Towards Modular Foundations for Protocol Security

Lúcás Críostóir Meier
lucas@cronokirby.com

February 7, 2023

Abstract

We do things with UC security.

1 Introduction

[Mei22]

Definition 1.1 (Adversaries). An adversary is a cool thing.

Theorem 1.1 (Cool Beans). Woah mama

And that's what matters.

■

Lemma 1.2. Woah mama again!

Corollary 1.3. Woah mama again!

Γ^0

$x \leftarrow 3$	
if $x + 2$:	
$\quad y \xleftarrow{\$} \mathbb{F}_q$	
$m \Rightarrow \langle \mathscr{P}_i, \mathscr{P}_j \rangle$	$y \leftarrow 4$
$m \Leftarrow \langle \text{OT}, \mathscr{P}_i \rangle$	$x \leftarrow 3$

$\underline{\text{Foo}(x, y)\text{:}}$
$\quad \text{Bar}(x, y)$

Game 1.1: Some Game

1.1 Relevance of Time Travel

stuff

IND-CCA
$x \leftarrow 4$

Protocol 1.2: Some Protocol

IND-CCA
$x \leftarrow 4$

Protocol 1.3: Some Protocol

IND-CCA
$x \leftarrow 3$

Functionality 1.4: Encryption

2 State-Separable Proofs

Our framework for describing protocols is based on *state-separable proofs* [BDF+18]. The security notions we develop for protocols ultimately find meaning in analogous notions of security for *packages*, the main object of study in state-seperable proofs.

This section is intended to be a suitable independent presentation of this formalism. In that spirit, we develop state-separable proofs "from scratch". Our starting point is merely that of computable randomized functions. This is in contrast to other protocol security frameworks like UC, whose foundational starting point is usually the more concrete notion of *interactive turing machines*.

We also take the opportunity to solidify the formalism of state-separable proofs, providing more complete definitions of various objects, completing several proofs left as mere sketches in the original paper, and proving a few additional properties we'll need later. This makes this section of interest to readers who are already familiar with state-separable proofs.

2.1 Some Notational Conventions

We write $[n]$ to denote the set $\{1, \ldots, n\}$.

We write `01` to denote the set $\{0, 1\}$, and write $\texttt{01}^*$ to denote binary strings. We write $\bullet$ to denote the empty string, which also serves as a "dummy" value in various contexts.

2.2 Probabilistic Functions

Our starting point is the notion of *randomized computable functions*. This is a notion we assume can be defined in a rigorous way, but whose concrete semantics we don't assign. We write $f : \texttt{01}^* \xrightarrow{\$} \texttt{01}^*$ to denote such a function (named f). Intuitively, this represents a function described by some algorithm, which takes in a binary string as an input, and produces a binary string as output, and is allowed to make randomized decisions to aid its computation.

We mainly consider *families* of functions, parametrized by a security parameter λ. Formally, this is in fact a function $f : \mathbb{N} \to \texttt{01}^* \xrightarrow{\$} \texttt{01}^*$, and we write $f_\lambda : \texttt{01}^* \xrightarrow{\$} \texttt{01}^*$ to denote a particular function in the family. In most cases, this security parameter is left *implicit*. In fact, all of the objects we consider from here on out will *implicitly* be *families* of objects, parametrized by a security parameter λ, and we will invoke this fact only as necessary.

Definition 2.1 (Efficient Functions). We assume that a function f has a runtime, denoted $T(f, x)$, measuring how long the function takes to execute on a given input $x \in \texttt{01}^*$.

We say that a function family is *efficient* if:

$$\forall \lambda.\ \forall x, |x| \in O(\text{poly}(\lambda)). \quad T(f_\lambda, x) \in O(\text{poly}(\lambda))$$

In other words, the runtime is always polynomial in λ, regardless of the input, or the random choices of the function.

□

Functions which are not necessarily efficient are said to be *unbounded*.

Considering efficient functions is essential, because the vast majority of cryptographic techniques depend on assuming that some problems are "hard" for adversaries with bounded computational resources, and so this notion of efficiency is critical to defining game-based security. Ironically, for protocol security, many protocols can be proved secure without this restriction.

Another crucial notion we need to develop is that of a *distance*, measuring how different two functions behave. This will underpin our later notion of security for games, which is based on saying that two different games are difficult to tell apart.

Definition 2.2 (Distance Function). Given a function $f : \bullet \xrightarrow{\$} \texttt{01}$, we assume that the probability $P[f \to 1]$ of the function returning 1 on the input $\bullet$ is well defined.

Given two functions f, g, we define their distance $\varepsilon(f, g)$ as:

$$\varepsilon(f, g) := |P[f \to 1] - P[g \to 1]|$$

□

In other words, the distance looks at how often one function returns 1 compared to the other. If the functions agree most of the time, then their distance will be small, whereas if they disagree very often, their distance will be large. This definition is actually quite natural. Since $P[f \to 1] = (1 - P[f \to 0])$, ε is actually just the total variation—or statistical—distance. This immediately implies that this distance has some nice properties, in particular that it forms a *metric*.

Lemma 2.1 (Distance is a Metric). ε is a valid metric, in particular, it holds for any functions f, g, h, that:

1. $\varepsilon(f, f) = 0$,
2. $\varepsilon(f, g) = \varepsilon(g, f)$,
3. $\varepsilon(f, h) \leq \varepsilon(f, g) + \varepsilon(g, h)$.

Proof:

1. Follows from the fact that $P[f \to 1] = P[f \to 1]$, so $\varepsilon(f, f) = 0$.

2. Follows from the fact that $|a - b| = |b - a|$.

3. Follows from the triangle inequality for $\mathbb{R}$ and the fact that:

$$|P[f \to 1] - P[h \to 1]| = |(P[f \to 1] - P[g \to 1]) + (P[g \to 1] - P[H \to 1])|$$

■

We actually skipped one property in our proof that ε is a valid metric, which requires that if $f \neq g$, then $\varepsilon(f, g) > 0$. This is because we haven't yet defined what equality should mean for functions. This metric property gives us a very natural definition though.

Definition 2.3 (Function Equality). Two functions, f and g, are *equal*, written $f = g$, when:

$$\varepsilon(f, g) = 0$$

□

It's easy to see that this is an equality relation, satisfying reflexivity, symmetry, and transitivity.

We can also generalize this to arbitrary functions, rather than just $f : \bullet \xrightarrow{\$} \texttt{01}$, by defining:

$$\varepsilon(f, g) := \sup_{x, y \in \texttt{01}^*} |P[f(x) \to y] - P[g(x) \to y]|$$

however, we will not really be needing this general definition, outside of a technical and very strong notion of equality for packages used in the following subsection.

While the functions we've considered so far only manipulate binary strings, it's useful to allow *typed* functions, with richer input and output types. This could be defined in several ways, but the end result means that a typed function $f : A \xrightarrow{\$} B$ can be interpreted as a function over binary strings, using a suitable encoding and decoding mechanism, as well as perhaps having a special output value that f can return if it fails to decode its input successfully.

Being able to quantify types is also useful for the formalism itself, and potentially even for some packages. This allows us to type functions such as:

$$\begin{aligned} &\text{id} : \forall s.\ s \to s \\ &\text{id} = x \mapsto x \end{aligned}$$

In this example, s is a quantified type variable. Formally, we can see id as a function parametrized by a type, with id_S being a concrete function, after having chosen this type.

2.3 Defining Packages

Our next goal is to define the central object of state-separable proofs: the *package*. Intuitively, a package has some kind of state, as well as functions which manipulate this state. You can interact with a package by calling the various output functions it provides. This makes packages a natural fit for security games. What distinguishes packages from games is that they can have *input* functions. A package can depend on another package, with each of its functions potentially using the functions provided by this other package. This modularity makes the common proof technique of "game-hopping" much more easily usable, and is the core strength of the state-separable proof formalism.

Before we get to packages, we first need to define a few convenient notions for functions manipulating a state, and parametrizing functions with other functions.

Our first definition will be a little bit of shorthand.

Definition 2.4 (Stateful Function). A *stateful* function is simply a function f of the form:

$$f : (S, \texttt{01}^*) \xrightarrow{\$} (S, \texttt{01}^*)$$

S represents the state being used and modified by the function. As a convenient shorthand, we write:

$$f : \circlearrowleft_S$$

□

It's useful to have a bit of typing to separate the state from the rest of the input and output, since it allows us to avoid defining inessential padding details inside the formalism itself.

We'll also want a notion of equality for these functions.

Definition 2.5 (Stateful Function Equality). Two stateful functions $f :\circlearrowleft_S$ and $f :\circlearrowleft_{S'}$ are equal, written $f = f'$, if there exists an isomorphism $\varphi : S \cong S'$, such that:

$$f = (s, i) \mapsto (s', o) \leftarrow f'(\varphi(s), i);\ (\varphi^{-1}(s'), o)$$

□

Basically, the states don't have to be literally the same, as long as they're isomorphic, and the natural way of making the two types match up produces equal functions. One can verify that this forms a valid equality relation. Note that this reduces to the standard notion of equality of functions by considering appropriate binary encodings of the two states.

We also need to consider functions parametrized by other functions. Intuitively, this arises when one function calls another. For example, consider:

$$f(x) := g(x) \oplus g(x)$$

which is well defined regardless of what g is. Here f is implicitly parametrized by g, but we could write this explicitly as $f(x) := g \mapsto g(x) \oplus g(x)$. We could write $f : (\texttt{01}^* \xrightarrow{\$} \texttt{01}^*) \to (\texttt{01}^* \xrightarrow{\$} \texttt{01}^*)$ as a potential type in this example. We write $f[g]$ for the instantiation of a parametrized function f with an input function g. It might also be the case that g is itself parametrized, in which case $f[g]$ is defined as:

$$f[g] := h \mapsto f[g[h]]$$

We can define a natural, albeit very strong, notion of equality for parametrized functions, saying that:

$$f = g \iff \forall h_1, \ldots, h_n.\ f[h_1, \ldots] = g[h_1, \ldots]$$

In other words, the two functions must be equal regardless of how we instantiate them.

We've now developed enough tools to define packages.

Definition 2.6 (Package). A package A consists of:

- a type S, for its state,
- a set of *input names* $\text{In}(A)$, of size m,

- a permutation $\pi_{\text{in}} : \text{In}(A) \leftrightarrow [m]$,
- a set of *output names* $\text{In}(A)$, of size n,
- a permutation $\pi_{\text{out}} : [m] \leftrightarrow \text{Out}(A)$,
- a set of parametrized functions $f_1, \ldots, f_n : \forall s.\ \circlearrowleft_s^m \to \circlearrowleft_{(S,s)}$, each of which has a distinct name $n_i \in \text{Out}(A)$.

We also only consider a package to be defined *up to* potentially renaming its input and output functions injectively.

■

The basic idea is that a package has an internal state S, which gets manipulated by each of the functions it exports. These functions, in turn, can depend on other input functions. If a stateful function f uses a stateful function g, then the result is a stateful function $f[g]$ manipulating *both* the state of f, and the state of g. Furthermore, f is defined in such way agnostic to what the state manipulated by g happens to be, which is why we use a *quantified* type instead, to allow instantiation with functions manipulating different kinds of state.

In practice, each function in a package is unlikely to use *all* of the input functions of the package, but it is much simpler to have each function parametrized by all the possible inputs, even if some are left unused. It's also much simpler to define an ordering of the input functions π, so be able to use $\circlearrowleft_s^m$ as the input type for the parametrized functions.

The semantics of a package without inputs are intuitively that of a stateful computer program or machine you can interact with. The machine has some kind of state, represented by S, along with various functions you can call, represented by $f_1, \ldots, f_n$. Each of these will use the input you provide, along with the current state of the machine, in order to supply you with an output, potentially modifying the state along the way. The input functions allow a package to interact with other packages itself.

We'll often use $\text{In}(A)$ or $\text{Out}(A)$ to talk about the input and output functions of a package. As a bit of a short hand notation, we write $\text{In}(A, B, \ldots)$ for the union $\text{In}(A) \cup \text{In}(B) \cup \ldots$, and similarly for $\text{Out}(\ldots)$.

We describe this kind of interaction using the formal notion of package *composition*.

Definition 2.7 (Package Composition). Given two packages A, B with $\text{In}(A) \subseteq \text{Out}(B)$, we define their composition $A \circ B$ as a package characterized by:

- a state type $(A.S, B.S)$,
- input names $\text{In}(B)$,

- output names $\text{Out}(A)$,
- $\pi_{\text{in}} := B.\pi_{\text{in}}$,
- $\pi_{\text{out}} := A.\pi_{\text{out}}$,
- output functions $A.f_1[\varphi(B.f_1), \ldots, \varphi(B.f_{B.n})], \ldots$.

In more detail, these functions have type $\forall s.\ \circlearrowleft_s^{B.m} \to \circlearrowleft_{((A.S, B.S), s)}$, and are of the form:

$$(h_1, \ldots, h_{B.m}) \mapsto A.f_i[\varphi_A(B.f_1)[h_1, \ldots], \ldots, \varphi_A(B.f_{B.n})[h_1, \ldots]]$$

where φ_A assigns each function $B.f_i$ to a slot in $[m]$ using $A.\pi_{\text{in}}$ on the name of that function, $B.n_i$. The same input functions h_j being given to all the functions used by $A.f_i$.

□

Package composition formally defines the intuitive notion of one package "using" the functions provided by another package. The result is a package providing the functions defined in A, and requiring the functions needed by B, but with the functions inside B itself now effectively inlined inside of $A \circ B$.

Next we'd like to prove that package composition satisfies some nice properties. For example $A \circ (B \circ C)$ is the same as $(A \circ B) \circ C$. There's one problem though, which is that we haven't defined what it means for two packages to be "the same".

Definition 2.8 (Literal Equality). We say that two packages A, B are *literally equal*, written $A \equiv B$, when:

- $A.S \cong B.S$,
- $\text{In}(A) = \text{In}(B)$,
- $\text{Out}(A) = \text{Out}(B)$,
- There exists a permutation $\pi : [n] \leftrightarrow [n]$ such that

$$\forall i \in [n].\ A.f_i = B.f_{\pi(i)} \land A.n_i = B.n_{\pi(i)}$$

□

We require strict equality for the input and output names, to avoid spurious comparisons between two packages with completely different names, although it should be noted that packages are only really defined up to renaming anyways, so this is essentially an isomorphism constraint. For the type of state, we consider an isomorphism directly, mainly so that $(A.S, (B.S, C.S))$ is considered to be the same state type as $((A.S, B.S), C.S)$, which might already be the case depending on how one defines equality for sets. The final condition also implies that π_{in} is the same for both packages.

This notion of equality is very strong, especially because of the equality it imposes on the functions defined in each package. While it suffices to explore basic properties of composition for packages, we'll want to abandon it quite quickly for a looser and more easily used notion of equality.

The first property we prove is the one used as an example above.

Lemma 2.2 (Associativity of Composition). Given packages A, B, C, it holds that:

$$A \circ (B \circ C) \equiv (A \circ B) \circ C$$

provided these expressions are well defined.

Proof: The input and output names are clearly equal on both sides. Furthermore, the state on the left is $(A.S, (B.S, C.S))$, $((A.S, B.S), C.S)$ on the right, and so the two states are isomorphic. All that's left is the final condition, talking about the equality of the functions defined in each package.

Now, for the equality of functions, we'll expand the functions of the package on the left, and then on the right, before comparing the results we get.

The functions in $B \circ C$ are of the form:

$$(h_1, \ldots) \mapsto B.f_i[\varphi_B(C.f_1)[h_1, \ldots], \ldots]$$

And then the functions in $A \circ (B \circ C)$ are of the form:

$$(h_1, \ldots) \mapsto A.f_i[\varphi_A(B.f_1)[\varphi_B(C.f_1)[h_1, \ldots]], \ldots]$$

From the other side, the functions in $A \circ B$ are of the form:

$$(h_1, \ldots) \mapsto A.f_i[\varphi_A(B.f_1)[h_1, \ldots], \ldots]$$

This makes the functions in $(A \circ B) \circ C$ of the form:

$$(h_1, \ldots) \mapsto A.f_i[\varphi_A(B.f_1)[\varphi_{A \circ B}(C.f_1)[h_1, \ldots]], \ldots]$$

The main difference is that we end up with $\varphi_{A \circ B}$ as our means of assigning the functions in C to the slots of B. However, φ_X only depends on $X.\pi_{\text{in}}$, and by definition $(A \circ B).\pi_{\text{in}} = B.\pi_{\text{in}}$, so $\varphi_{A \circ B} = \varphi_B$.

Another smaller difference is that the resulting stateful functions have different, but isomorphic states, which is allowed by stateful function equality.

So, in both cases, we end up with the same functions, concluding our proof.

■

This property is useful, since it lets us simply write $A \circ B \circ C$, without worrying about the order in which packages are composed.

Another more technical property we want composition to satisfy is that if *equality preservation*. If $B \equiv B'$, then it should be the case that $A \circ B \equiv A \circ B'$, or that $B \circ C \equiv B' \circ C$. If that weren't the case, then that would indicate that something is wrong with our definition of either equality or composition. The property we want for literal equality is that A and A' are completely interchangeable, and so once can always be replaced with the other, no matter the context, to the point that we can think of them as literally being the same package.

Thankfully, it turns out that composition and literal equality do in fact get along.

Lemma 2.3 (Composition Preserves Equality). Given any packages A, B, B', C it holds that:

- $B \equiv B' \implies A \circ B \equiv A \circ B'$,
- $B \equiv B' \implies B \circ C \equiv B' \circ C$,

provided these expressions are well defined.

Proof: In one case the state type is $(A.S, B.S)$ or $(A.S, B'.S)$, which are isomorphic if $B.S \cong B'.S$. Similarly, in the other case, we have $(B.S, C.S)$ vs $(B'.S, C.S)$, and the same observation holds.

Now, remember that $\text{In}(X \circ Y) = \text{In}(Y)$, and $\text{Out}(X \circ Y) = \text{Out}(X)$. Thus, since both $\text{In}(B) = \text{In}(B')$ and $\text{Out}(B) = \text{Out}(B')$ hold, we conclude that In and Out match up in both cases.

The trickier part is the 4th condition for equality.

In the first case, the functions are of the form:

$$A.\text{f}_i[\varphi_A(B.f_1), \ldots]$$

Now, φ_A orders the functions in B based only on their *names*. In particular, the ordering does not matter. Since the functions in B' are the same as B up to their ordering, including their names, φ_A will order them in the same way. Thus, the functions in $A \circ B$ and $A \circ B'$.

In the second case, the functions are of the form:

$$B.\text{f}_i[\varphi_B(C.f_1), \ldots]$$

Now, π_{in} is the same for both B and B', as we've remarked before. Thus, φ_B and $\varphi_{B'}$ are the same. Thus, the functions in $B \circ C$ are the same as $B \circ C'$, up to reordering, as required.

Having noted all of these points, we can conclude our proof.

■

Now, we look at the other kind of composition for packages: tensoring. The intuitive idea is that tensoring allows us to run two packages "in parallel". The result of tensoring two packages is a new package with the functions in both packages, and we can interact with one package or the other at will. We'll discuss the semantics a bit more after the formal definition.

Definition 2.9 (Package Tensoring). Given two packages A, B, with $\text{Out}(A) \cap \text{Out}(B) = \emptyset$, we can define their tensoring $A \otimes B$ as a package characterized by:

- a state type $(A.S, B.S)$,
- input names $\text{In}(A) \cup \text{In}(B)$,
- output names $\text{Out}(A) \cup \text{Out}(B)$,
- an output name assignment defined by:

$$\pi_{\text{out}}(i) := \begin{cases} A.\pi_{\text{out}}(i) & i \leq A.n \\ A.n + B.\pi_{\text{out}}(i - A.n) & i > A.n \end{cases}$$

- an input index assignment $\pi_{\text{in}}(n)$ which returns the index of n in the list of names In, sorted in lexicographic order.

Then, for the functions, we have two cases. We use a common helper function:

$$\begin{aligned} \text{lift}_1(f) &:= (((s_1, s_2), s), i) \mapsto (s_1', o) \leftarrow f(s_1, i); (((s_1', s_2), s), o) \\ \text{lift}_2(f) &:= (((s_1, s_2), s), i) \mapsto (s_2', o) \leftarrow f(s_2, i); (((s_1, s_2'), s), o) \end{aligned}$$

for $i \in 1, 2$, which uses the right side of the state, to lift a function operating on one side to operate on the whole state.

For $i \in [1, \ldots, A.n]$, we have:

$$f_i := (h_1, \ldots, h_m) \mapsto \text{lift}_1(A.f_i[h_{\pi_{\text{in}}(A.\pi_{\text{in}}^{-1}(j))} \mid j \in [A.m]])$$

Then, for $i \in [A.n + 1, \ldots, A.n + B.n]$, we have:

$$f_i := (h_1, \ldots, h_m) \mapsto \text{lift}_2(B.f_i[h_{\pi_{\text{in}}(B.\pi_{\text{in}}^{-1}(j))} \mid j \in [B.m]])$$

□

The state of $A \otimes B$ is just the state of both packages, and $A \otimes B$ also takes in the inputs of both packages, which may overlap, and produces the output functions of both packages. We require that these output functions do not overlap, to make it clear which function belongs to which "side" of the package.

Defining the output functions requires a little bit of technical juggling. One detail is that we start with functions expecting to receive just their state, but need to augment them to receive both states, and then place the result on the corresponding side. Another technical detail of our formalism shows up here as well, since $A.f_i$ and $B.f_i$ are paramterized functions, which pick up an extra state term s after being instantiated with their inputs, and so lift_i needs to also carry this term around. We also choose to arrange the output functions by A first, and then B, but the order we've chosen is arbitrary.

Now, the trickier details relate to the input functions. The basic issue is that we need to change the functions so that they technically accept all the input functions of $A \otimes B$, but ignore the ones irrelevant to either A or B. We do this by choosing an "arbitrary" permutation for π_{in}, and then pass in the right inputs to A or B by using their input permutations backwards, allowing us to look up the name associated with a given index, which we then use to figure out the right index according to π_{in}.

We choose π_{in} to be the lexicographic ordering, because it's a consistent ordering which does not depend on either A or B, and also doesn't care about the order in which packages are composed. This technically introduces a new assumption about names, since we haven't assumed anything about what a name is yet. However, assuming that names can be sorted alphabetically is not a very strong one.

Continuing our analogy of machines, we can see the tensoring of $A \otimes B$ as having two independent machines, side-by-side, that one can interact with at will. The state of one machine doesn't interfere with the state of the other, although both machines might be connected to some common machine "behind" them, through composition.

Like with composition, tensoring is also associative.

Lemma 2.4 (Tensoring is Associative). Given packages A, B, C, it holds that:

$$A \otimes (B \otimes C) \equiv (A \otimes B) \otimes C$$

provided these expressions are well defined.

Proof: The state types are $(A.S, (B.S, C.S))$ and $((A.S, B.S), C.S)$, which are isomorphic.

The input names are $\text{In}(A) \cup \text{In}(B) \cup \text{In}(C)$ on both sides, and the output names are $\text{Out}(A) \cup \text{Out}(B) \cup \text{Out}(C)$ for both sides as well.

Next, we get to the crux of the proof, which looks at the functions.

First, some observations about $\text{lift}_i(\text{lift}_j(f))$. These compositions can always be written in terms of a tuple with 3 elements:

$$\text{lift}'_j(f) := (((s_1, s_2, s_3), s), i) \mapsto (s_j, o) \leftarrow f(s_j, i);\ (((s_1, s_2, s_3), s), o)$$

The relation between them is that:

$$\begin{aligned}
\text{lift}_1(\text{lift}_1(f)) &= \text{lift}'_1(f) \\
\text{lift}_1(\text{lift}_2(f)) &= \text{lift}'_2(f) \\
\text{lift}_2(\text{lift}_1(f)) &= \text{lift}'_2(f) \\
\text{lift}_2(\text{lift}_2(f)) &= \text{lift}'_3(f)
\end{aligned}$$

So, in both $A \otimes (B \otimes C)$, and $(A \otimes B) \otimes C$, the functions will be of one of three forms:

1. $\text{lift}_1(A.f_i[\ldots])$,
2. $\text{lift}_2(B.f_i[\ldots])$,
3. $\text{lift}_3(C.f_i[\ldots])$.

The order of the functions will actually be the same in both cases.

The only remaining difference, potentially, is the instantiation. But, our definition ensures that the instantiation depends only on the names of the functions, but these are the same in both cases, so we conclude that the functions are equal.

■

Like with composition, associativity lets us forget about the way we group multiple tensorings together, letting us simply write $A \otimes B \otimes C$.

Tensoring also satisfies an additional property compared to composition. Because tensoring just provides the functions of both packages, it shouldn't actually matter which order we tensor packages together, since the resulting functions are the same.

Lemma 2.5 (Tensoring is Commutative). Given packages A, B, it holds that:

$$A \otimes B \equiv B \otimes A$$

provided these expressions are well defined.

Proof: The state on the left is $(A.S, B.S)$, and $(B.S, A.S)$ on the right. These states are isomorphic, as we've seen before.

Similarly, since $\cup$ is commutative, In and Out will match on both sides.

The inputs to each of the functions depend only on the set of names of the input functions, which are identical for both sides. The ordering is different though, but it suffices to swap f_i with $f_{i+A.n}$ to make the ordering the same.

Thus, we conclude that the two packages are the same.

■

So far, we've treated composition and tensoring as two separate operations, but very often we want to use them together. Very often we want to be able to decompose a large package into smaller components, using tensoring and composition. Then we'll rearrange these components around to make proving certain properties easier.

One key observation making this kind of rearrangement easier is related to how tensoring and composition interact with each other.

Lemma 2.6 (Interchange Lemma). Given packages A,B, C, D, such that $\text{In}(A) \cap \text{Out}(D) = \emptyset$ and $\text{In}(C) \cap \text{Out}(B) = \emptyset$

$$\begin{pmatrix} A \\ \otimes \\ C \end{pmatrix} \circ \begin{pmatrix} B \\ \otimes \\ D \end{pmatrix} \equiv \begin{matrix} (A \circ B) \\ \otimes \\ (C \circ D) \end{matrix}$$

Proof: The state on the left is $((A.S, C.S), (B.S, D.S))$, while the state on the right is $((A.S, B.S), (C.S, D.S))$. These states are isomorphic, of course.

Now, let's look at In and Out. On the left, we have:

$$\text{In}\left(\begin{pmatrix} A \\ \otimes \\ C \end{pmatrix} \circ \begin{pmatrix} B \\ \otimes \\ D \end{pmatrix}\right) = \text{In}\begin{pmatrix} B \\ \otimes \\ D \end{pmatrix} = \text{In}(B) \cup \text{In}(D)$$

On the right, we have:

$$\text{In}\begin{pmatrix} (A \circ B) \\ \otimes \\ (C \circ D) \end{pmatrix} = \text{In}(A \circ B) \cup \text{In}(C \circ D) = \text{In}(B) \cup \text{In}(D)$$

For Out, on the left we have:

$$\text{Out}\left(\begin{pmatrix} A \\ \otimes \\ C \end{pmatrix} \circ \begin{pmatrix} B \\ \otimes \\ D \end{pmatrix}\right) = \text{Out}\begin{pmatrix} A \\ \otimes \\ C \end{pmatrix} = \text{Out}(A) \cup \text{Out}(c)$$

On the right, we have:

$$\text{Out}\begin{pmatrix} (A \circ B) \\ \otimes \\ (C \circ D) \end{pmatrix} = \text{Out}(A \circ B) \cup \text{Out}(C \circ D) = \text{Out}(A) \cup \text{Out}(C)$$

Now, we look at the functions.

On the left, we start with functions of the form:

$$(h_1, \ldots) \mapsto \text{lift}_1(A.f_i[h_{(A\otimes C).\pi_{\text{in}}(A.\pi_{\text{in}}^{-1}(j))} \mid j \in [A.m]])$$
$$(h_1, \ldots) \mapsto \text{lift}_2(C.f_i[h_{(A\otimes C).\pi_{\text{in}}(C.\pi_{\text{in}}^{-1}(j))} \mid j \in [C.m]])$$

then, after composing with $B \otimes D$, using our assumption that A uses only functions from B, and C only functions from D, we get:

$$(h_1, \ldots) \mapsto \text{lift}_1(A.f_i[\text{lift}_1(\varphi_A(B.f_1)[h_{(B\otimes D).\pi_{\text{in}}(B.\pi_{\text{in}}^{-1}(j))} \mid j \in [B.m]]), \ldots])$$

$$(h_1, \ldots) \mapsto \text{lift}_2(C.f_i[\text{lift}_2(\varphi_C(D.f_1)[h_{(B\otimes D).\pi_{\text{in}}(B.\pi_{\text{in}}^{-1}(j))} \mid j \in [D.m]]), \ldots])$$

This is because in $A \otimes C$, the order parameters are instantiated depends only on the names of the function, and so the order will correspond with that of φ_A or φ_C, respectively.

From the right, the functions will be of the forms:

$$(h_1, \ldots) \mapsto \text{lift}_1(A.f_i[\varphi_A(B.f_1)[h_{\pi_{\text{in}}((A\circ B).\pi_{\text{in}}^{-1}(j))} \mid j \in [(A \circ B).m]]])$$

$$(h_1, \ldots) \mapsto \text{lift}_2(C.f_i[\varphi_C(D.f_1)[h_{\pi_{\text{in}}((C\circ D).\pi_{\text{in}}^{-1}(j))} \mid j \in [(C \circ D).m]]])$$

Now, $(A \circ B).m = B.m$, and ditto for $C \circ D$. Furthermore, the π_{in} used here is the same as $(B \otimes D).\pi_{\text{in}}$, since the function only depends on $\text{In}(B) \cup \text{In}(D)$.

The remaining difference is about

$$(A \otimes C).\text{lift}_i(f[(B \otimes D).\text{lift}_i(g)]) \stackrel{?}{=} ((A \otimes C) \circ (B \otimes D)).\text{lift}_i(f[g])$$

Expanding the right hand side, for $i = 1$, we get:

$$(((s_A, s_B, s_C, s_D), s), i) \mapsto (s_A, s_B, o) \leftarrow f[g](((s_A, s_B), s), i);\ (((s_A, s_B, s_C, s_D), s), o)$$

An equivalent way of writing this would be:

$$\begin{aligned}
&((((s_A, (s_B, s_D)), s_C), s), i) \mapsto \\
&\quad ((s_A, (s_B, s_D)), o) \leftarrow f[\text{lift}_1(g)](((s_A, (s_B, s_D)), s), i) \\
&\quad ((((s_A, (s_B, s_D)), s_C), s), o)
\end{aligned}$$

But this is just $\text{lift}_1(f[\text{lift}_1(g)])$. A similar argument works for $i = 2$ as well.

Having eliminated all differences between the functions for the packages we're comparing, we conclude our proof.

■

This proof marks the last very technical proof using the formal definition of packages. We've now developed almost all of the machinery we need to start reasoning about packages syntactically, using the fundamental operations and properties we've just defined.

We do need one more gadget though, which allows us to easily thread functions around.

Definition 2.10 (Identity Packages). Given a set of names N, we can define the identity package $1(N)$ as a package characterized by:

1. A state $S := \emptyset$,
2. $\text{In} = N$,
3. $\text{Out} = N$,
4. $\pi_{\text{in}} = \pi_{\text{out}}^{-1}$, based on a lexicographical ordering of N,
5. Functions $f_1, \ldots, f_{|N|}$ defined via:

$$f_i := (h_1, \ldots, h_m) \mapsto h_i$$

□

In other words, the identity package $1(N)$ simply uses some functions, and provides them without any changes whatsoever. This means that $1(\text{Out}(A)) \circ A \equiv A$, and $B \circ 1(\text{In}(B)) = B$, which is why we call this an identity package.

On its own, this might not seem all that useful, but it becomes essential when combined with tensoring, allowing us to define packages such as:

$$\begin{pmatrix} A \\ \otimes \\ 1(\text{Out}(B)) \end{pmatrix} \circ B$$

Here, B is used both by A, but its functions are also forwarded further. This kind of arrangement is very useful when defining packages.

We also have a few pieces of shorthand that are useful for identity packages. We write $1(A, B, \ldots)$ for $1(A \cup B \cup \ldots)$, and we also sometimes abuse notation to write $1(P)$ where P is a package, to mean $1(\text{Out}(P))$, since forwarding the entire output of a package is a very common operation.

2.4 Indistinguishability and Reductions

The goal of this subsection is to define more useful notions of equality. Literal equality is far too strict, since it will not allow for many modifications which yield packages that are *effectively* the same. Furthermore, in many situations, we want to consider packages that are hard to tell apart with limited computational resources; such "hard problems" are the basis of many cryptographic schemes. Furthermore, we want to relate the hardness of distinguishing one pair of packages to the hardness of distinguishing another pair: this is the notion of *reduction*.

First, we need to extend our notion of *efficiency* from functions to packages.

Definition 2.11 (Efficient Packages). A package P is said to be *efficient* if all of its functions are efficient.

In turn, a parametrized function f is *efficient* if for any efficient functions $h_1, \ldots,$, the instantiation $f[h_1, \ldots]$ is also efficient.

□

This is a very natural definition of efficiency, and one can verify that efficiency is preserved under both tensoring and composition.

The next notion we define is that of the *game*.

Definition 2.12 (Game). A game G is a package with $\text{In}(G) = \emptyset$.

□

This is a very simply distinction, but it's important, because when a package has no input functions, then one can interact with it as a complete machine already, there's nothing that needs to be plugged in before the machine can actually "run".

The next fundamental notion we define is that of the *adversary*. Intuitively, adversaries are trying to distinguish games with the same interface apart. A "hard" problem can be characterized by a pair of games that no efficient adversary can tell apart.

Definition 2.13 (Adversaries). An adversary $\mathcal{A}$ for a package P, is a package with no state, $\text{In}(\mathcal{A}) = \text{Out}(P)$, and $\text{Out}(\mathcal{A}) = \{\text{run}\}$, where $\text{run} : \bullet \xrightarrow{\$} \texttt{01}$.

□

We'll use adversaries to define some notions of indistinguishability for games first, but we already define adversaries as being for *packages*, to be ready for when we extended these notions later.

We can think of an adversary as playing a "game" of distinguishing between two packages. The goal of an adversary is to separate the two packages, by returning 0 in one case, and 1 in the other. The success of an adversary will be measured by how often it's able to distinguish the two packages.

Another point of view is that an adversary $\mathcal{A}$ is actually a mapping from games with a given interface to *functions* of type $\bullet \xrightarrow{\$} \texttt{01}$. Each game we feed to the adversary yields a different function. This is particularly convenient because we've already developed notions of equality and distance for functions, and we can use this mapping to lift the notions to packages as well.

This leads to our next definition:

Definition 2.14 (Adversarial Distance). Given two games G, H with $\text{Out}(G) = \text{Out}(H)$, and an adversary $\mathcal{A}$ for G or H, we define their adversarial distance relative to $\mathcal{A}$ as:

$$\varepsilon_{\mathcal{A}}(G, H) := \varepsilon(\mathcal{A} \circ G, \mathcal{A} \circ H)$$

Here we abuse notation a bit to let $\mathcal{A} \circ X$ denote the *function* we get by calling run.

□

As the name suggests, this relation also forms a distance metric.

Like with functions, this also leads to a natural notion of equality for games. But first, to avoid having to say $\text{Out}(G) = \text{Out}(H)$ many times, we define the following shorthand:

Definition 2.15 (Game Shape). Two games G,H are said to have the *same shape* if $\text{Out}(G) = \text{Out}(H)$.

□

We can then continue with our definition of equality.

Definition 2.16 (Game Equality). Given two games G and H with the same shape, we say that G and H are *equal*, written $G = H$, if for all adversaries $\mathcal{A}$, we have:

$$\varepsilon_{\mathcal{A}}(G, H) = 0$$

□

Note that we consider all adversaries, even potentially unbounded ones. Because adversarial distance is a metric, we also immediately conclude that this relation is a valid equality relation.

We've intentionally used the = symbol here, because we think that this is the most natural notion of equality for games. It allows for inessential differences to be ignored, such as two ways of sampling from the same distribution, but it's also not too loose of a notion either, since we consider *unbounded* adversaries. Any tangible difference in distributions can be sniffed out by such a powerful adversary.

We do nonetheless want to develop a looser notion of equality, which can both allow for a small possibility of success in distinguishing two games, as well as the possibility of genuinely hard problems, by restricting the resources of the adversary.

Definition 2.17 (Game Indistinguishability). Given two games G and H with the same shape, we say that G and H are indistinguishable up to ϵ, written $G \overset{\epsilon}{\approx}$

H, if for all *efficient* adversaries $\mathcal{A}$, we have:

$$\varepsilon_{\mathcal{A}}(G, H) \leq \epsilon$$

□

This definition only considers efficient adversaries to allow for hard problems to exist, and also allows a bit of a "gap", letting the adversary have some success at distinguishing the two games.

When we say that distinguishing a pair of games G_0, G_1 reduces to distinguishing a pair of games H_0, H_1, we mean that G_b is at least as hard as H_b, in the sense that any attack against G_b can be converted to an attack against H_b, with some reasonable relationship on the success probability.

More formally, a reduction is a statement of the form: "for all (efficient) adversaries $\mathcal{A}$ against G_b, there exists an (efficient) adversary $\mathcal{B}$ against H_b, such that $\varepsilon_{\mathcal{A}}(G_0, G_1)$ is at most $\varepsilon_{\mathcal{B}}(H_0, H_1)$". This statement is interesting because if $\varepsilon_{\mathcal{B}}$ is "small", then $\varepsilon_{\mathcal{A}}$ will also be "small". So, G_b is hard assuming H_b is.

The way we'd translate that in a statement about ε is by saying that:

$$\varepsilon_{\mathcal{A}}(G_0, G_1) \leq \sup_{\text{efficient } \mathcal{B}} \varepsilon_{\mathcal{B}}(H_0, H_1)$$

or, in shorthand:

$$G_0 \overset{H_b}{\approx} G_1$$

Now, the reduction statement above implies this one, since if the advantage $\mathcal{A}$ is bounded above by the advantage of some $\mathcal{B}$, then it is certainly bounded above by the one with the highest advantage. Using the supremum also has the advantage of summarizing the reduction property into a single real number. If a problem is actually hard, the supremum should be small too, since there shouldn't be any $\mathcal{B}$ with a large advantage.

We can also add these advantages together, as this following lemma shows.

Lemma 2.7 (Transitivity of Indistinguishability). Given games G, H, I satisfying:

$$G \overset{\epsilon_1}{\approx} H, \quad H \overset{\epsilon_2}{\approx} I$$

it holds that:

$$G \overset{\epsilon_1+\epsilon_2}{\approx} I$$

In more detail, for all efficient adversaries $\mathcal{A}$, we have:

$$\varepsilon_{\mathcal{A}}(G, I) \leq \epsilon_1 + \epsilon_2$$

Proof: Since $\varepsilon_{\mathcal{A}}$ is a metric, it satisfies the triangle inequality, so we have:

$$\varepsilon_{\mathcal{A}}(G, I) \leq \varepsilon_{\mathcal{A}}(G, H) + \varepsilon_{\mathcal{A}}(H, I)$$

Then, we just need to apply our assumptions to get the upper bound we need to prove.

■

This notion of transitivity is very useful, since it lets us argue that two different games are equal by appealing to several successive differences. For example, some system might use both encrytion and signing, and we can appeal to the hardness of both problems, one at a time, to argue that the system is secure. This kind of technique is called "game hopping", and one of the strengths of state-separable proofs is making the application of the technique as simple and route as possible.

Having defined these notions of equality for games, we now extend them to *packages*. The natural way to do this is by trying to turn a package into a game, and then using the notions we've just developed.

Let's look at a way to do this transformation.

Definition 2.18 (Completion). Given a package A, a completion of A is a game C, such that $\text{Out}(C) \supseteq \text{In}(A)$, and $\text{Out}(C) \cap \text{Out}(A) = \emptyset$.

We write:

$$\text{Compl}_C(A) := \begin{pmatrix} A \\ \otimes \\ 1(C) \end{pmatrix} \circ C$$

□

So, a completion is one way of turning a package into a game. It does so by filling in all of the input functions, but it also leaks extra information forward. The reason behind this is so that an adversary is also able to see what's happening "behind" the package A. Note that for completions, the names of the extra functions, those in $\text{Out}(C)/\text{In}(A)$ are very inessential, and should be considered as being distinct from any other name used by a real package.

Before we extend our notions of equality to packages, we need to quickly extend our notion of *shape* first.

Definition 2.19 (Package Shape). Two packages A, B, are said to have the *same shape* if $\text{Out}(A) = \text{Out}(B)$, and $\text{In}(A) = \text{In}(B)$.

□

We're now ready to define equality and indistinguishability for packages.

Definition 2.20 (Package Equality and Indistinguishability). Given two packages A, B with the same shape, we say that:

1. A is equal to B, written $A = B$, if for all completions C, we have $\text{Compl}_C(A) = \text{Compl}_C(B)$,
2. A is indistinguishable up to ϵ with B, written $A \overset{\epsilon}{\approx} B$, if for all *efficient* completions C, we have $\text{Compl}_C(A) \overset{\epsilon}{\approx} \text{Compl}_C(B)$.

□

A completion turns a package into a game, so it's natural to compare packages by using completions. However, there's no "canonical" completion, so it's not clear which one to use to compare the packages. We get around this problem by simply using all of them.

One way of looking at these notions of equality is that we have an adversary which completely surrounds a package A, seeing both the "front", via $\text{Out}(A)$, and the "back", via $\text{In}(A)$, and can distinguish the package from others by influencing either side. This is why it's important that the adversary can interact with C directly, so that $\mathscr{A}$ and C effectively form one unified adversary.

The basic properties of equality, like symmetry and transitivity, also hold for packages, given the definition in terms of games.

2.5 Some Properties of Equality

So far, we've seen three notions of equality:

1. Literal Equality ($\equiv$),
2. Equality ($=$),
3. Indistinguishability ($\overset{\epsilon}{\approx}$).

We've considered them in isolation, but in fact there's a very natural link between the three: each of them is strictly stronger than the other. We capture this fact in the following theorem.

Lemma 2.8 (Equality Hierarchy). For any packages A, B with the same shape, it holds that:

1. $A \equiv B \implies A = B$,
2. $A = B \implies A \overset{0}{\approx} B$.

Proof: For part one, if $A \equiv B$, then $\mathscr{A} \circ \text{Compl}_C(A) \equiv \mathscr{A} \circ \text{Compl}_C(B)$, since composition and tensoring preserve literal equality. But, in that case, by definition of $\equiv$, the run functions must be equal in both cases, which means that:

$$\varepsilon(\mathcal{A} \circ \text{Compl}_C(A), \mathcal{A} \circ \text{Compl}_C(B)) = 0$$

which is what we needed to prove.

For part 2, note that if for *every* adversary $\mathcal{A}$ and completion C, we have:

$$\varepsilon_{\mathcal{A}}(\text{Compl}_C(A), \text{Compl}_C(B)) = 0$$

then, in particular, this relation holds for every *efficient* adversary and completion as well, which is what we needed to prove.

■

This hierarchy is quite useful, since we can prove precise equality relations between packages, but then ultimately use them in game hopping, where only $\approx$ matters. The hierarchy also lets us basically forget about $\equiv$, since whenever we would've used it, we can just use $=$ instead, which is applicable to many more packages.

The main properties we need to prove to wrap up our formal discussion of packages is to show that the composition operations we've defined respect equality and indistinguishability. This is very important, since it lets us reason about large packages by arguing that small components are equal or indistinguishable, and will form the crux of most proofs.

We start with tensoring, since the proof is simpler.

Lemma 2.9 (Tensoring Respects Equality). Given packages A, B, B', it holds that:

1. $B = B' \implies A \otimes B = A \otimes B'$,
2. $B \overset{\epsilon}{\approx} B' \implies A \otimes B \overset{\epsilon}{\approx} A \otimes B'$.

provided that these expressions are well defined, and, for part 2, that A is efficient.

Proof:

1. Let C be some completion for $A \otimes B$. We have:

$$\text{Compl}_C(A \otimes B) = \begin{pmatrix} A \\ \otimes \\ B \\ \otimes \\ 1(C) \end{pmatrix} \circ C$$

Now, we apply interchange to write this as:

$$\begin{pmatrix} A \\ \otimes \\ 1(B) \\ \otimes \\ 1(C) \end{pmatrix} \circ \begin{pmatrix} B \\ \otimes \\ 1(C) \end{pmatrix} \circ C = W \circ \mathrm{Compl}_C(B)$$

for some package $\mathcal{W}$. For any adversary $\mathcal{A}$, we have:

$$\varepsilon_{\mathcal{A}}(\mathrm{Compl}_C(B), \mathrm{Compl}_C(B')) = 0$$

In particular, for any adversary $\mathcal{A}'$ against $\mathrm{Compl}_C(A \otimes B)$, we can apply this observation to $\mathcal{A}' \circ W$, giving us:

$$\varepsilon_{\mathcal{A}'}(W \circ \mathrm{Compl}_C(B), W \circ \mathrm{Compl}_C(B')) = 0$$

Since this observation holds for any $\mathcal{A}'$, we infer that:

$$W \circ \mathrm{Compl}_C(B) = W \circ \mathrm{Compl}_C(B')$$

Then, applying transitivity, we conclude that:

$$\mathrm{Compl}_C(A \otimes B) = \mathrm{Compl}_C(A \otimes B')$$

2. We apply the observation we had above, which is that:

$$\mathrm{Compl}_C(A \otimes B) = W \circ \mathrm{Compl}_C(B)$$

(and similarly for B'). Now, by assumption for any efficient adversary $\mathcal{A}$, we have:

$$\varepsilon_{\mathcal{A}}(\mathrm{Compl}_C(B), \mathrm{Compl}_C(B')) \leq \epsilon$$

In particular, we can apply this to $\mathcal{A}' \circ W$, for any adversary $\mathcal{A}'$ against $A \otimes B$, since W is efficient, by virtue of A being efficient. This gives us:

$$\varepsilon_{\mathcal{A}'}(W \circ \mathrm{Compl}_C(B), W \circ \mathrm{Compl}_C(B')) \leq \epsilon$$

This means that:

$$W \circ \mathrm{Compl}_C(B) \stackrel{\epsilon}{=} W \circ \mathrm{Compl}_C(B')$$

We then use transitivity to conclude that:

$$A \otimes B \stackrel{\epsilon}{=} A \otimes B'$$

■

Next, we prove the same kind of theorem about composition.

Lemma 2.10 (Composition Respects Equality). Given packages A, B, B', C, it holds that:

1. $B = B' \implies A \circ B = A \circ B'$,
2. $B \overset{\epsilon}{\approx} B' \implies A \circ B \overset{\epsilon}{\approx} A \circ B'$,
3. $B = B' \implies B \circ C = B' \circ C$,
4. $B \overset{\epsilon}{\approx} B' \implies B \circ C \overset{\epsilon}{\approx} B \circ C$,

provided that these expressions are well defined, and for parts 2 and 4, that A and C are efficient, respectively.

Proof:

1. For any completion C, we can write:

$$\text{Compl}_C(A \circ B) = \begin{pmatrix} A \circ B \\ \otimes \\ 1(C) \end{pmatrix} \circ C = \begin{pmatrix} A \\ \otimes \\ 1(C) \end{pmatrix} \circ \begin{pmatrix} B \\ \otimes \\ 1(C) \end{pmatrix} \circ C$$

by applying interchange. We can write this as:

$$W \circ \text{Compl}_C(B)$$

for some package W depending on A and $\text{Out}(C)$.

Then, we apply a similar logic as in our proof of Lemma 2.9. For any adversary $\mathcal{A}$, we have:

$$\varepsilon_{\mathcal{A}}(\text{Compl}_C, \text{Compl}_C(B')) = 0$$

Thus, for any $\mathcal{A}'$ against $A \circ B$, we apply the above to $\mathcal{A}' \circ W$, getting:

$$\varepsilon_{\mathcal{A}'}(W \circ \text{Compl}_C(B), W \circ \text{Compl}_C(B')) = 0$$

In other words, we have:

$$W \circ \text{Compl}_C(B) = W \circ \text{Compl}_C(B')$$

We can then apply transitivity to conclude that $A \circ B = A \circ B'$.

2. We start with the same observation, that:

$$\text{Compl}_C(A \circ B) = W \circ \text{Compl}_C(B)$$

for some package W. By applying our assumption to $\mathcal{A}' \circ W$ for any adversary $\mathcal{A}'$ against $A \circ B$, we see that:

$$\varepsilon_{\mathcal{A}'}(W \circ \text{Compl}_C(B), W \circ \text{Compl}_C(B')) \leq \epsilon$$

In other words,

$$W \circ \mathrm{Compl}_C(B) \overset{\epsilon}{\approx} W \circ \mathrm{Compl}_C(B')$$

and then apply transitivity to reach our conclusion.

3. For any completion C, we can write:

$$\mathrm{Compl}_C(B \circ C) = \begin{pmatrix} B \circ C \\ \otimes \\ 1(C) \end{pmatrix} \circ C = \begin{pmatrix} B \\ \otimes \\ 1(C) \end{pmatrix} \circ \begin{pmatrix} 1(B) \circ C \\ \otimes \\ 1(C) \end{pmatrix} \circ C$$

We can then see C as part of a new completion, writing:

$$\begin{pmatrix} B \\ \otimes \\ 1(C') \end{pmatrix} \circ C' = \mathrm{Compl}_{C'}(B)$$

But, by assumption, we have:

$$\mathrm{Compl}_{C'}(B) = \mathrm{Compl}_{C'}(B')$$

We then apply our initial observation in reverse, along with transitivity, to reach our conclusion.

4. Same as above, except our assumption gives us:

$$\mathrm{Compl}_{C'}(B) \overset{\epsilon}{=} \mathrm{Compl}_{C'}(B')$$

and then transitivity can be applied to reach our result once again.

■

2.6 Syntactical Conventions for Packages

3 Systems

3.1 Asynchronous Packages

While the intuition of yield statements is simple, defining them precisely is a bit more tricky.

Definition 3.1 (Yield Statements). We define the semantics of **yield** by compiling functions with such statements to functions without them.

Note that we don't define the semantics for functions which still contain references to oracles. Like before, we can delay the definition of semantics until all of the pseudo-code has been inlined.

A first small change is to make it so that the function accepts one argument, a binary string, and all yield points also accept binary strings as continuation. Like with plain packages, we can implement richer types on top by adding additional checks to the well-formedness of binary strings, aborting otherwise.

The next step is to make it so that all the local variables of the function F are present in the global state. So, if a local variable v is present, then every use of v is replaced with a use of the global variable $F.v$ in the package. This allows the state of the function to be saved across yields.

The next step is transforming all the control flow of a function to use **ifgoto**, rather than structured programming constructs like **while** or **if**. The function is broken into lines, each of which contains a single statement. Each line is given a number, starting at 0. The execution of a function F involves a special variable `pc`, representing the current line being executing. Excluding **yield** and **return** a single line statement has one of the forms:

$$\langle \texttt{var} \rangle \leftarrow \langle \texttt{expr} \rangle$$

$$\langle \texttt{var} \rangle \xleftarrow{\$} \langle \texttt{dist} \rangle$$

which have well defined semantics already. Additionally, after these statements, we set $\texttt{pc} \leftarrow \texttt{pc} + 1$.

The semantics of **ifgoto** $\langle \texttt{expr} \rangle i$ is:

$$\texttt{pc} \leftarrow \textbf{ if } \langle \texttt{expr} \rangle \textbf{ then } i \textbf{ else } \texttt{pc} + 1$$

This gives us a conditional jump, and by using `true` as the condition, we get a standard unconditional jump.

This allows us to define **if** and **while** statements in the natural way.

Finally, we need to augment functions to handle **yield** and **return** statements. To handle this, each function F also has an associated variable $F.\texttt{pc}$, which stores the program counter for the function. This is different than the local `pc` which is while the function is execution. $F.\texttt{pc}$ is simply used to remember the program counter after a yield statement.

The function now starts with:

$$\textbf{ifgoto } \texttt{true } F.\text{pc}$$

This has the effect of resuming execution at the saved program counter.

Furthermore, the input variable x to F is replaced with a special variable `input`, which holds the input supplied to the function. At the start of the function body, we add:

$$0 : F.x \leftarrow \texttt{input}$$

to capture the fact that the original input variable needs to get assigned to the input to the function.

The semantics of $F.m \leftarrow$ **yield** v are:

$$
\begin{aligned}
(i-1) &: F.\text{pc} \leftarrow i+1 \\
i &: \textbf{return } (\texttt{yield}, v) \\
(i+1) &: F.m \leftarrow \texttt{input}
\end{aligned}
$$

The semantics of **return** v become:

$$
\begin{aligned}
& F.\text{pc} \leftarrow 0 \\
& \textbf{return } (\texttt{return}, v)
\end{aligned}
$$

The main difference is that we annotate the return value to be different than yield statements, but otherwise the semantics are the same.

□

Note that while calling a function which can yield will notify the caller as to whether or not the return value was *yielded* or *returned*, syntactically the caller often ignores this, simply doing $x \leftarrow F(\ldots)$, meaning that they simply use return value x, discarding the tag.

Syntax 3.2 (Empty Yields). In many cases, no value is yielded, or returned back, which we can write as:

$$\textbf{yield}$$

which is shorthand for:

$$\bullet \leftarrow \textbf{yield } \bullet$$

i.e. just yielding a dummy value and ignoring the result.

□

Unless otherwise specified, we only consider empty yields from now on.

We define these semantics via the **await** statement.

Syntax 3.3 (Await Statements). We define the semantics of $v \leftarrow$ **await** $F(\ldots)$ in a straightforward way:

$$
\begin{aligned}
& (\text{tag}, v) \leftarrow (\texttt{yield}, \bot) \\
& \textbf{while } \text{tag} = \texttt{yield} : \\
& \quad \textbf{if } v \neq \bot : \\
& \quad\quad \textbf{yield} \\
& \quad (\text{tag}, v) \leftarrow F(\ldots)
\end{aligned}
$$

In other words, we keep calling the function until it actually returns its final value, but we do yield to our caller whenever our function yield, but we do yield to our caller whenever our function yields.

□

Sometimes we want to await several values at once, returning the first one which completes. To that end, we define the **select** statement.

Syntax 3.4 (Select Statements). Select statements generalize await statements in that they allow waiting for multiple events concurrently.

More formally, we define:

$$
\begin{aligned}
&\textbf{select}: \\
&\quad v_1 \leftarrow \textbf{await } F_1(\ldots): \\
&\quad\quad \langle \text{body}_1 \rangle \\
&\quad \vdots \\
&\quad v_n \leftarrow \textbf{await } F_n(\ldots): \\
&\quad\quad \langle \text{body}_n \rangle
\end{aligned}
$$

As follows:

$$
\begin{aligned}
&(\text{tag}_i, v_i) \leftarrow (\texttt{yield}, \bot) \\
&i \leftarrow 0 \\
&\textbf{while } \nexists i.\ \text{tag}_i \neq \texttt{yield}: \\
&\quad \textbf{if } i \geq n: \\
&\quad\quad i \leftarrow 0 \\
&\quad\quad \textbf{yield} \\
&\quad i \leftarrow i + 1 \\
&\quad (\text{tag}_i, v_i) \leftarrow F_i(\ldots) \\
&\langle \text{body}_i \rangle
\end{aligned}
$$

Note that the order in which we call the functions is completely deterministic, and fair. It's also important that we yield, like with await statements, but we only do so after having pinged each of our underlying functions at least once. This is so that if one of the function is immediately ready, we never yield.

□

Definition 3.5 (Asynchronous Packages). An *asynchronous* package P is a package which uses the additional syntax from Definition 3.1 and Syntax 3.3, 3.4.

□

Note that our syntax sugar definitions means that whenever one of the constructs such as yield and what not are used, they are immediately replaced with their underlying semantics. Thus, an asynchronous package *literally* is a package which does not use any of those syntactical constructs.

In/Out are well defined elaborate. Naturally, the definitions of $\circ$ and $\otimes$ for packages also generalize directly to asynchronous packages.

3.2 Channels and System Composition

Definition 3.6 (Systems). A *system* is a package which uses channels.

We denote by InChan(S) the set of channels the system receives on, and OutChan(S) the set of channels the system sends on, and define

$$\text{Chan}(S) := \text{OutChan}(S) \cup \text{InChan}(S)$$

Additionally we require that $\text{OutChan}(S) \cap \text{InChan}(S) = \emptyset$

□

We also define shorthands $\text{Chan}(A, B, \ldots) = \text{Chan}(A) \cup \text{Chan}(B) \cup \ldots$. expand.

Definition 3.7. We can compile systems to not use channels. We denote by $\text{NoChan}(S)$ the package corresponding to a system S, with the use of channels replaced with function calls.

Channels define three new syntactic constructions, for sending and receiving along a channel, along with testing how many messages are in a channel. We replace these with function calls as follows:

Sending, with $m \Rightarrow P$ becomes:

$$\text{Channels.Send}_P(m)$$

Testing, with $n \leftarrow \textbf{test } P$ becomes

$$n \leftarrow \text{Channels.Test}_P()$$

Receiving, with $m \Leftarrow P$ becomes:

$$m \leftarrow \textbf{await } \text{Channels.Recv}_P()$$

Receiving is an asynchronous function, because the channel might not have any available messages for us.

These function calls are parameterized by the channel, meaning that that we have a separate function for each channel.

□

```
Channels({A_1, ..., A_n})

q[A_i] ← FifoQueue.New()

Send_{A_i}(m):
  q[A_i].Push(m)

Test_{A_i}():
  return q[A_i].Length()

Recv_{A_i}():
  while q[A_i].IsEmpty()
    yield
  q[A_i].Next()
```

Game 3.1: Channels

One consequence of this definition with separate functions for each channel is that $\text{Channels}(S) \otimes \text{Channels}(R) = \text{Channels}(S \cup R)$.

Armed with the syntax sugar for channels, and the Channels game, we can convert a system S into a package via:

$$\text{SysPack}(S) := \text{NoChan}(S) \circ (\text{Channels}(\text{Chan}(S)) \otimes 1(\text{In}(S)))$$

This package will have the same input and output functions as the system S, but with the usage of channels replaced with actual semantics.

This allows us to lift our standard equality relations on packages onto *systems*.

Definition 3.8. Given systems A, B, we say that they have the same *shape* if

- $\text{In}(A) = \text{In}(B)$,
- $\text{Out}(A) = \text{Out}(B)$,
- $\text{InChan}(A) = \text{InChan}(B)$,
- $\text{OutChan}(A) = \text{OutChan}(B)$.

□

Definition 3.9 (Literal System Equality). Given systems A, B with the same shape, we say that they are *literally* equal, written $A \equiv B$ if

$$\text{NoChan}(A) = \text{NoChan}(B)$$

□

Definition 3.10 (System Tensoring). Given two systems, A and B, with $\text{Out}(A) \cap \text{Out}(B) = \emptyset$, we can define their tensor product $A * B$, which is any system $A * B$ satisfying:

- $\text{NoChan}(A * B) = \text{NoChan}(A) \otimes \text{NoChan}(B)$,
- $\text{InChan}(A * B) = \text{InChan}(A) \cup \text{InChan}(B)$,
- $\text{OutChan}(A * B) = \text{OutChan}(A) \cup \text{OutChan}(B)$,
- $\text{In}(A * B) = \text{In}(A) \cup \text{In}(B)$.

□

Note that combining the definition above with the definition of SysPack means that:

$$\text{SysPack}(A * B) = \begin{pmatrix} \text{NoChan}(A) \\ \otimes \\ \text{NoChan}(B) \end{pmatrix} \circ \begin{pmatrix} \text{Channels}(\text{Chan}(A) \cup \text{Chan}(B)) \\ \otimes \\ 1(\text{In}(A) \cup \text{In}(B)) \end{pmatrix}$$

This implies the following lemma.

Lemma 3.1. System tensoring is associative, i.e. $A * (B * C) \equiv (A * B) * C$.
Proof: This follows directly from the associativity of $\otimes$ for packages and $\cup$.

■

Lemma 3.2. System tensoring is commutative, i.e. $A * B \equiv B * A$

Proof: This follows from the commutativity of $\otimes$ and $\cup$.

■

Definition 3.11 (Overlapping Systems). Two systems A and B overlap if $\text{Chan}(A) \cap \text{Chan}(B) \neq \emptyset$.

In the case of non-overlapping systems, we write $A \otimes B$ instead of $A * B$, insisting on the fact that they don't communicate.

Definition 3.12 (System Composition). Given two systems, A and B, we can define their (horizontal) composition $A \circ B$ as any system, provided a few constraints hold:

- A and B do not overlap ($\text{Chan}(A) \cap \text{Chan}(B) = \emptyset$)
- $\text{In}(A) \subseteq \text{Out}(B)$

With these in place, we define the composition as any system $A \circ B$ such that:

- $\text{NoChan}(A \circ B) = \text{NoChan}(A) \circ \begin{pmatrix} \text{NoChan}(B) \\ \otimes \\ 1(\text{Channels}(\text{Chan}(A))) \end{pmatrix}$,

- $\text{InChan}(A \circ B) = \text{InChan}(A) \cup \text{InChan}(B)$,
- $\text{OutChan}(A \circ B) = \text{OutChan}(A) \cup \text{OutChan}(B)$,
- $\text{In}(A \circ B) = \text{In}(B)$.

□

Lemma 3.3. System composition is associative, i.e. $A \circ (B \circ C) \equiv (A \circ B) \circ C$.

Proof: This follows from the associativity of $\circ$ for *packages*.

■

Lemma 3.4 (Interchange Lemma). Given systems A, B, C, D such that $\text{In}(A) \cap \text{Out}(D) = \emptyset$, and $\text{In}(C) \cap \text{Out}(B) = \emptyset$, and neither A nor C overlap with B or D, the following relation holds:

$$\begin{pmatrix} A \\ * \\ C \end{pmatrix} \circ \begin{pmatrix} B \\ * \\ D \end{pmatrix} \equiv \begin{matrix} (A \circ B) \\ * \\ (C \circ D) \end{matrix}$$

provided these expressions are well defined.

Proof: InChan, OutChan, and In are equal for both of these systems, by associativity of $\cup$. We now look at NoChan. Starting with the right hand side, we get:

$$\text{NoChan}\begin{pmatrix} (A \circ B) \\ * \\ (C \circ D) \end{pmatrix} = \begin{pmatrix} \text{NoChan}(A \circ B) \\ \otimes \\ \text{NoChan}(C \circ D) \end{pmatrix} = \begin{pmatrix} \text{NoChan}(A) \circ \begin{pmatrix} \text{NoChan}(B) \\ \otimes \\ 1(\text{Channels}(\text{Chan}(A))) \end{pmatrix} \\ \otimes \\ \text{NoChan}(C) \circ \begin{pmatrix} \text{NoChan}(D) \\ \otimes \\ 1(\text{Channels}(\text{Chan}(C))) \end{pmatrix} \end{pmatrix}$$

Next, apply the interchange lemma for packages, to get:

$$\begin{pmatrix} \text{NoChan}(A) \\ \otimes \\ \text{NoChan}(C) \end{pmatrix} \circ \begin{pmatrix} \text{NoChan}(B) \\ \otimes \\ 1(\text{Channels}(\text{Chan}(A))) \\ \otimes \\ \text{NoChan}(D) \\ \otimes \\ 1(\text{Channels}(\text{Chan}(C))) \end{pmatrix}$$

Then, observe that:

$$\text{Channels}(S_1 \cup S_2) = \text{Channels}(S_1) \otimes \text{Channels}(S_2)$$

We can use this, along with the commutativity of $\otimes$ to get:

$$\begin{pmatrix}\text{NoChan}(A)\\ \otimes\\ \text{NoChan}(C)\end{pmatrix} \circ \begin{pmatrix}\text{NoChan}(B)\\ \otimes\\ \text{NoChan}(D)\\ \otimes\\ 1(\text{Channels}(\text{Chan}(A * C)))\end{pmatrix}$$

Which is just:

$$\text{NoChan}\left(\begin{pmatrix}A\\ *\\ C\end{pmatrix} \circ \begin{pmatrix}B\\ *\\ D\end{pmatrix}\right)$$

■

Definition 3.13 (System Games). Analogously to games, we define a *system game* as a system S with $\text{In}(S) = \emptyset$.

□

Definition 3.14 (System Equality). We say that two systems A, B with the same shape are equal, written $A = B$, if:

$$\text{SysPack}(A) = \text{SysPack}(B)$$

□

Definition 3.15 (System Indistinguishability). We say that two systems A, B with the same shape are indistinguishable up to ϵ, written $A \overset{\epsilon}{\approx} B$, if:

$$\text{SysPack}(A) \overset{\epsilon}{\approx} \text{SysPack}(B)$$

□

Lemma 3.5 (Transitivity of System Equality). Given systems A, B, C, we have:

1. $A \equiv B, B \equiv C \implies A \equiv C$,
2. $A = B, B = C \implies A = C$,
3. $A \overset{\epsilon_1}{\approx} B, B \overset{\epsilon_2}{\approx} C \implies A \overset{\epsilon_1+\epsilon_2}{\approx} C$.

provided these expressions are well-defined.

Proof: This follows immediately from the fact that equality and Indistinguishability for *packages* satisfy similar relations, and the notions for systems are defined in terms of the package SysPack(...).

■

Lemma 3.6 (Composition Compatability). Given systems A, B, B', we have:

1. $B = B' \implies A \circ B = A \circ B'$,
2. $B \overset{\epsilon}{\approx} B' \implies A \circ B \overset{\epsilon}{\approx} A \circ B'$.

provided these expressions are well-defined.

Proof: We prove that

$$\text{SysPack}(A \circ B) = \text{SysPack}(A) \circ \text{SysPack}(B)$$

which then clearly implies this lemma by application of the similar properties for packages.

We start with:

$$\text{SysPack}(A \circ B) = \text{NoChan}(A) \circ \begin{pmatrix} \text{NoChan}(B) \\ \otimes \\ 1(\text{Channels}(\text{Chan}(A))) \end{pmatrix} \circ \begin{pmatrix} \text{Channels}(\text{Chan}(A) \cup \text{Chan}(B)) \\ \otimes \\ 1(\text{In}(B)) \end{pmatrix}$$

We then use the fact that $\text{Channels}(S \cup R) = \text{Channels}(S) \otimes \text{Channels}(R)$, and the interchange lemma, to get:

$$\text{NoChan}(A) \circ \begin{pmatrix} \text{NoChan}(B) \\ \otimes \\ \text{Channels}(\text{Chan}(A)) \end{pmatrix} \circ \begin{pmatrix} \text{Channels}(\text{Chan}(B)) \\ \otimes \\ 1(\text{In}(B)) \end{pmatrix}$$

Apply interchange once more, to get:

$$\text{NoChan}(A) \circ \begin{pmatrix} 1(\text{In}(A)) \\ \otimes \\ \text{Channels}(\text{Chan}(A)) \end{pmatrix} \circ \text{NoChan}(B) \circ \begin{pmatrix} \text{Channels}(\text{Chan}(B)) \\ \otimes \\ 1(\text{In}(B)) \end{pmatrix}$$

Which is none other than:

$$\text{SysPack}(A) \circ \text{SysPack}(B)$$

concluding our proof.

■

Lemma 3.7 (Strict Tensoring Compatability). Given systems A, B, B', we have:

1. $B = B' \implies A \otimes B = A \otimes B'$,
2. $B \overset{\epsilon}{\approx} B' \implies A \otimes B \overset{\epsilon}{\approx} A \otimes B'$.

provided these expressions are well-defined.

Proof: Similar to Lemma 3.6, we start by proving:

$$\mathrm{SysPack}(A \otimes B) = \mathrm{SysPack}(A) \otimes \mathrm{SysPack}(B)$$

which then entails our theorem through similar properties for packages.

Our starting point is:

$$\mathrm{SysPack}(A \otimes B) = \begin{pmatrix} \mathrm{NoChan}(A) \\ \otimes \\ \mathrm{NoChan}(B) \end{pmatrix} \circ \begin{pmatrix} \mathrm{Channels}(\mathrm{Chan}(A) \cup \mathrm{Chan}(B)) \\ \otimes \\ 1(\mathrm{In}(A), \mathrm{In}(B)) \end{pmatrix}$$

We can write this as:

$$\begin{pmatrix} \mathrm{NoChan}(A) \\ \otimes \\ \mathrm{NoChan}(B) \end{pmatrix} \circ \begin{pmatrix} \mathrm{Channels}(\mathrm{Chan}(A)) \\ \otimes \\ 1(\mathrm{In}(A)) \\ \otimes \\ \mathrm{Channels}(\mathrm{Chan}(B)) \\ \otimes \\ 1(\mathrm{In}(B)) \end{pmatrix}$$

Crucially, we can use the fact that A and B do not overlap, in order to apply the interchange lemma, giving us:

$$\begin{matrix} \mathrm{NoChan}(A) \circ \begin{pmatrix} \mathrm{Channels}(\mathrm{Chan}(A)) \\ \otimes \\ 1(\mathrm{In}(A)) \end{pmatrix} \\ \otimes \\ \mathrm{NoChan}(B) \circ \begin{pmatrix} \mathrm{Channels}(\mathrm{Chan}(B)) \\ \otimes \\ 1(\mathrm{In}(B)) \end{pmatrix} \end{matrix}$$

Which is none other than:

$$\mathrm{SysPack}(A) \otimes \mathrm{SysPack}(B)$$

concluding our proof.

■

4 Protocols and Composition

Definition 4.1 (Protocols). A *protocol* $\mathscr{P}$ consists of:

- Systems $P_1, \ldots, P_n$, called *players*

- An asynchronous package F, called the *ideal functionality*
- A set Leakage $\subseteq \text{Out}(F)$, called the leakage

Furthermore, we also impose requirements on the channels and functions these elements use.

First, we require that the player systems are jointly closed, with no extra channels that aren't connected to other players:

$$\bigcup_{i \in [n]} \text{OutChan}(P_i) = \bigcup_{i \in [n]} \text{InChan}(P_i)$$

Second, we require that the functions the systems depend on are disjoint, outside of the ideal functionality:

$$\forall i, j \in [n]. \quad \text{In}(P_i) \cap \text{In}(P_j) \subseteq \text{Out}(F)$$

Third, we require that the functions the systems export on are disjoint:

$$\forall i, j \in [n]. \quad \text{Out}(P_i) \cap \text{Out}(P_j) = \emptyset$$

We can also define a few convenient notations related to the interface of a base protocol.

Let $\text{Out}_i(\mathscr{P}) := \text{Out}(P_i)$, and let $\text{In}_i(\mathscr{P}) := \text{In}(P_i)/\text{Out}(F)$. We then define $\text{Out}(\mathscr{P}) := \bigcup_{i \in [n]} \text{Out}_i(\mathscr{P})$ and $\text{In}(\mathscr{P}) := \bigcup_{i \in [n]} \text{In}_i(\mathscr{P})$. Let $\text{IdealIn}_i(\mathscr{P}) := \text{In}(P_i) \cap \text{Out}(F)$.

Finally, we define

$$\text{IdealIn}(\mathscr{P}) := \text{In}(F)$$

□

Definition 4.2 (Closed Protocol). We say that a protocol $\mathscr{P}$ is *closed* if $\text{In}(\mathscr{P}) = \emptyset$ and $\text{IdealIn}(\mathscr{P}) = \emptyset$.

□

Definition 4.3 (Literal Equality). Given two protocols $\mathscr{P}$ and $\mathscr{Q}$, we say that they are *literally equal*, written as $\mathscr{P} \equiv \mathscr{Q}$ when:

- $\mathscr{P}.n = \mathscr{Q}.n$
- There exists a permuation $\pi : [n] \leftrightarrow [n]$ such that $\forall i \in [n].\ \mathscr{P}.P_i \equiv \mathscr{Q}.P_{\pi(i)}$
- $\mathscr{P}.F = \mathscr{Q}.F$
- $\mathscr{P}.\text{Leakage} = \mathscr{Q}.\text{Leakage}$

□

Definition 4.4 (Vertical Composition). Given an protocol $\mathscr{P}$ and a package G, satisfying $\text{IdealIn}(\mathscr{P}) \subseteq \text{Out}(G)$, we can define the protocol $\mathscr{P} \circ G$.

$\mathscr{P} \circ G$ has the same players and leakage as $\mathscr{P}$, but its ideal functionality F becomes $F \circ G$.

□

Claim 4.1 (Vertical Composition is Associative). For any protocol $\mathscr{P}$, and packages G, H, such that their composition is well defined, we have

$$\mathscr{P} \circ (G \circ H) = (\mathscr{P} \circ G) \circ H$$

Proof: This follows from the definition of vertical composition and the associativity of $\circ$ for packages. ■

Definition 4.5 (Horizontal Composition). Given two protocols $\mathscr{P}, \mathscr{Q}$, we can define the protocol $\mathscr{P} \lhd \mathscr{Q}$, provided a few requirements hold.

First, we need: $\text{In}(\mathscr{P}) \subseteq \text{Out}(\mathscr{Q})$. We also require that the functions exposed by a player in $\mathscr{Q}$ are used by *exactly* one player in $\mathscr{P}$. We express this as:

$$\forall i \in [\mathscr{Q}.n].\ \exists! j \in [\mathscr{P}.n]. \quad \text{In}_j \cap \text{Out}_i \neq \emptyset$$

Second, we require that the players share no channels between the two protocols. In other words $\text{Chan}(\mathscr{P}.P_i) \cap \text{Chan}(\mathscr{Q}.P_j) = \emptyset$, for all P_i, P_j.

Third, we require that the ideal functionalities of one protocol aren't used in the other.

$$\begin{aligned} \text{Out}(\mathscr{P}.F) \cap \text{In}(\mathscr{Q}) = \emptyset \\ \text{Out}(\mathscr{Q}.F) \cap \text{In}(\mathscr{P}) = \emptyset \end{aligned}$$

Finally, we require that the ideal functionalities do not overlap, in the sense that $\text{Out}(\mathscr{P}.F) \cap \text{Out}(\mathscr{Q}.F) = \emptyset$

Our first condition has an interesting consequence: every player $\mathscr{Q}.P_j$ has its functions used by exactly one player $\mathscr{P}.P_i$. In that case, we say that $\mathscr{P}.P_i$ *uses* $\mathscr{Q}.P_j$.

With this in hand, we can define $\mathscr{P} \lhd \mathscr{Q}$.

The players will consist of:

$$\mathscr{P}.P_i \circ \begin{pmatrix} \displaystyle\mathop{\circledast}_{\mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i} \mathscr{Q}.P_j \\ \otimes \\ 1(\text{IdealIn}_i) \end{pmatrix}$$

And, because of our assumption, each player in $\mathscr{Q}$ appears somewhere in this equation.

The ideal functionality is $\mathscr{P}.F \otimes \mathscr{Q}.F$, and the leakage is $\mathscr{P}.\text{Leakage} \cup \mathscr{Q}.\text{Leakage}$.

We can also easily show that this definition is well defined, satisfying the required properties of an protocol. Because of the definition of the players, we see that:

$$\bigcup_{i \in [(\mathscr{P} \lhd \mathscr{Q}).n]} \text{OutChan}((\mathscr{P} \lhd \mathscr{Q}).P_i) = \left(\bigcup_{i \in [\mathscr{P}.n]} \text{OutChan}(\mathscr{P}.P_i) \right) \cup \left(\bigcup_{i \in [\mathscr{Q}.n]} \text{OutChan}(\mathscr{Q}.P_i) \right)$$

since $\text{OutChan}(A \circ B) = \text{OutChan}(A \otimes B) = \text{OutChan}(A, B)$. A similar reasoning applies to InChan, allowing us to conclude that:

$$\bigcup_{i \in [(\mathscr{P} \lhd \mathscr{Q}).n]} \text{OutChan}((\mathscr{P} \lhd \mathscr{Q}).P_i) = \bigcup_{i \in [(\mathscr{P} \lhd \mathscr{Q}).n]} \text{InChan}((\mathscr{P} \lhd \mathscr{Q}).P_i)$$

as required.

By definition, the dependencies In of each player in $\mathscr{P} \lhd \mathscr{Q}$ are the union of several players in $\mathscr{Q}$, and the ideal dependencies of players in $\mathscr{P}$, both of these are required to be disjoint, so disjointness property continues to hold.

Finally, since each player is of the form $\mathscr{P}.P_i \circ \ldots$, the condition on Out_i is also satisfied in $\mathscr{P} \lhd \mathscr{Q}$, since $\mathscr{P}$ does.

□

Lemma 4.2. Horizontal composition is associative, i.e. $\mathscr{P} \lhd (\mathscr{Q} \lhd \mathscr{R}) \equiv (\mathscr{P} \lhd \mathscr{Q}) \lhd \mathscr{R}$ for all protocols $\mathscr{P}, \mathscr{Q}, \mathscr{R}$ where this expression is well defined.

Proof: For the ideal functionalities, it's clear that by the associativity of $\otimes$ for systems, the resulting functionality is the same in both cases.

The trickier part of the proof is showing that the resulting players are identical.

It's convenient to define a relation for the players in $\mathscr{R}$ that get used in $\mathscr{P}$ via the players in $\mathscr{Q}$. To that end, we say that $\mathscr{P}.P_i$ *uses* $\mathscr{R}.P_j$ if there exists $\mathscr{Q}.P_k$ such that $\mathscr{P}.P_i$ uses $\mathscr{Q}.P_k$, and $\mathscr{Q}.P_k$ uses $\mathscr{R}.P_j$.

The players of $\mathscr{P} \lhd (\mathscr{Q} \lhd \mathscr{R})$ are of the form:

$$\mathscr{P}.P_i \circ \begin{pmatrix} \displaystyle\mathop{\ast}_{\mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i} \mathscr{Q}.P_j \circ \begin{pmatrix} \displaystyle\mathop{\ast}_{\mathscr{R}.P_k \text{ used by } \mathscr{Q}.P_j} \mathscr{R}.P_k \\ \otimes \\ 1(\mathscr{Q}.\text{IdealIn}_j) \end{pmatrix} \\ \otimes \\ 1(\mathscr{P}.\text{IdealIn}_i) \end{pmatrix}$$

While those in $(\mathscr{P} \lhd \mathscr{Q})\mathscr{R}$ are of the form:

$$\left(\mathscr{P}.P_i \circ \begin{pmatrix} \underset{\mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i}{\circledast} \mathscr{Q}.P_j \\ \otimes \\ 1(\mathscr{P}.\text{IdealIn}_i) \end{pmatrix}\right) \circ \begin{pmatrix} \underset{\mathscr{R}.P_k \text{ used by } \mathscr{P}.P_i}{\circledast} \mathscr{R}.P_k \\ \otimes \\ 1(\mathscr{Q}.\text{IdealIn}_j) \end{pmatrix}$$

Now, we can apply the associativity of $\circ$ for systems, and also group the $\mathscr{R}.P_k$ players based on which $\mathscr{Q}.P_j$ uses them:

$$\mathscr{P}.P_i \circ \begin{pmatrix} \underset{\mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i}{\circledast} \mathscr{Q}.P_j \\ \otimes \\ 1(\mathscr{P}.\text{IdealIn}_i) \end{pmatrix} \circ \left(\underset{\mathscr{Q}.P_j}{\circledast} \begin{pmatrix} \underset{\mathscr{R}.P_k \text{ used by } \mathscr{Q}.P_j}{\circledast} \mathscr{R}.P_k \\ \otimes \\ 1(\mathscr{Q}.\text{IdealIn}_j) \end{pmatrix} \right)$$

Now, the conditions are satisfied for applying the interchange lemma (Lemma 3.4), giving us:

$$\mathscr{P}.P_i \circ \begin{pmatrix} \underset{\mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i}{\circledast} \mathscr{Q}.P_j \circ \begin{pmatrix} \underset{\mathscr{R}.P_k \text{ used by } \mathscr{Q}.P_j}{\circledast} \mathscr{R}.P_k \\ \otimes \\ 1(\mathscr{Q}.\text{IdealIn}_j) \end{pmatrix} \\ \otimes \\ 1(\mathscr{P}.\text{IdealIn}_i) \end{pmatrix}$$

Which is non other than the players in $\mathscr{P} \lhd (\mathscr{Q} \lhd \mathscr{R})$.

■

Definition 4.6 (Concurrent Composition). Given two protocols $\mathscr{P}, \mathscr{Q}$, we can define their concurrent composition—or tensor product— $\mathscr{P} \otimes \mathscr{Q}$, provided a few requirements hold. We require that:

1. $\text{In}(\mathscr{P}) \cap \text{In}(\mathscr{Q}) = \emptyset$.
2. $\text{Out}(\mathscr{P}) \cap \text{Out}(\mathscr{Q}) = \emptyset$.
3. $\text{Out}(\mathscr{P}.F) \cap \text{Out}(\mathscr{Q}.F) = \emptyset$ *or* $\mathscr{P}.F = \mathscr{Q}.F$.
4. $\text{Leakage}(\mathscr{P}) \cap \text{In}(\mathscr{Q}) = \emptyset = \text{Leakage}(\mathscr{Q}) \cap \text{In}(\mathscr{P})$

The players of $\mathscr{P} \otimes \mathscr{Q}$ consist of all the players in $\mathscr{P}$ and $\mathscr{Q}$. The ideal functionality is $\mathscr{P}.F \otimes \mathscr{Q}.F$, unless $\mathscr{P}.F = \mathscr{Q}.F$, in which case the ideal functionality is simply $\mathscr{P}.F$. In either case, the leakage is $\mathscr{P}.\text{Leakage} \cup \mathscr{Q}.\text{Leakage}$. This use of $\otimes$ is well defined by assumption.

The resulting protocol is also clearly well defined.

The jointly closed property holds because we've simply taken the union of both player sets.

Since $\text{In}(\mathscr{P}) \cap \text{In}(\mathscr{Q}) = \emptyset$, it also holds that for every P_i, P_j in $\mathscr{P} \otimes \mathscr{Q}$, we have $\text{In}(P_i) \cap \text{In}(P_j) = \emptyset$, since each player comes from either $\mathscr{P}$ or $\mathscr{Q}$.

Finally, $\text{Out}(\mathscr{P}) \cap \text{Out}(\mathscr{Q}) = \emptyset$, we have that $\text{Out}(P_i) \cap \text{Out}(P_j) = \emptyset$, by the same reasoning.

□

The reason why we allow for $F = G$ is so that you can have like the same 1

Lemma 4.3. Concurrent composition is associative and commutative. I.e. $\mathscr{P} \otimes (\mathscr{Q} \otimes \mathscr{R}) \equiv (\mathscr{P} \otimes \mathscr{Q}) \otimes \mathscr{R}$, and $\mathscr{P} \otimes \mathscr{Q} \equiv \mathscr{Q} \otimes \mathscr{P}$ for all protocols $\mathscr{P}, \mathscr{Q}, \mathscr{R}$ where these expressions are well defined.

Proof:

By the definition of $\equiv$, all that matter is the *set* of players, and not their order. Because $\cup$ is associative, and so is $\otimes$ for systems, we conclude that concurrent composition is associative as well, since the resulting set of players and ideal functionality are the same in both cases.

Similarly, since $\cup$ and $\otimes$ (for systems) are commutative, we conclude that concurrenty composition is commutative.

■

4.1 Corruption and Simulation

Definition 4.7 ("Honest" Corruption). Given a system P, we define the "honest" corruption of P

$$\text{Corrupt}_H(P) := P$$

This is clearly equality preserving, by tautology.

□

Definition 4.8 (Semi-Honest Corruption). Given a system P, we can define the semi-honest corruption $\text{Corrupt}_{\text{SH}}(P)$.

This is a transformation of of P, providing access to its "view". More formally, $\text{Corrupt}_{\text{SH}}(P)$ is a system which works the same as P, but with an additional public variable log, which contains several sub logs:

1. $\text{log}.A_i$ for each sending channel A_i,
2. $\text{log}.B_i$ for each receiving channel B_i,
3. $\text{log}.F$ for each input function F.
4. $\text{log}.G$ for each output function G.

Each of these sub logs is initialized with log.$\bullet \leftarrow$ FifoQueue.New(). Additionally, $\text{Corrupt}_{\text{SH}}(P)$ modifies P by pushing events to these logs at different points in time. These events are:

- $(\texttt{call}, (x_1, \ldots, x_n))$ to log.F when a function call $F(x_1, \ldots, x_n)$ happens.
- $(\texttt{ret}, y)$ to log.F when the function F returns a value y.
- $(\texttt{input}, (x_1, \ldots, x_n))$ to log.G when the function G is called with $(x_i, \ldots)$ as input.
- m to log.A when a value m is sent on channel A.
- m to log.B when a value m is received on channel B.

This transformation is also equality respecting. First, note that if $P \equiv P'$ as systems, then then $\text{NoChan}(P) = \text{NoChan}(P')$, and so their logs will be the same.

$\square$

Definition 4.9 (Malicious Corruption). Given a system P with:

$$\begin{aligned}
\text{In}(P) &= \{F_1, \ldots, F_n\} \\
\text{OutChan}(P) &= \{A_1, \ldots, A_m\} \\
\text{InChan}(P) &= \{B_1, \ldots, B_l\}
\end{aligned}$$

we define the malicious corruption $\text{Corrupt}_M(P)$ as the following game:

$$\boxed{\begin{array}{l}
\text{Corrupt}_M(P) \\
\\
\underline{\text{Call}_{F_i}((x_1, \ldots, x_n)):} \\
\quad \textbf{return } F_i(x_1, \ldots, x_n) \\
\\
\underline{\text{Send}_{A_i}(m):} \\
\quad m \Rightarrow A_i \\
\\
\underline{\text{Test}_{B_i}():} \\
\quad \textbf{return test } B_i \\
\\
\underline{\text{Recv}_{B_i}():} \\
\quad \textbf{return } m \Leftarrow B_i
\end{array}}$$

In other words, malicious corruption provides access to the functions and channels used by P, but no more than that.

This is also equality preserving, since $\text{Corrupt}_M(P)$ depends only on the channels used by P and the functions called by P, all of which are the same for any $P' \equiv P$.

□

Lemma 4.4 (Simulating Corruptions). We can simulate corruptions using strong forms of corruption. In particular, there exists systems S_{SH} and S_{H} such that for all systems P, we have:

$$\begin{aligned}\text{Corrupt}_{\text{SH}}(P) &= S_{\text{SH}} \circ \text{Corrupt}_{M}(P) \\ \text{Corrupt}_{\text{H}}(P) &= S_{\text{H}} \circ \text{Corrupt}_{\text{SH}}(P)\end{aligned}$$

Proof: For the simulation of honest corruption, we can simply ignore the additional log variable, and set $S_{\text{H}} := 1(\text{Out}(P))$.

For semi-honest corruption, S_{SH} is formed by first transforming $\text{Corrupt}_{\text{SH}}(P)$, replacing:

- every function call with $\text{Call}_{F_i}(\ldots)$,
- every sending of a message m on A with $\text{Send}_A(m)$,
- every length test of B with $\text{Test}_B()$,
- every reception of a message on B with $\text{Recv}_B()$.

The result is clearly a perfect emulation of semi-honest corruption using malicious corruption.

■

Sometimes, it's useful to be able to talk about corruptions in general, in which case we write $\text{Corrupt}_\kappa(P)$, for $\kappa \in \{\text{H}, \text{SH}, \text{M}\}$.

Definition 4.10 (Corruption Models). Given a protocol $\mathscr{P}$ with players $P_1, \ldots, P_n$, a *corruption model* C is a function $C : [\mathscr{P}.n] \to \{\text{H}, \text{SH}, \text{M}\}$. This provides a corruption C_i associated with each player P_i. We can then define $\text{Corrupt}_C(P_i) := \text{Corrupt}_{C_i}(P_i)$.

Corruption models have a natural partial order associated with them. We have:

$$\text{H} < \text{SH} < \text{M}$$

and then we say that $C \geq C'$ if $\forall i \in [n].\quad C_i \geq C'_i$.

A *class of corruptions* $\mathscr{C}$ is simply a set of corruption models.

□

Some common classes are:

- The class of malicious corruptions, where all but one player is malicious.
- The class of malicious corruptions, where all but one player is semi-honest.

Definition 4.11 (Instantiation). Given a protocol $\mathscr{P}$ with $\text{In}(\mathscr{P}) = \emptyset$, and a corruption model C, we can define an *instantiation* $\text{Inst}_C(\mathscr{P})$, which is a system defining the semantics of the protocol.

First, we need to define a transformation of systems to use a *router* $\mathcal{R}$, which will be a special system allowing an adversary to control the order of delivery of messages.

Let $\{A_1, \ldots, A_n\} = \text{Chan}(P_1, \ldots, P_n)$. We then define $\mathcal{R}$ as the syten:

$$\boxed{\begin{array}{l} \mathcal{R} \\ \\ \underline{\text{Deliver}_{A_i}():} \\ \quad m \Leftarrow \langle A_i, \mathcal{R} \rangle \\ \quad m \Rightarrow \langle \mathcal{R}, A_i \rangle \end{array}}$$

Next, we define a transformation $\text{Routed}(S)$ of a system, which makes communication pass via the router:

- Whenever S sends m via A, $\text{Routed}(S)$ sends m via $\langle A, \mathcal{R} \rangle$.
- Whenever S receives m via B, $\text{Routed}(S)$ recieves m via $\langle \mathcal{R}, B \rangle$.

With this in hand, we define:

$$\text{Inst}_C(\mathscr{P}) := \begin{pmatrix} \circledast_{i \in [n]} \text{Routed}(\text{Corrupt}_C(P_i)) \\ * \\ \mathcal{R} \\ \otimes \\ 1(\text{Leakage}) \end{pmatrix} \circ F$$

$\square$

Lemma 4.5 (Properties of Routed). For any systems A, B, we have:

$$\begin{aligned} \text{Routed}(A \circ B) &= \text{Routed}(A) \circ \text{Routed}(B) \\ \text{Routed}(A * B) &= \text{Routed}(A) * \text{Routed}(B) \\ \text{Routed}(A \otimes B) &= \text{Routed}(A) \otimes \text{Routed}(B) \end{aligned}$$

(provided these expressions are well defined)

Proof: The Routed transformation simply renames each sending and receiving channel in a system. In all the cases above, even $A * B$, all of the channels present in A and B are present in the composition, and so all of these equations hold.

$\blacksquare$

Definition 4.12 (Compatible Corruptions). Given protocols $\mathscr{P}, \mathscr{Q}$, and a corruption model C for $\mathscr{Q}$, we can define a notion of a *compatible* corruption model C' for $\mathscr{P} \otimes \mathscr{Q}$ or $\mathscr{P} \circ \mathscr{Q}$, provided these expressions are well defined.

A corruption model C' for $\mathscr{P} \otimes \mathscr{Q}$. is compatible with C when every corruption of a player in $\mathscr{Q}$ is $\geq$ that of the corresponding corruption in C.

We say that a corruption model C' for $\mathscr{P} \circ \mathscr{Q}$ is compatible with a corruption model C for $\mathscr{Q}$ if for every $\mathscr{Q}.P_j$ used by $\mathscr{P}.P_i$, the corruption level of $\mathscr{Q}.P_j$ in $\mathscr{C}'$ is $\geq$ the corruption level of $\mathscr{P}.P_i$ in $\mathscr{C}$.

Furthermore, we say that C' is *strictly* compatible with C if the above property holds with $=$, and not just $\geq$.

This extends to corruption *classes* as well. A corruption class $\mathscr{C}'$ is (strictly) compatible with a class $\mathscr{C}$, if every $C' \in \mathscr{C}'$ is (strictly) compatible with some $C \in \mathscr{C}$.

□

Theorem 4.6 (Concurrent Breakdown). Given protocols $\mathscr{P}, \mathscr{Q}$, and a corruption model C for $\mathscr{Q}$, then for any corruption model C' for $\mathscr{P} \otimes \mathscr{Q}$ compatible with C, we have:

$$\text{Inst}_{C'}(\mathscr{P} \otimes \mathscr{Q}) = \text{Inst}_{C'}(\mathscr{P}) \otimes \text{Inst}_{C}(\mathscr{Q})$$

Proof: If we unroll $\text{Inst}_{C'}(\mathscr{P} \otimes \mathscr{Q})$, we get:

$$\begin{pmatrix} \mathcal{R} \\ * \\ \left(*_{i \in [\mathscr{P}.n]} \text{Routed}(\text{Corrupt}_{C'}(\mathscr{P}.P_i))\right) \\ * \\ \left(*_{i \in [\mathscr{Q}.n]} \text{Routed}(\text{Corrupt}_{C'}(\mathscr{Q}.P_i))\right) \\ \otimes \\ 1(\mathscr{P}.\text{Leakage}, \mathscr{Q}.\text{Leakage}) \end{pmatrix} \circ \begin{pmatrix} \mathscr{P}.F \\ \otimes \\ \mathscr{Q}.F \end{pmatrix}$$

We can apply a few observations here:

1. Since $\mathscr{C}'$ is compatible with $\mathscr{C}$, then $\mathscr{Q}.P_i$ follows a corruption from $\mathscr{C}$.
2. $\mathcal{R}$ can be written as $\mathcal{R}_{\mathscr{P}} \otimes \mathcal{R}_{\mathscr{Q}}$, with one system using channels in $\mathscr{P}$, and the other using channels in $\mathscr{Q}$.
3. Since protocols are closed, we can use $\otimes$ between the players in $\mathscr{P}$ and $\mathscr{Q}$, since they never send messages to each other.

This results in the following:

$$\begin{pmatrix} \mathcal{R}_{\mathscr{P}} * \left(*_{i \in [\mathscr{P}.n]} \text{Routed}(\text{Corrupt}_{C'}(\mathscr{P}.P_i)) \right) \otimes 1(\mathscr{P}.\text{Leakage}) \\ \otimes \\ \mathcal{R}_{\mathscr{Q}} * \left(*_{i \in [\mathscr{Q}.n]} \text{Routed}(\text{Corrupt}_{C}(\mathscr{Q}.P_i)) \right) \otimes 1(\mathscr{Q}.\text{Leakage}) \end{pmatrix} \circ \begin{pmatrix} \mathscr{P}.F \\ \otimes \\ \mathscr{Q}.F \end{pmatrix}$$

From here, we apply Lemma 3.4 (interchange), to get:

$$\begin{matrix} \text{Inst}_{C'}(\mathscr{P}) \\ \otimes \\ \text{Inst}_{C}(\mathscr{Q}) \end{matrix}$$

■

Theorem 4.7 (Horizontal Breakdown). Given protocols $\mathscr{P}, \mathscr{Q}$, and a corruption model C for $\mathscr{Q}$, then for any compatible corruption model C' for $\mathscr{P} \lhd \mathscr{Q}$, there exists systems $S_1, \ldots, S_{\mathscr{Q}.n}$ and a set $L_{\mathscr{Q}}$ such that:

$$\text{Inst}_{C'}(\mathscr{P} \lhd \mathscr{Q}) = 1(O) \circ \begin{pmatrix} \bigast_{i \in [\mathscr{P}.n]} \text{Routed}(\text{Corrupt}'_{C'}(\mathscr{P}.P_i)) \\ * \\ \mathcal{R}_{\mathscr{P}} \\ \otimes \\ 1(\text{Leakage}, L_{\mathscr{Q}}) \end{pmatrix} \circ \begin{pmatrix} \mathscr{P}.F \\ \otimes \\ 1(\text{Out}(\mathcal{R}_q)) \\ \otimes \\ 1(\mathscr{Q}.\text{Leakage}) \\ \otimes \\ \bigotimes_{i \in [\mathscr{Q}.n]} S_i \end{pmatrix} \circ \begin{pmatrix} \text{Inst}_C(\mathscr{Q}) \\ \otimes \\ 1(\text{In}(\mathscr{P}.F)) \end{pmatrix}$$

where $O := \text{Out}(\text{Inst}_{C'}(\mathscr{P} \lhd \mathscr{Q}))$, $\mathcal{R}_{\mathscr{P}} \circ \mathcal{R}_{\mathscr{Q}} = \mathcal{R}$ are a decomposition of the router $\mathcal{R}$ for $\mathscr{P} \lhd \mathscr{Q}$, and $\text{Corrupt}'_{C'}(\ldots)$ is the same as $\text{Corrupt}_{C'}$, except that malicious corruption contains no Call_{F_i} functions, for $F_i \notin \text{Out}(\mathscr{P}.F)$

Furthermore, if the models are *strictly* compatible, then $S_j = 1(\text{Out}(\text{Routed}(\text{Corrupt}_C(\mathscr{Q}.P_i))))$.

Proof: We start by unrolling $\text{Inst}_{C'}(\mathscr{P} \lhd \mathscr{Q})$, to get:

$$\text{Inst}_C(\mathscr{P} \lhd \mathscr{Q}) = \begin{pmatrix} \bigast_{i \in [\mathscr{P}.n]} \text{Routed}\left(\text{Corrupt}_{C'}\left(\mathscr{P}.P_i \circ \begin{pmatrix} *_{\mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i} \mathscr{Q}.P_j \\ \otimes \\ 1(\text{IdealIn}_i) \end{pmatrix} \right) \right) \\ * \\ \mathcal{R} \\ \otimes \\ 1(\text{Leakage}) \end{pmatrix} \circ \begin{pmatrix} \mathscr{P}.F \\ \otimes \\ \mathscr{Q}.F \end{pmatrix}$$

Our strategy will be to progressively build up an equivalent system to this one, starting with Corrupt_C, then Routed, etc.

First, some observations about $\text{Corrupt}_\kappa(P \circ (1(I) \otimes Q_1 * \cdots * Q_m))$, where $I \cap \text{In}(Q_1, \ldots) = \emptyset$.

In the case of malicious corruption, we have:

$$\mathrm{Corrupt}_M(P \circ (1(I) \otimes Q_1 * \cdots)) = 1(O) \circ \begin{pmatrix} \mathrm{Corrupt}'_M(P) \\ \otimes \\ 1(\mathrm{Out}(\mathrm{Corrupt}_M(Q_1)), \ldots) \end{pmatrix} \circ \begin{pmatrix} 1(I) \\ \otimes \\ \mathrm{Corrupt}_M(Q_1) \\ * \\ \cdots \end{pmatrix}$$

for $O = \mathrm{Out}(\mathrm{Corrupt}_M(P \circ (Q_1 * \cdots)))$. This holds by definition, since corruption $P \circ (Q_1 * \cdots)$ precisely allows sending messages on behalf of P or any Q_i, as well as calling the input functions to the Q_i systems. Since we can't call the functions that P uses, we use $\mathrm{Corrupt}'_M$, which modifies malicious corruption to only contain Send_{A_i}, Test_{B_i}, Recv_{B_i}, and Call_{F_i} for $F_i \in I$. In particular the $\mathrm{Call}_\bullet$ functions are omitted for the functions provided by $Q_1, \ldots, Q_m$. We can write this expression more concisely, using $1(L^M)$ for $L^M = \mathrm{Out}(\mathrm{Corrupt}_M(Q_1)) \cup \cdots$.

Next, we look at semi-honest corruption. Because the logs are divided into independent sub logs, we can write:

$$\mathrm{Corrupt}_{\mathrm{SH}}(P \circ (1(I) \otimes Q_1 * \cdots)) = 1(O) \circ \begin{pmatrix} \mathrm{Corrupt}_{\mathrm{SH}}(P) \\ \otimes \\ 1(\{Q_1.\mathrm{log}, \ldots\}) \end{pmatrix} \circ \begin{pmatrix} 1(I) \\ \otimes \\ \mathrm{Corrupt}_{\mathrm{SH}}(Q_1) \\ * \\ \cdots \end{pmatrix}$$

where $O = \mathrm{Out}(\mathrm{Corrupt}_{\mathrm{SH}}(P \circ (Q_1 * \cdots)))$

And for honest corruption, we have

$$\mathrm{Corrupt}_{\mathrm{H}}(P \circ (1(I) \otimes Q_1 * \cdots)) = P \circ (1(I) \otimes Q_1 * \cdots)$$

Now, the compatibility condition of C' relative to C does not guarantee that if $\mathscr{P}.P_i$ uses $\mathscr{Q}.P_j$, then $\mathscr{Q}.P_j$ has the same level of corruption: it only guarantees a level of corruption at least as strong. By Lemma 4.10, we can simulate a weaker form of corruption using a stronger form, via some simulator system S, depending on the levels of corruption.

Using these simulators, we get, slightly different results based on the level of corruption.

When $C'_i = \mathrm{M}$:

$$\mathrm{Corrupt}_{C'}((\mathscr{P} \lhd \mathscr{Q}).P_i) = 1(O_i) \circ \begin{pmatrix} \mathrm{Corrupt}'_{C'}(\mathscr{P}.P_i) \\ \otimes \\ 1(L_i) \end{pmatrix} \circ \begin{pmatrix} \displaystyle\mathop{\Large *}_{\mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i} \mathrm{Corrupt}_C(\mathscr{Q}.P_j) \\ \otimes \\ 1(\mathrm{IdealIn}_i) \end{pmatrix}$$

with $O_i = \mathrm{Out}(\mathrm{Corrupt}_{C'}(\mathscr{P} \triangleleft \mathscr{Q}).P_i)$, $L_i = \bigcup_{\mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i} \mathrm{Out}(\mathrm{Corrupt}_M(\mathscr{Q}.P_j))$. No simulation is needed, since the compatibility of C' with C guarantees that all of the players used by $\mathscr{P}.P_i$ are maliciously corrupted.

When $C'_i = \mathrm{SH}$:

$$\mathrm{Corrupt}_{C'}((\mathscr{P} \triangleleft \mathscr{Q}).P_i) = 1(O_i) \circ \begin{pmatrix} \mathrm{Corrupt}_{C'}(P) \\ \otimes \\ 1(L_i) \end{pmatrix} \circ \begin{pmatrix} \bigast_{\mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i} S_j \circ \mathrm{Corrupt}_C(\mathscr{Q}.P_j) \\ \otimes \\ 1(\mathrm{IdealIn}_i) \end{pmatrix}$$

with $O_i = \mathrm{Out}(\mathrm{Corrupt}_{C'}(\mathscr{P} \triangleleft \mathscr{Q}).P_i)$, $L_i = \{\mathscr{Q}.P_j.\mathrm{log} \mid \mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i\}$, and S_j depending on the level of corruption for $\mathscr{Q}.P_j$ in C:

- $S_j = S_{\mathrm{SH}}$ if $C_j = \mathrm{M}$
- $S_j = 1$ if $C_j = \mathrm{SH}$

When $C'_i = \mathrm{H}$:

$$\mathrm{Corrupt}_{C'}((\mathscr{P} \triangleleft \mathscr{Q}).P_i) = \mathrm{Corrupt}_{C'}(P) \circ \begin{pmatrix} \bigast_{\mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i} S_j \circ \mathrm{Corrupt}_C(\mathscr{Q}.P_j) \\ \otimes \\ 1(\mathrm{IdealIn}_i) \end{pmatrix}$$

with S_j depending on the level of corruption for $\mathscr{Q}.P_j$ in C:

- $S_j = S_{\mathrm{H}} \circ S_{\mathrm{SH}}$ if $C_j = \mathrm{M}$
- $S_j = S_{\mathrm{H}}$ if $C_j = \mathrm{SH}$
- $S_j = 1$ if $C_j = \mathrm{H}$

We can unify these three cases, writing:

$$\mathrm{Corrupt}'_{C'}((\mathscr{P} \triangleleft \mathscr{Q}).P_i) = 1(O_i) \circ \begin{pmatrix} \mathrm{Corrupt}_{C'}(P) \\ \otimes \\ 1(L_i) \end{pmatrix} \circ \begin{pmatrix} \ast_{\mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i} S_j \circ \mathrm{Corrupt}_C(\mathscr{Q}.P_j) \\ \otimes \\ 1(\mathrm{IdealIn}_i) \end{pmatrix}$$

with O_i and L_i depending on the corruption level of $\mathscr{P}.P_i$, and S_j depending on the corruption levels of both $\mathscr{P}.P_i$ and $\mathscr{Q}.P_j$.

By the properties of Routed (Lemma 4.5), we have:

$$\mathrm{Routed}(\mathrm{Corrupt}'_{C'}((\mathscr{P} \triangleleft \mathscr{Q}).P_i)) =$$

$$1(O_i) \circ \begin{pmatrix} \mathrm{Routed}(\mathrm{Corrupt}'_{C'}(P)) \\ \otimes \\ 1(L_i) \end{pmatrix} \circ \begin{pmatrix} \bigast_{\mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i} S_j \circ \mathrm{Routed}(\mathrm{Corrupt}_C(\mathscr{Q}.P_j)) \\ \otimes \\ 1(\mathrm{IdealIn}_i) \end{pmatrix}$$

Next, we need to add the router $\mathcal{R}$. We note that since $\mathscr{P}$ and $\mathscr{Q}$ have separate channels, we can write $\mathcal{R} = \mathcal{R}_{\mathscr{P}} \circ \mathcal{R}_{\mathscr{Q}}$, where the latter contains only the channels in $\mathscr{Q}$, and the former contains the channels in $\mathscr{P}$, and provides access to those in $\mathscr{Q}$ via its function dependencies. Combing this with the interchange lemma, we get:

$$\mathcal{R} * \mathop{\Large *}_{i \in [\mathscr{P}.n]} \text{Routed}(\text{Corrupt}'_{C'}((\mathscr{P} \lhd \mathscr{Q}).P_i)) * \mathcal{R} =$$

$$1(\text{Out}(\mathcal{R}), O_1, \ldots, O_{\mathscr{P}.n}) \circ \begin{pmatrix} \text{Routed}(\text{Corrupt}_{C'}(P)) \\ * \\ \mathcal{R}_{\mathscr{P}} \\ \otimes \\ 1(L_1, \ldots, L_{\mathscr{P}.n}) \end{pmatrix} \circ \begin{pmatrix} *_{j \in [\mathscr{Q}.n]} S_j \circ \text{Routed}(\text{Corrupt}_C(\mathscr{Q}.P_j)) \\ * \\ \mathcal{R}_{\mathscr{Q}} \\ \otimes \\ 1(\text{Out}(F)) \end{pmatrix}$$

All that remains is to add the ideal functionalities, giving us, after application of the interchange lemma:

$$\text{Inst}_{C'}(\mathscr{P} \lhd \mathscr{Q}) =$$

$$1(O) \circ \begin{pmatrix} \text{Routed}(\text{Corrupt}'_{C'}(P)) \\ * \\ \mathcal{R}_{\mathscr{P}} \\ \otimes \\ 1(\text{Leakage}, L_{\mathscr{Q}}) \end{pmatrix} \circ \begin{pmatrix} *_{j \in [\mathscr{Q}.n]} S_j \circ \text{Routed}(\text{Corrupt}_C(\mathscr{Q}.P_j)) \\ * \\ \mathcal{R}_{\mathscr{Q}} \\ \otimes \\ 1(\text{Leakage}, \text{Out}(F)) \end{pmatrix} \circ \begin{pmatrix} \mathscr{P}.F \\ \otimes \\ \mathscr{Q}.F \end{pmatrix}$$

with $O := \text{Out}(\text{Inst}_{C'}(\mathscr{P} \lhd \mathscr{Q}))$, and $L_{\mathscr{Q}} := \bigcup_{i \in [\mathscr{P}.n]} L_i$.

Now, because $\mathscr{Q}$ does not use any of the functions in $\mathscr{P}.F$, and because each simulator S_j does not use any channels, we can rewrite this as:

$$1(O) \circ \begin{pmatrix} \text{Routed}(\text{Corrupt}'_{C'}(P)) \\ * \\ \mathcal{R}_{\mathscr{P}} \\ \otimes \\ 1(\text{Leakage}, L_{\mathscr{Q}}) \end{pmatrix} \circ \begin{pmatrix} \mathscr{P}.F \\ \otimes \\ 1(\text{Out}(\mathcal{R}_{\mathscr{Q}})) \\ \otimes \\ 1(\mathscr{Q}.\text{Leakage}) \\ \otimes \\ \bigotimes_{j \in [\mathscr{Q}.n]} S_j \end{pmatrix} \circ \begin{pmatrix} \begin{pmatrix} *_{j \in [\mathscr{Q}.n]} \text{Routed}(\text{Corrupt}_C(\mathscr{Q}.P_j)) \\ * \\ \mathcal{R}_{\mathscr{Q}} \\ \otimes \\ 1(\mathscr{Q}.\text{Leakage}) \end{pmatrix} \circ \mathscr{Q}.F \\ \otimes \\ 1(\text{In}(\mathscr{P}.F)) \end{pmatrix}$$

We can then notice that the right hand side of this equation is simply $\text{Inst}_C(\mathscr{Q})$, concluding our proof.

■

4.2 Equality and Simulation

Definition 4.13 (Shape). We say that two protocols $\mathscr{P}, \mathscr{Q}$ have the same *shape* if there exists a protocol $\mathscr{Q}' \equiv \mathscr{Q}$ such that:

- $\mathscr{P}.n = \mathscr{Q}'.n$,
- $\forall i \in [n]. \quad \text{In}(\mathscr{P}.P_i) = \text{In}(\mathscr{Q}'.Q_i)$,
- $\forall i \in [n]. \quad \text{Out}(\mathscr{P}.P_i) = \text{Out}(\mathscr{Q}'.Q_i)$,
- $\text{Leakage}(\mathscr{P}) = \text{Leakage}(\mathscr{Q}')$,
- $\text{IdealIn}(\mathscr{P}) = \text{IdealIn}(\mathscr{Q}')$.

□

Definition 4.14 (Semantic Equality). We say that two closed protocols $\mathscr{P}$ and $\mathscr{Q}$, with the same shape, are equal under a class of corruptions $\mathscr{C}$, written as $\mathscr{P} =_{\mathscr{C}} \mathscr{Q}$, when we have:

$$\forall C \in \mathscr{C}. \quad \text{Inst}_C(\mathscr{P}) = \text{Inst}_C(\mathscr{Q})$$

as systems, with $\mathscr{Q}' \equiv \mathscr{Q}$ as per Definition 4.13.

□

Definition 4.15 (Indistinguishability). We say that two closed protocols $\mathscr{P}$ and $\mathscr{Q}$, with the same shape, are *indistinguishable* up to ϵ under a class of corruptions $\mathscr{C}$, written as $\mathscr{P} \overset{\epsilon}{\approx}_{\mathscr{C}} \mathscr{Q}$, when we have:

$$\forall C \in \mathscr{C}. \quad \text{Inst}_C(\mathscr{P}) \overset{\epsilon}{\approx} \text{Inst}_C(\mathscr{Q})$$

as systems, with $\mathscr{Q}' \equiv \mathscr{Q}$ as per Definition 4.13.

□

Definition 4.16 (Simulated Instantiation). A simulator S for a closed protocol $\mathscr{P}$ under a corruption model C is a system satisfying:

- $\text{InChan}(S), \text{OutChan}(S) = \emptyset$,
- $\text{In}(S) = \text{Leakage} \cup \left(\bigcup_{C_i = \text{M}} \text{Out}(\text{Corrupt}_{\text{M}}(P_i))\right) \cup \left(\bigcup_{C_i = \text{SH}} P_i.\text{log}\right)$,
- $\text{Out}(S) = \text{In}(S)$,

Given such a simulator, we can define the simulated instantiation of $\mathscr{P}$ under C with S as:

$$\text{SimInst}_{S,C}(\mathscr{P}) := \begin{pmatrix} S \\ \otimes \\ 1(\text{Out}(\text{Inst}_C(\mathscr{P}))/\text{Out}(S)) \end{pmatrix} \circ \text{Inst}_C(\mathscr{P})$$

□

Definition 4.17 (Simulatability). Given closed protocols $\mathscr{P}, \mathscr{Q}$ with the same shape, we say that $\mathscr{P}$ is *simulatable* up to ϵ by $\mathscr{Q}$ under a class of corruptions $\mathscr{C}$, written as $\mathscr{P} \overset{\epsilon}{\rightsquigarrow}_{\mathscr{C}} \mathscr{Q}$, when:

$$\forall C \in \mathscr{C}.\exists S. \quad \text{Inst}_C(\mathscr{P}) \overset{\epsilon}{\approx} \text{SimInst}_{S,C}(\mathscr{Q}')$$

as systems, with $\mathscr{Q}' \equiv \mathscr{Q}$ as per Definition 4.13.

□

Theorem 4.8 (Equality Hierarchy). For any corruption class $\mathscr{C}$, we have:

1. $\mathscr{P} \equiv \mathscr{Q} \implies \mathscr{P} =_{\mathscr{C}} \mathscr{Q}$.
2. $\mathscr{P} =_{\mathscr{C}} \mathscr{Q} \implies \mathscr{P} \overset{0}{\approx}_{\mathscr{C}} \mathscr{Q}$.
3. $\mathscr{P} \overset{\epsilon}{\approx}_{\mathscr{C}} \mathscr{Q} \implies \mathscr{P} \overset{\epsilon}{\rightsquigarrow}_{\mathscr{C}} \mathscr{Q}$.

Proof:

1. For any $C \in \mathscr{C}$, Corrupt_C and Routed are equality respecting, so we have:

$$\forall i \in [n]. \quad \text{Routed}(\text{Corrupt}_C(\mathscr{P}.P_i)) = \text{Routed}(\text{Corrupt}_C(\mathscr{Q}.P_i))$$

Furthermore, the equality of players between $\mathscr{P}$ and $\mathscr{Q}$ makes $\mathscr{P}.\mathcal{R} = \mathscr{Q}.\mathcal{R}$.

And then, the fact that $\mathscr{P}.F = \mathscr{Q}.F$ forces Leakage to be the same as well.

Finally, since $\circ, *, \otimes$ are respect $\equiv$, we can clearly see that $\text{Inst}_C(\mathscr{P}) = \text{Inst}_C(\mathscr{Q})$, since all the sub-components are literally equal.

2. For any systems A, B, we have $A = B \implies A \overset{0}{\approx} B$. Applying this to $\text{Inst}_C(\mathscr{P})$ and $\text{Inst}_C(\mathscr{Q})$ gives us our result.

3. It suffices to define a simulator S such that $\text{SimInst}_{S,C}(\mathscr{Q}) = \text{Inst}_C(\mathscr{Q})$, which will then show our result. We can simply take $S = 1(\ldots)$ for the right set.

■

Theorem 4.9 (Transitivity of Equality). For any closed protocols $\mathscr{L}, \mathscr{P}, \mathscr{Q}$ with the same shape, and any class of corruptions $\mathscr{C}$, we have:

1. $\mathscr{L} =_{\mathscr{C}} \mathscr{P}, \mathscr{P} =_{\mathscr{C}} \mathscr{Q} \implies \mathscr{L} =_{\mathscr{C}} \mathscr{Q}$,
2. $\mathscr{L} \overset{\epsilon_1}{\approx}_{\mathscr{C}} \mathscr{P}, \mathscr{P} \overset{\epsilon_2}{\approx}_{\mathscr{C}} \mathscr{Q} \implies \mathscr{L} \overset{\epsilon_1+\epsilon_2}{\approx}_{\mathscr{C}} \mathscr{Q}$,
3. $\mathscr{L} \overset{\epsilon_1}{\rightsquigarrow}_{\mathscr{C}} \mathscr{P}, \mathscr{P} \overset{\epsilon_2}{\rightsquigarrow}_{\mathscr{C}} \mathscr{Q} \implies \mathscr{L} \overset{\epsilon_1+\epsilon_2}{\rightsquigarrow}_{\mathscr{C}} \mathscr{Q}$.

Proof: The first two parts follow directly from Lemma 3.5 (transitivity for system equality). Indeed, we just look at $\text{Inst}_C(\mathscr{L})$, $\text{Inst}_C(\mathscr{P})$, and $\text{Inst}_C(\mathscr{Q})$ as systems, for any corruption model C.

For part 3, by assumption we have, for any $C \in \mathscr{C}$:

- $\text{Inst}_C(\mathscr{L}) \overset{\epsilon_1}{\approx} \begin{pmatrix} S_1 \\ \otimes \\ 1(O) \end{pmatrix} \text{Inst}_C(\mathscr{P})$,
- $\text{Inst}_C(\mathscr{P}) \overset{\epsilon_1}{\approx} \begin{pmatrix} S_2 \\ \otimes \\ 1(O) \end{pmatrix} \text{Inst}_C(\mathscr{Q})$.

This means that:

$$\text{Inst}_C(\mathscr{L}) \overset{\epsilon_1+\epsilon_2}{\approx} \begin{pmatrix} S_1 \\ \otimes \\ 1(O) \end{pmatrix} \circ \begin{pmatrix} S_2 \\ \otimes \\ 1(O) \end{pmatrix} \circ \text{Inst}_C(\mathscr{Q})$$

applying the properties we have for systems.

Then, we can apply interchange to write this as:

$$\begin{pmatrix} S_1 \circ S_2 \\ \otimes \\ 1(O) \end{pmatrix} \circ \text{Inst}_C(\mathscr{Q})$$

which concludes our proof, since $S_1 \circ S_2$ will be a valid simulator.

■

Theorem 4.10 (Malicious Completeness). Let $\mathscr{P}$ and $\mathscr{Q}$ closed protocols with the same shape. Given any class of corruptions $\mathscr{C}$, let $\mathscr{C}'$ be a related class, containing models in $\mathscr{C}$ with some malicious corruptions replaced with semi-honest corruptions. We then have:

1. $\mathscr{P} =_{\mathscr{C}} \mathscr{Q} \implies \mathscr{P} =_{\mathscr{C}'} \mathscr{Q}$,
2. $\mathscr{P} \overset{\epsilon}{\approx}_{\mathscr{C}} \mathscr{Q} \implies \mathscr{P} \overset{\epsilon}{\approx}_{\mathscr{C}'} \mathscr{Q}$,

Furthermore, if for any $C \in \mathscr{C}$ and its related model $C' \in \mathscr{C}$, there exists a simulator S_{M} such that $\text{Inst}_C(\mathscr{Q}) = \text{SimInst}_{S_{\text{M}},C'}(\mathscr{Q})$, then it additionally holds that:

3. $\mathscr{P} \overset{\epsilon}{\leadsto}_{\mathscr{C}} \mathscr{Q} \implies \mathscr{P} \overset{\epsilon}{\leadsto}_{\mathscr{C}'} \mathscr{Q}$

Proof: Lemma (simulating corruptions) is the crux of our proof. It implies that there existts a system S_{SH} such that

$$\text{Corrupt}_{\text{SH}}(P) = S_{\text{SH}} \circ \text{Corrupt}_M(P)$$

As a consequence, for any $C' \in \mathscr{C}'$ and the $C \in \mathscr{C}$ it's related to, there exists a *simulator* S_{SH} such that:

$$\text{Inst}_{C'}(\mathscr{P}) = \begin{pmatrix} S_{\text{SH}} \\ \otimes \\ 1(O) \end{pmatrix} \circ \text{Inst}_C(\mathscr{P})$$

which simulates all of the semi-honest corruptions in C' from the malicious ones in C.

This immediately implies parts 1 and 2, by the fact that $\circ$ for systems respects equality and indistinguishability.

For part 3, we apply the assumption in the implication to get:

$$\begin{pmatrix} S_{\text{SH}} \\ \otimes \\ 1(O) \end{pmatrix} \circ \begin{pmatrix} S \\ \otimes \\ 1(O) \end{pmatrix} \circ \text{Inst}_C(\mathscr{Q})$$

Then, apply our assumption about being able to simulate malicious corruption from semi-honest corruption to get:

$$\begin{pmatrix} S_{\text{SH}} \\ \otimes \\ 1(O) \end{pmatrix} \circ \begin{pmatrix} S \\ \otimes \\ 1(O) \end{pmatrix} \circ \begin{pmatrix} S_M \\ \otimes \\ 1(O) \end{pmatrix} \circ \text{Inst}_{C'}(\mathscr{Q})$$

which we can then apply interchange to to end up with:

$$\begin{pmatrix} S_{\text{SH}} \circ S \circ S_M \\ \otimes \\ 1(O) \end{pmatrix} \circ \text{Inst}_{C'}(\mathscr{Q}) = \text{SimInst}_{S',C'}(\mathscr{Q})$$

for $S' := S_{\text{SH}} \circ S \circ S_M$, concluding our proof.

■

Theorem 4.11 (Vertical Composition Theorem). For any protocol $\mathscr{P}$ and game G, such that $\mathscr{P} \circ G$ is well defined and closed, and for any corruption class $\mathscr{C}$, we have:

1. $G = G' \implies \mathscr{P} \circ G =_{\mathscr{C}} \mathscr{P} \circ G'$
2. $G \overset{\epsilon}{\approx} G' \implies \mathscr{P} \circ G \overset{\epsilon}{\approx}_{\mathscr{C}} \mathscr{P} \circ G'$

Proof: We start by noting that $\text{Inst}_C(\mathscr{P} \circ G) = A \circ F \circ G$, for some system A. Part 1 follows immediately from this, since $\circ$ is equality respecting.

Part 2 follows by applying Lemma **??**, which entails that for any system S, we have $S \circ G \overset{\epsilon}{\approx} S \circ G'$.

■

Theorem 4.12 (Concurrent Composition Theorem). Let $\mathscr{P}, \mathscr{Q}$ be protocols, with $\mathscr{P} \otimes \mathscr{Q}$ well defined and closed. For any compatible corruption classes $\mathscr{C}, \mathscr{C}'$ it holds that:

1. $\mathscr{Q} =_{\mathscr{C}} \mathscr{Q}' \implies \mathscr{P} \otimes \mathscr{Q} =_{\mathscr{C}'} \mathscr{P} \otimes \mathscr{Q}'$

2. $\mathscr{Q} \overset{\epsilon}{\approx}_{\mathscr{C}} \mathscr{Q}' \implies \mathscr{P} \otimes \mathscr{Q} \overset{\epsilon}{\approx}_{\mathscr{C}'} \mathscr{P} \otimes \mathscr{Q}'$

3. $\mathscr{Q} \overset{\epsilon}{\rightsquigarrow}_{\mathscr{C}} \mathscr{Q}' \implies \mathscr{P} \otimes \mathscr{Q} \overset{\epsilon}{\rightsquigarrow}_{\mathscr{C}'} \mathscr{P} \otimes \mathscr{Q}'$

Proof: Theorem 4.6 (concurrent breakdown) will be essential to our proof. This implies that $\forall C \in \mathscr{C}$, then for any compatible $C' \in \mathscr{C}'$ we have:

$$\text{Inst}_{C'}(\mathscr{P} \otimes \mathscr{Q}) = \text{Inst}_{C'}(\mathscr{P}) \otimes \text{Inst}_{C}(\mathscr{Q})$$

1. Since $\mathscr{Q} =_{\mathscr{C}} \mathscr{Q}'$, we have $\forall C \in \mathscr{C}$. $\text{Inst}_C(\mathscr{Q}) = \text{Inst}_C(\mathscr{Q}')$. Now, consider any $C' \in \mathscr{C}'$. By our assumption that $\mathscr{C}'$ is compatible with $\mathscr{C}$, there exists a $C \in \mathscr{C}$ that C' is compatible with. Using concurrent breakdown, we then have:

$$\text{Inst}_{C'}(\mathscr{P} \otimes \mathscr{Q}) = \text{Inst}_{C'}(\mathscr{P}) \otimes \text{Inst}_{C}(\mathscr{Q})$$

Then, since $\mathscr{Q} =_{\mathscr{C}} \mathscr{Q}'$, we have:

$$\text{Inst}_{C'}(\mathscr{P}) \otimes \text{Inst}_{C}(\mathscr{Q}) = \text{Inst}_{C'}(\mathscr{P}) \otimes \text{Inst}_{C}(\mathscr{Q}') = \text{Inst}_{C'}(\mathscr{P} \otimes \mathscr{Q}')$$

concluding our proof.

2. The proof here is similar to part 1. For any $C' \in \mathscr{C}'$, there exists a compatible $C \in \mathscr{C}$, and then we get:

$$\text{Inst}_{C'}(\mathscr{P} \otimes \mathscr{Q}) = \text{Inst}_{C'}(\mathscr{P}) \otimes \text{Inst}_{C}(\mathscr{Q})$$

Since $\mathscr{Q} \overset{\epsilon}{\approx}_{\mathscr{C}} \mathscr{Q}'$, we have:

$$\text{Inst}_{C'}(\mathscr{P}) \otimes \text{Inst}_{C}(\mathscr{Q}) \overset{\epsilon}{\approx} \text{Inst}_{C'}(\mathscr{P}) \otimes \text{Inst}_{C}(\mathscr{Q}')$$

since $\otimes$ for systems respects this operation. We can then conclude with

$$\text{Inst}_{C'}(\mathscr{P}) \otimes \text{Inst}_{C}(\mathscr{Q}') = \text{Inst}_{C'}(\mathscr{P} \otimes \mathscr{Q}')$$

3. Once more, for any $C' \in \mathscr{C}'$, there exists a compatible $C \in \mathscr{C}$ giving us:

$$\text{Inst}_{C'}(\mathscr{P} \otimes \mathscr{Q}) = \text{Inst}_{C'}(\mathscr{P}) \otimes \text{Inst}_{C}(\mathscr{Q})$$

We then apply our assumption that $\mathscr{Q} \overset{\epsilon}{\rightsquigarrow}_{\mathscr{C}} \mathscr{Q}'$ to get:

$$\text{Inst}_{C'}(\mathscr{P}) \otimes \text{Inst}_{C}(\mathscr{Q}) \overset{\epsilon}{\approx} \text{Inst}_{C'}(\mathscr{P}) \otimes ((S \otimes 1(\ldots)) \circ \text{Inst}_{C}(\mathscr{Q}'))$$

Next, we apply interchange to get:

$$\begin{matrix} 1(\text{Out}(\text{Inst}_{C'}(\mathscr{P}))) \circ \text{Inst}_{C'}(\mathscr{P}) \\ \otimes \\ ((S \otimes 1(\ldots)) \circ \text{Inst}_{C}(\mathscr{Q}')) \end{matrix} = \begin{pmatrix} 1(\text{Out}(\text{Inst}_{C'}(\mathscr{P}))) \\ \otimes \\ S \\ \otimes \\ 1(\text{Out}(\text{Inst}_{C}(\mathscr{Q}))/\text{Out}(S)) \end{pmatrix} \circ \begin{pmatrix} \text{Inst}_{C'}(\mathscr{P}) \\ \otimes \\ \text{Inst}_{C}(\mathscr{Q}') \end{pmatrix}$$

Applying concurrent breakdown in reverse, we get that the right hand side is $\text{Inst}_{C'}(\mathscr{P} \otimes \mathscr{Q})$, and that the left hand side is the simulator showing that $\mathscr{P} \otimes \mathscr{Q} \overset{\epsilon}{\rightsquigarrow}_{\mathscr{C}'} \mathscr{P} \otimes \mathscr{Q}'$. The left hand side is a valid simulator because $\text{Out}(\text{Inst}_C(\mathscr{Q})) = \text{Out}(\text{Inst}_{C'}(\mathscr{Q}))$, and all of the honest parts of $\mathscr{P}$ are left untouched, since all of it is.

■

Theorem 4.13 (Horizontal Composition Theorem). For any protocols $\mathscr{P}, \mathscr{Q}$ with $\mathscr{P} \lhd \mathscr{Q}$ well defined and closed, and for any compatible corruption classes $\mathscr{C}, \mathscr{C}$, we have:

1. $\mathscr{Q} =_{\mathscr{C}} \mathscr{Q}' \implies \mathscr{P} \lhd \mathscr{Q} =_{\mathscr{C}'} \mathscr{P} \lhd \mathscr{Q}'$
2. $\mathscr{Q} \overset{\epsilon}{\approx}_{\mathscr{C}} \mathscr{Q}' \implies \mathscr{P} \lhd \mathscr{Q} \overset{\epsilon}{\approx}_{\mathscr{C}'} \mathscr{P} \lhd \mathscr{Q}'$

Furthermore, if $\mathscr{C}'$ is *strictly* compatible with $\mathscr{C}$, we have:

3. $\mathscr{Q} \overset{\epsilon}{\rightsquigarrow}_{\mathscr{C}} \mathscr{Q}' \implies \mathscr{P} \lhd \mathscr{Q} \overset{\epsilon}{\rightsquigarrow}_{\mathscr{C}'} \mathscr{P} \lhd \mathscr{Q}'$

Proof: As one might expect, Theorem 4.7(horizontal breakdown) will be critical to proving each of these statements.

One crude summary of the theorem, in the case that the protocols are closed, is that given compatible corruption models C, C', there's a system Stuff such that

$$\text{Inst}_{C'}(\mathscr{P} \lhd \mathscr{Q}) = \text{Stuff} \circ \text{Inst}_C(\mathscr{Q})$$

This summary suffices to prove a couple statements already.

1. By assumption, for any $C' \in \mathscr{C}'$, there exists a compatible $C \in \mathscr{C}$. In this case, we have:

$$\text{Inst}_{C'}(\mathscr{P} \lhd \mathscr{Q}) = \text{Sutff} \circ \text{Inst}_C(\mathscr{Q})$$

If we then apply $\mathscr{Q} =_{\mathscr{C}} \mathscr{Q}'$, we get:

$$\text{Stuff} \circ \text{Inst}_C(\mathscr{Q}) = \text{Stuff} \circ \text{Inst}_C(\mathscr{Q}')$$

and then, applying breakdown in reverse, we end up with $\text{Inst}_{C'}(\mathscr{P} \lhd \mathscr{Q}')$, completing our proof.

2. We apply the same reasoning, with the difference that:

$$\text{Stuff} \circ \text{Inst}_C(\mathscr{Q}) \overset{\epsilon}{\approx} \text{Stuff} \circ \text{Inst}_C(\mathscr{Q}')$$

rather than being strictly equal.

3. At this point our crude summary of the breakdown theorem is not sufficient anymore. We start with the same reasoning. For any $C' \in \mathscr{C}'$, there exists a *strictly* compatible $C \in \mathscr{C}$, and we have:

$$\text{Inst}_{C'}(\mathscr{P} \lhd \mathscr{Q}) = \text{Stuff} \circ \text{Inst}_C(\mathscr{Q})$$

then, we apply our assumption that $\mathscr{Q} \overset{\epsilon}{\rightsquigarrow}_{\mathscr{C}} \mathscr{Q}'$, giving us:

$$\text{Stuff} \circ \text{Inst}_C(\mathscr{Q}) \overset{\epsilon}{\approx} \text{Stuff} \circ (S \otimes 1(\ldots)) \circ \text{Inst}_C(\mathscr{Q})$$

Our strategy will be to rearrange the right hand side to get

$$(S' \otimes 1(\ldots)) \circ \text{Inst}_{C'}(\mathscr{P} \lhd \mathscr{Q}')$$

We start by unrolling Stuff, using strict compatability, to get:

$$1(O) \circ \begin{pmatrix} \circledast_{i \in [\mathscr{P}.n]} \text{Routed}(\text{Corrupt}'_{C'}(\mathscr{P}.P_i)) \\ * \\ \mathcal{R}_{\mathscr{P}} \\ \otimes \\ 1(\text{Leakage}, L_{\mathscr{Q}'}) \end{pmatrix} \circ \begin{pmatrix} \mathscr{P}.F \\ \otimes \\ 1(\text{Out}(\mathcal{R}_q)) \\ \otimes \\ 1(\mathscr{Q}'.\text{Leakage}) \\ \otimes \\ \bigotimes_{i \in [\mathscr{Q}'.n]} 1_i \end{pmatrix} \circ \begin{pmatrix} S \\ \otimes \\ 1(O_{\bar{S}}) \end{pmatrix} \circ \text{Inst}_C(\mathscr{Q}')$$

with $O_{\bar{S}} := \text{Out}(\text{Inst}_C(\mathscr{Q}'))/\text{Out}(S)$, and with each $1_i := 1(\text{Out}(\text{Inst}_C(\mathscr{Q}'.P_i)))$. we can apply interchange a few times to get:

$$1(O) \circ \begin{pmatrix} \left(\circledast_{C'_i \neq \text{H}} \begin{pmatrix} \text{Routed}(\text{Corrupt}'_{C'}(\mathscr{P}.P_i)) \\ \otimes \\ 1(L_i) \\ \otimes \\ 1(\text{Leakage}) \end{pmatrix} \right) \circ \begin{pmatrix} S \\ \otimes \\ 1(O_S) \end{pmatrix} \circ \begin{pmatrix} \circledast_{C_i \neq \text{H}} \text{Routed}(\text{Corrupt}_C(\mathscr{Q}'.P_i)) \\ \otimes \\ 1(\text{Out}(\mathscr{P}.F), \text{Out}(\mathscr{Q}.F)) \end{pmatrix} \\ * \\ \circledast_{C'_i = \text{H}} \text{Routed}(\text{Corrupt}_{C'}((\mathscr{P} \lhd \mathscr{Q}').P_i)) \\ * \\ \mathcal{R}_{\mathscr{P}} \circ \mathcal{R}_{\mathscr{Q}'} \end{pmatrix} \circ \begin{pmatrix} \mathscr{P}.F \\ \otimes \\ \mathscr{Q}'.F \end{pmatrix}$$

with $O_S := O_{\bar{S}} \cup \text{Out}(\mathscr{P}.F)$ and L_i as per the horizontal breakdown theorem. The only functions that S masks are the leakage, the malicious corruption functions, and the logs from semi-honest corruption. Semi-honest corruption does not use any outputs of S, instead relying on the $\mathscr{Q}'.P_i$, accessible via $1(O_S)$. In the case of malicious corruption, since $\text{Corrupt}'_{C'}(\mathscr{P}.P_i)$ omits the Call_{F_i} functions, the system also has no dependencies on the output of S. Since none of these corrupted players depend on S, we can slide it forward, using interchange, to get:

$$1(O) \circ \begin{pmatrix} \begin{pmatrix} S \\ \otimes \\ 1(\ldots) \end{pmatrix} \circ \left(\circledast_{C'_i \neq \text{H}} \begin{pmatrix} \text{Routed}(\text{Corrupt}'_{C'}(\mathscr{P}.P_i)) \\ \otimes \\ 1(L_i) \\ \otimes \\ 1(\text{Leakage}) \end{pmatrix} \right) \circ \begin{pmatrix} \circledast_{C_i \neq \text{H}} \text{Routed}(\text{Corrupt}_C(\mathscr{Q}'.P_i)) \\ \otimes \\ 1(\text{Out}(\mathscr{P}.F), \text{Out}(\mathscr{Q}.F)) \end{pmatrix} \\ * \\ \circledast_{C'_i = \text{H}} \text{Routed}(\text{Corrupt}_{C'}((\mathscr{P} \lhd \mathscr{Q}').P_i)) \\ * \\ \mathcal{R}_{\mathscr{P}} \circ \mathcal{R}_{\mathscr{Q}'} \end{pmatrix} \circ \begin{pmatrix} \mathscr{P}.F \\ \otimes \\ \mathscr{Q}'.F \end{pmatrix}$$

which becomes:

$$\begin{pmatrix} S \\ \otimes \\ 1(\mathrm{Out}(\mathrm{Inst}_{C'}(\mathscr{P} \lhd \mathscr{Q}'))/\mathrm{Out}(S)) \end{pmatrix} \circ \mathrm{Inst}_{C'}(\mathscr{P} \lhd \mathscr{Q}')$$

From this chain of equalities we conclude that $\mathscr{P} \lhd \mathscr{Q}' \overset{\epsilon}{\rightsquigarrow} \mathscr{P} \lhd \mathscr{Q}'$

■

4.3 Global Functionalities

Definition 4.18 (Relatively Closed Protocols). A protocol $\mathscr{P}$ is *closed relative to* a game G if:

- $\mathrm{In}(\mathscr{P}) = \emptyset$
- $\mathrm{IdealIn}(\mathscr{P}) \subseteq \mathrm{Out}(G)$

□

Definition 4.19 (Relative Instantiation). Given a closed protocol $\mathscr{P}$ relative to G, we can define, for any corruption model C, the relative instantiation:

$$\mathrm{Inst}^G_C(\mathscr{P}) := \begin{pmatrix} \mathrm{Inst}_C(\mathscr{P}) \\ \otimes \\ 1(\mathrm{Out}(G)) \end{pmatrix} \circ G$$

We can also extend this to the case of simulated instantiation, defining, for any simulator S:

$$\mathrm{SimInst}^G_{S,C}(\mathscr{P}) := \begin{pmatrix} \mathrm{SimInst}_{S,C}(\mathscr{P}) \\ \otimes \\ 1(\mathrm{Out}(G)) \end{pmatrix} \circ G$$

□

Definition 4.20 (Relative Notions of Equality). Given closed protocols $\mathscr{P}, \mathscr{Q}$ relative to G, with the same shape, and a corruption class $\mathscr{C}$ for these protocols, we define:

- $\mathscr{P} =^G_{\mathscr{C}} \mathscr{Q} \iff \forall C \in \mathscr{C}.\ \mathrm{Inst}^G_C(\mathscr{P}) = \mathrm{Inst}^G_C(\mathscr{Q})$
- $\mathscr{P} \overset{\epsilon}{\approx}{}^G_{\mathscr{C}} \mathscr{Q} \iff \forall C \in \mathscr{C}.\ \mathrm{Inst}^G_C(\mathscr{P}) \overset{\epsilon}{\approx} \mathrm{Inst}^G_C(\mathscr{Q})$
- $\mathscr{P} \overset{\epsilon}{\rightsquigarrow}{}^G_{\mathscr{C}} \mathscr{Q} \iff \forall C \in \mathscr{C}.\ \exists S.\ \mathrm{Inst}^G_C(\mathscr{P}) \overset{\epsilon}{\approx} \mathrm{SimInst}^G_{S,C}(\mathscr{Q})$

□

Theorem 4.14 (Relative Equality Hierarchy). For any corruption class $\mathscr{C}$ and game G, we have:

1. $\mathscr{P} =^G_{\mathscr{C}} \mathscr{Q} \implies \mathscr{P} \overset{0}{\approx}{}^G_{\mathscr{C}} \mathscr{Q}$.
2. $\mathscr{P} \overset{\epsilon}{\approx}{}^G_{\mathscr{C}} \mathscr{Q} \implies \mathscr{P} \overset{\epsilon}{\rightsquigarrow}{}^G_{\mathscr{C}} \mathscr{Q}$.

Proof:

1. This follows from the fact that $A = B \implies A \overset{0}{\approx} B$ for any systems A, B.

2. In the proof of Theorem 4.8, we used the existence of a simulator S such that $\text{SimInst}_{S,C}(\mathscr{P}) = \text{Inst}_C(\mathscr{P})$. This simulator will also satisfy $\text{SimInst}^G_{S,C}(\mathscr{P}) = \text{Inst}^G_C(\mathscr{P})$, and can thus be used directly to prove this relation.

■

Theorem 4.15 (Transitivity of Relative Equality). For any protocols $\mathscr{L}, \mathscr{P}, \mathscr{Q}$ closed relative to a game G, and for any corruption class, we have:

1. $\mathscr{L} =^G_{\mathscr{C}} \mathscr{P}, \mathscr{P} =^G_{\mathscr{C}} \mathscr{Q} \implies \mathscr{L} =^G_{\mathscr{C}} \mathscr{Q}$,
2. $\mathscr{L} \overset{\epsilon_1}{\approx}{}^G_{\mathscr{C}} \mathscr{P}, \mathscr{P} \overset{\epsilon_2}{\approx}{}^G_{\mathscr{C}} \mathscr{Q} \implies \mathscr{L} \overset{\epsilon_1+\epsilon_2}{\approx}{}^G_{\mathscr{C}} \mathscr{Q}$,
3. $\mathscr{L} \overset{\epsilon_1}{\rightsquigarrow}{}^G_{\mathscr{C}} \mathscr{P}, \mathscr{P} \overset{\epsilon_2}{\rightsquigarrow}{}^G_{\mathscr{C}} \mathscr{Q} \implies \mathscr{L} \overset{\epsilon_1+\epsilon_2}{\rightsquigarrow}{}^G_{\mathscr{C}} \mathscr{Q}$.

Proof: Once again, the first two parts follow directly from Lemma 3.5, by considering the systems $\text{Inst}^G_C(\mathscr{L})$, $\text{Inst}^G_C(\mathscr{P})$, $\text{Inst}^G_C(\mathscr{Q})$ for any $C \in \mathscr{C}$.

For part 3, given any $C \in \mathscr{C}$, there exists S_1, S_2 such that:

- $\text{Inst}^G_C(\mathscr{L}) \overset{\epsilon_1}{\approx} \text{SimInst}^G_{S_1,C}(\mathscr{P})$,
- $\text{Inst}^G_C(\mathscr{P}) \overset{\epsilon_2}{\approx} \text{SimInst}^G_{S_2,C}(\mathscr{Q})$.

Next, observe that for any protocol $\mathscr{P}$, we can write:

$$\text{SimInst}^G_C = \begin{pmatrix} S \\ \otimes \\ 1(O) \end{pmatrix} \circ \text{Inst}^G_C(\mathscr{P})$$

where $O = \text{Out}(\text{Inst}_C(\mathscr{P}))/\text{Out}(S) \cup \text{Out}(G)$.

We then apply transitivity for systems and interchange get:

$$\text{Inst}^G_C(\mathscr{L}) \overset{\epsilon_1+\epsilon_2}{\approx} \begin{pmatrix} S_1 \circ S_2 \\ \otimes \\ 1(O) \end{pmatrix} \circ \text{Inst}^G_C(\mathscr{Q})$$

And the left side is simply $\text{SimInst}^G_{(S_1 \circ S_2),C}(\mathscr{Q})$, concluding our proof.

■

Theorem 4.16 (Global Malicious Completeness). Let $\mathscr{P}$ and $\mathscr{Q}$ closed protocols relative to G with the same shape. Given any class of corruptions $\mathscr{C}$, let $\mathscr{C}'$ be a related class, containing models in $\mathscr{C}$ with some malicious corruptions replaced with semi-honest corruptions. We then have:

1. $\mathscr{P} =^G_C \mathscr{Q} \implies \mathscr{P} =^G_{C'} \mathscr{Q}$,
2. $\mathscr{P} \overset{\epsilon}{\approx}{}^G_C \mathscr{Q} \implies \mathscr{P} \overset{\epsilon}{\approx}{}^G_{C'} \mathscr{Q}$,

Furthermore, if for any $C \in \mathscr{C}$ and its related model $C' \in \mathscr{C}$, there exists a simulator S_{M} such that $\text{Inst}^G_C(\mathscr{Q}) = \text{SimInst}^G_{S_{\text{M}},C'}(\mathscr{Q})$, then it additionally holds that:

3. $\mathscr{P} \overset{\epsilon}{\rightsquigarrow}{}^G_{\mathscr{C}} \mathscr{Q} \implies \mathscr{P} \overset{\epsilon}{\rightsquigarrow}{}^G_{\mathscr{C}'} \mathscr{Q}$

Proof: We proceed similarly to Theorem 4.10 (malicious completeness). In that theorem, the key observation was that for any $C' \in \mathscr{C}'$ and the related $C \in \mathscr{C}$, it holds that:

$$\text{Inst}_{C'}(\mathscr{P}) = \text{SimInst}_{S_{\text{SH}},C}(\mathscr{P})$$

(this observation also doesn't depend on $\mathscr{P}$ being fully closed, allowing us to use it here).

Now, this clearly implies that:

$$\text{Inst}^G_{C'}(\mathscr{P}) = \text{SimInst}^G_{S_{\text{SH}},C}(\mathscr{P})$$

And then, using our observation from Theorem 4.15, we can write this as:

$$\text{Inst}^G_{C'}(\mathscr{P}) = \begin{pmatrix} S_{\text{SH}} \\ \otimes \\ 1(O) \end{pmatrix} \circ \text{Inst}^G_C(\mathscr{P})$$

where $O = \text{Out}(\text{Inst}_C(\mathscr{P}))/\text{Out}(S) \cup \text{Out}(G)$.

This immediately implies parts 1 and 2.

For part 3, apply the assumption in the implication to get:

$$\begin{pmatrix} S_{\text{SH}} \\ \otimes \\ 1(O) \end{pmatrix} \circ \begin{pmatrix} S \\ \otimes \\ 1(O) \end{pmatrix} \circ \text{Inst}^G_C(\mathscr{Q})$$

Then apply the assumption about being able to simulate malicious corruption to get:

$$\begin{pmatrix} S_{\text{SH}} \\ \otimes \\ 1(O) \end{pmatrix} \circ \begin{pmatrix} S \\ \otimes \\ 1(O) \end{pmatrix} \circ \begin{pmatrix} S_{\text{M}} \\ \otimes \\ 1(O) \end{pmatrix} \circ \text{Inst}^G_{C'}(\mathscr{Q})$$

which can then be rearranged with interchange to get:

$$\begin{pmatrix} S_{\text{SH}} \circ S \circ S_{\text{M}} \\ \otimes \\ 1(O) \end{pmatrix} \circ \text{Inst}^G_{C'}(\mathscr{Q})$$

And then if we apply the same observation about SimInst^G, we realize that this is:

$$\text{SimInst}^G_{(S_{\text{SH}} \circ S \circ S_{\text{M}}),C'}(\mathscr{Q})$$

concluding our proof.

■

Theorem 4.17 (Global Vertical Composition Theorem). For any protocol $\mathscr{P}$ and game F, such that $\mathscr{P} \circ F$ is well defined and closed relative to G, and for any corruption class $\mathscr{C}$, we have:

1. $F = F' \implies \mathscr{P} \circ F =^G_{\mathscr{C}} \mathscr{P} \circ F'$
2. $F \overset{\epsilon}{\approx} F' \implies \mathscr{P} \circ F \overset{\epsilon}{\approx}{}^G_{\mathscr{C}} \mathscr{P} \circ F'$

Proof: The proof of Theorem 4.11 will be the basis for what we do here. Using it, we can write:

$$\text{Inst}^G_C(\mathscr{P} \circ F) = \begin{pmatrix} A \circ F \\ \otimes \\ 1(\text{Out}(G)) \end{pmatrix} \circ G$$

for some system A. At this point, the theorem immediately holds, since $\circ$ and $\otimes$ (for systems) respect both $=$ and $\approx$.

■

Theorem 4.18 (Global Concurrent Composition Theorem). Let $\mathscr{P}, \mathscr{Q}$ be closed protocols relative to G, with $\mathscr{P} \otimes \mathscr{Q}$ well defined. For any compatible corruption classes $\mathscr{C}, \mathscr{C}'$ it holds that:

1. $\mathscr{Q} =^G_{\mathscr{C}} \mathscr{Q}' \implies \mathscr{P} \otimes \mathscr{Q} =^G_{\mathscr{C}'} \mathscr{P} \otimes \mathscr{Q}'$
2. $\mathscr{Q} \overset{\epsilon}{\approx}{}^G_{\mathscr{C}} \mathscr{Q}' \implies \mathscr{P} \otimes \mathscr{Q} \overset{\epsilon}{\approx}{}^G_{\mathscr{C}'} \mathscr{P} \otimes \mathscr{Q}'$
3. $\mathscr{Q} \overset{\epsilon}{\rightsquigarrow}{}^G_{\mathscr{C}} \mathscr{Q}' \implies \mathscr{P} \otimes \mathscr{Q} \overset{\epsilon}{\rightsquigarrow}{}^G_{\mathscr{C}'} \mathscr{P} \otimes \mathscr{Q}'$

Proof: We start by using Theorem 4.6, giving us:

$$\text{Inst}^G_{C'}(\mathscr{P} \otimes \mathscr{Q}) = \begin{pmatrix} \text{Inst}_{C'}(\mathscr{P}) \\ \otimes \\ \text{Inst}_C(\mathscr{Q}) \\ \otimes \\ 1(\text{Out}(G)) \end{pmatrix} \circ G = \begin{pmatrix} \text{Inst}_{C'}(\mathscr{P}) \\ \otimes \\ 1(\text{Out}(\text{Inst}_C(\mathscr{Q}))) \\ \otimes \\ 1(\text{Out}(G)) \end{pmatrix} \circ \text{Inst}^G_C(\mathscr{Q})$$

We can then immediately derive parts 1 and 2.

For part 3, we apply the hypothesis to the last part of the above relation, to get:

$$\text{Inst}^G_{C'} \overset{\epsilon}{\approx} \begin{pmatrix} \text{Inst}_{C'}(\mathscr{P}) \\ \otimes \\ 1(\text{Out}(\text{Inst}_C(\mathscr{Q}))) \\ \otimes \\ 1(\text{Out}(G)) \end{pmatrix} \circ \text{SimInst}^G_{S,C}(\mathscr{Q})$$

Then, we unroll $\text{SimInst}^G_{S,C}(\mathscr{Q})$, to get:

$$\begin{pmatrix} \text{Inst}_{C'}(\mathscr{P}) \\ \otimes \\ 1(\text{Out}(\text{Inst}_C(\mathscr{Q}))) \\ \otimes \\ 1(\text{Out}(G)) \end{pmatrix} \circ \begin{pmatrix} \begin{pmatrix} S \\ \otimes \\ 1(\dots) \end{pmatrix} \circ \text{Inst}_C(\mathscr{Q}) \\ \otimes \\ 1(\text{Out}(G)) \end{pmatrix} \circ G$$

Then, we apply interchange to get:

$$\begin{pmatrix} \begin{pmatrix} 1(\dots) \\ \otimes \\ S \\ \otimes \\ 1(\dots) \end{pmatrix} \circ \begin{pmatrix} \text{Inst}_{C'}(\mathscr{P}) \\ \otimes \\ \text{Inst}_C(\mathscr{Q}) \end{pmatrix} \\ \otimes \\ 1(\text{Out}(G)) \end{pmatrix} \circ G$$

But this is just $\text{SimInst}^G_{S',C'}(\mathscr{P} \otimes \mathscr{Q})$, for some simulator S', applying concurrent breakdown in reverse.

■

Theorem 4.19 (Global Horizontal Composition Theorem). For any protocols $\mathscr{P}, \mathscr{Q}$ closed relative to G, with $\mathscr{P} \lhd \mathscr{Q}$ well defined, and for any compatible corruption classes $\mathscr{C}, \mathscr{C}$, we have:

1. $\mathscr{Q} =^G_{\mathscr{C}} \mathscr{Q}' \implies \mathscr{P} \lhd \mathscr{Q} =^G_{\mathscr{C}'} \mathscr{P} \lhd \mathscr{Q}'$
2. $\mathscr{Q} \overset{\epsilon}{\approx}{}^G_{\mathscr{C}} \mathscr{Q}' \implies \mathscr{P} \lhd \mathscr{Q} \overset{\epsilon}{\approx}{}^G_{\mathscr{C}'} \mathscr{P} \lhd \mathscr{Q}'$

Furthermore, if $\mathscr{C}'$ is *strictly* compatible with $\mathscr{C}$, we have:

3. $\mathscr{Q} \overset{\epsilon}{\leadsto}{}^G_{\mathscr{C}} \mathscr{Q}' \implies \mathscr{P} \lhd \mathscr{Q} \overset{\epsilon}{\leadsto}{}^G_{\mathscr{C}'} \mathscr{P} \lhd \mathscr{Q}'$

Proof: This proof is similar to that of Theorem 4.13. By compatability, for any $C' \in \mathscr{C}'$, we have a compatible $C \in \mathscr{C}$.

A crude summary of the horizontal breakdown theorem is that:

$$\text{Inst}_{C'}(\mathscr{P} \lhd \mathscr{Q}) = \text{Stuff} \circ \begin{pmatrix} \text{Inst}_C(\mathscr{Q}) \\ \otimes \\ 1(\text{In}(\mathscr{P}.F)) \end{pmatrix}$$

Using the fact that being closed relative to G means $\text{In}(\mathscr{P}.F) \subseteq \text{Out}(G)$, we get:

$$\text{Inst}^G_{C'}(\mathscr{P} \lhd \mathscr{Q}) = \begin{pmatrix} \text{Stuff} \\ \otimes \\ 1(\text{Out}(G)) \end{pmatrix} \circ \text{Inst}^G_C(\mathscr{Q})$$

Part 1 and 2 both follow immediately from this decomposition.

For part 3, we dig a bit deeper into the proof of Theorem 4.13. In that proof, it was actually shown that:

$$\text{Stuff} \circ \text{SimInst}_{S,C}(\mathscr{Q}') = \text{SimInst}_{S',C'}(\mathscr{P} \lhd \mathscr{Q}')$$

for some appropriate simulator S'.

We can start to apply this, first by using our hypothesis:

$$\text{Inst}^G_{C'}(\mathscr{P} \lhd \mathscr{Q}) = \begin{pmatrix} \text{Stuff} \\ \otimes \\ 1(\text{Out}(G)) \end{pmatrix} \circ \text{Inst}^G_C(\mathscr{Q}) \overset{\epsilon}{\approx} \begin{pmatrix} \text{Stuff} \\ \otimes \\ 1(\text{Out}(G)) \end{pmatrix} \circ \text{SimInst}^G_C(\mathscr{Q}')$$

Next, we unroll the right side, to get:

$$\begin{pmatrix} \text{Stuff} \\ \otimes \\ 1(\text{Out}(G)) \end{pmatrix} \circ \begin{pmatrix} \text{SimInst}_{S,C}(\mathscr{Q}') \\ \otimes \\ 1(\text{Out}(G)) \end{pmatrix} \circ G$$

Then, apply interchange, to get:

$$\begin{pmatrix} \text{Stuff} \circ \text{SimInst}_{S,C}(\mathscr{Q}') \\ \otimes \\ 1(\text{Out}(G)) \end{pmatrix} \circ G$$

And finally, apply the fact we dug up above, to get:

$$\begin{pmatrix} \text{SimInst}_{S',C'}(\mathscr{P} \lhd \mathscr{Q}) \\ \otimes \\ 1(\text{Out}(G)) \end{pmatrix} \circ G$$

which is none other than $\text{SimInst}^G_{S',C'}(\mathscr{P} \lhd \mathscr{Q})$.

■

4.4 Hopping Ideal Functionalities

Lemma 4.20 (Deidealization Lemma). Given a closed protocol $\mathscr{P}$ with an ideal functionality $F \otimes G$, there exists protocols $\mathscr{P}'$ and $\mathscr{G}$ such that:

$$\mathscr{P} \equiv \mathscr{P}' \lhd \mathscr{G}$$

and $\mathscr{P}'$ has ideal functionality F.

Proof: The players of $\mathscr{P}'$ are those of $\mathscr{P}$, except that each P_i's call to a function $g \in \text{Out}(G)$ is replaced with a renamed function g_i. $\mathscr{G}$ will have one player for each player in $\mathscr{P}'$. Each player $\mathscr{G}.P_i$ exports a function g_i for each input g_i of $\mathscr{P}'.P_i$, which immediately calls $g \in \text{Out}(G)$, and returns the result. The leakage of $\mathscr{G}$ will simply be $\mathscr{P}.\text{Leakage} \cap \text{Out}(G)$. From this definition, it's clear that $\mathscr{P}$ is literally equal to $\mathscr{P}' \lhd \mathscr{G}$, as when the players in the latter are formed, the calls to the intermediate g_i disappear, with each player calling $g \in \text{Out}(G)$ directly

■

Lemma 4.21 (Embedding Lemma). Given a protocol $\mathscr{P}$ closed relative to a game G, there exists a protocol $\text{Embed}_G(\mathscr{P})$ such that for any corruption model C, we have:

$$\text{Inst}_C^G(\mathscr{P}) = \text{Inst}_C(\text{Embed}_G(\mathscr{P}))$$

Proof: This one is quite simple. $\text{Embed}_G(\mathscr{P})$ has the same players as $\mathscr{P}$, with the ideal functionality becoming:

$$\begin{pmatrix} \mathscr{P}.F \\ \otimes \\ 1(\text{Out}(G)) \end{pmatrix} \circ G$$

and the leakage being $\mathscr{P}.\text{Leakage} \cup \text{Out}(G)$. The two instantiations will then clearly be equal under any corruption model.

■

4.5 Some Syntactical Conventions

5 Examples

5.1 Constructing Private Channels

In this section, we consider the problem of constructing a *private* channel from a *public* channel. A public channel leaks all messages sent over it to an adversary, whereas a private channel leaks a minimal amount of information: in our case, essentially just the length of messages sent over the channel.

We'll be constructing a two-party private channel from a public channel using an encryption scheme, and will also show that this construction is secure, even if one of the two parties using the channel is corrupted.

Let's start with the ideal functionality representing a public channel, as Game 5.1.

A few clarifications on the notation in this game:

- For $i \in \{1, 2\}$, we let $\bar{\imath}$ denote either 2 or 1, respectively.
- There are two versions of Send_i and Recv_i, for $i \in \{1, 2\}$.
- The pop function on queues is asynchronous, meaning that we wait until the queue is not empty to remove the oldest element from it.
- The queues are public in an *immutable* fashion: they can be read but not modified outside the package.

$F[\text{PubChan}]$

view $m_{1\to 2}, m_{2\to 1} \leftarrow \text{FifoQueue.new}()$

$\underline{\text{Send}_{i\to\bar{\imath}}(m)}$:	$\underline{\text{Recv}_{i\to\bar{\imath}}()}$:
$m_{i\to\bar{\imath}}.\text{push}(m)$	**return await** $m_{i\to\bar{\imath}}.\text{pop}()$

Game 5.1: Public Channel Functionality

The idea behind this functionality is that each party can send messages to, or receive messages from the other party. However, at any point, the currently stored messages are readable by the adversary. Note that this assignment of which functions are usable by which entities is not defined by the functionality *itself*, but rather merely suggested by its syntax, and enforced only by how protocols will eventually use the functionality.

Next, we look at a functionality for *private* channels, captured by Game 5.2.

The crucial difference is the nature of the leakage. Now, rather than being able to see the current state of either message queue, including the messages themselves, now the adversary can only see a historical log for each queue, describing only the *length* of the messages inserted into the queue. The reason for having a historical log, rather than just a snapshot of the lengths of the current messages, is to make the simulator's job easier in the eventual proof of security. For technical reasons, it's simpler to allow the log to be mutated, so that the simulator can "remember" which parts of the log they've already seen, by popping elements from the queue.

Now, we need to define the protocols. One protocol will use the private channel

$F[\text{PrivChan}]$

$\textbf{view } m_{1\to 2}, m_{2\to 1} \leftarrow \text{FifoQueue.new}()$
$\textbf{pub } l_{1\to 2}, l_{2\to 1} \leftarrow \text{FifoQueue.new}()$

$\underline{\text{Send}_{i\to \bar{i}}(m)\text{:}}$
$\quad m_{i\to \bar{i}}.\text{push}(m)$
$\quad l_{i\to \bar{i}}.\text{push}((\texttt{push}, |m|))$

$\underline{\text{Recv}_{i\to \bar{i}}()\text{:}}$
$\quad m \leftarrow \textbf{await } m_{i\to \bar{i}}.\text{pop}()$
$\quad l_{i\to \bar{i}}.\text{push}(\texttt{pop})$
$\quad \textbf{return } m$

Game 5.2: Private Channel Functionality

to send messages, and the other will try and implement the same behavior, but using only the public channel, aided by an encryption scheme.

Let's start with the simpler private channel protocol, which we'll call $\mathscr{Q}$, and defined via Protocol 5.3

$\mathscr{Q}$ is characterized by:

- Leakage := $\{l_{1\to 2}, l_{2\to 1}\}$,
- F := PrivChan,
- And two players defined via the following system (for $i \in \{1, 2\}$):

P_i

$\underline{\text{Send}_i(m)\text{:}}$
$\quad \text{Send}_{i\to \bar{i}}(m)$

$\underline{\text{Recv}_i()\text{:}}$
$\quad \textbf{return await } \text{Recv}_{\bar{i}\to i}()$

Protocol 5.3: Private Channel Protocol

This protocol basically just provides each player access with their corresponding functions in the functionality, and leaks the parts of the functionality that the adversary should have access to, as expected.

Next, we need to define a protocol providing an encrypted channel. We'll call this one $\mathscr{P}$. The basic idea is that $\mathscr{P}$ will encrypt messages before sending them over the public channel. We'll be using public-key encryption, as defined in todo. For the sake of simplicity, we'll be relying on an additional functionality, Keys, which will be used to setup each party's key pair, and allow each party to agree on the other's public key.

This functionality is defined in Game 5.4. The basic idea is that a key pair is generated for each party, and that party can see their secret key, along with the public key for the other party. Furthermore, we allow the adversary to see both public keys.

$$
\begin{array}{|l|}
\hline
\textsf{Keys} \\
\\
(\mathrm{sk}_1, \mathrm{pk}_1) \xleftarrow{\$} \mathrm{Gen}() \\
(\mathrm{sk}_2, \mathrm{pk}_2) \xleftarrow{\$} \mathrm{Gen}() \\
\\
\underline{\mathrm{Keys}_i():} \\
\quad \textbf{return } (\mathrm{sk}_i, \mathrm{pk}_{\bar{i}}) \\
\\
\underline{\mathrm{PKs}():} \\
\quad \textbf{return } (\mathrm{pk}_1, \mathrm{pk}_2) \\
\hline
\end{array}
$$

Game 5.4: Keys Functionality

With this in hand, we can define $\mathscr{P}$ itself, in Protocol 5.5.

$\mathscr{P}$ is characterized by:

- Leakage := $\{m_{1\to 2}, m_{2\to 1}, \mathrm{PKs}\}$,
- F := Keys $\otimes$ PrivChan,
- and two players defined via the following system (for $i \in \{1, 2\}$):

$$
\begin{array}{|ll|}
\hline
P_i & \\
\\
(\mathrm{sk}_i, \mathrm{pk}_{\bar{i}}) \leftarrow \mathrm{Keys}_i() & \\
\\
\underline{\mathrm{Send}_i(m):} & \underline{\mathrm{Recv}_i():} \\
\quad \mathrm{Send}_{i\to\bar{i}}(\mathrm{Enc}(\mathrm{pk}_{\bar{i}}, m)) & \quad c \leftarrow \textbf{await } \mathrm{Recv}_{\bar{i}\to i}() \\
 & \quad \textbf{return } \mathrm{Dec}(\mathrm{sk}_i, c) \\
\hline
\end{array}
$$

Protocol 5.5: Encrypted Channel Protocol

Each player will encrypt their message for the other player before sending it, and then decrypt it using their secret key after receiving it.

At this point we can state and prove the crux of this example:

Claim 5.1. Let $\mathscr{C}$ be the class of corruptions where up to 1 of 2 parties is either maliciously corrupt or semi-honestly corrupt. Then we have:

$$\mathscr{P} \overset{2\text{-IND}}{\rightsquigarrow}_{\mathscr{C}} \mathscr{Q}$$

Proof: We consider the cases where all the parties are honest and some of the parties are corrupted separately. Furthermore, we only need to consider malicious corruption, since the parties in $\mathscr{Q}$ just directly call functions from the ideal functionality, and so we can simulate malicious corruption from semi-honest corruption, and can thus apply part 3 of Theorem 4.10.

Honest Case: Let H be a corruption model where both parties are honest. We prove that $\mathscr{P} \overset{2\cdot\text{IND}}{\rightsquigarrow}_{\{\text{H}\}} \mathscr{Q}$.

The high level idea is that since ciphertexts should be indistinguishable from random encryptions, the information in the log we get as a simulator for $\mathscr{Q}$ is enough to fake all the ciphertexts the environment expects to see in $\mathscr{P}$.

We start by unrolling $\text{Inst}_\text{H}(\mathscr{P})$, obtaining:

$$
\text{Inst}_\text{H}(\mathscr{P}) = \boxed{\begin{array}{ll}
\Gamma^0 & \\
& \\
\textbf{view } c_{1\to 2}, c_{2\to 1} \leftarrow \text{FifoQueue.new}() & \\
(\text{sk}_i, \text{pk}_{\bar{\imath}}) \leftarrow \text{Keys}_i() & \\
& \\
\underline{\text{PKs}():} & \\
\quad \textbf{return } (\text{pk}_1, \text{pk}_2) & \\
\underline{\text{Send}_i(m):} & \underline{\text{Recv}_i():} \\
\quad c \leftarrow \text{Enc}(\text{pk}_{\bar{\imath}}, m) & \quad c \leftarrow \textbf{await } c_{\bar{\imath}\to i}.\text{pop}() \\
\quad c_{i\to\bar{\imath}}.\text{push}(c) & \quad \textbf{return } \text{Dec}(\text{sk}_i, c)
\end{array}} \circ \text{Keys}
$$

Note that we can ignore all parts of the instantiation related to channels, including the router, because the parties don't use any channels. We also took the liberty of renaming $m_{i\to\bar{\imath}}$ to $c_{i\to\bar{\imath}}$, to emphasize the fact that these queues contain ciphertexts, instead of messages.

Next, we pull a bit of a trick. It turns out that since both parties are honest, we don't need to actually decrypt the ciphertext. Instead, one party can simply send the plaintext via a separate channel to tthe other. Applying this gives us:

$$\Gamma^0 \circ \mathrm{Keys} = \boxed{\begin{array}{ll} \Gamma^1 & \\ & \\ \textbf{view } c_{1\to 2}, c_{2\to 1} \leftarrow \mathrm{FifoQueue.new}() & \\ \textbf{view } m_{1\to 2}, m_{2\to 1} \leftarrow \mathrm{FifoQueue.new}() & \\ (\bullet, \mathrm{pk}_{\bar{i}}) \leftarrow \mathrm{Keys}_i() & \\ & \\ \underline{\mathrm{PKs}():} & \\ \quad \textbf{return } (\mathrm{pk}_1, \mathrm{pk}_2) & \\ \underline{\mathrm{Send}_i(m):} & \underline{\mathrm{Recv}_i():} \\ \quad c \leftarrow \mathrm{Enc}(\mathrm{pk}_{\bar{i}}, m) & \quad c \leftarrow \textbf{await } \mathrm{c}_{\bar{i}\to i}.\mathrm{pop}() \\ \quad c_{i\to\bar{i}}.\mathrm{push}(c) & \quad m \leftarrow \textbf{await } \mathrm{m}_{\bar{i}\to i}.\mathrm{pop}() \\ \quad m_{i\to\bar{i}}.\mathrm{push}(m) & \quad \textbf{return } m \end{array}} \circ \mathrm{Keys}$$

This is equal because of the correctness property for encryption, which guarantees that $m = \mathrm{Dec}(\mathrm{Enc}(\mathrm{pk}, m))$. Furthermore, the timing properties are the same, since the size of both the $c_{i\to\bar{i}}$ and $m_{i\to\bar{i}}$ queues are always the same.

At this point, we can offload the decryption to the IND game, giving us:

$$\Gamma^1 \circ \mathrm{Keys} = \boxed{\begin{array}{ll} \Gamma^2 & \\ & \\ \textbf{view } c_{1\to 2}, c_{2\to 1} \leftarrow \mathrm{FifoQueue.new}() & \\ \textbf{view } m_{1\to 2}, m_{2\to 1} \leftarrow \mathrm{FifoQueue.new}() & \\ & \\ \underline{\mathrm{PKs}():} & \\ \quad \textbf{return } (\textbf{super}.\mathrm{pk}_1, \textbf{super}.\mathrm{pk}_2) & \\ \underline{\mathrm{Send}_i(m):} & \underline{\mathrm{Recv}_i():} \\ \quad c \leftarrow \mathrm{Challenge}_{\bar{i}}(m) & \quad c \leftarrow \textbf{await } \mathrm{c}_{\bar{i}\to i}.\mathrm{pop}() \\ \quad c_{i\to\bar{i}}.\mathrm{push}(c) & \quad m \leftarrow \textbf{await } \mathrm{m}_{\bar{i}\to i}.\mathrm{pop}() \\ \quad m_{i\to\bar{i}}.\mathrm{push}(m) & \quad \textbf{return } m \end{array}} \circ \begin{pmatrix} \mathrm{IND}_0 \\ \otimes \\ \mathrm{IND}_0 \end{pmatrix}$$

We use two instances of IND, and we disambiguate the functions in each instance by attaching 1 or 2 to each function.

Next, we can hop to IND_1, since:

$$\Gamma^2 \circ \begin{pmatrix} \mathrm{IND}_0 \\ \otimes \\ \mathrm{IND}_0 \end{pmatrix} \overset{\epsilon}{\approx} \Gamma^2 \circ \begin{pmatrix} \mathrm{IND}_1 \\ \otimes \\ \mathrm{IND}_1 \end{pmatrix}$$

with $\epsilon = 2 \cdot \text{IND}$.

If we unroll this last game, we get:

$$\Gamma^1 \circ \begin{pmatrix} \text{IND}_1 \\ \otimes \\ \text{IND}_1 \end{pmatrix} = \boxed{\begin{array}{ll} \Gamma^3 & \\ & \\ \textbf{view } c_{1\to 2}, c_{2\to 1} \leftarrow \text{FifoQueue.new}() & \\ \textbf{view } m_{1\to 2}, m_{2\to 1} \leftarrow \text{FifoQueue.new}() & \\ (\text{sk}_i, \text{pk}_i) \xleftarrow{\$} \text{Gen}() & \\ & \\ \underline{\text{PKs}():} & \\ \quad \textbf{return } (\text{pk}_1, \text{pk}_2) & \\ \underline{\text{Send}_i(m):} & \underline{\text{Recv}_i():} \\ \quad r \xleftarrow{\$} \mathbf{M}(|m|) & \quad c \leftarrow \textbf{await } \text{c}_{\bar{\imath}\to i}.\text{pop}() \\ \quad c_{i\to\bar{\imath}}.\text{push}(\text{Enc}(\text{pk}_{\bar{\imath}}, r)) & \quad m \leftarrow \textbf{await } \text{m}_{\bar{\imath}\to i}.\text{pop}() \\ \quad m_{i\to\bar{\imath}}.\text{push}(m) & \quad \textbf{return } m \end{array}}$$

Our next step will be to "defer" the creation of the fake ciphertexts, generating them on demand when the ciphertext queue is viewed by the adversary. To do this, we maintain a log which saves the length of messages being sent, and also lets us know when to remove ciphertexts from the log. This gives us:

Γ^5

$\Gamma^4 =$

pub $l_{1\to 2}, l_{2\to 1} \leftarrow \text{FifoQueue.new}()$
view $c_{1\to 2}, c_{2\to 1} \leftarrow \text{FifoQueue.new}()$
view $m_{1\to 2}, m_{2\to 1} \leftarrow \text{FifoQueue.new}()$
$(\text{sk}_i, \text{pk}_i) \xleftarrow{\$} \text{Gen}()$

PKs():
  return $(\text{pk}_1, \text{pk}_2)$

$c_{i\to\bar{\imath}}()$:
  while $\text{cmd} \leftarrow \text{l}_{i\to\bar{\imath}}.\text{pop}() \neq \bot$:
    if $\text{cmd} = \texttt{pop}$:
      $c_{i\to\bar{\imath}}.\text{pop}()$
    if $\text{cmd} = (\texttt{push}, |m|)$:
      $r \xleftarrow{\$} \mathbf{M}(|m|)$
      $c_{i\to\bar{\imath}}.\text{push}(\text{Enc}(\text{pk}_{\bar{\imath}}, r))$
  return $c_{i\to\bar{\imath}}$

$\text{Send}_i(m)$:
  $l_{i\to\bar{\imath}}.\text{push}((\texttt{push}, |m|))$
  $m_{i\to\bar{\imath}}.\text{push}(m)$

$\text{Recv}_i()$:
  $m \leftarrow$ **await** $\text{m}_{\bar{\imath}\to i}.\text{pop}()$
  $l_{i\to\bar{\imath}}.\text{push}((\texttt{pop}, |m|))$
  return m

But, at this point the behavior of Send_i and Recv_i is identical to that in $\mathcal{Q}$, allowing us to write:

$$\Gamma^5 = \boxed{\begin{array}{ll} \colorbox{lightblue}{S} & \\ & \\ \textbf{view } c_{1\to 2}, c_{2\to 1} \leftarrow \text{FifoQueue.new}() & \\ (\text{sk}_i, \text{pk}_i) \xleftarrow{\$} \text{Gen}() & \\ & \\ \underline{\text{PKs}():} & \underline{c_{i\to\bar{\imath}}():} \\ \quad \textbf{return } (\text{pk}_1, \text{pk}_2) & \quad \textbf{while } \text{cmd} \leftarrow 1_{i\to\bar{\imath}}.\text{pop}() \neq \bot: \\ & \quad\quad \textbf{if } \text{cmd} = \texttt{pop}: \\ & \quad\quad\quad c_{i\to\bar{\imath}}.\text{pop}() \\ & \quad\quad \textbf{if } \text{cmd} = (\texttt{push}, |m|): \\ & \quad\quad\quad r \xleftarrow{\$} \mathbf{M}(|m|) \\ & \quad\quad\quad c_{i\to\bar{\imath}}.\text{push}(\text{Enc}(\text{pk}_{\bar{\imath}}, r)) \end{array}} \circ \text{Inst}_{\text{H}}(\mathscr{Q})$$

$$\otimes$$

$$1(\text{Send}_i, \text{Recv}_i)$$

which concludes this part of our proof, having written out our simulator, and proven that $\text{Inst}_{\text{H}}(\mathscr{P}) \overset{\epsilon}{\approx} \text{SimInst}_{S,\text{H}}(\mathscr{Q})$.

Malicious Case: Without loss of generality, we can consider the case where P_1 is malicious. This is because the difference between the parties is just a matter of renaming variables, so the case where P_2 is malicious would be the same. Let M denote this corruption model. We prove that $\mathscr{P} \overset{0}{\rightsquigarrow}_{\{\text{M}\}} \mathscr{Q}$, which naturally implies the slightly higher upper bound of $2 \cdot \text{IND}$.

We start by unrolling $\text{Inst}_{\text{M}}(\mathscr{P})$, to get:

$$
\text{Inst}_{\text{M}}(\mathscr{P}) = \boxed{\begin{array}{ll}
\Gamma^1 & \\
& \\
\textbf{view } c_{1\to 2}, c_{2\to 1} \leftarrow \text{FifoQueue.new}() & \\
(\text{sk}_2, \text{pk}_1) \leftarrow \text{Keys}_2() & \\
& \\
\underline{\text{PKs}():} & \underline{\text{Keys}_1():} \\
\quad \textbf{return } (\text{pk}_1, \text{pk}_2) & \quad \textbf{return super}.\text{Keys}_1() \\
& \\
\underline{\text{Send}_1(c):} & \underline{\text{Recv}_1():} \\
\quad c_{1\to 2}.\text{push}(c) & \quad \textbf{return await } c_{2\to 1}.\text{pop}() \\
& \\
\underline{\text{Send}_2(m):} & \underline{\text{Recv}_2(m):} \\
\quad c \leftarrow \text{Enc}(\text{pk}_1, m) & \quad c \leftarrow \textbf{await } c_{1\to 2}.\text{pop}() \\
\quad c_{2\to 1}.\text{push}(c) & \quad \textbf{return } \text{Dec}(\text{sk}_2, c)
\end{array}} \circ \text{Keys}
$$

The key affordances for malicious corruption are that the adversary can now see the output of Keys_1, including their secret key, and the public key of the other party, and that they have direct access to $c_{1\to 2}$. This allows them to send potentially “fake” ciphertexts to the other party, rather than going through the decryption process.

Next, we explicitly include the code of Keys, and also include an additional key pair, used in Recv_2, this key pair encrypts and then immediately decrypts the message being received, and thus has no effect by the correctness property of encryption. Writing this out, we get:

$$
\Gamma^1 \circ \mathrm{Keys} = \boxed{
\begin{array}{ll}
\Gamma^2 & \\
& \\
\multicolumn{2}{l}{\textbf{view } c_{1\to 2}, c_{2\to 1} \leftarrow \mathrm{FifoQueue.new}()} \\
\multicolumn{2}{l}{(\mathrm{sk}_1, \mathrm{pk}_1),\ (\mathrm{sk}_2, \mathrm{pk}_2),\ (\mathrm{sk}_2', \mathrm{pk}_2') \leftarrow \mathrm{Gen}()} \\
& \\
\underline{\mathrm{PKs}():} & \underline{\mathrm{Keys}_1():} \\
\quad \textbf{return } (\mathrm{pk}_1, \mathrm{pk}_2) & \quad \textbf{return } (\mathrm{sk}_1, \mathrm{pk}_2) \\
& \\
\underline{\mathrm{Send}_1(c):} & \underline{\mathrm{Recv}_1():} \\
\quad c_{1\to 2}.\mathrm{push}(c) & \quad \textbf{return await } c_{2\to 1}.\mathrm{pop}() \\
& \\
\underline{\mathrm{Send}_2(m):} & \underline{\mathrm{Recv}_2(m):} \\
\quad c \leftarrow \mathrm{Enc}(\mathrm{pk}_1, m) & \quad c \leftarrow \textbf{await } c_{1\to 2}.\mathrm{pop}() \\
\quad c_{2\to 1}.\mathrm{push}(c) & \quad m \leftarrow \mathrm{Dec}(\mathrm{sk}_2, c) \\
& \quad c' \leftarrow \mathrm{Enc}(\mathrm{pk}_2', m) \\
& \quad m \leftarrow \mathrm{Dec}(\mathrm{sk}_2', c') \\
& \quad \textbf{return } m
\end{array}
}
$$

The next step we perform is a bit of a trick. We swap the names of sk_2 and sk_2', as well as pk_2 and pk_2', after all, renaming has no effect on a system. We also create a separate message queue $m_{1\to 2}$ which will be used to send messages directly.

This gives us:

$\Gamma^2 =$

$\boxed{\Gamma^3}$

view $m_{1\to 2}, c_{1\to 2}, c_{2\to 1} \leftarrow \text{FifoQueue.new}()$
$(\text{sk}_1, \text{pk}_1),\ (\text{sk}_2, \text{pk}_2),\ (\text{sk}'_2, \text{pk}'_2) \leftarrow \text{Gen}()$

<u>PKs():</u>
  return $(\text{pk}_1, \text{pk}'_2)$

<u>Keys$_1$():</u>
  return $(\text{sk}_1, \text{pk}'_2)$

<u>Send$_1(c)$:</u>
  $c_{1\to 2}.\text{push}(c)$
  $m \leftarrow \text{Dec}(\text{sk}'_2, c)$
  $m_{1\to 2}.\text{push}(m)$

<u>Send$_2(m)$:</u>
  $c \leftarrow \text{Enc}(\text{pk}_1, m)$
  $c_{2\to 1}.\text{push}(c)$

<u>Recv$_1$():</u>
  return await $c_{2\to 1}.\text{pop}()$

<u>Recv$_2(m)$:</u>
  $c \leftarrow$ **await** $c_{1\to 2}.\text{pop}()$
  $m \leftarrow$ **await** $m_{1\to 2}.\text{pop}()$
  $c' \leftarrow \text{Enc}(\text{pk}_2, m)$
  $m \leftarrow \text{Dec}(\text{sk}_2, c')$
  return m

Notice that at this point sk_2 and pk_2 now don't actually do anything, since they don't actually modify the message in Recv$_2$. The main remaining barrier to writing this as a simulator over $\mathscr{Q}$ is that the ciphertext queues $c_{i\to\bar{\imath}}$ are modified both in functions we control Send$_1$ and Recv$_1$, but also in the two functions which we don't control Send$_2$, and Recv$_2$, and will eventually need to become pass through functions for $\mathscr{Q}$.

For Recv$_2$, it modifies $c_{1\to 2}$ by popping elements off of it. We can emulate this behavior by reading the access log of $l_{1\to 2}$ we get from $\mathscr{Q}$, and using the pop commands inside to modify $c_{1\to 2}$ when necessary.

For Send$_2$, our task is a bit harder, since we need to create an encryption of m, and the log will only contain $|m|$. However, our simulator over $\mathscr{Q}$ will be able to receive messages on behalf of the first party, allowing us to retrieve the message, and then create a simulate ciphertext by encrypting it.

Putting these ideas together, we write:

$$
\Gamma^3 = \boxed{\begin{array}{ll}
\multicolumn{2}{l}{S} \\
\\
\multicolumn{2}{l}{c_{1\to 2}, c_{2\to 1} \leftarrow \text{FifoQueue.new}()} \\
\multicolumn{2}{l}{(\text{sk}_1, \text{pk}_1),\ (\text{sk}_2', \text{pk}_2') \leftarrow \text{Gen}()} \\
\\
\underline{\text{PKs}():} & \\
\quad \textbf{return } (\text{pk}_1, \text{pk}_2') & \underline{\text{Update}_{1\to 2}():} \\
 & \quad \textbf{while } \text{cmd} \leftarrow l_{1\to 2}.\text{pop}() \neq \bot: \\
\underline{\text{Keys}_1():} & \qquad \textbf{if } \text{cmd} = \texttt{pop}: \\
\quad \textbf{return } (\text{sk}_1, \text{pk}_2') & \qquad\quad c_{1\to 2}.\text{pop}() \\
\\
\underline{c_{i\to \bar{\imath}}():} & \underline{\text{Update}_{2\to 1}():} \\
\quad \text{Update}_{i\to \bar{\imath}}() & \quad \textbf{while } \text{cmd} \leftarrow l_{2\to 1}.\text{pop}() \neq \bot: \\
\quad \textbf{return } c_{i\to \bar{\imath}} & \qquad \textbf{if } \text{cmd} = (\texttt{push}, \bullet): \\
 & \qquad\quad m \leftarrow \textbf{await super}.\text{Recv}_1() \\
\underline{\text{Send}_1(c):} & \qquad\quad c_{2\to 1}.\text{push}(\text{Enc}(\text{pk}_1, m)) \\
\quad \text{Update}_{1\to 2}() & \\
\quad c_{1\to 2}.\text{push}(c) & \underline{\text{Recv}_1():} \\
\quad m \leftarrow \text{Dec}(\text{sk}_2', c) & \quad \text{Update}_{2\to 1}() \\
\quad \textbf{super}.\text{Send}_1(m) & \quad \textbf{return await } c_{2\to 1}.\text{pop}()
\end{array}} \circ \text{Inst}_{\text{M}}(\mathscr{Q})
$$

$$
\otimes
$$

$$
1(\{\text{Send}_2, \text{Recv}_2\})
$$

We make sure to update both queues whenever necessary. This includes when they're viewed by the adversary, but also whenever we modify the queues ourselves, so that we've popped or pushed everything that we need to before using the queue.

This simulator is effectively creating a man-in-the-middle attack on the adversary, by providing them with the wrong public key, allowing them to decrypt the ciphertexts they see. On the other side, the simulator can receive messages on behalf of the adversary, and then reencrypt them to create the fake ciphertext queue.

Having now proved the upper bound for all the corruption models in $\mathscr{C}$, we conclude that our claim holds.

■

5.2 Drawing a Random Value

The basic goal of this section is to develop a protocol for securely choosing a common random value. This process should such that no party can bias the resulting value. We will follow the common paradigm of "commit-reveal", where the parties first commit to their random values, then wait for all these commitments to have been made, before finally opening the random values and mixing them together. This ensures that no party can bias the result, since they have to choose their contribution before learning any information about the result.

We start by defining the ideal protocol for drawing a random value. We'll be working over an additive group $\mathbb{G}$, and assuming that we have parties numbered $1, \ldots, n$. The core functionality we use allows each party to set a random value, and then have the functionality add them together. This is contained in Game 5.6.

$F[\text{Add}]$

$x_1, \ldots, x_n \leftarrow \bot$

$(1)\text{Add}_i(x)$:	$\text{Leak}()$:
$x_i \leftarrow x$	**if** $\exists i.\ x_i = \bot$:
wait $\forall i.\ x_i \neq \bot$	**return** $(\texttt{waiting}, \{i \mid x_i = \bot\})$
return $\sum_i x_i$	**return** $(\texttt{done}, \sum_i x_i)$

Game 5.6: Addition Functionality

This game works by first collecting a contribution from each party, and then adding them together. At any point after all contributions have been gathered, the adversary can also see their sum through the Leak function. Note that we only allow a contribution to be provided once, as marked by the (1) in front of the function. This will be the case for the random sampling as well.

Using this functionality, we create an ideal protocol for sampling a random value, defined in Protocol 5.7

The idea is that each party samples a random value, and then submits that to the addition functionality. If at least one of the values was sampled randomly, then the final result is also random. Technically, this is an *endemic* random functionality, in the sense that malicious parties are allowed to choose their own randomness. We also don't embed the $F[\text{Add}]$ functionality into the protocol itself, which makes the ideal protocol technically $\mathscr{P}[\text{IdealRand}] \circ F[\text{Add}]$. We do this to allow considering a slightly modified variant of the protocol, which uses a version of the addition functionality leaking more information, defined in Game 5.8.

$\mathscr{P}[\text{IdealRand}]$ is characterized by:

- $F := 1(\text{Add})$,
- $\text{Leakage} = \{\text{Leak}\}$,
- And n players defined via the following system, for $i \in [n]$:

$$
\boxed{
\begin{array}{l}
P_i \\
\\
\underline{(1)\text{Rand}_i():} \\
\quad x \xleftarrow{\$} \mathbb{G} \\
\quad \textbf{return await } \text{Add}_i(x)
\end{array}
}
$$

Protocol 5.7: Ideal Random Protocol

$$
\boxed{
\begin{array}{ll}
F[\text{Add}'] & \\
 & \\
x_1, \ldots, x_n \leftarrow \bot & \\
 & \\
\underline{(1)\text{Add}_i(x):} & \underline{\text{Leak}():} \\
\quad x_i \leftarrow x & \quad \textbf{if } \exists i.\ x_i = \bot: \\
\quad \textbf{wait } \forall i.\ x_i \neq \bot & \quad\quad \textbf{return } (\texttt{waiting}, \{i \mid x_i = \bot\}) \\
\quad \textbf{return } \sum_i x_i & \quad \textbf{return } (\texttt{done}, [x_i \mid i \in [n]])
\end{array}
}
$$

Game 5.8: Addition Functionality

The difference in $F[\text{Add}']$ is simply that the entire list of contributions is leaked, rather than just their sum. We introduce this functionality because it will be simpler to show that our concrete protocol is simulated by this slightly stronger functionality. Thankfully, the difference doesn't matter in the end, because we can simulate the stronger functionality from the weaker one.

Claim 5.2. Let $\mathscr{C}$ be the corruption class where all up to $n-1$ parties are corrupted. It then holds that:

$$
\mathscr{P}[\text{IdealRand}] \circ F[\text{Add}'] \overset{0}{\leadsto}_{\mathscr{C}} \mathscr{P}[\text{IdealRand}] \circ F[\text{Add}]
$$

Proof: The crux of the proof is that we can simply invent random shares for the honest parties, subject to the constraint that the sum of all shares is the same.

Now, onto the more formal proof. We assume, without loss of generality, that $1, \ldots, h$ are the indices of the honest parties, and $h+1, \ldots, m$ the semi-honest parties. Another convention we use is that j is used as a subscript for semi-honest parties, and k for malicious parties.

The only difference between the instantiation of both protocols lies in Leak. Otherwise, the behavior of all the functions is identical. Thus, we simply need to write a simulator for that function. The basic idea is to intercept calls to the corrupted parties to learn their contributions, and then simply invent some fake but plausible contributions for the honest parties.

This gives us:

S

faked $\leftarrow$ `false`
$x'_1, \ldots, x'_n \leftarrow \bot$

(1)$\text{Add}_k(x)$:
 $x'_k \leftarrow x$
 return $\text{Add}_k(x)$

$\text{Contribution}_j()$:
 assert(`call`, x) $\in$ log.Add_j
 return x

Leak():
 out $\leftarrow$ **super**.Leak()
 if out = (`waiting`, on):
  return (`waiting`, on)
 if faked = `false`:
  faked $\leftarrow$ `true`
  for $j \in h+1, \ldots, m$:
   $x'_j \leftarrow \text{Contribution}_j()$
  $x'_2, \ldots, x_h \xleftarrow{\$} \mathbb{G}$
  $x'_1 \leftarrow \text{out} - \sum_{i \in [2,\ldots,n]} x_i$
 return (`done`, $[x'_i \mid i \in [n]]$)

$$\otimes$$
$$1(\ldots)$$

The shares of the malicious parties are obtained by catching them when the call to Add_k is made, whereas for the semi-honest party we instead fetch them from the log. Note that because the leakage is only made available once all the parties have contributed, we're guaranteed to have already seen the shares from the corrupted parties by the time we fake the other shares.

It should be clear that:

$$\text{Inst}_C(\mathscr{P}[\text{IdealRand}] \circ F[\text{Add}']) = \text{SimInst}_{S,C}(\mathscr{P}[\text{IdealRand}] \circ F[\text{Add}])$$

concluding our proof.

■

The next task on our hands is to write down the concrete protocol for sampling randomness via the commit-reveal paradigm. To do that, we first need to define an appropriate commitment functionality, which we do in Game 5.9

$F[\text{Com}]$

$c_1, \ldots, c_n \leftarrow \bot$
$o_1, \ldots, o_n \leftarrow \texttt{false}$

$\underline{(1)\text{Commit}_i(x)\text{:}}$
  $c_i \leftarrow x$

$\underline{(1)\text{Open}_i()\text{:}}$
  assert $c_i \neq \bot$
  $o_i \leftarrow \texttt{true}$

$\underline{\text{View}_i(x)\text{:}}$
  if $c_i = \bot$:
    return `empty`
  if $\neg o_i$:
    return `set`
  else:
    return $(\texttt{open}, c_i)$

Game 5.9: Commitment Functionality

This functionality acts as a one shot commitment for each participant. Each party can commit to a value, and then open it at a later point in time. At any time, each participant can view the state of another participant's commitment. This view tells us what stage of the commitment the participant is at, along with their committed value, once opened.

We can now define a protocol sampling randomness, thanks to this commitment scheme, in Protocol 5.10.

The idea is quite simple, everybody generates a random value, commits to it, and then once everybody has committed, they open the value, and sum up all the contributions. The result is, as we'll prove, a random value that no participant can bias.

Unfortunately, it's not quite the case that $\mathscr{P}[\text{Rand}]$ is simulated by $\mathscr{P}[\text{IdealRand}]$. The reason is a consequence of the timing properties of the protocols. Indeed, in $\mathscr{P}[\text{IdealRand}]$, it suffices to activate each participant once in order to learn the result, whereas in $\mathscr{P}[\text{Rand}]$, two activations are needed, once to commit, and another time to open.

Instead we introduce a sepate protocol, making use of a "synchronization" functionality, defined in Game 5.11.

This functionality allows the parties to first "synchronize", by waiting for each party to contribute, before being able to continue.

The protocol using this functionality is then called $\mathscr{Q}$, and defined in Protocol 5.12

$\mathscr{P}[\text{Rand}]$ is characterized by:

- $F := F[\text{Com}]$,
- Leakage $= \{\text{View}_1, \ldots, \text{View}_n\}$,
- And n players defined via the following system, for $i \in [n]$:

P_i

$(1)\text{Rand}_i()$:

$x \xleftarrow{\$} \mathbb{G}$
$\text{Commit}_i(x)$
wait $\forall i.\text{View}_i() \neq \texttt{empty}$
$\text{Open}_i()$
wait $\forall i.\text{View}_i() = (\texttt{open}, x_i)$
return $\sum_i x_i$

Protocol 5.10: Random Protocol

$F[\text{Sync}]$

view $\text{done}_1, \ldots, \text{done}_n \leftarrow \texttt{false}$

$(1)\text{Sync}_i()$:

$\text{done}_i \leftarrow \texttt{true}$
wait $\forall i.\ \text{done}_i = \texttt{true}$

Game 5.11: Synchronization Game

$\mathscr{Q}$ is characterized by:

- $F = F[\text{Sync}]$,
- Leakage $:= \{\text{done}_1, \ldots, \text{done}_n\}$,
- And n players defined by the following system, for $i \in [n]$:

P_i

$(1)\text{Rand}_i()$:

out $\leftarrow$ **await super**.$\text{Rand}_i()$
await $\text{Sync}_i()$
return out

Protocol 5.12: Synchronized Random Protocol

The full protocol we consider is $\mathscr{Q} \lhd (\mathscr{P}[\text{IdealRand}] \circ F[\text{Add}])$, which can perfectly simulate $\mathscr{P}[\text{Rand}]$, as we now prove.

Claim 5.3. Let $\mathscr{C}$ be the class of corruptions where up to $n-1$ parties are corrupt. Then it holds that:

$$\mathscr{P}[\text{Rand}] \overset{0}{\rightsquigarrow}_{\mathscr{C}} \mathscr{Q} \lhd (\mathscr{P}[\text{IdealRand}] \circ F[\text{Add}])$$

Proof: Thanks to the composition properties of protocols, it suffices to prove the above claim using $F[\text{Add}']$ instead, since we already proved that:

$$\mathscr{P}[\text{IdealRand}] \circ F[\text{Add}'] \overset{0}{\rightsquigarrow}_{\mathscr{C}} \mathscr{P}[\text{IdealRand}] \circ F[\text{Add}]$$

As before, we let $1, \ldots, h$ be the indices of honest parties, $h+1, \ldots, m$ the indices of semi-honest parties, and use i, j, k for denoting indices of honest, semi-honest, and malicious parties, respectively. We start by unrolling $\text{Inst}_C(\mathscr{P}[\text{Rand}])$, to get:

Γ^0

$x_1, \ldots, x_n, \text{rush}_{m+1}, \ldots, \text{rush}_n \leftarrow \bot$
$o_1, \ldots, o_n \leftarrow \texttt{false}$
$\text{log}_j \leftarrow \text{NewLog}()$

$\underline{(1)\text{Rand}_i():}$
  $x_i \xleftarrow{\$} \mathbb{G}$
  wait $\forall i.\ \text{View}_i \neq \texttt{empty}$
  $o_i \leftarrow \texttt{true}$
  wait $\forall i.\ \text{View}_i = (\texttt{open}, x_i)$
  return $\sum_i x_i$

$\underline{\text{View}_i():}$
  if $x_i = \bot$:
    return $\texttt{empty}$
  if $\neg o_i$:
    return $\texttt{set}$
  else:
    return $(\texttt{open}, c_i)$

$\underline{(1)\text{Rand}_j():}$
  $\text{log.Rand}_j.\text{push}(\texttt{input})$
  $x_i \xleftarrow{\$} \mathbb{G}$
  $\text{log}_j.\text{Commit}_j.\text{push}((\texttt{call}, x_i))$
  wait $\forall i.\ \text{View}_i \neq \texttt{empty}$
  $\text{log}_j.\text{Open}_j.\text{push}(\texttt{call})$
  $o_i \leftarrow \texttt{true}$
  wait $\forall i.\ \text{View}_i = (\texttt{open}, x_i)$
  return $\sum_i x_i$

$\underline{(1)\text{Commit}_k(x):}$
  $x_k \leftarrow x$

$\underline{(1)\text{Open}_k():}$
  assert $x_k \neq \bot$
  $o_k \leftarrow \texttt{true}$

Here we've just inlined the main elements of the game. The key difference for the semi-honest parties is that we're able to see the randomness they used, since they commit to it. For the malicious parties, they can commit to any value they want, and can also choose when to open their values.

We now rewrite this game slightly, to make the connection with what we're trying to simulate a bit clearer:

Γ^1

$x_1, \ldots, x_n, \text{rush}_{m+1}, \ldots, \text{rush}_n \leftarrow \bot$
$\text{done}_1, \ldots, \text{done}_n \leftarrow \texttt{false}$
$\text{log}_j \leftarrow \text{NewLog}()$

$\underline{(1)\text{Rand}_i():}$
$\quad x_i \xleftarrow{\$} \mathbb{G}$
$\quad \textbf{wait } \forall i.\ \text{View}_i \neq \texttt{empty}$
$\quad \text{done}_i \leftarrow \texttt{true}$
$\quad \textbf{wait } \forall i.\ \text{View}_i = (\texttt{open}, x_i)$
$\quad \textbf{return } \sum_i x_i$

$\underline{\text{View}_i():}$
$\quad \textbf{if } \text{Leak}() = (\texttt{waiting}, s) \land i \in s:$
$\quad\quad \textbf{return } \texttt{empty}$
$\quad \textbf{else if } \text{done}_i:$
$\quad\quad \textbf{if } \text{rush}_i \neq \bot:$
$\quad\quad\quad \textbf{return } (\texttt{open}, \text{rush}_i)$
$\quad\quad \textbf{assert } (\texttt{done}, [y_i]) = \text{Leak}()$
$\quad\quad \textbf{return } (\texttt{open}, y_i)$
$\quad \textbf{return } \texttt{set}$

$\underline{\text{log}_j():}$
$\quad \text{log}'_j \leftarrow \text{NewLog}()$
$\quad \text{log}'_j.\text{Rand}_j \leftarrow \text{log}_j.\text{Rand}_j$
$\quad \text{log}'_j.\text{Commit}_j \leftarrow \text{log}_j.\text{Add}_j$
$\quad \text{log}'_j.\text{Open}_j \leftarrow \text{log}_j.\text{Sync}_j$
$\quad \textbf{return } \text{log}'_j$

$\underline{(1)\text{Rand}_j():}$
$\quad \text{log}_j.\text{Rand}_j.\text{push}(\texttt{input})$
$\quad x_i \xleftarrow{\$} \mathbb{G}$
$\quad \text{log}_j.\text{Add}_j.\text{push}((\texttt{call}, x_i))$
$\quad \textbf{wait } \forall i.\ \text{View}_i \neq \texttt{empty}$
$\quad \text{log}_j.\text{Sync}_j.\text{push}(\texttt{call})$
$\quad o_i \leftarrow \texttt{true}$
$\quad \textbf{wait } \forall i.\ \text{View}_i = (\texttt{open}, x_i)$
$\quad \textbf{return } \sum_i x_i$

$\underline{(1)\text{Commit}_k(x):}$
$\quad \text{rush}_k \leftarrow x$
$\quad x_k \leftarrow x$

$\underline{(1)\text{Open}_k():}$
$\quad \textbf{assert } \text{rush}_k \neq \bot$
$\quad \text{done}_k \leftarrow \texttt{true}$

$\underline{\text{Leak}():}$
$\quad \textbf{if } \exists i.\ x_i = \bot:$
$\quad\quad \textbf{return } (\texttt{waiting}, \{i \mid x_i = \bot\})$
$\quad \textbf{return } (\texttt{done}, [x_i \mid i \in [n]])$

First of all, we've renamed several variables, like o_i becoming done_i, which has no effect on the game, of course. We've also introduced a secondary set of variables rush_k to hold the values the malicious parties are committing to. We do this to stress the fact that the simulator will be able to see and capture these values. We also modify the logging in the semi-honest parties to use different names, reflecting what will happen in the eventual semi-honest party of $\mathcal{Q}$. This requires introducing a log_j function which will produce a simulated log by renaming these entries.

Finally, the biggest change is in the View_i functions. We've rewritten the logic to be based on this Leak method we've introduced, which informs of us the status of the contributions. This gives us enough information to simulate the views accurately. For the honest parties, we know that they'll only open their values after everybody has already committed, so the assertion will always pass. This may not be the case for malicious parties, which may "rush", opening their values *before* the other parties have finished comitting. This is why it's important to keep track of their commitments separately, so that we can present them inside the view, if necessary.

At this point, the next step is to realize that all of this logic can in fact work inside of a simulator, written as:

S

$\text{rush}_{m+1}, \ldots, \text{rush}_n \leftarrow \bot$

$\underline{\text{View}_i():}$
- **if** $\text{Leak}() = (\texttt{waiting}, s) \land i \in s$:
  - **return** $\texttt{empty}$
- **else if** done_i:
  - **if** $\text{rush}_i \neq \bot$:
    - **return** $(\texttt{open}, \text{rush}_i)$
  - **assert** $(\texttt{done}, [y_i]) = \text{Leak}()$
  - **return** $(\texttt{open}, y_i)$
- **return** $\texttt{set}$

$\underline{(1)\text{Commit}_k(x):}$
- $\text{rush}_k \leftarrow x$
- $\text{Add}_k(x)$

$\underline{(1)\text{Open}_k():}$
- **assert** $\text{rush}_k \neq \bot$
- $\text{Sync}_k()$

$\underline{\text{log}_j():}$
- $\text{log}'_j \leftarrow \text{NewLog}()$
- $\text{log}'_j.\text{Rand}_j \leftarrow \textbf{super}.\text{log}_j.\text{Rand}_j$
- $\text{log}'_j.\text{Commit}_j \leftarrow \textbf{super}.\text{log}_j.\text{Add}_j$
- $\text{log}'_j.\text{Open}_j \leftarrow \textbf{super}.\text{log}_j.\text{Sync}_j$
- **return** log'_j

$$\otimes$$
$$1(\ldots)$$

And this concludes our proof, having shown that:

$$\text{Inst}_C(\mathscr{P}[\text{Rand}]) = \text{SimInst}_{S,C}(\mathscr{Q} \lhd (\mathscr{P}[\text{IdealRand}] \circ F[\text{Add}]))$$

■

6 Differences with UC Security

7 Further Work

8 Conclusion

References

[BDF+18] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State separation for code-based game-playing proofs. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 222–249. Springer, Heidelberg, December 2018.

[Mei22] Lúcás Críostóir Meier. MPC for group reconstruction circuits. Cryptology ePrint Archive, Report 2022/821, 2022. `https://eprint.iacr.org/2022/821`.

A Additional Game Definitions

In this section, we include explicit definitions of several games we use throughout the rest of this work. While we expect these notions to be familiar, we think the precise details are worth spelling out here.

A.1 Encryption

An encryption scheme consists of types $\mathbf{K}, \mathbf{M}, \mathbf{C}$, along with probabilistic functions $\text{Enc} : \mathbf{K} \times \mathbf{M} \xleftarrow{\$} \mathbf{C}$ and $\text{Dec} : \mathbf{K} \times \mathbf{C} \to \mathbf{M}$. By $\mathbf{M}(|m|)$ we denote the distribution of messages with the same length as m. We require that $\mathbf{K}$ and $\mathbf{M}(|m|)$ are efficiently sampleable.

The encryption scheme must satisfy a correctness property:

$$\forall k \in \mathbf{K},\ m \in \mathbf{M}.\ P[\text{Dec}(\text{Enc}(k, m)) = m] = 1$$

Encrypting and then decrypting a message should return that same message.

The security of an encryption scheme can be captured by the following game:

IND-CPA$_b$

$k \xleftarrow{\$} \mathbf{K}$

Challenge(m_0):
  $m_1 \xleftarrow{\$} \mathbf{M}(|m|)$
  return $\text{Enc}(k, m_b)$

Enc(m):
  return $\text{Enc}(k, m)$

In essence, an adversary cannot distinguish between an encryption of a message of their choice and that of a random message, even if they can encrypt messages at will.