ILC and SaUCy: A Calculus for Composable, Computational Cryptography

Kevin Liao
University of Illinois
Urbana-Champaign
kliao6@illinois.edu

Matthew Hammer
University of Colorado Boulder
matthew.hammer@colorado.edu

Andrew Miller
University of Illinois
Urbana-Champaign
soc1024@illinois.edu

Abstract

The universal composability (UC) framework is the established standard for analyzing cryptographic protocols in a modular way, such that security is preserved under concurrent composition with arbitrary other protocols. However, although UC is widely used for on-paper proofs, prior attempts at systemizing it have fallen short, either by using a symbolic model (thereby ruling out computational reduction proofs), or by limiting its expressiveness.

In this paper, we lay the groundwork for building a concrete, executable implementation of the UC framework. Our main contribution is a process calculus, dubbed the Interactive Lambda Calculus (ILC). ILC faithfully captures the computational model underlying UC—interactive Turing machines (ITMs)—by adapting ITMs to a subset of the π-calculus through its type system. In other words, *well-typed ILC programs are expressible as ITMs.* In turn, ILC's strong confluence property enables reasoning about cryptographic security reductions. We use ILC to develop our implementation of UC, called SaUCy. To demonstrate the use of SaUCy, we work through an instantiation of UC-secure commitments from the literature, define a composition operator and proving its associated theorem, as well as examine a subtle definitional issue involving race conditions.

1 Introduction

In cryptography, a proof of security in the simulation-based universal composability (UC) framework is considered the gold standard for demonstrating that a protocol "does its job securely" [12]. In particular, a UC-secure protocol enjoys the strongest notion of compositionality—it maintains all security properties even when run *concurrently* with arbitrary other protocol instances. This is in contrast with weaker property-based notions that only guarantee security in a standalone setting [30] or under sequential composition [20]. Thus, the benefit of using UC is modularity—it supports analyzing complex protocols by composing simpler building blocks. However, the cost of UC is that security proofs tend to be quite complicated. We believe that applying a PL-style of systemization to UC can help simplify its use, bring new clarity, and provide useful tooling. We envision a future where modularity of cryptographic protocol composition translates to modular implementation as well.

2018.

Reviewing prior efforts of applying PL techniques to cryptography, we find they run up against challenges when importing the existing body of UC theory. Either they do not support computational reasoning (which considers issues of probability and computational complexity) [9], do not support message-passing concurrency for distributed protocols [4], or are too expressive (allow for expressing nondeterminism with no computational interpretation) [2].

Our observation is that these approaches diverge from UC at a low level: UC is defined atop the underlying computational model of *interactive Turing machines* (ITMs). The significance of ITMs is that they have a clear computational interpretation, so it is straightforward to relate execution traces to a probabilistic polynomial time computation, as is necessary for cryptographic reduction proofs. The presence of (non-probabilistic) nondeterminism in alternative process calculi would frustrate such reduction proofs. ITMs avoid this issue by having a deterministic (modulo random coin tosses), single-threaded execution semantics. That is, 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, this order of activations is fully determined.

In this paper, we take up the challenge of faithfully capturing this sequentiality by adapting ITMs to a subset of the π-calculus through a type discipline. We call this process calculus the Interactive Lambda Calculus (ILC), and we use it to build a concrete, executable implementation of the UC framework, dubbed SaUCy.

1.1 Interactive Lambda Calculus

Where have existing concurrency models fallen short with respect to ITMs? On the one hand, some existing process calculi, such as the standard π-calculus [34] or its cryptography-oriented variants [1, 2, 29], are not a good fit to ITMs, since they permit non-confluent reductions by design (i.e., non-probabilistic nondeterminism). On the other hand, the various calculi that do enjoy confluence are restrictive, only allowing for fixed or two-party communications [9, 19, 23].

ILC fills this gap by adapting ITMs to a subset of the π-calculus through its type system. In other words, *well-typed ILC programs are expressible as ITMs.* To give a preview of the type system, it restricts the dataflow of read channels (via affine typing) and restricts the composition of write effects (forbidding write-mode parallel processes), with the net effect being confluence. Therefore, the only nondeterminism in an ILC program is due to random coin tosses taken by processes, which have a well-defined distribution. Additionally, any apparent concurrency hazards, such as adversarial scheduling of messages in an asynchronous network, are due to an explicit adversary process rather than uncertainty built into the model itself.

1.2 Contributions

To summarize, our main contributions are these:

- We design a foundational calculus for the purpose of systemizing UC called the Interactive Lambda Calculus, which exhibits confluence and is a faithful abstraction of ITMs.
- We use ILC to build a concrete, executable implementation of the UC framework called SaUCy.
- We then use SaUCy to port over a sampling of theory from UC literature, including a composition theorem, an instantiation proof of UC commitments [14], and an examination of a subtle definitional issue involving reentrant concurrency [11].

2 Overview

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

2.1 Background on Universal Composability

Security proofs in the UC framework follow the real/ideal paradigm [20]. 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*.

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.

2.2 ILC by Example

We make the above more concrete by running through an example of *commitment*, an essential building block in many cryptographic protocols [10]. 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 1 (left), we show $\mathcal{F}_{\text{COM}}$, an ideal functionality for one-time bit commitment, as it would appear in the cryptography literature [14]. 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 1 (right), we define a simplified version of $\mathcal{F}_{\text{COM}}$ in ILC to highlight some key features of the language, such as its affine and modal type system. Keep in mind that the overarching feature of ILC is confluence, so many of our design choices are for achieving this.

Affine types. Let us first examine the function signature of fCom, which (glossing over some details) takes as arguments two channels, a read channel frP (reads message from P) and a write channel toQ (writes messages to Q), and returns Unit. There are a number of things to unpack here:

- The signature consists of affine arrows $\multimap$ (or "lollipops"), which describe the types of functions that consume their argument at most once.
- Both lollipops and intuitionistic (unrestricted) arrows carry the *modal type* (or *mode*) of their function bodies. A mode can have one of three values: either write W, read R, or value V. In the example, the left lollipop has mode V, which we elide, and the right lollipop has mode R. We have more to say about modes later.
- Read channels are affinely typed to protect them from duplication.

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

```
fCom :: Rd Msg ⊸ !(Wr Msg) ⊸_R !Unit
let fCom frP !toQ =
  let ⟨!(Commit b), frP⟩ = rd frP in
    wr Receipt → toQ ;
    let ⟨!Open, frP⟩ = rd frP in
      !(wr (Opened b) → toQ)
```

Figure 1. An ideal functionality for a one-time commitment scheme in prose (left) and in ILC (right).

- Write channels and the unit value are intuitionistically typed. Hence, their types require the ! operator (pronounced "bang!") to be lifted into an affine type.

ILC achieves the above with a two-kind system. That is, types are bifurcated into a kind for intuitionistic types and a kind for affine types.

The main takeaway of the affine type system is this: It restricts the dataflow of read channels so that no confusion arises at the receiving ends of communications.

Modal types. ILC expressions are typed with one of three modes: W (write mode), R (read mode), or V (value mode). Modes can be composed either sequentially or in parallel to yield new modes.

To give the high-level idea, suppose expressions e_1 and e_2 have modes m_1 and m_2, respectively. Sequential mode composition shows up in the obvious case of sequentially evaluating the two expressions, so the mode of $e_1 \,;\, e_2$ is derived as $m1 \,;\, m2 \Rightarrow m3$.[1] Parallel mode composition shows in the case of forking a child process, so the mode of $e_1 \mathrel{|\!\triangleright} e_2$ (fork a process e_1 and continue as e_2) is derived as $m_1 \parallel m_2 \Rightarrow m_3$. The rules for mode composition are discussed in Section 3.

The main takeaway of the modal type system is this: Parallel composition of write mode processes is *not allowed*, i.e., $\mathsf{W} \parallel \mathsf{W} \Rightarrow p$ is not derivable for any mode p. This ensures that no confusion arises as to which process is currently writing, or put another way, which process is currently active.

Putting it all together. The function body of fCom closely follows $\mathcal{F}_{\text{COM}}$, but there are several points worth mentioning.

First, we introduce the letrd and wr typing rules. Typing judgments have the form $\Delta; \Gamma \vdash e : A \rhd m$, where Δ is an affine typing context, Γ is an intuitionistic typing context, and m is a mode.

$$\frac{\begin{array}{c}\Delta_1; \Gamma \vdash e_1 : \mathsf{Rd}\, A \\ \Delta_2, x_1 : {!}\,A, x_2 : \mathsf{Rd}\, A; \Gamma \vdash e_2 : B \rhd m\end{array}}{\Delta_1, \Delta_2; \Gamma \vdash \mathsf{let}_\pi \ \langle x_1, x_2 \rangle = \mathsf{rd}(e_1) \ \mathsf{in} \ e_2 : B \rhd \mathsf{R}} \ \text{letrd}$$

The letrd rule says that if we can partition the affine context Δ as Δ_1, Δ_2 such that e_1 has type $\mathsf{Rd}\, A$ and mode V (elided) under contexts $\Delta_1; \Gamma$, and e_2 has type B and mode m under contexts $\Delta_2, x_1 : {!}\,A, x_2 : \mathsf{Rd}\, A; \Gamma$, then the full expression has type B and mode R. Notice several things:

- Reading on a channel of type $\mathsf{Rd}\, A$ produces an affine pair (or "tensor") of type ${!}\,A \otimes \mathsf{Rd}\, A$.
- ILC only allows intuitionistically typed values to be sent over channels (A ranges over intuitionistic types), so the value read on the channel is lifted into the first element of the tensor as an affine type using the bang! operator.
- The read channel itself is rebound as the second tensor element, so that it may be used again.
- Pattern matching with the ! operator unpacks affine values, so the value bound to b in the first read from frP has an intuitionistic type.

$$\frac{\Delta_1; \Gamma \vdash e_1 : A \qquad \Delta_2; \Gamma \vdash e_2 : \mathsf{Wr}\, A}{\Delta_1, \Delta_2; \Gamma \vdash \mathsf{wr}(e_1, e_2) : \mathbb{1} \rhd \mathsf{W}} \ \text{wr}$$

The wr rule says that if we can partition the affine context Δ as Δ_1, Δ_2 such that e_1 has type A and mode V, and e_2 has a type $\mathsf{Wr}\, A$ and mode V, then the full expression has type $\mathbb{1}$ and mode W.

Revisiting fCom, observe that the two letrd expressions are composed sequentially, so the mode of the entire function body is derived as $\mathsf{R} \,;\, \mathsf{R} \Rightarrow \mathsf{R}$ according to our mode composition rules. This is reflected in the fact that the right lollipop in the function signature carries a mode R. Finally, because wr expressions return (), we wrap it inside a bang! to lift its type into the affine type !Unit.

3 Interactive Lambda Calculus

We now present the Interactive Lambda Calculus in full, formalizing its syntax, static semantics, and dynamic semantics.

3.1 Syntax

The syntax of ILC given in Figure 2 consists of types, modes, and expressions.

Types are bifurcated into two kinds: one for intuitionistic types (written A, B) and one for affine types (written X, Y). The standard intuitionistic types include unit ($\mathbb{1}$), products ($A \times_\pi B$), sums ($A + B$), and references ($\mathsf{Ref}\, A$). The more interesting ones are write channels ($\mathsf{Wr}\, A$) and arrows ($A \rightarrow_m B$), which importantly carry a mode. Affine types include read channels ($\mathsf{Rd}\, A$), bang! types (${!}\,A$), tensors ($X \otimes Y$), and lollipops ($X \multimap_m Y$).

[1]The expression $e_1 \,;\, e_2$ is syntactic sugar for $\mathsf{let}_\pi \ () = e_1 \ \mathsf{in} \ e_2$.

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

$$\begin{aligned} \text{Expressions} \quad e ::=\ & x \mid () \mid (e_1, e_2)_{\pi} \mid \mathsf{inj}^{i}_{\pi}(e) \mid \mathsf{ref}(e) \mid \mathsf{split}_{\pi}(e_1, x_1.x_2.e_2) \mid \mathsf{case}_{\pi}(e, x_1.e_1, x_2.e_2) \\ & \mid \mathsf{get}(e) \mid \mathsf{set}(e_1, e_2) \mid \mathsf{fix}_{\ell}(x.e) \mid \mathsf{let}_{\pi}\ x = e_1 \ \mathsf{in}\ e_2 \mid !\, e \mid ¡\, e \\ & \mid \lambda_{\pi} x.\, e \mid (e_1\ e_2)_{\ell} \mid \nu(x_1, x_2).\, e \mid \mathsf{wr}(e_1, e_2) \mid \mathsf{rd}(e) \mid e_1 \mathrel{|\!\triangleright} e_2 \mid e_1 \oplus e_2 \end{aligned}$$

Figure 2. Syntax of ILC.

Modes consist of W (write mode), R (read mode), and V (value mode). The rules for sequential and parallel composition of modes are given in Figure **??**.

The intuitionistic portion of ILC supports a fairly standard feature set. The more interesting expressions are those related to affinity and message-passing concurrency. Bang! types have introduction form $!\, e$ and elimination form let! $x = e_1$ in e_2. Lollipops have introduction form $\Lambda x.\, e$ and elimination form $\langle e_1\ e_2 \rangle$. Tensors have introduction form $\langle e_1, e_2 \rangle$ and elimination form $\mathsf{split}\langle e_1, x_1.x_2.e_2 \rangle$. The expressions for message-passing concurrency are these:

- *Restriction:* $\nu(x_1, x_2).\, e$ binds a read channel x_1 and a corresponding write channel x_2 in e.
- *Write:* $\mathsf{wr}(e_1, e_2)$ sends the result of evaluating e_1 on the write channel result of evaluating e_2.
- *Read:* $\mathsf{let}_{\pi}\ \langle x_1, x_2 \rangle = \mathsf{rd}(e_1)$ in e_2 reads a value from the read channel result of evaluating e_1, binding the value as x_1 and the read channel as x_2 in e_2.
- *Fork:* $e_1 \mathrel{|\!\triangleright} e_2$ forks a child process e_1 and continues as e_2.
- *Choice:* $e_1 \oplus e_2$ allows a process to continue as either e_1 or e_2 based on some initial read event in each of the processes.

3.2 Static Semantics

The type system of ILC is given in Figure 3. The intuitionistic typing judgment $\Delta; \Gamma \vdash e : A \triangleright m$ should be read as follows:

> Under affine context Δ and intuitionistic context Γ, the expression e has intuitionistic type A and mode m.

(The affine typing judgment follows similarly.) Importantly, the type system maintains these invariants:

- *No duplication of read channel ends.* Each channel (or "tape" in ITM parlance) has a read end and a write end. The read end of the channel is protected against duplication by binding it in the affine context Δ. This ensures that no nondeterminism arises at the receiving end of communications.
- *No parallel composition of write mode processes.* The typing rules do not allow parallel composition of two write mode processes ($\mathsf{W} \parallel \mathsf{W}$). This ensures that no nondeterminism arises at the sending end of communications.

Before we go through a quick tour of the typing rules, we point out a few things: Value mode derivations and ordinary type polymorphism are omitted to avoid clutter, and algorithmic typing rules are given in the Appendix.

Rules for preventing read nondeterminism. To protect read channels from duplication, the nu rule binds read channels in the affine context. When reading from a channel, the letrd rule rebinds the channel in the affine context so that it may be used again. The rules for typing lambda abstractions (abs), fixed points (fix), and bang! types (bang) stipulate that they must be closed with respect to affine variables. Otherwise, they could violate the affinity of read channels.

Rules for preventing write nondeterminism. The mode composition rules given in Figure **??** prevent write mode processes from being composed in parallel. This is reflected in the fork rule, which derives the mode of $e_1 \mathrel{|\!\triangleright} e_2$ as the parallel composition of the mode of e_1 and the mode of e_2. To ensure this property is preserved during normalization, we must also forbid sequential composition of write mode processes.

3.3 Dynamic Semantics

Figures 4 and 5 define the dynamic syntax and semantics of ILC, respectively. We define a *configuration* C as a tuple of dynamic channel and process names Σ, a shared mutable store σ, and a pool of running and terminated processes π.

To state and prove the meta theory of ILC (Sec. 4), we extend the type system given above with typing rules for configurations, including typing environments for channels and stores. We omit these details here for space reasons, and we refer the interested reader to Sec. B.

We read the configuration reduction judgment $C_1 \longrightarrow C_2$ as "configuration C_1 steps to configuration C_2," and the local stepping judgment $\sigma_1; e_1 \longrightarrow \sigma_2; e_2$ for a single process e as "under store σ_1, expression e_1 steps to store σ_2 and expression e_2. The rules of local stepping follow a standard call-by-value semantics, where we streamline the definition with an evaluation context E.

Configuration stepping consists of five rules, including a congruence rule congr that permits some of the other rules to be simpler, by making the order of the pool unimportant.

$\boxed{\Delta_{in}; \Gamma \vdash e : U \dashv \Delta_{out}}$ Under input contexts Δ_{in} and Γ, expression e has type U and output context Δ_{out}.

$$\dfrac{}{\Delta; \Gamma, x : A \vdash x : A \dashv \Delta}\ \text{ivar} \qquad \dfrac{}{\Delta, x : X; \Gamma \vdash x : X \dashv \Delta}\ \text{avar} \qquad \dfrac{}{\Delta; \Gamma \vdash () : \mathbb{1} \dashv \Delta}\ \text{unit}$$

$$\dfrac{\begin{array}{c}\Delta_1; \Gamma \vdash e_1 : U_1 \dashv \Delta_2 \\ \Delta_2; \Gamma \vdash e_2 : U_2 \dashv \Delta_3\end{array}}{\Delta_1; \Gamma \vdash (e_1, e_2)_\pi : U_1 \times_\pi U_2 \dashv \Delta_3}\ \text{pair} \qquad \dfrac{\begin{array}{c}i \in \{1, 2\} \\ \Delta_1; \Gamma \vdash e : U_i \dashv \Delta_2\end{array}}{\Delta_1; \Gamma \vdash \mathsf{inj}^i_\pi(e) : U_1 +_\pi U_2 \dashv \Delta_2}\ \text{inj} \qquad \dfrac{\Delta_1; \Gamma \vdash e : A \dashv \Delta_2}{\Delta_1; \Gamma \vdash \mathsf{ref}(e) : \mathsf{Ref}\, A \dashv \Delta_2}\ \text{ref}$$

$$\dfrac{\begin{array}{c}\Delta_1; \Gamma \vdash e_1 : U_1 \times_\pi U_2 \dashv \Delta_2 \\ [\Delta_2; \Gamma], x_1 : U_1, x_2 : U_2 \vdash e : V \dashv \Delta_3\end{array}}{\Delta_1; \Gamma \vdash \mathsf{split}_\pi(e_1, x_1.x_2.e_2) : V \dashv \Delta_3 \div (x_1 : U_1, x_2 : U_2)}\ \text{split}$$

$$\dfrac{\begin{array}{c}\Delta_1; \Gamma \vdash e : U_1 +_\pi U_2 \dashv \Delta_2 \\ [\Delta_2; \Gamma], x_1 : U_1 \vdash e_1 : V \dashv \Delta_3 \qquad [\Delta_2; \Gamma], x_2 : U_2 \vdash e_2 : V \dashv \Delta_3\end{array}}{\Delta_1; \Gamma \vdash \mathsf{case}_\pi(e, x_1.e_1, x_2.e_2) : V \dashv \Delta_3 \div (x_1 : U_1, x_2 : U_2)}\ \text{case} \qquad \dfrac{\Delta_1; \Gamma \vdash e : \mathsf{Ref}\, A \dashv \Delta_2}{\Delta_1, \textcircled{w}; \Gamma \vdash \mathsf{get}(e) : A \dashv \Delta_2, \textcircled{w}}\ \text{get}$$

$$\dfrac{\begin{array}{c}\Delta_1; \Gamma \vdash e_1 : \mathsf{Ref}\, A \dashv \Delta_2 \\ \Delta_2; \Gamma \vdash e_2 : A \dashv \Delta_3\end{array}}{\Delta_1, \textcircled{w}; \Gamma \vdash \mathsf{set}(e_1, e_2) : \mathbb{1} \dashv \Delta_3, \textcircled{w}}\ \text{set} \qquad \dfrac{\cdot; \Gamma, x : A \to U \vdash e : A \to U \dashv \cdot}{\Delta; \Gamma \vdash \mathsf{fix}_\infty(x.e) : A \to U \dashv \Delta}\ \text{ifix}$$

$$\dfrac{\Delta_1, x : X \multimap U; \Gamma \vdash e : X \multimap U \dashv \Delta_2}{\Delta_1; \Gamma \vdash \mathsf{fix}_1(x.e) : X \multimap U \dashv \Delta_2 \div (x : X \multimap U)}\ \text{afix} \qquad \dfrac{\Delta_1; \Gamma \vdash e_1 : U_1 \dashv \Delta_2 \qquad [\Delta_2; \Gamma], x : U \vdash e_2 : V_1 \dashv \Delta_3}{\Delta_1; \Gamma \vdash \mathsf{let}_\pi\ x = e_1\ \mathsf{in}\ e_2 : V \dashv \Delta_3 \div (x : U)}\ \text{let}$$

$$\dfrac{\Delta_1; \Gamma \vdash e : A \dashv \Delta_2}{\Delta_1; \Gamma \vdash !\, e : !\, A \dashv \Delta_2}\ \text{bang} \qquad \dfrac{\Delta_1; \Gamma \vdash e : !\, A \dashv \Delta_2}{\Delta_1; \Gamma \vdash ¡\, e : A \dashv \Delta_2}\ \text{unbang} \qquad \dfrac{\cdot; \Gamma, x : A \vdash e : U \dashv \cdot}{\Delta; \Gamma \vdash \lambda_\infty x.\, e : A \to U \dashv \Delta}\ \text{iabs}$$

$$\dfrac{\Delta_1, x : X; \Gamma \vdash e : U \dashv \Delta_2}{\Delta_1; \Gamma \vdash \lambda_1 x.\, e : X \multimap U \dashv \Delta_2 \div (x : X)}\ \text{aabs} \qquad \dfrac{\textcircled{w}; \Gamma \vdash e : U \dashv \Delta_2}{\Delta_1; \Gamma \vdash \lambda_{\mathsf{w}}().\, e : \mathbb{1} \mathrel{-\!\!*} U \dashv \Delta_1}\ \text{wabs} \qquad \dfrac{\begin{array}{c}\Delta_1; \Gamma \vdash e_2 : U \dashv \Delta_2 \\ \Delta_2; \Gamma \vdash e_1 : U \to_\pi V \dashv \Delta_3\end{array}}{\Delta_1; \Gamma \vdash (e_1\ e_2)_\pi : V \dashv \Delta_3}\ \text{app}$$

$$\dfrac{\begin{array}{c}\Delta_1; \Gamma \vdash e_2 : \mathbb{1} \dashv \Delta_2 \\ \Delta_2; \Gamma \vdash e_1 : \mathbb{1} \mathrel{-\!\!*} U \dashv \Delta_3\end{array}}{\Delta_1, \textcircled{w}; \Gamma \vdash (e_1\ e_2)_{\mathsf{w}} : U \dashv \Delta_3}\ \text{wapp} \qquad \dfrac{\Delta_1, x_1 : \mathsf{Rd}\, S; \Gamma, x_2 : \mathsf{Wr}\, S \vdash e : A \dashv \Delta_2}{\Delta_1; \Gamma \vdash \nu(x_1, x_2).\, e : A \dashv \Delta_2}\ \text{nu}$$

$$\dfrac{\Delta_1; \Gamma \vdash e_1 : S \dashv \Delta_2 \qquad \Delta_2; \Gamma \vdash e_2 : \mathsf{Wr}\, S \dashv \Delta_3}{\Delta_1, \textcircled{w}; \Gamma \vdash \mathsf{wr}(e_1, e_2) : \mathbb{1} \dashv \Delta_3}\ \text{wr} \qquad \dfrac{\textcircled{w} \notin \Delta_1 \qquad \Delta_1; \Gamma \vdash e : \mathsf{Rd}\, S \dashv \Delta_2}{\Delta_1; \Gamma \vdash \mathsf{rd}(e) : !\, S \otimes \mathsf{Rd}\, S \dashv \Delta_2, \textcircled{w}}\ \text{rd} \qquad \dfrac{\begin{array}{c}\Delta_1; \Gamma \vdash e_1 : U \dashv \Delta_2 \\ \Delta_2; \Gamma \vdash e_2 : V \dashv \Delta_3\end{array}}{\Delta_1; \Gamma \vdash e_1 \mathrel{|\!\triangleright} e_2 : V \dashv \Delta_3}\ \text{fork}$$

$$\dfrac{\begin{array}{c}\Delta_1; \Gamma \vdash e_1 : U \dashv \Delta_2 \\ \Delta_1; \Gamma \vdash e_2 : U \dashv \Delta_2\end{array}}{\Delta_1; \Gamma \vdash e_1 \oplus e_2 : U \dashv \Delta_2}\ \text{choice}$$

Figure 3. Typing rules.

The other four rules consist of local stepping (via local), creating new processes (via fork), creating new channels (via nu), and read-write interactions (via rw). To avoid allocating the same name twice, the name set Σ records names of allocated channels and processes. Rule rw uses the syntax R for a combination of several (mutually exclusive) read-mode expressions, with a chosen read filling the (single) hole. When synchronized, the rule uses this syntax to eliminate all of the unchosen read-mode expressions before control continues with the chosen read effect. The relation $c_1 \rightsquigarrow c_2$ holds when

Channel names	$c ::= \cdots$	Evaluation contexts	$E ::= \bullet \mid \mathsf{let}_\pi\ x = E \ \mathsf{in}\ e \mid \mathsf{let!}\ x = E\ \mathsf{in}\ e$
Process names	$p, q ::= \cdots$		$\mid (E\ e)_\pi \mid (v\ E)_\pi \mid \mathsf{ref}(E) \mid \mathsf{get}(E)$
Store locations	$\ell ::= \cdots$		$\mid \mathsf{set}(E, e) \mid \mathsf{set}(\ell, E)$
Name sets	$\Sigma ::= \varepsilon \mid \Sigma, c \mid \Sigma, p$		$\mid \mathsf{split}_\pi(E, x_1.x_2.e) \mid \mathsf{case}_\pi(E, x_1.e_1, x_2.e_2)$
Stores	$\sigma ::= \sigma \mid \sigma, \ell{:}v$	Read contexts	$R ::= \bullet \mid e \oplus R \mid R \oplus e$
Process pools	$\pi ::= \varepsilon \mid \pi, p{:}e$		
Configurations	$C ::= \langle \Sigma; \sigma; \pi \rangle$		

Figure 4. Channel names, process names, configurations and evaluation contexts.

$\boxed{C_1 \equiv C_2}$ Configurations C_1 and C_2 are equivalent.

$$\frac{\pi_1 \equiv_{\mathsf{perm}} \pi_2}{\langle \Sigma; \sigma; \pi_1 \rangle \equiv \langle \Sigma; \sigma; \pi_2 \rangle}\ \text{permProcs}$$

$\boxed{C_1 \longrightarrow C_2}$ Configuration C_1 reduces to C_2.

$$\frac{\sigma_1; e_1 \longrightarrow \sigma_2; e_2}{\langle \Sigma; \sigma_1; \pi, p{:}E[e_1] \rangle \longrightarrow \langle \Sigma; \sigma_2; \pi, p{:}E[e_2] \rangle}\ \text{local} \qquad \frac{q \notin \Sigma}{\langle \Sigma; \sigma; \pi, p{:}E[e_1 \mathrel{|\!\rhd} e_2] \rangle \longrightarrow \langle \Sigma, q; \sigma; \pi, q{:}e_1, p{:}E[e_2] \rangle}\ \text{fork}$$

$$\frac{C_1 \equiv C_1' \qquad C_1' \longrightarrow C_2' \qquad C_2' \equiv C_2}{C_1 \longrightarrow C_2}\ \text{congr} \qquad \frac{c_1, c_2 \notin \Sigma}{\langle \Sigma; \sigma; \pi, p{:}E[\nu(x_1, x_2).\ e] \rangle \longrightarrow \langle \Sigma, c_1, c_2; \sigma; \pi, p{:}E[[c_1/x_1][c_2/x_2]e] \rangle}\ \text{nu}$$

$$\frac{c_2 \rightsquigarrow c_1}{\langle \Sigma; \sigma; \pi, p{:}E_1[R[\mathsf{rd}(c_1)]], q{:}E_2[\mathsf{wr}(c_2, v)] \rangle \longrightarrow \langle \Sigma; \sigma; \pi, p{:}E_1[(v, c_1)], q{:}E_2[()] \rangle}\ \text{rw}$$

$\boxed{\sigma_1; e_1 \longrightarrow \sigma_2; e_2}$ Under store σ_1, expression e_1 reduces to $\sigma_2; e_2$.

$$\frac{}{\sigma; \mathsf{let}_\pi\ x = v\ \mathsf{in}\ e \longrightarrow \sigma; [v/x]e}\ \text{let} \qquad \frac{}{\sigma; \mathsf{let!}\ x = v\ \mathsf{in}\ e \longrightarrow \sigma; [v/x]e}\ \text{let!} \qquad \frac{}{\sigma; ((\lambda x.\ e)\ v)_\pi \longrightarrow \sigma; [v/x]e}\ \text{app}$$

$$\frac{}{\sigma; \mathsf{split}_\pi((v_1, v_2), x_1.x_2.e) \longrightarrow \sigma; [v_1/x_1][v_2/x_2]e}\ \text{split} \qquad \frac{}{\sigma; \mathsf{case}_\pi(\mathsf{inj}^i_\pi(v), x_1.e_1, x_2.e_2) \longrightarrow \sigma; [v/x_i]e_i}\ \text{case}$$

$$\frac{}{\sigma; \mathsf{fix}(x.e) \longrightarrow \sigma; [\mathsf{fix}(x.e)/x]e}\ \text{fix} \qquad \frac{\ell \notin dom(\sigma)}{\sigma; \mathsf{ref}(v) \longrightarrow \sigma, \ell{:}v; \ell}\ \text{ref} \qquad \frac{\sigma(\ell) = v}{\sigma; \mathsf{get}(\ell) \longrightarrow \sigma; v}\ \text{get} \qquad \frac{}{\sigma; \mathsf{set}(\ell, v) \longrightarrow \sigma\{\ell := v\}; ()}\ \text{set}$$

Figure 5. Reduction semantics.

channel names c_1 and c_2 refer to the write and read ends of a common channel. (The rules abstract over how channel names, and this relation, are structured.)

4 ILC Metatheory

Intuitively, ILC's type system design enforces that a configuration's reduction consists of a unique (*deterministic*) sequence of reader-writer process pairings, and is *confluent* with any other reduction choice that exchanges the order of *other* (non-interactive) reduction steps. As explained in Sec. 3, ILC's type system does so by restricting the dataflow of read channels (via affine typing) and by restricting the composition of write effects (forbidding write-mode parallel processes). The proofs of type soundness, whose statements we discuss next, establish the validity of these invariants. These language-level invariants support confluence theorems, also stated below. These theorems include *full confluence*: Any two full reductions of a configuration yield a pair of equivalent configurations (isomorphic, up to a renaming of non-deterministic name choices). For space reasons, we omit the full proof of confluence, but we include a detailed sketch in Sec. B.

4.1 Type Soundness

We prove type soundness of ILC via mostly-standard notions of progress and preservation. To state these theorems, we follow the usual recipe, except that we give a special definition of "program termination" which permits deadlocks. Informally, C **term** holds when:

1. C is fully normal: every process in C is normalized (consists of a value).
2. C is (at least partially) deadlocked: some (possibly-empty) portion of C is normal, and there exists one or more read-mode processes in C, or there exists one or more write-mode processes in C, however, no reader-writer pair exists for a common channel c.

Recall that ILC is concerned with enforcing *confluence* as its central meta theoretic property, *not* deadlock freedom. Confluence implies, among other things, that the order of reduction steps is inconsequential, and that no process scheduling choices will affect the final outcome.

Lemma 4.1 (Non-progress)**.** *For all configurations C, store typings Ω, channel typings Ψ, and process typings Φ, if $\Omega; \Psi \vdash C : \Phi$ and C **term**, then $\nexists C'$ such that $C \longrightarrow C'$.*

Theorem 4.2 (Progress)**.** *For all configurations C, store typings Ω, channel typings Ψ, and process typings Φ, if $\Omega; \Psi \vdash C : \Phi$ then either C **term**, or $\exists C'$ such that $C \longrightarrow C'$.*

Theorem 4.3 (Preservation)**.** *For all configurations C, C', store typings Ω, channel typings Ψ and process typings Φ, if $\Omega; \Psi \vdash C : \Phi$ and $C \longrightarrow C'$ then there exists store typing extension $\Omega' \supseteq \Omega$ channel typing extension $\Psi' \supseteq \Psi$ and process typing extension $\Phi' \supseteq \Phi$ such that $\Omega'; \Psi' \vdash C' : \Phi'$.*

4.2 Confluence

ILC's type system enforces confluence up to non-deterministic naming choices in rules ref, nu, and fork (Fig. 5). To account for different choices of dynamically-named store locations, channels and processes, respectively, we state and prove confluence with respect to a renaming function f, which consistently renames these choices in a related configuration:

Theorem 4.4 (Single-step confluence)**.** *For all well-typed configurations C, if $C \longrightarrow C_1$ and $C \longrightarrow C_2$ then there exists renaming function f such that either:*

1. *$C_1 = f(C_2)$, or*
2. *$\exists C_3$ such that $C_1 \longrightarrow C_3$ and $f(C_2) \longrightarrow C_3$.*

Intuitively, the sister configuration C_2 is either different because of a name choice (case 1), or a different process scheduling choice (case 2). In either case, there exists a renaming of any choice made to reach C_2, captured by function f. By composing multiple uses of this theorem, and the renaming functions that they construct, we prove a multi-step notion of confluence that reduces a single configuration C to two equivalent terminal configurations, C_1 and C_2:

Theorem 4.5 (Full confluence)**.** *For all well-typed configurations C, if $C \longrightarrow^* C_1$ and $C \longrightarrow^* C_2$ and C_1 **term** and C_2 **term** then there exists renaming function f such that $C_1 = f(C_2)$.*

5 Implementation

We have implemented an ILC interpreter in Haskell, which at present consists of 2.3K source lines of code. We plan to open source our implementations of ILC and SaUCy.

For economy of use, our implementation performs polymorphic type inference, bounded polymorphic mode inference, and affinity inference. Polymorphism on modes is bounded precisely due to our restriction on parallel write mode composition. Moreover, a consequence of any kind of mode polymorphism at all is that the modes of higher order functions can be dependent on the modes of its function arguments. We give a taste of this below.

This first example features full mode polymorphism and no dependent modes.

```
loop :: ∀ a b m . Rd a ⊸ !(a →ₘ b) ⊸_R b
letrec loop c !f = let ⟨!v, c⟩ = rd c in f v; loop c !f
```

The loop function takes as arguments a read channel c and an intuitionistic function wrapped in a bang!, which is unpacked as f. It says to first read from the channel c, unpack the read value as v, apply the function f to v, and then recursively call loop again. Here, the mode *m* carried over the function argument is fully polymorphic, since no parallel compositions take place, and the function's mode R is monomorphic.

This next example features bounded mode polymorphism and dependent modes.

```
par :: ∀ a b c {(m, n) | m ∥ n ⇒ p} .
       (a →ₘ b) → (a→ₙ c) → a →ₚ c
let par f g v = f v |▷ g v
```

The par function takes as arguments two functions f and g and a third argument v. It says to compute f v and g v in parallel. Here, because write mode expressions cannot be composed in parallel, at most one of the modes carried over the function arguments can have mode W. Additionally, the mode p carried over the rightmost arrow is dependent on the modes of the function arguments.

6 SaUCy

Using ILC, we build a concrete, executable implementation of the UC framework, dubbed SaUCy. Then, we demonstrate the versatility of SaUCy in three ways:

1. We define a protocol composition operator and its associated composition theorem.
2. We walk through an instantiation of UC commitments.
3. We use ILC's type system to reason about "reentrancy," a subtle definitional issue in UC that has only recently been studied.

6.1 Probabilistic Polynomial Time in ILC

The goal of cryptography reduction is to relate every bad event in a protocol to a *probabilistic polynomial time computation* that solves a hard problem. The ILC typing rules do not guarantee termination, let alone polynomial time normalization, so we must tackle this in metatheory. Also, since ILC is effectively deterministic (confluent), we will need to express random choices some other way. To meet these needs we define a judgment about ILC terms that take a security parameter and a stream of random bits.

Definition 6.1 (Polynomial time normalization)**.** The judgment that e is polynomial time normalizable, written PPT e, is defined as follows:

$$\frac{\begin{array}{c}\cdot\,;\cdot \vdash e : !\,\mathsf{Nat} \multimap\ !\,[\mathsf{Bit}] \multimap_m !\,\mathsf{Bit} \\ \forall\, \mathsf{k} \in \mathsf{Nat}.\ \forall\, r \in [\mathsf{Bit}]^{(poly(\mathsf{k}))}.\ \mathsf{e}\ !\mathsf{k}\ !\mathsf{r} \rightarrow^{poly(\mathsf{k})} \mathsf{v}\end{array}}{\mathsf{PPT\ e}}\ \text{ppt}$$

This says that if for all security parameters k and all bitstrings r with length polynomial in k the term e !k !r normalizes to a value v in $poly(\mathsf{k})$ steps.[2]

Definition 6.2 (Value Distribution)**.** Because processes are confluent, we know that if e !k !r $\rightarrow^*$ v then the value v is unique. We can therefore define the probability distribution ensemble $D(\mathsf{e}) = \{D_{\mathsf{e},\mathsf{k}}\}_{\mathsf{k}}$ based on a uniform distribution U_k over k-bit strings r, so the distribution $D_{\mathsf{e},\mathsf{k}}$ is given as

$$D_{\mathsf{e},\mathsf{k}}(\mathsf{v}) = \sum_{\mathsf{r} \in R} U_{\mathsf{k}}(\mathsf{r}), \quad \text{for } R = \{\mathsf{r} \mid \mathsf{e}\ !\mathsf{k}\ !\mathsf{r} \rightarrow^* \mathsf{v}\}.$$

Definition 6.3 (Indistinguishability)**.** What remains is to define a notion of indistinguishability for value distributions. However, we need to clarify when polynomial time normalization is an assumption or a proof obligation. To simplify things later, we define a partial order $\mathsf{e}_1 \leq \mathsf{e}_2$, which captures that e_2 must be PPT if e_1 is PPT, and if so, that their value distributions are statistically similar.

$$\frac{\mathsf{PPT\ e_1} \implies (\mathsf{PPT\ e_2\ and}\ D(\mathsf{e}_1) \sim D(\mathsf{e}_2))}{\mathsf{e}_1 \leq \mathsf{e}_2}\ \text{indist}$$

6.2 The UC Framework, Concretely

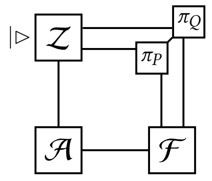


Figure 6. execUC.

The implementation of SaUCy is centered around a definition of UC execution model in ILC. For the most part, this just involves connecting channels as illustrated in Figure 6. For demonstration, we only show the case of two-party protocols (à la Simplified UC [13]), which will suffice for our example of instantiating universally composable commitments. Also, we will only aim to show the case of *static* corruptions, in which parties are corrupted at the onset of the execution. This is in contrast to *adaptive* corruptions, in which parties can be corrupted as the execution proceeds.

```
execUC :: ∀ ... . X_z ⊸ X_p ⊗ X_q ⊸ X_f ⊸ X_a
            ⊸ ! Nat ⊸ ! [Bit] ⊸ ! Crupt ⊸_m ! Bit
let execUC z (p,q) f a k !r crupt =
  ν ... . let ... in
          let (rf,ra,rp,rq,rz) = splitBits5 r in
              f !crupt ... k !rf
            |▷ a !crupt ... k !ra
            |▷ wrapperP p crupt ... k !rp
            |▷ wrapperQ q crupt ... k !rq
            |▷ z ... k !rz
```

The function execUC is parameterized by an environment z, protocol parties p and q, an adversary a, an ideal functionality f, a security parameter k, a random bitstring r, and a static corruption model crupt :: !Crupt. The Crupt datatype is defined below, with its variants denoting the cases when party p is corrupt, party q is corrupt, or no party is corrupt, respectively.

```
data Crupt = CruptP | CruptQ | CruptNone
```

Some careful programming is needed to handle the cases when the adversary sends messages on behalf of corrupted parties; this is captured by wrapper{P,Q} (shown in the Appendix).

Note that for space and readability, we elide channel allocation/distribution and wrapper definitions (with ellipses). We also abbreviate the type signature (e.g., X_z is the type of z). More details can be found in the Appendix.

6.3 Defining UC Security in ILC

The central security definition in UC is protocol emulation. The guiding principle is that π emulates ϕ if the environment cannot distinguish between the two protocols. Our first attempt is the following, where $\mathcal{S}$ is the simulator that translates every attack in the real world into an attack expressed in the ideal world:

$$\frac{\forall\, \mathcal{Z}.\ \mathsf{execUC}\ \mathcal{Z}\ \pi\ \mathcal{F}_1\ \mathbb{1}_{\mathcal{A}} \leq \mathsf{execUC}\ \mathcal{Z}\ \phi\ \mathcal{F}_2\ \mathcal{S}}{\mathcal{S} \vdash (\pi, \mathcal{F}_1) \approx (\phi, \mathcal{F}_2)}$$ ~~emulate~~

To remark on a few notation choices: we make the functionality explicit, so emulation is a relationship between protocol-functionality pairs. Here, $\mathbb{1}_{\mathcal{A}}$ is the dummy adversary, which just relays messages between the environment and the parties/functionality. We elide the standard dummy lemma that shows this is without loss of generality; the intuition is that whatever an adversary can do, the environment can achieve using $\mathbb{1}_{\mathcal{A}}$.

Unfortunately this simple definition turns out to be vacuous: a degenerate protocol π can emulate anything simply failing to be PPT, e.g. by diverging. To put it another way, the problem is the definition imposes a proof obligation on the simulator $\mathcal{S}$ but not on π. What we want to say is that the *real*

[2]Note that the normalization must be polynomial time for all r, though we could have defined it to hold with high probability.

world protocol $(\pi, \mathcal{F}_1)$ must be well behaved whenever the *ideal world* $(\phi, \mathcal{F}_2)$ is. However, even a reasonable protocol can result in non-PPT executions if paired with a divergent environment. In fact, giving a precise but composable notion of polynomial-time for interactive processes has been an ongoing challenge in UC. In GNUC, the approach is to define a well behaved environment, independently of its execution context—roughly that the total size of its outgoing messages is bounded by a polynomial, and that its running time is bounded if its total received input size is bounded [21]. This notion is composable as desired, although its use requires additional distinctions between "invited" and "uninvited" messages, which seems cluttered. RSIM makes analogous choices [3]. While we think these could be equally applied to ILC, our goal here is to provide a simpler notion.

We define protocol emulation by requiring a simulation in both directions, so every behavior in the ideal world must correspond to a behavior in the real world and vice versa.

Definition 6.4 (Protocol Emulation)**.** The judgment that one protocol-functionality pair $(\pi, \mathcal{F}_1)$ securely emulates another $(\phi, \mathcal{F}_2)$ (as proven the simulators $\mathcal{S}_\mathcal{R}, \mathcal{S}_\mathcal{I}$) is defined as

$$\frac{\begin{array}{l}\forall\, \mathcal{Z}.\ \mathsf{execUC}\ \mathcal{Z}\ \phi\ \mathcal{F}_2\ \mathbb{1}_\mathcal{A} \le \mathsf{execUC}\ \mathcal{Z}\ \pi\ \mathcal{F}_1\ \mathcal{S}_\mathcal{R} \\ \qquad \mathsf{execUC}\ \mathcal{Z}\ \pi\ \mathcal{F}_1\ \mathbb{1}_\mathcal{A} \le \mathsf{execUC}\ \mathcal{Z}\ \phi\ \mathcal{F}_2\ \mathcal{S}_\mathcal{I}\end{array}}{\mathcal{S}_\mathcal{R}, \mathcal{S}_\mathcal{I} \vdash (\pi, \mathcal{F}_1) \approx (\phi, \mathcal{F}_2)}\ \text{emulate}$$

We remark this definition goes against the UC convention of requiring simulation in one direction only. One direction is preferable intuitively because it should be OK if the protocol is even more secure than its specification. Requiring both could be too restrictive, although we have not encountered such problem in our examples. In any case, the benefit is this simplifies the polynomial time notion: vacuous protocols are clearly ruled out by the top condition, and both simulations are only required to be PPT if $\mathcal{Z}$ is.

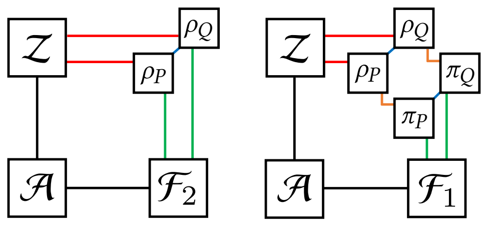
6.4 A Composition Theorem in SaUCy

As a first demonstration of SaUCy, we work through the development of a composition operator, and give a theorem explaining its use.

Definition 6.5 (UC realizes)**.** To set out, we introduce the notation of "realizes," which views a protocol as a way of instantiating a specification functionality $\mathcal{F}_2$ from a setup assumption functionality $\mathcal{F}_1$,

$$\frac{(\pi,\ \mathcal{F}_1) \approx (\mathsf{id}_\pi, \mathcal{F}_2)}{\mathcal{F}_1 \xrightarrow{\pi} \mathcal{F}_2}\ \text{realizes}$$

where id_π is the *dummy protocol*, which simply relays messages between the environment and the functionality. This notation is convenient because it suggests a categorical approach to composition.



$$\begin{array}{l}
\textbf{let}\ (\circ)\ (\rho_P,\ \rho_Q)\ (\pi_P,\ \pi_Q) \\
\qquad\qquad \mathsf{w\rho_P2Z\ rZ2\rho_P\ w\rho_Q2Z\ rZ2\rho_Q} \\
\qquad\qquad \mathsf{w\rho_P2F\ rF2\rho_P\ w\rho_Q2F\ rF2\rho_Q} \\
\qquad\qquad \mathsf{w\rho_P2\rho_Q\ r\rho_Q2\rho_P\ w\rho_Q2\rho_P\ r\rho_P2\rho_Q} = \\
\ \nu \ldots .\ \pi_P\ \mathsf{w\pi_P2\rho_P\ r\rho_P2\pi_P\ w\rho_P2F\ rF2\rho_P\ w\pi_P2\pi_Q\ r\pi_Q2\pi_P} \\
\quad |\triangleright\ \pi_Q\ \mathsf{w\pi_Q2\rho_Q\ r\rho_Q2\pi_Q\ w\rho_Q2F\ rF2\rho_Q\ w\pi_Q2\pi_P\ r\pi_P2\pi_Q} \\
\quad |\triangleright\ \rho_P\ \mathsf{w\rho_P2Z\ rZ2\rho_P\ w\rho_P2\pi_P\ r\pi_P2\rho_P\ w\rho_P2\rho_Q\ r\rho_Q2\rho_P} \\
\quad |\triangleright\ \rho_Q\ \mathsf{w\rho_Q2Z\ rZ2\rho_Q\ w\rho_Q2\pi_Q\ r\pi_Q2\rho_Q\ w\rho_Q2\rho_P\ r\rho_P2\rho_Q}
\end{array}$$

Figure 7. Protocol composition operator.

Theorem 6.1 (Composition Theorem)**.**

$$\frac{\mathcal{F}_1 \xrightarrow{\pi} \mathcal{F}_2 \qquad \mathcal{F}_2 \xrightarrow{\rho} \mathcal{F}_3}{\mathcal{F}_1 \xrightarrow{\rho\, \circ\, \pi} \mathcal{F}_3}$$

The idea is that the $\rho \circ \pi$ can be defined in a natural way, where the ideal functionality channel of ρ is connected to the environment channel of π, as illustrated and defined in Figure 7.

Proof. To prove the theorem we construct the simulators $\mathcal{S}_{\mathcal{R},\rho} \circ \mathcal{S}_{\mathcal{R},\pi}$ (respectively $\mathcal{S}_{\mathcal{I},\rho} \circ \mathcal{S}_{\mathcal{I},\pi}$) in the natural way as well (given in the Appendix). Our proof obligation is to introduce an arbitrary environment $\mathcal{Z}$ and conclude

$$\mathsf{execUC}\ \mathcal{Z}\ (\rho \circ \pi)\ \mathcal{F}_1\ \mathbb{1}_\mathcal{A} \le \mathsf{execUC}\ \mathcal{Z}\ \mathbb{1}_\pi\ \mathcal{F}_3\ (\mathcal{S}_{\mathcal{I},\rho} \circ \mathcal{S}_{\mathcal{I},\pi})$$

The main idea is to notice that that we can bring ρ from the composed protocol into the environment as $(\mathcal{Z} \circ \rho)$, reflecting the fact that the environment is meant to represent arbitrary outer protocols. The following derivation completes the proof:

$$\begin{array}{lr}
\quad \mathsf{execUC}\ \mathcal{Z}\ (\rho \circ \pi)\ \mathcal{F}_1\ \mathbb{1}_\mathcal{A} & \\
= \mathsf{execUC}\ (\mathcal{Z} \circ \rho)\ \pi\ \mathcal{F}_1\ \mathbb{1}_\mathcal{A} & \text{(By inspection)} \\
\le \mathsf{execUC}\ (\mathcal{Z} \circ \rho)\ \mathsf{id}_\pi\ \mathcal{F}_2\ \mathcal{S}_{\mathcal{I},\pi} & \text{(From } \mathcal{F}_1 \xrightarrow{\pi} \mathcal{F}_2\text{)} \\
= \mathsf{execUC}\ (\mathcal{S}_{\mathcal{I},\pi} \circ \mathcal{Z})\ \rho\ \mathcal{F}_2\ \mathbb{1}_\mathcal{A} & \text{(By inspection)} \\
\le \mathsf{execUC}\ (\mathcal{S}_{\mathcal{I},\pi} \circ \mathcal{Z})\ \mathsf{id}_\pi\ \mathcal{F}_3\ \mathcal{S}_{\mathcal{I},\rho} & \text{(From } \mathcal{F}_2 \xrightarrow{\rho} \mathcal{F}_3\text{)} \\
= \mathsf{execUC}\ \mathcal{Z}\ \mathsf{id}_\pi\ \mathcal{F}_3\ (\mathcal{S}_{\mathcal{I},\pi} \circ \mathcal{S}_{\mathcal{I},\rho}) & \text{(By inspection)}
\end{array}$$

The remaining case for $\mathcal{S}_{\mathcal{R},\rho} \circ \mathcal{S}_{\mathcal{R},\pi}$ is symmetric. □

Other notions of composition. Our composition operator above is just a starting point. The "universal composition" [12] operator essentially multiplexes sessions identified by unique tags (*session ids*), while a joint state composition theorem collapses multiple subroutines into one [15]. Despite its name, development in UC often involves defining additional composition operators. For example, interesting composition often happens "in the functionality" through higher order "wrapper" functionalities [22, 24] which we would express through abstraction. Some security properties require a generalized notion of ideal functionality that the environment can interact with directly. All the above motivate the development of the ILC core calculus as a flexible foundation; developing them in ILC is important future work.

6.5 Instantiating UC Commitments

We now walk through an instantiation of UC commitments (à la Canetti and Fischlin [14]). Instantiation proofs in SaUCy follow a standard rhythm. We start with a security definition as an ideal functionality (such as $\mathcal{F}_{\text{COM}}$), give the protocol, construct a simulator, and finally complete the relational analysis on paper.

UC commitments can be instantiated with standard cryptographic assumptions, for example the RSA problem [30]. They also rely on a "trusted setup," or common reference string, which are essentially public parameters generated ahead of time (modeled as an ideal functionality $\mathcal{F}_{\text{CRS}}$).

Extending ILC with cryptographic primitives. UC Commitments are realized from cryptographic primitives, such as trapdoor permutations, which require extensions to ILC. The new syntactic forms are given with their static and dynamic semantics in Figure 10.

Commitment Protocol. The commitment protocol by Canetti and Fischlin [14] is implemented in ILC as follows:

```
committer :: ∀ ... . ! Wr Msg ··· ! Nat ⊸ ! [Bit] ⊸_R ! 𝟙
let committer toZ frZ !toF frF !toQ frQ !k !bits =
  let ⟨!(Commit b), frZ⟩= rd frZ in
    wr GetCRS → toF ;
    let ⟨!(PublicStrings σ pk_0 pk_1), frF⟩ = rd frF in
    let r = take k bits in
    let x = if b == 0 then prg pk_0 r
            else xors (prg pk_1 r) σ in
      wr Commit' x → toQ ;
      let ⟨!Open, frZ⟩= rd frZ in
        !(wr (Open' b r) → toQ)

receiver :: ∀ ... . ! Wr Msg ··· ! Nat ⊸ ! [Bit] ⊸_R ! 𝟙
let receiver !toZ frZ toF frF toP frP k bits =
  let ⟨!(Commit' x), frP⟩ = rd frP in
    wr GetCRS → toF ;
    let ⟨!(PublicStrings σ pk_0 pk_1), frF⟩ = rd frF in
      wr Receipt → toZ ;
      let ⟨!(Open' b r), frP⟩ = rd frP in
        if (b == 0 && x == prg pk_0 r) ||
           (b == 1 && x == xors (prg pk_1 r) σ)
        then !(wr (Opened b) → toZ)
        else error "Cannot occur in honest case."
```

To briefly summarize what is going on: The setup CRS functionality (given in the Appendix) samples a random string σ and two trapdoor pseudorandom generators (prgs pk_0, pk_1). To commit to b, the committer produces a string y that is the result of applying one or the other of the prgs, and if $b = 1$ additionally applying xor with σ. The intuitive explanation why this is hiding is that without the trapdoor, it is difficult to tell whether a random $4k$-bit string is in the range of either prg. To open the commitment, the committer simply reveals the preimage and the receiver checks which of the two cases applies. The intuitive explanation why this is binding is that it is difficult to find a pair $y, y \oplus \sigma$ that are respectively in the range of both prgs.

Defining the simulator. The UC proof consists of two simulators, one for the ideal world and one for the real world. The ideal world simulator is ported directly from the UC literature [14]. The non-standard real world simulator, given in the Appendix, is trivial, but necessary because our protocol emulation definition requires simulation in both directions.

The ideal world simulator generates its own "fake" CRS for which it stores the trapdoors. The string σ is not truly random, but instead is the result of combining two evaluations of the prgs. In Figure 8 we show the case that the committer P is corrupt (the other case is in the Appendix). The simulator is activated when $\mathcal{Z}$ sends a message (Commit′ y); in the real world, this is relayed by the dummy adversary to Q, who outputs Committed back to the environment. Hence to achieve the same effect in the ideal word, the simulator must send (Commit b) to $\mathcal{F}_{\text{COM}}$. To extract b from y, the simulator makes use of the prg trapdoor check which one has y in its range. It is necessary to argue by cryptographic reduction that this simulation is sound.

Relational argument. The goal of the relational analysis is to show that an environment's output in the real world is indistinguishable from its output in the ideal world. The proof follows the one in Canetti and Fischlin [14].

Proof Sketch. Consider the following ensembles:

$$D_{\mathcal{R}} = D(\text{execUC } \mathcal{Z} \text{ (committer, receiver) fCrs dummyA})$$
$$D'_{\mathcal{R}} = D(\text{execUC } \mathcal{Z} \text{ (committer, receiver) bCrs dummyA})$$
$$D_{\mathcal{I}} = D(\text{execUC } \mathcal{Z} \text{ (dummyP, dummyQ) fCom simI})$$

The ensemble $D_{\mathcal{R}}$ is over the output of $\mathcal{Z}$ in a real world execution. The ensemble $D'_{\mathcal{R}}$ is similar, except $\mathcal{Z}$ runs with a bad functionality that computes fake public strings in the same way that the simulator does. The ensemble $D_{\mathcal{I}}$ is over the output of $\mathcal{Z}$ in an ideal world execution. The goal is to show that $D_{\mathcal{R}} \sim D_{\mathcal{I}}$. The proof proceeds by first showing that distinguishing between $D_{\mathcal{R}}$ and $D'_{\mathcal{R}}$ reduces to breaking the pseudorandomness of tdp (hence, $D_{\mathcal{R}} \sim D'_{\mathcal{R}}$),

```
let simI !crupt !toZ frZ !toF frF !toP frP !toQ frQ !k !bits =
  let rbits = ref bits in
  let (pk_0,td_0) = kgen k in
  let (pk_1,td_1) = kgen k in
  let r_0 = sample k rbits in
  let r_1 = sample k rbits in
  let σ = xors (prg pk_0 r_0) (prg pk_1 r_1) in
    match crupt with
    | CruptP ⇒
      let ⟨!GetCRS, frZ⟩ = rd frZ in
        wr (X2Z (PublicStrings σ pk_0 pk_1)) →toZ ;
        let ⟨!(A2P (Commit' y)), frZ⟩ = rd frZ in
          if check td_0 pk_0 y then
            wr (Commit 0) → toP
          else
            if check td_1 pk_1 (xors y σ) then
              wr (Commit 1) → toP
            else error "Fail" ;
          let ⟨!(A2P (Open' b r)), frZ⟩ = rd frZ in
            if b == 0 && y == prg pk_0 r ||
              b == 1 && y == xors (prg pk_1 r) σ
            then !(wr Open → toP)
            else error "Fail"
    | ...
```

Figure 8. Ideal world simulator (excerpt) for UC commitment (full version in Appendix).

and then by showing that distinguishing between $D'_{\mathcal{R}}$ and $D_{\mathcal{I}}$ also reduces to breaking the pseudorandomness of tdp (hence, $D'_{\mathcal{R}} \sim D_{\mathcal{I}}$). By the transitivity of indistinguishability, $D_{\mathcal{R}} \sim D_{\mathcal{I}}$. □

6.6 Reentrancy in SaUCy

The cryptography community has recently identified subtleties in defining UC ideal functionalities that relate to reentrancy and the scheduling of concurrent code, such that several functionalities in the literature are ambiguous as ITMs [11]. Although concerning, these issues have no cryptographic flavor, but instead are better addressed from the PL viewpoint. To illustrate, consider the following fragment of (untyped) ILC syntax, which allows an adversary to control the delivery schedule of messages from P to Q (an asynchronous channel):

```
let reentrantF frP frA ... =
  loop frP !(λ m . (let ⟨!Ok, frA⟩ = rd frA in wr m → toQ)
                  |▷ wr m → toA)
```

After receiving input from party P, it notifies the adversary, then forks a background thread to wait for Ok before delivering the message. This introduces a race condition: suppose input message m_1 is sent by P, but then the adversary $\mathcal{A}$, before sending Ok, instead returns control to $\mathcal{Z}$, which passes P a second input m_2. Now there are two queued messages. Which one gets delivered when the adversary sends Ok?

To resolve this paradox, notice that this fragment is untypeable in ILC. The race condition occurs because the read channel frA is duplicated. Since frA is linear in the function body, the function would not be typeable as intuitionistic as required by the loop construct. Camenisch et al. [11] identified several strategies for resolving this problem in UC, which in turn are expressible ILC. One approach is to make the process explicitly sequential, such that the arrival of a second message before the first is delivered causes execution to get stuck:

```
letrec sequentialF frP frA ... =
  let ⟨!m, frP⟩ = rd frP in
    wr m → toA;
    let ⟨!Ok, frA⟩ = rd frA in
      wr m → toQ;
      sequentialF frP frA
```

Alternatively, we may discard such messages arriving out of order, returning them to sender; we express this in ILC using the external choice operator,

```
letrec discardingF frP frA ... =
  let ⟨!m, frP⟩ = rd frP in
    wr m → toA;
    letrec iloop frP frA =
       (let ⟨_, frP⟩ = rd frP in
          wr Discard → toP;
          iloop frP frA)
       ⊕ (let ⟨!Ok, frA⟩ = rd frA in
          wr m → toQ;
          discardingF frP frA ...)
     in iloop frP frA
```

7 Related Work

7.1 Process Calculi

Process calculi have a long and rich history. ILC occupies a point in this space that is particularly suited to faithfully capturing interactive Turing machines (and hence, computational cryptography), but plenty of existing calculi are also cryptographically-flavored and/or enjoy similar properties to ILC. We survey some of them here.

With symbolic semantics. Two early adaptations of process calculi for reasoning about cryptographic protocols were the spi calculus [2] and the applied π-calculus [1], both of which extend the π-calculus with cryptographic operations [34]. Symbolic UC [9] is a simulation-based security framework in this setting. However, protocols proven secure in the symbolic setting may not be realizable with any cryptographic primitives based on hardness assumptions.

With computational semantics. Naturally, ensuing work has turned to bridging the gap between this PL-style of formalization and the computational model of cryptography by outfitting these calculi with a computational semantics. Lincoln et al. [29] give a computational semantics to a variant of the π-calculus, which allows one to define communicating probabilistic polynomial-time processes; Mateus et al. [31] adapts their calculus to explore (sequential) compositionality properties in protocols. A drawback of these protocols is that they embed probabilistic choices directly into the definition—essentially when faced with non-determinism, each path has equal probability. Laud [27] gives a computational semantics to the spi calculus, which additionally includes a type system for ensuring well-typed protocols preserve the secrecy of messages given to it by users.

With confluence. There are a number of other process calculi that enjoy confluence. Berger et al. [6] describe a type system for capturing deterministic (sequential) computation in the π-calculus. The type system uses affineness and stateless replication to achieve deterministic computation. Fowler et al. [19] present a core linear lambda calculus with (binary) session-typed channels and exception handling that enjoys confluence and termination. The calculus only considers two-party protocols, so for our multiparty setting, ILC requires a sophisticated type system to achieve confluence.

7.2 Tools for Cryptographic Analysis

There are a variety of computer-aided tools for cryptographic analysis. Among the symbolic tools include the NRL protocol analyzer [32], Maude-NPA [18], and Proverif [8]. Among the computational tools include CertiCrypt [5], EasyCrypt [4], CryptoVerif [7], Cryptol [28], and F* [35]. Although the computational tools do not support message-passing concurrency, it would be interesting to embed ILC into EasyCrypt or F* in order to use its tooling.

7.3 Variations of the UC Framework

A number of variants of the UC framework have been proposed to address some of its hitches or to make it easier to use. The reactive simulatability framework (RSIM) [3] is a framework for representing and analyzing cryptographic protocols in a composable way. In contrast with UC, which uses ITMs as its computational model, RSIM uses probabilistic IO automata, which is amenable automated reasoning tools. In contrast to RSIM, ILC is intended to be the basis for a convenient and flexible programming language to which we can easily port existing UC pseudocode. The IITMs model [25, 26] has similar goals to ours, namely to provide a simple basis for UC. However, our approach is simpler, for example our type system subsumes their notion of *compatibility* and *connectability*, we avoid the distinctions of *net*, *external*, *io*, etc. tapes, and give a concrete programming model.

Simplified universal composability (SUC) [13] gives a simpler and restricted variant of the UC framework that is suitable for two-party multiparty computation tasks. The main difference from vanilla UC is that the set of parties is fixed, which greatly simplifies polynomial time reasoning and protocol composition while maintaining the same strong properties. We follow this in our execUC definition.

8 Conclusion and Future Work

The universal composability (UC) framework is widely used in cryptography for proofs. SaUCy takes a step towards mechanizing UC as a programming framework for constructing and analyzing large systems. We envision using SaUCy to tackle, for example, applications involving blockchains and smart contracts [16, 17, 33], which comprise an array of cryptography and distributed computing components and suffer from increasingly unwieldy formalisms.

We can view ILC typechecking of simulators in SaUCy as a partial mechanization of UC proofs, though the indistinguishability analysis is still on paper. Even partial mechanization is useful for catching bugs; we imagine using SaUCy to systematically implement functionalities and protocols from the literature and fuzz test them. Future work would be to embed ILC within a mechanized proof system, such as F* or EasyCrypt.

References

[1] Martín Abadi and Cédric Fournet. 2001. Mobile values, new names, and secure communication. In *ACM Sigplan Notices*, Vol. 36. ACM, 104–115.

[2] Martın Abadi and Andrew D Gordon. 1999. A calculus for cryptographic protocols: The spi calculus. *Information and computation* 148, 1 (1999), 1–70.

[3] Michael Backes, Birgit Pfitzmann, and Michael Waidner. 2007. The reactive simulatability (RSIM) framework for asynchronous systems. *Information and Computation* 205, 12 (2007), 1685–1720.

[4] G. Barthe, B. Grégoire, S. Heraud, and S. Béguelin. 2011. Computer-aided security proofs for the working cryptographer. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.

[5] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. *ACM SIGPLAN Notices* 44, 1 (2009), 90–101.

[6] Martin Berger, Kohei Honda, and Nobuko Yoshida. 2001. Sequentiality and the π-calculus. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 29–45.

[7] Bruno Blanchet. 2007. CryptoVerif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar "Formal Protocol Verification Applied*. 117.

[8] Bruno Blanchet, V Cheval, X Allamigeon, and B Smyth. 2010. Proverif: Cryptographic protocol verifier in the formal model. *URL http://prosecco. gforge. inria. fr/personal/bblanche/proverif* (2010).

[9] Florian Böhl and Dominique Unruh. 2016. Symbolic universal composability. *Journal of Computer Security* 24, 1 (2016), 1–38.

[10] Gilles Brassard, David Chaum, and Claude Crépeau. 1988. Minimum disclosure proofs of knowledge. *J. Comput. System Sci.* 37, 2 (1988), 156–189.
[11] Jan Camenisch, Robert R Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. 2016. Universal composition with responsive environments. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 807–840.
[12] R. Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*.
[13] Ran Canetti, Asaf Cohen, and Yehuda Lindell. 2015. A simpler variant of universally composable security for standard multiparty computation. In *Annual Cryptology Conference*. Springer, 3–22.
[14] Ran Canetti and Marc Fischlin. 2001. Universally composable commitments. In *Annual International Cryptology Conference*. Springer, 19–40.
[15] Ran Canetti and Tal Rabin. 2003. Universal composition with joint state. In *Annual International Cryptology Conference*. Springer, 265–281.
[16] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. [n. d.]. *Perun: Virtual payment channels over cryptographic currencies*. Technical Report.
[17] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. 2018. General State Channel Networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 949–966.
[18] Santiago Escobar, Catherine Meadows, and José Meseguer. 2009. Maude-NPA: Cryptographic protocol analysis modulo equational properties. In *Foundations of Security Analysis and Design V*. Springer, 1–50.
[19] Simon Fowler, Sam Lindley, J Garrett Morris, and Sára Decova. 2018. Session Types without Tiers. (2018).
[20] 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.
[21] Dennis Hofheinz and Victor Shoup. 2015. GNUC: A new universal composability framework. *Journal of Cryptology* 28, 3 (2015), 423–508.
[22] Jonathan Katz. 2007. Universally composable multi-party computation using tamper-proof hardware. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 115–128.
[23] Naoki Kobayashi, Benjamin C Pierce, and David N Turner. 1999. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 5 (1999), 914–947.
[24] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 839–858.
[25] Ralf Kusters. 2006. Simulation-based security with inexhaustible interactive turing machines. In *Computer Security Foundations Workshop, 2006. 19th IEEE*. IEEE, 12–pp.
[26] Ralf Küsters and Max Tuengerthal. 2009. Computational soundness for key exchange protocols with symmetric encryption. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 91–100.
[27] Peeter Laud. 2005. Secrecy types for a simulatable cryptographic library. In *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 26–35.
[28] Jeffrey R Lewis and Brad Martin. 2003. Cryptol: High assurance, retargetable crypto development and validation. In *Military Communications Conference, 2003. MILCOM'03. 2003 IEEE*, Vol. 2. IEEE, 820–825.
[29] Patrick Lincoln, John Mitchell, Mark Mitchell, and Andre Scedrov. 1998. A probabilistic poly-time framework for protocol analysis. In *Proceedings of the 5th ACM conference on Computer and communications security*. ACM, 112–121.
[30] Yehuda Lindell and Jonathan Katz. 2014. *Introduction to modern cryptography*. Chapman and Hall/CRC.
[31] Paulo Mateus, J Mitchell, and Andre Scedrov. 2003. Composition of cryptographic protocols in a probabilistic polynomial-time process calculus. In *International Conference on Concurrency Theory*. Springer, 327–349.
[32] Catherine Meadows. 1996. The NRL protocol analyzer: An overview. *The Journal of Logic Programming* 26, 2 (1996), 113–131.
[33] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. 2017. Sprites: Payment channels that go faster than lightning. *CoRR abs/1702.05812* (2017).
[34] Robin Milner. 1999. *Communicating and mobile systems: the pi calculus*. Cambridge university press.
[35] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, et al. 2016. Dependent types and multi-monadic effects in F*. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

A ILC Metatheory Attempt

Lemma A.1 (Canonical forms). *If*

$$\Delta_{in}; \Gamma; \Omega \vdash v : U; \Delta_{out}$$

then

1. *If U is $\mathbb{1}$, then $v = ()$.*
2. *If U is $A \times B$, then $v = (v_1, v_2)_\infty$ where*
$$\Delta_{in}; \Gamma; \Omega \vdash v_1 : A; \Delta_{out}$$
and
$$\Delta_{in}; \Gamma; \Omega \vdash v_2 : B; \Delta_{out}.$$
3. *If U is $A_1 + A_2$, then $v = inj^i_\pi(v_i)$ where $i \in \{1, 2\}$ and*
$$\Delta_{in}; \Gamma; \Omega \vdash v_i : A_i; \Delta_{out}.$$
4. *If U is $V_1 \to_\pi V_2$, then $v = \lambda_\pi x.\, e$ where*
$$\Delta_{in}; \Gamma; \Omega \vdash x : V_1; \Delta_{out}$$
and
$$\Delta_{in}; \Gamma; \Omega \vdash e : V_2; \Delta_{out}.$$
5. *If U is $Wr\, S$, then $v =$* TODO: .
6. *If U is $Ref\, A$, then $v = ref(v')$ where*
$$\Delta_{in}; \Gamma; \Omega \vdash v' : A; \Delta_{out}.$$
7. *If U is $Rd\, S$, then $v =$* TODO: .
8. *If U is $!\, A$, then $v =$* TODO: .
9. *If U is $X \otimes Y$, then $v = (v_1, v_2)_1$ where*
$$\Delta_{in}; \Gamma; \Omega \vdash v_1 : X; \Delta_{out}$$
and
$$\Delta_{in}; \Gamma; \Omega \vdash v_2 : Y; \Delta_{out}.$$

Theorem A.2 (Local progress). *If*

$$\Delta_{in}; \Gamma; \Omega \vdash e : U; \Delta_{out}$$

$$\Gamma; \Omega \vdash \sigma$$

*then either e **lterm** or there exists $\sigma'; e'$ such that*

$$\sigma; e \longrightarrow \sigma'; e'.$$

Proof. By induction on the derivation of e.

- **Case** $\dfrac{}{\Delta; \Gamma, x : A \vdash x : A; \Delta}\ \text{var}$

□

Theorem A.3 (Local preservation). *If*

$$\Delta_{in}; \Gamma; \Omega \vdash e : U; \Delta_{out}$$
$$\Gamma; \Omega \vdash \sigma$$
$$\sigma; e \to \sigma'; e'$$

then there exists store typing extension $\Omega' \supseteq \Omega$ *such that*

$$\Delta_{in}; \Gamma; \Omega' \vdash e' : U; \Delta_{out}$$
$$\Gamma; \Omega'; \vdash \sigma'.$$

B ILC meta theory

B.1 Additional syntax

We define syntax for store, process and channel typings, which each map a kind of identifier (location, process name, or channel name) to its associated type:

Store typings	$\Omega ::= \cdot \mid \Omega, \ell : A$
Process pool typings	$\Phi ::= \cdot \mid \Phi, p : A \mid \Phi, p : X$
Channel pool typings	$\Psi ::= \cdot \mid \Psi, c : A \mid \Psi, c : X$

B.2 Configuration typings

Using the syntax above, we define configuration typing as a straightforward extension of single-process typing, given in Sec. 3.2:

$\boxed{\Omega; \Psi \vdash C : \Phi \rhd m}$ Configuration C is well-typed

$$\frac{\Omega; \Psi \vdash \sigma : \Omega}{\Omega; \Psi \vdash \langle \Sigma; \sigma; \varepsilon \rangle : \cdot \rhd \mathsf{V}}\ \text{empty}$$

$$\frac{\begin{array}{c}\Omega; \Psi \vdash e : X \rhd m_1 \\ \Omega; \Psi \vdash \langle \Sigma; \sigma; \pi \rangle : \Phi \rhd m_2 \qquad m_1 \parallel m_2 \Rightarrow m_3\end{array}}{\Omega; \Psi \vdash \langle \Sigma; \sigma; \pi, p : e \rangle : \Phi, p : X \rhd m_3}\ \text{aff}$$

$$\frac{\begin{array}{c}\Omega; \Psi \vdash e : A \rhd m_1 \\ \Omega; \Psi \vdash \langle \Sigma; \sigma; \pi \rangle : \Phi \rhd m_2 \qquad m_1 \parallel m_2 \Rightarrow m_3\end{array}}{\Omega; \Psi \vdash \langle \Sigma; \sigma; \pi, p : e \rangle : \Phi, p : A \rhd m_3}\ \text{int}$$

Notably, parallel processess with modes m_1 and m_2 must have modes that compose, where $m_1 \parallel m_2$ is defined; this invariant ensures that there is at most one write mode process in a well-typed pool.

Because of the premise in the base case rule (empty), all well-typed configurations consist of well-typed stores. We define store typing in the standard way:

$\boxed{\Omega; \Psi \vdash \sigma : \Omega}$ Under Ω and Ψ, store σ is well-typed

$$\frac{}{\Omega; \Psi \vdash \cdot : \cdot}\ \text{empty} \qquad \frac{\Omega; \Psi \vdash v : A \qquad \Omega; \Psi \vdash \sigma : \Omega}{\Omega; \Psi \vdash (\sigma, \ell : v) : (\Omega, \ell : A)}\ \text{loc}$$

B.3 Type soundness

Lemma B.1 (Non-progress). *For all configurations C, store typings Ω, channel typings Ψ, and process typings Φ, if $\Omega; \Psi \vdash C : \Phi$ and C **term**, then $\nexists C'$ such that $C \longrightarrow C'$.*

Theorem B.2 (Progress). *For all configurations C, store typings Ω, channel typings Ψ, and process typings Φ, if $\Omega; \Psi \vdash C : \Phi \rhd m$ then either C **term**, or $\exists C'$ such that $C \longrightarrow C'$.*

Theorem B.3 (Preservation). *For all configurations C, C', store typings Ω, channel typings Ψ and process typings Φ, if $\Omega; \Psi \vdash C : \Phi \rhd m$ and $C \longrightarrow C'$ then there exists store typing extension $\Omega' \supseteq \Omega$ channel typing extension $\Psi' \supseteq \Psi$ and process typing extension $\Phi' \supseteq \Phi$ such that $\Omega'; \Psi' \vdash C' : \Phi' \rhd m'$.*

B.4 Confluence

The following lemmas state structural invariants over write effects and read channels of a well-typed configuration: no two writes are in parallel composition, and every read channel end is a non-duplicable (affine) resource.

Lemma B.4 (Unique writer process). *If C is a well-typed configuration with process pool π, then there exists at most one write-mode process in π.*

Proof. By structural induction over the typing derivation for C. □

Lemma B.5 (Unique reader process). *If C is a well-typed configuration with process pool π, and c is a read channel in this configuration, then there exists at most one process in π where c appears.*

Proof. By structural induction over the typing derivation for C. □

Theorem B.6 (Single-step confluence). *For all well-typed configurations C, if $C \longrightarrow C_1$ and $C \longrightarrow C_2$ then there exists renaming function f such that either:*

1. *$C_1 = f(C_2)$, or*
2. *$\exists C_3$ such that $C_1 \longrightarrow C_3$ and $f(C_2) \longrightarrow C_3$.*

Proof. By induction on the pair of steps $\langle C \longrightarrow C_1, C \longrightarrow C_2 \rangle$.

We consider the following cases:

Case: congruence: if either step uses congr, we apply the inductive hypothesis.

Case: independent processes: the two steps advance distinct processes, using any of the rules local, fork and nu, we produce C_3 by combining those two (independent) steps.

Case: one process: the two steps advance the same process, we show that this was determinisitic (up to naming) by constructing the naming function f such that $C_2 = f(C_1)$. Most cases are straightforward since they perform no non-determinisitic choices. The only source of non-determinism is the name choices, in rules nu, fork, and new; in each case, we map the name choice from the second step to that of the first step.

Case: read-write interaction: In the case that either step uses rw, we rely on lemmas B.4 and B.5. to show that both steps use rw, and that the reader-writer process pair is unique.

□

By composing multiple uses of this theorem we prove multi-step confluence. However, to carry forth this composition, we need a more general notion of single-step confluence, which is parameteric in a renaming function for the initial configurations.

Theorem B.7 (Single-step confluence, generalized). *For all well-typed configurations C and renaming functions f, if $C \longrightarrow C_1$ and $f(C) \longrightarrow C_2$ then there exists renaming function g such that either:*

1. *$C_1 = g(C_2)$, or*
2. *$\exists C_3$ such that $C_1 \longrightarrow C_3$ and $g(C_2) \longrightarrow C_3$.*

Proof. (Analogous to first single-step confluence proof.) □

We prove a full confluence theorem that is generalized similarly, by accepting a renaming function f to produce a new one g:

Theorem B.8 (Full confluence). *For all well-typed configurations C, and renaming functions f, if $C \longrightarrow^* C_1$ and $f(C) \longrightarrow^* C_2$ and C_1 **term** and C_2 **term** then there exists renaming function g such that $C_1 = g(C_2)$.*

Proof. By induction on the reduction sequence pair $\langle C \longrightarrow^* C_1, f(C) \longrightarrow^* C_2 \rangle$. Because of single-step confluence, we know that if if either reduction sequence is empty, then the other must be empty, and that if either takes a step, the other must take a step.

Empty case: When empty, we have the resulting renaming function g via single-step confluence.

Step case: We consider the case where each reduction consists of at least one step: $C \longrightarrow C_1'$ and $C_1' \longrightarrow^* C_1$ and $f(C) \longrightarrow C_2'$ and $C_2' \longrightarrow^* C_2$. By single-step confluence, we have that there exists g_0 such that $g_0(C_2') = C_1'$. By the inductive hypothesis, we have that there exists g such that $C_1 = g(C_2)$. □

C ILC Implementation of execUC

The full definition of the UC execution experiment is given in Figure 9.

```
execUC :: ∀ ... . X_z ⊸ X_p ⊗ X_q ⊸ X_f ⊸ X_a ⊸ ! Nat ⊸ ! [Bit] ⊸ ! Crupt ⊸_m ! Bit
let execUC z (p,q) f a k !r crupt =
  ν (rZ2P, wZ2P), (rP2Z, wP2Z)
  , (rZ2Q, wZ2Q), (rQ2Z, wQ2Z)
  , (rP2F, wP2F), (rF2P, wF2P)
  , (rQ2F, wQ2F), (rF2Q, wF2Q)
  , (rF2A, wF2A), (rA2F, wA2F)
  , (rA2Z, wA2Z), (rZ2A, wZ2A)
  , (rP2A, wP2A), (rA2P, wA2P)
  , (rQ2A, wQ2A), (rA2Q, wA2Q)
  , (rP2Q, wP2Q), (rQ2P, wQ2P)
  . let wrapperP p !crupt toZ frZ toF frF toA frA toQ frQ k bits =
      if crupt == CruptP then
           loop frZ !(λ _ . error "Z cannot write to corrupted party.")
         |▷ fwd frF toA
         |▷ fwd frP toA
         |▷ fwd frA toF
      else
         p toZ frZ toF  frF toQ frQ k bits in
    let wrapperQ p !crupt toZ frZ toF frF toA frA toP frP k bits =
      if crupt == CruptQ then
           loop frZ !(λ _ . error "Z cannot write to corrupted party.")
         |▷ fwd frF toA
         |▷ fwd frP toA
         |▷ fwd frA toF
      else
         p toZ frZ toF  frF toP frP k bits in
    let (rf,ra,rp,rq,rz) = splitBits5 r in
       f crupt !wF2P rP2F !wF2Q rQ2F !wF2A rA2F k !rf
    |▷ a crupt !wA2Z rZ2A !wA2F rF2A !wA2P rP2A !wA2Q rQ2A k !ra
    |▷ wrapperP p crupt !wP2Z rZ2P !wP2F rF2P !wP2A rA2P !wP2Q rQ2P k !rp
    |▷ wrapperQ q crupt !wQ2Z rZ2Q !wQ2F rF2Q !wQ2A rA2Q !wQ2P rP2Q k !rq
    |▷ z !wZ2P rP2Z !wZ2Q rQ2Z !wZ2A rA2Z k !rz
```

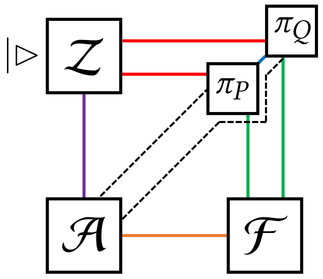


Figure 9. Full definition of execUC. The channels follow a uniform naming scheme. The read end of a channel is prefixed with r- and the write end of a channel is prefixed with w-. The channel rZ2P denotes the read end of communications from the environment z to the party p. First, the random bitstring is split amongst each of the five parties. Then, the functionality, the adversary, and both protocol parties are spawned in a child process (given the appropriate channels and parameters), and the process continues as the environment process. Notice that parties are run in wrapper functions, which alter their behavior depending on whether or not they are corrupted. If a party is corrupted, then the adversary masquerades as the party. The mode carried over the rightmost lollipop is derived as $\{(m_f, m_a, m_z) \mid m_f \parallel (m_a \parallel (R \parallel (R \parallel m_z))) \Rightarrow m\}$.

D Extending ILC with Trapdoor Permutations

UC Commitments are realized from cryptographic primitives, such as trapdoor permutations, which require extensions to ILC. The new syntactic forms are kgen, tdp, inv, and hc with the static and dynamic semantics shown in Figure 10. The semantics are written in terms of the cryptographic objects themselves.

The key generation function keygen takes as input a random bitstring and outputs a random public key v_{pk} and a trapdoor v_{td}. The trapdoor permutation function tdp takes as inputs a key v_{pk} and a bitstring v_{in} and outputs a bitstring v_{out}. The inv function takes as inputs a key-trapdoor pair (v_{pk}, v_{td}) and a bitstring v_{in} and outputs a bitstring v_{out}. The hardcore predicate function hc takes as input a key v_{pk} and outputs a single bit.

We can use these to implement a special pseudorandom number generator $G_{pk}\colon \{0,1\}^k \to \{0,1\}^{4k}$ that has a trapdoor property, i.e., it is easy to compute, but difficult to invert except with special information called the “trapdoor.”

$$G_{pk}(r) = \left(\mathbf{f}_{pk}^{(3n)}(r), \mathbf{B}(\mathbf{f}_{pk}^{(3n-1)}(r)), \ldots, \mathbf{B}(\mathbf{f}_{pk}(r)), \mathbf{B}(r)\right)$$

Here, $\mathbf{f}_{pk}$ is a trapdoor permutation over $\{0,1\}^k$, with $\mathbf{f}_{pk}^{(i)}(r)$ denoting the i^{th}-fold application of $\mathbf{f}_{pk}$, and $\mathbf{B}$ is a hardcore predicate for $\mathbf{f}_{pk}$. In ILC, this can be implemented as:

```
iterate :: ∀ a . Int → (a → a) → a → a
prg :: [Bit] → [Bit] → Nat → [Bit]
let prg pk r k =
  letrec aux j =
    if j ≤ 0 then [hc r]
    else hc (iterate j (tdp pk) r) : aux pk r (j − 1) in
  iterate (3 ∗ k) (tdp pk) r ++ aux pk r (3 ∗ k − 1)
```

E Universally Composable Commitment Protocol

In this section we give the full elaboration of our UC commitment instantiation. The specification functionality is given in the body in Figure 1, along with the protocol implementation in Section 6.5. Our development follows closely from the psuedocode in the UC literature [14], which we show here in Algorithm 1. The protocol relies on the CRS functionality which we define here in Figure 13. To briefly summarize what is going: the setup CRS samples a random string σ and two trapdoor pseudorandom generators (prgs $\mathsf{pk}_0, \mathsf{pk}_1$). To commit to the bit b, the commiter produces a string y that is the result of applying one or the other of the prgs, and if $b = 1$ additionally applying xor with σ. The intuitive explanation why this is hiding is that without the trapdoor, it is difficult to tell whether a random $4k$-bit string is in the range of either prg. To open the commitment, the committer simply reveals the preimage and the receiver checks which of the two cases applies. The intuitive explanation why this is binding is that it is difficult to find a pair $y, y \oplus \sigma$ that are respectively in the range of both prgs.

The UC proof consists of two simulators, one for the ideal world and one for the real world. The ideal world simulator, given in Figure 15 is ported directly from the UC literature [14], while the non-standard real world simulator, given in Figure 16, is required because our protocol emulation definition requires simulation in both directions. The key to the ideal world simulator is to allow the simulator to generate its own “fake” CRS, for which it stores the trapdoors. The string σ is not truly random, but instead is the result of combining two evaluations of the prgs. The ideal world simulator consists of two cases, depending on which of the parties is corrupt.

In the case that the committer P is corrupt, the simulator needs to be able to *extract* the committed value. The simulator is activated when $\mathcal{Z}$ sends a message (Commit′ y); in the real world, this is relayed by the dummy adversary to Q, who outputs Committed back to the environment. Hence to achieve the same effect in the ideal word, the simulator must send (Commit b) to $\mathcal{F}_{\text{COM}}$. To extract b from y, the simulator makes use of the prg trapdoor check which one has y in its range. It is necessary to argue by cryptographic reduction that this simulation is sound. To show this, we would define an alternative execution where the prg is substituted for a truly random function (i.e., a random oracle). If an environment $\mathcal{Z}$ could distinguish between these two worlds, then we could adapt the execution to distinguish the prg from random, violating the prg assumption.

In the case that the receiver Q is corrupt, the simulator needs to *equivocate*. The simulator is activated when $\mathcal{Z}$ inputs (Commit b) to P, after which $\mathcal{F}_{\text{COM}}$ sends Committed to the simulator. In the real world, the environment receives a commitment message (Commit′ y) from corrupted Q for some seemingly-random y. To achieve the same effect, the simulator must choose y. However, the simulator is next activated when the $\mathcal{Z}$ inputs (Open b) to P, after which the simulator learns b from $\mathcal{F}_{\text{COM}}$. However, in the real world the environment receives a valid opening (Opened′ b r) that is consistent with y and with the value chosen by the environment. Thus the simulator must initially choose y so that it can later be opened to either value b may take. The simulator achieves this by choosing σ and y ahead of time while generating the fake CRS. The reduction step is the same, and involves replacing prg with a true random function.

Recall that the motivation for the real world simulator is to rule out degenerate protocols that diverge in some way. For every well behaved environment such that the ideal world is PPT, we need to demonstrate an adversary in the real world that is also PPT. Fortunately, the real world simulator, shown in Figure 16 is much simpler than ideal world simulator. Essentially the simulator runs a copy of the honest protocol

$$\text{Expressions} \quad e ::= \mathsf{kgen}(e) \mid \mathsf{tdp}(e_1, e_2) \mid \mathsf{inv}(e_1, e_2) \mid \mathsf{hc}(e)$$

$\boxed{\Delta; \Gamma \vdash e : A \rhd m}$ Under Δ and Γ, expression e has intuitionistic type A and mode m.

$$\frac{\Delta; \Gamma \vdash e : [\mathsf{Bit}]}{\Delta; \Gamma \vdash \mathsf{kgen}(e) : ([\mathsf{Bit}], [\mathsf{Bit}])}\ \text{kgen} \qquad \frac{\Delta_1; \Gamma \vdash e_1 : [\mathsf{Bit}] \qquad \Delta_2; \Gamma \vdash e_2 : [\mathsf{Bit}]}{\Delta_1, \Delta_2; \Gamma \vdash \mathsf{tdp}(e_1, e_2) : [\mathsf{Bit}]}\ \text{tdp}$$

$$\frac{\Delta_1; \Gamma \vdash e_1 : [\mathsf{Bit}] \times_\pi [\mathsf{Bit}] \qquad \Delta_2; \Gamma \vdash e_2 : [\mathsf{Bit}]}{\Delta_1, \Delta_2; \Gamma \vdash \mathsf{inv}(e_1, e_2) : [\mathsf{Bit}]}\ \text{inv} \qquad \frac{\Delta; \Gamma \vdash e : [\mathsf{Bit}] \to \mathsf{Bit}}{\Delta; \Gamma \vdash \mathsf{hc}(e) : \mathsf{Bit}}\ \text{hc}$$

$\boxed{\sigma_1; e_1 \longrightarrow \sigma_2; e_2}$ Under store σ_1, expression e_1 reduces to $\sigma_2; e_2$.

$$\frac{\mathbf{Gen}(v_r) = (v_{pk}, v_{td}) \qquad v_{pk}, v_{td} \in \{0,1\}^k}{\sigma; \mathsf{kgen}(v_r) \longrightarrow \sigma; (v_{pk}, v_{td})}\ \text{kgen} \qquad \frac{\mathbf{f}(v_{pk}, v_{in}) = v_{out} \qquad \mathbf{f}\colon \{0,1\}^k \to \{0,1\}^k \to \{0,1\}^k}{\sigma; \mathsf{tdp}(v_{pk}, v_{in}) \longrightarrow \sigma; v_{out}}\ \text{tdp}$$

$$\frac{\mathbf{Inv}((v_{pk}, v_{td}), v_{in}) = v_{out} \qquad \mathbf{Inv}\colon \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^k \to \{0,1\}^k}{\sigma; \mathsf{inv}((v_{pk}, v_{td}), v_{in}) \longrightarrow \sigma; v_{out}}\ \text{inv} \qquad \frac{\mathbf{B}(v_{pk}) = v \qquad \mathbf{B}\colon \{0,1\}^k \to \{0,1\}}{\sigma; \mathsf{hc}(v_{pk}) \longrightarrow \sigma; v}\ \text{hc}$$

Figure 10. Extending ILC with trapdoor permutations. The semantics are parameterized by a security parameter k.

for each of the corrupted parties. The simulation that results in this case is identical.

Protocol 1: Universally Composable Commitment

```
1  Public strings:
2      $\sigma$: Random string in $\{0,1\}^{4n}$
3      $pk_0, pk_1$: Keys for generator $G_k\colon \{0,1\}^n \to \{0,1\}^{4n}$
4  Commit($b$):
5      $r \leftarrow \{0,1\}^n$
6      $y := G_{pk_b}(r)$
7      if $b = 1$ then $y := y \oplus \sigma$
8      Send (Commit, $y$) to receiver.
9      Upon receiving (Commit, $y$) from $A$, $B$ outputs
         (Receipt).
10 Decommit($x$):
11     Send $(b, r)$ to receiver.
12     Receiver checks $y = G_{pk_b}(r)$ for $b = 0$, or
         $y = G_{pk_b}(r) \oplus \sigma$ for $b = 1$. If verification succeeds,
         then $B$ outputs (Open, $b$).
```

```
dummy :: ∀ a ... . ! Crupt ⊸ ··· ⊸ ! Nat ⊸ ! [Bit] ⊸ a
let dummyA !crupt !toZ frZ !toF frF !toP frP toQ frQ !toQasP toPasQ k r =
  let fwd2Z c = loop c !(λ m . wr (X2Z m) → toZ) in
       loop frZ !(λ x . match x with
                    | A2F m ⇒ wr m → toF
                    | A2P m ⇒ if crupt == CruptP
                              then wr m → toQasP
                              else wr m → toP)
    |▷ fwd2Z frF
    |▷ fwd2Z frP
    |▷ fwd2Z frQ
```

Figure 11. Dummy adversary. The dummy adversary forwards messages from the environment to either the functionality (if the message has constructor A2F) or the party p (if the message has constructor A2P). Similarly, the dummy adversary forwards messages from the functionality or the procotol parties to the environment.

```
dummyP :: ∀ a b ... . ! Wr a ⊸ ··· ⊸ ! Nat ⊸ ! [Bit] ⊸ b
let dummyP toZ frZ toF frF k r = fwd frZ toF |▷ fwd frF toZ
```

Figure 12. Dummy party. The dummy party simply relays information between the environment and the functionality.

```
fCrs :: ∀ ... . ! Crupt ⊸ ··· ⊸ ! Nat ⊸ ! [Bit]
let fCrs crupt !toP frP !toQ frQ !toA frA !k !bits =
  let rbits = ref bits in
  let pubref = ref Nothing in
  let replyCrs fr to = loop fr !(match @pubref with
                                 | Nothing ⇒ let σ = sample (4*k) rbits in
                                             let pk_0 = kgen k (sample k rbits) in
                                             let pk_1 = kgen k (sample k rbits) in
                                             let pub = (σ, pk_0,pk_1) in
                                               set pubref := Just pub ; wr pub → to
                                 | Just pub ⇒ wr pub → to) in
      replyCrs frP toP
    |▷ replyCrs frQ toQ
    |▷ replyCrs frA toA
```

Figure 13. Ideal functionality for common reference string.

```
fCom :: ∀ ... . ! Crupt ⊸ ··· ⊸ ! Nat ⊸_R ! 𝟙
let fCom crupt toP frP !toQ frQ toA frA k bits =
  let ⟨!(Commit b), frP⟩ = rd frP in
    wr Receipt → toQ ;
    let ⟨!Open, frP⟩ = rd frP in
      !(wr (Opened b) → toQ)
```

Figure 14. Ideal functionality for one-time bit commitment.

```
let simI !crupt !toZ frZ !toF frF !toP frP !toQ frQ !k !bits =
  let rbits = ref bits in
  let ($pk_0$,$td_0$) = kgen k in
  let ($pk_1$,$td_1$) = kgen k in
  let $r_0$ = sample k rbits in
  let $r_1$ = sample k rbits in
  let $\sigma$ = xors (prg $pk_0$ $r_0$) (prg $pk_1$ $r_1$) in
    match crupt with
    | CruptP ⇒
      let ⟨!GetCRS, frZ⟩ = rd frZ in
        wr (X2Z (PublicStrings $\sigma$ $pk_0$ $pk_1$)) →toZ ;
        let ⟨!(A2P (Commit' y)), frZ⟩ = rd frZ in
          if check $td_0$ $pk_0$ y then
            wr (Commit 0) → toP
          else
            if check $td_1$ $pk_1$ (xors y $\sigma$) then
              wr (Commit 1) → toP
            else error "Fail" ;
          let ⟨!(A2P (Open' b r)), frZ⟩ = rd frZ in
            if b == 0 && y == prg $pk_0$ r ||
              b == 1 && y == xors (prg $pk_1$ r) $\sigma$
            then !(wr Open → toP)
            else error "Fail"
    | CruptQ ⇒
      let ⟨!GetCRS, frZ⟩ = rd frZ in
        wr (X2Z (PublicStrings $\sigma$ $pk_0$ $pk_1$)) → toZ ;
        let ⟨!Receipt, frQ⟩ = rd frQ in
        let y = prg $pk_0$ $r_0$ in
          wr (X2Z (Commit' y)) → toZ ;
          let ⟨!(Opened b'), frQ⟩ = rd frQ in
            if (b' == 0) then
              !(wr (X2Z (Opened' $r_0$)) → toZ)
            else
              !(wr (X2Z (Opened' $r_1$)) → toZ)
    | CruptNone ⇒ error "Fail"
```

Figure 15. Ideal world simulator for UC commitment.

```
let simR !crupt !toZ frZ !toF frF toP frP !toQ frQ !k !bits =
  match crupt with
  | CruptP ⇒
    let ⟨!(Commit b), frZ⟩ = rd frZ in
      wr GetCRS → toF ;
      let ⟨!(PublicStrings σ pk_0 pk_1), frF⟩ = rd frF in
      let r = take k bits in
      let y = if b == 0 then prg pk_0 r else xors (prg pk_0 r) σ in
        wr (Commit' y) → toQ ;
        let ⟨!(Open), frZ⟩= rd frZ in
          !(wr (Open' b r) → toQ)
  | CruptQ ⇒
    let ⟨!(Commit' y), frQ⟩ = rd frQ in
      wr Receipt → toZ ;
      let ⟨!(Open' b r), frQ⟩ = rd frQ in
        !(wr (Opened b) → toZ)
  | CruptNone ⇒ error "Fail"
```

Figure 16. Real world simulator for UC commitment.