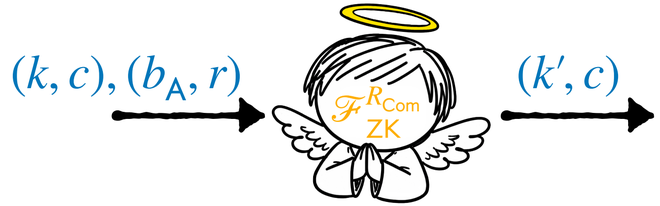
When $\mathcal{F}_{\text{CT-Abort}}$ is invoked, it samples $b \leftarrow \{0,1\}$ and sends b directly to $\mathcal{S}$. If $\mathcal{S}$ replies "OK", then $\mathcal{F}_{\text{CT-Abort}}$ sends b to the parties. If $\mathcal{S}$ replies "Abort", then $\mathcal{F}_{\text{CT-Abort}}$ sends $\perp$ to the parties.

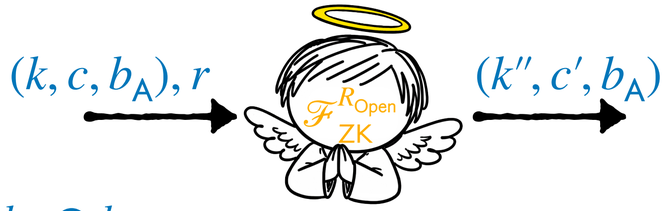
Blum's Coin Tossing

$b_A \leftarrow \{0,1\}$ $\qquad$ $b_B \leftarrow \{0,1\}$

$r \leftarrow$ randomness domain for Com

$(c, d) \leftarrow \text{Com}_k(b_A; r)$





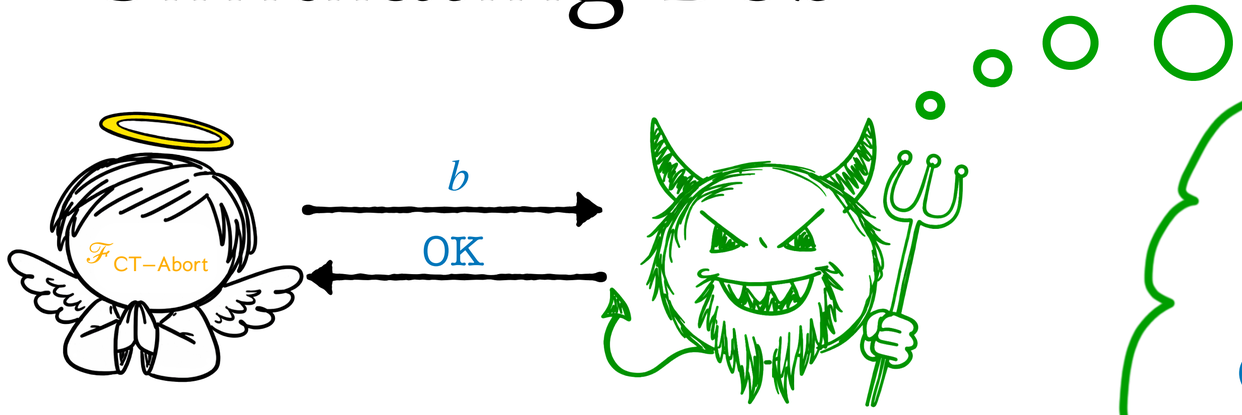
outputs $b_A \oplus b_B$

outputs $b_A \oplus b_B$ if $k'' = k' = k \wedge c' = c$ otherwise outputs $\perp$

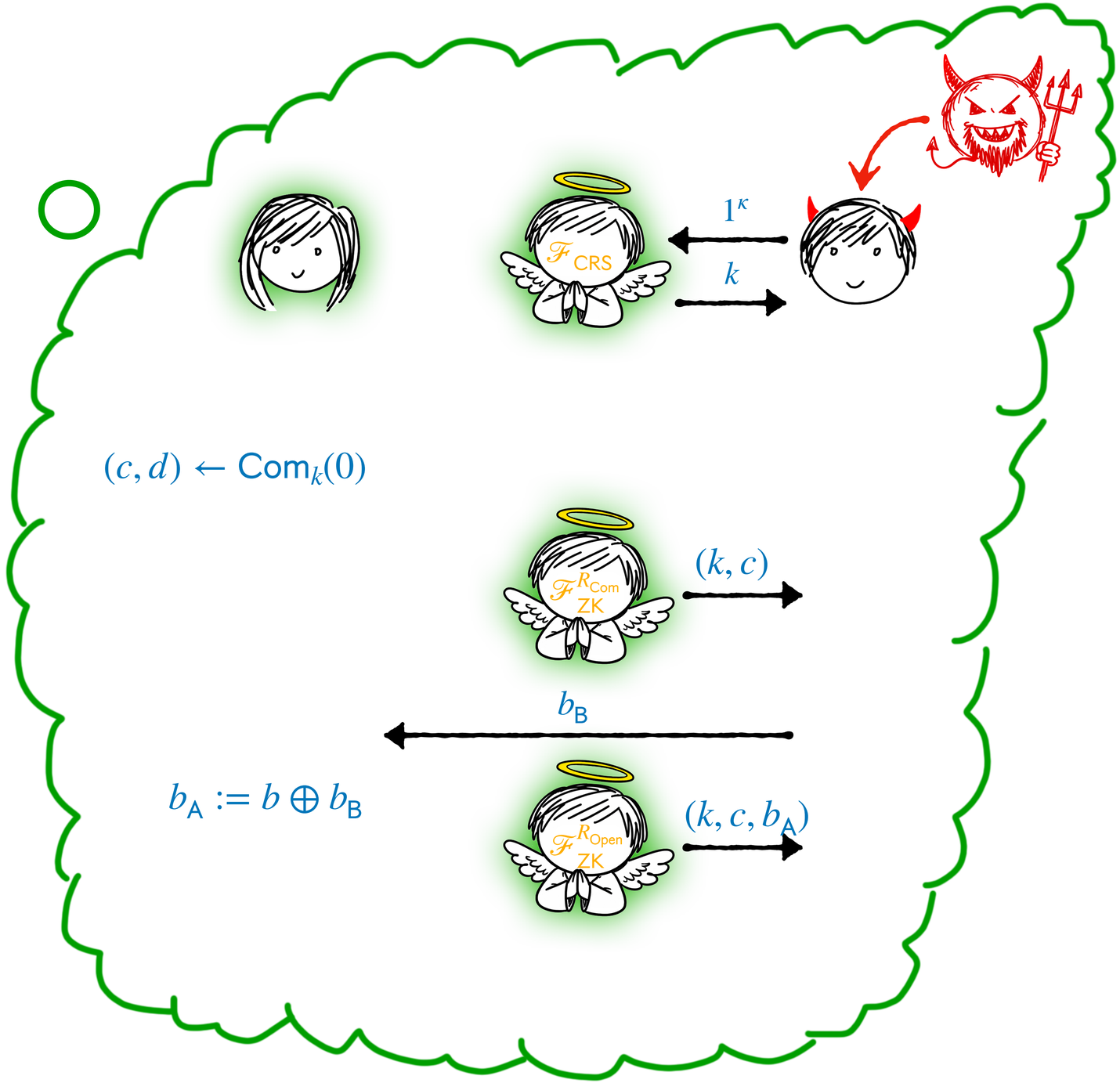
- Let (Gen, Com, Open) be a commitment scheme.
- Let L_{Com} be the language of valid commitments.
  i.e., $R_{\text{Com}}((k, c), (m, r)) = 1 \iff \text{Com}_k(m; r) = c$.
- Let L_{Open} be the language of valid openings.
  i.e., $R_{\text{Open}}((k, c, m), r) = 1 \iff \text{Com}_k(m; r) = c$.
- We will reformulate Blum's Coin-tossing protocol in the $(\mathcal{F}_{\text{ZK}}^{R_{\text{Com}}}, \mathcal{F}_{\text{ZK}}^{R_{\text{Open}}}, \mathcal{F}_{\text{CRS}})$-hybrid model, and prove:

Theorem 2: Our reformulation of Blum's Coin Tossing Protocol realizes $\mathcal{F}_{\text{CT-Abort}}$ in the presence of a malicious adversary corrupting either party.

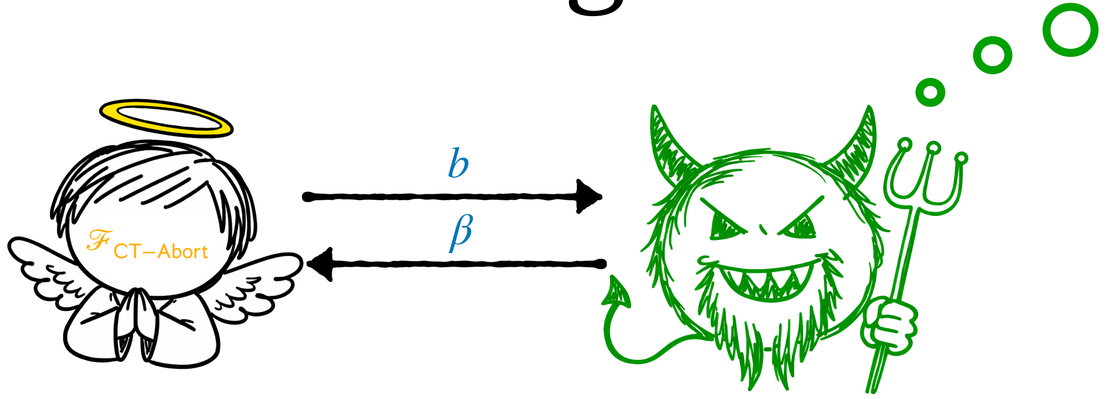
Simulating Bob



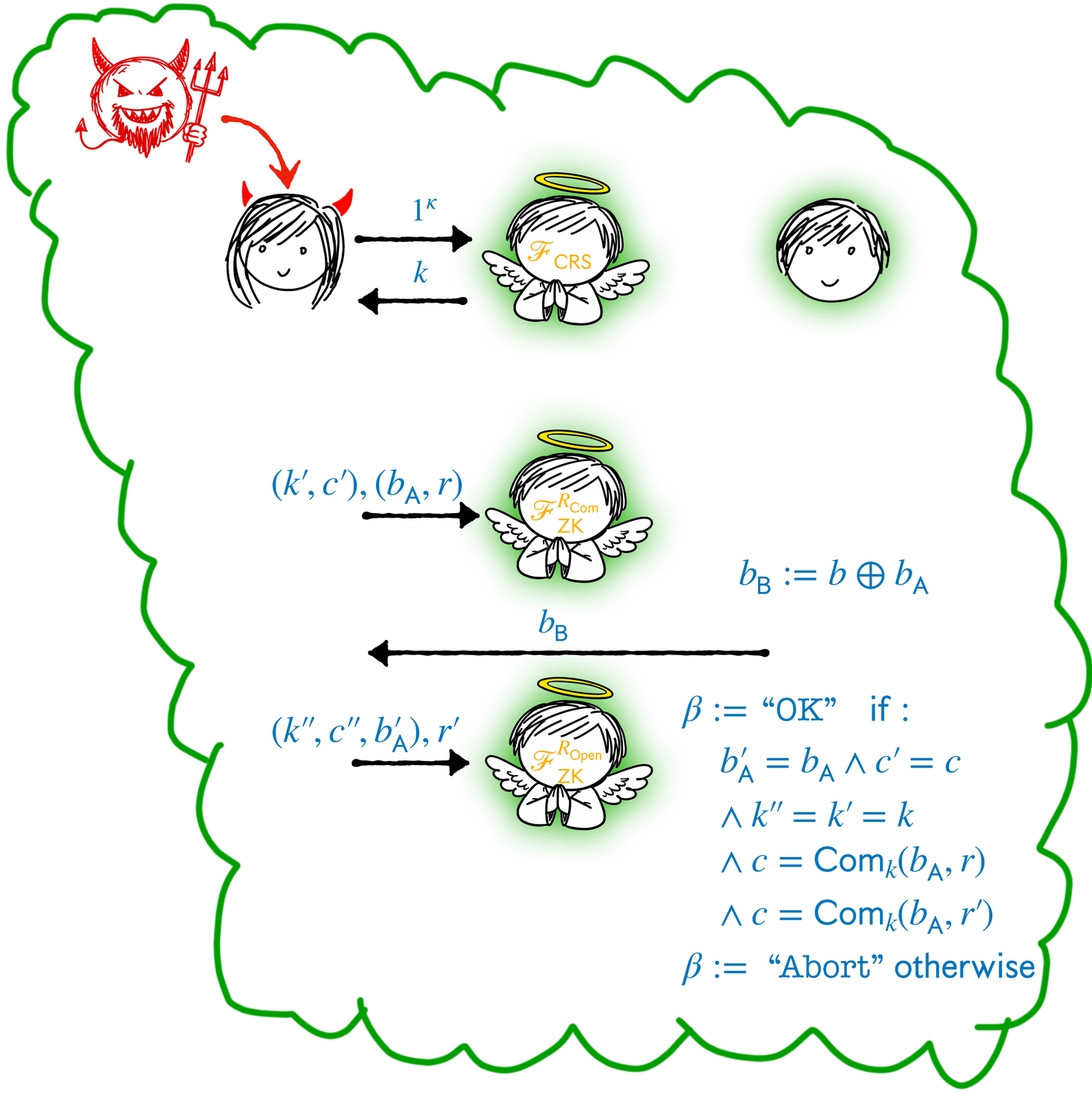
- The only difference between the view of corrupt Bob when he interacts with $\mathcal{S}_{\mathsf{B}}$, relative to the real world, is that c is a commitment to 0 instead of b_{A}. Otherwise, they are identically distributed.
- Thus, we can write a reduction to the hiding game for the commitment scheme, and conclude that the real world and the ideal world are computationally indistinguishable to a corrupt Bob.



Simulating Alice



- The view of corrupt Alice when she interacts with $\mathcal{S}_{\mathsf{A}}$ is identical to her view in the real world, *unless* $\mathcal{A}$ finds two different openings $(b_{\mathsf{A}}, r), (b'_{\mathsf{A}}, r')$ to a single commitment c such that $b_{\mathsf{A}} \neq b'_{\mathsf{A}}$
- Thus, we can write a reduction to the binding game for the commitment scheme, and conclude that the real world and the ideal world are computationally indistinguishable to a corrupt Alice. ∎

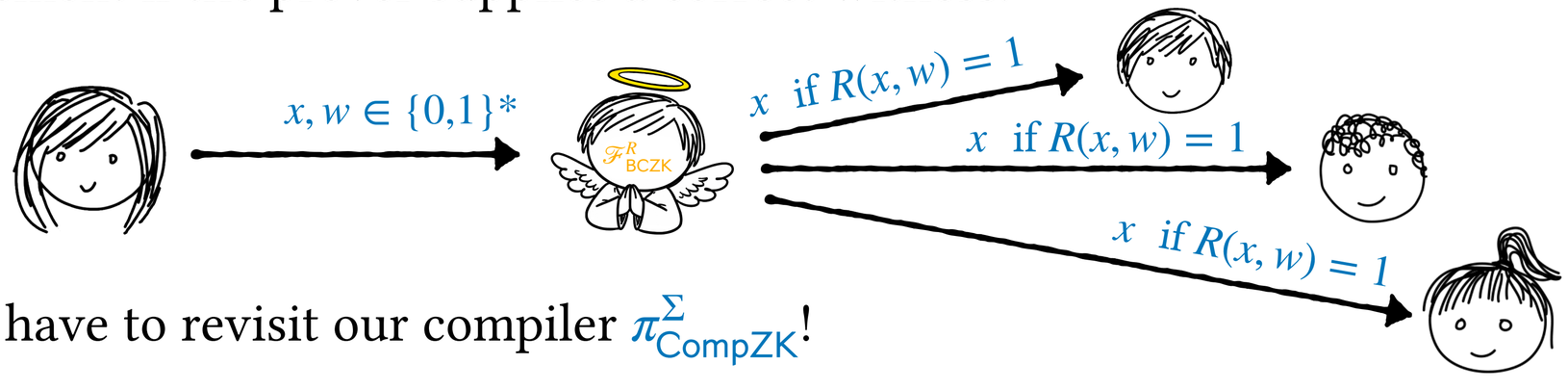


Towards Multi-Party Coin Tossing

- In the two-party setting, when one party is corrupt, *consistency* is meaningless, and so we only have to worry about *bias*. We want to generalize this idea to *many* participants, however.
- The most natural extension would be to have every party P_i send a single commitment to their random bit b_i over a broadcast channel (that is, using $\mathcal{F}_{\mathsf{BC}}$). Since they all agree on the commitment, they should all agree on the bit (unless P_i breaks binding).
- $\mathcal{S}$ must be able to extract all corrupt contributions, regardless of which parties are corrupt, so that it can equivocate the honest contributions appropriately.
- The zero-knowledge functionality $\mathcal{F}_{\mathsf{ZK}}$ that we defined only interacts with two parties.
- What if we just use it pairwise? Broadcast a commitment, then prove to everyone individually that you know what the commitment contains. Binding implies you must be proving the same contents to everyone. *Is there anything else that can go wrong?*
- The parties might not agree on whether to abort! If only one proof fails, only one party aborts.
- But $\mathcal{F}_{\mathsf{CT\text{-}Abort}}$ says that they *must* agree on aborts.

Towards Multi-Party Coin Tossing

- But $\mathcal{F}_{\text{CT-Abort}}$ says that they *must* agree on aborts.
- In the context of coin tossing, we *could* ask everyone to broadcast another bit indicating whether they will abort.
- Later, it will be important to make sure that malicious verifiers can't falsely accuse honest provers of cheating.
- When we need the latter property, we *cannot* rely on a pairwise $\mathcal{F}_{\text{ZK}}$. Instead, we need a *broadcast* zero-knowledge functionality $\mathcal{F}_{\text{BCZK}}$ that allows one party to prove to many verifiers.
- $\mathcal{F}_{\text{BCZK}}$ needs a *consistency* property, which says that an honest verifier receives the statement if and only if all other honest verifiers do, and a *validity* property, which says that the honest verifiers always receive the statement if the prover supplies a correct witness.



- In order to achieve this, we will have to revisit our compiler $\pi^{\Sigma}_{\text{CompZK}}$!

Recall: Sigma Protocols

Recall **Definition 1:** A Sigma Protocol is an IPS with the following 3-message structure:

1. The prover sends a statement x and a commitment message a to the verifier.
2. The verifier samples a challenge e *uniformly* from some set E, and sends it to he prover.
3. The prover sends a response z to the challenge.

Sigma protocols are *public coin* zero-knowledge, which means that the verifier has no secret state. They simply sample random values and send them to the prover.

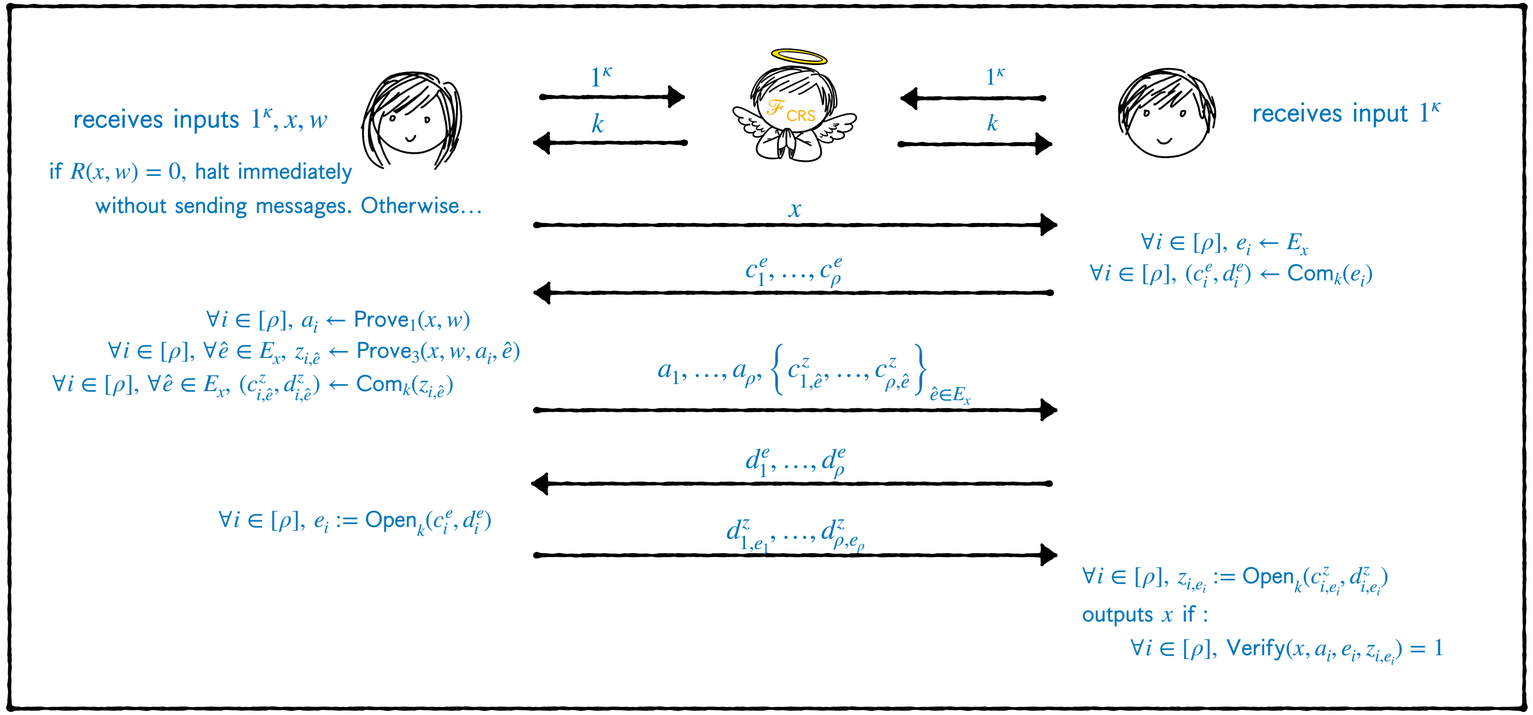
This is convenient in our case, because it means that anyone who sees the *transcript* of the proof, they can also check the outcome.

Specifically, if both prover and verifier are corrupt, and a third party checks the transcript, that third party does *not* get any soundness guarantee. After all, the verifier could have colluded with the prover to make the protocol succeed.

However, this does mean that corrupt parties cannot frame honest ones: Since a third party can see whether or not the prover sent a valid response, a corrupt verifier can't claim an honest prover failed.

A Compiler for Composable *Broadcast* ZK

- We will begin from our two-party compiler $\pi^{\Sigma}_{\mathsf{CompZK}}$ from earlier.
- Suppose we have parties $P_1, \ldots, P_n$. P_1 wishes to prove x to the others.
- First ,the group receives a single k from $\mathcal{F}_{\mathsf{CRS}}$, and P_1 broadcasts x.
- Then, for every $i \in [2,n]$, P_1 and P_i run the *rest* of $\pi^{\Sigma}_{\mathsf{CompZK}}$, but they send all of their messages using $\mathcal{F}_{\mathsf{BC}}$ instead of a private channel.



- At the end, every verifiers run one another's checks. If any verifier *justifiably* aborts, then they all do.
- It's easy to extend our original security proof: if multiple verifiers are corrupt, the SHVZK simulator allows us to produce appropriate transcripts for all of them, even if they correlate their challenges.
- If the prover and a verifier are corrupt, the others can determine whether that verifier *should* abort.
- Extraction works independently for every copy of $\pi^{\Sigma}_{\mathsf{CompZK}}$ that has an honest verifier.

A Compiler for Composable *Broadcast* ZK

Thus we have…

Theorem 3: Let $n, t \in \mathbb{N}$ such that $n > t$, let ℓ be a polynomial, and let L be a language with membership-verifier algorithm R. If there exists an $\ell(|x|)$-special-sound SHVZK sigma protocol for L, and an extractable commitment scheme, then there exists a protocol in the $(\mathcal{F}_{\mathsf{CRS}}, \mathcal{F}_{\mathsf{BC}})$-hybrid model that realizes $\mathcal{F}^{R}_{\mathsf{BCZK}}$ among n parties, in the presence of a malicious adversary that statically corrupts up to t parties.

Corollary 1: Let $n, t \in \mathbb{N}$ such that $n > t$. For every NP-language L with membership-verifier algorithm R, there exists a protocol in the PKI+CRS model that realizes $\mathcal{F}^{R}_{\mathsf{BCZK}}$ among n parties, in the presence of a malicious adversary that statically corrupts up to t parties, assuming the Decisional Diffie-Hellman Assumption is true.

Proof: If the DDH assumption is true, then the DL assumption is true, which implies that OWFs exist, which implies that signatures exist via Lamport's construction, which implies that we can realize $\mathcal{F}_{\mathsf{BC}}$ in the PKI model via the Dolev-Strong protocol. As before, Cook-Levin plus our sigma protocol for L_{G3C} give us an appropriate sigma protocol for any NP-language. ∎

Multi-Party Coin Tossing

Let $\kappa \in \mathbb{N}$ be a security parameter and let $(\mathsf{Gen}, \mathsf{Com}, \mathsf{Open})$ be a commitment scheme.
Let $R_{\mathsf{Com}}((k, c), (m, r)) = 1 \iff \mathsf{Com}_k(m; r) = c$ and $R_{\mathsf{Open}}((k, c, m), r) = 1 \iff \mathsf{Com}_k(m; r) = c$.
Let $P_1, \ldots, P_n$ be a set of parties with access to $\mathcal{F}_{\mathsf{CRS}}$, $\mathcal{F}_{\mathsf{BCZK}}^{R_{\mathsf{Com}}}$, and $\mathcal{F}_{\mathsf{BCZK}}^{R_{\mathsf{Open}}}$.
The protocol $\pi_{\mathsf{CT\text{-}Abort}}$ is as follows:

Setup Phase:

1. Every P_i sends 1^κ to $\mathcal{F}_{\mathsf{CRS}}$, which samples $k \leftarrow \mathsf{Gen}(1^\kappa)$ and sends k to all of them.

Online Phase:

2. Every P_i samples $b_i \leftarrow \{0,1\}$ and r_i from the appropriate domain, computes $c_i := \mathsf{Com}_k(b_i, r_i)$, and sends $(k, c_i), (b_i, r_i)$ to $\mathcal{F}_{\mathsf{BCZK}}^{R_{\mathsf{Com}}}$, which sends (k', c_i) to everyone. They all verify that $k' = k$. If this condition does not hold, they all abort.

3. Every P_i sends $(k, c_i, b_i), r_i$ to $\mathcal{F}_{\mathsf{BCZK}}^{R_{\mathsf{Open}}}$, which sends (k'', c_i', b_i) to everyone, and they all verify that $k'' = k \wedge c_i' = c_i$. If this condition does not hold, they all abort. Otherwise, every party computes $b := b_1 \oplus \ldots \oplus b_n$ and outputs b.

Multi-Party Coin Tossing

Theorem 4: Let $n, t \in \mathbb{N}$ such that $n > t$. Assuming that $(\mathsf{Gen}, \mathsf{Com}, \mathsf{Open})$ is a secure commitment scheme, $\pi_{\mathsf{CT\text{-}Abort}}$ realizes $\mathcal{F}_{\mathsf{CT\text{-}Abort}}$ among n parties, in the presence of a malicious adversary that statically corrupts up to t parties.

Proof: $\mathcal{S}$ generalizes the simulator Blum's two-party coin tossing protocol, which we saw earlier.

Since $\mathcal{F}_{\mathsf{CT\text{-}Abort}}$ takes no inputs from the parties, $\mathcal{S}$ receives b from $\mathcal{F}_{\mathsf{CT\text{-}Abort}}$ immediately. $\mathcal{S}$ then emulates the real world experiment in its head toward the adversary $\mathcal{A}$, which runs as a subroutine, and performs the following steps:

1. Sample $k \leftarrow \mathsf{Gen}(1^\kappa)$ and sends k to all corrupt parties on behalf of $\mathcal{F}_{\mathsf{CRS}}$.
2. For every honest P_i, compute $c_i \leftarrow \mathsf{Com}(0)$ and send (k, c_i) to all corrupt parties on behalf of $\mathcal{F}_{\mathsf{BCZK}}^{R_{\mathsf{Com}}}$.

   From every corrupt P_j, receive $(k'_j, c_j), (b_j, r_j)$ on behalf of $\mathcal{F}_{\mathsf{BCZK}}^{R_{\mathsf{Com}}}$. If $k'_j \neq k$, or if $c_j \neq \mathsf{Com}_k(b_j; r_j)$, then send "Abort" to $\mathcal{F}_{\mathsf{CT\text{-}Abort}}$.

Multi-Party Coin Tossing

3. For every honest P_i, sample $b_i \leftarrow \{0,1\}$ subject to the restriction that $b_1 \oplus \ldots \oplus b_n = b$.

   For every honest P_i, send (k, c_i, b_i) to all corrupt parties on behalf of $\mathcal{F}_{\text{BCZK}}^{R_{\text{Com}}}$.

   From every corrupt P_j, receive $(k_j'', c_j', b_j'), r_j'$ on behalf of $\mathcal{F}_{\text{BCZK}}^{R_{\text{Com}}}$. If $k_j'' \neq k \vee c_j' \neq c_j \vee b_j' \neq b_j$, or if $c_j \neq \mathsf{Com}_k(b_j'; r_j')$, then send "`Abort`" to $\mathcal{F}_{\text{CT-Abort}}$.

   If no abort has occurred, send "`OK`" to $\mathcal{F}_{\text{CT-Abort}}$.

The argument for indistinguishability uses the same ideas that we saw in the context of Blum's protocol, although we need to introduce one hybrid experiment per party: in each honest party's hybrid experiment, we change the value it commits in Step 2 to 0, and prove indistinguishability from the previous hybrid by reducing to the hiding property of the commitment scheme.

In each corrupt party P_j's hybrid experiment, we add an explicit abort if $b_j' \neq b_j$ in Step 3, and prove indistinguishability from the previous hybrid by reducing to the binding property of the commitment scheme.

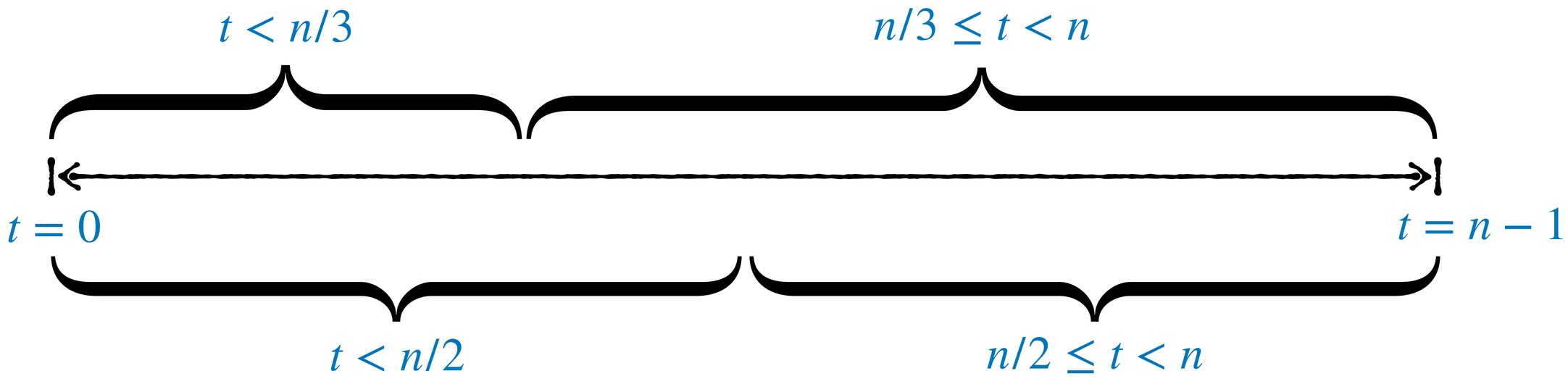
The hybrids can be traversed in any order. By the transitivity, we have the theorem. ∎

Where are We? The Landscape of Malicious Coin Tossing

(Let n be the number of parties and t be the number of corruptions)

Perfect Coin Tossing from malicious BGW in the plain model

No non-trivial bias-freeness in the plain model [CHOR 16]



Computational coin tossing from the compiled GMW protocol in the ($\mathcal{F}_{\mathsf{PKI}}$,$\mathcal{F}_{\mathsf{CRS}}$)-hybrid model.

Bias inversely proportional to rounds even assuming PKI if we want guaranteed output (Cleve)...

...but we can toss fair coins with with *abort.*

Since Lecture 18, when we introduced malicious adversaries and the GMW compiler for the first time, we've explored a lot of new ideas.

Specifically, we have introduced broadcast, coin tossing, and zero-knowledge, and determined exactly when each of these primitives can be realized.

Let's review our plan for the compiler and see where we stand…

Recall: intuitive preview of the GMW Compiler

- How will our compiler work? We start with a semi-honest protocol and change it:
- First, we will move all communication onto a *broadcast channel*. If the original protocol says two parties should communicate privately, they must encrypt their messages to one another, and broadcast the ciphertexts to *everyone*.
- At the beginning, the parties use secure coin-tossing to sample whatever private randomness every party might eventually need. Everyone receives a *commitment* to everyone else's randomness, which binds them to what was sampled.
- Similarly, all parties produce commitments that bind them to their *inputs*.
- All messages in the original protocol can be computed deterministically from the inputs, randomness, and prior transcript. Whenever a party in the original protocol sends a message, the party in the compiled protocol must prove in zero knowledge that the message they sent was computed *correctly* given their committed input and randomness, and the transcript of prior messages on the broadcast channel.

Recall: intuitive preview of the GMW Compiler

- What if somebody cheats?
- Because broadcast, coin-tossing, and zero-knowledge proofs are *ideal*, cheating can only take the form of failing to send a message or sending a message that fails its corresponding proof of correctness. Since all parties see the same broadcast channel, all parties can agree on who cheated.
- If there are $t \geq n/2$ parties, then the honest parties *abort* and identify the cheater.
- If there are $t < n/2$ parties, then our *commitments* can take a special form known as *verifiable secret sharing*. Since the honest parties agree who cheated, they can kick out the cheater, reconstruct the cheater's inputs and randomness from secret shares (which they verified to be correct when they received them), and try again!
- This is a very intuitive overview. The first time you saw this slide, I said the devil was in the details. If you didn't agree then, maybe you do now!

Another look at the role of Zero Knowledge

- All messages in the original protocol can be computed deterministically from the inputs, randomness, and prior transcript. Whenever a party in the original protocol sends a message, the party in the compiled protocol must prove in zero knowledge that the message they sent was computed *correctly* given their committed input and randomness, and the transcript of prior messages on the broadcast channel.
- Let's focus just on this step for a moment. The way we wrote it originally is complicated and a little imprecise. Let's be a little more careful.
- In the original protocol, parties send private messages using secret state.
- In the compiled protocol we want to *commit* to all secret state.
- Each time P_i sends a message privately to P_j in the original protocol, we need to send the same message in the compiled protocol in such a way that the message is still private, but the *other* convinced it was delivered, and everyone is convinced that it's was correct.

Another look at the role of Zero Knowledge

- Let's focus just on this step for a moment. The way we wrote it originally is complicated and a little imprecise. Let's be a little more careful.
- In the original protocol, parties send private messages using secret state.
- In the compiled protocol we want to *commit* to all secret state.
- Each time P_i sends a message privately to P_j in the original protocol, we need to send the same message in the compiled protocol in such a way that the message is still private, but the *other* parties are convinced it was delivered, and everyone is convinced that it's was correct.
- We're going to call this idea *authenticated computation.* Specifically, we want to:
  - perform *private image transmission* (send one party the output of a function)
  - on *publicly-committed data* (the inputs should be committed to everyone)
  - with *public authentication* (everyone must be convinced computation was correct)

Commit-and-Privately-Evaluate Functionality

Definition 6 ($\mathcal{F}_{\text{CPE}}$):

This functionality interacts with n parties, $P_1, \ldots, P_n$.

- At initialization time, $\mathcal{F}_{\text{CPE}}$ sets $W_i := \lambda$ in memory for every $i \in [n]$.
- Upon receiving (commit, w) from P_i, where $w \in \{0,1\}^*$, $\mathcal{F}_{\text{CPE}}$ updates $W_i := W_i \| w$ in memory and sends $(\text{received}, i)$ to all other parties.
- Upon receiving (eval, j, C) from P_i, where $j \in [n] \backslash \{i\}$ and C is description of a boolean circuit that takes $|W_i|$ bits of input:
  1. $\mathcal{F}_{\text{CPE}}$ sends $(\text{image}, i, C, C(W_i))$ to P_j.
  2. $\mathcal{F}_{\text{CPE}}$ sends $(\text{evaluated}, i, j, C)$ to every P_k such that $k \in [n] \backslash \{i, j\}$.
  3. $\mathcal{F}_{\text{CPE}}$ updates $W_j := W_j \| C(W_i)$ in memory.

For the proof, we will consider a version of this functionality where we only commit *one* bit at a time, and C may only produce *one* bit. The original functionality can be built from this one-bit version easily by repeatedly calling commit or eval.

Commit-and-Privately-Evaluate Protocol

Theorem: Let $n, t \in \mathbb{N}$ such that $n > t$. Assuming the existence of commitment schemes and CPA-secure encryption, there is a protocol in the $(\mathcal{F}_{\mathsf{CRS}}, \mathcal{F}_{\mathsf{BC}}, \mathcal{F}_{\mathsf{BCZK}})$-hybrid model that realizes (the 1-bit version of) $\mathcal{F}_{\mathsf{CPE}}$ in the presence of a malicious adversary that statically corrupts up to t parties.

Proof Sketch:

Suppose that (CGen, Com, Verify) is a commitment scheme and (EGen, Enc, Dec) is a CPA-secure encryption scheme. Suppose also that we define the following languages implicitly by their membership checker functions:

- $R_{\mathsf{Com}}((k, c), (m, r)) = 1 \iff \mathsf{Com}_k(m; r) = c \wedge m \in \{0,1\}$ is the valid bit commitments.

Commit-and-Privately-Evaluate Protocol

Proof Sketch:

Suppose that (CGen, Com, Verify) is a commitment scheme and (EGen, Enc, Dec) is a CPA-secure encryption scheme. Suppose also that we define the following languages implicitly by their membership checker functions:

- $R_{\mathsf{Com}}((k, c), (m, r)) = 1 \iff \mathsf{Com}_k(m; r) = c \wedge m \in \{0,1\}$ is the valid bit commitments.
- For every $\ell \in \mathbb{N}$ and every boolean circuit C that takes ℓ bits of input and outputs one bit, $R_{\mathsf{Com}\text{-}C\text{-}\mathsf{Enc}}((\mathsf{pk}, \mathsf{ct}, k, c_1, \ldots, c_\ell, c_{\ell+1}), (r_{\mathsf{ct}}, w_1, \ldots, w_\ell, r_1, \ldots, r_{\ell+1})) = 1$

$$\iff \bigwedge_{i \in [\ell]} \mathsf{Com}_k(w_i; r_i) = c_i$$
$$\wedge\ \mathsf{Com}_k(C(w_1 \| \ldots \| w_\ell); r_{\ell+1}) = c_{\ell+1}$$
$$\wedge\ \mathsf{Enc}_{\mathsf{pk}}(C(w_1 \| \ldots \| w_\ell) \| r_{\ell+1}; r_{\mathsf{ct}}) = \mathsf{ct}$$

  defines the language where $c_{\ell+1}$ is a commitment the output of C on the values committed in $c_1, \ldots, c_\ell$, and ct is an encryption under pk of the *opening* for $c_{\ell+1}$.

Commit-and-Privately-Evaluate Protocol

Finally, the protocol π_{CPE}:

Setup Phase:

1. Every P_i sends 1^κ to $\mathcal{F}_{\mathsf{CRS}}$, which samples $k \leftarrow \mathsf{Gen}(1^\kappa)$ and sends k to all of them.
2. Every P_i samples $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{EGen}(1^\kappa)$ and sends pk_i to $\mathcal{F}_{\mathsf{BC}}$, which in turn delivers pk_i to everybody.
3. The amount of every party's accumulated witness is initialized as zero in every party's view.

Commit Phase:

4. When P_i receives its ℓ^{th} commit instruction, $(\mathsf{commit}, w_{i,\ell})$, where $w_{i,\ell} \in \{0,1\}$, P_i samples $r_{i,\ell}$ from the appropriate domain, computes $c_{i,\ell} := \mathsf{Com}_k(w_{i,\ell}; r_{i,\ell})$, and sends $(k, c_{i,\ell}), (w_{i,\ell}, r_{i,\ell})$ to $\mathcal{F}_{\mathsf{BCZK}}^{R_{\mathsf{Com}}}$.

   The other parties receive $(k', c_{i,\ell})$ from $\mathcal{F}_{\mathsf{BCZK}}^{R_{\mathsf{Com}}}$, and if $k' = k$ they output $(\mathsf{received}, i)$ and add 1 to the amount of P_i's accumulated witness. Otherwise they do nothing.

Commit-and-Privately-Evaluate Protocol

Eval Phase:

5. When P_i receives an (eval, j, C) instruction, such that $j \in [n] \backslash \{i\}$ and C takes ℓ bits of input, where ℓ is the amount of P_i's accumulated witness:

   Let ℓ' be the amount of P_j's accumulated witness. P_i samples $r_{\mathsf{ct}}, r_{\ell'+1}$ as appropriate, computes

$$
\begin{aligned}
y &:= C(w_{i,1} \| \ldots \| w_{i,\ell}) \\
c_{j,\ell'+1} &:= \mathsf{Com}_k(y; r_{\ell'+1}) \\
\mathsf{ct} &:= \mathsf{Enc}_{\mathsf{pk}_j}(y \| r_{\ell'+1}; r_{\mathsf{ct}})
\end{aligned}
$$

   and then sends $(\mathsf{pk}_j, \mathsf{ct}, k, c_{i,1}, \ldots, c_{i,\ell}, c_{j,\ell'+1}), (r_{\mathsf{ct}}, w_{i,1}, \ldots, w_{i,\ell}, r_{i,1}, \ldots, r_{i,\ell}, r_{j,\ell'+1})$ to $\mathscr{F}_{\mathsf{BCZK}}^{R_{\mathsf{Com}\text{-}C\text{-}\mathsf{Enc}}}$.

Commit-and-Privately-Evaluate Protocol

$$y := C(w_{i,1} \| \ldots \| w_{i,\ell})$$
$$c_{j,\ell'+1} := \mathsf{Com}_k(y; r_{\ell'+1})$$
$$\mathsf{ct} := \mathsf{Enc}_{\mathsf{pk}_j}(y \| r_{\ell'+1}; r_{\mathsf{ct}})$$

and then sends $(\mathsf{pk}_j, \mathsf{ct}, k, c_{i,1}, \ldots, c_{i,\ell}, c_{j,\ell'+1}), (r_{\mathsf{ct}}, w_{i,1}, \ldots, w_{i,\ell}, r_{i,1}, \ldots, r_{i,\ell}, r_{j,\ell'+1})$ to $\mathscr{F}_{\mathsf{BCZK}}^{R_{\mathsf{Com}\text{-}C\text{-}\mathsf{Enc}}}$.

6. The other parties receive $(\mathsf{pk}_j', \mathsf{ct}, k', c'_{i,1}, \ldots, c'_{i,\ell}, c'_{j,\ell'+1})$ from $\mathscr{F}_{\mathsf{BCZK}}^{R_{\mathsf{Com}\text{-}C\text{-}\mathsf{Enc}}}$.

   If $(k', c'_{i,1}, \ldots, c'_{i,\ell}) = (k, c_{i,1}, \ldots, c_{i,\ell})$ and there exists some $j \in [n] \backslash \{i\}$ such that $\mathsf{pk}_j' = \mathsf{pk}_j$, then:

   - P_j computes $w_{j,\ell'+1} \| r_{j,\ell'+1} := \mathsf{Dec}_{\mathsf{sk}_j}(\mathsf{ct})$ and outputs $(\mathsf{image}, i, C, w_{j,\ell'+1})$
   - The others output $(\mathsf{evaluated}, i, j, C)$
   - Everyone adds 1 to the amount of P_j's accumulated witness.

   Otherwise, the parties do nothing.

Commit-and-Privately-Evaluate Simulator

We will sketch a strategy for constructing $\mathcal{S}$:

In the protocol, all interactions are performed using $\mathcal{F}_{\mathsf{CRS}}$, $\mathcal{F}_{\mathsf{BC}}$, and $\mathcal{F}_{\mathsf{BCZK}}$. No messages are ever directly transmitted. Furthermore, no commitments issued by the honest parties are ever opened using the regular Open algorithm. Keep this in mind.

We need to handle 6 types of interaction:

1. *Commitments by honest parties:* $\mathcal{S}$ simply commits to 0.
2. *Commitments by corrupt parties:* $\mathcal{S}$ can extract the committed value via $\mathcal{F}_{\mathsf{BCZK}}$, and send the committed value to $\mathcal{F}_{\mathsf{CPE}}$ on behalf of the corrupt party.
3. *Evaluations by honest, to corrupt:* $\mathcal{S}$ receives y (with source and destination indices) from $\mathcal{F}_{\mathsf{CPE}}$, commits to it and encrypts it with the corrupt public key, and delivers it via $\mathcal{F}_{\mathsf{BCZK}}$.
4. *Evaluations by corrupt, to honest:* $\mathcal{S}$ receives ct (with source and destination indices) via $\mathcal{F}_{\mathsf{BCZK}}$. Since $\mathcal{S}$ sampled the secret key under which ct was encrypted, $\mathcal{S}$ can easily decrypt y and send it to $\mathcal{F}_{\mathsf{CPE}}$ on behalf of the corrupt party.
5. *Evaluations by honest, to honest:* $\mathcal{S}$ simply commits to and encrypts 0.

Commit-and-Privately-Evaluate Simulator

3. *Evaluations by honest, to corrupt:* $\mathcal{S}$ receives y (with source and destination indices) from $\mathcal{F}_{\mathsf{CPE}}$, commits to it and encrypts it with the corrupt public key, and delivers it via $\mathcal{F}_{\mathsf{BCZK}}$.

4. *Evaluations by corrupt, to honest:* $\mathcal{S}$ receives ct (with source and destination indices) via $\mathcal{F}_{\mathsf{BCZK}}$. Since $\mathcal{S}$ sampled the secret key under which ct was encrypted, $\mathcal{S}$ can easily decrypt y and send it to $\mathcal{F}_{\mathsf{CPE}}$ on behalf of the corrupt party.

5. *Evaluations by honest, to honest:* $\mathcal{S}$ simply commits to and encrypts 0.

6. *Evaluations by corrupt, to corrupt:* The corrupt parties are required to prove to everyone that they commit and encrypt correctly using the appropriate public keys, and that they are committed to the preimage of the image they are transmitting, even if the destination is another corrupt party. Since $\mathcal{S}$ *already* has knowledge of the values the corrupt parties have committed, and the circuit to be evaluated is deterministic, $\mathcal{S}$ can easily recompute the output.

   In other words, if the corrupt parties wish to perform private evaluations toward *one another* in such a way that the honest parties *recognize* that their communication occurred, then $\mathcal{S}$ can extract the values they transmit to one another, and then send the extracted values to $\mathcal{F}_{\mathsf{CPE}}$.

We can prove that the above simulation is indistinguishable from the real protocol by a sequence of hybrid experiments, which we will sketch.

Commit-and-Privately-Evaluate Hybrids

1. *One hybrid per honest commitment:* Any time an honest party issues a commitment and doesn't transmit the associated randomness to a corrupt party, we replace the commitment with a commitment to 0. By reduction to the hiding property of the of the commitment scheme, each of these hybrids is indistinguishable from the last.
2. *One additional hybrid per evaluation operation between honest parties:* Any time an honest party transmits a ciphertext to another honest party, we replace it with an encryption of 0. By reduction to the IND-CPA-security of the encryption scheme, each of these hybrids is indistinguishable from the last.

   Notice that the final hybrid of this form does not use any value outside of the adversary's view, except for the case that an honest party evaluates towards a corrupt one, in which case the honest party's inputs must be known.
3. *One hybrid for the corrupt commitments*: In the next hybrid, the experiment checks that the corrupt parties *actually* use the same committed value whenever they are expected to prove knowledge of the contents of a commitment. By reduction to the binding property of the of the commitment scheme, this hybrid is indistinguishable from the previous one.

Commit-and-Privately-Evaluate Hybrids

2. *One additional hybrid per evaluation operation between honest parties:* Any time an honest party transmits a ciphertext to another honest party, we replace it with an encryption of 0. By reduction to the IND-CPA-security of the encryption scheme, each of these hybrids is indistinguishable from the last.

   Notice that the final hybrid of this form does not use any value outside of the adversary's view, except for the case that an honest party evaluates towards a corrupt one, in which case the honest party's inputs must be known.

3. *One hybrid for the corrupt commitments*: In the next hybrid, the experiment checks that the corrupt parties *actually* use the same committed value whenever they are expected to prove knowledge of the contents of a commitment. By reduction to the binding property of the of the commitment scheme, this hybrid is indistinguishable from the previous one.

4. *Finally, the ideal experiment:* Relative to the previous hybrid, $\mathcal{F}_{\mathsf{CPE}}$ now computes the outputs of the honest parties, and $\mathcal{S}$ extracts all of the inputs of corrupt parties perfectly via $\mathcal{F}_{\mathsf{BCZK}}$. $\mathcal{F}_{\mathsf{CPE}}$ also computes the outputs of all honest evaluations toward corrupt parties (using the same deterministic C as honest parties would); $\mathcal{S}$ merely encrypts and commits to those outputs, and delivers them. Apart from this, the same constraints are enforced in the two experiments; their distributions are identical. ∎

Next Time...


Broadcast
Encryption
Coin-Tossing
Commitments
Oblivious Transfer
Protocol Compilers
Zero-Knowledge Proofs

CS4501 Cryptographic Protocols Lecture 26: Schnorr Protocols, Coin Tossing Part 2, the GMW Compiler

https://jackdoerner.net/teaching/#2026/Spring/CS4501