Nomos-UC: a programming framework for cryptography based on resource-aware session types

Given Name Surname
University of Science

Given Name Surname
dept. name of organization (of Aff.)

Given Name Surname
dept. name of organization (of Aff.)

Given Name Surname
dept. name of organization (of Aff.)

Fiven Name Surname
dept. name of organization (of Aff.)

Given Name Surname
dept. name of organization (of Aff.)

***Abstract*—This document is a model and instructions for LaTeX. This and the IEEEtran.cls file define the components of your paper [title, text, heads, etc.]. *CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

We want better tools for analyzing cryptographic applications, especially complex ones like smart contracts which make use of varied cryptographic tools (like zero knowledge proofs and multiparty computation) and have modular layered constructions. In cryptography, the leading framework for managing such compositions is Universal Composability (UC) [**?**]. The central idea is the Ideal Functionality: the security specification of a protocol is given by a concrete instance of a program that, if run by a trusted central party, would exhibit all the desired behaviors of the protocol. This leads to a modular framework, where protocols proven secure in UC are gauranteed to be secure when composed with arbitrary other protocols running concurrently. Providing programming language support for this framework has been a focus of recent works like ILC, EasyUC, IPDL, and others. However, these have several shortcomings, which we address through the following questions:

- RQ1: Cryptographic applications often involve several different parties communicating with each other in a pre-defined protocol. However, existing tools often do not provide a streamlined approach of expressing and enforcing such protocols. As cryptographic protocols become more and more complex, it is likely that the implementation of such applications deviates from its intended behavior. In such scenarios, can the programming language provide some feedback to its user that such a deviation has occurred?
- RQ2: Execution cost analysis plays a central role in UC, since we are only concerned with attackers possessing polynomial-time computational power. However, existing tools provide almost no support in enforcing such restrictions on adversaries. In addition, UC configurations contain several different components: simulator, environment, adversary, ideal functionality, and honest and corrupt parties. Thus, it is important to ensure that polynomial-time bounds are enforced on the correct components, which again can become challenging as the complexity of the cryptographic protocol increases. Can we introduce resource-awareness into programming languages such that these restrictions are automatically enforced?

We answer these questions by designing a new language, NomosUC that combines ideas from Nomos [**?**], a resource aware session-typed language, and ILC [**?**], a process calculus for UC. Session types in Nomos allow us to express communication protocols in both the ideal and real world implementations of cryptographic applications. For instance, for the two-party bit commitment protocol [], session types guarantee AM: ideal world guarantee in the ideal world, and AM: real world guarantee in the real world. The Nomos type checker then statically enforces that these protocols are adhered to at runtime. Failure to typecheck provides precise feedback to the programmer on exactly where the implementation departs from its intended protocol.

The challenge of "polynomial runtime" in UC is that individual processes must be judged as polynomial, but when eveluated in context with other concurrently running process it is difficult to assign blame. The current best way to define polynomial runtime, found in the 2019 and later version of UC, is based a concept of "import tokens." This turns out to be a good fit for the "Potential" from Nomos and Resource-Aware ML. Our construction can reason about local polynomial time for open terms based only on the channel types and assert polynomial-bound in the security parameter by checking valid polynomial and import bounds, and it even guarnatees termination. The result is a deep connection between session type semantics and the formal foundation of UC. The Preservation theorem we prove associated with our type system and operational semantics proves the following: well-typed terms in Nomos UC are "locally polynomial time", in the sense required of UC, meaning they do not take more steps that some polynomial function $T(N)$ of the net number of import tokens it has received. AM: Explain better connection between Preservation theorem and what is needed

Identify applicable funding agency here. If none, delete this.

in UC proofs

II. Background

A. Universal Composability

The universal composability framework [**?**] proposes a new framework for proving the security of cryptographic and distributed protocol. Compared to previous works, the UC framework provides a stronger notion of security where protocols that are UC-secure are secure even when composed with arbitrary other protocols running concurrently.

Such a strong notion of security is achieved through the real-ideal world paradigm. The ideal world encompasses an ideal implementation of a protocol, called the *ideal functionality* $\mathcal{F}$, which acts as a trusted third party that caputures all the desired security properties. The ideal functionality is usually a simple definition making it trivial to prove its security properties. The real world, on the other hand, consists of parties running an actual protocol, π, against a real adversary.

Security proofs in UC involve creating a simulator $\mathcal{S}$ in the ideal world that can simulate every potential attack on a real protocol in the real world. If $\mathcal{S}$ can make the two worlds indistinguishable for any real world adversary $\mathcal{A}$ for all distinguishing environments $\mathcal{Z}$, then we say the protocol π UC-emulates the ideal functionality $\mathcal{F}$. Indistinguishability of the two worlds to any $\mathcal{Z}$ implies that the protocol π must exhibit the same security properties as the ideal functionality $\mathcal{F}$ otherwise there should be sobe distinguishing environment. More formally, indistinguishability is stated:

$$\text{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$$

B. The Import Mechanism

A notion of resource-bound computation is necessary for the UC framework to reason about computationally efficient algorithms as well as the capabilities of ITIs under a particular resource constraint. Often we would like to reason about adversarial capabilities under such constraints and perform efficient transformations (transforming an adversary into a simulator).

Current definitions of polynomial-time computation take the form of bounding the computation of an ITI by some polynomial T: given an input of length n a machine μ halts within $T(n)$ steps. However, using the length of the inputs to the machine as n, in this case leads to an infinite runs problems identified by Canetti [**?**]. Machines that are locally $T(n)$-bounded are able to spawn other machines to the point that an infinite chain of such machines can be spawed where, although, each is locally T-bounded, the whole system of of ITMs is no longer PPT. Therefore, a new notion of n is needed.

The UC paper defines an import mechanism where the first ITI, the environment, is spawned with a polynomial amount of import which can be thought of as tokens or coins. The environment can then activate other ITIs with some import tokens allowing them to run for as long as their import token balance permits. In this new definition, an ITI that is T-bounded takes at most $T(n')$ steps where n' is the difference between the import it has received from incoming messages and outgoing import it's given to other machines. Any ITI can write to any other ITI in it's communication set and send it import tokens until the balance of import is exhausted. Eventually, the system of ITMs must halt when it runs out of import until it receives more from the environment.

a) Realizing Import in saUCy: Need to add potentia in Nomos here

III. Nomos UC

Cryptographic and distributed protocols are, well, protocols and therefore, follow a predefined communication pattern. Our key innovation is representing protocols and ideal functioalities in the UC framework using *session types*.

We describe Nomos UC and session types through an example ideal funtionality: a cryptographic commitment. The commitment functionality $\mathcal{F}_{\mathsf{com}}$ encapsulates the security properties of a two-phase, two-party commitment, which, given its simplicity is an ideal learning example.

A. Ideal Commitment

As an example, consider the two-phase commitment protocol. The ideal functionality of this protocol consists of a *sender* S and *receiver* R connected to a trusted third-party, which we name $\mathcal{F}_{\mathsf{com}}$.

The session types for the committer and receiver encapsulate this protocol: In the session-typed setting, we use typed channels to connect two parties. For instance, the channel connecting S to $\mathcal{F}_{\mathsf{com}}$ has the session type sender defined as

$$\textit{stype}\ \mathsf{sender} = \oplus\{\mathbf{commit} : \mathsf{bit} \times \mathsf{scommitted}\}$$

$$\textit{stype}\ \mathsf{scommitted} = \oplus\{\mathbf{open} : \mathsf{sopen}\}$$

$$\textit{stype}\ \mathsf{sopen} = 1$$

The type constructor $\oplus\{\ \}$ denotes an *internal choice* (here with only one choice) dictating that S must send the **commit** message to $\mathcal{F}_{\mathsf{com}}$. Next, we use the type constructor $\times$ to denote that S sends a value of type bit ($\mathsf{bit} \times \ldots$). We then use the $\oplus$ constructor again enforcing that S sends **open** to $\mathcal{F}_{\mathsf{com}}$.

Analogously, the channel connecting R and $\mathcal{F}_{\mathsf{com}}$ has type receiver defined as

$$\textit{stype}\ \mathsf{receiver} = \&\{\mathbf{commit} : \mathsf{rcommitted}\}$$

$$\textit{stype}\ \mathsf{rcommitted} = \&\{\mathbf{open} : \mathsf{bit} \to \mathsf{ropen}\}$$

$$\textit{stype}\ \mathsf{ropen} = 1$$

Type constructor $\&\{\ \}$ represents *external choice* which is the dual to internal choice. It prescribes that R must receive a **commit** message from $\mathcal{F}_{\mathsf{com}}$, followed by an **open** message (using another $\&$ constructor). R must then receive a bit using the $\to$ constructor (dual to $\times$). Finally, the session terminates as indicated by type $\mathbf{1}$.

Protocols expressed via session types are strictly enforced by process definitions. As an illustration, consider the $\mathcal{F}_{\mathsf{com}}$

process that is connected to both S and R. The process is written as

```
proc Fcom:
  (k: int), (rng: [Bit]), (sid: SID),
  ($S: sender), ($R: receiver)
    |– ($fc: 1) =
{
  case $S (
    commit =>
      b = recv $S ;
      $R.commit ;
      case $S (
        open =>
          $R.open ;
          send $R b ;
}
```

Here, $\mathcal{F}_{\mathsf{com}}$ is the name of the process, and S and R are the names of channels *used* by $\mathcal{F}_{\mathsf{com}}$, while F is the channel *provided* by $\mathcal{F}_{\mathsf{com}}$. Every session-typed process provides a unique channel while acting as a client of a non-negative number of channels. The used channels with their types are written to the left of the turnstile ($\vdash$) while the offered channel and type are written on the right. This is analogous to function definitions where used channels correspond to arguments, while offered channel corresponds to the result. The process first case analyzes on channel S branching on the message received. Since there is only one choice **commit**, we only have one branch in the definition. $\mathcal{F}_{\mathsf{com}}$ then receives the bit b (line 3) on S, followed by sending the commit message on channel R (line 4). Once $\mathcal{F}_{\mathsf{com}}$ receives the **open** message on S, it sends the **open** message on R (line 6), followed by the bit b (line 7). Is closing strictly necessary in NomosUC

A session type uniquely defines the protocol for only one role in an ideal functionality. The sender and receiver do different things, therefore, must have their own channels to $\mathcal{F}_{\mathsf{com}}$ rather than communicating over a common channel.

B. Formal Description of the NomosUC Language

The core calculus of the NomosUC language is based on *session types*: a type discipline for communication-centric programming based on message passing via channels. Session-typed channels describe and enforce the protocol of communication among processes. The base system of session types is derived from a Curry-Howard interpretation [**?**] of intuitionistic linear logic [**?**]. As a result, purely linear propositions can be viewed as resources that must be used *exactly once* in a proof yielding the sequent $A_1, \ldots, A_n \vdash C$ where $A_1, \ldots, A_n$ are linear antecedents, while C is the linear succedent. Under this correspondence, a process term P is assigned to the above judgment and each hypothesis as well as the conclusion is labeled with a *channel*:

$$x_1 : A_1, \ldots, x_n : A_n \vdash P :: (z : C)$$

The resulting judgment states that process P *provides* a service of session type A along channel z, *using* the services of session types $A_1, \ldots, A_n$ provided along channels $x_1, \ldots, x_n$ respectively. We mandate all channel names to be distinct for the judgment to be *well-formed*. The antecedents are often abbreviated to Δ.

The operational semantics for session-typed programs are formalized as a system of *multiset rewriting rules* [**?**]. We introduce semantic objects $\mathsf{proc}(c, P)$ and $\mathsf{msg}(c, M)$ describing process P (or message M) providing service along channel c. Remarkably, in this formulation, a message is just a particular form of process, thereby not requiring any special rules for typing; it can be typed just as processes.

Formally, the typing judgment for processes in NomosUC is written as $\mathcal{T}\ ;\ \Psi\ ;\ \Delta \left|\frac{q}{q'}\right. P :: (x : A)$. Here, Ψ denotes the functional data structures and Δ collects the session-typed channels along with an optional write token $\textcircled{w}$ (to resolve non-determinism in the semantics). The process is denoted by P that offers channel x of type A. Finally, token context $\mathcal{T}$ contains the total and current (= received - sent) tokens of each type contained by the process. Similar to import tokens, the natural number annotations q and q' on the turnstile denote the total and current potential stored in the process. We also extend the semantic objects to $\mathsf{proc}(c, w, P)$ and $\mathsf{msg}(c, w, P)$ where work counter w stores the work performed by process P (resp. message M). We will gradually explain each component of the language, initiating with the basic system of session types. For simplicity of exposition, we will display the yet unexplained parts of the system in blue.

The Curry-Howard correspondence gives each linear logic connective an interpretation as a session type. In this article, we restrict to a subset of these connectives that are sufficient for our language and purposes. We follow a detailed description of each of these session type constructors.

Internal Choice: The internal choice $\oplus\{\ell : A_\ell\}_{\ell \in L}$ constructor is an n-ary labeled generalization of the additive disjunction $A \oplus B$. A process that provides $x : \oplus\{\ell : A_\ell\}_{\ell \in L}$ can send any label $k \in L$ along x and then continue by providing $x : A_k$. The corresponding process is written as $(x.k\ ;\ P)$, where P is the continuation that provides A_k. This typing is formalized by the *right rule* $\oplus R$ in linear sequent calculus. The corresponding client branches on the label received along x as specified by the *left rule* $\oplus L$.

$$\frac{(k \in L) \qquad \mathcal{T}\ ;\ \Psi\ ;\ \Delta \left|\frac{q}{q'}\right. P :: (x : A_k)}{\mathcal{T}\ ;\ \Psi\ ;\ \textcircled{w}, \Delta \left|\frac{q}{q'}\right. (x.k\ ;\ P) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R$$

$$\frac{(\forall \ell \in L) \qquad \mathcal{T}\ ;\ \Psi\ ;\ \textcircled{w}, \Delta, (x : A_\ell) \left|\frac{q}{q'}\right. Q_\ell :: (z : C)}{\mathcal{T}\ ;\ \Psi\ ;\ \Delta, (x : \oplus\{\ell : A_\ell\}_{\ell \in L}) \left|\frac{q}{q'}\right. \mathsf{case}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)}\ \oplus L$$

Additionally, the provider should possess the write token to be able to send the label k. Dually, the client receives the write token with the label to continue execution.

Operationally, since communication is asynchronous, the process $(c.k\ ;\ P)$ sends a message k along c and continues as P without waiting for it to be received. As a technical device to ensure that consecutive messages on a channel arrive in order, the sender also creates a fresh continuation channel c' so that the message k is actually represented as $(c.k\ ;\ c \leftarrow c')$ (read:

send k along c and continue as c'). The provider substitutes c' for c enforcing that the next message is sent on c'. The work counter of the process remains unaltered, and the new message is created with work 0.

$$(\oplus S) : \mathsf{proc}(c, w, c.k\ ;\ P) \ \mapsto\ \mathsf{proc}(c', w, [c'/c]P),\ \mathsf{msg}(c, 0, c.k\ ;\ c \leftarrow c')$$

When the message k is received along c, the client selects branch k and also substitutes the continuation channel c' for c, thereby ensuring that it receives the next message on c'. This implicit substitution of the continuation channel ensures the ordering of the messages. The client process also collects the work performed by the message.

$$(\oplus C) : \mathsf{msg}(c, w, c.k\ ;\ c \leftarrow c'), \mathsf{proc}(d, w', \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)) \mapsto\ \mathsf{proc}(d, w + w', [c'/c]Q_k)$$

***External Choice**:* The dual of internal choice is *external choice* $\&\{\ell : A_\ell\}_{\ell \in L}$, the n-ary labeled generalization of the additive conjunction $A \& B$. This dual operator simply reverses the role of the provider and client. The provider process of $x : \&\{\ell : A_\ell\}_{\ell \in L}$ branches on receiving a label $k \in L$ (described in $\&R$), while the client sends this label (described in $\&L$).

$$\frac{(\forall \ell \in L) \qquad \mathcal{T}\ ;\ \Psi\ ;\ \textcircled{w}, \Delta \left|\frac{q}{q'}\right. P_\ell :: (x : A_\ell)}{\mathcal{T}\ ;\ \Psi\ ;\ \Delta \left|\frac{q}{q'}\right. \mathsf{case}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (x : \&\{\ell : A_\ell\}_{\ell \in L})}\ \&R$$

$$\frac{\mathcal{T}\ ;\ \Psi\ ;\ \Delta, (x : A_k) \left|\frac{q}{q'}\right. Q :: (z : C)}{\mathcal{T}\ ;\ \Psi\ ;\ \textcircled{w}, \Delta, (x : \&\{\ell : A_\ell\}_{\ell \in L}) \left|\frac{q}{q'}\right. x.k\ ;\ Q :: (z : C)}\ \&L$$

Dual to internal choice, the client contains the write token which is sent to the provider along with the label. The operational semantics rules are just the inverse of internal choice, and therefore skipped for brevity.

***Termination**:* The type $\mathbf{1}$, the multiplicative unit of linear logic, represents termination of a process, which (due to linearity) is not allowed to use any channels. A terminating process offering on $x : \mathbf{1}$ simply closes channel x while the client waits for this close message to arrive.

$$\frac{q = 0}{\mathcal{T}\ ;\ \Psi\ ;\ \textcircled{w} \left|\frac{q}{q'}\right. \mathsf{close}\ x :: (x : \mathbf{1})}\ \mathbf{1}R$$

$$\frac{\mathcal{T}\ ;\ \Psi\ ;\ \textcircled{w}, \Delta \left|\frac{q}{q'}\right. Q :: (z : C)}{\mathcal{T}\ ;\ \Psi\ ;\ \Delta, (x : \mathbf{1}) \left|\frac{q}{q'}\right. (\mathsf{wait}\ x\ ;\ Q) :: (z : C)}\ \mathbf{1}L$$

Similar to internal choice, the closing process transfers the write token to its waiting client along with the close message. Additionally, the terminating process does not store any potential since it cannot take any further execution steps.

***Exchanging Functional Data**:* Communicating a *value* of the functional fragment along a channel is expressed at the type level by adding the following two session types.

$$A ::= \ldots \mid \tau \to A \mid \tau \times A$$

Here, τ describes a functional type, e.g. $\mathsf{int}, \mathsf{bool}, \tau\ \mathsf{list}$, etc (we assume the language contains standard functional types). The type $\tau \to A$ prescribes receiving a value of type τ with continuation type A, while its dual $\tau \times A$ prescribes sending a value of type τ with continuation A. The corresponding typing rules for arrow ($\to R, \to L$) are given below (rules for $\times$ are inverse).

$$\frac{\mathcal{T}\ ;\ \Psi, (v : \tau)\ ;\ \textcircled{w}, \Delta \left|\frac{q}{q'}\right. P :: (x : A)}{\mathcal{T}\ ;\ \Psi\ ;\ \Delta \left|\frac{q}{q'}\right. v \leftarrow \mathsf{recv}\ x\ ;\ P :: (x : \tau \to A)} \to R$$

$$\frac{r' = p + q' \qquad \Psi \curlyvee (\Psi_1, \Psi_2) \qquad \Psi_1 \Vdash^p M : \tau \qquad \mathcal{T}\ ;\ \Psi_2\ ;\ \Delta, (x : A) \left|\frac{q}{q'}\right. Q :: (z_k : C)}{\mathcal{T}\ ;\ \Psi\ ;\ \textcircled{w}, \Delta, (x : \tau \to A) \left|\frac{q}{r'}\right. \mathsf{send}\ x\ M\ ;\ Q :: (z : C)} \to L$$

As indicated in the $\to R$ rule, receiving a value $y : \tau$ on a channel $x : \tau \to A$ adds it to the functional context Ψ. On the other hand, sending (value of) expression M on channel $x : \tau \to A$ requires that M has type τ (third premise). The premises indicated in blue describe how potential is divided across the functional and session-typed layers and will be described next. Intuitively, the potential in functional context Ψ is *shared* between Ψ_1 and Ψ_2 (second premise); Ψ_1 is used to type M while Ψ_2 is passed on to the continuation Q.

***Import Tokens**:* A defining aspect of NomosUC is the representation of import tokens in the type system. This enables a static reasoning of the import mechanism in NomosUC. To this end, we introduce a novel token context $\mathcal{T}$ in the process typing judgment to denote the real and virtual tokens. This context contains the information on the total and current tokens of each type. Before we explain the token context we first motivate the need for virtual tokens It is a common technique in UC, especially in simulators, to internally run, or simulate, the code of other ITIs. In NomosUC, we wish to enable the same sandbox running of processes, but its channels may have import requirements. It doesn't make sense to send real import to such a process, because, intuitively the internal process should use the import and, therefore, potential available to the "host" process [1]. Therefore, in order to satisfy the types of internally simulated processes we introduce a virtual tokens construction. By default, every process contains a unique real token type K_0 and corresponding number of total and current tokens n and n' resp. denoted by $K_0 \hookrightarrow (n, n')$. There is no mechanism to create a real token; they can only be passed on to a process during its creation, or be exchanged between processes during communication. Virtual tokens, on the other hand, can be created (under certain conditions, see below) by a process. However, all tokens follow a *token hierarchy*: $K_0 \to K_1 \to K_2 \to \ldots K_m$ such that we can only use tokens of type K_i to withdraw tokens of type K_{i+1} [2]. In addition, we use a global function $\mathfrak{f}$ as the connection rate between two successive token types.

[1] As far as resource-contraints go, simulating a process should be no different from natively executing its code.

[2] ITIs in UC can have arbitrary simulation depth, i.e. a process simulates another process which simulates another process. Despite this, we can statically define the token heirarchy because it is statically known the maximum simulation depth of any process in the UC execution.

To maintain well-typedness of a process, an implicit side condition is that the token context must always be *valid*. This involves ensuring that if the context contains m' current tokens of type K_i, it can only contain at most $\mathfrak{f}(m)$ total tokens of type K_{i+1}. The inductive rules for validity of a token context are below.

$$\frac{}{K_0 \hookrightarrow (t_0, t_0')\ \ \mathsf{valid}}$$

$$\frac{\mathcal{T}\ \ \mathsf{valid} \qquad K_i \hookrightarrow (t_i, t_i') \in \mathcal{T} \qquad t_{i+1} \leq \mathfrak{f}(t_i')}{\mathcal{T}, K_{i+1} \hookrightarrow (t_{i+1}, t_{i+1}')\ \ \mathsf{valid}}$$

Since validity of a token context is a side condition, we mandate that it is implicitly satisfied by all the process typing rules presented in our paper. From an implementation point-of-view, this validity check only needs to be performed when the token context changes (in the rules that follow).

As a first step in introducing program notation for import tokens, we need syntax for creating new tokens of a given token type. We call this construct $\mathsf{withdrawToken}\ K_i\ n\ K_{i+1}$.

$$\frac{\mathcal{T}, K_{i+1} \hookrightarrow (t_{i+1} + n, t_{i+1}' + n)\ ;\ \Psi\ ;\ \textcircled{w}, \Delta \left|\frac{q}{q'}\right.\ P :: (x : A)}{\mathcal{T}, K_{i+1} \hookrightarrow (t_{i+1}, t_{i+1}')\ ;\ \Psi\ ;\ \textcircled{w}, \Delta \left|\frac{q}{q'}\right.\ \mathsf{withdrawToken}\ K_i\ n\ K_{i+1}\ ;\ P :: (x : A)}\ \mathsf{tok}$$

The above construct generates n new tokens of type K_{i+1} and adds them to both the total and current count for K_{i+1} in the token context $\mathcal{T}$. The implicit side condition of the validity of the token context ensures that $t_{i+1} + n \leq \mathfrak{f}(t_i')$ where $K_i \hookrightarrow (t_i, t_i') \in \mathcal{T}$. If this side condition fails, the above construct would fail to typecheck.

In addition, we also introduce two dual constructs for exchanging tokens between processes. To this end, we first introduce two new type constructors.

$$A ::= \ldots \mid \triangleright^{\{r:K\}} A \mid \triangleleft^{\{r:K\}} A$$

The provider of $x : \triangleleft^{\{r:K\}} A$ is required to receive r import tokens of type K from the client using the construct $\mathsf{get}\ x\ \{r : K\}$. Dually, the client needs to pay this import using the construct $\mathsf{pay}\ x\ \{r : K\}$. The corresponding typing rules are

$$\frac{\mathcal{T}, K_i \hookrightarrow (t_i, t_i' + r)\ ;\ \Psi\ ;\ \textcircled{w}, \Delta \left|\frac{q}{q'}\right.\ P :: (x : A)}{\mathcal{T}, K_i \hookrightarrow (t_i, t_i')\ ;\ \Psi\ ;\ \Delta \left|\frac{q}{q'}\right.\ \mathsf{get}\ x\ \{r : K_i\}\ ;\ P :: (x : \triangleleft^{\{r:K_i\}} A)}\ \triangleleft R$$

$$\frac{\mathcal{T}, K_i \hookrightarrow (t_i, t_i')\ ;\ \Psi\ ;\ \Delta, (x : A) \left|\frac{q}{q'}\right.\ P :: (z : C)}{\mathcal{T}, K_i \hookrightarrow (t_i, t_i' + r)\ ;\ \Psi\ ;\ \textcircled{w}, \Delta, (x : \triangleleft^{\{r:K_i\}} A) \left|\frac{q}{q'}\right.\ \mathsf{pay}\ x\ \{r : K_i\}\ ;\ P :: (z : C)}\ \triangleleft L$$

In the rule $\triangleleft R$, process P storing (t_i, t_i') import tokens of type K_i receives r additional K_i tokens adding it to the current token counter, thus the continuation executes with $(t_i, t_i' + r)$ tokens of type K_i. Note that validity of token context is trivially satisfied in this case since the process is gaining import tokens. In the dual rule $\triangleleft L$, a process containing $(t_i, t_i' + r)$ tokens of type K_i pays r units along channel x leaving (t_i, t_i') import tokens of type K_i with the continuation. In this case, the validity of the token context establishes that $t_{i+1} \leq \mathfrak{f}(t_i')$, a condition that is necessary for successful typechecking. The typing rules for the dual constructor $\triangleright^{\{r:K\}} A$ are the exact inverse. Similar to prior rules, the sender transfers the write token $\textcircled{w}$ along with the potential to the receiver.

The need for virtual tokens in UC arises because machines often simulate other machines as part of their construction. The program notation for $\mathsf{withdrawToken}$ does not require an inverse to exchange tokens *back* from type K' to K. The reason is that virtual tokens only exist to allow re-use of existing processes and satisfy their types. Type K tokens are not deducted when new ones of type K' are created is because, in reality, siulating a process by calling it or simply running its code natively should be equivalent in cost. Therefore, there is also no need to include an inverse of $\mathsf{withdrawToken}$ which exchanges from K' to K.

Potential: The main purpose of import tokens is to bound the number of execution steps of ITMs. We achieve that purpose in NomosUC by introducing the notion of *potential*. Potential is an abstract quantity represented by a natural number stored within each process. To take an execution step, a process consumes *one* unit of potential. Therefore, the total potential stored in a process upper bounds the total number of execution steps that will ever be taken by the process. Furthermore, potential is represented syntactically, thus providing a static upper bound on the execution cost. Since execution cost needs to eventually connect to the import tokens, all we need is a mechanism to generate potential using import tokens. To this end, we introduce a novel construct $\mathsf{genPot}\ r$.

$$\frac{q + r \leq \mathfrak{f}(t_{\mathfrak{d}}') \qquad K_{\mathfrak{d}} \hookrightarrow (t_{\mathfrak{d}}, t_{\mathfrak{d}}') \in \mathcal{T} \qquad \mathcal{T}\ ;\ \Psi\ ;\ \textcircled{w}, \Delta \left|\frac{q+r}{q'+r}\right.\ P :: (x : A)}{\mathcal{T}\ ;\ \Psi\ ;\ \textcircled{w}, \Delta \left|\frac{q}{q'}\right.\ \mathsf{genPot}\ r\ ;\ P :: (x : A)}\ \mathsf{pot}$$

A process initially storing (q, q') potential units generates r potential so that the continuation contains $(q + r, q' + r)$ potential units. Note, however, that the maximum potential allowed is bounded by the number of import tokens a process contains. To this end, we introduce a *token depth*: $\mathfrak{d}$ that signifies the number of token types that exist in the token hierarchy. Thus, when generating potential, the typechecker verifies that the total new potential (i.e., $q + r$) is bounded by $\mathfrak{f}(t_{\mathfrak{d}}')$ where $(t_{\mathfrak{d}}, t_{\mathfrak{d}}')$ is the number of tokens of type $K_{\mathfrak{d}}$, the highest token in the hierarchy.

The purpose of introducing potential into NomosUC is to bound the number of execution steps. Therefore, we introduce the $\mathsf{tick}\ (r)$ construct that consumes r potential from the stored process potential q, and the continuation remains with $p = q - r$ units, as described in the rule below.

$$\frac{\mathcal{T}\ ;\ \Psi\ ;\ \textcircled{w}, \Delta \left|\frac{q}{q'}\right.\ P :: (x : A)}{\mathcal{T}\ ;\ \Psi\ ;\ \textcircled{w}, \Delta \left|\frac{q}{q'+r}\right.\ \mathsf{tick}\ (r)\ ;\ P :: (x : A)}\ \mathsf{tick}$$

NomosUC is equipped with a cost instrumentation engine that automatically inserts a tick (1) construct before each primitive operation. This enables us to simulate the cost model that counts the total number of execution steps. However, since ticks are not tied directly to the type system, the programmer can modify the cost model to only count the resource they are interested in (e.g., message exchange, process spawns, etc.).

Shared Channels: Until now, we have only described the linear fragment of session types in Nomos. Unfortunately, this fragment imposes a strong restriction on programs. The only provision to spawn new processes is when a parent process creates a new child process, and uses an exclusive linear channel to communicate with the child. Thus, any two processes connected by a channel inherently maintain this parent-child relationship. Intuitively, this leads to a linear tree-like hierarchy among the processes, thus preventing a cycle in the process graph.

Unfortunately, this restriction precludes practical programming scenarios where process topologies indeed have a cyclic dependency (e.g. ring networks, dining philosophers, etc.). Recognizing this limitation, Balzer et al. [**?**] proposed a *shared* extension of session types that allows arbitrary process topologies. The types are extended as follows:

$$A_L ::= \downarrow^{\mathsf{S}}_{\mathsf{L}} A_S \mid \ldots \text{(all linear types } A \text{ so far)} \ldots$$
$$A_S ::= \uparrow^{\mathsf{S}}_{\mathsf{L}} A_L$$

We have found this extension exceedingly helpful in the design and implementation of cryptographic protocols.

Shared session types impose an *acquire-release* discipline on processes; a client must acquire the channel offered by a shared process to interact with it and must release this channel after the interaction. The corresponding typing rules are

$$\frac{\mathcal{T}\ ;\ \Psi\ ;\ \Delta, (y : A_L) \left|\frac{q}{q'}\right. Q :: (z : C)}{\mathcal{T}\ ;\ \Psi\ ;\ \textcircled{w}, \Delta, (x : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_L) \left|\frac{q}{q'}\right. y \leftarrow \mathsf{acquire}\ x\ ;\ Q :: (z : C)}\ \uparrow^{\mathsf{S}}_{\mathsf{L}} L$$

$$\frac{\mathcal{T}\ ;\ \Psi\ ;\ \textcircled{w}, \Delta \left|\frac{q}{q'}\right. P :: (y : A_L)}{\mathcal{T}\ ;\ \Psi\ ;\ \Delta \left|\frac{q}{q'}\right. y \leftarrow \mathsf{accept}\ x\ ;\ P :: (x : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_L)}\ \uparrow^{\mathsf{S}}_{\mathsf{L}} R$$

The $\uparrow^{\mathsf{S}}_{\mathsf{L}} L$ rule describes a client acquiring a shared channel x and obtaining a private linear channel y along which it can communicate with the corresponding acquired process. Correspondingly, the $\uparrow^{\mathsf{S}}_{\mathsf{L}} R$ rule describes the shared process accepting the acquire request and creating the fresh linear channel y. The release-detach rules corresponding to the $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ type constructor are exact dual of acquire-accept.

An important caveat here is that shared channels can introduce non-determinism in the semantics. The only source of non-determinism is that a shared process can latch on to any of the acquiring clients. To address this problem, we require that the acquiring client *possess the write token*. Since write tokens are treated as a linear quantity, only one of the client can possess it enabling only that process to acquire the shared channel. Remarkably, this write token can resolve both read and write non-determinism due to linearity of the channels.

Process Definitions and Sandboxing: Process definitions have the form $\Psi\ ;\ \Delta \left|\frac{q}{q'}\right. f\{\mathcal{T}\} :: (x : A) = P$ where f is the name of the process and P its definition. We parameterize the process f with the number and type of real tokens it would need. All definitions are collected in a fixed global process signature Σ. Also, since process definitions are mutually recursive, it is required that for every process in the signature is well-typed w.r.t. Σ. A new instance of a defined process f can be spawned with the expression $x \leftarrow f\{\mathcal{T}\}\ \overline{y}\ ;\ Q$ where $\overline{y}$ is a sequence of variables matching the antecedents Ψ and Δ. Sometimes a process invocation is a *tail call*, written without a continuation as $x \leftarrow f\{\mathcal{T}\}\ \overline{y}$. This is a short-hand for $x' \leftarrow f\{\mathcal{T}\}\ \overline{y}\ ;\ x \leftarrow x'$ for a fresh variable x', that is, a fresh channel is created and immediately identified with x.

An important note here is that NomosUC allows executing processes in a *sandbox*. Therefore, a process invocation can either be *regular* or in a *sandbox*. Syntactically, we use the same term for both but the two invocations are distinguished via the token type passed into the call. For a regular call, the parent process passes in a real token type, while for a sandboxed call, a virtual token type is passed in. We have a similar distinction for pay and get expressions: if a real token is passed into these terms, it's a regular token exchange; if a virtual token is passed in, it's a sandboxed pay and get.

C. Preservation and Progress

The main type safety theorems that exhibit the deep connection between our type system and the operational semantics are the usual *type preservation* and *progress*, sometimes called *session fidelity* and *deadlock freedom*, respectively.

To exhibit these theorems, we first need to introduce semantic objects $\mathsf{proc}(c, w, P)$ and $\mathsf{msg}(c, w, M)$. The former (resp. latter) denotes a process (resp. message) executing expression P (resp. M) offering channel c and having performed work w so far. The work counter keeps track of execution steps taken by a process, giving rise to the following semantics rule:

$$(\mathsf{tick}) : \mathsf{proc}(c, w, \mathsf{tick}\ (r)\ ;\ P)\ \mapsto\ \mathsf{proc}(c, w + r, P)$$

A multiset of such semantic objects communicating with each other is known as a *configuration*. A configuration is typed w.r.t. a signature providing the type declaration of each process. A signature Σ is *well formed* if (a) every type definition $V = A_V$ is *contractive*, and (b) every process definition $\Psi\ ;\ \Delta \vdash f\{\mathcal{T}\} = P :: (x : A)$ in Σ is well typed according to the process typing judgment, i.e. $\mathcal{T}\ ;\ \Psi\ ;\ \Delta \vdash P :: (x : A)$.

A key question then is how to type these configurations. Since they consist of both processes and messages, they both *use* and *provide* a collection of channels. Another goal with the type safety theorems is to establish a connection between the statically determined import tokens of a process, its total potential, and the dynamically evolving work counters that account for the total number of execution steps. We use the following judgment to type a configuration.

$$\Delta_1 \underset{W}{\overset{(T,Q)}{\models}} \mathcal{C} :: \Delta_2$$

$$\frac{}{\Delta \overset{(0,0)}{\underset{0}{\models}} (\cdot) :: \Delta}\ \mathsf{empty}$$

$$\frac{\Delta_0 \overset{(T_1,Q_1)}{\underset{W_1}{\models}} \mathcal{C}_1 :: \Delta_1 \qquad \Delta_1 \overset{(T_2,Q_2)}{\underset{W_2}{\models}} \mathcal{C}_2 :: \Delta_2}{\Delta_0 \overset{(T_1+T_2,Q_1+Q_2)}{\underset{W_1+W_2}{\models}} (\mathcal{C}_1\ \mathcal{C}_2) :: \Delta_2}\ \mathsf{compose}$$

$$\frac{\mathcal{T}, K_0 \hookrightarrow t\ ;\ \cdot\ ;\ \Delta_1 \left|\frac{q}{P}:: (c : A)\right.}{\Delta, \Delta_1 \overset{(t,q)}{\underset{w}{\models}} \mathsf{proc}(c, w, P) :: (\Delta, (c : A))}\ \mathsf{proc}$$

$$\frac{\mathcal{T}, K_0 \hookrightarrow t\ ;\ \cdot\ ;\ \Delta_1 \left|\frac{q}{M}:: (c : A)\right.}{\Delta, \Delta_1 \overset{(t,q)}{\underset{w}{\models}} \mathsf{msg}(c, w, M) :: (\Delta, (c : A))}\ \mathsf{msg}$$

Fig. 1: Typing rules for a configuration

It states that the configuration $\mathcal{C}$ uses the channels in the context Δ_1 and provides the channels in the context Δ_2. In addition, T and Qdenote the total number of real tokens potential contained in a configuration. Similarly, W denotes the total work performed by a configuration. All these quantities are computed by adding the individual tokens, potential, and work of each semantic object.

The configuration typing judgment is defined using the rules presented in Figure 1. The rule empty defines that an empty configuration is well-typed with $(T, Q, W) = (0, 0, 0)$ and uses and provides the same set of channels. The compose rule combines two configurations by canceling out the common channels and adding the individual tokens, potential, and work. The proc rule creates a configuration out of a single process and uses its tokens, potential, and work as the annotations for the configuration. Similarly, the msg rule creates a configuration out of a single message.

Theorem 1 (Type Preservation)**.** Suppose we have a well-typed configuration $\Delta \overset{(T_1,Q_1)}{\underset{W_1}{\models}} \mathcal{C}_1 :: \Delta'$ such that there exists a polynomial $\mathfrak{p}$ such that $\mathfrak{p}(T_1) \geq Q_1 + W_1$. If $\mathcal{C}_1 \mapsto \mathcal{C}_2$, then there exist T_2, Q_2, and W_2 such that $\Delta \overset{(T_2,Q_2)}{\underset{W_2}{\models}} \mathcal{C}_2 :: \Delta'$, and $\mathfrak{p}(T_2) \geq Q_2 + W_2$.

Proof. By case analysis on the transition rule, applying inversion to the given typing derivation, and then assembling a new derivation of $\mathcal{D}$. □

A process or message is said to be *poised* if it is trying to communicate along the channel that it provides. A poised process is comparable to a value in a sequential language. A configuration is poised if every process or message in the configuration is poised. Conceptually, this implies that the configuration is trying to communicate externally, i.e. along one of the channel it provides. The progress theorem then shows that either a configuration can take a step or it is poised. To prove this I show first that the typing derivation can be rearranged to go strictly from right to left and then proceed by induction over this particular derivation.

Theorem 2 (Global Progress)**.** If $\cdot \overset{(T,Q)}{\underset{W}{\models}} \mathcal{C} :: \Delta$ then either

(i) $\mathcal{C} \mapsto \mathcal{C}'$ for some $\mathcal{C}'$, or
(ii) $\mathcal{C}$ is poised.

Proof. By induction on the right-to-left typing of $\mathcal{C}$ so that either $\mathcal{C}$ is empty (and therefore poised) or $\mathcal{C} = (\mathcal{D}\ \mathsf{proc}(c, w, P))$ or $\mathcal{C} = (\mathcal{D}\ \mathsf{msg}(c, w, M))$. By induction hypothesis, $\mathcal{D}$ can either take a step (and then so can $\mathcal{C}$), or $\mathcal{D}$ is poised. In the latter case, I analyze the cases for P and M, applying multiple steps of inversion to show that in each case either $\mathcal{C}$ can take a step or is poised. □

D. UC Communicators

One illustration of the use of shared session types is a *communicator*. We use communicators as message buffers between two arbitrary processes: a *sender* and a *receiver*. The communicator is connected to both the sender and the receiver using a shared channel.

Intuitively, the communicator receives *push* requests from the sender followed by receiving a message and stores them internally. Analogously, the communicator receives *pop* requests from the receiver, and responds appropriately with the message if one is stored inside the communicator. Formally, a communicator has the following polymorphic session type

$$\begin{aligned}&\textit{stype}\ \mathsf{comm}[\mathsf{K}][\mathsf{msg}]\{\mathsf{n}\} = \\ &\quad \uparrow^{\mathsf{S}}_{\mathsf{L}} \triangleleft^{\{n+1:K\}} \& \{\mathbf{push} : \mathsf{msg} \rightarrow \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{comm}[\mathsf{msg}], \\ &\qquad\qquad \mathbf{pop} : \oplus\{\mathbf{yesmsg} : \mathsf{msg} \times \downarrow^{\mathsf{S}}_{\mathsf{L}} \triangleright^{\{n:K\}} \mathsf{comm}[\mathsf{msg}], \\ &\qquad\qquad\qquad \mathbf{nomsg} : \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{comm}[\mathsf{msg}]\}\}\end{aligned}$$

The type comm is parameterized by the type msg, i.e., the type of messages in the buffer, and import type parameter, i.e. the amount of import tokens sent with the message. The type initiates with an $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ denoting that comm is a shared session type. The type prescribes that the communicator needs to be acquired by the sender (or receiver) for further interaction. Such an acquire-release discipline is automatically enforced by the shared session type. Once acquired, the communicator can either receive **push** (from sender) or **pop** requests (from receiver). In the former case, the communicator receives a message of type msg and $n + 1 : K$ import tokens, and then detaches from the client using the dual $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ operator. In the latter case, the communicator checks if it internally contains a message for the receiver. If yes, the communicator replies with the **yesmsg** label followed by sending the message (the $\times$ constructor) and $n : K$ import tokens. Otherwise, the communicator replies with the **nomsg** label. In either case, the communicator then detaches from the client matching the $\downarrow^{\mathsf{S}}_{\mathsf{L}}$ operator. Internally, the communicator stores these messages in a first-in-first-out order.

It is important to note that our communicators need at least 1 token of import to use themselves to handle a potentially polynomial number of activations. Therefore, it requires $n + 1$ units of import from the sender and sends the intended n tokens to the receiver when requested.

The communicator is also the perfect opportunity to implement an unreliable message buffer that can drop or reorder messages. All we would need to do is change the internal implementation of the communicator *without* changing the offered session type.

IV. Related Works

There is a large volume of work that introduce tools for modelling UC protocols and reasoning about UC security with some mechanizing UC proof-generation. Most prior work, however, is not able to capture the polynomial-time notion that underpins UC security.

The EasyUC [**?**] work introduces a toolset built on top of the existing EasyCrypt [**?**] to model UC protocols and generate proofs of security. It moves past the limitations of EasyCrypt which is limited to game-based security definitions rather than the simulation-based definitions of UC. Building on top of EasyCrypt constraints EasyUC to a stack-based computation model which requires a unique communication model and unique addressing scheme for machines. A large limitation, that the authors admit, is that it can not detect deviant behavior like the adversary and functionality exchanging messages indefinitely. NomosUC on the other hand can ensure that all machines are locally polynomially bound in the security parmaeter and ensures that well-typed programs always terminate in polynomial time.

Liao at al. [**?**] introduce a new process calculus, that extendes π-calculus, called ILC. Unlike EasyUC, ILC does not attempt to mechanize proof generation and provides some notion poynomial time. The definition, however, is restrictive and does not allow reasoning about PPT for individual protocols or functionalities. Open terms require enumerating all possible ways they can be closed making it difficult even simple ping-pong servers to be proved polynomial in this definition.

There is large body of similar work that introduces process calculi, some extensions of π-calculus, like ILC. Mateus et al. [**?**] for example introduces process calulus for simpler, sequential composition but is constraints to a schedular-based construction where probabilistic state transitions follow unifor distribution at every step. SymbolicUC [**?**] finish oter symbolic logic and state their weaknesses and that ther are subsumed by ILC.

Morrisett et al. [**?**] propose IPDL which aims to imprtove on EasyUC by providing a better notion of emulation, more akint to the UC framework. IPDL symbolically tracks the run time of programs but can only do so for straight-line programs or those with statically upper-bounded loops, i.e. programs without dynamic loops or recursion. Further, the protocols in IPDL can not dynamically choose which channels to write on because channels have static dependencies for when to "fire".

The added expressiveness of Nomos also allows full composition in the manner of UC. IPDL and the ILC calculus only allow composition of a fixed numeber of functionalities and protocols. IPDL for example must statically define the upper bound on the number of instances of a functionality, and could not, say, adequately handle the number of instances determined by probabilisitically input on a channel. ILC suffers from a similar drawback.

V. ExecUC

In this section we describe how the UC experiment and composition theorem are defined in NomosUC. We also express a critically important result in the UC framework: the dummy lemma. An important part of our definition is using code-generation techniques [**?**] to constructs some processes in the UC experiment as well ass useful operators to achieve full composition in the sense of Canetti et al. [**?**].

We first introduce some convenient notation. For the remainder of this section, when we refer to a protocol, we actually refer to a pair of ITMs as in Definition V.1.

Definition V.1. A *protocol* is a pair of terms $(\pi, \mathcal{F})$ where π is the protocol run by honest parties and $\mathcal{F}$ is an ideal functionality that parties an access.

In the ideal world, π is replaced by an an ideal protocol, $\mathbb{I}_d$, which is a dummy protocol: it frorwards message between $\mathcal{Z}$ to $\mathcal{F}$ for honest parties and between $\mathcal{A}$ and $\mathcal{F}$ for corrupt parties. In the real world, $\mathcal{F}$ is called the *hybrid functionality* and stands in for a real protocol that emulates it. For protocols that don't make calls to any hybrid functionality, $\mathcal{F}$ is just the dummy protocol which does nothing on activation by any other ITM. The dummy functionality accepts all messages, does nothing, and continues to wait for more messages.

A. The UC Experiment

The UC experiment is an execution of a main protocol, called the *challenge protocol*, consisting of protocol parties and an ideal functionality, reacting to input by an adversary $\mathcal{A}$ or the $\mathcal{Z}$. The expeirment is created by an `execUC` functio which spawns all the relevant processes: the environment, a `protocol wrapper`, the adversary, the `functionality wrapper`. The protocol and functionality wrappers encapsulate the protocol parties and the ideal functionality, respectively. We motivate their use and explain how they work later in this section, but point out that they are necessary to make use of sesion types while capturing the full UC framework.

The statically-typed nature of NomosUC requires generic process like `execUC` to be parametric in the message and import token types of the protocol and functionality being executed. Though statically-typed the `execUC` function is one that is generated uniquely based on the protocol being execute, and the reason for doing some comes down to our virtual tokens construction. For any protocol where some process, say the ideal world adversary (the simulator), internally simulates an existing UC protocol, virtual tokens must be used. The

number of virtual tokens used, and therefore statically defined, varies based on the specific processes being used. Therefore, execUC needs to spawn processes with a variable number of import token types. We rely on code-generation to take a protocol specification and create execUC that accepts the appropriate type parameters. For the commitment example we've used throughout this work, the ideal world execUC process definition looks like this:

Listing 1: The process definition of the execUC function.

```
proc execUC[K][z2p][p2z]...{z2pn}{p2zn}... :
  (k: int), (r: [Bit]) |– ($d: Bit)
```

In the ideal world, none of the processes, including the simulator which we will present later, require any virtual tokens. Hence there is only one virtual token type: K. The input parameters to the function are all generic and are required as part of a UC execution: the security parameter k selected by the user and the source of random bits r. The security parameter k is also selected by the user and the source of randomness r is given to all the ITMs by execUC. The type of the channel that execUC offers, $d, is Bit because it returns the environment's output: a bit representing its guess as to which word it thinks it's in.

We separate NomosUC into these two modules: generic code and protocol specific code. Therefore, the execUC definition doesn't include process definitions as parameters and refers to them through a modue PS. The module PS is must implement the processes PS.env for $\mathcal{Z}$, PS.func for the functionality, PS.prot for the protocol, and PS.adv for the adversary. The environment PS.env must have the following process definition generic to all environments:

```
type EtoZ[a] = +{init: a ^ list[pid] –> exec}
type exec = &{start : output_bit} ;
type output_bit = +{bit: Bit –> 1} ;
proc PS.env[K][z2p][p2z]...{z2p}{p2zn} :
  (k: int), (r: [Bit]),
  (#ztop: comm[z2p]), (#ptoz: comm[p2z])...
    |– ($z : EtoZ)
```

The type parameters specify the token type K, and the types of the channels $\mathcal{Z}$ has to others (only between $\mathcal{Z}$ and protocol wrapper is shown). Similarly, there are type parameters specifying the import parameters as well ({z2pn}). The {z2pn}, for example, specifies the amount of import that must be sent by $\mathcal{Z}$ with every message to a protocol party. The rest of the processes—the adversary, protocol wrapper, and functionality–are declared in the same way except the type parameters they expect are different: the protocol wrapper would of course accept type parameters for communication between it and $\mathcal{F}$, $\mathcal{A}$, and $\mathcal{Z}$. Finally the type of the offered channel $z is the same for all environments and communicates the sid and set of corrupt parties chosen by $\mathcal{Z}$ to execUC. The sid and corrupt list are parameters to the rest of the processes that execUC spawns.

The type parameters given to execUC are not directly given to the corresponding processes. All message types are wrapped in generic parametric types that represent communication between two entitis in the UC execution. Messages between the main entities of the execution, for example $\mathcal{Z}$ and $\mathcal{A}$, all have the same parameteric types such as:

```
type p2zmsg[a] = P2Z of pid ^ a ;
type p2fmsg[a] = P2F of pid ^ a ;
type f2pmsg[a] = F2P of pid ^ a ;
```

The channel from $\mathcal{Z}$ to the protocol wrapper will typed as: comm[z2pmsg[z2p]]{z2pn} where z2p is a type parameter to execUC.

The first thing execUC does is create the communicators and the corresponding channels for the main processes of the execution. Recall from the discussion on communicators in Section **??** that the communicators are typed as p2zn+1 because they need at least one unit of import to handle being activated a potentially polynomial numbers of times.

```
#ptoz <– communicator_init[K][p2zmsg[p2z]]
          {p2zn+1} ;
#ztop <– communicator_init[K][z2pmsg[z2p]]
          {z2pn+1} ;
...
```

This is strictly a design decision where user-defined protocols only specify import token requirements for the protocol not taking communicators into account. The processes that they write, though, must conform to this standard and send an import token in addition to the amount they need. An alternate design would be that all import type parameters take the additional import token into account and communicators send one less than they receive. The drawback of the latter approach is that that when communicating with a process directly rather than through communicators (say, when simulating), you're sending one extra token for no reason.

Next, the environment PS.env is spawned, execUC receives the sid and corrupt list from $\mathcal{Z}$, and, finally, the remainder of the processes are spawned with these inputs. Finally, the environment executes its own code when activated by $z.start, given the initial amoutn of import n, and returns a bit through $z.output_bit which is its guess whether it's in the real or ideal world:

```
$z.start
pay $z {n}
$d <– $z
```

B. The Protocol Wrapper

The execUC definition introduces a new construct called the *protocol wrapper*. It is instantiated with the protocol that honest parties run, and spawns new ones on-demand.

We introduce the protocol wrapper when working with session types for the following reason. First, in the UC execution we need $\mathcal{Z}$ to be able to spawn any number of protocol parties. If instead there was a single channel through which all parties communicated with the functionality, the session type would the product of all possible sessions for each party. Although possible, dealing with such session types

is cumbersome, and they so not allow the type to change to add new parties. Therefore, we optto have a protocol wrapper that sandboxes all protocol parties, communicates with them through their session type, but converts messages to/from them into functionally typed messages when communicating with $\mathcal{F}$, $\mathcal{A}$, and $\mathcal{Z}$. The user defines both the functional and session types for the protocol and the protocol wrapper handles converting between them. We use the two-party commitment protocol as an example to demonstrate how the construction works and show that it is generic enough to allow code generation of a wrapper for any protocol.

Recall the session types for the sender and receiver, with $\mathcal{F}_{\mathsf{com}}$, introduced in Section III-A. At the beginning there are no parties. When the $\mathcal{A}$ or $\mathcal{Z}$ write to a party with some pid p the protocol wrapper creates p if it does not exist. It creates and stores the session-typed channels or all of the parties in channels corresponding to their type and moves channels between them as the type evolves. For example for parties' channel with $\mathcal{Z}$ the protocol wrapper generates the following lists:

$$\mathsf{R1L1}[\mathsf{sender}] \tag{1}$$
$$\mathsf{R1L2}[\mathsf{scommitted}] \tag{2}$$
$$\mathsf{R2L1}[\mathsf{receiver}] \tag{3}$$
$$\mathsf{R2L2}[\mathsf{rcommitted}] \tag{4}$$

At the beginning of the protocol, the committer's z2p channel would be in `R1L1` and the receiver's is in `R2L1`. The channel's connection the protocol wrapper to the rest are still functionally typed. It's channel `p2f` will still be typed `#ptof: comm[p2fmsg[comp2f]]` where `comp2f` is:

```
type comp2f = Commit of Bit | Open ;
type comf2p = Committed | Open of Bit ;
```

Succinctly, the protocol wrapper functions as follows:

- When the protocol wrapper receives a message for some pid, if the party doesn't exist the protocol wrapper creates all the party's channel parameterized by the correct session types (the role of the party is determined by the incoming message). The channels are stored and outgoing channels are waiting to be read from.
- The message sent to the party with the session type is determined by the functional type. The wrapper creates a session-typed message and forwards the contents of the message to the party. Despite accepting functionally typed messages, the protocol wrapper uses session types in this way to allow the type checker to catch invalid and out of order environments, adversaries and functionalities. For example, in the commitment protocol when the committer tries to commit a bit in $\mathcal{F}_{\mathsf{com}}$ like this:

```
$p2f.commit
send $p2f b
pay {p2fn} $p2f
```

```
$p2f.SEND
send $p2f pid
send $p2f Commit(b)
pay {p2fn} $p2f
```

The party wrapper intercepts and converts it to a functionally typed message:

- For outgoing messages, the protocol wrapper does a similar conversion where it reads the sesion typed channel output by the party and converts its message to the appropriate functional message type.

a) Functionality Wrapper: We adopt a similar construction for functionalities, in Nomos, with the functionality wrapper. The wrappers enables functionalities to also use session types but interact with the other ITIs in the execution through shared channels and functional message types. It differs from the protocol wraper in that it only spawns a single instance of the functionality, and it is created by `execUC` rather than dynamically like protocol parties.

C. Polynomial Bound

An important constribution of this work is supporting runtime analysis in UC with a notion of "polynomial" time. The UC framework provides a way to define poylynomial time computation and resource-bounds through the import mechanism. It ensures that a single ITM's execution is upper-bounded by some value $T(n)$ where T is a poynomial and n is the total units of import the ITM eve receives. An suitable analogy for import are tokens: a machine can receive tokens, send them out, and is limited in capability by the net number of tokens it holds.

In NomosUC, we build import into the type system and allow reasoning about ITMs being polynomial both locally and in the context of other ITMS (say, $\mathcal{F}$ is polynomial locally and when connected an adversary $\mathcal{A}$).

Definition V.2 (PPT Term)**.** A *PPT term* is a *well-typed* term $e(k, r)$ that is *closed* except for security parameter k, random bit sequence r.

We first-define terms that are well-typed in the traditional session-types-sense in Definition V.2, i.e. without any resource constraints [**?**]. Such terms are closed except for the security parameter k and some uniformly random bit sequence r.

However, we also want to reason about terms that are well-typed when connected to another Nomos terms. We introduce the term *well-matched* to mean a PPT term e is well-typed when connected to another term e'.i Simply put, the channels that e and e' share are of the same type. Specifically, we want to exclude processes that logically share a channel, say the channel from `p` to `f`, but send (their types message types don't match). This new definition becomes important when we discuss UC emulation below as we want to reason about environments that are *well-matched* for a protocol π or a specific adversary $\mathcal{A}$.

Definition V.3 (Well-Matched)**.**

$$\frac{\mathcal{T}_1, K\ ;\ \Delta_1 \vdash C_1 :: \Delta_1'\ ;\ \mathcal{T}_2, K'\ ;\ \Delta_2 \vdash C_2 :: \Delta_2' \qquad S \equiv \Delta_1 \bigcap \Delta_2 \neq \emptyset}{\Delta_1 \equiv_S \Delta_2\ ;\ K \equiv K'}\ \textsc{Well-matched}$$

Notice that in Definition V.3 we are concerned with two terms that are *open* even when connected. We only care about being well-matched, when connected to another term, on the channels over which they are connected.

Next we introduce our definition of a polynomial-bound in the security parameter k. Terms that are PPT in k are dubbed *well-resource-typed*.

Theorem 3 (PPT in k)**.** A *PPT Term* $e(k, r)$ is well-resource-typed if, given initial import $n(k) \in poly(k)$, there exists a polynomial T s.t. $\forall k, r, e(k, r)\{n(k)\}$ terminates in at most $T(n)$ steps.

Proof. The Nomos type system only type checks programs for which a satisfying assignment of polynomial T is possible. Given an $n \in poly(k)$, all programs that type check must be *well-resource-typed.* □

D. Emulation

A proof of security in the UC framework relies upon emulation of different executions.

In general, we say that a protocol π posesses the same security properties as another protocol ϕ if no environment can distinguish between them for any adversary. In most cases we compare a real protocol π with an idealized protocol $(\mathbb{I}_d, \mathcal{F})$ which is actually just an ideal functionality with dummy parties. The ideal functionality is known to achieve the desired security processes because it acts like a simple, trusted third party.

Given the random choices ITMs in UC can make, it is clear that the outputs of execUC produces and ensemble of distributions over all possible random bitstrings and security parameters. Emulation, then, is about the ensembles created by two UC environments being indistinguishable from each other. We define indistinguishabiliy between ensembles in a standard way using *statistical distance* in Definition V.4.

Definition V.4 (Indisinguishability)**.** Two ensembles $\mathcal{D}_{1,k}, \mathcal{D}_{2,k}$ are indistinguishable, $\mathcal{D}_{1,k} \sim \mathcal{D}_{2,k}$, if their statistical distance is at most $negl(k), \forall k$.

a) Valid Protocols: We want to make a clarification to what are considered *valid* functionalities, protocols, and environments. When we talk about a valid functionality, we expect it to take as parameters channels to/from $\mathcal{A}$, $\mathcal{Z}$, and the protocol wrapper. We also care about what it means for an entity, say an adversary, to be well matched with the functionality or protocol. It means that the types of the channels they expect are the same. Clearly, an adversary for protocol π_1 may not be well-matched with some other protool π_2. When talking about emulation we care about ensuring well-matchedness between adversaries and environments/protocols. We use the notation validP and validF to denote valid protocols and functionalities, and we use the notation $\mathcal{A} \leftrightarrow \mathcal{F}$ to denote that that the two processes are well-matched.

Indisintiguishability between two protocols is defined as follows (we shorten the communicator type comm to c):

Definition V.5 (Emulation)**.** Given two protocols $(\pi, \mathcal{F}_1), (\phi, \mathcal{F}_2)$ that are well-resource-typed then if $\forall \mathcal{A}$ well-matched with $(\pi, \mathcal{F}_1)$, $\exists \mathcal{S}$ s.t. $\forall \mathcal{Z}$ well-matched with $\mathcal{A}$ and $(\pi, \mathcal{F}_1)$: $\mathcal{S}$ is well-matched with $(\phi, \mathcal{F}_2)$, $\mathcal{Z}$ is well-matched with $(\phi, \mathcal{S})$, and $\mathsf{execUC}(\pi, \mathcal{F}_1, \mathcal{Z}, \mathcal{A}) \approx \mathsf{execUC}(\phi, \mathcal{F}_2, \mathcal{Z}, \mathcal{S})$:

$$\frac{\begin{array}{c}. \models \mathsf{execUC}[\mathcal{K}][\alpha] :: \Delta[\mathcal{K}][\alpha] \\ \mathsf{validP}\ \pi \to \Delta_1' \ ;\ \mathsf{validP}\phi \to \Delta_2' \ ;\ \langle \pi \leftrightarrow \mathcal{F}_2 \rangle, \langle \phi \leftrightarrow \mathcal{F}_1 \rangle \\ \Delta_1'[\mathcal{K}][\mathrm{T}_\pi] \equiv_{\mathcal{Z}} \Delta_1 \ ;\ \Delta_2'[\mathcal{K}][\mathrm{T}_\phi] \equiv_{\mathcal{Z}} \Delta_2 \\ \forall \mathcal{A}, (\exists (\Delta_4, \Delta_4') | \Delta_4 \vdash \mathcal{A} :: \Delta_4', \ \langle \mathcal{A} \leftrightarrow \pi \rangle, \langle \mathcal{A} \leftrightarrow \mathcal{F}_1 \rangle \\ \Rightarrow \exists (\Delta_3, \Delta_3') | \Delta_3 \vdash \mathcal{S}_\mathcal{A} :: \Delta_3', \langle \mathcal{S}_\mathcal{A} \leftrightarrow \phi \rangle, \langle \mathcal{S}_\mathcal{A} \leftrightarrow \mathcal{F}_2 \rangle \\ \forall \mathcal{Z} (\langle \mathcal{Z} \leftrightarrow \mathcal{A} \rangle, \langle \mathcal{Z} \leftrightarrow \pi \rangle \Rightarrow \langle \mathcal{Z} \leftrightarrow \mathcal{S}_\mathcal{A} \rangle, \langle \mathcal{Z} \leftrightarrow \phi \rangle \\ \mathsf{execUC}\ \pi\ \mathcal{Z}\ \mathcal{F}_1\ \mathcal{A} \approx\ \mathsf{execUC}\ \phi\ \mathcal{Z}\ \mathcal{F}_2\ \mathcal{S}_\mathcal{A}))\end{array}}{\lambda \mathcal{A}. \mathcal{S}_\mathcal{A} \vdash \mathcal{F}_1 \xrightarrow{\pi} \mathcal{F}_2}\ \text{EMULATE}$$

The emulation definition above starts with two processes π and ϕ that must conform to the type-parametric definition of all protocols and that they are respectively well-matched to two functionalities $\mathcal{F}_1$ and $\mathcal{F}_2$. It states that for all adversaries that are well-matched to both the protocol (real world) and the functionality (ideal world), there exists a well-matched simulator such that the two are indistinguisable for all environments. The definition ensures that for emulation to hold, the constructed simulator must be well-matched everywhere $\mathcal{A}$ is well-matched: for all environments $\mathcal{A}$ is well-matched with the $\mathcal{S}$ must also be well-matched with.

b) UC Realize: When we talk about emulation, we particularly care about emulation with respect to an ideal protocol ϕ which is really just $(\mathbb{I}_d, \mathcal{F})$ where $\mathbb{I}_d$ is the protocol which forwards all messages to/from $\mathcal{Z}$ and $\mathcal{F}$. We say the protocol π (potentially with a hybrid functionality $\mathcal{F}_1$) UC-realizes an ideal functionality $\mathcal{F}_2$ if Definition V.5 holds for $(\pi, \mathcal{F}_1)$ and $\phi = (\mathbb{I}_d, \mathcal{F}_2)$

Definition V.6 (UC-Realize)**.** A protocol π UC-realized an ideal functionality $\mathcal{F}_1$ if $(\pi, \mathcal{F}_2) \sim (\mathbb{I}_d, \mathcal{F}_1)$ for some $\mathcal{F}_2$.

E. Dummy Lemma

The Dummy Lemma is an important lemma in the UC framework that requires only one simulator to work with a dummy adversary in order to prove emulation with respect to any adversary. The proof of the lemma makes use of the withdrawTokens definition from Section III. The instruction allows for re-use of existing machines and make simulator construction to use the real-world adversary, or other sub-simulators, in a black-box manner.

The Lemma states that if dummy simulator satisfies emulation with respect to the dummy adversary, then for any $\mathcal{A}$ a simulator can be constructed with the dummy simulator. The constructed simulator simply runs $\mathcal{A}$ and $\mathcal{S}_D$ internally, and it sends messages from $\mathcal{Z}$ to $\mathcal{A}$ and outputs of $\mathcal{A}$ to $\mathcal{S}_D$. At a high leve, the proof relies on the emulation definition where dummy emulation covers environments that run $\mathcal{A}$ internally. Here, we are only moving $\mathcal{A}$ into the execution

Theorem 4 (Dummy Lemma)**.** If $\exists \mathcal{S}_D$ s.t. $\mathcal{A}_\mathcal{D}, \mathcal{S}_D \vdash \mathcal{F}_2 \xrightarrow{\pi} \mathcal{F}_1$ then $\forall \mathcal{A}\ \exists \mathcal{S}_\mathcal{A}$ s.t. $\mathcal{S}_\mathcal{A} \vdash \mathcal{F}_2 \xrightarrow{\pi} \mathcal{F}_1$)

Proof. The constructed simulator $\mathcal{S}_\mathcal{A}$ internally simulates $\mathcal{S}_D$ and $\mathcal{A}$ through a virtual token type K'. We describe the simulation pattern below to simulate messages to $\mathcal{S}_D$ and $\mathcal{A}$. Recall that the virtual tokens consturction is a tool to make witing complex protocols easier, and has no impact on the import token requirements of the simulating machine. Simply put, simulating as a black-box should be equivalent, with respect to import, as $\mathcal{S}$ running the code natively. The only difference in running a simulation internally is additional potential usage in using `withdrawToken` and routing messages.

On input from $\mathcal{Z}$ on channel z2p, $\mathcal{S}$:

```
msg = recv $z2a ;
get $z2a {z2an : K} ;
withdrawTokens f K K1 z2an ;
send $a_z2a msg ;
pay {z2an : K1} $a_z2a ;
```

Similarly, on output from $\mathcal{A}$ to a protocol party on channel a2p

```
pid = recv $a_a2p ;
msg = recv $a_a2p ;
get K1 $aa2p {a2pn} ;
send $sd_z2a A2P(pid, msg) ;
pay $sd_z2a {z2an : K1} ;
```

$\mathcal{S}_\mathcal{A}$ accepts input from $\mathcal{Z}$ and forwards it to the internal $\mathcal{A}$, which outputs to either the protocol parties or the ideal functionality. $\mathcal{S}$ forwards this output to $\mathcal{S}_D$ acting as input from the environment (here we fallback to the notion that $\mathcal{A}$ can be run internally by $\mathcal{Z}$) and forward any outputs it creates to the intended recipients. The UC framework proves the emulation definition, however our proof oblication is to ensure that such a simulator in NomosUC is well-resource-typed for all well-resource-typed $\mathcal{A}$. The $\mathcal{S}_\mathcal{A}$ performs constant overhead on the simulattion of $\mathcal{A}$ and $\mathcal{S}_D$. Therefore, a sufficient bounding polynomial on the runtime of $\mathcal{S}_\mathcal{A}$ can be given as:

$$T(n) = T_{\mathcal{A},\mathcal{S}_D}(n) + T_{\mathcal{A},\mathcal{S}_D}(n) + O(n)$$

where $T_{\mathcal{A},\mathcal{S}_D}(n)$ is the greater of the two bounding polynomials for $\mathcal{S}_D$ and $\mathcal{A}$ evaluated at n, and n is the import that $\mathcal{Z}$ sends to $\mathcal{A}$ and $\mathcal{S}$. We must also reason about the use of virtual tokens. Given that $\mathcal{A}$ and $\mathcal{S}_D$ are well-resource-typed we can conclude that the virtual import tokens generated for activating $\mathcal{A}$ and $\mathcal{S}_D$ never exceeds a polynomial in the number of real import tokens received by $\mathcal{S}$. □

F. Single Composition

In this section we present a simplified composition theorem and a second theorem we call the *squash theorem*. These two theorems combine later in the section to prove the full UC composition theorem as it appears in the UC framework [**?**].

The composition operator defines a way for a protocol ρ that uses a functionality $\mathcal{F}_1$ to swap it for a protocol $(\pi, \mathcal{F}_2)$, which realizes $\mathcal{F}_1$, such that $\mathcal{F}_1 \xrightarrow{\rho} \mathcal{F}_3 \Rightarrow \mathcal{F}_2 \xrightarrow{\rho\circ\pi} \mathcal{F}_3$. The $\circ$ composition operator is defined in Nomos in Figure 2.

Include a graphical illustration of wtf is going on, and going on inside the party wrapper as

Theorem 5 (Composition)**.**

$$\frac{\mathcal{F}_1 \xrightarrow{\pi} \mathcal{F}_2 \;;\; \mathcal{F}_2 \xrightarrow{\rho} \mathcal{F}_3}{\mathcal{F}_1 \xrightarrow{\rho\circ\pi} \mathcal{F}_3}\ \textsc{single-compose}$$

If *well-typed* $(\pi, \mathcal{F}_1)$ realizes $\mathcal{F}_2$ and $(\rho, \mathcal{F}_2)$ realizes some $\mathcal{F}_3$, then $(\rho \circ \pi, \mathcal{F}_2)$ is *well-typed* and realizes $\mathcal{F}_3$ when $\circ$ is defined as in Figure 2.

Proof. The pre-condition ensures the existence of a *well-resource-typed* simulator $\mathcal{S}_\pi$ for $(\pi, \mathcal{F}_1) \sim (\mathbb{I}_d, \mathcal{F}_2)$. We construct a simulator S which relies only on $\mathcal{S}_\pi$ for:

$$\mathsf{execUC}\ (\rho \circ \pi)\ \mathcal{F}_1\ \mathcal{Z}\ \mathcal{A} \approx \mathsf{execUC}\ \mathbb{I}_d\ \mathcal{F}_3\ \mathcal{Z}\ \mathcal{S}$$

We don't need to perform simulation on any inputs by $\mathcal{Z}$ to the main parties of ρ (it's the same protocol in both worlds). The constructed simulator $\mathcal{S}$ simulates $\mathcal{S}_\pi$ internally and passes messages intended for the parties of π, or for $\mathcal{F}_2$, to $\mathcal{S}_\pi$ and simulates its computation. Similariy, $\mathcal{S}$ sends any message from $\mathcal{F}_3$ to $\mathcal{S}_\pi$ for simulation. Input to any party of the main protocol ρ from $\mathcal{Z}$, or outout from them to $\mathcal{S}$, are forwarded without any modification or simulation. The constructed simulator performs constant overhead in routing messages to the simulated $\mathcal{S}_\pi$ and forwrading messages to/from parties of $\rho/\mathcal{Z}$. Given that $\mathcal{S}_\pi$ is *well-resource-typed*, with bounding polynomial $T_{\mathcal{S}_\pi}$, it suffices to show that an additional linear term is sufficient to create a bounding polynomial for $\mathcal{S}$.

□

We give a simpler, high-level idea of the proof here which can be understood visually:

$$\begin{aligned}
&\mathsf{execUC}\ \mathcal{Z}\ (\rho \circ \pi)\ \mathcal{F}_1\ \mathcal{A}_\mathcal{D} && (5)\\
\equiv\ &\mathsf{execUC}\ (\mathcal{Z} \circ \rho)\ \pi\ \mathcal{F}_1\ \mathcal{A}_\mathcal{D} && (6)\\
\approx\ &\mathsf{execUC}\ (\mathcal{Z} \circ \rho)\ \mathbb{I}_d\ \mathcal{F}_2\ \mathcal{S}_\pi && (7)\\
\equiv\ &\mathsf{execUC}\ \mathcal{Z}\ \rho\ \mathcal{F}_2\ \mathcal{S}_\pi && (8)
\end{aligned}$$

The $\equiv$ operator is a result of moving around ITMs (some from within other ITMs into the main UC execution) and $\sim$ refers to indistinguishability. In line (13) above, ρ is moved into the execution environment with an unchanged simulator as no additional simulation is required: the simulator allows unfettered communication between parties of ρ and $\mathcal{Z}$.

G. Multisession

The multi-session extension of a protocol or functionality, specified by the ! operator (such as $!\rho$ or $!\mathcal{F}$), allows multiple instances to be run within a sinlge ITM. The ITM simulates multiple instances of the protocol/functionality intnerally and multiplexes input/output to/from them in same way as the party wrapper for protocol parties. The channel from the protocol wrapper to the multisession operator can be typed as:

$$\textit{stype}\ \mathsf{P2MS[a]\{n\}} = \&\{\mathbf{push} : pid\hat{}ssid\hat{}a \to |\{n\} > \mathsf{P2MS[a]\{n\}}\} \quad (9)$$

```
proc compose[K][z2r][r2z][f2r][r2f][p2f][f2p] :
    (pid: Int), ($z_to_p: c[K][z2p]), ($p_to_z: c[K][r2z]),
    ($f_to_p: c[K][f2r]), ($p_to_f: c[K][r2f]) |– ($D : 1) =
{
        $rho_to_pi <– createchan[K][p2f];
        $pi_to_rho <– createchan[K][f2p];

         <– pi  <–                  $rho_to_pi $pi_to_rho $p_to_f $f_to_p ;
         <– phi <– $z_to_p $p_to_z $rho_to_pi $pi_to_rho ;
}
```

Fig. 2: Composition operator in Nomos that connects a protocol ρ to a protocol π that uses some functionality $\mathcal{F}$. The operators creates new channels to connect the realizing π and it's hybrid $\mathcal{F}$. Output from ρ intended for the replace functionality are actually send to parties of ρ, and channels outgoing from the parties to the functionality are given to π.

The operator accepts messages of the form (ssid, msg) from a particular pid, where ssid is a sub-session identifier. If an instance of the functionality with sid := ssid then $!\mathcal{F}$ creates one and forwards the message to it. Additionally, $!\mathcal{F}$ listens for outgoing messages from each of the instances and forwards them to the outside execution. The operator differs from the party wrapper in one crucial way: it only works with functional messages types and does not wrap around any session types like any other standalone functionality in Nomos UC.

The multisession behaves like the protocol wrapper in that we rely on code generation to create the operator for a particular functionality. The reason behind this is that the operator simulates many instances of a functionality and must use virtual tokens to communicate with them. For the commitment example we've used throughout this paper, the multisession needs only one virtual token type alongside the real token type. The commitment functionality doesn't internally simulate any other machines and therefore does not need any virtual token type itself. The process definition for $!\mathcal{F}_{\mathsf{com}}$ is shown in Figure 3 accepting two token types: the real token type K and the virtual token type K_1 for instances of $\mathcal{F}_{\mathsf{com}}$.

The communicators between $!\mathcal{F}$and the other ITMs all use the real token type. Only the internal channels that it creates use virtual token types. The communication pattern between the operator and the simulated functionalities works in the same was as Listing V-E.

Theorem 6 (PPT !)**.** If a functionality $\mathcal{F}$ is well-resource-typed, then it's multisession extension $!\mathcal{F}$ is well-resource-typed.

Proof. A *well-resource-typed* $\mathcal{F}$ guarantees a polynomial $T_{\mathcal{F}}$ bounding its execution. In the worse-case, the multisession operator must spawn a new instance of $\mathcal{F}$ an every activation. Let $N_{\mathcal{F}}$ denote the total number of instances (and, hence, number of activations) of $\mathcal{F}$ created by the operator. Note that $N_{\mathcal{F}}$ is polynomial in the security parameter k for all well-typed environments, protocols, and adversary. Therefore, there always exists a bounding polynomial to bound a polynomial number of simulated instances of $\mathcal{F}$. The polynomial can be given as:

$$P_{!\mathcal{F}}(n) = N_{\mathcal{F}} P_{\mathcal{F}}(n) + \mathcal{O}(N_{\mathcal{F}})$$

where the $\mathcal{O}(N_{\mathcal{F}})$ is due to the overhead of maintaining and accessing the set of all instances.

Similarly, $\mathcal{F}$ being *well-resource-typed* ensures a valid token context for all processes it may simulate. Therefore, it is clear that there exists a global connecting poltnomial f that ensures a valid token context for $!\mathcal{F}$. □

Theorem 7 (Squash Theorem)**.**

$$\frac{\textit{well-resource-typed } \mathcal{F}}{!\mathcal{F} \xrightarrow{\mathsf{squash}} !!\mathcal{F}} \text{ SQUASH}$$

Proof. First we describe the squash protocol in figure 4. Note that $!!\mathcal{F}$ is nested ! operators. The top level process maintains multiple sessions of $!\mathcal{F}$ each with their own ssid. Functionalities in each $!\mathcal{F}[\mathsf{ssid}]$ have their own sid.

In $(\mathbb{I}_d, !!\mathcal{F})$, $\mathbb{I}_d$ expects to receive messages of the form $(\mathsf{ssid}_1, (\mathsf{ssid}_2, m))$ where ssid_2 is a sub-session of $\mathcal{F}$ (i.e. instance) inside some $!\mathcal{F}$ with sub-session id ssid_1 inside of $!!\mathcal{F}$ (the message accesses functionality $!!\mathcal{F}[\mathsf{ssid}_1][\mathsf{ssid}_2]$). The squash protocol flattens the indexing of instances of $\mathcal{F}$ and combines session ids ssid_1 and ssid_2 into a single ssid: $\mathsf{ssid}_3 := \mathsf{ssid}_1 \cdot \mathsf{ssid}_2$. If follows intuitively that the view for the environment remains the same.

We construct a simulator such that:

$$\mathsf{execUC}\ \mathcal{Z}\ \mathbb{I}_d\ !!\mathcal{F}\ \mathcal{S}_{\mathsf{squash}} \approx \mathsf{execUC}\ \mathcal{Z}\ \mathsf{squash}\ !\mathcal{F}\mathcal{A}_{\mathcal{D}}$$

The simulator is very simple. Inputs to/from parties/$\mathcal{Z}$ for a corrupt party is forwarded unmodified. Input intended for $!\mathcal{F}$ of the form $(\mathsf{ssid}_1 \cdot \mathsf{ssid}_2, msg)$ is sent as $(\mathsf{ssid}_1, (\mathsf{ssid}_2, msg))$ to $!!\mathcal{F}$. Output from $!!\mathcal{F}$ is modified inversely and sent to $\mathcal{Z}$.

The proposed simulator is trivially analyzed to be *well-resource-typed*. It performs constant work per activation and does "real" simulation other than message modification to/from $!!\mathcal{F}$.

□

H. UC Composition

Composition in the UC setting is not limited to replacement of a single instance of a protocol. Instead, it permits replacement of any number of instances of a protocol ϕ, each with their own session id, with instances of a realizing protocol π.

```
type sid[a] = SID of String ^ a ;

proc bangF_1[K, K1][p2f][f2p][a2f][f2a]{p2fn}{f2pn}{a2fn} :
    ($pw_to_f: P2MS[K][p2f]), ($f_to_pw: MS2P[K][f2p]), ($f_to_a: MS2A[K][f2a]), ($a_to_f: A2MS[K][a2f]),
        ($1 : list[sender]), (\l2: list[scommitted]), ($l3: list[receiver]), ($l4: list[rcommitted]) |– ($ms: 1)
```

Fig. 3: The type definition for the multisession operator for functionalities and the correspond message type and import parameters.

```
type p2bFmsg[a] = P2bF of ssid ^ a ;
type p2bbFmsg[a] = P2bbF of ssid ^ ssid ^ a ;
(* z2p : comm[z2pmsg[p2bbf[a]]] *)
(* p2f : comm[p2fmsg[P2bf[a]]] *)
pid = recv $z2p ;
m = recv $z2p ;
case m (
  P2bbF(ssid1, ssid2, m) =>
    send $p2f P2bF(ssid1 + ssid2, m) ;
)
```

Fig. 4: The *squash protocol* accepts a message intended intended for $!!\mathcal{F}$ of the form $(\mathsf{ssid}_1, (\mathsf{ssid}_1, msg))$ and "flattens" it into a single $\mathsf{ssid}_3 = \mathsf{ssid}_1 + \mathsf{ssid}_2$ that can be passed to $!\mathcal{F}$. The result is the same number of instances of $\mathcal{F}$ but behind only a single ! operator.

This generalized form of composition follows directly from Theorems 5 and 7.

Theorem 8 (Composition)**.**

$$\frac{!\mathcal{F}_1 \xrightarrow{\pi} \mathcal{F}_2 \; ; \; !\mathcal{F}_2 \xrightarrow{\rho} \mathcal{F}_3}{!\mathcal{F}_1 \xrightarrow{\rho \circ !\pi \circ \mathsf{squash}} \mathcal{F}_3} \text{ COMPOSE}$$

Proof. The proof of full composition follows directly from the single composition Theorem 5 and the Squash Theorem 7. By Theorem 5 we can infer $!!\mathcal{F}_1 \xrightarrow{\rho \circ !\pi} \mathcal{F}_3$. Theorem 7 allows us to "squash" $!!\mathcal{F}_1$ and construct a simulator for $!\mathcal{F}_1 \xrightarrow{\rho \circ !\pi \circ \mathsf{squash}} \mathcal{F}_3$ □

VI. Commitment

In this section we fully flesh out the commitment example used throughout this paper. We implement the ideal functionality $\mathcal{F}_{\mathsf{com}}$ as well as a realizing protocol π_{com} using the random oracle $\mathcal{F}_{\mathsf{RO}}$. In doing so, we discuss how session types influence the design of the ideal functionality, and how the import mechanism is used. Finally, we present a simulatr for the dummy adversary to prove emulation.

A. The Random Oracle

The random oracle model assumes the existence of an idealized hash function in the real world, denoted by $\mathcal{F}_{\mathsf{RO}}$. We augment the random oracle functionality into $\mathcal{F}_{\text{P2P-RO}}$ the ability for parties to send messages to each other. However, we alter the design of this functionality compared to $\mathcal{F}_{\mathsf{com}}$ for two reasons: it interacts with a dynamic set of parties and a constraint imposed by session types. Usually, in p2p channel functionalities, $\mathcal{F}$ activates the receiving party with the message when delivering it. When combined with the random oracle functionality, $\mathcal{F}_{\text{P2P-RO}}$, at any given point, can receive input from the receiver or be asked to deliver a message to it by the sender. In a session type, the former is an *external choice* and the latter is an *internal choice*. Session types restrict to one or the other at any time and not potentially both. Therefore, for $\mathcal{F}_{\text{P2P-RO}}$ we opt to have one session-typed channel for all parties which incorporates the `pid` into the message:

$$\textit{stype}\ \mathsf{party}[a] = \&\{\mathbf{hash} : \mathsf{pid} \to \mathsf{int} \times \mathsf{hashing}[a], \mathbf{send} :$$

$$\mathsf{spid} \to \mathsf{rpid} \to \mathsf{a} \times \mathsf{sendmsg}[a]\}$$

$$\textit{stype}\ \mathsf{hashing}[a] = \oplus\{\mathbf{shash} : \mathsf{pid} \to \mathsf{int} \times \mathsf{party}[a]\}$$

$$\textit{stype}\ \mathsf{sendmsg}[a] = \oplus\{\mathbf{send} : pid \to \mathsf{a} \times \mathsf{party}[a]\}$$

The session type above suggests such a consruction is even more well-suited to $\mathcal{F}_{\text{P2P-RO}}$ because, as you can see, the session type returns to hashing on ever activation of $\mathcal{F}_{\text{P2P-RO}}$.

The corresponding functional message type between the protocl wrapper and $\mathcal{F}_{\text{P2P-RO}}$ is given by the simple types:

```
type rop2f = SHash of Int | Send of Int ;
type rof2p = Smsg of Int ;
```

Note that all messages from the protocol wrapper to the functionality are wrapped in `P2FMsg` which include the `pid` of the sending party.

B. Ideal World

The $\mathcal{F}_{\mathsf{com}}$ and presented earlier in this work accepts messages from parties on two channels: one from the sender and one from the receiver (refer to the previous section for their types). The channels don't directly connect to the two parties (recall the functionality wrapper and the protocol wrapper exists between the two).

We first describe at a high-level, the conversion happening within the functionality wrapper for $\mathcal{F}_{\mathsf{com}}$ with the `commit` message sent by the sender. When the wrapper receives the `commit` message, the functionality wrapper executes the following:

```
(* l1 : list[pid ^sender] *)
case $p2f (
  yes =>
    pid = recv $p2f ; msg = recv $p2f ;
    case msg (
      Commit(b) =>
```

```
        $ch_ = get_channel_by_pid pid $l1 $l2 ...
        $ch_.commit ;
        send $ch_ b ;
        $l2' <- append $ch_ $l2 ;
```

The wrapper also spawns a process to read from each of $\mathcal{F}_{\mathsf{com}}$'s outgoing channels, case analyze on their value and send out the corresponding functional message to the protocol wrapper.

a) The Ideal World Execution: We summarize the message types in use by describing the type parameters to the ideal world execution. For the ideal, `execUC` is invoked as follows (refer to the `execUC` definition in Figure **??** for what each of the parameters refers to):

```
type z2amsg[a][b] = Z2A2P of pid ^ a
                  | Z2A2F of b ;
(* ideal world *)
execUC[K1,K2][comf2p][comp2f][comp2f][comf2p]
  [comf2p][comp2f][comf2a][coma2f][rof2a][roa2f]
  [rof2p][rop2f]
```

Notice that the message types over `p2z` and over `p2f` are the same, because the ideal world parties are dummy parties which simply forward the messages to $\mathcal{F}_{\mathsf{com}}$. The type parameters for the adversary in the ideal world, though, still take the form of the types of $\mathcal{F}_{\mathsf{RO}}$. This is because the inputs $\mathcal{Z}$ gives to both worlds (the dummy adversary in the real world) is intended for $\mathcal{F}_{\mathsf{RO}}$ when communicating with the functionality through the adverasary or through corrupt parties

The party inputs from `a2p` in the ideal world are intended for $\mathcal{F}_{\mathsf{com}}$, not $\mathcal{F}_{\mathsf{RO}}$, so they are of the same type `comp2f`. Similarly, output from corrupt parties to the adversary in the ideal world is the same as output from $\mathcal{F}_{\mathsf{com}}$, and, therefore the message type is the same as `comf2p`. The messages type parameters are wrapped in channel-specific types as well. For example, the channel `z2a` is typed as follows:

```
type z2amsg[a][b] = Z2A2P of pid ^ a
                  | Z2A2F of b ;
#z_to_a: comm[z2amsg[a2p][a2f]]
(* a2p=rop2f, a2f=roa2f in execUC above*)
```

b) Ideal Protocol: The ideal world protocl is a dummy party which forwards all messages to the functionality. The message content from `z2p` and `p2f` is the same in the ideal world, but it is wrapped in different parameteric typed when it is sent from the `z` or `p`. As shown below the messages themselves are the same (`comp2f` but typed differently (`z2pmsg` vs `p2fmsg`):

```
type z2pmsg[a] = Z2P of pid ^ a ;
type p2fmsg[a] = P2F of pid ^ a ;
#z_to_p: comm[z2pmsg[comp2f]] ;
#p_to_f: comm[p2fmsg[comp2f]] ;
```

The protocol wrapper only needs to forward the message unaltered but using a different type construtor.

C. Real World

The real world protocol π_{com} necessarily accepts the same message type, `z2pmsg[comp2f]`, as the ideal protocol to ensure its not trivially distinguishable to all environments. It also usesa hybrid functionality: the random oracle, or $\mathcal{F}_{\mathsf{RO}}$.

```
case $z2p (
  commit =>
    b = recv $z2p ;
    bits = sample k r ;
    $p2f.hash ;
    send $p2f b + bits ;
    case $p2f (
      hash =>
        h = recv $p2f ;
        $p2f.send ;
        send $p2f hash ;
```

Fig. 5: The code for the committer in π_{com} when it receives a commit message from $\mathcal{Z}$. It obtains a hash of the message from $\mathcal{F}_{\mathsf{RO}}$ over p2f and sends it to the receiver through the same functionality.

```
case $f2p (
  hashmsg =>
    $p2z.committed ;
    case $f2p (
      openmsg =>
        h = recv $f2p ;
TODO: finish this code snippet
```

Fig. 6: The code for the receiver when it is activated by the committer's commitment with a hashmsg. Its output to $\mathcal{Z}$ is the same as $\mathcal{F}_{\mathsf{com}}$: a committed message. It then waits to receive the opening of the commitment and confirms with that random oracle that it is correct.

The protocol wrapper performs the message conversion outlined in the previous section. Here we only present what the protocol party itself does with session typed channels. We provide the simple code of the committer in Figure 5.

Similarly, the receiver, which doesn't pass any input to $\mathcal{F}_{\mathsf{RO}}$ informs $\mathcal{Z}$ when a commitment has been made in Figure 6.

The protocol works as follows:

1) When the committer receives a `Commit(b)` message from $\mathcal{Z}$, it samples some random bits r and generates a hash h by sending `SHash(b + r)` to $\mathcal{F}_{\mathsf{RO}}$.
2) It then sends the commitment to the receiver who notifies $\mathcal{Z}$ with a `committed` message.
3) Finally, when $\mathcal{Z}$ instructs the committer to `Open` the commitment, it sends bit `b` and randomness `r` to the receiver. The receiver checks the commitment, with `b` and `r`, against $\mathcal{F}_{\mathsf{RO}}$ and outputs `Open(b)` to $\mathcal{Z}$ if it checks out.

D. Simulation

The simulator for commitment in the random oracle model relies on simulating the random oracle internally and creating/storing entries queried by the environment. There are two corruptions modes for the protocol: the committer is corrupt or the receiver is corrupt. When the receiver is

corrupt, the simulator generates some random commitment value h when it receives `Committed` from $\mathcal{F}_{\mathsf{com}}$ and forwards `P2A2Z(pid,SMsg(h))` to $\mathcal{Z}$. Finally, when the committed bit is revealed, the simulator returns `SMsg(b + x)`, where x is some randomly generated nonce, and stores the mapping $b + x \rightarrow h$ in its internal instance of $\mathcal{F}_{\mathsf{RO}}$.

```
m = recv $p2a ;
case m (
  P2A(pid, msg) =>
        case msg (
      Committed =>
        h = sample r k ;
        send $z2a P2A2Z(pid, SMsg(h)) ;
      Open(b) =>
            x = sample r k ;
        send $z2a P2A2Z(pid, SMsg(b, x)) ;
```

When the committer is corrupt, the simulator has to do more work. The key point that makes commitment in the RO model realizable is that $\mathcal{Z}$ acquires commitment to give to the corrupt committer from $\mathcal{F}_{\mathsf{RO}}$ through $\mathcal{A}$. In th ideal world, the simulator recrods all the (key,value) pairs generated by its simulation of $\mathcal{F}_{\mathsf{RO}}$ and therefore can always determine the bit $\mathcal{Z}$ is committed to. When $\mathcal{Z}$ queries $\mathcal{S}$ with `Z2A2F(SHash(b + x))` it stores the value and generates a hash value for it and returns it to $\mathcal{Z}$. When instructed to give input to the corrupt party the simulator, $\mathcal{S}$ gives input `A2P(1, Commit(b))` where b is the bit whos hash was requested.

There is a unique edge case where $\mathcal{Z}$ give some random bit sequence to §as the commitment which wasn't generated by the random oracle. In this case §chooses a bit at random annd passes `A2P(1, Commit(b))` to the protocol wrapper where b is randomly selected. When prompted to open the commitment, §does nothing as the real world receiver would fail to confirm whether the commitment corresponds to the bit b.

ACKNOWLEDGMENT

REFERENCES

APPENDIX