ILC: A Calculus for Composable, Computational Cryptography

[ ]

ACM Reference Format:
. 2020. ILC: A Calculus for Composable, Computational Cryptography. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ILC is a process calculus for writing programs as interactive turing machines, the model of computation underlying the Universal Composability (UC) framework. Our calculus adapts ITMs to a suset of the π-calculus through affine typing discipline. Therefore, well-typd ILC programs will type-check only if they are expressable as ITMs. In this document we provide a brief primer on UC to make sense of the typing rules, and we then demonstrate and explain the language by working through an example of a commitment protocol from the functionality to the full UC execution.

1 Overview

We first provide background on the universal composability framework and then give a tour of ILC.

1.1 Background on Universal Composability

Security proofs in the UC framework follow the real/ideal paradigm [3]. To carry out some cryptographic task in the real world, we define a distributed protocol that achieves the task across *many untrusted processes*. Then, to show that it is secure, we compare it with an idealized protocol in which processes simply rely on a *single trusted process* to carry out the task for them (and so security is satisfied trivially).

The program for this single trusted process is called an *ideal functionality* as it provides a uniform way to describe all the security properties we want from the protocol. Roughly speaking, we say a protocol π *realizes* an ideal functionality $\mathcal{F}$ (i.e., it meets its specification) if every adversarial behavior in the real world can also be exhibited in the ideal world.

Once we have defined π and $\mathcal{F}$, proving realization formally follows a standard rhythm:

1. The first step is a construction: We must provide a *simulator* $\mathcal{S}$ that translates any attack $\mathcal{A}$ on the protocol π into an attack on $\mathcal{F}$.
2. The second step is a relational analysis: We must show that running π under attack by any adversary $\mathcal{A}$ (the real world) is *indistinguishable* from running $\mathcal{F}$ under attack by $\mathcal{S}$ (the ideal world) to any distinguisher $\mathcal{Z}$ called the *environment*.

In particular, $\mathcal{Z}$ is an adaptive distinguisher: It interacts with both the real world and the ideal world, and the simulation is sound if no $\mathcal{Z}$ can distinguish between the two.

All of this is quite abstract so far, so it helps to visualize what the UC indistinguishability experiment looks like. Figure 1 illustrates the UC experiment with the real world on the left, the ideal world on the right, and with communication channels as connecting lines. There are several things worth noting here:

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

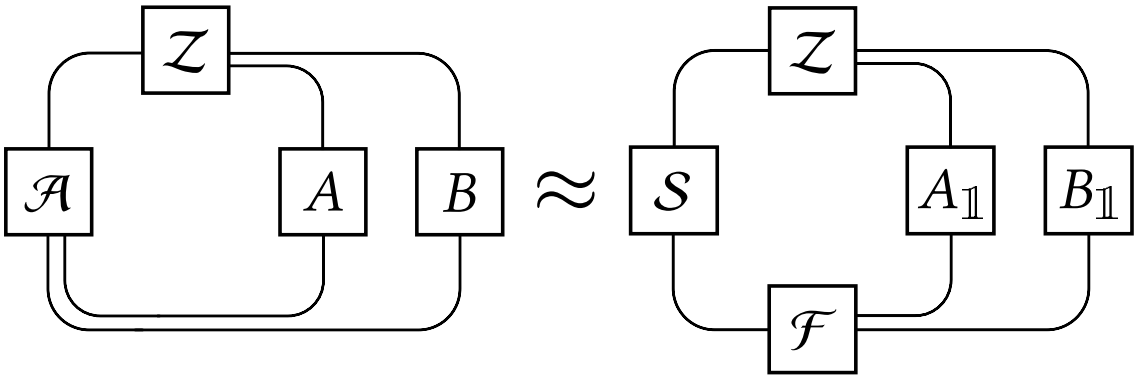

Figure 1. UC experiment with real world (left) and ideal world (right).

- For simplicity, protocols have two parties: A and B in the real world, and $A_{\mathbb{1}}$ and $B_{\mathbb{1}}$ in the ideal world.
- In the real world, parties A and B execute the protocol between each other, with all messages flowing through the adversary.
- In the ideal world, parties $A_{\mathbb{1}}$ and $B_{\mathbb{1}}$ are "dummy" parties, which simply relay messages between the environment $\mathcal{Z}$ and the ideal functionality $\mathcal{F}$.
- An environment $\mathcal{Z}$ can interact with each of the worlds in the same way, but the main underlying difference is that it is (indirectly) interacting with the ideal functionality in the ideal world. If it is the case that the real world protocol emulates $\mathcal{F}$, then no environment should be able to tell the two worlds apart.

As mentioned, the primary goal of this framework is *compositionality*. Suppose a protocol π is a protocol module that realizes a functionality $\mathcal{F}$ (a specification of the module), and suppose a protocol ρ, which relies on $\mathcal{F}$ as a subroutine, in turn realizes an application specification functionality $\mathcal{G}$. Then, the composed protocol $\rho \circ \pi$, in which calls to $\mathcal{F}$ are replaced by calls to π, also realizes $\mathcal{G}$. Instead of analyzing the composite protocol consisting of ρ and π, it suffices to analyze the security of ρ itself in the simpler world with $\mathcal{F}$, the idealized version of π.

Finally, the UC framework is defined atop the underlying computational model of interactive Turing machines (ITMs). In the ITM model, processes pass control from one to another each time a message is sent so that *exactly one process is active at any given time*, and, moreover, *the order of activations is fully determined*. This gives ITMs a clear computational interpretation, which is necessary for the above proofs (in particular, cryptographic reductions) to go through.

1.2 ILC by Example

We make the above more concrete by running through an example of *commitment*, an essential building block in many cryptographic protocols [1]. The idea behind commitment is simple: A *committer* provides a *receiver* with the digital equivalent of a "sealed envelope" containing some value that can later be revealed. The commitment scheme must be *hiding* in the sense that the commitment itself reveals no information about the committed value, and *binding* in the sense that the committer can only open the commitment to a single value. For security under composition, an additional *non-malleability* property is required, which roughly prevents an attacker from using one commitment to derive another related one.

All of these properties are captured at once using an ideal functionality. In Figure 2 (left), we show a simplified ideal functionality for one-time bit commitment, $\mathcal{F}_{\text{COM}}$, as it would appear in the cryptography literature [2]. The functionality simply waits for the committer P to commit to some bit b, notifies the receiver Q that it has taken place, and reveals b to Q upon request by P. Notice that Q never actually sees a commitment to b (only the (Receipt) message), so the three properties hold trivially.

In Figure 2 (right), we implement a simplified version of $\mathcal{F}_{\text{COM}}$ in ILC to highlight some key features of the language. The function fCom takes two channel endpoints as arguments. The first is a write endpoint toQ : Wr Msg (for

$\mathcal{F}_{\text{COM}}$ proceeds as follows, running with committer P and receiver Q.

1. Upon receiving a message (Commit, b) from P, where $b \in \{0, 1\}$, record the value b and send the message (Receipt) to Q. Ignore any subsequent Commit messages.
2. Upon receiving a message (Open) from P, proceed as follows: If some value b was previously recorded, then send the message (Open, b) to Q and halt. Otherwise, halt.

```
1  fCom :: Wr Msg → Rd Msg ⊸ 𝟙
2  let fCom toQ frP =
3    let (!(Commit b), frP) = rd frP in
4      wr Receipt → toQ ;
5      let (!Open, frP) = rd frP in
6        wr (Opened b) → toQ
```

Figure 2. An ideal functionality for a one-time commitment scheme in prose (left) and in ILC (right).

```
1  data Msg = Commit Int
2                 | Open
3           | Opened Int
4           | Receipt
5
6  let fCom toQ frP =
7    let (!(Commit b), frP) = rd frP in
8      wr Receipt → toQ ;
9      let (!Open, frP) = rd frP in
10       wr (Opened b) → toQ
11
12 let exec bit frQ toP =
13   wr (Commit bit) → toP
14     let (!Receipt, frQ) = rd frQ in
15       wr Open → toP;
16       let (!msg, frQ) = rd frQ in msg
17
18 nu (frP, toP),
19    (frQ, toQ) .
20 let bit = 1 in
21 |> fCom toQ frP
22 |> exec bit frQ toP
```

Figure 3. The complete code to execute the $\mathcal{F}_{\text{COM}}$ functionality. This include the type defintion for the messages passed between the functionality and parties P and Q.

sending messages of type Msg to Q), and the second is a read endpoint frP : Rd Msg (for receiving messages of type Msg from P).

Note that that the type definitions requires are missing in this case and that ILC doesn't support directly defining the type signature of functions like in 2. The complete runnable program that implmeents $\mathcal{F}_{\text{COM}}$ is shown in Figure 3.

For the reaminder of this section, unless otherwise, we refer to the code in Figure 3.

Let Expressions The `let` expressions in ILC serve two purposes. One is to define a function (line 6) and the other is to assign a value to a variable (line 20). `let` expressions that assigne a value to a variable must be followed by an expression. Notice in line 20 that the `let` expression is followed by `in`. This denotes that in the code block succieeding the let expression, the variable `bit` is assigned to be 1. Similarly, the same syntax is used in line 14 where the `let` expression assigned the values resulting from the operation `rd frQ`. A point of note in line 14 is

```
1  letrec loop f frS = let (!v, frS) = rd frS in f v; loop f frS
```

Figure 4. The loop function for continuously reading from a channel and passing it into a function.

that the first variable `Receipt` is a type constructor as defined in the line 1. Because this type constructor takes no arguments, there is no variable be assigned. Instead one can think of it as pattern matching where the expression expects a `Receipt` message. The channel endpoing frQ *is* being assigned here as reading from frQ consumes the read endpoint. In general, in order to use a read endpoint again, one *has* to follow this syntax and re-bind the variable frQ to a new read endpoin. The exact details of why `rd` operations behave like this is explained later on when the ILC type system is discussed.

Functions Functions in ILC are created with the let syntax as described above. Parameters are passed in a space separated list. ILC does not support explicity type signatures for functions, instead the type is inferred. Functions return the result of the last expression in their body as their return value. For example in the function `exec` the last expression appears on line 16 after the `in` keyword. Therefore the function `exec` will return whatever `msg` evaluates to. In the case of our commitment example, the function will return `Opened b` where `b` is the bit that was committed to.

Creating Channels In the UC framework, ITMs communicate through input tapes. In ILC we abstract ITM tapes with channels. Channels are created throgh the `nu` operator as shown in Figure 3 line 18. The `nu` operator creates pairs of channel endpoints. In the example frP and toP and two endpoints of the same channel: the first is the read endpoint (that processes can read from) and a write endpoint (that can be written to). Reading and writing to and from channels is shown in lines 13 and 14 respectively. evident below in our discussion of forking.

Forking The code in Figure 3 creates 2 channels on line 18 then calls the two functions `exec` and `fCom` with the fork operator `|>`. The operator is followed by an expression (in this case a function call) and executes the expression as a new process. When the fork operation is used within a function like this

```
1  let foo a b =
2    let x = 3 in
3      |> bar x
4      |> test a b x
```

the return value of this function will be the return value of the *last* forked expression. In this it will be the return value of the `test` function.

Loops The loop construct defined in Figure 4 is very handy and used throughout the rest of this document. The loop accepts two parameters as input: a function `f` and a read endpoint `frS`. The function is recursive and continuously tries to read from `frS` and pass the read value to the funtion `f`. Finally, it recurses. Such a design pattern of iterated reads on a channel endpoint is common and most easily accomplished through simple recursion like this.

Lambdas and Non-top-level Functions Sometimes it's helpful to define internal functions within a function to serve some simple purpose. Consider an ITM whose main purpose is to forward all messages to another ITM. This is useful when implementing a dummy party from the previous section. Similarly, a dummy adversary is an adversary that is completely controlled by the environment. Consider such an adversary in Figure 5. The dummy adversary wants to forward all messages from the read endpoints `frA/B/C` and send them to `D` through its read endpoing `toD`. The function `fwd2D` takes a single parameter as an argument, a channel endpoint `c`. The body of the lambda expression is a loop that forwards all messages from the input channel to the channel `toD`. Finally, the function simple forks processes for each channel is wants to forward.

Syntax Note: you'll notice the first argument in the `fwd2D` function is the open parenthesis: `lam ()`. The reason behind this, which will be clearer when we cover the type sytem is that if the first argument

```
1  let foo k bits crupt toD frB frC =
2    let fwd2D = lam () . lam c . loop (lam m . wr m ⟶ toZ) c in
3      |▷ fwd2D () frA
4      |▷ fwd2D () frB
5      |▷ fwd2D () frC
```

Figure 5. This ITM forwards all messages incoming from `A`, `B` and `C`, and it forwards them all to `D`.

```
1  letrec loop2 table f frS =
2    let (!v, frS) = rd frS in let table = f v table in loop2 table f frS
3
4  let atable k bits frA toA =
5      loop2 [] ( lam x . lam tt .
6                  if (isin x tt) then
7                    let h = lookup x tt in
8                      wr h ⟶ toA; tt
9                  else
10                     let (h,bits) = splitAt k bits in
11                       let tt = (x,h) : tt in
12                         wr () ⟶ toA; tt ) frA
```

Figure 6. A table that stores key-value pairs. If a queried key doesn't exist, it assigns it a random k-bit value, updates the table and sends back the value for the key.

was read endpoint (affine type) the function would also be affine and therefore would be consumed on use.

Mutable Stores Another tricky situation in ILC is needing a shared data structure, like a list or a map. Imagine an ideal functionality that maintains a table of key-value pairs. ILC does not implement mutable stores so to implement a list that multipe processes can modify and access the data structure must exist in a data structure of its own. We implement such a list in Figure 6. The table takes advantage of a new loop construct we've defined, `loop2`. `loop2` takes an additional variable as input which is the data structure to be updates and stored. In our case we simple want a list `[]`. The loop construct reads from a channel like `loop` and passes the input into a function `f` along with the data stucture in question. The difference is that the body of `f` is expected to return an updated version of the data structure. In the the figure, the loop is initialized with an empty list `[]`. When input is received on the channel, the loop checks whether this key exists in the list already. If it does it returns it and returns the input list, `tt`, unchanged. If it doesn't exist it assigns the key a random k-bit value, updates the table `tt` and returns the updated table. The remainder of the `loop2` function saves the new table and recurses.

1.3 ILC Type System Overview

ILC terms have either an unrestricted type, meaning they can be freely copied, or an affine type, meaning they can be used at most once. Affine typing serves a special purpose, namely, to ensure that ILC processes have a determined sequence of activations, as is required in ITMs. This is achieved through the following invariants:

- *Only one process is active at any given time.* Processes implicitly pass around an affine "write token" ⓦ by virtue of where they perform read and write effects. In order for process A to write to process B, process A must first own the write token. Because the write token is unique, at most one process owns the write token ("is active"

```
1  fCom :: Wr Msg ⟶ Rd Msg ⊸ 𝟙
2  let fCom toQ frP =
3    let (!(Commit b), frP) = rd frP in
4      wr Receipt ⟶ toQ ;
5      wr Receipt ⟶ toQ ;
6      let (!Open, frP) = rd frP in
7        wr (Opened b) ⟶ toQ
```

Figure 7. An example of ILC code that would fail to type check. The program writes to the write endponig toQ in line 4 but then attempts to write again in line 5 without having re-acquiring the write token through a "rd" operation. This code example will fail to type check as the invariants descibed earlier would be violated.

All types	$U, V ::= A \mid X$	Syntax labels	$\ell ::= \pi \mid \mathsf{w}$
Sendable types	$S, T ::= \mathbb{1} \mid S \times T \mid S + T$	Multiplicity labels	$\pi ::= 1 \mid \infty$
Unrestricted types	$A, B ::= S \mid \mathsf{Wr}\, S \mid A \times B \mid A + B \mid A \to_{\infty \mid \mathsf{w}} U$	Unrestricted typings	$\Gamma ::= \cdot \mid \Gamma, x : A$
Affine types	$X, Y ::= !\, A \mid \mathsf{Rd}\, S \mid X \otimes Y \mid X \oplus Y \mid X \to_1 U$	Affine typings	$\Delta ::= \cdot \mid \Delta, x : X \mid \Delta, ⓦ$

Values	$v ::= () \mid (v_1, v_2)_\ell \mid \mathsf{inj}^1_\ell(v) \mid \mathsf{inj}^2_\ell(v) \mid \lambda_\ell\, x.\, e \mid c \mid !\, v$
Channel endpoints	$c ::= \mathsf{Read}(d) \mid \mathsf{Write}(d)$
Channel names	$d ::= \cdots$
Expressions	$e ::= x \mid () \mid (e_1, e_2)_\ell \mid \mathsf{inj}^i_\ell(e) \mid \mathsf{split}_\ell(e_1, x_1.x_2.e_2) \mid \mathsf{case}_\ell(e, x_1.e_1, x_2.e_2)$ $\mid \lambda_\ell\, x.\, e \mid (e_1\, e_2)_\ell \mid \mathsf{fix}_\ell(x.e) \mid \mathsf{let}_\pi(e_1, x.e_2) \mid !\, e \mid ¡\, e$ $\mid nu(x_1, x_2).\, e \mid \mathsf{wr}(e_1, e_2) \mid \mathsf{rd}(e_1, x.e_2) \mid \mathsf{ch}(e_1, x_1.e_3, e_2, x_2.e_4) \mid e_1 \mid\!\triangleright\, e_2$

Figure 8. ILC Syntax.

or "can write") at any given time. When process B reads the message from A, process B earns the write token, thereby conserving its uniqueness and now allowing process B to write to some other process.

- *The order of activations is deterministic.* Each channel (or "tape" in ITM parlance) has a read endpoint and a write endpoint. The read endpoint is an affine resource, and so it is owned by at most one process. This ensures that each write operation corresponds to a single, unique read operation.

Intuitively, the first invariant rules out the possibility of write nondeterminism. Consider the case in Figure 7 in which two processes are trying to execute writes in parallel. Allowing such code would violate the first invariant above. This code does not type check typecheck, since the affine write token belongs to at most one process. One might justifiably wonder why write endpoints are unrestricted and read endpoints are affine. Note that if two processes are trying to write in parallel, the two write endpoints need not be the same, so making write endpoints affine would not help our case in eliminating write nondeterminism.

Dually, the second invariant rules out the possibility of read nondeterminism. Consider the case in which two processes A and B are listening on the same read endpoint. If a process C writes on the corresponding write endpoint, which of A or B (or both) gets activation? If only one of them is activated, then we have a source of nondeterminism. If both are activated, now A and B both own write tokens, violating its affinity. In any case, this does not typecheck since read endpoints are affine resources, making it impossible for two processes A and B to listen on the same read endpoint. Together, these invariants ensure that processes have a determined sequence of activations as desired.

1.4 ILC Syntax

The syntax of ILC is given in Figure 8. Types (written U, V) are bifurcated into unrestricted types (written A, B) and affine types (written X, Y).

A subset of the unrestricted types are sendable types (written S, T), i.e., the types of values that can be sent over channels. This restriction ensures that channels model network channels, which send only data. The sendable types include unit ($\mathbb{1}$), products ($S \times T$), and sums ($S + T$).

The unrestricted types include the sendable types, write endpoint types ($\mathsf{Wr}\, S$), products ($A \times B$), sums ($A + B$), arrows ($A \to_\infty U$ or simply $A \to U$), and write arrows ($A \to_\mathsf{w} U$). Write arrows specify unrestricted abstractions for which the write token can be moved into the affine context of the abstraction body during β-reduction.

The affine types include bang types ($!\, A$), read endpoint types ($\mathsf{Rd}\, S$), products ($X \otimes Y$), sums ($X \oplus Y$), and arrows ($X \to_1 U$ or simply $X \multimap U$). Notice that the write token ⓦ lives in the affine context, though it cannot be bound to any variable. Instead, it flows around implicitly by virtue of where read and write effects are performed.

For concision, certain syntactic forms are parameterized by a multiplicity π to distinguish between the unrestricted (∞) and affine (1) counterparts; other syntactic forms are parameterized by a syntax label ℓ, which includes the multiplicity labels and the write label w (related to write effects). On introduction and elimination forms for functions (abstraction, application, and fixed points), the label w denotes variants that move around the write token as explained above. On introduction and elimination forms for products and sums, the label w denotes the sendable variants.

Values in ILC (written v) include unit, pairs, sums, lambda expressions, channel endpoints (written c), and banged values. We distinguish between the names of channel endpoints—$\mathsf{Read}(d)$ and $\mathsf{Write}(d)$—and the channel d itself that binds them. ILC supports a fairly standard feature set of expressions. Bang-typed values have introduction form $!\, e$ and elimination form ¡ e. The more interesting expressions are those related to communication and concurrency:

- *Restriction:* $nu(x_1, x_2).\ e$ binds a read endpoint x_1 and a corresponding write endpoint x_2 in e. The syntax looks like this

```
1        nu (frP, toP),
2           (frQ, toQ) .
3          let x = 5 in x
```

The above expression will evaluate to the 5.

- *Write:* $\mathsf{wr}(e_1, e_2)$ sends the value that e_1 evaluates to on the write endpoint that e_2 evaluates to.

```
1        wr (Opened 5) → toQ
```

- *Read:* $\mathsf{rd}(e_1, x.e_2)$ reads a value from the read endpoint that e_1 evaluates to and binds the value-endpoint pair as x in e_2.

```
1        let (!msg, frP) = rd frP in msg
```

In this expression `msg` is the value and the endpoint `frP` is re-bound to a new endpoint. The expression above evaluates to `msg` as that is the last expression in the code block.

- *Choice:* $\mathsf{ch}(e_1, x_1.e_3, e_2, x_2.e_4)$ allows a process to continue as either e_3 or e_4 based on some initial read event on one of the read endpoints that e_1 and e_2 evaluate to. The value read over the channel and the two read endpoints are rebound in a 3-tuple as x_1 in e_3 or x_2 in e_4. Here, we show only binary choice, but it can be generalized to the n-ary case.
- *Fork:* $e_1 \mid\!\triangleright\ e_2$ spawns a child process e_1 and continues as e_2.

2 Commitment Example

In order to concretize the syntax descriptions above we will work through an example protocol for cryptographic commitment in UC. Along with this document we also have present a code-file in ILC that implements the complete commiement UC example covered in this section. The code can be found here. The idea behind commitment is simple: A *committer* provides a *receiver* with the digital equivalent of a "sealed envelope" containing some value that can later be revealed. The commitment scheme must be *hiding* in the sense that the commitment itself reveals no information about the committed value, and *binding* in the sense that the committer can only open the commitment to a single value. For security under composition, an additional *non-malleability* property is required, which roughly prevents an attacker from using one commitment to derive another related one.

```
data Msg = Commit Int
                 | Open
         | Opened Int
         | Receipt

let fCom toQ frP =
  let (!(Commit b), frP) = rd frP in
    wr Receipt ⟶ toQ ;
    let (!Open, frP) = rd frP in
      wr (Opened b) ⟶ toQ
```

Figure 9. The $\mathcal{F}_{\text{COM}}$ functionality in ILC

```
data Crupt = CruptP | CruptQ | CruptNone

data Msg' = GetCRS
          | PublicStrings [Bit] [Bit] [Bit]
        | Commit' [Bit]
        | Opened' [Bit]
        | Open' Int [Bit]

data Msg = Commit Int
           | Open
        | Opened Int
        | Receipt
```

Figure 10. Defining the types used in the UC Commitment Example

All of these properties are captured at once using an ideal functionality. In Figure 2 (left), we show a simplified ideal functionality for one-time bit commitment, $\mathcal{F}_{\text{COM}}$, as it would appear in the cryptography literature [2]. The functionality simply waits for the committer P to commit to some bit b, notifies the receiver Q that it has taken place, and reveals b to Q upon request by P. Notice that Q never actually sees a commitment to b (only the (Receipt) message), so the three properties hold trivially.

In Figure 2 (right), we implement a simplified version of $\mathcal{F}_{\text{COM}}$ in ILC to highlight some key features of the language. The function fCom takes two channel endpoints as arguments. The first is a write endpoint toQ : Wr Msg (for sending messages of type Msg to Q), and the second is a read endpoint frP : Rd Msg (for receiving messages of type Msg from P). At a high level, it should be clear how the communication pattern in fCom follows that in $\mathcal{F}_{\text{COM}}$, but there are a few details that require further explanation. These details are better explained in the context of ILC's type system, which we give a quick tour of next.

The first step is to define some simple types to go along with this protocol example. The first line denotes the corruption mode of the protocol: either P is corrupt or Q is corrupt but not both. The next two define message types. The first `Msg'` are messages used by the real world protocl and the second is used in the ideal world. Finally, the `Z2A` definition is used to denote where messages are being sent.

2.1 Ideal Functionality

The ideal functionality is much simpler than the real world protocl so we begin here. Figure 11 shows the ILC code for the commitment functionality. The first three parameters the functionality accepts are common to ever

```
1 let fCom k bits crupt toP toQ toA frP frQ frA =
2   let (!(Commit b), frP) = rd frP in
3     wr Receipt → toQ ;
4     let (!Open, frP) = rd frP in
5       wr (Opened b) → toQ
```

Figure 11. Commitment ideal functionality in ILC

$\mathcal{F}_{\text{RO}}$ proceeds as follows parameterized by a security parameter k , parties $P_1, ..., P_n$ and an adversary $\mathcal{S}$:

1. $\mathcal{F}_{\text{RO}}$ stores a list L (initially empty) of pairs of bitstrings.
2. When activated by P_i or $\mathcal{S}$ with a value m:
   - If there is a pair (m, h) in L, send h to the activating machine P_i or $\mathcal{S}$.
   - If no such pair, sample $h \in \{0,1\}^k$, store the pair (m, h) in L and send h to activating machine.

Figure 12. The Random Oracal (RO) functionality. It mantains a list of key-value pairs and implements an idealized hash function. Hash values are chosen at random at the first query of some pre-image.

functionality (every ITM) in UC: the security parameter `k`, a sequence of random bits `bits`, and a corruption model `crupt`. How the corrupt model is defined will changed from functionality to functionality. For example, the corruption model in this example, shown in Figure **??**, only considers corruption of one of two parties[1].

In UC, functionalities have interfaces to protocol parties and the adversary. Hence the two channels to parties P and Q (represented by endpoints frP, toP, frQ and toQ) and one channel to the adversary (frA, toA) .

The code of the functionality is quite simpe. It first waits for a "Commit b" messages from P (where b is the bit being committed to). It then notifies Q of the commitment. Finally, when P wants to open the commitment, Q is sent the bit b.

2.2 Commitment Protocol

In our commitment protocol we are working in the random oracle (RO) model. The purpose of the RO model is to capture an idealized version of a hash function. The functionality $\mathcal{F}_{\text{RO}}$ is black box that randonly assigned a "hash" to any query given. All hashes are stored in a table for lookup when the same pre-image is requested twice. We describe the functionality in Figure 12.

Above we described the commitment functionality. Here we define an implementation of the commitment protocol by Fischlin and Canetti [**?**]. The protocol described in this work relies on a trusted setup called the Common Reference String (CRS). A CRS is a common string that is drawn from some public distribution that all parties have access to. It is implemented via the functionality $\mathcal{F}_{\text{CRS}}$ shown in Figure **??**.

2.3 The Real World

The real world consists of an implementation run by the protocol parties that realizes the ideal functionality and an adversary. The strongest adversary for UC emuation is the dummy adversary which we describe first.

Dummy Adversary The dummy adversary just relays messages from the environment. If the enviroment is tryin to communicate with a corrupted party, it sends a message to the dummy adversary which forwards it to the intended party. The code for the dummy adversary is shown in Figure 13. Notice that in our case $\mathcal{F}_{\text{RO}}$ can be queries by the adversary as well so the adverasry also relays messages bound for the functionality. The adversary cannot, however, give input to honest parties thus it rejects an attempt by the environent to give input to an honest party.

On the receiving end, the adversary just forwards all messages to the environment regardless of its origin. For example if the adversary is told to query the random oracle, it sends the environment the functionality's reply.

[1] A different protocol, such as a byzantine broadcast, requires a corruption model capable of denoting corruptions over some larger set of parties

```
let dummyA k bits crupt toZ toF toP toQ frZ frF frP frQ =
  let fwd2Z = lam () . lam c . loop (lam m . wr (X2Z m) > toZ) c in
    loop (lam x . match x with
      | A2F m ⇒ wr m ⟶ toF
      | A2P m ⇒ if crupt == CruptP
                then wr m ⟶ toP
                else if crupt == CruptQ
                then wr m ⟶ toQ
                else error "Can't do that") frZ
    |▷ fwd2Z () frF
    |▷ fwd2Z () frP
    |▷ fwd2Z () frQ
```

Figure 13. The dummy adversary

```
let committer k bits crupt toZ toF toQ frZ frF frQ =
  let (!(Commit b), frZ) = rd frZ in
    let (nonce, bits) = splitAt k bits in
    wr (P2RO (nonce, b)) ⟶ toF;
      let (!h, frF) = rd frF in
        wr (Commit' h) ⟶ toQ;
        let (!Open, frZ) = rd frZ in
          wr (Open' (nonce,b)) ⟶ toQ; ()
```

Figure 14. The committers protocol for the $\mathcal{F}_{\text{RO}}$ − *hybrid* world. The committer uses the random oracle as an ideal hash function in order to hide his commitment and reveal the pre-image to the receiver at a later time

Commitment Protocol We describe the protocol in ILC for both the committer and the receiver. The committer's protocol is given in Figure 14. When activated by the environment, the committer generates some random nonce value and queries the random oracle for a hash value for the pre-image (nonce, b). The `splitAt` function is a helper function provided in the ILC codebase to split a list. It then sends the hash value, h, to the counter party Q. Finally when it's time to open, it reveals the preimage (nonce, b) to Q which queries the random oracle to obtain the same hash value, h.

Similarly, the receiver's code is even simpler. It is shown in Figure 15. The receiver simply accepts the commitment from the committer P, notifies the environment and then waits to receive the pre-image of the commitment and check it against the random oracle.

[Syntax Note]: Notice at the end of the receiver code, the last expression is `()`. This is because the operations such as `wr` and `print` do not return anything meaningful in themselves, and, since the function (a `let` expression) must end with another expression, a `()` is placed at the end.

Recall from the discussion above that we require the commitment scheme to be *hiding* and *binding*. It is clear to see that the random oracle provides both:

- *hiding*: the random oracle randomly sample a hash value from the input ranom stream therefore can reveal nothing about the pre-image queried.
- *binding*: similarly, the random oracle maintains an internal table for each (pre-image,hash) pair ensuring a one-to-one mapping. This implies that only one pre-image (the one used) can be used to open a commitment.

```
1  let receiver k bits crupt toZ toF toP frZ frF frP =
2    let (!(Commit' h), frP) = rd frP in
3      wr Committed ⟶ toZ;
4      let (!(Open' m), frP) = rd frP in
5        let (nonce, b) = m in
6          wr (P2RO (nonce, b)) ⟶ toF;
7          let (!hh, frF) = rd frF in
8            wr (Opened b) ⟶ toZ; ()
```

Figure 15. The receiver only reacts to the committers messages and notifies the environment of a commitment attmepts. When the commitment is revealed it checks the random oracle to ensure the pre-image matches the commitment hash sent by the committer.

```
1  letrec fwd toR frS =
2    let (!msg, frS) = rd frS in wr msg ⟶ toR; fwd toR frS
3
4  letrec cruptfwd toP toF frS =
5    let (!msg, frS) = rd frS in
6      match msg with
7      | P2F m => wr m ⟶ toF; cruptfwd toP toF frS
8      | P2Q m => wr m ⟶ toP; cruptfwd toP toF frS
9
10 let corruptOrNot p k bits iscrupt toZ toF toA toQ frZ frF frA frQ=
11   if iscrupt then ()
12     |▷ fwd toA frF
13     |▷ fwd toA frQ
14     |▷ cruptfwd toQ toF frA
15   else
16     p k bits CruptNone toZ toF toQ frZ frF frQ
```

Figure 16. Wrappers that encapsulate the parties P and Q and determine their action based on which of the two is corrupt.

Handling Corruptions Corruptions in UC means that a party is under complete control by the adverasry. In our case, as we're dealing with a dummy adversary, the environment controls controls corrupted parties. For simplicity and ease of use we define corrupt parties to behave just like dummy parties in in the ideal world with one difference. Instead of relaying information to and from the environment and functionalities/other parties, the corrupt parties receive input from the adversary and relay output to the adverasry. We capture this notion through the use of a party wrapper `corruptOrNot` that forwards messages to the adverasry if the party is corrupt. The code for `corruptOrNot` is shown in Figure 16. The wrapper is aided by two functions `fwd` and `cruptfwd`.

- `fwd`: a recursive function that does what it's name suggests. It reads from the input read endpoint and forwards the message to the input write endpoint.
- `cruptfwd`: for messages send by the adversary to a corrupt party, the adversary can tell the party to write relay the message to a functionality or to the Q. This function pattern matches the message and forwards it on the right channel.

```
1  fCrs :: ∀ a ... . Nat → [Bit] → Crupt → ··· a
2  let fCrs k bits crupt toP toQ toA frP frQ frA =
3    let (σ, bits) = sample (4*k) bits in
4    let (r0, bits) = sample k bits in
5    let (r1, bits) = sample k bits in
6    let pk0 = kgen k r0 in
7    let pk1 = kgen k r1 in
8    let pub = PublicStrings σ pk0 pk1 in
9    let replyCrs to fr = loop (λ _ . wr pub → to) fr in
10         replyCrs toP frP
11     |▷ replyCrs toQ frQ
12     |▷ replyCrs toA frA
```

Random Oracle The random oracle functionality maintains a table that stores all key-value pairs generated during the execution of the protocol. This is trickier than it may seem in ILC as ILC does not have mutable stores.

3 ILC Type System

A Selection of Typing Rules. To see these invariants in action, we walk through the typing rules for fork, write, and read expressions. We read the typing judgement $\Delta; \Gamma \vdash e : U$ as "under affine context Δ and unrestricted context Γ, expression e has type U." The metavariables U and V range over all types (both unrestricted and affine).

The fork expression $e_1 \mid\triangleright e_2$ spawns a child process e_1 and continues as e_2.

$$\frac{\Delta_1; \Gamma \vdash e_1 : U \qquad \Delta_2; \Gamma \vdash e_2 : V}{\Delta_1, \Delta_2; \Gamma \vdash e_1 \mid\triangleright e_2 : V}\ \text{fork}$$

Its typing rule says that if we can partition the affine context as Δ_1, Δ_2 such that e_1 has type U under contexts $\Delta_1; \Gamma$ and e_2 has type V under contexts $\Delta_2; \Gamma$, then the expression has type V. Notice that affine resources (e.g., read endpoints and the write token) must be split between the child process and the parent process, thereby preventing their duplication.

The write expression $\mathsf{wr}(e_1, e_2)$ sends the value that e_1 evaluates to on the write endpoint that e_2 evaluates to. One thing to mention is that only values of a sendable type (ranged over by S) can be sent over channels (more on this later).

$$\frac{\Delta_1; \Gamma \vdash e_1 : S \qquad \Delta_2; \Gamma \vdash e_2 : \mathsf{Wr}\, S}{\Delta_1, \Delta_2, \textcircled{w}; \Gamma \vdash \mathsf{wr}(e_1, e_2) : \mathbb{1}}\ \text{wr}$$

Its typing rule says that if we own the write token and we can partition the affine context as Δ_1, Δ_2 such that e_1 has type S under contexts $\Delta_1; \Gamma$ and e_2 evaluates to a write endpoint (of type $\mathsf{Wr}\, S$) under contexts $\Delta_2; \Gamma$, then the expression has type $\mathbb{1}$ (unit). Notice that typing a write expression spends the write token, and so it cannot execute another write until it gets "reactivated" by reading from some other process.

The read expression $\mathsf{rd}(e_1, x.e_2)$ reads a value on the read endpoint that e_1 evaluates to and binds the value-endpoint pair as x in the affine context of e_2. Rebinding the read endpoint allows it to be reused.

$$\frac{\textcircled{w} \notin \Delta_2 \qquad \Delta_1; \Gamma \vdash e_1 : \mathsf{Rd}\, S \qquad \Delta_2, \textcircled{w}, x : !\, S \otimes \mathsf{Rd}\, S; \Gamma \vdash e_2 : U}{\Delta_1, \Delta_2; \Gamma \vdash \mathsf{rd}(e_1, x.e_2) : U}\ \text{rd}$$

Its typing rule says that if we can partition the affine context as Δ_1, Δ_2 such that e_1 evaluates to a read endpoint (of type $\mathsf{Rd}\, S$) under contexts $\Delta_1; \Gamma$, and e_2 has type U under contexts $\Delta_2, \textcircled{w}, x : !\, S \otimes \mathsf{Rd}\, S; \Gamma$, then the expression has type U.

There are a few things to unpack here. First, we explain the affine product type $! S \otimes \mathsf{Rd}\, S$. Since sendable values are unrestricted and read endpoints are affine, the value read on the channel is wrapped in a ! operator (pronounced "bang") so that it can be placed in an affine pair. Next, observe that $\textcircled{w}$ is available in the body e_2 of the read expression (i.e., it is conserved), but only under the condition that it is not already in the affine context Δ_2 (otherwise, a process could arbitrarily mint write tokens, violating its affinity).

Revisiting fCom. Having gone through several typing rules, we now revisit fCom from Figure 2 (right). In particular, we should convince ourselves that fCom respects the invariants of the type system.

The type signature tells us that toQ : Wr Msg is unrestricted ($\multimap$ is the type connective for unrestricted arrows) and frP : Rd Msg is affine ($\multimap$ is the type connective for affine arrows). As we mentioned, write endpoints *are not* affine, since this restriction does not help in preventing write nondeterminism; read endpoints *are* affine, which does prevent read nondeterminism. To see that frP is being used affinely, notice that it is rebound when deconstructing the value-endpoint pair from each read operation, so it can be used again.

To see that the write token is being passed around appropriately, notice that the read and write effects are interleaved. Before each read operation, fCom does not own the write token: In the first read operation, only frP : RdMsg is present in the affine context; in the second read operation, the first write operation has already spent the write token. Before each write operation, fCom does own the write token: Each is preceded by a read operation.

References

[1] Gilles Brassard, David Chaum, and Claude Crépeau. 1988. Minimum disclosure proofs of knowledge. *J. Comput. System Sci.* 37, 2 (1988), 156–189.

[2] Ran Canetti and Marc Fischlin. 2001. Universally composable commitments. In *Annual International Cryptology Conference*. Springer, 19–40.

[3] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 218–229.