Towards Modular Foundations for Protocol Security

Lúcás Críostóir Meier
lucas@cronokirby.com

February 13, 2023

Abstract

Universally composable (UC) security [Can00] is the most widely used framework for analyzing the security of cryptographic protocols. Many variants and simplifications of the framework have been proposed and developed, nonetheless, many practitioners find UC proofs to be both difficult to construct and understand. We remedy this situation by proposing a new framework for protocol security. We believe that our framework provides proofs that are both easier to write, but also more rigorous, and easier to understand. Our work is based on state-separable proofs [BDF+18], allowing for *modular* proofs, by decomposing complicated protocols into simple components.

1 Introduction

Universally composable (UC) security [Can00] has seen widespread success since its introduction over two decades ago, becoming the dominant framework for analyzing the security of cryptographic protocols.

This success is understandable, because the guarantees the framework provides are very useful. In essence, when a protocol is proved secure in the UC framework, then arbitrary instances of that protocol can be used concurrently in the context of a larger system, without modifying the behavior of these protocols. This allows the study of isolated components to be used to guarantee the security of a system as a whole; this modular approach is essential to being able to scale formal analysis to larger systems.

However, the framework is not ubiquitous. For many cryptographic schemes, standalone security with games is a more suitable approach. Even for some protocols, the use of game-based security remains popular. The analysis of threshold signatures and messaging protocols has seen the development of increasingly intricate games, which describe all the ways in which an adversary can attack a protocol. This is the main disadvantage of game-based security for protocols: it's not always clear what the "right" game is to analyze the security of a pro-

tocol, and these games often need to provide explicit capabilities to the adversary.

This problem is alleviated somewhat in UC security. The security goals of a protocol are described by an "ideal functionality", which represents how the protocol could function if one had access to a perfectly trusted third party. This makes it easier to determine if a given notion of security reflects the kinds of attacks that need to be analyzed.

The game-based approach is still sometimes preferred because of its perceived ease of use as compared to UC proofs. Some of these difficulties are inherent: because UC security provides stronger guarantees than standalone security, it's not surprising that proofs would involve more "work".

Some of these difficulties don't seem to be inherent though, which is why a series of works have provided improvements, simplifications, and variants of UC security. GNUC [HS15] was an early variant, simplifying many aspects of UC, and also patching several foundational gaps present in the paper at the time.

One disadvantage of developing a new framework is that proofs in one framework may not necessarily or automatically translate to UC proofs. One approach to addressing this is to develop a "higher level" language for simpler proofs, which is then compiled down to an actual UC proof. This was done in [CCL15], which provided a simplified version of UC, suitable for the common setting of multiparty computation, but also a way to interpret proofs in this simplified model as actually being UC proofs.

Another interesting alternative to UC is that presented in [CD+15]. This approach defines a kind of UC security in terms of a calculus for *interactive systems*, and their composition. This is an interesting departure from the interactive Turing machine foundations, and does away with many inessential details. This approach is the most similar one to the framework we develop in this work.

In practice, UC proofs are often quite informal, without explicitly mentioning the various details that the formalism might require. For example, the framework might specify protocols in terms of interactive Turing machines, but in practice, proofs are written with an informal description of what the protocol does. We think that this informality actually makes proofs harder to write and understand, because it isn't clear what exactly a proof can consist of, nor what certain informal patterns mean precisely.

In this work, we propose a new framework for analyzing the security of protocols, which we believe to be both less informal, but also simpler to understand and use.

1.1 Our Framework

Our framework—which we call, "modular protocol security", or "MPS"—attempts to provide a simple, high-level language for reasoning about protocol security, while also having a well-defined formal model that is close to how proofs are actually written.

MPS tries to be *modular*, in the sense that large proofs for complicated protocols can be built up from smaller proofs for simpler protocols.

The first way we try and achieve this is by allowing modular specifications of protocols: being able to describe a protocol as the composition of other protocols. The two fundamental operations we have are:

1. tensoring, written $\mathscr{P} \otimes \mathscr{Q}$, which allows writing a protocol as involving two distinct protocols running at the same time,
2. composition, written $\mathscr{P} \triangleleft \mathscr{Q}$, which allows the participants in one protocol to use another as a kind of "sub-protocol", in which each player may play several roles.

These operations can simplify proofs by allowing large protocols to be decomposed into smaller components, and then for security to be argued component by component.

The second approach is to have the notion of simulation be between *protocols*, rather than between a protocol and an ideal functionality, as in UC security. The ultimate goal is to prove that a protocol can be simulated by some clear ideal functionality, but this often requires writing a complicated simulator which can achieve this large jump all at once. By allowing protocols to be simulated by other protocols, we can transform this jump into several smaller hops, which are then composed together. This can help break down a large proof into a series of simpler proofs.

MPS builds on state-separable proofs [BDF+18], a recent framework for standalone security with games. Our work can be seen as an attempt to lift the modular properties of this framework for games into a modular framework for protocol security. Ultimately, the semantics of protocols will be defined in terms of state-separable games.

This provides an interesting advantage, in that proofs and techniques using games can be used to reason about the security of protocols. This can also help motivate complicated games used in the analysis of protocols, which can be seen as being related to protocols written in this framework.

We also take the opportunity to present state-separable proofs in a more formal way, filling in several proofs left as sketches in the original paper. Instead of using interactive Turing machines as our foundational object, like in UC security, we

instead simply assume the existence of computable randomized functions, and some pseudo-code to describe them.

1.2 Overview

Here we provide a basic overview of the rest of this work.

In Section 2, we develop the framework of state-separable proofs for standalone security from scratch, filling some gaps left in the original paper, and proving a few additional properties that we'll be needing in the rest of this work.

In Section 3, we generalize state-separable games into *systems*, which are a kind of game that has the ability to communicate with other games by sending and receiving messages along a channel. We also need to develop a notion of *asynchronous* games in this section, allowing us to model games which can delay answers to queries, if they're not yet ready to provide them. This arises naturally with channels, since a game may be waiting to receive a message along a channel before being able to respond.

In Section 4, we use the newly developed notion of systems to define *protocols*. We then define various ways to compose protocols together, then notions of equality and simulation for protocols, before showing that the various ways of composing protocols behave well with respect to equality and simulation, allowing us to decompose proofs in a modular fashion. We also describe how to incorporate global functionalities, such as a common random oracle, into our framework.

In Section 5, we provide a couple of examples of proofs written in our framework. Our first example constructs a private channel from one that leaks all messages sent over it, by using a public key encryption scheme. Our second example constructs a protocol for sampling an unbiased random value, using the common "commit reveal" paradigm.

In Section 6, we compare the MPS framework we've developed with that of UC security, illustrating several differences between the two approaches.

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

By $x \mapsto f(x)$, we mean a function taking in an input x, and returning the value $f(x)$. Sometimes we'll need to extend this syntax to more complicated expressions, writing:

$$x \mapsto y \leftarrow f(x);\ g(x, y)$$

to mean a function taking in an input value x, then calling f to produce a value y, before then using both x and y to return the value $g(x, y)$.

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 $f : \mathtt{01}^* \xrightarrow{\$} \mathtt{01}^*$ 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, parameterized by a security parameter λ. Formally, this is in fact a function $f : \mathbb{N} \to \mathtt{01}^* \xrightarrow{\$} \mathtt{01}^*$, and we write $f_\lambda : \mathtt{01}^* \xrightarrow{\$} \mathtt{01}^*$ 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, parameterized by a security parameter λ, but 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 much time the function takes to execute on a given input $x \in \mathtt{01}^*$.

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 its random choices, or its input (as long as that input is of a reasonable size).

□

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

Considering efficient functions is essential for game-based security, because the vast majority of cryptographic techniques depend on assuming that some problems are "hard" for adversaries with bounded computational resources. Ironically, for protocol security, many protocols can be proven 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 let $P[f \rightarrow 1]$ denote the probability 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:

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

□

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 $P[f \rightarrow 1] = (1 - P[f \rightarrow 0])$, 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:

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

■

Another property not included in our proof that ε is a valid metric requires that if $f \neq g$, then $\varepsilon(f, g) > 0$. We omitted this property, because we haven't yet defined what = should mean for functions. Since we'd like this property to hold, we can simply define equality in such a way that it does.

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 $f : \bullet \xrightarrow{\$} \mathtt{01}$, by defining:

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

In other words, we look at the maximum difference across all possible inputs and outputs.

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. As an example, consider the function $\mathrm{id}(x)$ which

immediately returns x. This function is valid regardless of what type x has. Because of this, we might write this function formally as:

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

assigning it the type $\forall s.\ s \to s$. In this type, s is a quantified type variable, as indicated by the $\forall s$. Formally, we can see id as a function parameterized 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 parameterizing 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:

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

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 parameterized 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 parameterized by g, but we could write this explicitly as $f(x) := g \mapsto g(x) \oplus g(x)$. We could write $f : (\mathtt{01}^* \xrightarrow{\$} \mathtt{01}^*) \to (\mathtt{01}^* \xrightarrow{\$} \mathtt{01}^*)$ as a potential type in this example. We write $f[g]$ for the instantiation of a parameterized function f with an input function g. It might also be the case that g is itself parameterized, 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 parameterized 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* $\mathrm{In}(A)$, of size m,
- a permutation $\pi_{\mathrm{in}} : \mathrm{In}(A) \leftrightarrow [m]$,
- a set of *output names* $\mathrm{Out}(A)$, of size n,
- a permutation $\pi_{\mathrm{out}} : [n] \leftrightarrow \mathrm{Out}(A)$,
- a set of parameterized functions $f_1, \ldots, f_n : \forall s.\ \circlearrowleft_s^m \to \circlearrowleft_{(S,s)}$, each of which has a distinct name $n_i \in \mathrm{Out}(A)$.

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

■

Note that here $\circlearrowleft_s^m$ denotes a tuple type containing m values of type $\circlearrowleft_s$.

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

The motivation behind this definition 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 $f : \forall s.\ \circlearrowleft_s \to \circlearrowleft_{(S_1,s)}$ uses a stateful function $g : \circlearrowleft_{S_2}$, then the result is a stateful function $f[g] : \circlearrowleft_{(S_1,S_2)}$ 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. If g used a state type S_2', then $f[g]$ would have type $\circlearrowleft_{(S_1,S_2')}$ instead.

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 parameterized 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 that we can use $\circlearrowleft_s^m$ as the input type for the parameterized 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 describe this kind of interaction using the formal notion of package *composition*.

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

- a state type $(A.S, B.S)$,
- input names $\mathrm{In}(B)$,
- output names $\mathrm{Out}(A)$,
- $\pi_{\mathrm{in}} := B.\pi_{\mathrm{in}}$,
- $\pi_{\mathrm{out}} := A.\pi_{\mathrm{out}}$,
- 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 are 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 $(A \circ B) \circ C$. Before we can prove such properties, we need to define 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

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

□

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 using this new definition is the one used as an example before.

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 $(A.S, (B.S, C.S))$, and $((A.S, B.S), C.S)$ 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 of *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 one 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 $\mathrm{In}(X \circ Y) = \mathrm{In}(Y)$, and $\mathrm{Out}(X \circ Y) = \mathrm{Out}(X)$. Thus, since both $\mathrm{In}(B) = \mathrm{In}(B')$ and $\mathrm{Out}(B) = \mathrm{Out}(B')$ 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:

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

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:

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

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, allowing us to 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 $\mathrm{Out}(A) \cap \mathrm{Out}(B) = \emptyset$, we can define their tensoring $A \otimes B$ as a package characterized by:

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

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

- an input index assignment $\pi_{\mathrm{in}}(n)$ 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:

$$\mathrm{lift}_1(f) := (((s_1, s_2), s), i) \mapsto (s_1', o) \leftarrow f(s_1, i); (((s_1', s_2), s), o)$$
$$\mathrm{lift}_2(f) := (((s_1, s_2), s), i) \mapsto (s_2', o) \leftarrow f(s_2, i); (((s_1, s_2'), s), o)$$

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

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

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

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

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

□

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 parameterized 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 strong assumption.

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, and 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 match.

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: this allows us 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 $\mathrm{In}(A) \cap \mathrm{Out}(D) = \emptyset$ and $\mathrm{In}(C) \cap \mathrm{Out}(B) = \emptyset$

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

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:

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

On the right, we have:

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

For Out, on the left we have:

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

On the right, we have:

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

Now, we look at the functions.

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

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

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:

$$(h_1, \ldots) \mapsto \mathrm{lift}_1(A.f_i[\mathrm{lift}_1(\varphi_A(B.f_1)[h_{(B\otimes D).\pi_{\mathrm{in}}(B.\pi_{\mathrm{in}}^{-1}(j))} \mid j \in [B.m]]), \ldots])$$
$$(h_1, \ldots) \mapsto \mathrm{lift}_2(C.f_i[\mathrm{lift}_2(\varphi_C(D.f_1)[h_{(B\otimes D).\pi_{\mathrm{in}}(B.\pi_{\mathrm{in}}^{-1}(j))} \mid j \in [D.m]]), \ldots])$$

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:

$$(h_1, \ldots) \mapsto \mathrm{lift}_1(A.f_i[\varphi_A(B.f_1)[h_{\pi_{\mathrm{in}}((A\circ B).\pi_{\mathrm{in}}^{-1}(j))} \mid j \in [(A \circ B).m]]])$$
$$(h_1, \ldots) \mapsto \mathrm{lift}_2(C.f_i[\varphi_C(D.f_1)[h_{\pi_{\mathrm{in}}((C\circ D).\pi_{\mathrm{in}}^{-1}(j))} \mid j \in [(C \circ D).m]]])$$

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

The remaining difference is about

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

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:

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

But this is just $\mathrm{lift}_1(f[\mathrm{lift}_1(g)])$. 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 parameterized 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 $\mathrm{In}(G) = \emptyset$.

□

This is a very simple 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, $\mathrm{In}(\mathcal{A}) = \mathrm{Out}(P)$, and $\mathrm{Out}(\mathcal{A}) = \{\mathrm{run}\}$, where $\mathrm{run} : \bullet \xrightarrow{\$} \mathtt{01}$.

□

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 $\bullet \xrightarrow{\$} \mathtt{01}$. 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 the 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 encryption 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 routine 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 $\mathrm{Out}(C) \supseteq \mathrm{In}(A)$, and $\mathrm{Out}(C) \cap \mathrm{Out}(A) = \emptyset$.

We write:

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

□

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 $\mathrm{Out}(C)/\mathrm{In}(A)$, 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 $\mathrm{Out}(A) = \mathrm{Out}(B)$, and $\mathrm{In}(A) = \mathrm{In}(B)$.

□

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

□

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 $\mathrm{Out}(A)$, and the "back", via $\mathrm{In}(A)$, 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 $\mathcal{A}$ 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 $\mathcal{A} \circ \mathrm{Compl}_C(A) \equiv \mathcal{A} \circ \mathrm{Compl}_C(B)$, since composition and tensoring preserve literal equality. But, in that case, by defini-

tion 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 relate to showing 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:

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

by applying interchange. We can write this as:

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

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

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

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

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

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

In other words, we have:

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

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

2. We start with the same observation, that:

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

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:

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

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:

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

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

■

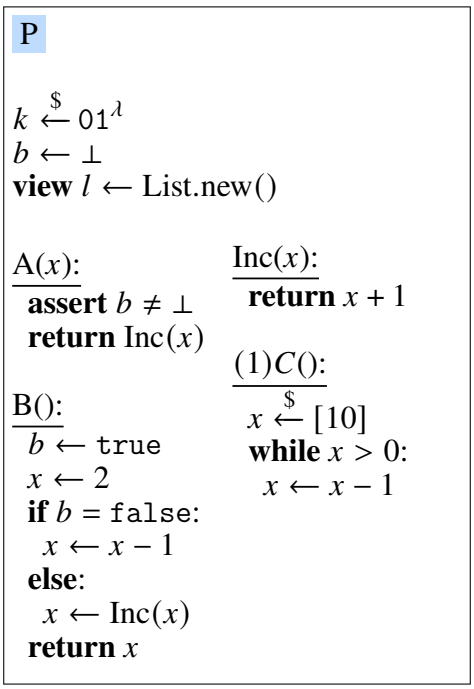
These lemmas form the conceptual crux of how proofs in the state-separable style work. If you want to prove that two large packages are indistinguishable, you do so by a series of observations, each of which breaks down the package as a composition of many smaller packages. Sometimes you'll be able to use theorems you've already proved, or problems assumed to be hard, in order to argue that small pieces are indistinguishable, and thus apply the lemmas we've just demonstrated in order to lift that indistinguishability to the large composition. By applying a series of such hops, you eventually produce a reduction of the security of this large package to that of several smaller packages.

2.6 Syntactical Conventions for Packages

In the previous subsections, we developed a formal model of how packages work. In practice, packages are described using a kind of pseudo-code, which corresponds with these formal objects. Some some of the rules governing packages are also relaxed in practice. In this subsection we give some examples of how

this pseudo-code works. Note that the details here aren't essential, and one could imagine using a different kind of pseudo-code instead.

We start with an example package, containing various syntactical constructs, which we'll then explain in more detail.



So, the basic idea is that a package is usually described by a box like this, with the name of the package—P, in this case—at the top of the box. A package has some initialization code, along with exported functions. In this case, the exported functions are A, B, C, Inc, and l.

The meaning of **view** is that the value is exported in a read only fashion. So, there's a function l which copies the list l and then returns it. The caller can modify their copy, but this has no effect on the original list.

Now, one slight deviation from the formal specification is that we allow a package to call functions that it exports internally, like we do for Inc. The semantics of this are that the code of Inc are inlined at the call site. So, in this case, every place where Inc(x) is used can be replaced with just $x + 1$.

We also have standard control flow constructs, like **if**, **else**, **while**, **for**, **return**, etc. Functions don't have to return a value, in which case we assume they return some pre-defined dummy value, like $\bullet$.

Another construct we have is **assert**. This should be seen as immediately returning a special value indicating that an assertion failed, if the condition is indeed false. This is useful to restrict when a function can be called. One very common

such restriction is on the number of times a function can be called. Since we wrote $(1)C$, we're indicating that this function can only be called once. This is shorthand for having a variable keep track of the number of calls, and an assertion checking that this count is low enough.

Another slight deviation from the formal specification is the use of initialization code. Formally, a package just exports functions, it doesn't have any code running before those functions do. One way to add initialization is to have a special function—say, `Init`—which must be called before any other of the functions. The initialization code could then be placed there.

We also don't particularly care about the names and variables of variables and functions, as long as it's clear which packages are using what functions. If it's not ambiguous, we could refer to one of the functions in P as just A, but we may want to explicitly write down $P.A$, to disambiguate this function from another, say, $Q.A$. We might also tensor multiple versions of P together, calling one function A_1, and the other A_2, for example. Another common way to disambiguate names is to use $\mathbf{super}.A$, to refer to calling an input function A.

Sometimes, we'll also compare two packages for equality, with one package exporting more functions than the other. This is usually shorthand for writing $1(\ldots) \circ P = Q$, ignoring some of the functions provided by one of the packages.

We stress that these rules are merely conventions, and are intended to provide a means of clearly expressing what a package is doing, while also not being excessively verbose.

For further examples of how state-separable proofs work, we point the reader to the original paper [BDF+18], or to other works making use of the paradigm [Ros, Mei22].

3 Systems

The goal of this section is to extend the notion of packages to that of *systems*. Intuitively, systems are like packages, except that they can send messages to other systems via channels. This is very useful, since it lets us model the kind of interaction we need to describe protocols.

Continuing with the machine analogy, we can see packages like machines, arranged into rows, with each row using output functions provided by the row behind it. Systems have the same setup, except that now all the machines within a given row have the ability to communicate with each other via channels.

3.1 Asynchronous Packages

Before we get to channels, we first need to define a notion of packages that have *asynchronous* functions. This becomes necessary to have channels, since we need to be able to handle the case where a system is receiving a message along channel, and is waiting for that message to arrive. A natural way to model this is an asynchronous process, where a system can *yield* control back to the caller, indicating that it isn't able to provide an answer yet, because it's waiting on something else to happen first.

Syntactically, this gives us functions such as:

$$
\begin{aligned}
&\underline{\mathrm{F}(x_1)\text{:}} \\
&\quad x_2 \leftarrow \textbf{yield } 3 \\
&\quad \textbf{return } x_1 + x_2
\end{aligned}
$$

This function takes in an input x_1, and then immediately yield control to the caller, with the value 3. The caller can then resume the function with some value, which gets stored in the variable x_2, and the function returns $x_1 + x_2$. If this function were part of a package, it could now be called again, starting from the top once more.

While the intuition of yielding control is simple, defining it precisely is a bit more tricky. Ultimately, the definition we provide isn't very elegant, but we think it's a very straightforward approach providing a clear meaning to yield statements.

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:

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

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.

In many cases, **yield** is used purely to yield control, and not to exchange any value between the caller and the function. We have a special shorthand for this kind of use.

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. In other contexts, being able to yield intermediate values can be useful, but for modeling channels, we only need empty yields.

Very often, a package just wants to run another asynchronous process to completion. It's not enough to simply loop until the process completes, because this might cause an infinite loop, as some external intervention might be necessary to cause the process to make progress. Instead, we want to poll the process, and yield ourselves if the process is not yet ready. 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.

□

In practice, **await** is the most common way that asynchronous functions will be called. Most systems will await other functions directly, and maybe only adversaries will care about being able to see the underlying polling process.

However, 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:

select :
  $v_1 \leftarrow$ **await** $F_1(\ldots)$:
    $\langle \text{body}_1 \rangle$
  $\vdots$
  $v_n \leftarrow$ **await** $F_n(\ldots)$:
    $\langle \text{body}_n \rangle$

As follows:

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

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.

□

This kind of situation can arise quite often when defining protocols, where you might be waiting on a message from any one of several parties. Using a select statement lets a package wait for the first message that happens to arrive.

Another variant of waiting occurs when we want to wait for some *condition* to be true. For example, we could set up a lock over a shared value, and we might

need to wait for the lock to be free so we can modify the value. We model this kind of situation with a **wait** statement.

Definition 3.5 (Wait Statement). We define the semantics of **wait** ⟨cond⟩ as equivalent to:

while ¬⟨cond⟩:
yield

□

So, we simply keep yielding until the condition is true. This is simple, but surprisingly useful.

We've defined the various asynchronous gadgets we'll be needing, so the natural next step is to define a kind of package which uses these gadgets.

Definition 3.6 (Asynchronous Packages). An *asynchronous* package P is a package which uses the additional syntax from Definition 3.1 and Syntax 3.3, 3.4, 3.5.

□

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. Naturally, the definitions of $\circ$ and $\otimes$ for packages also generalize directly to asynchronous packages.

3.2 Defining Systems

Our next goal is to define systems, by first defining channels, and then giving them meaning in terms of asynchronous packages. We'll then define various composition operations for systems, and show that they satisfy similar properties to those of packages.

Our first task is defining channels. We start by just defining some syntax for using channels, and defer defining the precise meaning of this syntax until later.

Syntax 3.7 (Channels). Using channels involves two syntactic constructs:

1. $m \Rightarrow P$, for sending a message m on a channel P,
2. $m \Leftarrow P$, for receiving a message m on a channel P,
3. $n \leftarrow$ **test** P, for checking how many messages are on a channel P.

□

Like with functions, channels have distinct names. The two fundamental operations are sending messages, and receiving messages. We also add an operation for testing how many messages are waiting on a channel. This is useful to allow

a package to change its behavior based on whether or not a channel is empty, in which case **test** P will return 0. We consider testing to be a kind of operation that a system can do on the channels it's allowed to receive on.

Next, we need to give packages the ability to use these channels. We call these, *systems*.

Definition 3.8 (Systems). A *system* is a package which uses channels.

We denote by InChan(S) the set of channels the system receives on, or uses **test** 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 $\text{Chan}(A, B, \ldots) = \text{Chan}(A) \cup \text{Chan}(B) \cup \ldots$, and similarly for InChan and OutChan. The set of channels can be seen as another part of the interface of a system. A package has input and output functions, while a system additionally has input and output channels. Like with packages, this set is often implicit, based on whatever channels the system happens to use syntactically. We can also consider a system to be "using" channels that don't actually appear in the body of a package as well.

So far, we've defined what systems are, but we haven't formally defined what their semantics actually are, although we might already have some intuition, at this stage. The simplest way of defining the semantics of a system is to compile it down into an asynchronous package, which we developed a well defined meaning for.

Definition 3.9. We can compile systems to not use channels. We denote by $\text{NoChan}(S)$ the asynchronous 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 external function calls as follows:

Sending, with $m \Rightarrow P$ becomes:

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

Testing, with $n \leftarrow$ **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.

□

This definition makes reference to external functions, so we need to define a package providing these functions. We do so in Game 3.1, via the Channels package.

Channels($\{A_1, \ldots, A_n\}$)

$q[A_i] \leftarrow$ FifoQueue.New()

$\overline{\text{Send}_{A_i}(m)}$:
  $q[A_i]$.Push(m)

$\overline{\text{Test}_{A_i}()}$:
  return $q[A_i]$.Length()

$\overline{\text{Recv}_{A_i}()}$:
  while $q[A_i]$.IsEmpty()
    yield
  $q[A_i]$.Pop()

Game 3.1: Channels

Basically, this game has a queue for each channel, and then provides the functions need to send, receive, and test that channel. We use a FifoQueue which pops messages in the same order that they get pushed in, which models the semantics of a channel delivering messages in order.

One consequence of defining separate functions for each channel is that:

$$\text{Channels}(S) \otimes \text{Channels}(R) = \text{Channels}(S \cup R)$$

which will prove to be a useful property.

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.

At this point, we can also define a notion of efficiency for systems.

Definition 3.10 (Efficient Systems). A system S is said to be *efficient* if $\text{NoChan}(S)$ is an efficient package.

□

Note that we use NoChan rather than SysPack, because this captures the fact that a system only needs to be efficient provided that sending and receiving on channels responds efficiently. Unless otherwise specified, we only consider *efficient* systems from here on.

Our next steps will be defining the basic operations we can use to compose systems, along with some notions of equality we can use to compare systems. The first notion of equality we want to define is the strongest one, *literal equality*, which we'll use to define fundamental properties of our composition operations, like associativity, commutativity, and so on.

First, we need to define a notion of *shape*, like we did for packages, since our various equality relations will require the systems to have the same shape.

Definition 3.11. Given systems A, B, we say that they have the same *shape* if

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

□

This is what you might expect, the functions and channels need to all match for two systems to be considered to have the same shape.

Next, we can define the most basic notion of equality for systems.

Definition 3.12 (Literal System Equality). Given systems A, B with the same shape, we say that they are *literally* equal, written $A \equiv B$ if

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

□

This is a very strong notion of equality, which doesn't take into account the semantics of channels in practice. Basically, it requires that regardless of what

messages the channels might start out with, or even what the semantics of channels are, that the behavior is identical. This is good enough for fundamental properties of our composition operations, but we'll move on to a looser notion for standard equality later, like we did with packages.

3.3 Composing Systems

Now, we move on to define the various ways to compose systems together. Naturally, we can compose systems together like we did for packages by having one system call the functions provided by another, or having two systems used together independently, but we also want to compose systems so that they can communicate with each other using channels.

It's this kind of composition, allowing for communication across channels, that we define first, and call *tensoring*.

Definition 3.13 (System Tensoring). Given two systems, A and B, with $\text{Out}(A) \cap \text{Out}(B) = \emptyset$, we can define their tensoring $A * B$, 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:

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

The intuition for this definition is that tensoring is like $\otimes$ for packages, except that now the systems can interact by exchanging messages. This interaction only happens through the fact that they share a common Channels package, which well then store the messages sent by one system, so that the other can receive them, and vice versa.

We can also gain some confidence in the quality of this definition by proving that it's both associative and commutative.

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 $\cup$.

■

We've also made our lives quite easy, by defining literal equality in terms of NoChan, so we can lean heavily on the work we did in proving that package tensoring is associative and commutative.

In many situations, we'll have systems that don't actually share any channels, and we'll want to compose them as well, while benefiting from some nicer properties.

We define this situation formally.

Definition 3.14 (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.

□

One very common way this situation arises is if a system doesn't use any channels at all. For example, we might write $A \otimes 1(\ldots)$, since $1(\ldots)$ can be considered as a system with no use of channels, and so won't overlap with A. This is why we can see $*$ as the natural generalization of $\otimes$ for systems, because it literally becomes $\otimes$ when used for systems that do not use channels.

Next, we define the analogue of package composition for systems, which allows one system to use the functions provided by the other.

Definition 3.15 (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:

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

- $\text{In}(A \circ B) = \text{In}(B)$.

□

It's very important that the systems do not overlap. Our intention with system composition is that the two systems interact only via the functions that one system provides to the other, and not via any channels. This is like the machine analogy we had earlier, where machines within a row communicate across channels, but are only connected via functions to the rows behind them.

As one might expect, this definition of composition is also associative.

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*.

■

We've now defined tensoring and system composition, and are in the same position as with packages, in that we need some way of characterizing how these operations behave together, so that we can do the various manipulations we need inside proofs.

Thankfully, Lemma 2.6 (interchange) generalizes to systems as well, allowing us to reason in the same way as we can 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:

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

■

This lemma plays the same critical role as it did for packages, and we'll be applying it quite often throughout the rest of this work.

3.4 System Equality and Indistinguishability

Next, we define some looser notions of equality for systems, like those we defined for packages, and then show that the various operations we've defined respect the notions of equality, with one exception.

First, we define the standard notion of equality we'll be using.

Definition 3.16 (System Equality). We say that two systems A, B with the same shape are equal, written $A = B$, if:

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

□

This is the natural definition of equality, since SysPack tries and capture the actual semantics of a system. This, comparing two systems using SysPack allows us to compare the behavior of the two systems, disregarding inessential details, and actually looking at how the use of channels affects their behavior.

If we can compare systems for equality using SysPack, we should also be able to compare them for indistinguishability in the same way.

Definition 3.17 (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:

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

□

This allows for small differences that a bounded adversary can't notice to pop up, and this is the notion of equality that we'll target most often in proofs.

We've seen three notions of equality so far, but we haven't commented that much on how well behaved they are. Thankfully, they all satisfy all the properties we'd expect from an equality relation, including transitivity, which we prove here explicitly.

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 these relations, and the notions for systems are defined in terms of NoChan or SysPack.

■

Next, we need to prove whether or not our various operations respect these notions of equality, like we did for packages. This is very useful, since it allows using the characteristic modular proofs that we have for packages in the context of systems. We can break down a large package into smaller components, and then appeal to the indistinguishability of those small components alone, in order to make an argument about the system as a whole.

The first operation we target is composition.

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.

■

This proof is made relatively simple by being able to appeal to the work we already did in proving the analogous property for packages.

Next, we look at strict tensoring, for systems that do not overlap.

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:

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

which then entails our theorem through similar properties for packages.

Our starting point is:

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

We can write this as:

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

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

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

Which is none other than:

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

concluding our proof.

■

The assumption that the systems do not overlap is in fact essential, because this lemma *does not* hold for tensoring in general.

Here's some intuition for a counter example. The idea is that you can insert a back door into a package by having a channel which is never sent on. The back door is triggered if the package can successfully receive from this channel. If the

use of this back door allows distinguishing two packages, then in isolation they will be equal, since it's not possible to trigger a message being sent to open the back door. However, when composed with another package, that package might be able to unlock the door by sending a message, and thus the composed system can be distinguishable again.

4 Protocols

The goal of this section will be to define *protocols*, along with ways to compose and compare protocols. Intuitively, a protocol is a kind of algorithm involving several players, cooperating together to achieve a desired goal. The protocol specifies how each player should behave.

The first way of composing protocols we look at is concurrent composition, which lets us run two protocols involving separate players in parallel, with on interaction between them. The second way of composing protocols is more interesting. We can have one protocol invoke another as a sub-protocol, with each player in the first playing the role of several players in the latter. These two operations are useful in tandem, allowing us to decompose large protocols into smaller ones, allowing for modular reasoning.

When it comes to the equality of protocols, the preferred notion is that of simulation, which we'll explain in more detail later. For now, the basic idea is that simulation turns attacks on one protocol into attacks on the other. Beyond just simulation, we also define two stronger notions of equality, which allow describing the fact that two protocols behave exactly the same, or almost the same, even without simulation.

This section follows the basic road map we've used for both packages and systems. We first define what protocols are formally, as well as the ways in which they compose. We then define notions of corruption, and then define the semantics of protocols, based on which participants in the protocol are corrupted. Finally, we define notions of equality for protocols, and explore the ways in which these notions are preserved under composition.

4.1 Defining Protocols and Composition

We start by defining protocols. All of the work we expended in defining systems was ultimately to describe protocols, so naturally we'll be using systems again here. The basic idea is that each player is described by a system, and also that the players have access to a package, representing the ideal functionality of the protocol, if any. Rather than considering "real" and "ideal" world protocols, we only ever consider protocols in the hybrid model.

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 $\subseteq \mathrm{Out}(F)$, 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:

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

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

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

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

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

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

Let $\mathrm{Out}_i(\mathscr{P}) := \mathrm{Out}(P_i)$, and let $\mathrm{In}_i(\mathscr{P}) := \mathrm{In}(P_i)/\mathrm{Out}(F)$. We then define:

- $\mathrm{Out}(\mathscr{P}) := \bigcup_{i \in [n]} \mathrm{Out}_i(\mathscr{P})$,
- $\mathrm{In}(\mathscr{P}) := \bigcup_{i \in [n]} \mathrm{In}_i(\mathscr{P})$,
- $\mathrm{IdealIn}_i(\mathscr{P}) := \mathrm{In}(P_i) \cap \mathrm{Out}(F)$,
- $\mathrm{IdealIn}(\mathscr{P}) := \mathrm{In}(F)$.

□

The ideal functionality can be asynchronous, which lets us model things like channels with certain properties inside of the functionality itself. For convenience, we also allow multiple players to use the same functions from the ideal functionality. We explicitly define a leakage set, which will be more important later. For now, we can think of it as part of the ideal functionality that adversaries attacking the protocol will be able to interact with directly. The functions that the protocol can depend on are either provided by other protocols, or other functionalities, which is why we defined In and IdealIn that way.

The condition that the protocol dependencies be distinct will make more sense later, when we define protocol composition, but for now, the basic idea is that if one player uses the functions provided by a player in a sub-protocol, then that means that this players must completely take over the role of the player in the sub-protocol, and we want this relationship to be clear.

We can also define a notion of *efficient* protocols.

Definition 4.2 (Efficient Protocol). A protocol $\mathscr{P}$ is said to be *efficient*, if every player is an efficient system, and its ideal functionality is an efficient package.

□

From now on, we only consider efficient protocols, unless otherwise specified.

Similar to how we often talk about games, rather than just packages, we'll often want to talk about protocols without any dependencies.

Definition 4.3 (Closed Protocol). We say that a protocol $\mathscr{P}$ is *closed* if $\text{In}(\mathscr{P}) = \emptyset$ and $\text{IdealIn}(\mathscr{P}) = \emptyset$.

□

When we eventually get to defining notions of equality and simulation for protocols, these will be targeting *closed* protocols, whose semantics are well defined, since no dependencies are left unfulfilled.

We can, however, define a very strong notion of equality right now.

Definition 4.4 (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 permutation $\pi : [n] \leftrightarrow [n]$ such that $\forall i \in [n].\ \mathscr{P}.P_i \equiv \mathscr{Q}.P_{\pi(i)}$
- $\mathscr{P}.F = \mathscr{Q}.F$
- $\mathscr{P}.\text{Leakage} = \mathscr{Q}.\text{Leakage}$

□

So, the functionalities need to be the same, and the players need to be *literally* equal, up to potential reordering. We use literal equality because we're most likely comparing systems with plenty of open channels, and we want each player to behave the same regardless of what the rest of the protocol is doing.

Like with literal equality for packages and systems, the main purpose of this notion is to talk about the fundamental properties of the composition operations.

Now, we get to our first notion of composition. Protocols can depend on other protocols, but also other functionalities. One natural kind of composition is to fill this demand, by composing a protocol with another functionality.

Definition 4.5 (Vertical Composition). Given a 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 $F \circ G$.

□

This is more useful than it might seem at first. We can use this kind of composition to separate out components of the ideal functionality, which can then allow us to appeal to theorems we've already proved about games to argue about protocols. This kind of composition can be seen as providing a sort of "bridge" between the world of games and the world of protocols.

This composition is also well behaved, satisfying associativity.

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.

■

The next kind of composition we look at allows a protocol to use another as a kind of sub-protocol. The idea is that each player in one protocol plays the role of one or several players in the sub-protocol. The definition is somewhat involved, so we provide some more motivation later.

Definition 4.6 (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.

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

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

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 $\mathscr{P} \triangleleft \mathscr{Q}$.

The players will consist of:

$$\mathscr{P}.P_i \circ \left(\begin{array}{c} \ast \qquad\qquad \mathscr{Q}.P_j \\ {\scriptstyle \mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i} \\ \otimes \\ 1(\mathrm{IdealIn}_i) \end{array} \right)$$

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 $\mathscr{P}.\mathrm{Leakage} \cup \mathscr{Q}.\mathrm{Leakage}$.

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:

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

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

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

as required.

By definition, the dependencies In of each player in $\mathscr{P} \triangleleft \mathscr{Q}$ 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 the disjointedness 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 $\mathscr{P} \triangleleft \mathscr{Q}$, since $\mathscr{P}$ does.

□

The second part of the definition is in fact a proof that the definition produces a valid protocol. The conditions guarantee that the two protocols are isolated from each other, beyond the fact that the players in $\mathscr{P}$ are able to control the players in $\mathscr{Q}$ via the functions they provide. The protocols don't share any channels, or an

ideal functionality. The end result is a protocol in which each player is "emulating" the behavior of the players in the sub-protocol, and where even though the channels are now shared, it's clear whether a message is intended for the main protocol, or for a specific player in the sub-protocol.

Horizontal composition is also well behaved. For example, it satisfies associativity.

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:

$$\mathscr{P}.P_i \circ \left(\begin{array}{c} \ast \\ {\scriptstyle \mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i} \end{array} \mathscr{Q}.P_j \circ \left(\begin{array}{c} \ast \\ {\scriptstyle \mathscr{R}.P_k \text{ used by } \mathscr{Q}.P_j} \end{array} \mathscr{R}.P_k \otimes 1(\mathscr{Q}.\text{IdealIn}_j) \right) \otimes 1(\mathscr{P}.\text{IdealIn}_i) \right)$$

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

$$\left(\mathscr{P}.P_i \circ \left(\begin{array}{c} \ast \\ {\scriptstyle \mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i} \end{array} \mathscr{Q}.P_j \otimes 1(\mathscr{P}.\text{IdealIn}_i) \right) \right) \circ \left(\begin{array}{c} \ast \\ {\scriptstyle \mathscr{R}.P_k \text{ used by } \mathscr{P}.P_i} \end{array} \mathscr{R}.P_k \otimes 1(\mathscr{Q}.\text{IdealIn}_j) \right)$$

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:

$$\mathscr{P}.P_i \circ \left(\begin{array}{c} \ast \\ {\scriptstyle \mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i} \end{array} \mathscr{Q}.P_j \otimes 1(\mathscr{P}.\text{IdealIn}_i) \right) \circ \left(\ast_{\mathscr{Q}.P_j} \left(\begin{array}{c} \ast \\ {\scriptstyle \mathscr{R}.P_k \text{ used by } \mathscr{Q}.P_j} \end{array} \mathscr{R}.P_k \otimes 1(\mathscr{Q}.\text{IdealIn}_j) \right) \right)$$

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

giving us:

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

Which is none other than the players in $\mathscr{P} \triangleleft (\mathscr{Q} \triangleleft \mathscr{R})$.

■

Next, we define another fundamental way to compose protocols: concurrent composition. The idea here is that this allows two protocols to run side by side, without any interaction. The resulting protocol will have two independent sets of players, each running their own protocol together.

Definition 4.7 (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.

□

One detail which might seem odd at first is that we allow for $\mathscr{P}.F = \mathscr{Q}.F$, handling that case a bit separately. This is useful because it allows accommodating a common situation where both protocols have a functionality of the form $1(S)$,

for some set S, and we want to allow them to be composed, to then later write $(\mathscr{P} \otimes \mathscr{Q}) \circ G$, for some package G. Having a shared functionality between concurrent protocols is something we do want to be possible, so handling this edge case is necessary.

This notion of composition is also well behaved, as we now prove.

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 concurrently composition is commutative.

■

The utility of concurrent composition and horizontal composition is enhanced even more when combined together. As an example, consider the common situation where a protocol involves several tasks executed in sequence. One way of writing this would be:

$$\mathscr{O} \lhd \begin{pmatrix} \mathscr{Q}_1 \\ \otimes \\ \mathscr{Q}_2 \end{pmatrix}$$

where $\mathscr{Q}_i$ are sub-protocols for each task, and $\mathscr{O}$ is an orchestration protocol running the tasks in sequence. This decomposition allows a more fine-grained analysis of the protocol's security.

4.2 Corruption

The goal of this section is to formally define the semantics of protocol. We've defined a protocol so far in terms of isolated players, with strong hints as to how the players will interact, but we haven't actually defined how to compose these players together to form an actual system. The idea is that if we know which players are corrupted, and in what way, we can then compile the protocol into a system that an adversary can interact with. They will be able to use the corrupted and honest players to drive the execution of the protocol, in an attempt to distinguish it from other protocols.

An important consideration as we define various kinds of corruption is that if two players are equal, then the way the corrupted players behave should also be

equal. We've encountered this kind of equality preservation before, and it's a property we'll keep an eye out for in this subsection as well.

The first kind of corruption we consider is that of a party which isn't actually corrupted.

Definition 4.8 ("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.

□

This is nonetheless a useful definition to have, since we don't have to treat the honest players as being completely separate from the dishonest players, but rather just corrupted in a different way.

Next, we look at semi-honest corruption. The intuition here is that such a corrupted party will still follow the protocol's execution, but the adversary gains additional visibility into the execution of that player.

Definition 4.9 (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 $\text{log}.\bullet \leftarrow$ 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 $\text{log}.F$ when a function call $F(x_1, \ldots, x_n)$ happens.
- $(\texttt{ret}, y)$ to $\text{log}.F$ when the function F returns a value y.
- $(\texttt{input}, (x_1, \ldots, x_n))$ to $\text{log}.G$ when the function G is called with $(x_i, \ldots)$ as input.
- m to $\text{log}.A$ when a value m is sent on channel A.
- m to $\text{log}.B$ 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.

□

The use of different logs is very useful, since it makes manipulating the log easier, avoiding the need to parse the log to separate out events by function.

One important detail is that the log also contains entries for when the player itself is activated through one of its input functions. This will be useful when reasoning about how protocol composition behaves, because the input events in one players log can become the output function events in the log of a player calling that sub-protocol.

Note that, unlike other definitions of semi-honest corruption, we do not provide access to the randomness sampled by a player, at least not directly. The reason for doing this is ultimately that defining corruption in that way is very difficult to do while preserving equality. There are many equivalent ways to write a given player which result in different sampling patterns. In practice, we don't think this is a strong limitation, because we can also see all of the output functions and channels used by the player, so significant randomness can still be observed.

Now, onto malicious corruption:

Definition 4.10 (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:

$\text{Corrupt}_M(P)$

$\underline{\text{Call}_{F_i}((x_1, \ldots, x_n)):}$
return $F_i(x_1, \ldots, x_n)$

$\underline{\text{Send}_{A_i}(m):}$
$m \Rightarrow A_i$

$\underline{\text{Test}_{B_i}():}$
return test B_i

$\underline{\text{Recv}_{B_i}():}$
return $m \Leftarrow B_i$

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 $P' \equiv P$.

□

The intuitive idea behind malicious corruption is that this party can deviate arbitrarily from the protocol. The adversary corrupting this party can call any function this party is allowed to call, and use any channel this party uses.

An interesting property of the kinds of corruption is that each form of corruption is stronger than the other. A semi-honest party provides more information than an honest party, and a malicious party doesn't even need to follow the protocol anymore. We can capture this hierarchy formally.

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:

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

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.

■

We'll be using this lemma later, where it will help us show that in some situations, it suffices to consider only malicious corruption, which can simplify many proofs.

Next, we'll be defining what it means to actually execute a protocol with some players being corrupted. The first notion we'll need to develop is that of a *cor-*

ruption model, which is just a way of specifying which players in a protocol are corrupted, and how.

Definition 4.11 (Corruption Models). Given a protocol $\mathscr{P}$ with players $P_1, \ldots, P_n$, a *corruption model* C is a function $C : [\mathscr{P}.n] \to \{\mathrm{H}, \mathrm{SH}, \mathrm{M}\}$. This provides a corruption C_i associated with each player P_i. We also define a little syntax to talk about corruptions in general, writing $\mathrm{Corrupt}_\kappa(P)$, for $\kappa \in \{\mathrm{H}, \mathrm{SH}, \mathrm{M}\}$, which we can then use to define $\mathrm{Corrupt}_C(P_i) := \mathrm{Corrupt}_{C_i}(P_i)$.

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

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

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.

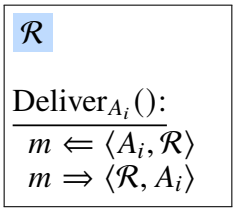
The notion of class is very useful, and is what we usually end up proving things about. For example, we prove that two protocols are the same under a given class of corruptions. That proof will involve looking at each model inside the class, as we'll see later.

Now, let's define what the semantics of a protocol are under a given corruption model. These semantics should define how an adversary can run and interact with a protocol, having corrupted some of the parties.

Definition 4.12 (Instantiation). Given a protocol $\mathscr{P}$ with $\mathrm{In}(\mathscr{P}) = \emptyset$, and a corruption model C, we can define an *instantiation* $\mathrm{Inst}_C(\mathscr{P})$, 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 $\{A_1, \ldots, A_n\} = \mathrm{Chan}(P_1, \ldots, P_n)$. We then define $\mathcal{R}$ as the system:



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

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

With this in hand, we define:

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

□

The basic idea is that the adversary can call the functions provided by any player, along with the leakage exposed by the ideal functionality. This provides a very asynchronous notion of execution, where the adversary is able to run different parts of the protocol at will. There isn't even a clear "entry point". Each player might have multiple functions that are provided, and the adversary is able to start with whichever one they want.

The use of the router allows the adversary to control the flow of messages in the protocol, deciding when a message should be delivered. In this model, the adversary can't reorder messages, but one can model protocols in which this happens either via a functionality, or by having each message delivered on a separate channel.

Our next steps will be exploring how instantiation behaves for composed protocols, so that we can extract some insights that we can use when proving properties of the various notions of equality we define later.

We start this exploration by looking at a few convenient properties of our routing transformation.

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

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

(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.

■

This will be very useful for our proofs soon enough.

Eventually, we'll want to prove things like "if $\mathcal{Q}$ behaves the same as $\mathcal{Q}'$, then $\mathcal{P} \triangleleft \mathcal{Q}$ behaves the same as $\mathcal{P} \triangleleft \mathcal{Q}'$". Whenever we talk about behavior, we need to define a corruption model, so that we can actually instantiate the protocol. There is a slight issue here, in that a corruption model is specific to one protocol. A corruption model might say how $\mathcal{Q}$'s players should be corrupted, but how does this then apply to $\mathcal{P} \triangleleft \mathcal{Q}$? Thankfully, for the ways of composing protocols we've defined, there are natural notions of corruption which make sense in such a situation. If $\mathcal{Q}$ is corrupted in a certain way, then this implies that certain corruptions need to happen in $\mathcal{P} \triangleleft \mathcal{Q}$ as well.

We define this more formally, through the notion of *compatible* corruptions.

Definition 4.13 (Compatible Corruptions). Given protocols $\mathcal{P}, \mathcal{Q}$, and a corruption model C for $\mathcal{Q}$, we can define a notion of a *compatible* corruption model C' for $\mathcal{P} \otimes \mathcal{Q}$ or $\mathcal{P} \triangleleft \mathcal{Q}$, provided these expressions are well defined.

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

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

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 $\mathcal{C}'$ is (strictly) compatible with a class $\mathcal{C}$, if every $C' \in \mathcal{C}'$ is (strictly) compatible with some $C \in \mathcal{C}$.

□

For tensoring, compatibility is quite simple, we just need the players that belong to $\mathcal{P}$'s "side" of the protocol to be corrupted in the same way, or worse. For composition, the idea is that for a player in a sub-protocol to be corrupted, then the player using it in the main protocol needs to be at least as corrupt. For technical reasons, we'll also be needing the notion of *strict* compatibility. This avoids situations where a player P uses two players Q_1, and Q_2, but only Q_2 is malicious. In that case, we'd require P to be malicious for compatibility, but if Q_1 is honest, the corruption of P might be too strong now for certain properties to hold. If this doesn't quite make sense now, hopefully it will be clearer when reading the proofs that make use of this strict property.

As a first use of this notion of compatibility, we make a fundamental observation

about instantiating the concurrent composition of protocols. This is elevated to a theorem, because this breakdown observation will be used as the crux of all of our equality-related theorems about concurrent composition.

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:

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

■

This is an extremely useful theorem, since it breaks down the instantiation of the tensor product into another tensor product of systems. This observation is the

cornerstone of proving the properties that concurrent composition satisfies with respect to equality and simulation.

Now, we tackle horizontal composition. Unfortunately, the statement we have here is not quite as elegant as that of concurrent composition.

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 $\mathscr{P} \triangleleft \mathscr{Q}$, there exists systems $S_1, \ldots, S_{\mathscr{Q}.n}$ and a set $L_{\mathscr{Q}}$ such that:

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

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

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

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

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

Our strategy will be to progressively build up an equivalent system to this one, starting with $\mathrm{Corrupt}_C$, then Routed, etc.

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

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 $O = \text{Out}(\text{Corrupt}_M(P \circ (Q_1 * \cdots)))$. 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 $\text{Corrupt}'_M$, 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 $\text{Call}_\bullet$ 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 $L^M = \text{Out}(\text{Corrupt}_M(Q_1)) \cup \cdots$.

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

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

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

And for honest corruption, we have

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

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 $C'_i = \text{M}$:

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

with $O_i = \text{Out}(\text{Corrupt}_{C'}(\mathscr{P} \triangleleft \mathscr{Q}).P_i)$, $L_i = \bigcup_{\mathscr{Q}.P_j \text{ used by } \mathscr{P}.P_i} \text{Out}(\text{Corrupt}_M(\mathscr{Q}.P_j))$. 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 $C'_i = \text{SH}$:

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

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

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

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

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

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

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

We can unify these three cases, writing:

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

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:

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

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:

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

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

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

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

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

with $O := \text{Out}(\text{Inst}_{C'}(\mathscr{P} \triangleleft \mathscr{Q}))$, 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:

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

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

■

If you squint at this theorem, it's basically saying that

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

while also allowing the inputs to $\mathscr{P}.F$ to flow in. This is the core observation we need for later. If we look at the decomposition more closely, the front is almost like the $\text{Inst}(\mathscr{P})$, except that more information needs to flow through, since the adversary also gets leakage information and routing control from $\mathscr{Q}$. Furthermore, the interaction with $\mathscr{Q}$ is mediated via the S_i, which exist because compatibility only requires that the corruption in $\mathscr{Q}$ is at least as strong, so these S_i are there to weaken what the players in $\mathscr{P}$ have access to.

4.3 Equality and Simulation

In this subsection, we finally get to the various notions of equality and simulation we've been foreshadowing. To skip ahead a bit, we define three main notions here, which are about claims that:

1. two protocols behave identically,
2. two protocols behave indistinguishably,
3. one protocol is simulated by another.

After defining these notions, we also show that all the kinds of protocol composition we've defined respect these notions, and satisfy a form of transitivity. This allows us to make such claims about a large protocol, by decomposing it into smaller protocols, and then making several hops, appealing to claims about these smaller protocols. This is analogous to the strategy that the state-separable proof paradigm takes to proving things about game. Indeed, this analogy is the principal motivation for this framework.

Like with packages and systems, to even compare two protocols, they need to have the same shape.

Definition 4.14 (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:

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

□

The reason we use the $\mathscr{Q}'$ equivalent to $\mathscr{Q}$ is just so that we can get the order of players to be the same as in $\mathscr{P}$.

The first notion of equality we capture is about arguing that two protocols have the same behavior under a class of corruptions.

Definition 4.15 (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:

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

as systems, with $\mathscr{Q}' \equiv \mathscr{Q}$ as per Definition 4.14.

□

For closed protocols, this is a more natural notion of equality than $\equiv$, since it allows for behaviors that are effectively identical, while not technically the same.

This notion is too strict for many protocols, which make use of hard problems. In this case, we want to appeal to indistinguishability instead.

Definition 4.16 (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:

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

as systems, with $\mathscr{Q}' \equiv \mathscr{Q}$ as per Definition 4.14.

□

Like with systems and packages, this notion allows for small differences, and restricts the adversary to only have a limited amount of computation, allowing for hard problems to exist, and be used inside the protocol.

The notions we've seen so far are natural extensions of the ones we've defined for packages and systems. This next notion, on the other hand, is novel. This is the notion of *simulation*, and is the typical kind of security claim made about protocols. Simulation allows for many more protocols to be compared, because it allows for a simulator S to interface between the adversary and one of the protocols. The intuition here is that the simulator translates attacks made on one protocol to attacks made on another. If a protocol is simulated by a protocol under which no attack is possible, then we can conclude that no attack is possible against the concrete protocol, since that would immediately translate into an attack against the secure one.

To get to this notion of simulation, we first need to formally define what a simulator is, and what it means to instantiate a protocol with that simulator.

Definition 4.17 (Simulated Instantiation). A simulator S for a closed protocol $\mathscr{P}$ under a corruption model C is a system satisfying:

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

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

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

□

Basically, the simulator S is allowed to touch all of the "adversarial" parts of the instantiation. This is basically everything except the honest parts of the protocol. This includes the input functions for semi-honest parties, but not their logs. We can think of this simulator as translating attacks, as mentioned above. We can also think of the simulator as trying to "trick" the adversary into thinking it's interacting with one protocol, whereas in fact it's interacting with another.

This leads to a kind of notion of equality, called *simulation*.

Definition 4.18 (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:

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

as systems, with $\mathscr{Q}' \equiv \mathscr{Q}$ as per Definition 4.14.

□

Note that this is not a symmetric notion. There's a clear directionality to claims of simulation, as indicated by the choice of notation. One important technical detail is that the simulator can depend on the specific corruption model. In many cases, we even provide an explicit case-by-case proof, using different simulator strategies for each kind of corruption.

As one might expect, these notions of equality and simulation form a nice hierarchy, which we can formalize as follows.

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 $C \in \mathscr{C}$, $\mathrm{Corrupt}_C$ and Routed are equality respecting, so we have:

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

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 $\mathrm{Inst}_C(\mathscr{P}) = \mathrm{Inst}_C(\mathscr{Q})$, 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.

■

The fact that equality implies indistinguishability is unsurprising, since we've seen that hold already for packages and systems. For simulation, the key to the proof was that you can define a simulator that doesn't do anything, in which case indistinguishability and simulation are effectively the same.

For these equality notions to be useful, we also want some kind of transitivity, so that we can decompose proofs into smaller hops. Thankfully, we also have analogous notions of transitivity.

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:

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

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

■

The crux of the proof was that simulators compose together, which allows simulation to become transitive. As we've seen for packages and systems, transitivity is a critical part of what makes proofs more modular.

From the definitions we've seen so far, it's necessary to consider both semi-honest and malicious corruption, if the class happens to include them. It turns out that for $=$ and $\approx$, we can always disregard semi-honest corruption in favor of malicious corruption, and in some cases we can also do this for simulation as well.

Theorem 4.10 (Malicious Completeness). Let $\mathcal{P}$ and $\mathcal{Q}$ 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. $\mathcal{P} =_{\mathscr{C}} \mathcal{Q} \implies \mathcal{P} =_{\mathscr{C}'} \mathcal{Q}$,
2. $\mathcal{P} \overset{\epsilon}{\approx}_{\mathscr{C}} \mathcal{Q} \implies \mathcal{P} \overset{\epsilon}{\approx}_{\mathscr{C}'} \mathcal{Q}$,

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

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

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

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

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:

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

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:

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

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

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

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

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

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

■

For simulation, unfortunately we get stuck in the proof without the additional assumption. The fundamental issue is that malicious corruption helps both the adversary and the simulator. The simulator might make use of the extra powers they get from malicious corruption, which are then no longer available to them in the semi-honest case. One particular power is the ability to change the input being provided, as noted in [HL10], in which this conundrum is explored further, providing some example protocols secure under malicious, but not semi-honest corruption. The condition we've added might seem a bit odd—being able to simulate semi-honest corruption from malicious corruption—but it does actually show up somewhat often. For example, if a protocol just consists of calling part of an ideal functionality, then semi-honest and malicious corruption are the same, as we'll see later.

The next step will be to show how the various notions of composition we've defined interact with these notions of equality and simulation. Thankfully, all of the ways of composing protocols respect both equality and simulation in the natural ways, allowing the use of modular proofs like the ones we can create for packages.

First, we look at composing protocols with functionalities.

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 3.6, which entails that for any system S, we have $S \circ G \overset{\epsilon}{\approx} S \circ G'$.

■

This property is quite useful, since it allows separating out part of a functionality, and then appealing to the indistinguishability of two games, to argue that one protocol simulates another. This allows a kind of bridging between game-based proofs and protocols, allowing us to make use of indistinguishability proofs for games to aid in proving properties of protocols.

Next, we look at composing protocols concurrently. We'll need to use the notion of compatibility for corruption classes that we've defined before.

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.

■

Critically, the use of the concurrent breakdown theorem was essential in proving this theorem. Basically, all the hard work had already been done, and we just need to apply some of the notions we've developed for systems to finish the details of the proof.

Finally, we can look horizontal composition of protocols. Like with the breakdown theorems, this one is a tad more complicated, and is where we need to deploy the notion of *strict* compatibility we developed earlier.

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

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

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

3. $\mathcal{Q} \overset{\epsilon}{\rightsquigarrow}_{\mathcal{C}} \mathcal{Q}' \implies \mathcal{P} \triangleleft \mathcal{Q} \overset{\epsilon}{\rightsquigarrow}_{\mathcal{C}'} \mathcal{P} \triangleleft \mathcal{Q}'$

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

$$\text{Inst}_{C'}(\mathcal{P} \triangleleft \mathcal{Q}) = \text{Stuff} \circ \text{Inst}_{C}(\mathcal{Q})$$

This summary suffices to prove a couple statements already.

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

$$\text{Inst}_{C'}(\mathcal{P} \triangleleft \mathcal{Q}) = \text{Stuff} \circ \text{Inst}_{C}(\mathcal{Q})$$

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

$$\text{Stuff} \circ \text{Inst}_{C}(\mathcal{Q}) = \text{Stuff} \circ \text{Inst}_{C}(\mathcal{Q}')$$

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

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

$$\text{Stuff} \circ \text{Inst}_{C}(\mathcal{Q}) \overset{\epsilon}{\approx} \text{Stuff} \circ \text{Inst}_{C}(\mathcal{Q}')$$

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 $C' \in \mathcal{C}'$, there exists a *strictly* compatible $C \in \mathcal{C}$, and we have:

$$\text{Inst}_{C'}(\mathcal{P} \triangleleft \mathcal{Q}) = \text{Stuff} \circ \text{Inst}_{C}(\mathcal{Q})$$

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

$$\text{Stuff} \circ \text{Inst}_{C}(\mathcal{Q}) \overset{\epsilon}{\approx} \text{Stuff} \circ (S \otimes 1(\ldots)) \circ \text{Inst}_{C}(\mathcal{Q})$$

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

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

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

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

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

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

with $O_S := O_{\bar{S}} \cup \mathrm{Out}(\mathscr{P}.F)$ 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 $\mathrm{Corrupt}'_{C'}(\mathscr{P}.P_i)$ 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:

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

which becomes:

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

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

■

The reason we needed strict compatibility was that we needed to move the simulator S for $\mathscr{Q}$, to instead become a simulator for $\mathscr{P}$. When we have strictly compatible corruption, there are no barriers to doing this, since S is able to get all the information it needs about $\mathscr{Q}$ via $\mathscr{P} \triangleleft \mathscr{Q}$. However, if we don't have strict compatibility, we might run into the issue that S requires a stronger kind of corruption than $\mathscr{P}$ ends up using, and so we wont be able to move S to the left hand side, as we did here. This is why we demand strict compatibility. In practice, this condition shouldn't be very demanding, because in many cases the number of parties is the same for both protocols, or we're focusing on a complete corruption class, like "up to $n - 1$ corruptions".

At this point, we've covered the crux of our modular framework for protocols. We've defined the common notion of simulation, which is usually the kind of statement we want to prove. Furthermore, we've shown that the various means of composing protocols respect this simulation. So, if one small part of a protocol is simulated by another, then we can argue that a larger protocol making use of the component will be simulated by replacing this component. This allows reasoning about large protocols via small components, and makes composing isolated protocols into larger systems in a secure way much easier. Furthermore, the framework we've defined so far also integrates nicely with games, since ideal functionalities are simply packages. We can even use notions of indistinguishability for these functionalities to argue that the protocols that use them simulate one another.

4.4 Global Functionalities

Next, we redo a bit of what we've developed so far, this time with the goal of incorporating *global functionalities*. The basic example one should have in mind throughout this section is that of a hash function, treated as a *global random oracle*. This hash can be used by various protocols, but yet it should be treated as one common random oracle throughout all of the protocols. We need a notion of simulation which can account for this example. Basically, we want to say that one protocol simulates another, even in the presence of a shared random oracle—or some other global functionality—and this notion of simulation should have the nice composability properties that we've come to expect.

This development does involve rehashing some of the work we've done in the previous subsection. We could have avoided some of the repetition here, but we feel like having a separate subsection provides more clarity, especially since in many cases a global functionality isn't being used.

First, when a protocol depends on a global functionality, this is because it isn't closed. This dependency will be from the fact that its ideal functionality still has some dependencies on the global functionality.

We can formalize this "closed but for the global functionality" notion.

Definition 4.19 (Relatively Closed Protocols). A protocol $\mathscr{P}$ is *closed relative to* a game G if:

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

□

As one might expect, we now define notions of instantiation and equality for such relatively closed protocols.

Definition 4.20 (Relative Instantiation). Given a closed protocol $\mathscr{P}$ relative to G, we can define, for any corruption model C, the relative instantiation:

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

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

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

□

One key aspect of relative instantiation is that the adversary is always able to interact with G completely. Going back to our example of the hash function, this would also be the case. The adversary is able to call the global random oracle at will. This complete access is key to composability.

We can now use relative instantiation to define the same notions of equality that we did for standard protocols and regular instantiation.

Definition 4.21 (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:

- $\mathscr{P} =_{\mathscr{C}}^G \mathscr{Q} \iff \forall C \in \mathscr{C}.\ \text{Inst}_C^G(\mathscr{P}) = \text{Inst}_C^G(\mathscr{Q})$
- $\mathscr{P} \overset{\epsilon}{\approx}_{\mathscr{C}}^G \mathscr{Q} \iff \forall C \in \mathscr{C}.\ \text{Inst}_C^G(\mathscr{P}) \overset{\epsilon}{\approx} \text{Inst}_C^G(\mathscr{Q})$
- $\mathscr{P} \overset{\epsilon}{\rightsquigarrow}_{\mathscr{C}}^G \mathscr{Q} \iff \forall C \in \mathscr{C}.\ \exists S.\ \text{Inst}_C^G(\mathscr{P}) \overset{\epsilon}{\approx} \text{SimInst}_{S,C}^G(\mathscr{Q})$

□

These are all the same, except for the replacement of instantiation with relative instantiation with respect to G. As one might expect, the same kind of equality hierarchy also holds here as well.

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.

■

Furthermore, these notions of equality are transitive in the exact same way as before.

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:

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

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

■

In many cases, we want to avoid proving semi-honest security explicitly, if we can get away with just proving malicious security. Thankfully, the same observation we made for standalone protocols also applies to ones that use a global functionality.

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:

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

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

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

which can then be rearranged with interchange to get:

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

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.

■

As we've foreshadowed, our next task will be to prove that the various notions of composition we have respect the relative notions of equality we've developed. Thankfully, our restriction that the adversary can access the entirety of the global functionality makes these theorems easy to prove. Since we've already seen these theorems before, in the standalone context, we won't provide much commentary. The proofs are usually based on the proofs in the previous subsection as well.

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:

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

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:

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

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:

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

Then, we unroll $\mathrm{SimInst}^{G}_{S,C}(\mathscr{Q})$, to get:

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

Then, we apply interchange to get:

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

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 $\mathscr{P} \triangleleft \mathscr{Q}$ well defined, and for any compatible corruption classes $\mathscr{C}, \mathscr{C}$, we have:

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

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

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

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

A crude summary of the horizontal breakdown theorem is that:

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

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

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

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:

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

for some appropriate simulator S'.

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

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

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:

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

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

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

which is none other than $\text{SimInst}^{G}_{S',C'}(\mathcal{P} \triangleleft \mathcal{Q})$.

■

So, the three big theorems we proved in the standalone context also hold in the global context. We could have saved some repetition by just doing everything in the global context, but since we expect most proofs to be done standalone, we felt that it was clearer to present that in more detail, and then present the global generalizations more rapidly.

4.5 Hopping Ideal Functionalities

One difference between the framework we've developed and other frameworks is that we always make statements about protocols. Protocols are equal to each other, or simulate one another, etc. In UC security, statements are usually between protocols and ideal functionalities. One says that a protocol is simulated by a *functionality*, and not another protocol.

Sometimes we want to be able to make this kind of statement as well, and the following lemma can help us with that.

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

$$\mathcal{P} \equiv \mathcal{P}' \triangleleft \mathcal{G}$$

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

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

■

The idea is that if we're using some ideal functionality inside of a protocol, we can actually write this as using a *sub-protocol*, where that sub-protocol is the one using the ideal functionality. This sub-protocol will be a kind of "dummy protocol", which just immediately forwards inputs to the functionality. This allows us to capture the kind of "a protocol is simulated by functionality" statement that one might want to make. We would prove that a protocol simulates the *dummy protocol* associated with a given functionality. Then, whenever that functionality is used inside of a protocol, we can appeal to the deidealization lemma to argue that we can replace the functionality with the concrete protocol.

An example might help. Let's say we have a functionality G. It's dummy protocol is, say, $\mathcal{G}$. We might succeed in proving that $\mathcal{Q} \rightsquigarrow \mathcal{G}$—ignoring corruption classes and the epsilon—for a concrete protocol $\mathcal{Q}$. Then, if we have a protocol $\mathcal{P} \circ G$, by deidealization, we can write this as: $\mathcal{P}' \triangleleft \mathcal{G}$, and then we use the fact that composition respects simulation to conclude that

$$\mathcal{P}' \triangleleft \mathcal{Q} \rightsquigarrow \mathcal{P}' \triangleleft \mathcal{G} = \mathcal{P} \circ G$$

allowing us to replace a functionality with a concrete protocol.

Another similar idea allows us to turn global instantiation into normal instantiation, by just embedding in the global functionality.

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

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

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

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

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

■

The utility here is that we can prove a bunch of things in the global model, and then choose to actually instantiate the functionality with a local version of it at some concrete point. We might even then appeal to deidealization to replace the functionality with a real protocol. For example, some protocols proven secure in the global random oracle model could be composed together, and then the global random oracle could be replaced by a local one, shared by all the protocols, and then we could even replace this local random oracle with a protocol to actually sample randomness.

5 Examples

In this section, we provide a couple example proofs in the framework, to illustrate how it works, and some of the advantages it provides. The two examples we provide are that of constructing a private channel from one that leaks all messages sent on it, and that of sampling an unbiased random value using the ubiquitous paradigm of "commit reveal".

5.1 Constructing Private Channels

In this subsection, 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. This example was also used in [CD+15].

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[PubChan]

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

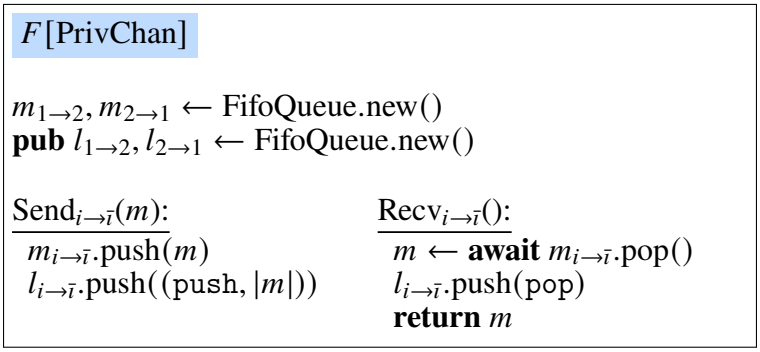
$\underline{\text{Send}_{i\to\bar{\imath}}(m)}$:	$\underline{\text{Recv}_{i\to\bar{\imath}}()}$:
$m_{i\to\bar{\imath}}$.push(m)	**return await** $m_{i\to\bar{\imath}}$.pop()

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.



Game 5.2: Private Channel Functionality

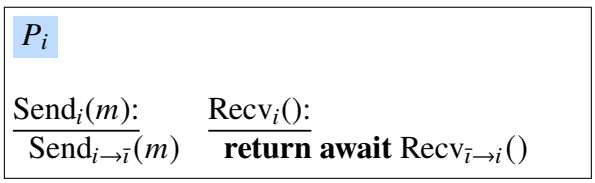
The crucial difference is the nature of the leakage. Rather than being able to see the current state of either message queue, including the messages themselves, now the adversary can only see a historical log for each queue, describing only the *length* of the messages inserted into the queue. The reason 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 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 $\mathcal{Q}$, and defined via Protocol 5.3

$\mathcal{Q}$ is characterized by:

- Leakage := $\{l_{1\to2}, l_{2\to1}\}$,
- F := PrivChan,
- And two players defined via the following system (for $i \in \{1, 2\}$):



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 Appendix A.1. 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.

Keys

$(\mathrm{sk}_1, \mathrm{pk}_1) \xleftarrow{\$} \mathrm{Gen}()$
$(\mathrm{sk}_2, \mathrm{pk}_2) \xleftarrow{\$} \mathrm{Gen}()$

$\underline{\mathrm{Keys}_i()}$:
return $(\mathrm{sk}_i, \mathrm{pk}_{\bar{i}})$

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

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

P_i

$(\text{sk}_i, \text{pk}_{\bar{i}}) \leftarrow \text{Keys}_i()$

$\underline{\text{Send}_i(m)\text{:}}$	$\underline{\text{Recv}_i()\text{:}}$
$\text{Send}_{i\to\bar{i}}(\text{Enc}(\text{pk}_{\bar{i}}, m))$	$c \leftarrow$ **await** $\text{Recv}_{\bar{i}\to i}()$
	return $\text{Dec}(\text{sk}_i, c)$

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:

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

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 $\mathrm{Inst}_{\mathrm{H}}(\mathscr{P})$, obtaining:

$\mathrm{Inst}_{\mathrm{H}}(\mathscr{P}) =$

Γ^0

view $c_{1\to 2}, c_{2\to 1} \leftarrow \mathrm{FifoQueue.new}()$
$(\mathrm{sk}_i, \mathrm{pk}_{\bar{i}}) \leftarrow \mathrm{Keys}_i()$

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

$\mathrm{Send}_i(m)$:
  $c \leftarrow \mathrm{Enc}(\mathrm{pk}_{\bar{i}}, m)$
  $c_{i\to\bar{i}}.\mathrm{push}(c)$

$\mathrm{Recv}_i()$:
  $c \leftarrow$ **await** $\mathrm{c}_{\bar{i}\to i}.\mathrm{pop}()$
  return $\mathrm{Dec}(\mathrm{sk}_i, c)$

$\circ$ 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 $m_{i\to\bar{i}}$ to $c_{i\to\bar{i}}$, 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 the other. Applying this gives us:

$\Gamma^0 \circ \mathrm{Keys} =$

Γ^1

view $c_{1\to 2}, c_{2\to 1} \leftarrow \mathrm{FifoQueue.new}()$
$m_{1\to 2}, m_{2\to 1} \leftarrow \mathrm{FifoQueue.new}()$
$(\bullet, \mathrm{pk}_{\bar{i}}) \leftarrow \mathrm{Keys}_i()$

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

$\mathrm{Send}_i(m)$:
  $c \leftarrow \mathrm{Enc}(\mathrm{pk}_{\bar{i}}, m)$
  $c_{i\to\bar{i}}.\mathrm{push}(c)$
  $m_{i\to\bar{i}}.\mathrm{push}(m)$

$\mathrm{Recv}_i()$:
  $c \leftarrow$ **await** $\mathrm{c}_{\bar{i}\to i}.\mathrm{pop}()$
  $m \leftarrow$ **await** $\mathrm{m}_{\bar{i}\to i}.\mathrm{pop}()$
  return m

$\circ$ 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:

$$\Gamma^1 \circ \text{Keys} = \boxed{\begin{array}{l}\Gamma^2 \\ \\ \textbf{view } c_{1\to2}, c_{2\to1} \leftarrow \text{FifoQueue.new}() \\ m_{1\to2}, m_{2\to1} \leftarrow \text{FifoQueue.new}() \\ \\ \underline{\text{PKs}():} \\ \quad \textbf{return } (\textbf{super}.\text{pk}_1, \textbf{super}.\text{pk}_2) \\ \underline{\text{Send}_i(m):} \qquad\qquad \underline{\text{Recv}_i():} \\ \quad c \leftarrow \text{Challenge}_{\bar{i}}(m) \qquad c \leftarrow \textbf{await } \text{c}_{\bar{i}\to i}.\text{pop}() \\ \quad c_{i\to\bar{i}}.\text{push}(c) \qquad m \leftarrow \textbf{await } \text{m}_{\bar{i}\to i}.\text{pop}() \\ \quad m_{i\to\bar{i}}.\text{push}(m) \qquad \textbf{return } m\end{array}} \circ \begin{pmatrix}\text{IND}_0 \\ \otimes \\ \text{IND}_0\end{pmatrix}$$

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:

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

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

If we unroll this last game, we get:

$$\Gamma^1 \circ \begin{pmatrix}\text{IND}_1 \\ \otimes \\ \text{IND}_1\end{pmatrix} = \boxed{\begin{array}{l}\Gamma^3 \\ \\ \textbf{view } c_{1\to2}, c_{2\to1} \leftarrow \text{FifoQueue.new}() \\ m_{1\to2}, m_{2\to1} \leftarrow \text{FifoQueue.new}() \\ (\text{sk}_i, \text{pk}_i) \overset{\$}{\leftarrow} \text{Gen}() \\ \\ \underline{\text{PKs}():} \\ \quad \textbf{return } (\text{pk}_1, \text{pk}_2) \\ \underline{\text{Send}_i(m):} \qquad\qquad \underline{\text{Recv}_i():} \\ \quad r \overset{\$}{\leftarrow} \mathbf{M}(|m|) \qquad c \leftarrow \textbf{await } \text{c}_{\bar{i}\to i}.\text{pop}() \\ \quad c_{i\to\bar{i}}.\text{push}(\text{Enc}(\text{pk}_{\bar{i}}, r)) \qquad m \leftarrow \textbf{await } \text{m}_{\bar{i}\to i}.\text{pop}() \\ \quad m_{i\to\bar{i}}.\text{push}(m) \qquad \textbf{return } m\end{array}}$$

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:

$\Gamma^4 =$

Γ^5

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

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

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

$c_{i\to\bar{\imath}}()$:
  while $\text{cmd} \leftarrow \text{l}_{i\to\bar{\imath}}.\text{pop}() \neq \bot$:
    if cmd = 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 $c_{i\to\bar{\imath}}$

$\text{Recv}_i()$:
  $m \leftarrow$ **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:

$$\Gamma^5 = \left[\begin{array}{l} S \\ \\ \textbf{view } c_{1\to 2}, c_{2\to 1} \leftarrow \text{FifoQueue.new}() \\ (\text{sk}_i, \text{pk}_i) \xleftarrow{\$} \text{Gen}() \\ \\ \underline{\text{PKs}():} \\ \quad \textbf{return } (\text{pk}_1, \text{pk}_2) \end{array} \quad \begin{array}{l} \underline{c_{i\to\bar{\imath}}():} \\ \quad \textbf{while } \text{cmd} \leftarrow \text{l}_{i\to\bar{\imath}}.\text{pop}() \neq \bot: \\ \quad\quad \textbf{if } \text{cmd} = \texttt{pop}: \\ \quad\quad\quad c_{i\to\bar{\imath}}.\text{pop}() \\ \quad\quad \textbf{if } \text{cmd} = (\texttt{push}, |m|): \\ \quad\quad\quad r \xleftarrow{\$} \mathbf{M}(|m|) \\ \quad\quad\quad c_{i\to\bar{\imath}}.\text{push}(\text{Enc}(\text{pk}_{\bar{\imath}}, r)) \end{array} \right] \circ \text{Inst}_{\text{H}}(\mathcal{Q}) \otimes 1(\text{Send}_i, \text{Recv}_i)$$

which concludes this part of our proof, having written out our simulator, and proven that $\text{Inst}_{\text{H}}(\mathcal{P}) \overset{\epsilon}{\approx} \text{SimInst}_{S,\text{H}}(\mathcal{Q})$.

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 $\mathcal{P} \overset{0}{\rightsquigarrow}_{\{\text{M}\}} \mathcal{Q}$, which naturally implies the slightly higher upper bound of $2 \cdot \text{IND}$.

We start by unrolling $\mathrm{Inst}_M(\mathscr{P})$, to get:

$\mathrm{Inst}_M(\mathscr{P}) =$

Γ^1

view $c_{1\to 2}, c_{2\to 1} \leftarrow \mathrm{FifoQueue.new}()$
$(\mathrm{sk}_2, \mathrm{pk}_1) \leftarrow \mathrm{Keys}_2()$

PKs():	$\mathrm{Keys}_1()$:
return $(\mathrm{pk}_1, \mathrm{pk}_2)$	**return super**.$\mathrm{Keys}_1()$
$\mathrm{Send}_1(c)$:	$\mathrm{Recv}_1()$:
$c_{1\to 2}.\mathrm{push}(c)$	**return await** $c_{2\to 1}.\mathrm{pop}()$
$\mathrm{Send}_2(m)$:	$\mathrm{Recv}_2(m)$:
$c \leftarrow \mathrm{Enc}(\mathrm{pk}_1, m)$	$c \leftarrow$ **await** $c_{1\to 2}.\mathrm{pop}()$
$c_{2\to 1}.\mathrm{push}(c)$	**return** $\mathrm{Dec}(\mathrm{sk}_2, c)$

$\circ$ 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:

$\Gamma^1 \circ \text{Keys} =$

Γ^2

view $c_{1\to2}, c_{2\to1} \leftarrow \text{FifoQueue.new}()$
$(\text{sk}_1, \text{pk}_1),\ (\text{sk}_2, \text{pk}_2),\ (\text{sk}'_2, \text{pk}'_2) \leftarrow \text{Gen}()$

PKs():	$\text{Keys}_1()$:
return $(\text{pk}_1, \text{pk}_2)$	**return** $(\text{sk}_1, \text{pk}_2)$
$\text{Send}_1(c)$:	$\text{Recv}_1()$:
$c_{1\to2}.\text{push}(c)$	**return await** $c_{2\to1}.\text{pop}()$
$\text{Send}_2(m)$:	$\text{Recv}_2(m)$:
$c \leftarrow \text{Enc}(\text{pk}_1, m)$	$c \leftarrow$ **await** $c_{1\to2}.\text{pop}()$
$c_{2\to1}.\text{push}(c)$	$m \leftarrow \text{Dec}(\text{sk}_2, c)$
	$c' \leftarrow \text{Enc}(\text{pk}'_2, m)$
	$m \leftarrow \text{Dec}(\text{sk}'_2, c')$
	return m

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 $m_{1\to2}$ which will be used to send messages directly.

This gives us:

$\Gamma^2 =$

Γ^3

$m_{1\to 2}$, **view** $c_{1\to 2}$, **view** $c_{2\to 1} \leftarrow$ FifoQueue.new()
$(\mathrm{sk}_1, \mathrm{pk}_1),\ (\mathrm{sk}_2, \mathrm{pk}_2),\ (\mathrm{sk}'_2, \mathrm{pk}'_2) \leftarrow \mathrm{Gen}()$

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

$\mathrm{Keys}_1()$:
return $(\mathrm{sk}_1, \mathrm{pk}'_2)$

$\mathrm{Send}_1(c)$:
$c_{1\to 2}.\mathrm{push}(c)$
$m \leftarrow \mathrm{Dec}(\mathrm{sk}'_2, c)$
$m_{1\to 2}.\mathrm{push}(m)$

$\mathrm{Recv}_1()$:
return await $c_{2\to 1}.\mathrm{pop}()$

$\mathrm{Recv}_2(m)$:
$c \leftarrow$ **await** $c_{1\to 2}.\mathrm{pop}()$
$m \leftarrow$ **await** $m_{1\to 2}.\mathrm{pop}()$
$c' \leftarrow \mathrm{Enc}(\mathrm{pk}_2, m)$
$m \leftarrow \mathrm{Dec}(\mathrm{sk}_2, c')$
return m

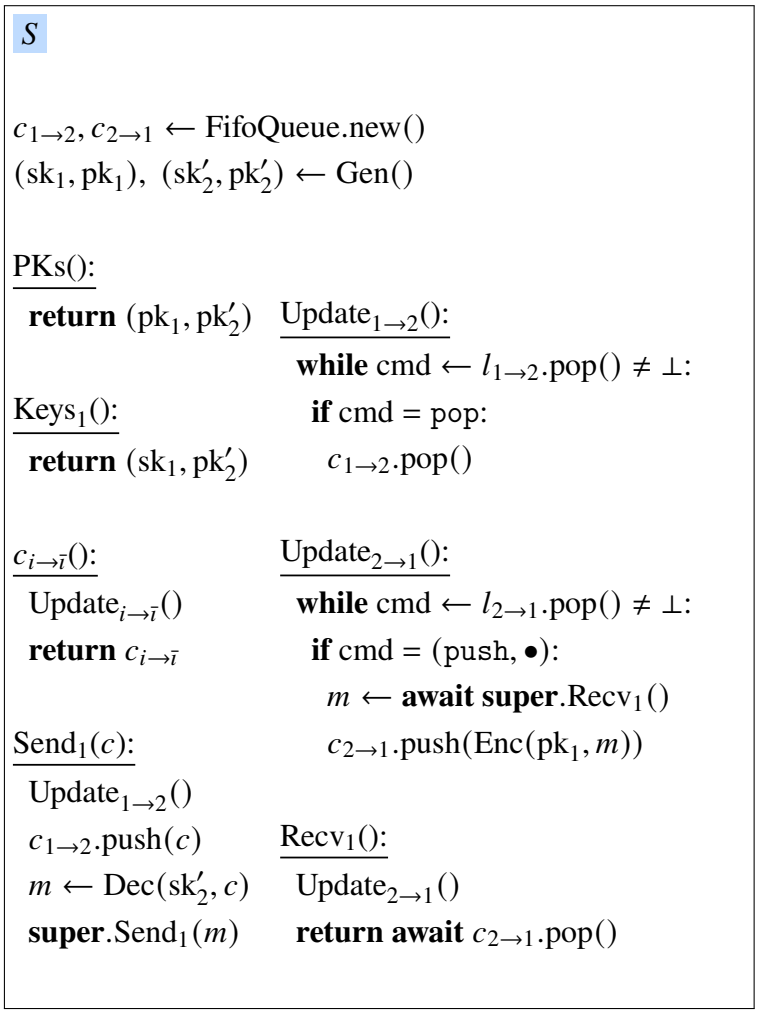
$\mathrm{Send}_2(m)$:
$c \leftarrow \mathrm{Enc}(\mathrm{pk}_1, m)$
$c_{2\to 1}.\mathrm{push}(c)$

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 $\mathcal{Q}$ is that the ciphertext queues $c_{i\to\bar{i}}$ 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 $\mathcal{Q}$.

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 $\mathcal{Q}$, 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 $\mathcal{Q}$ will be able to receive messages on behalf of the first party, allowing us to retrieve the message, and then create a simulated ciphertext by encrypting it.

Putting these ideas together, we write:

$\Gamma^3 =$ 

$\circ\ \mathrm{Inst}_{\mathrm{M}}(\mathcal{Q})$

$\otimes$

$1(\{\mathrm{Send}_2, \mathrm{Recv}_2\})$

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 re-encrypt 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 subsection 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.

$F[\text{Add}]$

$x_1, \ldots, x_n \leftarrow \bot$

$(1)\text{Add}_i(x)$:	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

$\mathscr{P}$[IdealRand] is characterized by:

- $F := 1(\text{Add})$,
- Leakage = {Leak},
- And n players defined via the following system, for $i \in [n]$:

P_i

(1)Rand$_i$():
$x \xleftarrow{\$} \mathbb{G}$
return await Add$_i(x)$

Protocol 5.7: Ideal Random Protocol

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[Add] functionality into the protocol itself, which makes the ideal protocol technically $\mathscr{P}$[IdealRand] $\circ$ F[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.

F[Add′]

$x_1, \ldots, x_n \leftarrow \bot$

(1)Add$_i(x)$:
$x_i \leftarrow x$
wait $\forall i.\ x_i \neq \bot$
return $\sum_i x_i$

Leak():
if $\exists i.\ x_i = \bot$:
return (`waiting`, $\{i \mid x_i = \bot\}$)
return (`done`, $[x_i \mid i \in [n]]$)

Game 5.8: Addition Functionality

The difference in F[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 cor-

rupted. It then holds that:

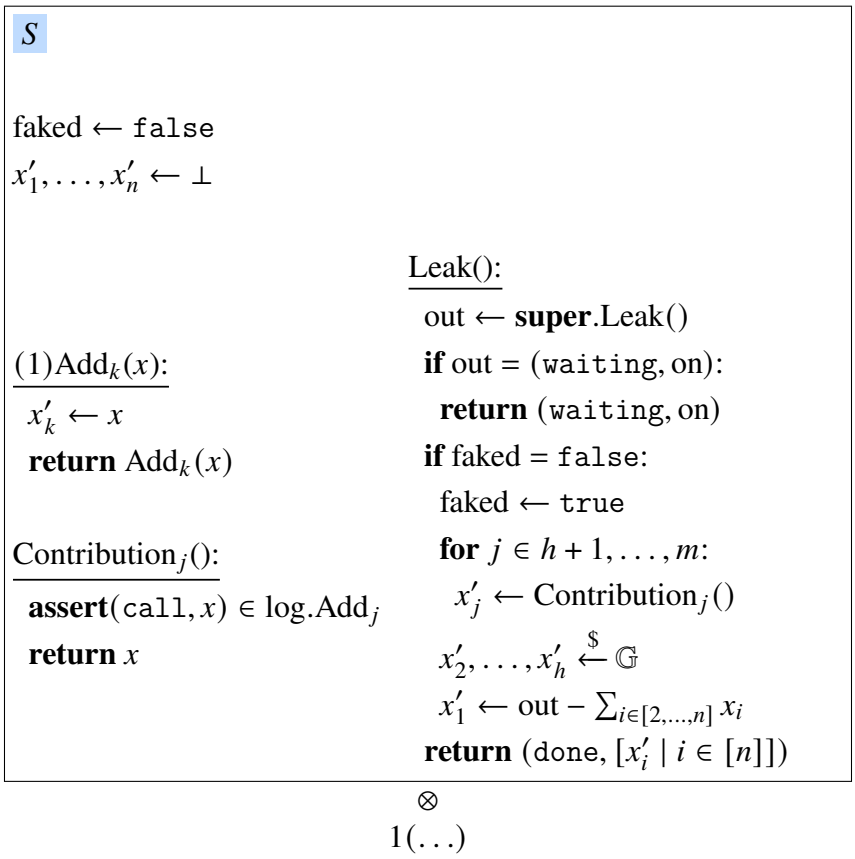
$$\mathscr{P}[\text{IdealRand}] \circ F[\text{Add}'] \overset{0}{\rightsquigarrow}_{\mathscr{C}} \mathscr{P}[\text{IdealRand}] \circ F[\text{Add}]$$

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:



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

$F[\text{Com}]$

$c_1, \ldots, c_n \leftarrow \bot$
$o_1, \ldots, o_n \leftarrow \texttt{false}$

(1)$\text{Commit}_i(x)$:
$c_i \leftarrow x$

(1)$\text{Open}_i()$:
assert $c_i \neq \bot$
$o_i \leftarrow \texttt{true}$

$\text{View}_i(x)$:
if $c_i = \bot$:
return `empty`
if $\neg o_i$:
return `set`
else:
return $(\texttt{open}, c_i)$

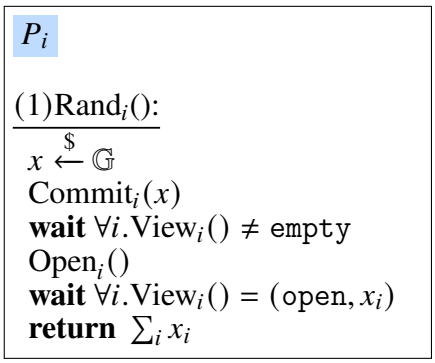
Game 5.9: Commitment Functionality

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.

$\mathscr{P}[\text{Rand}]$ is characterized by:

- $F := F[\text{Com}]$,
- Leakage $= \{\text{View}_1, \ldots, \text{View}_n\}$,
- And n players defined via the following system, for $i \in [n]$:

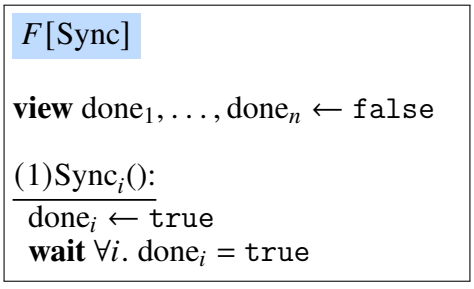


Protocol 5.10: Random Protocol

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 separate protocol, making use of a "synchronization" functionality, defined in Game 5.11.



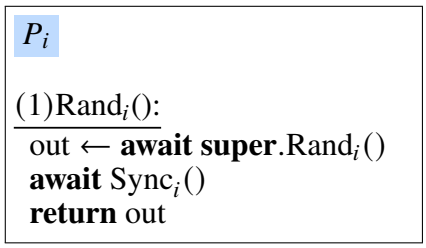
Game 5.11: Synchronization Game

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 $\mathcal{Q}$, and defined in Protocol 5.12

$\mathcal{Q}$ is characterized by:

- $F = F[\text{Sync}]$,
- Leakage := $\{\text{done}_1, \ldots, \text{done}_n\}$,
- And n players defined by the following system, for $i \in [n]$:



Protocol 5.12: Synchronized Random Protocol

The full protocol we consider is $\mathcal{Q} \lhd (\mathcal{P}[\text{IdealRand}] \circ F[\text{Add}])$, which can perfectly simulate $\mathcal{P}[\text{Rand}]$, as we now prove.

Claim 5.3. Let $\mathcal{C}$ be the class of corruptions where up to $n-1$ parties are corrupt. Then it holds that:

$$\mathcal{P}[\text{Rand}] \overset{0}{\rightsquigarrow}_{\mathcal{C}} \mathcal{Q} \lhd (\mathcal{P}[\text{IdealRand}] \circ F[\text{Add}])$$

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:

$$\mathcal{P}[\text{IdealRand}] \circ F[\text{Add}'] \overset{0}{\rightsquigarrow}_{\mathcal{C}} \mathcal{P}[\text{IdealRand}] \circ F[\text{Add}]$$

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 $\text{Inst}_C(\mathcal{P}[\text{Rand}])$, to get:

Γ^0

$x_1, \ldots, x_n, \text{rush}_{m+1}, \ldots, \text{rush}_n \leftarrow \bot$
$\text{o}_1, \ldots, \text{o}_n \leftarrow \texttt{false}$
$\text{log}_j \leftarrow \text{NewLog}()$

(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** $\sum_i x_i$

$\text{View}_i()$:
- **if** $x_i = \bot$:
  - **return** `empty`
- **if** $\neg o_i$:
  - **return** `set`
- **else**:
  - **return** $(\texttt{open}, c_i)$

(1)$\text{Rand}_j()$:
- $\text{log}_j.\text{Rand}_j.\text{push}(\texttt{input})$
- $x_i \xleftarrow{\$} \mathbb{G}$
- $\text{log}_j.\text{Commit}_j.\text{push}((\texttt{call}, x_i))$
- **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** $\sum_i x_i$

(1)$\text{Commit}_k(x)$:
- $x_k \leftarrow x$

(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

$x_1, \ldots, x_n, \text{rush}_{m+1}, \ldots, \text{rush}_n \leftarrow \bot$
$\text{done}_1, \ldots, \text{done}_n \leftarrow \texttt{false}$
$\log_j \leftarrow \text{NewLog}()$

(1)$\text{Rand}_i()$:
- $x_i \xleftarrow{\$} \mathbb{G}$
- **wait** $\forall i.\ \text{View}_i \neq \texttt{empty}$
- $\text{done}_i \leftarrow \texttt{true}$
- **wait** $\forall i.\ \text{View}_i = (\texttt{open}, x_i)$
- **return** $\sum_i x_i$

$\text{View}_i()$:
- **if** $\text{Leak}() = (\texttt{waiting}, s) \wedge i \in s$:
  - **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** $\texttt{set}$

$\log_j()$:
- $\log'_j \leftarrow \text{NewLog}()$
- $\log'_j.\text{Rand}_j \leftarrow \log_j.\text{Rand}_j$
- $\log'_j.\text{Commit}_j \leftarrow \log_j.\text{Add}_j$
- $\log'_j.\text{Open}_j \leftarrow \log_j.\text{Sync}_j$
- **return** $\log'_j$

(1)$\text{Rand}_j()$:
- $\log_j.\text{Rand}_j.\text{push}(\texttt{input})$
- $x_i \xleftarrow{\$} \mathbb{G}$
- $\log_j.\text{Add}_j.\text{push}((\texttt{call}, x_i))$
- **wait** $\forall i.\ \text{View}_i \neq \texttt{empty}$
- $\log_j.\text{Sync}_j.\text{push}(\texttt{call})$
- $o_i \leftarrow \texttt{true}$
- **wait** $\forall i.\ \text{View}_i = (\texttt{open}, x_i)$
- **return** $\sum_i x_i$

(1)$\text{Commit}_k(x)$:
- $\text{rush}_k \leftarrow x$
- $x_k \leftarrow x$

(1)$\text{Open}_k()$:
- **assert** $\text{rush}_k \neq \bot$
- $\text{done}_k \leftarrow \texttt{true}$

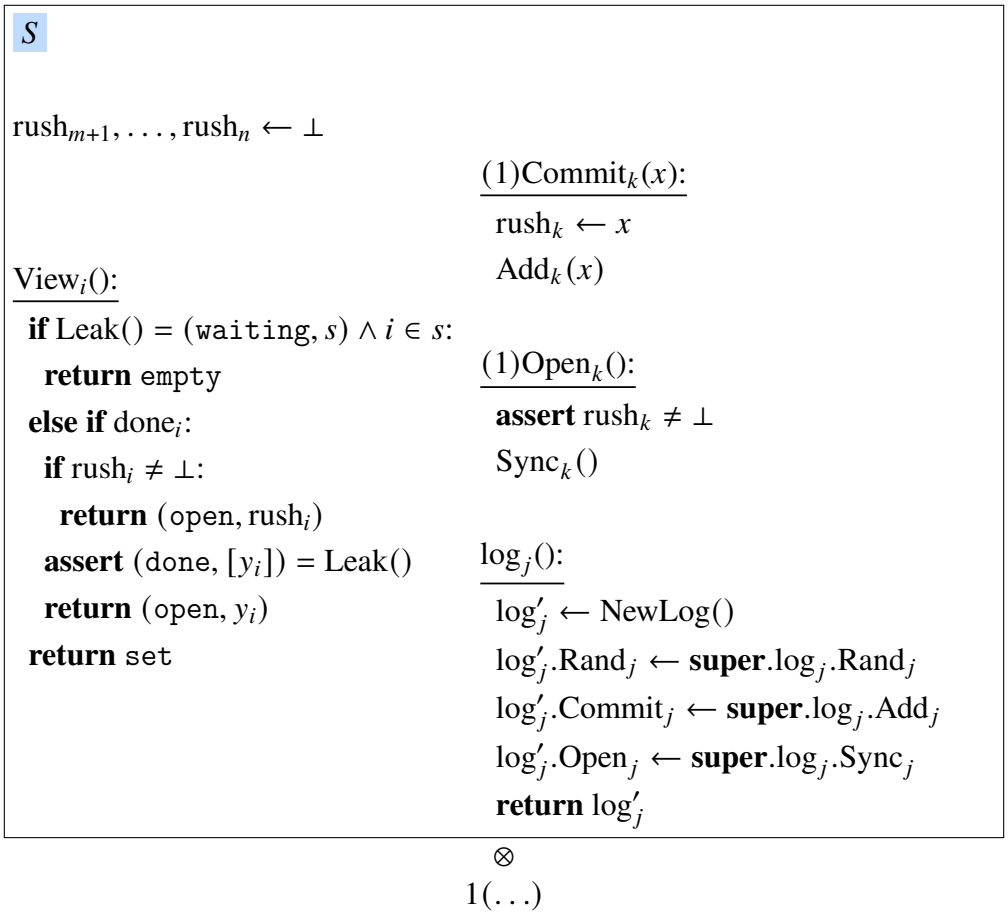
$\text{Leak}()$:
- **if** $\exists i.\ x_i = \bot$:
  - **return** $(\texttt{waiting}, \{i \mid x_i = \bot\})$
- **return** $(\texttt{done}, [x_i \mid i \in [n]])$

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 committing. 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:



And this concludes our proof, having shown that:

$$\text{Inst}_C(\mathscr{P}[\text{Rand}]) = \text{SimInst}_{S,C}(\mathscr{Q} \triangleleft (\mathscr{P}[\text{IdealRand}] \circ F[\text{Add}]))$$

■

6 Differences with UC Security

In this section, we outline a few differences between the framework we've developed, MPS, and that of UC security [Can00]. Despite these differences, we

think that the frameworks ultimately remain quite compatible, in that proofs in one framework should translate well to proofs in the other. This process is not, by any means, automatic, as is the case for other variants of UC, such as [CCL15].

Foundations without Turing Machines

One major technical difference is that MPS doesn't specify a concrete computational model. Rather than using interactive randomized Turing machines, as most frameworks do, we instead just assume the existence of computable randomized functions, and then build everything on top of that foundation.

We believe that this makes the foundations simpler to understand, since the complicated details of Turing machines and various tapes are never mentioned, but it also makes proofs closer to the actual formalism.

In principle, UC proofs would need to make reference to interactive Turing machines writing messages on each other's tapes. In practice, a much higher level language is used. The advantage of basing ourselves on state-separable proofs is that we can give a formal justification for this kind of high-level language, by providing precise semantics for the pseudo-code we use. Thus, we expect proofs in the MPS framework to be writable in a style close to the formalism itself, while also proving a high level of abstraction.

Semi-Honest Security without Randomness

Another technical difference is that our notion of semi-honest security does not allow an adversary to see the randomness sampled by a given party. Instead, they're allowed to see all function calls and messages sent by the party. As explained before, the main reason for this difference is that we ultimately want two protocols with equal parties to be consider equal protocols, under any corruption model, and being able to see the exact randomness being sampled is often enough to distinguish otherwise equal parties.

In practice, we don't expect this difference to matter, because meaningful randomness should affect the output calls and behavior of the adversary, and so the difference between these models are likely to come from more pathological examples.

Hybrid Only

In the usual presentation of UC security, simulation happens between a protocol in the *hybrid* world, where parties can potentially interact with an ideal functionality, and the *ideal* world, where the parties don't communicate between each other, instead interacting only with the ideal functionality.

In MPS, only the hybrid world exists. Protocols aren't simulated by ideal functionalities, instead, protocols are simulated by other protocols, and all protocols may make use of ideal functionalities.

The advantage of this approach is that it allows decomposing a larger simulation proof into multiple smaller proofs, which can then be stringed together via transitivity. The larger the gap between the protocols being simulated, the more complicated the simulator needs to be, and so this style of proof can be much simpler.

Corruption Agnostic Ideal Functionalities

Another technical difference is that in MPS, ideal functionalities are not aware of which parties are corrupted, whereas some UC functionalities make use of this fact.

We don't think this is a necessary feature of the framework itself, since it can be modeled by having slightly more complicated protocols on top of an ideal functionality. For example, one common use of this kind of "corruption aware" functionality is to describe *endemic* functionalities, where malicious parties are allowed to choose their own randomness. This can be written by having the functionality alter its behavior based on which parties are corrupt, allowing them to choose their own randomness.

In MPS, we can instead just have a small wrapping protocol around the functionality, where honest parties sample a random value before calling the functionality. Malicious parties are then free to sample a biased value, deviating from the protocol.

In general, one can always have the functionality behave differently for certain inputs, and then restrict honest parties to never trigger this behavior, thus allowing the functionality to behave differently for malicious parties.

The Lack of Adversaries

In the traditional presentations of UC, simulation is a statement of the form "for all adversaries, there exists a simulator, such that for all environments...". In MPS, we eliminate the notion of adversary entirely, instead simply considering the environment to be the adversary.

This is actually a possibility in UC itself. Subsequent versions of [Can00] include an explicit proof that it suffices to prove security against the "dummy adversary", which simply does whatever the environment tells it to do. We can thus consider MPS to implicitly use such a dummy adversary.

The Lack of Session IDs

Another big difference is that we do away with the use of "session IDs", at least explicitly. These are most often used to distinguish between multiple instances of a protocol in a given execution. These can still be used in our case, but are more implicit.

For example, multiple instances of a protocol would be written $\mathscr{P} \otimes \mathscr{P}$. Technically, this is disallowed, but we could fix this by renaming all of the functions in one instance of the protocol, so that there's no longer any conflict. If we use this protocol, in practice it means that we have a way of distinguishing between the messages belonging to once instance of the protocol from the other instance. One way of accomplishing this would be assigning session ids, but these aren't a formal part of our framework.

An exploration of secure composition without session IDs was also conducted for UC security and other models in [KT11].

"Timing Side-Channels"

One unfortunate strength of the MPS framework is that the adversary is able to observe more timing properties of protocol execution. Indeed, they are able to observe how many times a given function yields before returning a result, or simply whether or not a function can return a result given the current state of execution. This is a consequence of the more asynchronous nature of execution we have for protocols.

This is arguably present in some variants of UC already, depending on the precision of the proofs. Indeed, if the adversary is able to stall or abort execution, then this needs to be reflected in the functionality targeted by the simulation proof. This is how the notion of "MPC with abort" arises.

In some cases though, it seems like the visible delays are an undesirable consequence of simulation that is required to be, perhaps, too precise. We think that further work could develop more relaxed notions of simulation, which can paper over inessential differences like those of timing and delay.

Clearer Connection with Games

Finally, we believe that a major advantage of the MPS framework is that it provides a much simpler bridge between standalone security with games, and the composable security of protocols. Ideal functionalities are simply games, and we have theorems showing that we can use indistinguishability results for games to produce simulation results for protocols. Furthermore, simulation arguments ultimately boil down to an argument about games, and so this can motivate the

intricate games that one might find in the analysis of protocols such as messaging.

7 Conclusion

In this work, we sought to develop a modular framework for analyzing the security of protocols. We did this by extending the standalone security formalism of state-separable proofs [BDF+18]. The result is a framework for protocol security with similar modular properties to those of state-separable proofs, and with a strong connection with that formalism, allowing for results in standalone security to be used in showing the security of protocols.

While we believe our framework is already suitable for proving the security of protocols, we expect shortcomings to be discovered as the framework sees more use. The novelty of the framework also provides a disadvantage in that not many proofs have been written in it, and many common UC idioms may not translate directly either. We hope that this disadvantage can diminish over time as more more work is conducted using this or similar frameworks.

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.

[Can00] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `https://eprint.iacr.org/2000/067`.

[CCL15] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 3–22. Springer, Heidelberg, August 2015.

[CD+15] Ronald Cramer, Ivan Bjerre Damgård, et al. *Secure multiparty computation*. Cambridge University Press, 2015.

[HL10] Carmit Hazay and Yehuda Lindell. A note on the relation between the definitions of security for semi-honest and malicious adversaries. Cryptology ePrint Archive, Report 2010/551, 2010. `https://eprint.iacr.org/2010/551`.

[HS15] Dennis Hofheinz and Victor Shoup. GNUC: A new universal composability framework. *Journal of Cryptology*, 28(3):423–508, July 2015.

[KT11] Ralf Küsters and Max Tuengerthal. Composition theorems without pre-established session identifiers. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011*, pages 41–50. ACM Press, October 2011.

[Mei22] Lúcás Críostóir Meier. State-separable proofs for the curious cryptographer. `https://cronokirby.com/posts/2022/05/state-separable-proofs-for-the-curious-cryptographer/`, 2022.

[Ros] Mike Rosulek. The joy of cryptography. `https://joyofcryptography.com`.

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

A public key encryption scheme consists of types $\mathbf{PK}$, $\mathbf{PK}$, $\mathbf{DK}$, $\mathbf{M}$, $\mathbf{C}$, along with probabilistic functions $\text{Enc} : (\mathbf{PK}, \mathbf{M}) \xleftarrow{\$} \mathbf{C}$ and $\text{Dec} : (\mathbf{SK}, \mathbf{C}) \to \mathbf{M}$. By $\mathbf{M}(|m|)$ we denote the distribution of messages with the same length as m. We require that $\mathbf{M}(|m|)$ is efficiently sampleable, and we require that we can sample $(\mathbf{SK}, \mathbf{PK})$ via an algorithm Gen.

The encryption scheme must satisfy a correctness property:

$$\forall(\text{sk}, \text{pk}) \leftarrow \text{Gen}(),\ m \in \mathbf{M}.\ P[\text{Dec}(\text{sk}, \text{Enc}(\text{pk}, m)) = m] = 1$$

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_b

$(\text{sk}, \text{pk}) \leftarrow \text{Gen}()$

Challenge(m_0):
$m_1 \xleftarrow{\$} \mathbf{M}(|m|)$
return $\text{Enc}(\text{pk}, m_b)$

In essence, an adversary cannot distinguish between an encryption of a message of their choice and that of a random message.