ILC: A Calculus for Composable, Computational Cryptography

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

Matthew A. Hammer
DFINITY, USA
matthew@dfinity.org

Andrew Miller
University of Illinois
Urbana-Champaign, USA
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 an affine typing discipline. 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 a simplified implementation of UC called SaUCy.

CCS Concepts • **Security and privacy → Formal security models**.

Keywords Provable security, universal composability, process calculus, type systems

ACM Reference Format:
Kevin Liao, Matthew A. Hammer, and Andrew Miller. 2019. ILC: A Calculus for Composable, Computational Cryptography. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314607

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314607

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" [16]. 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 [37] or under sequential composition [25]. Thus, the benefit of using UC is *modularity*—it supports analyzing complex protocols by composing simpler building blocks. However, the cost of using 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.

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) [10], 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 (concurrent) 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 models of concurrency would frustrate such reduction proofs. ITMs sidestep 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, *the order of activations is fully determined.*

In this paper, we take up the challenge of faithfully capturing these idioms by designing a new process calculus called the Interactive Lambda Calculus (ILC), which adapts

ITMs to a subset of the π-calculus [43] through an affine typing discipline. In other words, *well-typed ILC programs are expressible as ITMs.* We then use ILC to build a concrete, executable implementation of a simplified UC framework, dubbed SaUCy.

1.1 Interactive Lambda Calculus

Why do we need another process calculus in the first place? Where do existing ones fall short? On the one hand, process calculi such as the π-calculus [43] and its cryptography-oriented variants [1, 2, 35] are not a good fit to ITMs, since they permit non-confluent reductions by design (i.e., non-probabilistic nondeterminism). On the other hand, various other calculi that do enjoy confluence are overly restrictive, only allowing for fixed or two-party communications [10, 24, 29].

ILC fills this gap by adapting ITMs to a subset of the π-calculus through an affine typing discipline. To maintain that only one process is active (can write) at any given time, processes implicitly pass around an affine "write token" by virtue of where they perform read and write effects: When process A writes to process B, process A "spends" the write token and process B "earns" the write token. Moreover, to maintain that the order of activations is fully determined, the read endpoints of channels are (non-duplicable) affine resources, and so each write operation corresponds to a single, unique read operation. Together, these give ILC its central metatheoretic property of *confluence*.

The importance of confluence is that 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. This eliminates non-probabilistic nondeterminism, and so ILC programs are amenable to the reasoning patterns necessary for establishing computational security guarantees.

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 a simplified 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 [19], and an examination of a subtle definitional issue involving reentrant concurrency [14].

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

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

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

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

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

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

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

2.2 ILC by Example

We make the above more concrete by running through an example of *commitment*, an essential building block in many

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

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

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

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

cryptographic protocols [11]. 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 a simplified ideal functionality for one-time bit commitment, $\mathcal{F}_{\text{COM}}$, as it would appear in the cryptography literature [19]. 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 implement a simplified version of $\mathcal{F}_{\text{COM}}$ in ILC to highlight some key features of the language. The function fCom takes two channel endpoints as arguments. The first is a write endpoint toQ : Wr Msg (for sending messages of type Msg to Q), and the second is a read endpoint frP : Rd Msg (for receiving messages of type Msg from P). At a high level, it should be clear how the communication pattern in fCom follows that in $\mathcal{F}_{\text{COM}}$, but there are a few details that require further explanation. These details are better explained in the context of ILC's type system, which we give a quick tour of next.

2.3 ILC Type System

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

- *Only one process is active at any given time.* Processes implicitly pass around an affine "write token" ⓦ by virtue of where they perform read and write effects. In order for process A to write to process B, process A must first own the write token. Because the write token is unique, at most one process owns the write token ("is active" or "can write") at any given time. When process B reads the message from A, process B earns the write token, thereby conserving its uniqueness and now allowing process B to write to some other process.
- *The order of activations is deterministic.* Each channel (or "tape" in ITM parlance) has a read endpoint and a write endpoint. The read endpoint is an affine resource, and so it is owned by at most one process. This ensures that each write operation corresponds to a single, unique read operation.

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

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

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

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

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

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

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

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

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

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

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

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

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

Revisiting fCom. Having gone through several typing rules, we now revisit fCom from Figure 1 (right). In particular, we should convince ourselves that fCom respects the invariants of the type system.

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

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

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 is given in Figure 2. Types (written U, V) are bifurcated into unrestricted types (written A, B) and affine types (written X, Y).

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

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

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

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

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

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

Figure 2. ILC Syntax.

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

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

3.2 Static Semantics

The typing rules of ILC are given in Figure 3. An algorithmic version of the rules appears in the online appendix [34].

To recap, the typing rules maintain that only one process is active at any given time (unique ownership of the write token), and the order of activations is deterministic (unique ownership of read endpoints). We read the typing judgement $\Delta; \Gamma \vdash e : U$ as "under affine context Δ and unrestricted context Γ, expression e has type U." In full detail, the typing judgement also includes a typing context Ψ, which maps channel names d to sendable types S. However, it is only used in two special rules for typing channel endpoints that do not arise for source level programs, but will be needed to typecheck a running program that has performed channel allocation:

$$\frac{\Psi(d) = S}{\Psi; \Delta; \Gamma \vdash \mathsf{Read}(d) : \mathsf{Rd}\, S}\ \text{rdend}$$

$$\frac{\Psi(d) = S}{\Psi; \Delta; \Gamma \vdash \mathsf{Write}(d) : \mathsf{Wr}\, S}\ \text{wrend}$$

This pair of rules establish the canonical forms for the types of channel endpoints, $\mathsf{Rd}\, S$ and $\mathsf{Wr}\, S$. We use the metavariable c to range over these two canonical forms.

The typing rules for the functional fragment of ILC are fairly standard, except that they now have unrestricted and affine variants (and for some, sendable variants).

The rule for unrestricted abstraction (uabs) extends the unrestricted context Γ with $x : A$ before checking the body e of the abstraction. Notice that because unrestricted abstractions can be duplicated, the body must be affinely closed (cannot contain free affine variables).

The rule for write abstraction (wabs) is similar to uabs. The only difference is that wabs extends the affine context with the write token before checking the body e of the abstraction. Dually, the write application rule (wapp) stipulates that a process must own the write token in order to apply a write abstraction.

The rule for affine abstraction (aabs) is analagous to uabs, but notice that the body *need not* be affinely closed, since affine abstractions cannot be duplicated. It turns out that most affine functions we write *are* affinely closed, and so such a function $f : X \multimap U$ can be made into an unrestricted function $g : A \to X \multimap U$ by adding a leading unrestricted argument.

The bang rule turns an unrestrictedly typed expression $e : A$ into an affinely typed expression $e : !\, A$. Dually, the gnab rule turns an affinely typed expression $e : !\, A$ into an unrestrictedly typed expression $e : A$.

The typing rules for fork, write, and read were covered in Section 2.3, so this leaves channel restriction (nu) and external choice (choice) as the remaining typing rules related to communication and concurrency.

$\boxed{\Delta;\Gamma \vdash e : U}$ Under affine context Δ and unrestricted context Γ, expression e has type U.

$$\frac{\Gamma(x) = A}{\Delta;\Gamma \vdash x : A}\ \text{uvar} \qquad \frac{\Delta(x) = X}{\Delta;\Gamma \vdash x : X}\ \text{avar} \qquad \frac{}{\Delta;\Gamma \vdash () : \mathbb{1}}\ \text{unit} \qquad \frac{\Delta_1;\Gamma \vdash e_1 : A_1 \qquad \Delta_2;\Gamma \vdash e_2 : A_2}{\Delta_1,\Delta_2;\Gamma \vdash (e_1,e_2)_\infty : A_1 \times A_2}\ \text{upair}$$

$$\frac{\Delta_1;\Gamma \vdash e_1 : S_1 \qquad \Delta_2;\Gamma \vdash e_2 : S_2}{\Delta_1,\Delta_2;\Gamma \vdash (e_1,e_2)_\mathsf{w} : S_1 \times S_2}\ \text{spair} \qquad \frac{\Delta_1;\Gamma \vdash e_1 : X_1 \qquad \Delta_2;\Gamma \vdash e_2 : X_2}{\Delta_1,\Delta_2;\Gamma \vdash (e_1,e_2)_1 : X_1 \otimes X_2}\ \text{apair} \qquad \frac{i \in \{1,2\} \qquad \Delta;\Gamma \vdash e : A_i}{\Delta;\Gamma \vdash \mathsf{inj}^i_\infty(e) : A_1 + A_2}\ \text{uinj}$$

$$\frac{i \in \{1,2\} \qquad \Delta;\Gamma \vdash e : S_i}{\Delta;\Gamma \vdash \mathsf{inj}^i_\mathsf{w}(e) : S_1 + S_2}\ \text{sinj} \qquad \frac{i \in \{1,2\} \qquad \Delta;\Gamma \vdash e : X_i}{\Delta;\Gamma \vdash \mathsf{inj}^i_1(e) : X_1 \oplus X_2}\ \text{ainj} \qquad \frac{\begin{array}{c}\Delta_1;\Gamma \vdash e_1 : A_1 \times A_2 \\ \Delta_2;\Gamma, x_1 : A_1, x_2 : A_2 \vdash e : U\end{array}}{\Delta_1,\Delta_2;\Gamma \vdash \mathsf{split}_\infty(e_1, x_1.x_2.e_2) : U}\ \text{usplit}$$

$$\frac{\begin{array}{c}\Delta_1;\Gamma \vdash e_1 : S_1 \times S_2 \\ \Delta_2;\Gamma, x_1 : S_1, x_2 : S_2 \vdash e : U\end{array}}{\Delta_1,\Delta_2;\Gamma \vdash \mathsf{split}_\mathsf{w}(e_1, x_1.x_2.e_2) : U}\ \text{ssplit} \qquad \frac{\begin{array}{c}\Delta_1;\Gamma \vdash e_1 : X_1 \otimes X_2 \\ \Delta_2, x_1 : X_1, x_2 : X_2;\Gamma \vdash e : U\end{array}}{\Delta_1,\Delta_2;\Gamma \vdash \mathsf{split}_1(e_1, x_1.x_2.e_2) : U}\ \text{asplit}$$

$$\frac{\begin{array}{c}\Delta_1;\Gamma \vdash e : A_1 + A_2 \\ \Delta_2;\Gamma, x_1 : A_1 \vdash e_1 : U \qquad \Delta_2;\Gamma, x_2 : A_2 \vdash e_2 : U\end{array}}{\Delta_1,\Delta_2;\Gamma \vdash \mathsf{case}_\infty(e, x_1.e_1, x_2.e_2) : U}\ \text{ucase} \qquad \frac{\begin{array}{c}\Delta_1;\Gamma \vdash e : S_1 + S_2 \\ \Delta_2;\Gamma, x_1 : S_1 \vdash e_1 : U \qquad \Delta_2;\Gamma, x_2 : S_2 \vdash e_2 : U\end{array}}{\Delta_1,\Delta_2;\Gamma \vdash \mathsf{case}_\mathsf{w}(e, x_1.e_1, x_2.e_2) : U}\ \text{scase}$$

$$\frac{\begin{array}{c}\Delta_1;\Gamma \vdash e : X_1 \oplus X_2 \\ \Delta_2, x_1 : X_1;\Gamma \vdash e_1 : U \qquad \Delta_2, x_2 : X_2;\Gamma \vdash e_2 : U\end{array}}{\Delta_1,\Delta_2;\Gamma \vdash \mathsf{case}_1(e, x_1.e_1, x_2.e_2) : U}\ \text{acase} \qquad \frac{\cdot;\Gamma, x : A \vdash e : U}{\Delta;\Gamma \vdash \lambda_\infty\, x.\, e : A \to_\infty U}\ \text{uabs} \qquad \frac{\textcircled{w};\Gamma, x : A \vdash e : U}{\Delta;\Gamma \vdash \lambda_\mathsf{w}\, x.\, e : A \to_\mathsf{w} U}\ \text{wabs}$$

$$\frac{\Delta, x : X;\Gamma \vdash e : U}{\Delta;\Gamma \vdash \lambda_1\, x.\, e : X \to_1 U}\ \text{aabs} \qquad \frac{\Delta_1;\Gamma \vdash e_2 : A \qquad \Delta_2;\Gamma \vdash e_1 : A \to_\infty U}{\Delta_1,\Delta_2;\Gamma \vdash (e_1\, e_2)_\infty : U}\ \text{uapp}$$

$$\frac{\Delta_1;\Gamma \vdash e_2 : A \qquad \Delta_2;\Gamma \vdash e_1 : A \to_\mathsf{w} U}{\Delta_1,\Delta_2,\textcircled{w};\Gamma \vdash (e_1\, e_2)_\mathsf{w} : U}\ \text{wapp} \qquad \frac{\Delta_1;\Gamma \vdash e_2 : X \qquad \Delta_2;\Gamma \vdash e_1 : X \to_1 U}{\Delta_1,\Delta_2;\Gamma \vdash (e_1\, e_2)_1 : U}\ \text{aapp}$$

$$\frac{\cdot;\Gamma, x : A \to_\infty U \vdash e : A \to_\infty U}{\Delta;\Gamma \vdash \mathsf{fix}_\infty(x.e) : A \to_\infty U}\ \text{ufix} \qquad \frac{\cdot;\Gamma, x : A \to_\mathsf{w} U \vdash e : A \to_\mathsf{w} U}{\Delta;\Gamma \vdash \mathsf{fix}_\mathsf{w}(x.e) : A \to_\mathsf{w} U}\ \text{wfix} \qquad \frac{x : X \to_1 U;\Gamma \vdash e : X \to_1 U}{\Delta;\Gamma \vdash \mathsf{fix}_1(x.e) : X \to_1 U}\ \text{afix}$$

$$\frac{\begin{array}{c}\Delta_1;\Gamma \vdash e_1 : A \\ \Delta_2;\Gamma, x : A \vdash e_2 : U\end{array}}{\Delta_1,\Delta_2;\Gamma \vdash \mathsf{let}_\infty(e_1, x.e_2) : U}\ \text{ulet} \qquad \frac{\begin{array}{c}\Delta_1;\Gamma \vdash e_1 : X \\ \Delta_2, x : X;\Gamma \vdash e_2 : U\end{array}}{\Delta_1,\Delta_2;\Gamma \vdash \mathsf{let}_1(e_1, x.e_2) : U}\ \text{alet} \qquad \frac{\Delta;\Gamma \vdash e : A}{\Delta;\Gamma \vdash\, !\, e : !\, A}\ \text{bang} \qquad \frac{\Delta;\Gamma \vdash e : !\, A}{\Delta;\Gamma \vdash\, ¡\, e : A}\ \text{gnab}$$

$$\frac{\Delta, x_1 : \mathsf{Rd}\, S;\Gamma, x_2 : \mathsf{Wr}\, S \vdash e : U}{\Delta;\Gamma \vdash \nu(x_1, x_2).\, e : U}\ \text{nu} \qquad \frac{\begin{array}{c}\Delta_1;\Gamma \vdash e_1 : S \\ \Delta_2;\Gamma \vdash e_2 : \mathsf{Wr}\, S\end{array}}{\Delta_1,\Delta_2,\textcircled{w};\Gamma \vdash \mathsf{wr}(e_1, e_2) : \mathbb{1}}\ \text{wr} \qquad \frac{\begin{array}{c}\textcircled{w} \notin \Delta_2 \qquad \Delta_1;\Gamma \vdash e_1 : \mathsf{Rd}\, S \\ \Delta_2, \textcircled{w}, x : !\, S \otimes \mathsf{Rd}\, S;\Gamma \vdash e_2 : U\end{array}}{\Delta_1,\Delta_2;\Gamma \vdash \mathsf{rd}(e_1, x.e_2) : U}\ \text{rd}$$

$$\frac{\begin{array}{c}\textcircled{w} \notin \Delta_3 \qquad \Delta_1;\Gamma \vdash e_1 : \mathsf{Rd}\, S \qquad \Delta_2;\Gamma \vdash e_2 : \mathsf{Rd}\, T \\ \Delta_3, \textcircled{w}, x_1 : !\, S \otimes \mathsf{Rd}\, S \otimes \mathsf{Rd}\, T;\Gamma \vdash e_3 : U \\ \Delta_3, \textcircled{w}, x_2 : !\, T \otimes \mathsf{Rd}\, S \otimes \mathsf{Rd}\, T;\Gamma \vdash e_4 : U\end{array}}{\Delta_1,\Delta_2,\Delta_3;\Gamma \vdash \mathsf{ch}(e_1, x_1.e_3, e_2, x_2.e_4) : U}\ \text{choice} \qquad \frac{\begin{array}{c}\Delta_1;\Gamma \vdash e_1 : U \\ \Delta_2;\Gamma \vdash e_2 : V\end{array}}{\Delta_1,\Delta_2;\Gamma \vdash e_1 \mid\!\triangleright\, e_2 : V}\ \text{fork}$$

Figure 3. ILC typing rules.

Process names	$p, q ::= \cdots$
Name sets	$\Sigma ::= \varepsilon \mid \Sigma, d \mid \Sigma, p$
Process pools	$\pi ::= \varepsilon \mid \pi, p : e$
Configurations	$C ::= \langle \Sigma; \pi \rangle$

Evaluation contexts
$$E ::= \bullet \mid (E, e)_\ell \mid (v, E)_\ell \mid \mathsf{inj}^i_\ell(E) \mid \mathsf{split}_\ell(E, x_1.x_2.e) \mid \mathsf{case}_\ell(E, x_1.e_1, x_2.e_2) \mid (E\, e)_\ell \mid (v\, E)_\ell \mid \mathsf{let}_\pi(E, x.e) \mid !\, E \mid ¡\, E \mid \mathsf{wr}(E, e) \mid \mathsf{wr}(v, E) \mid \mathsf{rd}(E, x.e) \mid \mathsf{ch}(E, x_1.e_3, e_2, x_2.e_4) \mid \mathsf{ch}(c, x_1.e_3, E, x_2.e_4)$$

Figure 4. ILC dynamic syntax.

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

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

$\boxed{c_1 \rightsquigarrow c_2}$ Write endpoint c_1 connects to read endpoint c_2.

$$\frac{}{\mathsf{Write}(d) \rightsquigarrow \mathsf{Read}(d)}\ \text{bind}$$

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

$$\frac{e_1 \longrightarrow e_2}{\langle \Sigma; \pi, p : E[e_1] \rangle \longrightarrow \langle \Sigma; \pi, p : E[e_2] \rangle}\ \text{local} \qquad \frac{q \notin \Sigma}{\langle \Sigma; \pi, p : E[e_1 \mid\rhd e_2] \rangle \longrightarrow \langle \Sigma, q; \pi, q : e_1, p : E[e_2] \rangle}\ \text{fork}$$

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

$$\frac{c_2 \rightsquigarrow c_1}{\langle \Sigma; \pi, p : E_1[\mathsf{rd}(c_1, x.e)], q : E_2[\mathsf{wr}(v, c_2)] \rangle \longrightarrow \langle \Sigma; \pi, p : E_1[[(!v, c_1)_1/x]e], q : E_2[()] \rangle}\ \text{rw}$$

$$\frac{c \rightsquigarrow c_i \qquad i \in \{1, 2\}}{\langle \Sigma; \pi, p : E_1[\mathsf{ch}(c_1, x_1.e_1, c_2, x_2.e_2)], q : E_2[\mathsf{wr}(v, c)] \rangle \longrightarrow \langle \Sigma; \pi, p : E_1[[(!v, c_1, c_2)_1/x_i]e_i], q : E_2[()] \rangle}\ \text{cw}$$

$\boxed{e_1 \longrightarrow e_2}$ Expression e_1 reduces to e_2.

$$\frac{}{\mathsf{let}_\pi(v, x.e) \longrightarrow [v/x]e}\ \text{let} \qquad \frac{}{((\lambda_\ell\, x.\, e)\, v)_\ell \longrightarrow [v/x]e}\ \text{app} \qquad \frac{}{\mathsf{split}_\ell((v_1, v_2)_\ell, x_1.x_2.e) \longrightarrow [v_1/x_1][v_2/x_2]e}\ \text{split}$$

$$\frac{}{\mathsf{case}_\ell(\mathsf{inj}^i_\ell(v), x_1.e_1, x_2.e_2) \longrightarrow [v/x_i]e_i}\ \text{case} \qquad \frac{}{\mathsf{fix}_\ell(x.e) \longrightarrow [\mathsf{fix}_\ell(x.e)/x]e}\ \text{fix} \qquad \frac{}{¡\,(!\,v) \longrightarrow v}\ \text{gnab}$$

Figure 5. ILC reduction rules.

The nu rule extends the affine context Δ with a read endpoint $x_1 : \mathsf{Rd}\, S$ and the unrestricted context Γ with a corresponding write endpoint $x_2 : \mathsf{Wr}\, S$ before typing the body e.

The choice rule partitions the affine context as $\Delta_1, \Delta_2, \Delta_3$. The first two affine contexts are used to type $e_1 : \mathsf{Rd}\, S$ and $e_2 : \mathsf{Rd}\, T$, respectively. The third affine context Δ_3 is extended with the affine write token and a variable x_1 (or x_2) binding an affine 3-tuple containing the read value and the two read endpoints before checking the continuation e_3 (or e_4). While somewhat cumbersome, the generality of this rule allows both read endpoints to be used in either continuation.

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, and a pool of running and terminated processes π.

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 $e_1 \longrightarrow e_2$ for a single process e as "expression e_1 steps to 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 six rules. These include a congruence rule congr that permits some of the other rules to be simpler, by making the order of the pool unimportant. The relation $\pi_1 \equiv_{\mathsf{perm}} \pi_2$ holds when π_2 is a permutation of π_1. The other five rules consist of local stepping (via local), creating new processes (via fork), creating new channels (via nu), read-write interactions (via rw), and choice-write interactions (via cw). To avoid allocating the same name

twice, the name set Σ records names of allocated channels and processes. We define the relation $c_1 \rightsquigarrow c_2$ to hold when c_1 is the write endpoint of a corresponding read endpoint c_2.

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 Section 3, ILC's type system does so by restricting the write effects (via an affine write token) and read effects (via affine read endpoints) of 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 nondeterministic name choices).

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 that permits deadlocks. (Recall that ILC is concerned with enforcing *confluence* as its central metatheoretic property, *not* deadlock freedom.) Informally, C **term** holds when either:

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

We also extend the type system given in Section 3.2 with typing rules for configurations, including process pool typings Φ from process names p to types U. These details, along with the proofs of progress and preservation, can be found in the online appendix.

Theorem 4.1 (Progress). *If $\Psi \vdash C : \Phi$, then either C **term** or there exists C' such that $C \longrightarrow C'$.*

Theorem 4.2 (Preservation). *If $\Psi \vdash C : \Phi$ and $C \longrightarrow C'$, then there exists $\Psi' \supseteq \Psi$ and $\Phi' \supseteq \Phi$ such that $\Psi' \vdash C' : \Phi'$.*

4.2 Confluence

Confluence implies, among other things, that the order of reduction steps is inconsequential, and that no process scheduling choices will affect the final outcome. ILC's type system enforces confluence up to nondeterministic naming choices in rules nu and fork (Figure 5). To account for different choices of dynamically-named 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 4.3 (Single-step confluence). *For all well-typed configurations C, if $C \longrightarrow C_1$ and $C \longrightarrow C_2$, then there exists a renaming function f such that either:*

1. *$C_1 = f(C_2)$, or*
2. *there 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 4.4 (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 $C_1 = f(C_2)$.*

The proofs of these statements can be found in the online appendix.

5 Implementation

Using this on-paper design as a guide, we have implemented an ILC interpreter in Haskell, which at present consists of 2.3K source lines of code. The implementation of ILC and our concrete implementation of the UC framework called SaUCy (Section 6) are publicly available. Access to the latest developments can be found here:

https://github.com/initc3/SaUCy.

6 SaUCy

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

1. We define a protocol composition operator and prove 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:

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

This says that if for all security parameters k and all bitstrings r (of length polynomial in k) the term $e\ k\ r$ normalizes to a value v in poly(k) steps, then PPT e.

Here, we have chosen a simple definition of polynomial time defined only for closed terms (i.e., an entire system of ITMs), and that requires polynomial time normalization for every choice of random bits, not just in expectation or with high probability.

We note that most UC variants use a more nuanced definition in which the individual ITM entities, such as the environment or protocol, can be judged polynomial time independently of their surrounding context [3, 16, 26]. Looking ahead to Section 6.3, this choice will constrain our definition of secure protocol emulation. Hofheinz et al. [27] give a detailed discussion of subtle issues arising with various polynomial time definitions and their consequences for defining UC security. Regardless, the present notion suffices for our examples. We consider this issue complementary to the design of ILC itself, and adapting other notions of polynomial time to ILC as important future work. As an example, the polynomial time notion used in IITMs [3] relies on a distinction between "invited" and "uninvited" messages, which could be captured through refinement types à la the RCF calculus [12].

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

$$D_{e,k}(v) = \sum_{r \in R} U_k(r), \quad \text{for } R = \{r \mid e\ k\ r \to^* v\}.$$

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 $e_1 \le e_2$, which captures that e_2 must be PPT if e_1 is PPT, and if so, that their value distributions are statistically similar.

$$\frac{\mathsf{PPT}\ e_1 \implies (\mathsf{PPT}\ e_2 \text{ and } D(e_1) \sim D(e_2))}{e_1 \le e_2}\ \text{indist}$$

6.2 SaUCy Execution Model

The implementation of SaUCy is centered around a definition of the UC execution model in ILC. For space and readability, we elide endpoint allocation/distribution with ellipses. We also abbreviate the type signature (e.g., A_z is the type of z). More details can be found in the online appendix.

```
execUC :: ∀ ... . A_z →_w A_p × A_q → A_f → A_a →
                  Crupt → Nat → [Bit] → Bit
let execUC z (p,q) f a crupt k r =
  ν ... . let (rf,ra,rp,rq,rz) = splitBits r in
            f k rf crupt ...
          |▷ a k ra crupt ...
          |▷ corruptOrNot p k rp (crupt == CruptP) ...
          |▷ corruptOrNot q k rq (crupt == CruptQ) ...
          |▷ z k rz ...
```

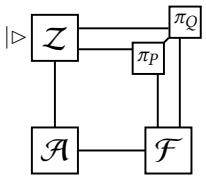


Figure 6. execUC.

The function execUC takes as arguments an environment z, a pair of protocol processes (p, q), a functionality f, an adversary a, a corruption model crupt, a security parameter k, and a random bitstring r. At a high level (ignoring details related to corruptions for now), it runs each of the processes (allocating random bits to each of them) and connects channels as illustrated in Figure 6. (The protocol processes p and q correspond to π_P and π_Q, respectively.) The execution is centered on the environment z in the sense that z first gets the write token (notably, it has type $\mathsf{Nat}\to_{\mathsf{w}} \cdots \to \mathsf{Bit}$), and the experiment concludes when z returns a single bit value.

Next, we explain some of our main modeling choices and the consequences they have for the ILC implementation. To start with, we make several simplifications to standard UC, for example, focusing on the special case of two-party protocols (à la Simplified UC [18]). We also only aim to show the case of *static* corruptions, in which the corrupt parties are determined at the onset. This is achieved by parameterizing the entire experiment by a value crupt : Crupt denoting which parties are corrupt (if any). The data type Crupt is defined as follows.

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

For a more general model with adaptive corruptions, execUC would need to accept requests from the environment to add to the crupt list as the execution proceeds.

Our corruption model is Byzantine, meaning the adversary gets to exert complete control over the corrupted parties. For each party, depending on the value of crupt, either we run a copy of the honest party, or connect the channels to the adversary. This is implemented in the function corruptOrNot.

```
fwd :: ∀ a b . Wr a → Rd a ⊸ b
letrec fwd toR frS =
```

```
  let (!msg, frS) = rd frS in wr msg → toR ; fwd toR frS

corruptOrNot :: ∀ ... . A_p → Nat → [Bit] → Bool → ···
let corruptOrNot p k bits iscrupt toZ toF toA toQ
                                    frZ frF frA frQ =
    if iscrupt then
        let _ = rd frZ in error "Z can't wr to corrupt"
      |▷ fwd toA frF
      |▷ fwd toA frQ
      |▷ fwd toF frA
    else
      p k bits toZ toF toQ frZ frF frQ
```

The fwd function simply forwards messages received on the read endpoint frS to the write endpoint toR. In corruptOrNot, if a party is corrupted, messages from the functionality and the other protocol party are forwarded to the adversary; messages from the adversary are forwarded to the functionality. Otherwise, the party is run as normal.

We also model a strong form of communication channels between the parties: P and Q are connected by a pair of raw ILC channels. Communication over these channels happens immediately, without activating the adversary or leaking even the existence of the message. In a more realistic model, the parties would only be able to communicate over a network channel modeled as a functionality, $\mathcal{F}_{\text{SMT}}$ or $\mathcal{F}_{\text{SYN}}$ [16]. Consequently our $\mathcal{F}_{\text{COM}}$ functionality would need to be weakened by leaking some (model-specific) information about the message to the adversary.

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:

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

To remark on a few notational 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 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. To solve this problem, 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 by the simulators $\mathcal{S}_{\mathcal{R}}, \mathcal{S}_{\mathcal{I}}$) is defined as

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

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 fine if the protocol is even more secure than its specification. This does not pose any problem for our commitment example; however, a protocol that leaks even less information than its ideal functionality requires would be impossible to prove secure under this definition. 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 when the environment $\mathcal{Z}$ is well-behaved.

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 $\mathcal{F}_1$,

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

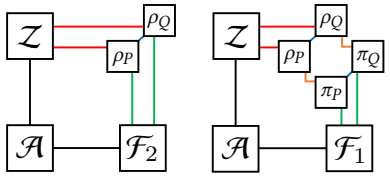
where id_π is the *dummy protocol*, which simply relays messages between the environment and the functionality. This notation is convenient because it suggests a categorical approach to composition.

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 $S_{\mathcal{R},\rho} \circ S_{\mathcal{R},\pi}$ (respectively $S_{\mathcal{I},\rho} \circ S_{\mathcal{I},\pi}$) in the natural way as




```
let (∘) (ρP, ρQ) (πP, πQ)
                wρP2Z wρQ2Z wρP2F wρQ2F
                wρP2ρQ wρQ2ρP rZ2ρP rZ2ρQ
                rF2ρP rF2ρQ rρQ2ρP rρP2ρQ =
  ν ... . πP wπP2ρP wρP2F wπP2πQ rρP2πP rF2ρP rπQ2πP
    |▷ πQ wπQ2ρQ wρQ2F wπQ2πP rρQ2πQ rF2ρQ rπP2πQ
    |▷ ρP wρP2Z wρP2πP wρP2ρQ rZ2ρP rπP2ρP rρQ2ρP
    |▷ ρQ wρQ2Z wρQ2πQ wρQ2ρP rZ2ρQ rπQ2ρQ rρP2ρQ
```

Figure 7. Protocol composition operator.

well (see the online appendix). Our proof obligation is to introduce an arbitrary environment $\mathcal{Z}$ and conclude

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

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. This transformation results in an equivalent term, given that ILC configurations are invariant to channel renaming and reordering of processes in a configuration (as in Section 4). The following derivation completes the proof:

$$\begin{aligned}
&\text{execUC}\ \mathcal{Z}\ (\rho \circ \pi)\ \mathcal{F}_1\ \mathbb{1}_{\mathcal{A}} \\
\equiv\ &\text{execUC}\ (\mathcal{Z} \circ \rho)\ \pi\ \mathcal{F}_1\ \mathbb{1}_{\mathcal{A}} && \text{(By equivalence)} \\
\leq\ &\text{execUC}\ (\mathcal{Z} \circ \rho)\ \text{id}_{\pi}\ \mathcal{F}_2\ \mathcal{S}_{I,\pi} && \text{(From } \mathcal{F}_1 \xrightarrow{\pi} \mathcal{F}_2\text{)} \\
\equiv\ &\text{execUC}\ (\mathcal{S}_{I,\pi} \circ \mathcal{Z})\ \rho\ \mathcal{F}_2\ \mathbb{1}_{\mathcal{A}} && \text{(By equivalence)} \\
\leq\ &\text{execUC}\ (\mathcal{S}_{I,\pi} \circ \mathcal{Z})\ \text{id}_{\pi}\ \mathcal{F}_3\ \mathcal{S}_{I,\rho} && \text{(From } \mathcal{F}_2 \xrightarrow{\rho} \mathcal{F}_3\text{)} \\
\equiv\ &\text{execUC}\ \mathcal{Z}\ \text{id}_{\pi}\ \mathcal{F}_3\ (\mathcal{S}_{I,\pi} \circ \mathcal{S}_{I,\rho}) && \text{(By equivalence)}
\end{aligned}$$

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" [16] operator essentially multiplexes sessions identified by unique tags (*session ids*), while a joint state composition theorem collapses multiple subroutines into one [20]. 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 [28, 30] 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 next walk through an instantiation of UC commitments (à la Canetti and Fischlin [19]). 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.

While commitments are one of the simplest UC primitives, as a case study, this serves two main purposes. First, the proof demonstrates several representative UC techniques [36], in particular the simulator makes use of a "trusted setup" and extracts inputs from a corrupt sender. Second, the protocol makes use of computational primitives and thus requires a reduction step in the proof, which can go through because of ILC's confluent design.

Extending ILC with cryptographic primitives. The UC commitment protocol makes use of a cryptographic primitive, namely a trapdoor pseudorandom generator. This is provided by extending ILC with new syntactic forms, along with their static and dynamic semantics (given in the online appendix). While in a symbolic setting we would instantiate these with algebraic data, in ILC we give the stepping rule in terms of an arbitrary pseudorandom function family, i.e., the actual computational definition. This can be instantiated concretely for execution (e.g., with an RSA-based function) or treated abstractly in the metatheory when we get to the reduction step of the proof.

The commitment protocol also relies on a "trusted setup," or common reference string (CRS), which is essentially public parameters generated ahead of time. The common reference string is modeled as an ideal functionality $\mathcal{F}_{\text{CRS}}$ (implemented in ILC as fCrs in the online appendix).

Commitment Protocol. We implement the commitment protocol by Canetti and Fischlin [19] in ILC as follows:

```
committer :: ∀ ... . Nat → [Bit] → ··· ⊸ 𝟙
let committer k bits crupt toZ toF toQ frZ frF frQ =
  let (!(Commit b), frZ)= rd frZ in
    wr GetCRS → toF ;
    let (!(PublicStrings σ pk0pk1), frF) = rd frF in
    let r = take k bits in
    let x = if b == 0 then prg pk0 r
            else xors (prg pk1 r) σ in
      wr Commit' x → toQ ;
      let (!Open, frZ)= rd frZ in
        wr (Open' b r) → toQ

receiver :: ∀ ... . Nat → [Bit] → ··· ⊸ 𝟙
let receiver k bits crupt toZ toF toP frZ frF frP =
  let (!(Commit' x), frP) = rd frP in
    wr GetCRS → toF ;
    let (!(PublicStrings σ pk0pk1), frF) = rd frF in
```

```
    wr Receipt ⟶ toZ ;
    let (!(Open' b r), frP) = rd frP in
      if (b == 0 && x == prg pk0 r) ||
         (b == 1 && x == xors (prg pk1 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 fCom samples a random string σ and two trapdoor pseudorandom generator (PRG) keys pk_0 and 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 SaUCy 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 [19]. The nonstandard real world simulator, given in the online 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 online 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 Receipt 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, which we do next.

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 [19].

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

```
let simI k bits crupt toZ toF toP toQ frZ frF frP frQ =
  let (pk0,td0) = kgen k in
  let (pk1,td1) = kgen k in
  let (r0, bits) = sample k bits in
  let (r1, bits) = sample k bits in
  let σ = xors (prg pk0 r0) (prg pk1 r1) in
    match crupt with
    | CruptP ⇒
      let (!GetCRS, frZ) = rd frZ in
        wr (X2Z (PublicStrings σ pk0 pk1)) ⟶toZ ;
        let (!(A2P (Commit' y)), frZ) = rd frZ in
          if check td0 pk0 y then
            wr (Commit 0) ⟶ toP
          else
            if check td1 pk1 (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 pk0 r ||
               b == 1 && y == xors (prg pk1 r) σ
            then wr Open ⟶ toP
            else error "Fail"
    | ...
```

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

Here, ILC's confluence property plays a critical role: It is necessary for defining the probability ensembles $D_{\mathcal{R}}$, $D'_{\mathcal{R}}$, and $D_{\mathcal{I}}$, without which we would not be able to obtain a reduction from some computationally hard problem to distinguishing the real world and ideal world ensembles.

6.6 Reentrancy in SaUCy

Camenisch et al. [14] recently identified subtleties in defining UC ideal functionalities (related to reentrancy and the scheduling of concurrent code) such that several functionalities in the literature are ambiguous as ITMs. Although concerning, these issues have no cryptographic flavor, and so they are better addressed from a PL standpoint. To illustrate, consider the following (untypeable) ILC process reentrantF, which

allows an adversary $\mathcal{A}$ to control the delivery schedule of messages from P to Q (i.e., an asynchronous channel):

```
loop :: ∀ a b . (a ⟶ b) ⟶ Rd a ⊸ b
letrec loop f frS = let (!v, frS) = rd frS in f v; loop f frS
let reentrantF ... frP frA =
  loop (λ msg . (let (!Ok, frA) = rd frA in wr msg ⟶ toQ)
                |▷ wr msg ⟶ toA) frP
```

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 $\mathcal{A}$, 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 issue, notice that reentrantF is untypeable in ILC. The race condition occurs because the read endpoint frA is duplicated (appears free in an unrestricted function). Camenisch et al. [14] 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 (!msg, frP) = rd frP in
    wr msg ⟶ toA ;
    let (!Ok, frA) = rd frA in
      wr msg ⟶ 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 (!msg, frP) = rd frP in
    wr msg ⟶ toA ;
    letrec iloop () frP frA =
      choice
      | (_,frP,frA)@(rd frP) ⇒ wr Discard ⟶ toP ;
                               iloop () frP frA
      | (_,frP,frA)@(rd frA) ⇒ wr msg ⟶ toQ ;
                               discardingF ... frP frA
      in iloop () frP frA
```

Ultimately, Camenisch et al. propose a different strategy, which is to restrict how the environment/adversary respond to certain "urgent" messages that are used to exchange meta-information (modeling related messages). That is, upon receiving an urgent message from process P, the environment (or adversary) must return control back to P immediately. Modeling this solution is left as future work, but ILC provides an ideal starting point—restrictions on the environment/adversary could be expressed by behavior refinements: upon receiving an urgent message from P, the environment (or adversary) must not send a message on its other channels before sending a message to P.

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 [43]. Symbolic UC [10] 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. [35] give a computational semantics to a variant of the π-calculus, which allows one to define communicating probabilistic polynomial-time processes; Mateus et al. [38] 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 nondeterminism, each path has equal probability. Laud [32] 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. [24] 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

Computer-aided tools for cryptographic analysis operate in either the symbolic model or the computational model. The survey by Blanchet [8] highlights some of their differences.

Symbolic tools include the NRL protocol analyzer [41], Maude-NPA [23], and Proverif [9]. In the symbolic setting,

cryptographic operations are abstracted as term algebras (a variant of the applied π-calculus in the case of Proverif), and adversary capabilities are nondeterministic applications of deduction rules over these terms. Here, nondeterminism allows the adversary to find attack traces (if there are any), whereas the presence of nondeterminism in the computational setting would frustrate cryptographic reduction proofs.

Computational tools include CertiCrypt [5], EasyCrypt [4], CryptoVerif [7], Cryptol [33], and F* [45]. These tools focus on game-based security, which, in contrast to simulation-based definitions (such as UC), only guarantee security in a standalone setting (no composition guarantees). While they are not specifically purposed for simulation-based proofs, it would be interesting to embed ILC into EasyCrypt or F* to use their tooling.

7.3 Variations of Universal Composability

A number of models for universal composability have been proposed in the literature [3, 10, 13–18, 20, 26, 39, 40, 44]. We highlight a few that have similar goals to ours.

In contrast with UC, which uses ITMs as its computational model, the reactive simulatability framework (RSIM) [3] uses probabilistic IO automata, which are amenable to automated reasoning. In contrast with RSIM, ILC is intended to be the basis for a convenient and flexible programming language to which we can easily port existing UC pseudocode.

Models based on inexhaustible interactive Turing machines (IITMs) [15, 31] aim to address drawbacks of UC models for which polynomial time ITMs can be "exhausted" (by having other machines send useless messages, forcing them to halt). In turn, models with exhaustible ITMs are less expressive. Because IITMs maintain the "single-threaded" execution semantics of ITMs, ILC can be used to build a concrete programming model for IITM-based frameworks as well.

The abstract cryptography framework [40] advocates a top-down approach: developing theory at an abstract level (ignoring low level details such as computational models and complexity notions) to simplify definitions. While we stick to a bottom-up approach, we aim to simplify UC via PL formalisms.

Simplified universal composability (SUC) [18] gives a simpler and restricted variant of the UC framework. The main difference from vanilla UC [16] 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 implementation.

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 [21, 22, 42], 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.

Acknowledgements

We thank our shepherd, Amal Ahmed, and the anonymous reviewers for their valuable feedback. This material is based on work supported by the National Science Foundation under Grant No. 1801321 and a Graduate Research Fellowhip.

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. 2012. Security protocol verification: Symbolic and computational models. In *Proceedings of the First international conference on Principles of Security and Trust*. Springer-Verlag, 3–29.

[9] 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).

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

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

[12] Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. 2015. Affine refinement types for secure distributed programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37, 4 (2015), 11.

[13] Jan Camenisch, Manu Drijvers, and Björn Tackmann. [n. d.]. Multi-Protocol UC and its Use for Building Modular and Efficient Protocols.

([n. d.]).

[14] 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.

[15] Jan Camenisch, Stephan Krenn, Ralf Küsters, and Daniel Rausch. [n. d.]. iUC: Flexible Universal Composability Made Simple (Full Version). ([n. d.]).

[16] R. Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*.

[17] Ran Canetti, Ling Cheung, Dilsun Kaynar, Moses Liskov, Nancy Lynch, Olivier Pereira, and Roberto Segala. 2008. Analyzing security protocols using time-bounded task-PIOAs. *Discrete Event Dynamic Systems* 18, 1 (2008), 111–159.

[18] 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.

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

[20] Ran Canetti and Tal Rabin. 2003. Universal composition with joint state. In *Annual International Cryptology Conference*. Springer, 265–281.

[21] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. [n. d.]. *Perun: Virtual payment channels over cryptographic currencies*. Technical Report.

[22] 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.

[23] 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.

[24] Simon Fowler, Sam Lindley, J Garrett Morris, and Sára Decova. 2018. Session Types without Tiers. (2018).

[25] 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.

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

[27] Dennis Hofheinz, Dominique Unruh, and Jörn Müller-Quade. 2013. Polynomial runtime and composability. *Journal of Cryptology* 26, 3 (2013), 375–441.

[28] 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.

[29] 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.

[30] 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.

[31] Ralf Kusters. 2006. Simulation-based security with inexhaustible interactive turing machines. In *Computer Security Foundations Workshop, 2006. 19th IEEE*. IEEE, 12–pp.

[32] 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.

[33] 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.

[34] Kevin Liao, Matthew A. Hammer, and Andrew Miller. 2019. ILC: A Calculus for Composable, Computational Cryptography (Extended Version). https://eprint.iacr.org/2019/402. (2019).

[35] 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.

[36] Yehuda Lindell. 2017. How to simulate it–a tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*. Springer, 277–346.

[37] Yehuda Lindell and Jonathan Katz. 2014. *Introduction to modern cryptography*. Chapman and Hall/CRC.

[38] 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.

[39] Ueli Maurer. 2011. Constructive cryptography–a new paradigm for security definitions and proofs. In *Theory of Security and Applications*. Springer, 33–56.

[40] Ueli Maurer and Renato Renner. 2011. Abstract cryptography. In *In Innovations in Computer Science*. Citeseer.

[41] Catherine Meadows. 1996. The NRL protocol analyzer: An overview. *The Journal of Logic Programming* 26, 2 (1996), 113–131.

[42] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. 2017. Sprites: Payment channels that go faster than lightning. *CoRR abs/1702.05812* (2017).

[43] Robin Milner. 1999. *Communicating and mobile systems: the pi calculus*. Cambridge university press.

[44] Birgit Pfitzmann and Michael Waidner. 2001. A model for asynchronous reactive systems and its application to secure message transmission. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE, 184–200.

[45] 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)*.