Games That Talk:
New Foundations for Composable Security

Lúcás Críostóir Meier
lucas@cronokirby.com

January 5, 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!

```
$\Gamma^0$

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

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

Game 1.1: Some Game

1.1 Relevance of Time Travel

stuff

```
IND-CCA
$x \leftarrow 4$
```

Protocol 1.2: Some Protocol

```
IND-CCA
$x \leftarrow 4$
```

Protocol 1.3: Some Protocol

```
IND-CCA
$x \leftarrow 3$
```

Functionality 1.4: Encryption

2 State-Separable Proofs

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

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

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

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

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

The semantics of **return** v become:

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

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

□

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

Syntax 3.2. 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 $v \leftarrow$ **await** $F(\ldots)$ in a straightforward way:

```
$(\text{tag}, v) \leftarrow (\texttt{yield}, \bot)$
while $\text{tag} = \texttt{yield}$ :
  if $v \neq \bot$ :
    yield
  $(\text{tag}, v) \leftarrow F(\ldots)$
```

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:

```
**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.

□

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

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

□

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:

$$\text{Channels.Send}(m, P)$$

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.

□

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

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

Send($m, A_i$):
  $q[A_i]$.Push($m$)

Recv($A_i$):
  while $q[A_i]$.IsEmpty()
    yield
  $q[A_i]$.Next()
```

Game 3.1: Channels

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 definining:

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

□

Definition 3.8 (System Tensoring). Given two systems, A and B, 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:

$$\begin{aligned} &\text{NoChan}(A * B) = \text{NoChan}(A) \otimes \text{NoChan}(B) \\ &\text{Chan}(A * B) = \text{Chan}(A) \cup \text{Chan}(B) \\ &\text{In}(A * B) = \text{In}(A) \cup \text{In}(B) \end{aligned}$$

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 SysPack$(A * (B * C)) =$ 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 Chan$(A) \cap$ 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 satisfying: add constraints like for packages

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

□

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*. ■

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 3.4. $A \circ G_b \leq G_b$.

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

Lemma 3.5. There exists system games A, G_B such that G_B is secure but $A * G_b$ is insecure.

Proof: Consider:

```
$G_b$

Cheat():
  $b \Rightarrow P$
  $\hat{b} \Leftarrow Q$
  return $\hat{b}$
```

```
A

Run():
  $b \Leftarrow P$
  $b \Rightarrow Q$
```

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 (Open Protocols). An *open 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:

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

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

□

Definition 4.2 (Vertical Composition). Given an open protocol $\mathcal{P}$ and a package G, satisfying $\text{IdealIn}(\mathcal{P}) \subseteq \text{Out}(G)$, we can define the open protocol $\mathcal{P} \circ G$.

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

□

Definition 4.3 (Horizontal Composition). Given two open protocols $\mathcal{P}$, $\mathcal{Q}$, we can define the open protocol $\mathcal{P} \lhd \mathcal{Q}$, provided a few requirements hold.

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

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

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

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.