Games That Talk:
New Foundations for Composable Security

Lúcás Críostóir Meier
lucas@cronokirby.com

January 16, 2023

Abstract

We do things with UC security.

1 Introduction

[Mei22]

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

Theorem 1.1 (Cool Beans). Woah mama

And that's what matters.

■

Lemma 1.2. Woah mama again!

Corollary 1.3. Woah mama again!

Γ^0	
$x \leftarrow 3$	
if $x + 2$:	
$\quad y \xleftarrow{\$} \mathbb{F}_q$	
$m \Rightarrow \langle \mathcal{P}_i, \mathcal{P}_j \rangle$	$y \leftarrow 4$
$m \Leftarrow \langle \text{OT}, \mathcal{P}_i \rangle$	$x \leftarrow 3$
<u>Foo(x, y):</u>	
$\quad$Bar(x, y)	

Game 1.1: Some Game

1.1 Relevance of Time Travel

stuff

IND-CCA
$x \leftarrow 4$

Protocol 1.2: Some Protocol

IND-CCA
$x \leftarrow 4$

Protocol 1.3: Some Protocol

IND-CCA
$x \leftarrow 3$

Functionality 1.4: Encryption

2 State-Separable Proofs

3 Games That Talk

3.1 Async Functions

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

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

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

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

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

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

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

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

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

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

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

This gives us a conditional jump, and by using $\mathtt{true}$ 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 $F.\mathtt{pc}$, which stores the program counter for the function. This is different than the local $\mathtt{pc}$ which is while the function is execution. $F.\mathtt{pc}$ is simply used to remember the program counter after a yield statement.

The function now starts with:

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

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 $\mathtt{input}$, which holds the input supplied to the function. At the start of the function body, we add:

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

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:

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

The semantics of **return** v become:

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

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

□

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

Syntax 3.2. 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.

□

In such situations, often we don't particularly care about the intermediate yields of a function, and want to wait for the final result, potentially yielding to our own caller. We define these semantics via the **await** statement.

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

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

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

□

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

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

More formally, we define:

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

As follows:

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

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.

□

3.2 Channels and System Composition

Definition 3.5 (Systems). A *system* is a package which uses channels.

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

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

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

□

Definition 3.6. We can compile systems to not use channels. We denote by $\text{NoChan}(S)$ the package corresponding to a system S, with the use of channels replaced with function calls.

Channels define two new syntactic constructions, for sending and receiving along a channel. We replace these with function calls as follows:

Sending, with $m \Rightarrow P$ becomes:

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

Receiving, with $m \Leftarrow P$ becomes:

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

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

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

□

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

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

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

Recv$_{A_i}()$:
  while $q[A_i]$.IsEmpty()
    yield
  $q[A_i]$.Next()

Game 3.1: Channels

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

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

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

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

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

Definition 3.7. Given some equality relation $\sim$ on packages, we can lift that relation to systems by defininig:

$$A \sim B \iff \text{SysPack}(A) \sim \text{SysPack}(B)$$

□

Definition 3.8 (System Tensoring). Given two systems, A and B, with $\text{Out}(A) \cap \text{Out}(B) = \emptyset$, we can define their tensor product $A * B$, which is any system satisfying:

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

□

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

$$\text{NoChan}(A * B) = \text{NoChan}(A) \otimes \text{NoChan}(B)$$
$$\text{(Out/In)Chan}(A * B) = \text{(Out/In)Chan}(A) \cup \text{(Out/In)Chan}(B)$$
$$\text{In}(A * B) = \text{In}(A) \cup \text{In}(B)$$

This implies the following lemma.

Lemma 3.1. System tensoring is associative, i.e. $A * (B * C) = (A * B) * C$.
Proof: Starting from the definition of tensoring, we have:

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

We can then apply the corrollaries we've just derived to show that this is equal to:

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

(Using the associativity of $\otimes$ for *packages* as well).

With the same reasoning, we can derive the same package from $(A * B) * C$, letting us conclude that $\text{SysPack}(A * (B * C)) = \text{SysPack}((A * B) * C)$, and thus that $A * (B * C) = (A * B) * C$.

■

Lemma 3.2. System tensoring is commutative, i.e. $A * B = B * A$ **Proof:** This follows from the commutativity of $\otimes$ and $\cup$. ■

Definition 3.9 (Overlapping Systems). Two systems A and B overlap if $\text{Chan}(A) \cap \text{Chan}(B) \neq \emptyset$.

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

Definition 3.10 (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 such that:

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

□

Lemma 3.3. System composition is associative, i.e. $A \circ (B \circ C) = (A \circ B) \circ C$. **Proof:** This follows from the associativity of $\circ$ for *packages*. ■

Lemma 3.4 (Interchange Lemma). Given systems A, B, C, D such that $A \circ B$ and $C \circ D$ are well defined, $A * C$ and $B * D$ are well defined, and neither A nor C overlap with B or D, i.e. the following relation holds:

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

Proof: First, we need to develop a few general facts about $\text{SysPack}(A \circ B)$, $\text{Chan}(A \circ B)$ and $\text{NoChan}(A \circ B)$, like those we developed for $A * B$.

As a consequence of how $A \circ B$ is defined, by unrolling $\text{SysPack}(A \circ B)$, we get:

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

Applying the interchange lemma for packages a couple times, we then get:

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

And then, recalling that $\text{Channels}(S) \otimes \text{Channels}(R) = \text{Channels}(S \cup R)$, we conclude that:

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

Next we apply these facts, along with those derived for $A * B$ to tackle the main lemma.

Starting from $\text{SysPack}((A * C) \circ (B * D))$, we can apply the above results to get:

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

Then, applying what we know about $A * B$ in general, we get:

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

Applying the interchange lemma for packages again, along with what we know about $A \circ B$, we get:

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

Noting that $\text{Chan}(A, B, C, D) = \text{Chan}(A \circ B, C \circ D)$, and that $\text{In}(B, D) = \text{In}(A \circ B, C \circ D)$, we realize that the expression above is equal to:

$$\text{SysPack}((A \circ B) * (C \circ D))$$

■

Definition 3.11 (System Games). Analogously to games, we define a *system game* as a system S with $\text{In}(S) = \emptyset$.

Definition 3.12 (System Game Reductions). We can also define notions of reductions for system game (pairs).

First, we define:

$$\epsilon(\mathcal{A} \circ S_b) := \epsilon(\mathcal{A} \circ \text{SysPack}(S_b))$$

We then also use the syntax sugar of:

$$S_b \leq f(G_b^1, G_b^2, \ldots)$$

as shorthand for, $\forall \mathcal{A}.\ \exists \mathcal{B}_1, \ldots$:

$$\epsilon(\mathcal{A} \circ S_b) \leq f(\epsilon(\mathcal{B}_1 \circ G_b^1), \epsilon(\mathcal{B}_2 \circ G_b^2), \ldots)$$

We also sometimes omit explicitly writing S_b, instead writing just S, if it's clear that we're talking about a pair of systems.

□

Similar properties hold for reductions:

Lemma 3.5. $A \circ G_b \leq G_b$.

Proof: $\text{SysPack}(A \circ G_b) = \text{SysPack}(A) \circ \text{SysPack}(G_b) \leq \text{SysPack}(G_b)$. ■

Lemma 3.6. There exists system games A, G_B such that G_B is secure but $A * G_b$ is insecure.

Proof: Consider:

G_b

<u>Cheat():</u>
$b \Rightarrow P$
$\hat{b} \Leftarrow Q$
return $\hat{b}$

A

<u>Run():</u>
$b \Leftarrow P$
$b \Rightarrow Q$

Clearly, G_b is secure in isolation, since no other system is present to provide a value on Q, so G_b will block forever in the cheating function.

However, when linked with A, this cheating function will return b, allowing an adversary to break the game with probability 1.

■

4 Protocols and Composition

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

- Systems $P_1, \ldots, P_n$, called *players*
- A package F, called the *ideal functionality*

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

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

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

Second, we require that the functions the systems depend on are disjoint:

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

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

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

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

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

Finally, we define

$$\begin{aligned}
&\text{IdealIn}(\mathcal{P}) := \text{In}(F) \\
&\text{Leakage}(\mathcal{P}) := \text{Out}(F) / \left(\bigcup_{i \in [n]} \text{In}(P_i) \right)
\end{aligned}$$

□

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

□

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

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

$\square$

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

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

$\square$

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

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

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

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

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

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

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

Third, we require that the leakages of one game aren't use in the other:

$$\begin{aligned}\text{Leakage}(\mathcal{P}) \cap \text{In}(\mathcal{Q}) = \emptyset \\ \text{Leakage}(\mathcal{Q}) \cap \text{In}(\mathcal{P}) = \emptyset\end{aligned}$$

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

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

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

The players will consist of:

$$\mathcal{P}.P_i \circ \left(\mathop{\Large *}_{\mathcal{Q}.P_j \text{ used by } \mathcal{P}.P_i} \mathcal{Q}.P_j \right)$$

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

The ideal functionality is $\mathcal{P}.F \otimes \mathcal{Q}.F$.

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:

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

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

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

as required.

By definition, the dependencies In of each player in $\mathcal{P} \lhd \mathcal{Q}$ are the union of several players in $\mathcal{Q}$, so disjointness property continues to hold.

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

□

Lemma 4.2. Horizontal composition is associative, i.e. $\mathcal{P} \lhd (\mathcal{Q} \lhd \mathcal{R}) \equiv (\mathcal{P} \lhd \mathcal{Q}) \lhd \mathcal{R}$ for all protocols $\mathcal{P}, \mathcal{Q}, \mathcal{R}$ 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 $\mathcal{R}$ that get used in $\mathcal{P}$ via the players in $\mathcal{Q}$. To that end, we say that $\mathcal{P}.P_i$ *uses* $\mathcal{R}.P_j$ if there exists $\mathcal{Q}.P_k$ such that $\mathcal{P}.P_i$ uses $\mathcal{Q}.P_k$, and $\mathcal{Q}.P_k$ uses $\mathcal{R}.P_j$.

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

$$\mathcal{P}.P_i \circ \left(\mathop{\Large *}_{\mathcal{Q}.P_j \text{ used by } \mathcal{P}.P_i} \mathcal{Q}.P_j \circ \left(\mathop{\Large *}_{\mathcal{R}.P_k \text{ used by } \mathcal{Q}.P_j} \mathcal{R}.P_k \right) \right)$$

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

$$\left(\mathcal{P}.P_i \circ \mathop{\Large *}_{\mathcal{Q}.P_j \text{ used by } \mathcal{P}.P_i} \mathcal{Q}.P_j\right) \circ \left(\mathop{\Large *}_{\mathcal{R}.P_k \text{ used by } \mathcal{P}.P_i} \mathcal{R}.P_k\right)$$

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

$$\mathcal{P}.P_i \circ \left(\mathop{\Large *}_{\mathcal{Q}.P_j \text{ used by } \mathcal{P}.P_i} \mathcal{Q}.P_j\right) \circ \left(\mathop{\Large *}_{\mathcal{Q}.P_j} \left(\mathop{\Large *}_{\mathcal{R}.P_k \text{ used by } \mathcal{Q}.P_j} \mathcal{R}.P_k\right)\right)$$

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

$$\mathcal{P}.P_i \circ \left(\mathop{\Large *}_{\mathcal{Q}.P_j \text{ used by } \mathcal{P}.P_i} \mathcal{Q}.P_j \circ \left(\mathop{\Large *}_{\mathcal{R}.P_k \text{ used by } \mathcal{Q}.P_j} \mathcal{R}.P_k\right)\right)$$

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

■

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

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

The players of $\mathcal{P} \otimes \mathcal{Q}$ consist of all the players in $\mathcal{P}$ and $\mathcal{Q}$. The ideal functionality is $\mathcal{P}.F \otimes \mathcal{Q}.F$, unless $\mathcal{P}.F = \mathcal{Q}.F$, in which case the ideal functionality is simply $\mathcal{P}.F$. 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 $\text{In}(\mathcal{P}) \cap \text{In}(\mathcal{Q}) = \emptyset$, it also holds that for every P_i, P_j in $\mathcal{P} \otimes \mathcal{Q}$, we have $\text{In}(P_i) \cap \text{In}(P_j) = \emptyset$, since each player comes from either $\mathcal{P}$ or $\mathcal{Q}$.

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

□

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

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

Proof:

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

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

■

4.1 Corruption and Simulation

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

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

This is clearly equality preserving, by tautology.

□

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

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

1. log.A_i for each sending channel A_i,
2. log.B_i for each receiving channel B_i,
3. log.F for each input function F.

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

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

- m to log.B when a value m is received on channel B.

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

□

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

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

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

$\text{Corrupt}_M(P)$

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

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

$\underline{\text{Recv}_{B_i}()\text{:}}$
  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 $P' = P$.

□

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

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

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

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

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

This results in a system S', which we then compose to get:

$$S_{\text{SH}} := \begin{pmatrix} S' \\ \otimes \\ 1(\{\text{Next}\}) \end{pmatrix} \circ \begin{pmatrix} \text{SharedRand} \\ \otimes \\ 1(\text{In}(S')) \end{pmatrix}$$

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

■

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

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

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

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

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

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

□

Some common classes are:

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

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

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

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

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

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

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

With this in hand, we define:

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

□

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

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

(provided these expressions are well defined)

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

■

Definition 4.12 (Associated Corruption Classes). Given two protocols $\mathcal{P}, \mathcal{Q}$ where $\otimes$ is well defined, a corruption class $\mathcal{C}$ for $\mathcal{Q}$ has a natural corruption class $\mathcal{C}'$ for $\mathcal{P} \otimes \mathcal{Q}$.

For each model $C \in \mathcal{C}$, the resulting $\mathcal{C}'$ will contain a model for each possible honest corruption of the players in P with efficient agents $A_1, \ldots$. In other words, the corruptions in this class will be those of $\mathcal{C}$, with $\mathcal{P}$ always behaving honestly.

We can also do the same for $\mathcal{P} \circ \mathcal{Q}$, but the corruption class $\mathcal{C}'$ is a bit trickier. We say that a corruption model C for $\mathcal{P}$ 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 $\mathcal{C}$. A corruption model C for $\mathcal{P}$ is compatible with a *class* of corruptions $\mathcal{C}$, if there exists a compatible model C' in $\mathcal{C}$.

With this in hand, the corruption class $\mathcal{C}'$ for $\mathcal{P} \circ \mathcal{Q}$ is the largest (closed) corruption class $\mathcal{C}'$ for $\mathcal{P}$ such that each $C \in \mathcal{C}'$ is compatible with $\mathcal{C}$. Because of the definition of $\circ$, a corruption model for $\mathcal{P}$ naturally yields a model for $\mathcal{P} \circ \mathcal{Q}$, so this is well defined.

□

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 $\mathcal{P} \circ \mathcal{Q}$, 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 $\mathcal{P} \circ \mathcal{Q}$ 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 $\mathcal{C}$.

This extends to corruption *classes* as well. A corruption class $\mathcal{C}'$ is compatible with a class $\mathcal{C}$, if for every $C \in \mathcal{C}$ there exists a compatible $C' \in \mathcal{C}'$.

□

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

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

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

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

We can apply a few observations here:

1. Since $\mathcal{C}'$ is compatible with $\mathcal{C}$, then $\mathcal{Q}.P_i$ follows a corruption from $\mathcal{C}$.

2. $\mathcal{R}$ can be written as $\mathcal{R}_{\mathcal{P}} \otimes \mathcal{R}_{\mathcal{Q}}$, with one system using channels in $\mathcal{P}$, and the other using channels in $\mathcal{Q}$.
3. Since protocols are closed, we can use $\otimes$ between the players in $\mathcal{P}$ and $\mathcal{Q}$, since they never send messages to each other.

This results in the following:

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

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

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

■

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

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

where $O := \text{Out}(\text{Inst}_{C'}(\mathcal{P} \lhd \mathcal{Q}))$, and $\mathcal{R}_{\mathcal{P}} \circ \mathcal{R}_{\mathcal{Q}} = \mathcal{R}$ are a decomposition of the router $\mathcal{R}$ for $\mathcal{P} \lhd \mathcal{Q}$.

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

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

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

First, some observations about $\text{Corrupt}_\kappa(P \circ (Q_1 * \cdots))$.

In the case of malicious corruption, we have:

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

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. 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:

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

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

And for honest corruption, we have

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

Now, the compatibility condition of C' relative to C does not guarantee that if $\mathcal{P}.P_i$ uses $\mathcal{Q}.P_j$, then $\mathcal{Q}.P_j$ has the same level of corruption: it only guarantees a level of corruption at least as strong. By Lemma 4.4, 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 $C'_i = \text{M}$:

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

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

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

$$\text{Corrupt}_{C'}((\mathcal{P} \lhd \mathcal{Q}).P_i) = 1(O_i) \circ \begin{pmatrix} \text{Corrupt}_{\text{C'}}(P) \\ \otimes \\ 1(L_i) \end{pmatrix} \circ \left(\mathop{\Large *}_{\mathcal{Q}.P_j \text{ used by } \mathcal{P}.P_i} S_j \circ \text{Corrupt}_C(\mathcal{Q}.P_j) \right)$$

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

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

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

$$\text{Corrupt}_{C'}((\mathcal{P}\lhd\mathcal{Q}).P_i) = \text{Corrupt}_{C'}(P)\circ\left(\mathop{\Large\circledast}_{\mathcal{Q}.P_j \text{ used by } \mathcal{P}.P_i} S_j \circ \text{Corrupt}_C(\mathcal{Q}.P_j)\right)$$

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

$$\text{Corrupt}_{C'}((\mathcal{P}\lhd\mathcal{Q}).P_i) = 1(O_i)\circ\begin{pmatrix}\text{Corrupt}_{C'}(P)\\ \otimes\\ 1(L_i)\end{pmatrix}\circ\left(\mathop{\Large\circledast}_{\mathcal{Q}.P_j \text{ used by } \mathcal{P}.P_i} S_j \circ \text{Corrupt}_C(\mathcal{Q}.P_j)\right)$$

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

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

$$\text{Routed}(\text{Corrupt}_{C'}((\mathcal{P} \lhd \mathcal{Q}).P_i)) =$$
$$1(O_i) \circ \begin{pmatrix}\text{Routed}(\text{Corrupt}_{C'}(P))\\ \otimes\\ 1(L_i)\end{pmatrix} \circ \left(\mathop{\Large\circledast}_{\mathcal{Q}.P_j \text{ used by } \mathcal{P}.P_i} S_j \circ \text{Routed}(\text{Corrupt}_C(\mathcal{Q}.P_j))\right)$$

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

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

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

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

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

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

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

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

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

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

■

4.2 Equality and Simulation

Definition 4.14 (Shape). We say that two protocols $\mathcal{P}$, $\mathcal{Q}$ have the same *shape* if there exists a protocol $\mathcal{Q}' \equiv \mathcal{Q}$ such that:

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

Definition 4.15 (Semantic Equality). We say that two closed protocols $\mathcal{P}$ and $\mathcal{Q}$, with the same shape, are equal under a class of corruptions $\mathcal{C}$, written as $\mathcal{P} =_{\mathcal{C}} \mathcal{Q}$, when we have:

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

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

□

Definition 4.16 (Indistinguishability). We say that two protocols $\mathcal{P}$

Lemma 4.8 (Literal $\implies$ Semantic). For any corruption class $\mathcal{C}$

$$\mathcal{P} \equiv \mathcal{Q} \implies \mathcal{P} =_{\mathcal{C}} \mathcal{Q}.$$

Proof: This follows directly from the definition of $\text{Inst}_C(\ldots)$, which will yield an equal system if its components are equal. ■

5 Differences with UC Security

6 Examples

7 Further Work

8 Conclusion

References

[Mei22] Lúcás Críostóir Meier. MPC for group reconstruction circuits. Cryptology ePrint Archive, Report 2022/821, 2022. `https://eprint.iacr.org/2022/821`.