Universal Composability
alias
Reactive Simulatability

Recap: secure MPC

We have seen:

- 2-party, computational, semi-honest, constant-round.
- 2- or n-party, computational, semi-honest($< n$), linear-round.
- n-party, unconditional, semi-honest($< n/2$), linear-round.
- n-party, computational, malicious($< n/2$), constant-round.
- n-party, unconditional, malicious($< n/3$), linear-round.
  - Possible to have less than $n/2$ malicious parties, using ZK-techniques to convince other parties that you behave as prescribed.
  - Has exponentially small probability of failure.

What we have not seen

- Secure MPC with malicious majority ($\geq n/2$ malicious parties)
  - Possible only in the computational setting
  - In the beginning, commit to your randomness. During computation, prove (in ZK) that you are using the committed randomness.
  - Malicious parties can interrupt the protocol.
- Asynchronous MPC
  - All messages arbitrarily delayed, but eventually delivered.
    - The delays are not controlled by the adversary.
  - No difference in semi-honest case.
  - With fail-stop adversary need $< n/3$ corrupted parties.
  - With malicious adversary need $< n/4$ corrupted parties.
    - ...with unconditional security.

On security definitions

- Real vs. ideal functionality...
- The ideal functionality for computing the function f with n inputs and outputs:
  - Parties $P_1, \ldots, P_n$ hand their inputs $x_1, \ldots, x_n$ over to the functionality.
  - The ideal functionality computes $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$.
    - ...tossing coins if f is randomized.
  - The ideal functionality sends y_i to P_i.

Ideal functionality $MPC_n^{\textsf{Ideal}}$

- Has n input ports and n output ports.
- Initial state: $x_1, \ldots, x_n$ are undefined.
- On input (input, v) from port $in_i?$:
  - If x_i is defined, then do nothing.
  - If x_i is not defined, then set $x_i := v$.
- If $x_1, \ldots, x_n$ are all defined then compute $(y_1, \ldots, y_n)$.
- For all i, write y_i to port $out_i!$.

Ideal functionality MPC_n^{Ideal}

- Has n input ports and n output ports.
- Initial state: $x_1, \ldots, x_n$ are undefined.
- On input (input, v) from port $in_i?$:
  - If x_i is defined, then do nothing.
  - If x_i is not defined, then set $x_i := v$.
- If $x_1, \ldots, x_n$ are all defined then compute $(y_1, \ldots, y_n)$.
- For all i, write y_i to port $out_i!$.

How do we run it (connections, scheduling)? What it means for a party to be corrupted?

Real functionality $MPC_n^{\textsf{Real}}$

- Conceptually made up of n identical machines P_i.
  - Has ports $in_i?$, $out_i!$, network ports...
- Initialization: P_i learns his name i.
- On input (input, v) from port $in_i?$ put $x_i := v$ and start executing the MPC protocol...
- If the protocol has finished execution then write y_i to $out_i!$.

Real functionality $MPC_n^{\textsf{Real}}$

- Conceptually made up of n identical machines P_i.
  - Has ports $in_i?$, $out_i!$, network ports...
- Initialization: P_i learns his name i.
- On input (input, v) from port $in_i?$ put $x_i := v$ and start executing the MPC protocol...
- If the protocol has finished execution then write y_i to $out_i!$.

- Cannot speak about the indistinguishability of $MPC^{\textsf{Ideal}}$ and $MPC^{\textsf{Real}}$ because the set of ports is different.
  - We have to simulate something...

Probabilistic I/O automata

A PIOA M has

- The set of possible states Q^M;
- The initial state $q_0^M \in Q^M$ and final states $Q_F^M \subseteq Q^M$;
- The sets of ports:
  - input ports $\mathbf{IPorts}^M$,
  - output ports $\mathbf{OPorts}^M$,
  - clocking ports $\mathbf{CPorts}^M$;
- A probabilistic transition function δ^M:
  - domain: $Q^M \times \mathbf{IPorts}^M \times \{0,1\}^*$;
  - range: $Q^M \times (\mathbf{OPorts}^M \to (\{0,1\}^*)^*) \times (\mathbf{CPorts}^M \cup \{\perp\})$

  ... in our examples implemented by a PPT algorithm.

  - Q^M, Q_F^M and q_0^M may (uniformly) depend on the security parameter.

Channels and collections

- A set $\mathbf{Chans}$ of channel names is given.
- There is a distinguished $clk \in \mathbf{Chans}$, representing global clock.
- For a channel c, its input, output and clocking ports are $c?$, $c!$ and $c^{\triangleleft}!$.
- A closed collection C is a set of PIOAs, such that
  - no port is repeated;
  - For each $c \in \mathbf{Chans} \backslash \{clk\}$ occurring in C: the ports $c?$, $c!$ and $c^{\triangleleft}!$ are all present.
  - $clk?$ is present. $clk!$ and $clk^{\triangleleft}!$ are not present.
- A collection C is a set of PIOAs that can be extended to a closed collection.
  - Let freeports(C) be the set of ports that the machines in C' certainly must have for $C \cup C'$ to be a closed collection.

Internal state of a closed collection

The state of a closed collection C consists of

- the states of all PIOA-s in C;
  - Initially q_0^M for all $M \in C$.
- the message queues of all channels c in C;
  - I.e. sequences of (still undelivered) messages.
  - Initially the empty queues for all $c \in C$.
- the currently running PIOA M, its input message v and channel c.
  - Initially X, ε and clk, where X is the machine with the port $clk?$.

Execution step of a closed collection

- Invoke the transition function of M with message v on input port $c?$.
  - Update the internal state of M.
  - If $(v_1, \ldots, v_k)$ was written to port $c'!$ then append $v_1, \ldots, v_k$ to the end of the message queue of c'.
- If M is X and it reached the final state then stop the execution.
- Otherwise, if M picked a clock port $c'^{\lhd}!$ and the queue of c' is not empty, then define the new (M, v, c):
  - c is c';
  - v is the first message in the queue of c', which is removed from the queue;
  - M is the machine with the port $c'?$.
- Otherwise set $(M, v, c) := (X, \varepsilon, clk)$.

Trace of the execution

Each execution step adds a tuple consisting of

- the machine that made the step;
- the incoming message and the channel;
- the random coins that were generated and the new state and messages that were produced.

to the end of the trace so far.

The semantics of a closed collection is a probability distribution over traces (for a given security parameter).

Trace of the execution

Each execution step adds a tuple consisting of

- the machine that made the step;
- the incoming message and the channel;
- the random coins that were generated and the new state and messages that were produced.

to the end of the trace so far.

The semantics of a closed collection is a probability distribution over traces (for a given security parameter).

Given trace tr and a set of machines $\mathcal{M}$, the restriction of the trace $tr|_{\mathcal{M}}$ consists of only those tuples where the machine belongs to $\mathcal{M}$.

Combining PIOAs

The combination of PIOAs $M_1, \ldots, M_k$ is a PIOA M with

- the state space $Q^M = Q^{M_1} \times \cdots \times Q^{M_k}$;
- initial state $q_0^M = (q_0^{M_1}, \ldots, q^{M_k})$;
- final states $Q_F^M = \bigcup_i Q^{M_1} \times \cdots \times Q^{M_{i-1}} \times Q_F^{M_i} \times Q^{M_{i+1}} \times \cdots \times Q^{M_k}$;
- ports $\mathbf{XPorts}^M = \bigcup_i \mathbf{XPorts}^{M_i}$ with $\mathbf{X} \in \{\mathbf{I}, \mathbf{O}, \mathbf{C}\}$;
- Transition function δ^M, where $\delta^M((q_1, \ldots, q_k), c?, v)$ is evaluated by
  - Let i be such that $c? \in \mathbf{IPorts}^{M_i}$.
  - Evaluate $(q_i', f_i, p) \leftarrow \delta^{M_i}(q_i, c?, v)$.
  - Output $((q_1, \ldots, q_{i-1}, q_i', q_{i+1}, \ldots, q_k), f, p)$, where

$$f(c'!) = \begin{cases} f'(c'!), & \text{if } c'! \in \mathbf{OPorts}^{M_i} \\ \varepsilon, & \text{otherwise.} \end{cases}$$

Exercise. How does the semantics of a closed collection change if we replace certain machines in this collection with their combination?

Security-oriented structures

- A structure consists of
  - a collection C;
  - a set of ports $\mathsf{S} \subseteq \mathsf{freeports}(C)$.
    - C offers the intended service on S.
    - The ports $\mathsf{freeports}(C) \setminus \mathsf{S}$ are for the adversary.
- A system is a set of structures.
- A configuration consists of a structure (C, S) and two PIOA-s H and A, such that
  - H has no ports in $\mathsf{freeports}(C) \setminus \mathsf{S}$,
  - $C \cup \{H, A\}$ is a closed collection.
- Let $\mathbf{Confs}(C, \mathsf{S})$ be the set of pairs (H, A), such that (C, S, H, A) is a configuration.

Exercise. What parts of (C, S) determine $\mathbf{Confs}(C, \mathsf{S})$?

Reactive simulatability

- Let (C_1, S) and (C_0, S) be two structures.
- (C_1, S) is at least as secure as (C_0, S) if
  - for all H,
  - for all A, such that $(H, A) \in \mathbf{Confs}(C_1, \mathsf{S})$
  - exists S, such that $(H, S) \in \mathbf{Confs}(C_0, \mathsf{S})$

  such that $[\![C_1 \cup \{H, A\}]\!]|_H \approx [\![C_0 \cup \{H, S\}]\!]|_H$.
- We also say that (C_0, S) simulates (C_1, S).
- The simulatability is universal if the order of quantifiers is $\forall A \exists S \forall H$.
- The simulatability is black-box if
  - there exists a PIOA Sim, such that
  - for all $(H, A) \in \mathbf{Confs}(C_1, \mathsf{S})$ holds

$$(H, A \| Sim) \in \mathbf{Confs}(C_0, \mathsf{S}) \text{ and } [\![C_1 \cup \{H, A\}]\!]|_H \approx [\![C_0 \cup \{H, A, Sim\}]\!]|_H.$$

Exercise. Show that universal and black-box simulatability are equivalent (if the port names do not collide).

Simulatability for systems

- A system Sys_1 is at least as secure as a system Sys_0 if for all structures $(C_1, \mathsf{S}) \in Sys_1$ there exists a structure $(C_0, \mathsf{S}) \in Sys_0$, such that (C_1, S) is at least as secure as (C_0, S).

Example: secure channels for n parties

- Ideal PIOA $\mathcal{I}$ has ports $in_i?$ and $out_i!$ for communicating with the i-th party.
- Input (j, M) on $in_i?$ causes (i, M) to be written to $out_j!$.
- Should model API calls, hence it also has the ports $out_i{}^{\triangleleft}!$.

Example: secure channels for n parties

- Ideal PIOA $\mathcal{I}$ has ports $in_i?$ and $out_i!$ for communicating with the i-th party.
- Input (j, M) on $in_i?$ causes (i, M) to be written to $out_j!$.
- Should model API calls, hence it also has the ports $out_i{}^{\triangleleft}!$.
- Real structure uses public-key cryptography to provide confidentiality and authenticity.
  - Message M from i to j encoded as $\mathcal{E}_j(\mathsf{sig}_i(M))$.
- Consists of PIOA-s $M_1, \ldots, M_n$. M_i has ports $in_i?$ and $out_i!$.
- M_i has ports $net_i^{\rightarrow}!$, $net_i^{\rightarrow\triangleleft}!$ and $net_i^{\leftarrow}?$ for (insecure) networking.
- Public keys are distributed over authentic channels.
  - M_i has ports $aut_{i,j}^{\rightarrow}!$, $aut_{i,j}^{\mathrm{a}}!$ and $aut_{j,i}^{\mathrm{a}}?$ for authentically communicating with party M_j.
  - M_i always writes identical messages to $aut_{i,j}^{\rightarrow}!$ and $aut_{i,j}^{\mathrm{a}}!$.

Example: secure channels for n parties

- Ideal PIOA $\mathcal{I}$ has ports $in_i?$ and $out_i!$ for communicating with the i-th party.
- Input (j, M) on $in_i?$ causes (i, M) to be written to $out_j!$.
- Should model API calls, hence it also has the ports $out_i{}^{\triangleleft}!$.
- Real structure uses public-key cryptography to provide confidentiality and authenticity.
  - Message M from i to j encoded as $\mathcal{E}_j(\mathsf{sig}_i(M))$.
- Consists of PIOA-s $M_1, \ldots, M_n$. M_i has ports $in_i?$ and $out_i!$.
- M_i has ports $net_i^{\rightarrow}!$, $net_i^{\rightarrow\triangleleft}!$ and $net_i^{\leftarrow}?$ for (insecure) networking.
- Public keys are distributed over authentic channels.
  - M_i has ports $aut_{i,j}^{\rightarrow}!$, $aut_{i,j}^{\mathrm{a}}!$ and $aut_{j,i}^{\mathrm{a}}?$ for authentically communicating with party M_j.
  - M_i always writes identical messages to $aut_{i,j}^{\rightarrow}!$ and $aut_{i,j}^{\mathrm{a}}!$.
- $\mathsf{S} = \{in_1!, \ldots, in_n!, in_1{}^{\triangleleft}!, \ldots, in_n{}^{\triangleleft}!, out_1?, \ldots, out_n?\}$.

$\mathcal{I}$ is way too ideal

- Sending a message without initialization.
  - generating keys and distributing the public keys.
- Sending messages without delays. Guaranteed transmission.
- Traffic analysis.
- Concealing the length of messages.
- Transmitting only a number of messages polynomial to η.

$\mathcal{I}$ is way too ideal

- Sending a message without initialization.
    - generating keys and distributing the public keys.
- Sending messages without delays. Guaranteed transmission.
- Traffic analysis.
- Concealing the length of messages.
- Transmitting only a number of messages polynomial to η.

To simplify the presentation, we'll also

- Allow reordering and repetition of messages from one party to another.

The state of the PIOA $\mathcal{I}$

- Boolean $init_i$ — "has M_i generated the keys?"
- Boolean $init_{i,j}$ — "has M_j received the public keys of M_i?"
- Sequence of bit-strings $D_{i,j}$ — the messages party i has sent to party j.
- ℓ_i — the total length of messages party i has sent so far.

Initial values — `false`, ε, or 0.

The state of the PIOA $\mathcal{I}$

- Boolean $init_i$ — "has M_i generated the keys?"
- Boolean $init_{i,j}$ — "has M_j received the public keys of M_i?"
- Sequence of bit-strings $D_{i,j}$ — the messages party i has sent to party j.
- ℓ_i — the total length of messages party i has sent so far.

Initial values — `false`, ε, or 0.

To set these values, $\mathcal{I}$ has to communicate with the adversary, too. It has the ports $adv^{\rightarrow}!$, $adv^{\rightarrow\triangleleft}!$ and $adv^{\leftarrow}?$ for that.

The transition function $\delta^{\mathcal{I}}$

- On input (init) from $in_i?$: Set $init_i$ to `true`, write (init, i) to $adv^{\rightarrow}!$ and raise $adv^{\rightarrow\triangleleft}!$.
- On input (init, i, j) from $adv^{\leftarrow}?$: Set $init_{i,j}$ to $init_i$.
- On input (send, j, M) from $in_i?$: Do nothing if one of the following holds:
  - $|M| + \ell_i > p(\eta)$ for a fixed polynomial p;
  - $init_i \wedge init_{j,i} =$ `false`.

  Otherwise add $|M|$ to ℓ_i and append M to $D_{i,j}$. Write $(\text{sent}, i, j, |M|)$ to $adv^{\rightarrow}!$ and raise $adv^{\rightarrow\triangleleft}!$.
- On input (recv, i, j, x) from $adv^{\leftarrow}?$: Do nothing if one of the following holds:
  - $init_j \wedge init_{i,j} =$ `false`;
  - $x \leq 0$ or $|D_{i,j}| < x$.

  Otherwise write $(\text{received}, i, D_{i,j}[x])$ to $out_j!$ and raise $out_j^{\triangleleft}!$.

The state of the PIOA M_i

- The decryption key K_i^{d} and signing key K_i^{s}.
- The encryption keys K_j^{e} and verification keys K_j^{v} of all parties j.
- The length ℓ_i of the messages sent so far.

To operate, we have to fix

- IND-CCA-secure public key encryption system;
- EF-CMA-secure signature scheme.

The transition function δ^{M_i}

- On input (init) from $in_i?$: Generate keys $(K_i^{\mathrm{e}}, K_i^{\mathrm{d}})$ and $(K_i^{\mathrm{v}}, K_i^{\mathrm{s}})$. Ignore further (init)-requests. Write $(K_i^{\mathrm{e}}, K_i^{\mathrm{v}})$ to ports $aut_{i,j}^{\rightarrow}!$ and $aut_{i,j}^{\mathrm{a}}!$.
- On input $(k^{\mathrm{e}}, k^{\mathrm{v}})$ from $aut_{j,i}^{\mathrm{a}}?$: Initialize K_j^{e} and K_j^{v}.
- On input (send, j, M) from $in_i?$: If $|M| + \ell_i \leq p(\eta)$ and $K_i^{\mathrm{s}}, K_j^{\mathrm{e}}$ are defined
  - Let $v \leftarrow \mathcal{E}_{K_j^{\mathrm{e}}}(\mathsf{sig}_{K_i^{\mathrm{s}}}(i, j, M))$.
  - Add $|M|$ to ℓ_i.
  - Write (sent, j, v) to $net_i^{\rightarrow}!$ and raise $net_i^{\rightarrow\triangleleft}!$.
- On input (recv, j, v) from $net_i^{\leftarrow}?$: If the necessary keys are initialized and decryption and verification succeed (giving message M) then write $(\mathsf{received}, j, M)$ to $out_i!$ and raise $out_i^{\triangleleft}!$.

The simulator

- The simulator translates between the ideal structure $\mathcal{I}$ and the "real" adversary.
- It has the following ports:
  - $adv^{\rightarrow}?$, $adv^{\leftarrow}!$, $adv^{\leftarrow\triangleleft}!$ for communicating with $\mathcal{I}$.
  - $net_i^{\rightarrow}!$, $net_i^{\rightarrow\triangleleft}!$, $net_i^{\leftarrow}?$, $aut_{i,j}^{\rightarrow}!$, $aut_{i,j}^{\mathrm{a}}!$, $aut_{j,i}^{\mathrm{a}}?$ for communicating with the "real" adversary.
    - Both ends of the channel $aut_{i,j}^{\mathrm{a}}$ are at Sim.
    - But the adversary schedules this channel.

Exercise. Construct the simulator.

Composition

Let the structures $(C_1, \mathsf{S}_1), \ldots, (C_k, \mathsf{S}_k)$ be given. We say that (C, S) is the composition of those structures if

- $C_1, \ldots, C_k$ are pairwise disjunct;
- the sets of ports of $C_1, \ldots, C_k$ are pairwise disjunct;
- $C = C_1 \cup \cdots \cup C_k$;
- $\mathsf{freeports}(C_i) \backslash \mathsf{S}_i \subseteq \mathsf{freeports}(C) \backslash S$ for all i.

Write $(C, \mathsf{S}) = (C_1, \mathsf{S}_1) \times \cdots \times (C_k, \mathsf{S}_k)$.

Composition

Let the structures $(C_1, \mathsf{S}_1), \ldots, (C_k, \mathsf{S}_k)$ be given. We say that (C, S) is the composition of those structures if

- $C_1, \ldots, C_k$ are pairwise disjunct;
- the sets of ports of $C_1, \ldots, C_k$ are pairwise disjunct;
- $C = C_1 \cup \cdots \cup C_k$;
- $\mathsf{freeports}(C_i) \backslash \mathsf{S}_i \subseteq \mathsf{freeports}(C) \backslash S$ for all i.

Write $(C, \mathsf{S}) = (C_1, \mathsf{S}_1) \times \cdots \times (C_k, \mathsf{S}_k)$.

Theorem. Let

- $(C, \mathsf{S}) = (C_1, \mathsf{S}_1) \times (C_0, \mathsf{S}_0)$ and $(C', \mathsf{S}) = (C_1, \mathsf{S}_1) \times (C'_0, \mathsf{S}_0)$;
- $(C_0, \mathsf{S}_0) \geq (C'_0, \mathsf{S}'_0)$.

Then $(C, \mathsf{S}) \geq (C', \mathsf{S})$.

Proof on the blackboard.

Simulation for secure messaging

1. Separate encryption; replace it with an ideal encryption machine.
2. Define a probabilistic bisimulation with error sets between the states of $M_1 \| \cdots \| M_n$ and $\mathcal{I} \| Sim$.
3. Show that error sets have negligible probability.

- The errors correspond to forging a signature or generating the same random value twice.
- The first case may also be handled by defining a separate signature machine.
- The second case may also be handled by defining the ideal machines in the appropriate way.

The PIOA $\mathcal{E}nc^n$

- Has ports $ein_i?$, $eout_i!$, $eout_i{}^{\triangleleft}!$ for $1 \leq i \leq n$.
- The machine M_i will get ports $ein_i!$, $ein_i{}^{\triangleleft}!$, $eout_i?$.
- On input (gen) from $ein_i?$: generate a new keypair (k^+, k^-), store (i, k^+, k^-), write k^+ to $eout_i!$, clock.
- On input (enc, k^+, M) from $ein_i?$: if k^+ has been stored as a public key, then compute $v \leftarrow \mathcal{E}(k^+, M)$, write v to $eout_i!$, clock.
- On input (dec, k^+, M) from $ein_i?$: if (i, k^+, k^-) has been stored, write $\mathcal{D}(k^-, M)$ to $eout_i!$, clock.

The PIOA $\mathcal{E}nc_{\mathrm{s}}^{n}$

- Has ports $ein_i?$, $eout_i!$, $eout_i{}^{\triangleleft}!$ for $1 \leq i \leq n$.
- The machine M_i will get ports $ein_i!$, $ein_i{}^{\triangleleft}!$, $eout_i?$.
- On input (gen) from $ein_i?$: generate a new keypair (k^+, k^-), store (i, k^+, k^-), write k^+ to $eout_i!$, clock.
- On input (enc, k^+, M) from $ein_i?$: if k^+ has been stored as a public key, then compute $v \leftarrow \mathcal{E}(k^+, 0^{|M|})$, store (k^+, M, v), write v to $eout_i!$, clock.
    - Recompute v until it differs from all previous v-s.
- On input (dec, k^+, M) from $ein_i?$: if (i, k^+, k^-) has been stored, then
    - if (k^+, M, v) has been stored for some v, then write v to $eout_i!$, clock.
    - otherwise write $\mathcal{D}(k^-, M)$ to $eout_i!$, clock.

$\mathcal{E}nc^n \geq \mathcal{E}nc_{\mathrm{s}}^{n}$ (black-box). **Exercise.** Describe the simulator.

The PIOA $\mathcal{S}ig^n$

- Has ports $sin_i?$, $sout_i!$, $sout_i^{\triangleleft}!$ for $1 \leq i \leq n$.
- The machine M_i will get necessary ports for using $\mathcal{S}ig^n$ as by API calls.
- On input (gen) from $sin_i?$: generate a new keypair (k^+, k^-), store (i, k^+, k^-), write k^+ to $sout_i!$, clock.
- On input (sig, k^+, M) from $sin_i?$: if (i, k^+, k^-) has been stored then compute $v \leftarrow \mathsf{sig}(k^-, M)$, write v to $sout_i!$, clock.
- On input (ver, k^+, s) from $sin_i?$: if k^+ has been stored then write $\mathsf{ver}(k^+, s)$ to $sout_i!$, clock.

The PIOA $\mathcal{S}ig_{\mathrm{s}}^{n}$

- Has ports $sin_i?$, $sout_i!$, $sout_i{}^{\triangleleft}!$ for $1 \leq i \leq n$.
- The machine M_i will get necessary ports for using $\mathcal{S}ig^{n}$ as by API calls.
- On input (gen) from $sin_i?$: generate a new keypair (k^+, k^-), store (i, k^+, k^-), write k^+ to $sout_i!$, clock.
- On input (sig, k^+, M) from $sin_i?$: if (i, k^+, k^-) has been stored then compute $v \leftarrow \mathsf{sig}(k^-, M)$, store (k^+, M), write v to $sout_i!$, clock.
- On input (ver, k^+, s) from $sin_i?$: if k^+ has been stored then write $\mathsf{ver}(k^+, s) \wedge$ "(k^+, M) has been stored" to $sout_i!$, clock.

Modified real structure

- Instead of generating the encryption keys, and encrypting and decrypting themselves, machines M_i query the machine $\mathcal{E}nc^n$.
- We can then replace $\mathcal{E}nc^n$ with $\mathcal{E}nc^n_{\mathrm{s}}$. The original structure was at least as secure as the modified structure.
- Same for signatures. . .
- Denote the modified machines by $\tilde{M}_i$.

The state of the real structure

- State of $\tilde{M}_i$ — the keys K_j^{e} and K_j^{v} $(1 \leq j \leq n)$.
  - If some K is defined at several machines, then they are equal.
- State of $\mathcal{E}nc_{\mathrm{s}}^n$:
  - key triples (i, k^+, k^-), where k^+ is the same as K_i^{e}.
  - text triples (k^+, M, v), where k^+ also occurs in a key triple.
- State of $\mathcal{S}ig_{\mathrm{s}}^n$:
  - key triples (i, k^+, k^-), where k^+ is the same as K_i^{v}.
  - text pairs (k^+, M), where k^+ also occurs in a key triple.
- Possibly (during initialization) the keys in the buffers of the channels $aut_{i,j}^{\mathrm{a}}$.
- No messages are in the buffers of newly introduced channels ein_i etc.
- The buffers of channels connected to H or A, are not part of the state.

The simulator Sim

- Consists of the real structure and one extra machine $Cntr$. Its initial state contains counters z_{ij} for all $1 \leq i, j \leq n$.
- The ports $in_i?$, $out_i!$, $out_i{}^{\triangleleft}!$ of $\tilde{M}_i$ are renamed to $cin_i?$, $cout_i!$, $cout_i{}^{\triangleleft}!$.
- Machine $Cntr$ has ports $cin_i!$, $cin_i{}^{\triangleleft}!$, $cout_i?$, $adv^{\leftarrow}!$, $adv^{\leftarrow\triangleleft}!$, $adv^{\rightarrow}?$.
- On input (init, i) from $adv^{\rightarrow}?$ write (init) to $cin_i!$ and clock it.
- On input $(k^{\text{e}}, k^{\text{v}})$ from $aut^{\text{a}}_{j,i}?$: the machine $\tilde{M}_i$ additionally writes $(\text{recvkeys}, j)$ to $cout_i!$ and clocks it.
- Receiving $(\text{recvkeys}, j)$ from $cout_i?$, machine $Cntr$ writes (init, j, i) to $adv^{\leftarrow}!$ and clocks it.
- Receiving (send, i, j, l) from $adv^{\rightarrow}?$, the machine $Cntr$ generates a new* message M of length l, increments z_{ij}, stores (i, j, z_{ij}, M), writes (send, j, M) to $cin_i!$, clocks it.
- Reciving $(\text{received}, i, M)$ from $cout_j?$, the machine $Cntr$ locates the tuple (i, j, x, M), writes (recv, i, j, x) to $adv^{\leftarrow}!$, clocks it.

The state of $\mathcal{I}\|Sim$

- Same as real structure.
- For each i, j, the sequences $D'_{i,j}$ of messages (z, M) that the machine $Cntr$ has generated.
- The counters z_{ij}.
- Initialization bits $init_i$, $init_{i,j}$.
- The sequences of messages $D_{i,j}$ that party i has sent to party j.

The state of $\mathcal{J}\|Sim$

- Same as real structure.
- For each i, j, the sequences $D'_{i,j}$ of messages (z, M) that the machine $Cntr$ has generated.
- The counters z_{ij}.
- Initialization bits $init_i$, $init_{i,j}$.
- The sequences of messages $D_{i,j}$ that party i has sent to party j.

Lemma. If $\mathcal{J}\|Sim$ is not currently running, then

- $z_{ij} = |D_{ij}| = |D'_{i,j}|$ and the lengths of the messages in the sequences $D_{i,j}$ and $|D'_{i,j}|$ are pairwise equal.
- If $init_i$ then $\tilde{M}_i$ has requested the generation of keys. If $init_{i,j}$ then $\tilde{M}_j$ has received the keys of $\tilde{M}_i$. The opposite also holds.
- The signed messages in $\mathcal{S}ig^n_{\mathrm{s}}$ are exactly of the form (i, j, M) where M is in the sequence $D'_{i,j}$. The encrypted messages in $\mathcal{E}nc^n_{\mathrm{s}}$ are exactly those signed messages.

Probabilistic bisimulations

- Let $(S, A, \rightarrow, s_0)$ be a probabilistic transition system. I.e.
  - S and A are the sets of states and transitions. $s_0 \in S$.
  - $\rightarrow$ is a partial function from $S \times A$ to $\mathcal{D}(S)$ (probability distributions over S).
- An equivalence relation $\mathcal{R}$ over S is a probabilistic bisimulation if $s \mathcal{R} s'$ implies
  - for each $a \in A$, $s \xrightarrow{a} D$ implies that there exists D', such that $s' \xrightarrow{a} D'$, and
  - for each $t \in S$: $\sum_{t' \in t/\mathcal{R}} D(t') = \sum_{t' \in t/\mathcal{R}} D'(t')$.
- Two probabilistic transition systems $(S, A, \rightarrow, s_0)$ and $(T, A, \Rightarrow, t_0)$ are bisimilar if there exists a probabilistic bisimulation $\mathcal{R}$ of $(S \dot{\cup} T, A, \rightarrow \cup \Rightarrow)$ that relates s_0 and t_0.

Probabilistic bisimilarity

Bisimilarity of systems $(S, A, \rightarrow, s_0)$ and $(T, A, \Rightarrow, t_0)$ means that

- The sets S and T can be partitioned into $S_1 \dot\cup \cdots \dot\cup S_k$ and $T_1 \dot\cup \cdots \dot\cup T_k$, such that
  - ...also define $S_0 = T_0 = \emptyset$
- there exists a permutation σ of $\{0, \ldots, k\}$, such that
  - in other words, σ defines a relation $\mathcal{R} \subseteq S \times T$, such that $s \mathcal{R} t$ iff $s \in S_i, t \in T_{\sigma(i)}$ for some i.
- For all $s \in S_i$, $t \in T_{\sigma(i)}$, $a \in A$:
- If $s \xrightarrow{a} D$ then $t \overset{a}{\Rightarrow} E$. Also, for each j: $\sum_{s' \in S_j} D(s') = \sum_{t' \in T_j} E(t')$.
- $s_0 \mathcal{R} t_0$.

Bisimilarity for secure channels

Relating the states of real and (ideal‖simulator) structures:

- The states of $\tilde{M}_i$, $\mathcal{E}nc^n_{\mathrm{s}}$, $\mathcal{S}ig^n_{\mathrm{s}}$ must be equal.
- The rest of the state of $\mathcal{I}\|Sim$ must satisfy the lemma we had above.

The relationship must hold only if either H or A is currently running.

- Now consider all possible inputs that the real structure or (ideal‖simulator) may receive. Show that they react to it in the identical manner.

Home exercise

Present a simulatable functionality for secure channels (not allowing corruptions) that preserves the order of messages and does not allow their duplication.

Can raise the exam result by up to 10%.

Deadline: January 5th.

An UC voting functionality

Let there be m voters and n talliers. Let the possible votes be in $\{0, \ldots, L-1\}$.
All voters will give their votes. All authorities agree on the result. The adversary will not learn individual votes.

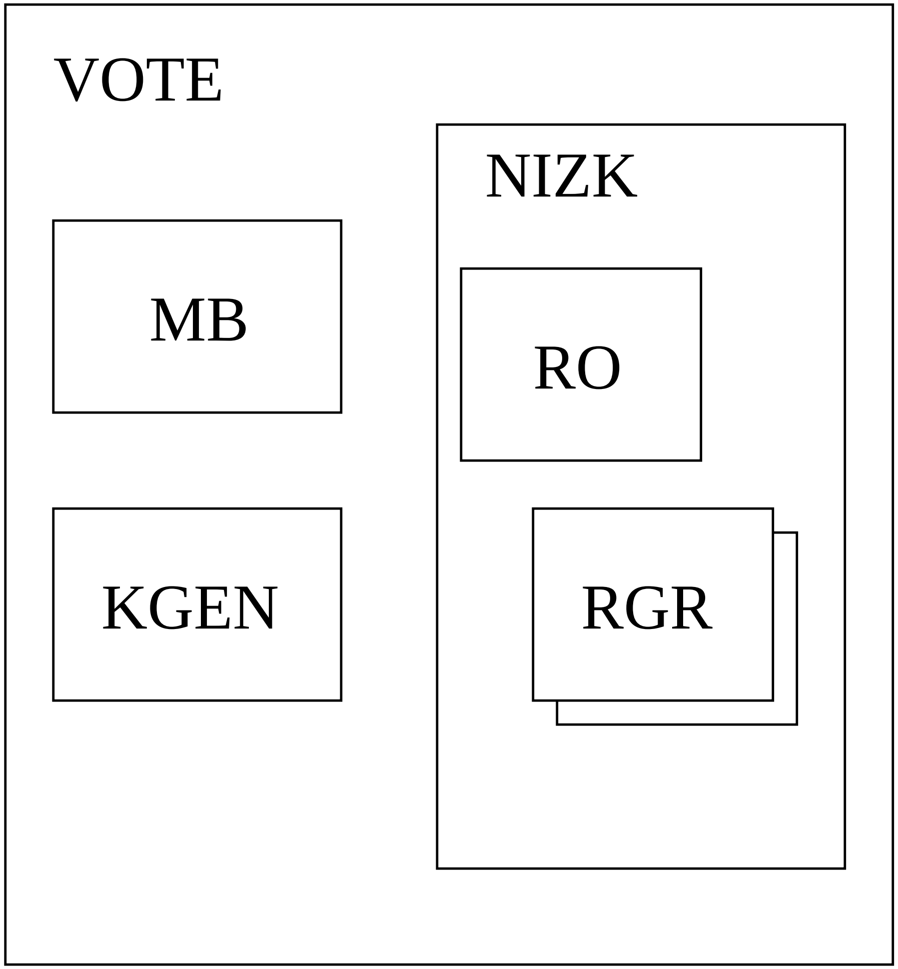
An UC voting functionality

Let there be m voters and n talliers. Let the possible votes be in $\{0, \ldots, L-1\}$.
All voters will give their votes. All authorities agree on the result. The adversary will not learn individual votes.

- The ideal functionality $\mathfrak{I}_{\text{VOTE}}$ has the standard ports… $in_i^V?$, $out_i^V!$, $out_i^{V\triangleleft}!$, $in_i^T?$, $out_i^T!$, $out_i^{T\triangleleft}!$, $adv^{\leftarrow}?$, $adv^{\rightarrow}!$, $adv^{\rightarrow\triangleleft}!$.
- First expect (init, sid)-command from the adversary.
- On input (vote, sid, v) from V_i store $(\text{vote}, sid, V_i, v, 0)$, send (vote, sid, V_i) to the adversary, ignore further votes from V_i in session sid.
- On input $(\text{accept}, sid, V_i)$ from the adversary, change the flag from 0 to 1 in $(\text{vote}, sid, V_i, v, \_)$.
- On input (result, sid) from the adversary, add up the votes in session sid with flag 1, store (result, sid, r) and send it to the adversary.
- On input $(\text{giveresult}, sid, i)$ from the adversary send (result, sid, r) to voter V_i or tallier T_{i-m}.

Building blocks

- Message board
  - Synchronous communication
- Homomorphic threshold encryption
  - MPC (for key generation)
- NIZK proofs
  - Random oracle
  - Generation of random elements of a group



Message board

Ideal functionality $\mathcal{I}_{\text{MB}}$ for parties $P_1, \ldots, P_n$ is the following:

- On input (bcast, sid, v) from P_i, store $(\text{bcast}, i, sid, v)$. Accept no further $(\text{bcast}, sid, \ldots)$-queries from P_i. Send $(\text{bcast}, sid, i, v)$ to the adversary.
- On input (pass, sid, i) from the adversary, if $(\text{bcast}, i, sid, v)$ has been stored, store (post, sid, i, v).
- On input (tally, sid) from the adversary, accept no more $(\text{bcast}, sid, \ldots)$ and $(\text{pass}, sid, \ldots)$-requests.
- On input $(\text{request}, sid, i)$ from P_j, if (tally, sid) has been received before, send all stored $(\text{post}, sid, \ldots)$-tuples to P_j (as a single message).

Realization requires reliable channels or smth.

ZK proofs

The ideal functionality $\mathcal{I}_{\text{ZK}}$ for parties $P_1, \ldots, P_n$ and witnessing relation $\mathcal{R}$ is the following

- On input $(\text{prove}, sid, P_j, x, w)$ from a party P_i:
  - Check that $(x, w) \in \mathcal{R}$;
  - Store (P_i, P_j, sid, x);
  - Send $(\text{prove}, P_i, P_j, sid, x)$ to the adversary.
  - Accept no more $(\text{prove}, sid, \ldots)$ queries from P_i.
- On input $(\text{proofok}, P_i, P_j, sid, x)$ from the adversary send $(\text{proof}, sid, P_i, x)$ to P_j.

NIZK proofs

The ideal functionality $\mathcal{F}_{\text{NIZK}}$ for parties $P_1, \ldots, P_n$ and witnessing relation $\mathcal{R}$ is the following

- On input $(\text{prove}, sid, x, w)$ from a party P_i:
  - Check that $(x, w) \in \mathcal{R}$;
  - Send (proof, sid, x) to the adversary.
  - Accept no more $(\text{prove}, sid, \ldots)$ queries from P_i.
  - Wait for a query of the form $(\text{proof}, sid, x, \pi)$ from the adversary.
  - Store (sid, x, π).
  - Send $(\text{proof}, sid, x, \pi)$ to P_i.
- On input $(\text{prove}, sid, x, w, \pi)$ from the adversary:
  - Check that $(x, w) \in \mathcal{R}$;
  - Store (sid, x, π).

NIZK proofs

- On input (verify, sid, x, π) from P_j check whether (sid, x, π) is stored. If it is then
  - Return (verifyok, sid, x).

  If it is not then
  - Send (witness?, sid, x) to the adversary.
  - Wait for a query of the form (prove, sid, x, w, π) from the adversary.
  - Handle (prove, sid, x, w, π) as on previous slide.
  - If $(x, w) \in \mathfrak{R}$ then return (verifyok, sid, x) to P_j.

Waiting for some message

- Before stopping, record the form of the expected message and the point where the execution was interrupted.
- At the next invocation check whether the expected message was received.
  - If yes, then continue from where we left off.
  - If no, then handle the received message normally.

  In both cases, clear the waiting state.

Random oracles

The random oracle functionality $\mathcal{I}_{\mathrm{RO}}$ for n parties is the following:

- On input x by any party or the adversary
  - If (x, r) is already stored for some r, return r.
  - Otherwise generate $r \in_R \{0,1\}^{p(\eta)}$, store (x, r) and return r.

$\mathcal{I}_{\mathrm{RO}}$ works as a subroutine.

Generating a random element of a group

Let G be a fixed group (depends on η only), with a prime cardinality and hard DDH problem. The functionality $\mathfrak{I}_{\text{RGR}}$ is the following:

- On input (init) by the adversary generates a random element of G and returns it to the adversary.
- On input (init, i) marks that it may answer to party P_i.
- On input (get) from a party returns the generated element, if allowed.

Realization:

- The machines M_i are initialized by the adversary.
- M_i generates a random element $g_i \in G$, secret shares it;
- The shared values are multiplied and the result is opened.
- A (get) by a party allows it to learn the computed value.
- Uses secure channels functionality.

Exercise. How to simulate?

Protocol realizing NIZK

- Idea: on input $(\text{prove}, sid, x, w)$ from party P_i the machine M_i commits to w and outputs x, $C(w)$, and a NIZK proof that $C(w)$ is hiding a witness for x.
- Initialization: parties get two random elements $g, h \in G$ using two copies of $\mathcal{F}_{\mathrm{RGR}}$.
  - Ignore user's query if (get) to $\mathcal{F}_{\mathrm{RGR}}$-s gets no response.
- Let us use the following commitment scheme (G is a group with cardinality $\#G$ and hard DDH problem):
  - To commit to $m \in G$, generate a random $r \in \{0, \ldots \#G - 1\}$. The commitment is $(g^r, m \cdot h^r)$.
  - The opening of the previous commitment is r.

Exercise. How to verify? What is this commitment scheme? What can be said about its security?

Protocol realizing NIZK

- There exists a ZK protocol for proving that a commitment c hides a witness w, such that $(x, w) \in \mathcal{R}$.
- For honest verifiers, this protocol has three rounds — commitment (or witness), challenge and response.
  - It depends on $\mathcal{R}$ (and the commitment scheme).
  - Let $A(x, C(w), w, r)$ generate the witness and $Z(x, C(w), w, r, a, c)$ compute the response.
  - Challenge is a random string. Let $\mathcal{V}(x, C(w), a, c, z)$ be the verification algorithm at the end.
- The whole proof π for (x, sid) consists of
  - $C(w)$, a random string $\bar{r}$;
  - $a \leftarrow A(x, C(w), w, r)$;
  - $z \leftarrow Z(x, C(w), w, r, a, H(x, a, sid, \bar{r}))$
- $(\text{proof}, sid, x, \pi)$ is sent back to the user.

Protocol realizing NIZK

- On input (verify, sid, x, π) from the user, machine M_j verifies that proof:
  - Computes $c = H(x, a, sid, \bar{r})$ (by invoking $\mathcal{F}_{\text{RO}}$) and verifies $\mathcal{V}(x, C, a, c, z)$.

  If correct, responds with (verifyok, sid, x).

Simulation

The simulator communicates with

- the ideal functionality: possible commands are
  - $(\mathsf{proof}, i, sid, x)$;
  - $(\mathsf{witness?}, sid, x, \pi)$.
- the real adversary: possible commands are
  - (init) and (init, i) for two copies of $\mathcal{I}_{\mathrm{RGR}}$;
  - queries to the random oracle $\mathcal{I}_{\mathrm{RO}}$.
    - Answer the queries to $\mathcal{I}_{\mathrm{RO}}$ in the normal way.

Simulator: initialization

On the very first invocation:

- Generate random elements $g, h \in G$.

On (init) and (init, i) from the adversary for functionalities $\mathcal{I}_{\text{RGR}}$:

- Record that these commands have been received.

Simulating $(\mathsf{proof}, i, sid, x)$

- The query $(\mathsf{prove}, sid, x, w)$ was made by party P_i to $\mathcal{F}_{\mathrm{NIZK}}$.
- Where do we get w?

Simulating $(\mathsf{proof}, i, sid, x)$

- The query $(\mathsf{prove}, sid, x, w)$ was made by party P_i to $\mathcal{I}_{\text{NIZK}}$.
- Where do we get w? **We don't get it at all.**
- Let C be the commitment of a random element w';
- **Simulate** the ZK proof of $(x, w') \in \mathcal{R}$:
  - Let c be a random challenge.
  - Let (a, z) be suitable witness and response for showing that C is the commitment of a suitable witness of x in $\mathcal{R}$.
- Let $\bar{r}$ be a random string, such that $(x, a, sid, \bar{r})$ has not been a query to $\mathcal{I}_{\text{RO}}$.

Simulating $(\text{proof}, i, sid, x)$

- The query $(\text{prove}, sid, x, w)$ was made by party P_i to $\mathcal{I}_{\text{NIZK}}$.
- Where do we get w? **We don't get it at all.**
- Let C be the commitment of a random element w';
- **Simulate** the ZK proof of $(x, w') \in \mathcal{R}$:
  - Let c be a random challenge.
  - Let (a, z) be suitable witness and response for showing that C is the commitment of a suitable witness of x in $\mathcal{R}$.
- Let $\bar{r}$ be a random string, such that $(x, a, sid, \bar{r})$ has not been a query to $\mathcal{I}_{\text{RO}}$.
- **Define** $H(x, a, sid, \bar{r}) := c$. Let $\pi = (C, \bar{r}, a, z)$.
- Send $(\text{proof}, sid, x, i, \pi)$ to $\mathcal{I}_{\text{NIZK}}$.

(*Programmable* random oracle)

Simulating (witness?, sid, x, π)

This is called if the real adversary has independently constructed a valid proof.

- Change the simulator as follows:
  - Initialization: the simulator generates g and h so, that it knows $\log_g h$.
- On a (witness?, ...)-query, the simulator checks whether the proof $\pi = (C, \bar{r}, a, z)$ is correct.
- If it is, then it extracts the witness w from C by ElGamal decryption.
- After that, it sends (prove, sid, x, w, π) to $\mathcal{I}_{\text{NIZK}}$.

Exercise. What if C does not contain a valid witness?

Corruptions

- The real adversary may send (corrupt)-command to some machine M_i.
  - Static corruptions — only at the beginning.
  - Adaptive corruptions — any time.
- The machine responds with its current state.
- Afterwards, M_i "becomes a part of" the adversary.
  - Forwards all received messages to the adversary.
  - M_i accesses other components on behalf of the adversary.
  - No more traffic between M_i and the user.
- Possibility to corrupt players has to be taken into account when specifying ideal functionalities.
  - The ideal adversary may send $(\text{corrupt}, i)$ to the functionality.
    - The simulator will make these queries if the real adversary corrupted someone.
  - The functionality may change the handling of the i-th party.

Corruptions and functionalities

- Random oracles — impossible to corrupt.
- Generating a random element of the group:
  - Implementations uses MPC techniques.
  - Tolerates adaptive corruptions of less than $n/3$ participants.
  - If party i is corrupted, then $\mathcal{I}_{\mathrm{RGR}}$
    - Gives no output to the i-th party.
    - Forwards to the adversary all requests from the i-th party.
  - If too many parties are corrupted (at least $n/3$) then $\mathcal{I}_{\mathrm{RGR}}$ gives all control to the adversary.
  - The simulator simply acts as a forwarder between a corrupted party and the adversary.

Corrupting $\mathcal{J}_{\mathrm{NIZK}}$

- The realization of NIZK uses $\mathcal{J}_{\mathrm{RGR}}$.
  - It fails if there are at least $n/3$ corrupt parties.
- It has no other weaknesses.

Corrupting $\mathcal{I}_{\text{NIZK}}$

- The realization of NIZK uses $\mathcal{I}_{\text{RGR}}$.
  - It fails if there are at least $n/3$ corrupt parties.
- It has no other weaknesses.
- If party i is corrupted in $\mathcal{I}_{\text{NIZK}}$ then it stops talking to the user.
  - The adversary may prove things on user's behalf.
- If at least $n/3$ parties are corrupted then $\mathcal{I}_{\text{NIZK}}$ gives up.

Corrupting $\mathcal{J}_{\text{NIZK}}$

- The realization of NIZK uses $\mathcal{J}_{\text{RGR}}$.
  - It fails if there are at least $n/3$ corrupt parties.
- It has no other weaknesses.
- If party i is corrupted in $\mathcal{J}_{\text{NIZK}}$ then it stops talking to the user.
  - The adversary may prove things on user's behalf.
- If at least $n/3$ parties are corrupted then $\mathcal{J}_{\text{NIZK}}$ gives up.
- The simulator corrupts i-th party of $\mathcal{J}_{\text{NIZK}}$ if M_i is corrupted or the i-th party in $\mathcal{J}_{\text{RGR}}$ is corrupted.

Exercise

How should corruptions be integrated to $\mathcal{I}_{\text{MB}}$?

Ideal functionality $\mathcal{I}_{\text{MB}}$ for parties $P_1, \ldots, P_n$ is the following:

- On input (bcast, sid, v) from P_i, store $(\text{bcast}, i, sid, v)$. Accept no further $(\text{bcast}, sid, \ldots)$-queries from P_i. Send $(\text{bcast}, sid, i, v)$ to the adversary.
- On input (pass, sid, i) from the adversary, if $(\text{bcast}, i, sid, v)$ has been stored, store (post, sid, i, v).
- On input (tally, sid) from the adversary, accept no more $(\text{bcast}, sid, \ldots)$ and $(\text{pass}, sid, \ldots)$-requests.
- On input $(\text{request}, sid, i)$ from P_j, if (tally, sid) has been received before, send all stored $(\text{post}, sid, \ldots)$-tuples to P_j (as a single message).

Homomorphic encryption

- A public-key encryption system $(\mathcal{K}, \mathcal{E}, \mathcal{D})$.
- The set of plaintexts is a ring.
- There is an operation $\oplus$ on ciphertexts, such that if $\mathcal{D}(k^-, c_1) = v_1$ and $\mathcal{D}(k^-, c_2) = v_2$ then $\mathcal{D}(k^-, c_1 \oplus c_2) = v_1 + v_2$.
- Security — IND-CPA.

Homomorphic encryption

- A public-key encryption system $(\mathcal{K}, \mathcal{E}, \mathcal{D})$.
- The set of plaintexts is a ring.
- There is an operation $\oplus$ on ciphertexts, such that if $\mathcal{D}(k^-, c_1) = v_1$ and $\mathcal{D}(k^-, c_2) = v_2$ then $\mathcal{D}(k^-, c_1 \oplus c_2) = v_1 + v_2$.
- Security — IND-CPA.
- In a threshold encryption system, the secret key is shared. There are shares $k_1^-, \ldots, k_n^-$.
- Also, there are public verification keys $k_1^{\mathrm{v}}, \ldots, k_n^{\mathrm{v}}$ that are used to verify that the authorities have correctly computed the shares of the plaintext.
  - ... like in verifiable secret sharing.
- We use secure MPC to generate $k^+, k_1^-, \ldots, k_n^-, k_1^{\mathrm{v}}, \ldots, k_n^{\mathrm{v}}$.
  - This can be modeled by an ideal functionality $\mathcal{I}_{\mathrm{KGEN}}$.
  - There are more efficient means of generation than general MPC.

Key generation

The ideal functionality $\mathcal{I}_{\text{KGEN}}$ for m users and n authorities works as follows:

- On input $(\text{generate}, sid)$ from the adversary, generates new keys. and gives the keys $k^+, k_1^{\text{v}}, \ldots, k_n^{\text{v}}$ to the adversary.
- On input $(\text{getkeys}, sid)$ from a party, gives the party this party's generated keys. (works like subroutine)
- Breaks down if there are at least $(m+n)/3$ corrupt parties.

Each voting session needs new keys, otherwise chosen-ciphertext attacks are possible.

Voting protocol

- Voter machines $M_1^V, \ldots, M_m^V$, tallier machines $M_1^T, \ldots, M_n^T$.
- The first time some M_i^V or M_i^T is activated, it asks for its key(s) from $\mathcal{I}_{\text{KGEN}}$ and receives them.
- On input (vote, sid, v) from the user the machine M_i^V
  - Let $c_i \leftarrow \mathcal{E}_{k+}(\text{Encode}(v))$. Make a NIZK proof π_i that c_i contains a correct vote. Send $(\text{bcast}, sid\|0, (c_i, \pi_i))$ to $\mathcal{I}_{\text{MB}}$.
- On input (count, sid) from the adversary the machine M_i^T
  - Sends $(\text{request}, sid\|0, i)$ to $\mathcal{I}_{\text{MB}}$ and receives all the votes and correctness proofs $(c_1, \pi_1), \ldots, (c_m, \pi_m)$.
  - Checks the validity of the proofs, using $\mathcal{I}_{\text{NIZK}}$.
  - Multiplies the valid votes and decrypts the result, using k_i^-. Let the result of the decryption be d_i. Makes a NIZK proof ξ_i that d_i is a valid decryption and sends $(\text{bcast}, sid\|1, (d_i, \xi_i)$ to $\mathcal{I}_{\text{MB}}$.
    - The proof also uses k_i^{v}.

Voting protocol

- On input (result, sid) from the adversary any machine
  - Sends $(\text{request}, sid\|0, i)$ to $\mathcal{I}_{\text{MB}}$ and receives all the votes and correctness proofs $(c_1, \pi_1), \ldots, (c_m, \pi_m)$.
  - Checks the validity of the proofs, using $\mathcal{I}_{\text{NIZK}}$.
  - Multiplies the valid votes, let the result be c.
  - Sends $(\text{request}, sid\|1, i)$ to $\mathcal{I}_{\text{MB}}$ and receives the shares of the result $d_1, \ldots, d_n$ together with proofs $\xi_1, \ldots, \xi_n$.
  - Check the validity of those proofs.
  - Combines a number of valid shares to form the final result r.
  - Sends (result, sid, r) to the user.

Exercise. What kind of corruptions are tolerated here?

The simulator — interface

The simulator encapsulates $\mathcal{J}_{\text{MB}}$, $\mathcal{J}_{\text{NIZK}}$, $\mathcal{J}_{\text{KGEN}}$.
The simulator handles the following commands:

- From $\mathcal{J}_{\text{VOTE}}$:
  - (vote, sid, i) — V_i has voted (but don't know, how).
  - (result, sid, r) — the result of the voting session sid.
- From the real adversary:
  - (count, sid) for M_i^T — produce the share of the voting result.
  - (result, sid) for any M — combine the shares of the result and send it to the user.
  - Corruptions; messages on behalf of corrupted parties.

The simulator — interface

- From the real adversary (on behalf of $\mathcal{F}_{\text{MB}}$):
  - (pass, sid, i) — lets the message sent by M_i to pass.
  - (tally, sid) — finishes round sid.
  - $(\text{bcast}, sid, i, v)$ — broadcast by a corrupt party.
- From the real adversary (on behalf of $\mathcal{F}_{\text{NIZK}}$):
  - $(\text{proof}, sid, x, \pi)$ — generate a proof token π for an honest prover.
  - $(\text{prove}, sid, x, w, \pi)$ — the adversary proves something himself.
- From the real adversary (on behalf of $\mathcal{F}_{\text{KGEN}}$):
  - $(\text{generate}, sid)$ — generates the keys.

The simulator — interface

The simulator issues the following commands:

To $\mathcal{I}_{\text{VOTE}}$:

(init, sid)
(accept, sid, i)
(result, sid)
(giveresult, sid, i)
(corrupt, i)
(vote, sid, i, v)

To the real adversary (as $\mathcal{I}_{\text{MB}}$):

(bcast, sid, i, v)

To the real adversary (as $\mathcal{I}_{\text{NIZK}}$):

(proof, i, sid, x)
(witness?, sid, x, π)

To the real adversary (as $\mathcal{I}_{\text{KGEN}}$):

(keys, $sid, k^+, k_1^{\text{v}}, \ldots, k_n^{\text{v}}$)

The simulator — initialization

- On the first activation with a new sid:
  - Generates keys $k^+, k_1^-, \ldots, k_n^-, k_1^{\mathrm{v}}, \ldots, k_n^{\mathrm{v}}$ for this session.
- When receiving $(\text{generate}, sid)$ from the adversary for $\mathcal{I}_{\mathrm{KGEN}}$,
  - marks that voting can now commence;
  - sends (init, sid) to $\mathcal{I}_{\mathrm{VOTE}}$.
- Corruptions by the adversary are forwarded to $\mathcal{I}_{\mathrm{VOTE}}$ and recorded.

The simulator — voting

- On input (vote, sid, i) from $\mathcal{I}_{\text{VOTE}}$:
    - Let the encrypted vote be $c \leftarrow \mathcal{E}_{k^+}(0)$.
    - Make a NIZK proof π that this vote is valid.
        - Going to $\mathcal{I}_{\text{NIZK}}$'s waiting state, as necessary.
    - Broadcast (using $\mathcal{I}_{\text{MB}}$) the pair (c, π) on behalf of voter i.
- On input (pass, sid, i), if the vote was broadcast for the voter P_i:
    - Send (accept, sid, i) back to $\mathcal{I}_{\text{VOTE}}$.
- If a corrupt party i puts a vote to the message board and makes a valid proof for it:
    - Decrypt that vote. Let its value be v.
    - Send (vote, sid, i, v) to $\mathcal{I}_{\text{VOTE}}$.

The simulator — tallying

On input $(\text{tally}, sid\|0)$ from the adversary for $\mathcal{I}_{\text{MB}}$:

- Close the voting session sid, accept counting queries.
- Send (result, sid) to $\mathcal{I}_{\text{VOTE}}$.
- Get the voting result r from $\mathcal{I}_{\text{VOTE}}$ and store it.

The simulator — counting

On input (count, sid) from the adversary for the tallier T_i:

- Check the proofs of all votes (c_i, π_i) using $\mathcal{F}_{\text{NIZK}}$.
  - Going to wait-state, if necessary.
- Let C be the product of all votes with valid proofs.
- For talliers $T_1, \ldots, T_n$, let $d_1, \ldots, d_n$ be
  - if T_i is corrupt, then $d_i = \mathcal{D}(k_i^-, C)$;
  - if T_i is honest, then a d_i is simulated value

  such that $d_1, \ldots, d_n$ combine to r.
  - $d_1, \ldots, d_n$ are generated at the first (count, sid)-query.
- Make a NIZK proof ξ_i for the share d_i.
- Broadcast (d_i, ξ_i) in session $sid\|1$ using $\mathcal{F}_{\text{MB}}$.
- A corrupt tallier can broadcast anything. But only (d_i, ξ_i) for the valid d_i is accepted at the next step.

The simulator — reporting the results

On input (result, sid) from the adversary for any voter or tallier i:

- Takes all votes (c_j, π_j) and all shares of the result (d_j, ξ_j).
- Verifies all correctness proofs of votes.
- Multiplies the valid votes.
- Verifies the correctness proofs of shares.
- If sufficiently many proofs are correct then sends $(\text{giveresult}, sid, i)$ to $\mathcal{I}_{\text{VOTE}}$.

Damgård-Jurik encryption system

- A homomorphic threshold encryption system
- Somewhat RSA-like
  - Operations are modulo n^s, where n is a RSA modulus.
  - Easy to recover i from $(1+n)^i \bmod n^s$.
- Maybe in the lecture. . .
- Otherwise see http://www.daimi.au.dk/˜ivan/GenPaillier_finaljour.ps

Secure MPC from thresh. homom. encr.

Computationally secure against malicious coalitions with size less than the threshold.

- Function given as a circuit with multiplications and additions.
- The value on each wire is represented as its encryption, known to all.

- Addition gate — everybody can add encrypted values by themselves.
- Multiplication of a and b (encryptions are $\overline{a}$ and $\overline{b}$):
  - Each party P_i chooses a random d_i, broadcasts $\overline{d_i}$, proves in ZK that it knows d_i.
  - Let $d = d_1 + \cdots + d_n$. Then $\overline{d} = \overline{d_1} \oplus \cdots \oplus \overline{d_n}$.
  - Decrypt $\overline{a} \oplus \overline{d} = \overline{a + d}$, let everybody know it.
  - Let $\overline{a_1} = \overline{a + d} \ominus \overline{d_1}$ and $\overline{a_i} = \ominus \overline{d_i}$. P_i knows a_i.
  - P_i broadcasts $a_i \odot \overline{b} = \overline{a_i b}$ and proves in ZK that he computed it correctly.
  - Everybody computes $\overline{a_1 b} \oplus \cdots \oplus \overline{a_n b} = \overline{ab}$.