Secure Synthesis of Distributed Cryptographic Applications

Coşku Acay Joshua Gancher Rolph Recto Andrew C. Myers

Abstract—**Developing secure distributed systems is difficult, and even harder when advanced cryptography must be used to achieve security goals. Following prior work, we advocate using *secure program partitioning* to synthesize cryptographic applications: instead of implementing a system of communicating processes, the programmer implements a *centralized, sequential* program which is automatically compiled into a secure distributed version that uses cryptography.**

While this approach is promising, formal results for the security of such compilers are limited in scope. In particular, no security proof yet simultaneously addresses subtleties essential for robust, efficient applications: multiple cryptographic mechanisms, malicious corruption, and asynchronous communication.

In this work, we develop a compiler security proof that handles these subtleties. Our proof relies on a novel unification of simulation-based security, information-flow control, choreographic programming, and sequentialization techniques for concurrent programs. While our proof targets hybrid protocols, which abstract cryptographic mechanisms as idealized functionalities, our approach offers a clear path toward leveraging Universal Composability to obtain end-to-end, modular security results with fully instantiated cryptographic mechanisms.

Finally, following prior observations about simulation-based security, we prove that our result guarantees *robust hyperproperty preservation*, an important criterion for compiler correctness that preserves all source-level security properties in target programs.

I. Introduction

Ensuring security for modern distributed applications remains a difficult challenge, as such systems can cross administrative boundaries and involve parties that do not fully trust each other. To defend their security policies, some applications employ sophisticated mechanisms such as complex distributed protocols [1, 2], trusted hardware [3, 4, 5], and advanced uses of cryptography. These technologies add significant complexity to software development and require expertise to use successfully [6, 7, 8].

To ease the development of secure distributed applications, prior work leverages compilers that translate high-level programs into distributed protocols that employ advanced security mechanisms. Unfortunately, most compilers only target a single mechanism—such as multiparty computation [9, 10, 11, 12], zero-knowledge proofs [13, 14, 15, 16], or homomorphic encryption [17, 18, 19]—and thus do not support secure combinations of mechanisms. On the other hand, compilers that perform *secure program partitioning* [20, 21, 22, 23, 24, 25] do combine mechanisms, but come with limited or informal correctness guarantees.

In this work, we give the first formal security result for program partitioning that targets multiple cryptographic mechanisms, arbitrary corruption, and adversarially controlled scheduling. We formalize our result in the *simulation-based* security framework, which establishes a modular foundation for cryptographic protocol security [26]. Our security proof is primarily concerned with program partitioning itself, and thus does not reason about the concrete instantiation of cryptographic mechanisms; however, we discuss how to extend our results to reason about concrete mechanisms.

Our security proof incorporates multiple techniques for simulation-based security: information-flow type systems [27] to define the security policy and to guide partitioning, choreographies [28] to define global programs for distributed executions, and a novel information-flow guided technique for concurrent program sequentialization [29].

- We formalize a variant of Simplified Universal Composability (SUC) [30] enriched with information flow, allowing us to capture distributed protocols in the presence of adversarial scheduling and corruption.
- We show how to model secure program partitioning using security-typed choreographies [28]. The input to program partitioning is a sequential program representing an idealized execution on a single, trusted security domain, while the output is a distributed protocol with message-passing concurrency between mutually distrusting agents.
- We prove simulation-based security for our model of program partitioning. Informally, we show that any adversary interacting with the compiled distributed program is no more powerful than a corresponding adversary (a *simulator*) interacting with the source program.
- We show that, in our setting, simulation implies *robust hyperproperty preservation* [31], a strong criterion for compiler correctness that ensures security conditions of source programs are preserved in target programs.

II. Overview

We illustrate compilation via the classic Millionaires' Problem [32], expressed as the source program in fig. 2a. Here, Alice and Bob learn who is richer without revealing their net worth to each other. To do so, the program collects inputs from Alice and Bob representing their net worth (lines 1 and 2); compares these (line 3), and outputs the result to Alice and Bob (lines 4 and 5).

A. Information Flow Control

Source programs prevent insecure information flows using a security type system [27], which assigns a *label* to every variable. Labels track the *confidentiality* and *integrity* of data.

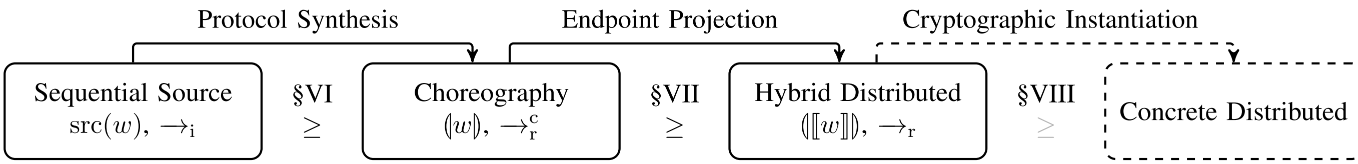


Figure 1. Overview of compilation and the correctness proof. Left-to-right arrows are compilation steps; $\geq$ are proof steps. Term w is a choreography, $[\![\cdot]\!]$ is endpoint projection, $\mathtt{src}(\cdot)$ is the inverse of protocol synthesis, and $(\!|\cdot|\!)$ models corruption. Dashed components represent proof sketches.

```
let a : A = input Alice;
let b : B = input Bob;
let x = declassify(endorse a < endorse b,
                   A ∧ B → A ⊓ B);
output x to Alice
output x to Bob
```

(a) Source program with information-flow labels.

```
let Alice.a = input;
Alice.a ⇝ MPC(Alice, Bob).a′;
let Bob.b = input;
Bob.b ⇝ MPC(Alice, Bob).b′;
let MPC(Alice, Bob).x =
    declassify (endorse a′ < endorse b′);
MPC(Alice, Bob).x ⇝ Alice.x₁;
MPC(Alice, Bob).x ⇝ Bob.x₂;
output x₁ to Alice
Alice.0 ⇝ Bob._; // Sync outputs
output x₂ to Bob
```

(b) Choreography with explicit communication and synchronization.

```
// Alice
let a = input;
send a to MPC(Alice, Bob)
let x₁ = receive MPC(...);
output x₁
send 0 to Bob // Sync
```

```
// Bob
let b = input;
send b to MPC(Alice, Bob)
let x₂ = receive MPC(...);
let _ = receive Alice; // Sync
output x₂
```

```
// MPC(Alice, Bob)
let a′ = receive Alice;
let b′ = receive Bob;
let x = declassify (endorse a′ < endorse b′);
send x to Alice
send x to Bob
```

(c) Hybrid distributed program derived by projecting choreography.

```
// Alice
let a = input;
// Calls to MPC library
output x₁
send 0 to Bob // Sync
```

```
// Bob
let b = input;
// Calls to MPC library
let _ = receive Alice; // Sync
output x₂
```

(d) Concrete distributed program derived by instantiating idealized hosts.

Figure 2. Compiling the Millionaires' Problem

Our security type system follows prior work [33, 34] in using *downgrading mechanisms*—**declassify** and **endorse** expressions—to selectively allow information flows that would otherwise be deemed insecure. As in prior work [34], the type system constrains these downgrading mechanisms to prevent improper usage. These constraints turn out to be crucial.

In fig. 2a, the **declassify** expression explicitly allows revealing the result of the comparison $a < b$ to Alice and Bob, which is by default disallowed since the computation reads secrets from both parties. Dually, the **endorse** expressions allow untrusted data coming from Alice and Bob to influence the output from the comparison, which must be trusted since the value is output to both parties.

Downgrading require explicit *source* and *target* labels. Figure 2a suppresses these labels for the **endorse** expressions, but shows the **declassify** expression that moves from $\mathrm{A} \wedge \mathrm{B}$ to $\mathrm{A} \sqcap \mathrm{B}$. Label $\mathrm{A} \wedge \mathrm{B}$ is too secret for either Alice or Bob to see the value; label $\mathrm{A} \sqcap \mathrm{B}$ allows *both* parties to see it.

B. Compilation

Source programs serve as *specifications* of intended behavior, and correspond to *ideal functionalities* from Universal Composability [26]. Source programs act as trusted third parties, perfectly and securely executing the program on behalf of the involved hosts. The source language is high-level by design, and its simple, sequential semantics facilitate reasoning. Our compiler generates a concurrent distributed program that correctly implements the same behavior.

Figure 1 gives an overview of the compilation pipeline. First, *protocol synthesis* compiles the source program into a *choreography*, a single, centralized program that represents a distributed computation between many hosts. In addition to *parties* such as Alice and Bob, choreographies may mention *idealized hosts* such as MPC(Alice, Bob), which represents a maliciously secure multiparty computation protocol between Alice and Bob. Idealized hosts can perform computations that require high confidentiality or integrity. Next, *endpoint projection*, a standard procedure in choreographic programming [35, 36], partitions the choreography into a distributed program, where hosts run in parallel and interact via message passing. The distributed program still contains idealized hosts, so it corresponds to a *hybrid program* in UC. Finally, *cryptographic instantiation* replaces idealized hosts with concrete cryptographic algorithms.

Figure 2b shows a choreography where Alice and Bob perform their respective **input** and **output** statements, while MPC(Alice, Bob) does the comparison. Explicit communication statements move data between hosts. Choreographies have *concurrent* semantics, so statements at different hosts may step out of program order. The penultimate line has *synchronization* between Alice and Bob: Bob must wait on an input from Alice before delivering output. This synchronization step is necessary for the distributed program to match the sequential source program, in which Bob's output happens after Alice's. Figure 2c shows the distributed program obtained by projecting the choreography in fig. 2b. Endpoint projection converts communication statements to **send/receive** pairs, and projects local computations to their corresponding hosts. Finally, fig. 2d shows the result of cryptographic instantiation. Each **send/receive** statement that interacts with MPC(Alice, Bob) is replaced with a call to a cryptographic library.

C. Defining Correctness

Our main contribution is a proof that compilation is correct. A correct compiler preserves properties of source programs in generated target programs. For generality, we demand that the compiler preserve all *hyperproperties* [37] guaranteed by the source program. Hyperproperties capture many common notions of security, including secure information flow.

Formally, preserving all hyperproperties is defined via *robust hyperproperty preservation* (RHP) [31]. Following prior observations [38, 39], we do not prove RHP directly, but instead prove *simulation*, which we show implies RHP in our framework. Simulation requires every attack by an adversary against the target program to be possible against the source program. Ideally, the source program is "obviously secure," meaning it has straightforward, sufficiently abstract semantics and a narrow attack surface. In contrast, the target program faithfully models real code and has a realistic attack surface.

As with UC, our framework abstracts concrete cryptographic mechanisms as idealized hosts, yielding an *extensible* proof that is generic over the set of supported cryptographic mechanisms. Indeed, we sketch how our framework may be embedded into UC to leverage the UC composition theorem [26]; using the composition theorem, we can instantiate idealized hosts with cryptographic mechanisms proven secure separately.

a) Threat Model: Simulation demands we carefully define the capabilities of adversaries for each language in the pipeline. Adversaries are characterized by two sets of labels $\mathcal{P}$ and $\mathcal{U}$ representing *public* and *untrusted* labels. These label sets induce a partitioning of hosts into three sets: *honest* (secret and trusted), *semi-honest* (public and trusted), and *malicious* (public and untrusted). Intuitively, malicious hosts are fully controlled by the adversary, and semi-honest hosts follow the protocol but leak all their data to the adversary [40]. We say a host is *dishonest* if it is semi-honest or malicious, and *nonmalicious* if it is honest or semi-honest.

Source programs are fully sequential, so adversaries do not control scheduling. Moreover, adversaries cannot read or change intermediate data within a source program. However, adversaries can read messages from **input/output** expressions involving dishonest hosts, read the results of **declassify** expressions with public target labels, and change the results of **endorse** expressions with untrusted source labels.

Choreographies and hybrid distributed programs have the same semantics, so their adversaries have equal power. These programs are concurrent and the adversary controls all scheduling. The adversary also fully controls malicious hosts and can read messages involving at least one semi-honest host. However, the adversary cannot carry out computational attacks, since cryptographic mechanisms are modeled as idealized hosts.

Hosts	h	$\in$	$\mathbb{H}$
Endpoints	c	$\in$	$\mathbb{C} = \{\mathrm{Adv}, \mathrm{Env}\} \cup \mathbb{H}$
Values	v	$\in$	$\mathbb{V} = \{0, \ldots\}$
Messages	$m \in \mathbb{M}$	$::=$	$c_1 c_2 v$
Actions	$a \in \mathbb{A}$	$::=$	$?m \mid\ !m$

$$\boxed{\mathrm{actor}(a) = c} \quad \mathrm{actor}(?c_1 c_2 v) = c_2 \qquad \mathrm{actor}(!c_1 c_2 v) = c_1$$

Figure 3. Syntax of messages and actions.

The adversary can view all message *headers* (source and destination), but not necessarily message *content*. The adversary may not drop, duplicate, or modify messages. This abstraction of secure channels can be realized by standard techniques, such as TLS [41]. In our model as in most models of cryptographic protocols [26, 42], the adversary can exploit *timing* and *progress* channels since it controls scheduling. These channels make secret control flow insecure: any discrepancy in timing or progress behavior between different control-flow paths can be detected by the adversary. Therefore, we only prove security for programs that make control flow decisions based on public, trusted data.

D. Proving Correctness

Simulation is transitive, which allows breaking the correctness proof into smaller simulations. First, we prove that sequential source programs are simulated by the choreographies the compiler generates. There is a wide semantic gap between choreographies and source programs: choreographies are concurrent and specify security through message passing, whereas source programs are sequential and specify security through information-flow labels (**declassify/endorse** expressions). We split the correctness of protocol synthesis into intermediate steps. The primary challenge of moving from source programs to choreographies is ensuring that choreographies are *well-synchronized*; i.e., that they faithfully realize a sequential program, even though they have concurrent semantics.

Second, we prove that choreographies are simulated by their endpoint projections. The choreographic programming literature [28, 35, 36, 43, 44] establishes a strong correspondence between the semantics of choreographies and their projections. We extend this connection to handle corruption, and establish the correctness of endpoint projection.

Finally, we sketch how hybrid distributed programs are simulated by concrete distributed programs, which make use of actual cryptographic mechanisms. In particular, we show how to embed our framework in the SUC [30] framework, which in turn embeds into the full UC framework. We can then leverage the UC composition theorem to instantiate idealized hosts one at a time, appealing to existing correctness proofs for cryptographic mechanisms.

III. Semantic Framework

We capture the semantics of programs using labeled transition systems (LTSs), where labels are *actions* a drawn from the grammar in fig. 3. An action a is either the input $?m$ or

Processes	w		
Configurations	W	$::=$	$w_1 \parallel \cdots \parallel w_n$

$$\boxed{W \xrightarrow{a} W'}$$

$$W\text{-INPUT}\quad \frac{\forall i \,.\, w_i \xrightarrow{?m} w_i'}{w_1 \parallel \cdots \parallel w_n \xrightarrow{?m} w_1' \parallel \cdots \parallel w_n'}$$

$$W\text{-OUTPUT}\quad \frac{w_i \xrightarrow{!m} w_i' \qquad \forall j \neq i \,.\, w_j \xrightarrow{?m} w_j'}{w_1 \parallel \cdots \parallel w_n \xrightarrow{!m} w_1' \parallel \cdots \parallel w_n'}$$

Figure 4. Syntax and semantics of configurations.

the output $!m$ of a message m. A message m specifies the endpoints c_1 and c_2 of communication and carries a value v. An endpoint is either a host h, the adversary Adv, or the external environment Env. Values are drawn from an arbitrary set $\mathbb{V}$, which we assume contains at least 0. We define $\mathrm{actor}(a)$ as the host performing a: the sender performs output actions and the receiver performs input actions. Internal steps are represented as self-communication $!hh0$; which allows identifying the host making progress without adding a new syntactic form.

a) Configurations and Parallel Composition: A configuration, W, is the parallel composition of a finite set of processes w_i, which are arbitrary LTSs. Following prior work [45, 46], processes must be *input-total*: for every state w and input message $?m$, there exists a state w' such that $w \xrightarrow{?m} w'$. Figure 4 gives the semantics of configurations. A configuration W steps with an input $?m$ if all processes in W do, and steps with an output $!m$ if one of the processes outputs m, and the rest input m.

b) Adversaries: As with processes, an adversary $\mathcal{A}$ or $\mathcal{S}$ is an arbitrary LTS. The rules for running an adversary in parallel with a configuration are the same as in fig. 4. In contrast to processes, adversaries are *not* input-total, which enables adversaries to control scheduling: to schedule an endpoint c_1, $\mathcal{A}$ refuses to step with actions of the form $?c_1'c_2m$ where $c_1' \neq c_1$, but accepts actions $?c_1c_2m$.

Due to the definition of parallel composition, a copy of every message from the configuration and the environment is delivered to the adversary; and any output of the adversary is delivered to the configuration and the environment. However, the adversary can only read a message if at least one endpoint is dishonest, and can only forge a message from a malicious host.

Definition III.1 (Adversary Interface)**.** For all $\mathcal{A}$:

1) If c_1 and c_2 are honest, then $\mathcal{A} \xrightarrow{?c_1c_2v_1} \mathcal{A}'$ if and only if $\mathcal{A} \xrightarrow{?c_1c_2v_2} \mathcal{A}'$ for all v_1 and v_2.
2) If $\mathcal{A} \xrightarrow{!c_1c_2v}$, then either $c_1 = \mathrm{Adv}$ or c_1 is malicious.

c) Determinism: To match UC, the adversary must resolve all nondeterminism, so that $\mathcal{A} \parallel W$ is deterministic. We ensure determinism with the following restrictions.

- Configurations and adversaries are *internally deterministic*: if $w \xrightarrow{a} w_1$ and $w \xrightarrow{a} w_2$, then $w_1 = w_2$.
- Adversaries are *output deterministic*: if $\mathcal{A} \xrightarrow{!m_1}$ and $\mathcal{A} \xrightarrow{!m_2}$, then $m_1 = m_2$.
- Configurations are *output deterministic per channel*: if $w \xrightarrow{!m_1}$, $w \xrightarrow{!m_2}$, and $\mathrm{actor}(!m_1) = \mathrm{actor}(!m_2)$, then $m_1 = m_2$.
- Adversaries are *channel selective*: if $w \xrightarrow{?m_1}$ and $w \xrightarrow{?m_2}$, then $\mathrm{actor}(?m_1) = \mathrm{actor}(?m_2)$.

d) Simulation: Simulation determines when a configuration W_2 securely realizes configuration W_1: that is, if *every* adversary $\mathcal{A}$ interacting with W_2 can be simulated by another adversary $\mathcal{S}$ (with the same interface) running against W_1 [26]. The latter adversary is called a *simulator*.

Definition III.2 (Simulation)**.** W_1 is simulated by W_2, written $W_1 \geq W_2$, when the two systems are indistinguishable to *any* external environment:

$$\forall \mathcal{A} \,.\, \exists \mathcal{S} \,.\, \mathbb{T}_{\mathrm{Env}}(\mathcal{S} \parallel W_1) = \mathbb{T}_{\mathrm{Env}}(\mathcal{A} \parallel W_2)$$

Here, $\mathbb{T}_{\mathrm{Env}}(W)$ is the set of *traces* of W but containing only the actions that communicate with the environment. Given trace $tr = a_1, \ldots, a_n$, we have $\mathbb{T}_{\mathrm{Env}}(W) = \{tr_{|\mathrm{Env}} \mid W \xrightarrow{tr}\}$, where restriction $tr_{|\mathrm{Env}}$ removes all actions in tr where neither the source nor the destination is Env.

Our definition of simulation guarantees *perfect* (i.e., information-theoretic) security. In §VIII, we discuss how to transfer our results to the SUC framework.

IV. Specifying Security Policies

To succinctly capture both security policies and the adversary's power, we use a label model that can describe confidentiality and integrity simultaneously [47, 48, 49].

A security label $\ell \in \mathbb{L}$ is a pair of the form $\langle p, q \rangle$ where p and q are elements of an arbitrary bounded distributive lattice $\mathbb{P}$. Here, p describes confidentiality and q describes integrity. Elements of $\mathbb{P}$ are called *principals*. Principals can be thought of as negation-free boolean formulas over a set $\{\mathrm{A}, \mathrm{B}, \mathrm{C}, \ldots\}$ of *atomic principals*.

The *acts-for* relation ($\Rightarrow$) orders principals by authority, and coincides with logical implication: for example, $p \wedge q \Rightarrow p$ and $q \Rightarrow p \vee q$. The most powerful principal is $\mathbf{0}$ and the least powerful, $\mathbf{1}$, so we have $\mathbf{0} \Rightarrow p \Rightarrow \mathbf{1}$ for any principal p.

We lift $\wedge$, $\vee$, and $\Rightarrow$ to labels in the obvious pointwise manner. Whenever appropriate, we write p for the security label $\langle p, p \rangle$. Confidentiality and integrity projections $\ell^{\rightarrow}$ and $\ell^{\leftarrow}$ completely weaken the other component of a label: $\langle p, q \rangle^{\rightarrow} = \langle p, \mathbf{1} \rangle$ and $\langle p, q \rangle^{\leftarrow} = \langle \mathbf{1}, q \rangle$.

As in FLAM [49], the authority ordering on principals defines secure information flow. Flow policies become more restrictive as they become *more* secret and *less* trusted:

$$\begin{aligned} \langle p_1, q_1 \rangle \sqsubseteq \langle p_2, q_2 \rangle &\iff p_2 \Rightarrow p_1 \text{ and } q_1 \Rightarrow q_2 \\ \langle p_1, q_1 \rangle \sqcup \langle p_2, q_2 \rangle &= \langle p_1 \wedge p_2, q_1 \vee q_2 \rangle \\ \langle p_1, q_1 \rangle \sqcap \langle p_2, q_2 \rangle &= \langle p_1 \vee p_2, q_1 \wedge q_2 \rangle \end{aligned}$$

The least restrictive information flow policy is $\mathbf{0}^{\leftarrow}$ ("public trusted"), describing information that can be used anywhere, while the most restrictive is $\mathbf{0}^{\rightarrow}$ ("secret untrusted").

A. Authority of Hosts

Protocol synthesis places computations on hosts that have enough authority to securely execute them. The authority of each host h is captured with a label $\mathbb{L}(h)$ [20]. For our example, we take $\mathbb{L}(\text{Alice}) = \text{A}$ and $\mathbb{L}(\text{Bob}) = \text{B}$. Following an insight from Viaduct [25], idealized hosts like $\text{MPC}(\text{Alice}, \text{Bob})$ have a derived label that conservatively approximates the security guarantees of the cryptographic mechanism. For maliciously secure MPC, a reasonable label is $\mathbb{L}(\text{MPC}(\text{Alice}, \text{Bob})) = \mathbb{L}(\text{Alice}) \wedge \mathbb{L}(\text{Bob}) = \text{A} \wedge \text{B}$, meaning that $\text{MPC}(\text{Alice}, \text{Bob})$ may view secrets of Alice and Bob, and also has enough integrity to compute values for them.

B. Capturing Attacks with Labels

The power of the adversary is determined by partitioning labels $\mathbb{L}$ across the two axes: public/secret and trusted/untrusted; we denote these sets as $\mathcal{P}/\mathcal{S}$ and $\mathcal{T}/\mathcal{U}$, respectively. We only consider sets that form *valid attacks*: these are sensible restrictions on how these sets might be chosen [34], discussed in §A. The rest of our development is parameterized over a valid partitioning.

Recalling the threat model from §II-C, an honest host has a secret, trusted label ($\mathbb{L}(h) \in \mathcal{S} \cap \mathcal{T}$); a semi-honest host has a public, trusted label ($\mathbb{L}(h) \in \mathcal{P} \cap \mathcal{T}$); and a malicious host has a public, untrusted label ($\mathbb{L}(h) \notin \mathcal{T}$). A host with a secret, untrusted label does not make any sense: an untrusted host is fully controlled by the adversary, so it cannot hide information from the adversary. We rule out such corruptions by requiring all host labels to be *uncompromised* [50]. A valid partitioning never classifies an uncompromised label as secret and untrusted.

Definition IV.1 (Uncompromised Label)**.** Label $\ell = \langle p, q \rangle$ is uncompromised, written $\blacktriangledown \ell$, if it is at least as trusted as it is secret: $q \Rightarrow p$.

Theorem IV.2. *Under a valid attack, if $\blacktriangledown \ell$, then we have $\ell \notin \mathcal{S} \cap \mathcal{U}$.*

V. Protocol Synthesis

The first program transformation, protocol synthesis, takes a sequential source program to a choreography.

A. Source Language

Figure 5 gives the syntax of source programs. The language supports an abstract set of operators f over values. We distinguish pure, atomic expressions t from expressions e that may have side effects. The **declassify** expression marks locations where private data is explicitly allowed to flow to public data. Similarly, **endorse** marks where untrustworthy data may influence trusted data. The **input**/**output** expressions allow programs to interact with the external environment [51, 52].

Statement **let** $h.x = e; s$ performs the local computation e at host h, binds the result to variable x, and continues

Variables $x \in \mathbb{X}$ Labels $\ell \in \mathbb{L}$ Operators $f \in \mathbb{F}$

$$
\begin{array}{llcl}
\text{Atomic Expr.} & t & ::= & v \mid x \\
\text{Expressions} & e & ::= & f(t_1, \ldots, t_n) \\
 & & \mid & \mathbf{declassify}(t, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) \\
 & & \mid & \mathbf{endorse}(t, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) \\
 & & \mid & \mathbf{input} \mid \mathbf{output}\ t \\
\text{Statements} & s & ::= & \mathbf{let}\ h.x = e; s \\
 & & \mid & \mathbf{if}(h.t, s_1, s_2) \\
 & & \mid & \mathbf{skip} \\
\text{Buffers} & B & \in & \mathbb{C} \times \mathbb{C} \to \mathbb{V}^* \\
\text{Processes} & w & ::= & \langle \mathbb{H}, B, s \rangle
\end{array}
$$

Figure 5. Syntax of the source language.

$$
\begin{array}{llcl}
\text{Expressions} & e & ::= & \ldots \mid \mathbf{receive}\ h \mid \mathbf{send}\ t\ \mathbf{to}\ h \\
\text{Statements} & s & ::= & \ldots \mid h_1.t \rightsquigarrow h_2.x; s \mid h_1[v] \rightsquigarrow h_2; s \\
\text{Processes} & w & ::= & \langle H \subseteq \mathbb{H}, B, s \rangle
\end{array}
$$

Figure 6. Syntax of choreographies as an extension to source syntax (fig. 5). The **send**/**receive** expressions are only relevant for the security proof.

as s. In source programs, h is only relevant for **input** and **output** expressions: we write **let** $h.x =$ **input** and **let** $h.x =$ **output** t in contrast to our example in fig. 2a, where we write **let** $x =$ **input** h and **let** $x =$ **output** t **to** h. For all other expressions, h is $*$, a single fully trusted host. Representing source programs with host annotations allows smoothly extending the language later on.

A source-program configuration is a logically centralized process $\langle \mathbb{H}, B, s \rangle$. The component $\mathbb{H}$ indicates the process acts for all hosts. The second component, B, is a *buffer* mapping pairs of endpoints to first-in-first-out queues of values. Processes buffer input so that output on other processes (the adversary and the environment) is nonblocking.

B. Choreography Language

Choreographies are centralized representations of distributed computations. Unlike source programs, they make explicit the location of computations, storage, and communication.

Figure 6 gives the syntax of choreographies, which we present as an extension of the source syntax. Host annotations on **let** and **if** statements are no longer restricted to $*$ and can be any host $h \in \mathbb{H}$. The *global communication* statement $h_1.t \rightsquigarrow h_2.x; s$ represents host h_1 sending the value of t to h_2, which stores it in variable x. The *selection* statement $h_1[v] \rightsquigarrow h_2; s$ communicates control flow decisions, and is used to establish *knowledge of choice* [28, 35]. We extend expressions with **send** and **receive**, however, the compiler never generates choreographies with these expressions; they are only used in proofs to model malicious corruption (§V-E).

As for source programs, configurations are single processes $\langle H, B, s \rangle$, but H may be a strict subset of $\mathbb{H}$ and does not contain the ideal process $*$.

$$\boxed{h.e \xrightarrow{a}_{\mathrm{i}} v} \qquad \frac{\ell_{\mathrm{f}} \notin \mathcal{P} \qquad \ell_{\mathrm{t}} \in \mathcal{P}}{h.\mathbf{declassify}(v, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) \xrightarrow{!h\mathrm{Adv}v}_{\mathrm{i}} v}$$

$$\frac{\ell_{\mathrm{f}} \notin \mathcal{T} \qquad \ell_{\mathrm{t}} \in \mathcal{T}}{h.\mathbf{endorse}(v, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) \xrightarrow{?\mathrm{Adv}hv'}_{\mathrm{i}} v'}$$

$$\frac{\mathbb{L}(h) \in \mathcal{T}}{h.\mathbf{input} \xrightarrow{?\mathrm{Env}hv}_{\mathrm{i}} v} \qquad \frac{\mathbb{L}(h) \in \mathcal{T}}{h.\mathbf{output}\ v \xrightarrow{!h\mathrm{Env}v}_{\mathrm{i}} 0}$$

$$\boxed{s \xrightarrow{a}_{\mathrm{r}} s'}$$

$$h_1.v \rightsquigarrow h_2.x; s \xrightarrow{!h_1h_2v}_{\mathrm{r}} s[v/x] \qquad h_1[v] \rightsquigarrow h_2; s \xrightarrow{!h_1h_2v}_{\mathrm{r}} s$$

$$\boxed{s \xrightarrow{a}{}^{\mathrm{c}}_{\alpha} s'} \qquad \frac{s \xrightarrow{a}{}^{\mathrm{c}}_{\alpha} s' \qquad \mathrm{actor}(a) \notin \mathrm{hosts}(E)}{\mathbf{let}\ h.x = e; s \xrightarrow{a}{}^{\mathrm{c}}_{\alpha} \mathbf{let}\ h.x = e; s'}$$

Figure 7. Select ideal, real, and concurrent stepping rules.

C. *Operational Semantics of Choreographies*

Following §III, we give operational semantics to programs using labeled transition systems. Since choreographies strictly extend the syntax of source programs, it suffices to define a semantics for choreographies.

Following fig. 1, we define two transition relations: *ideal* stepping $\to_{\mathrm{i}}$ gives meaning to source programs and to idealized choreographies (an intermediate language for our simulation proof), and *real* stepping $\to_{\mathrm{r}}$ gives meaning to choreographies and distributed programs. Additionally, we lift ideal and real stepping to concurrent versions, written $\to^{\mathrm{c}}_{\mathrm{i}}$ and $\to^{\mathrm{c}}_{\mathrm{r}}$, to capture the concurrent semantics of choreographies. Figure 7 gives a selection of key rules; we defer full definitions to §B.

1) Ideal Semantics: We write $h.e \xrightarrow{a}_{\mathrm{i}} v$ to mean expression e evaluates to value v at host h with action a. We assume operators are total; partial operators (like division) can be made total using defaults. Formally, we give meaning to operator application assuming a denotation function $\mathrm{eval} : \mathbb{F} \times \mathbb{V}^* \to \mathbb{V}$. We model **declassify** and **endorse** expressions as *interactions* with the adversary endpoint Adv. When a value is declassified from a secret label to a public one, the program *outputs* the value to Adv. Dually, when a value is endorsed from an untrusted label to a trusted one, the program takes *input* from Adv, and uses that value instead. When the confidentiality/integrity of the value does not change, these expressions act as the identity function and take internal steps. The **input/output** expressions communicate with the environment endpoint Env, except on malicious hosts; there, they step internally and always return 0. In source programs and idealized choreographies, **receive/send** expressions only communicate with malicious hosts; we suppress them (they take internal steps and always return 0) to give less power to the adversary in the ideal setting.

For statements, we write $s \xrightarrow{a}_{\mathrm{i}} s'$ to mean statement s steps to s' with action a. Statement stepping rules are as expected: **let** statements step using substitution, **if** statements pick a branch based on their conditional, and communication and selection statements step internally, naming the "sending host" as the host performing the action.

2) Real Semantics: Real stepping rules modify ideal stepping rules. The **declassify/endorse** expressions always step internally instead of communicating with Adv. The **receive/send** expressions communicate a value with the specified host. Finally, communication and selection statements step with a visible action instead of internally.

3) Concurrent Lifting for Choreographies: Concurrent stepping rules, written $s \xrightarrow{a}{}^{\mathrm{c}}_{\alpha} s'$, lift an underlying statement-stepping judgment ($\to_{\mathrm{i}}$ or $\to_{\mathrm{r}}$). Concurrent stepping allows choreographies to step statements at different hosts out of program order as long as there are no dependencies between the hosts, and is the standard way choreographies model the behavior of a distributed system [28]. The key rule allows skipping over **let** statements to step a statement in the middle of a program. This rule requires the actor of the performed action to be different from the hosts of the statements being skipped over, matching the behavior of target programs where code running on a single host is single-threaded.

a) Synchronous vs. Asynchronous Choreographies: When skipping over **let** statements, requiring only the *actor* to be missing from the context leads to an *asynchronous* semantics [53]. In a *synchronous* setting, the side condition would require both endpoints to be missing: $\mathrm{hosts}(a) \cap \mathrm{hosts}(E) = \emptyset$. Consider the following program:

$$\begin{array}{l}\mathbf{let}\ \mathrm{Alice}.x_1 = \mathbf{input};\\ \mathrm{Bob}.0 \rightsquigarrow \mathrm{Alice}.x_2;\end{array}$$

Alice is waiting on an input, so is not ready to receive from Bob. In a synchronous setting, these statements must execute in program order since Bob can only send if Alice is ready to receive. In an asynchronous setting, sends are nonblocking, so the second statement can execute first.

4) Processes: A buffer B behaves as a FIFO queue for each channel: it can input a message by appending the received value at the end of the corresponding queue, and can output the value at the beginning of any queue. Buffers guarantee in-order delivery within a single channel c_1c_2, but messages across different channels may be reordered. A process w forwards its input to its buffer if the message is addressed to a relevant host; otherwise, w discards the message. A process takes an internal step when its buffer delivers a message to its statement, and an output step when its statement outputs.

D. *Compiling to Choreographies*

Instead of committing to a specific algorithm, we give validity criteria for the output of protocol synthesis, which generalizes our results to and beyond prior work [21, 25, 54]. Because a source program can be realized as many different choreographies, protocol synthesis cannot be modeled as a function from source programs to choreographies. Instead,

$\boxed{\mathrm{src}(s) = s'}$

$$\mathrm{src}(\mathbf{let}\ h.x = e; s) = \begin{cases} \mathbf{let}\ h.x = e; \mathrm{src}(s) & \mathrm{I/O}(e) \\ \mathbf{let}\ *.x = e; \mathrm{src}(s) & \neg \mathrm{I/O}(e) \end{cases}$$
$$\mathrm{src}(h_1.t \rightsquigarrow h_2.x; s) = \mathrm{src}(s)[t/x]$$
$$\mathrm{src}(h_1[v] \rightsquigarrow h_2; s) = \mathrm{src}(s)$$
$$\mathrm{src}(\mathbf{if}(h.t, s_1, s_2)) = \mathbf{if}(*.t, \mathrm{src}(s_1), \mathrm{src}(s_2))$$
$$\mathrm{src}(\mathbf{skip}) = \mathbf{skip}$$
$$\mathrm{I/O}(e) = (e = \mathbf{input}) \vee (\exists t \,.\, e = \mathbf{output}\ t)$$

Figure 8. Canonical source program from a choreography.

we capture a valid protocol synthesis as a mapping from choreographies to source programs.

Definition V.1 (Valid Protocol Synthesis)**.** Choreography s' is a valid result of protocol synthesis on source program s if $\mathrm{src}(s') = s$, $\epsilon \vdash s'$, and $\Delta \Vdash s'$ for some Δ.

Figure 8 defines the function $\mathrm{src}(\cdot)$, which maps a choreography to its canonical source program by removing communication and selection statements and replacing all host annotations with $*$ (except those associated with **input** and **output**). The judgment $\Gamma \vdash s$ denotes that choreography s has secure information flows, and $\Delta \Vdash s$ denotes s is well-synchronized. We define these judgments next.

1) Information-Flow Type System: First, we give a type system for choreographies based on information-flow control [20, 21, 22, 25] which validates that hosts have enough authority to execute their assigned statements.

Figure 9 gives the typing rules. A label context Γ maps a variable to its host and label, as a set of bindings $x : h.\ell$. The judgment $\Gamma \vdash e : h.\ell$, means that e at host h has label ℓ in the context Γ. Rule ℓ-VARIABLE ensures hosts only use variables they own. Rules ℓ-DECLASSIFY and ℓ-ENDORSE enforce nonmalleable information flow control (NMIFC) [34] by requiring source and target labels to be uncompromised [50, 34]. NMIFC requires declassified data to be trusted, enforcing *robust declassification*, and endorsed data to be public, enforcing *transparent endorsement*. These restrictions prevent the adversary from exploiting downgrades. Enforcing NMIFC is crucial for our simulation result, which we discuss in §VI-C.

In choreographies, **receive**/**send** expressions model communication with malicious hosts. Choreographies exclude code for malicious hosts, which exhibit arbitrary behavior; thus, labels for **receive**/**send** expressions must be approximated. Rule ℓ-RECEIVE ensures data coming from malicious hosts is considered untrusted; it treats the data as fully public since we do not care about preserving the confidentiality of malicious hosts. Rule ℓ-SEND ensures secret data is not sent to malicious hosts; it ignores integrity since malicious hosts are untrusted.

Statement checking rules have the form $\Gamma \vdash s$; they are largely standard [27], but do not track program counter labels since we require programs to only branch on public, trusted values. Rules ℓ-LET and ℓ-COMMUNICATE check that the host storing a variable has enough authority to do so. This is the key condition governing secure host selection and prevents, for example, Bob's secret data being placed on Alice, or high-integrity data being placed on an untrusted host. Rule ℓ-SELECT ensures that if host h_1 informs h_2 of a branch being taken, then h_1 has at least as much integrity as h_2. So malicious hosts cannot influence control flow on nonmalicious hosts. Finally, rule ℓ-IF requires control flow to be public and trusted.

2) Synchronization Checking: Next, we define a novel *synchronization* judgment, $\Delta \Vdash s$, which guarantees that all external actions in s happen in sequential program order. For example, any **endorse** statement that happens after a **declassify** must logically *depend* on the **declassify**. Since these statements may be run on different hosts, the **declassify** could happen before the **endorse**, violating program order. To prevent this, we require that the host running the **endorse** *synchronizes* with the host running the **declassify**.

Synchronization becomes more complex with *corruption*. For example, if Alice and Bob synchronize through another host h (Alice $\rightsquigarrow h \rightsquigarrow$ Bob) and h is malicious, h might give Bob the go-ahead before confirming with Alice. We use integrity labels to ensure synchronization even under corruption.

Figure 10 defines the synchronization-checking judgment $\Delta \Vdash s$. Intuitively, a choreography is well-synchronized when for any *external* (input or output) expression e, a high-integrity communication path exists from e to all *output* expressions following e in the program order.[1] Integrity of a communication path $h_1 \rightsquigarrow \cdots \rightsquigarrow h_n$ is determined by the hosts in the path:

$$\mathbb{L}(h_1 \rightsquigarrow \cdots \rightsquigarrow h_n) = \mathbb{L}(h_1)^{\leftarrow} \vee \cdots \vee \mathbb{L}(h_n)^{\leftarrow}$$

Hosts can be malicious, so each host on the path weakens integrity, which is captured by disjunction ($\vee$). Multiple paths between the same hosts increase integrity, which we capture by taking the conjunction ($\wedge$) of path labels:

$$\mathbb{L}(\mathrm{paths}(h_1, h_2)) = \bigwedge_{path \in \mathrm{paths}(h_1, h_2)} \mathbb{L}(path)$$

We track the integrity of paths using the context Δ, which maps pairs of hosts $\Delta(h_1, h_2)$ to the integrity label $\mathbb{L}(\mathrm{paths}(h_1, h_2))$.

Rule SYNC-EXTERNAL checks a **let** statement that executes an external expression e on h. The continuation is checked under a context where the label of all paths *from* h to any other host are set to $\mathbf{1}$ (this corresponds to removing the paths), since these hosts now need to synchronize with h. Also, if e is an output expression, h must be synchronized with all hosts through the following condition, which ensures that if neither h_1 nor h_2 is malicious, a communication path exists from h_1 to h_2 that could not have been influenced by the adversary:

$$\mathbb{L}(\mathrm{paths}(h_1, h_2)) \sqsubseteq \mathbb{L}(h_1) \vee \mathbb{L}(h_2) \qquad (1)$$

Rules SYNC-COMMUNICATE and SYNC-SELECT update Δ using $\mathrm{sync}(\Delta, h_1 \rightsquigarrow h_2)$. The function captures that a path from some h to h_1 implies there is a path from h to h_2 that goes through h_1. Further, all existing paths are still valid.

[1] Input expressions are **input** and **endorse**; output expressions are **output** and **declassify**.

$\boxed{\Gamma \vdash t : h.\ell}$ $\boxed{\Gamma \vdash e : h.\ell}$

$$\ell\text{-VALUE}\ \frac{}{\Gamma \vdash v : h.\ell} \qquad \ell\text{-VARIABLE}\ \frac{\ell' \sqsubseteq \ell}{\Gamma, x : h.\ell' \vdash x : h.\ell} \qquad \ell\text{-OPERATOR}\ \frac{\forall i \,.\, \Gamma \vdash t_i : h.\ell}{\Gamma \vdash f(t_1, \dots, t_n) : h.\ell}$$

$$\ell\text{-DECLASSIFY}\ \frac{\Gamma \vdash t : h.\ell_{\mathrm{f}} \quad \ell_{\mathrm{f}}^{\leftarrow} = \ell_{\mathrm{t}}^{\leftarrow} \quad \blacktriangledown\ell_{\mathrm{f}} \quad \blacktriangledown\ell_{\mathrm{t}} \quad \ell_{\mathrm{t}} \sqsubseteq \ell}{\Gamma \vdash \mathbf{declassify}(t, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) : h.\ell}$$

$$\ell\text{-ENDORSE}\ \frac{\Gamma \vdash t : h.\ell_{\mathrm{f}} \quad \ell_{\mathrm{f}}^{\rightarrow} = \ell_{\mathrm{t}}^{\rightarrow} \quad \blacktriangledown\ell_{\mathrm{f}} \quad \blacktriangledown\ell_{\mathrm{t}} \quad \ell_{\mathrm{t}} \sqsubseteq \ell}{\Gamma \vdash \mathbf{endorse}(t, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) : h.\ell} \qquad \ell\text{-INPUT}\ \frac{\mathbb{L}(h) \sqsubseteq \ell}{\Gamma \vdash \mathbf{input} : h.\ell}$$

$$\ell\text{-OUTPUT}\ \frac{\Gamma \vdash t : h.\mathbb{L}(h)}{\Gamma \vdash \mathbf{output}\ t : h.\ell} \qquad \ell\text{-RECEIVE}\ \frac{\mathbb{L}(h')^{\leftarrow} \sqsubseteq \ell}{\Gamma \vdash \mathbf{receive}\ h' : h.\ell} \qquad \ell\text{-SEND}\ \frac{\Gamma \vdash t : h.\mathbb{L}(h')^{\rightarrow}}{\Gamma \vdash \mathbf{send}\ t\ \mathbf{to}\ h' : h.\ell}$$

$\boxed{\Gamma \vdash s}$

$$\ell\text{-LET}\ \frac{\Gamma \vdash e : h.\ell \quad \mathbb{L}(h) \Rightarrow \ell \quad \Gamma, x : h.\ell \vdash s}{\Gamma \vdash \mathbf{let}\ h.x = e; s} \qquad \ell\text{-COMMUNICATE}\ \frac{\Gamma \vdash t : h_1.\ell \quad \mathbb{L}(h_2) \Rightarrow \ell \quad \Gamma, x : h_2.\ell \vdash s}{\Gamma \vdash h_1.t \rightsquigarrow h_2.x; s}$$

$$\ell\text{-SELECT}\ \frac{\mathbb{L}(h_1)^{\leftarrow} \sqsubseteq \mathbb{L}(h_2)^{\leftarrow} \quad \Gamma \vdash s}{\Gamma \vdash h_1[v] \rightsquigarrow h_2; s} \qquad \ell\text{-IF}\ \frac{\Gamma \vdash t : h.\mathbf{0}^{\leftarrow} \quad \Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash \mathbf{if}(h.t, s_1, s_2)} \qquad \ell\text{-SKIP}\ \frac{}{\Gamma \vdash \mathbf{skip}}$$

Figure 9. Information-flow typing rules for expressions and statements in choreographies.

$\boxed{\Delta \Vdash s}$

$$\text{SYNC-EXTERNAL}\ \frac{\mathrm{external}(e) \quad \mathrm{reset}(\Delta, h) \Vdash s \quad \mathrm{outputting}(e) \implies \mathrm{synched}(\Delta, h)}{\Delta \Vdash \mathbf{let}\ h.x = e; s} \qquad \text{SYNC-INTERNAL}\ \frac{\mathrm{internal}(e) \quad \Delta \Vdash s}{\Delta \Vdash \mathbf{let}\ h.x = e; s}$$

$$\text{SYNC-COMMUNICATE}\ \frac{\mathrm{sync}(\Delta, h_1 \rightsquigarrow h_2) \Vdash s}{\Delta \Vdash h_1.t \rightsquigarrow h_2.x; s} \qquad \text{SYNC-SELECT}\ \frac{\mathrm{sync}(\Delta, h_1 \rightsquigarrow h_2) \Vdash s}{\Delta \Vdash h_1[v] \rightsquigarrow h_2; s} \qquad \text{SYNC-IF}\ \frac{\Delta \Vdash s_1 \quad \Delta \Vdash s_2}{\Delta \Vdash \mathbf{if}(h.t, s_1, s_2)} \qquad \text{SYNC-SKIP}\ \frac{}{\Delta \Vdash \mathbf{skip}}$$

$\boxed{\mathrm{synched}(\Delta, h)}$ $\boxed{\mathrm{reset}(\Delta, h) = \Delta'}$ $\boxed{\mathrm{sync}(\Delta, h_1 \rightsquigarrow h_2) = \Delta'}$

$$\mathrm{synched}(\Delta, h) = \forall h' \,.\, \Delta(h', h) \sqsubseteq \mathbb{L}(h') \vee \mathbb{L}(h) \qquad \mathrm{reset}(\Delta, h) = \Delta[h, * := \mathbf{1}][h, h := \mathbb{L}(h)]$$

$$\mathrm{sync}(\Delta, h_1 \rightsquigarrow h_2) = \Delta[*, h_2 := \Delta(*, h_2) \wedge (\Delta(*, h_1) \vee \mathbb{L}(h_2))]$$

Figure 10. Checking that a concurrent choreography has sequential behavior.

E. Modeling Malicious Corruption

Malicious hosts are fully controlled by the adversary. Following UC [26], we entirely remove processes that correspond to malicious hosts in hybrid distributed programs, and allow the adversary to forge arbitrary messages in their stead. This is reflected in choreographies by rewriting them to elide statements that involve malicious hosts.

Figure 11 defines the *corruption* $(\!|s|\!)$ of a choreography s. The operation considers each statement in turn. If *all* hosts involved in a statement are nonmalicious, the statement stays as is. If *all* hosts involved in a statement are malicious, the statement is removed entirely. Otherwise, only *some* involved hosts are malicious, and we rewrite the statement. Communication statements become either a **send** or a **receive**, depending on whether the sending or the receiving host is nonmalicious. Selection statements are similar, except we cannot have a malicious sending host and a nonmalicious receiving host (rule ℓ-SELECT). Similarly, **if** statements cannot be at a malicious host since we require trusted control flow (rule ℓ-IF).

$\boxed{(\!|s|\!) = s'}$

$$(\!|\mathbf{let}\ h.x = e; s|\!) = \begin{cases} \mathbf{let}\ h.x = e; (\!|s|\!) & h \in \mathcal{T} \\ (\!|s|\!) & \text{o/w} \end{cases}$$

$$(\!|h_1.t \rightsquigarrow h_2.x; s|\!) = \begin{cases} h_1.t \rightsquigarrow h_2.x; (\!|s|\!) & h_1, h_2 \in \mathcal{T} \\ \mathbf{let}\ h_1.\_ = \mathbf{send}\ t\ \mathbf{to}\ h_2; (\!|s|\!) & h_1 \in \mathcal{T} \\ \mathbf{let}\ h_2.x = \mathbf{receive}\ h_1; (\!|s|\!) & h_2 \in \mathcal{T} \\ (\!|s|\!) & \text{o/w} \end{cases}$$

$$(\!|h_1[v] \rightsquigarrow h_2; s|\!) = \begin{cases} h_1[v] \rightsquigarrow h_2; (\!|s|\!) & h_1, h_2 \in \mathcal{T} \\ \mathbf{let}\ h_1.\_ = \mathbf{send}\ v\ \mathbf{to}\ h_2; (\!|s|\!) & h_1 \in \mathcal{T} \\ (\!|s|\!) & \text{o/w} \end{cases}$$

$$(\!|\mathbf{if}(h.t, s_1, s_2)|\!) = \begin{cases} \mathbf{if}(h.t, (\!|s_1|\!), (\!|s_2|\!)) & h \in \mathcal{T} \\ \bot & \text{o/w} \end{cases}$$

$$(\!|\mathbf{skip}|\!) = \mathbf{skip}$$

$\boxed{(\!|w|\!) = w'}$ $\qquad (\!|\langle H, B, s\rangle|\!) = \langle \{h \in H \mid h \in \mathcal{T}\}, B, (\!|s|\!)\rangle$

Figure 11. Modeling malicious corruption. We write $h \in \mathcal{T}$ for $\mathbb{L}(h) \in \mathcal{T}$.

VI. Correctness of Protocol Synthesis

We prove the correctness of protocol synthesis by demonstrating a simulation between source programs and their corresponding choreographies. For $w = \langle H, B, s \rangle$, we write $\Gamma \vdash w$ and $\Delta \Vdash w$ if $\Gamma \vdash s$ and $\Delta \Vdash s$, respectively, and define $\mathrm{src}(w) = \langle H, B, \mathrm{src}(s) \rangle$.

Theorem VI.1. *If $\epsilon \vdash w$, and $\Delta \Vdash w$ for some Δ, then $\langle \mathrm{src}(w), \rightarrow_{\mathrm{i}} \rangle \geq \langle (\!|w|\!), \rightarrow_{\mathrm{r}} \rangle$.*

We prove theorem VI.1 through a series of intermediate simulations, following fig. 12 from left to right. First, in §VI-A, we show idealized, sequential choreographies simulate their canonical source programs. Then, we show in §VI-B that our synchronization judgment ensures all externally visible actions happen in program order. Finally, in §VI-C, we move from the ideal semantics $\rightarrow_{\mathrm{i}}^{\mathrm{c}}$ to the real semantics $\rightarrow_{\mathrm{r}}^{\mathrm{c}}$.

For each simulation, we define a simulator that emulates the adversary "in its head". We ensure that the emulated adversary's view is the same as the real adversary even though the simulator only has access to public information. Concretely, we establish a (weak) bisimulation relation [55, 56] between the ideal world (simulator running against ideal configuration) and the real world (adversary running against real configuration).

A. Correctness of Host Selection

Finally, we show that the sequential choreography simulates the original source program.

Theorem VI.2. *If $\epsilon \vdash w$, then $\langle \mathrm{src}(w), \rightarrow_{\mathrm{i}} \rangle \geq \langle (\!|w|\!), \rightarrow_{\mathrm{i}} \rangle$.*

This is shown via simpler simulations. First we add host annotations and explicit communication, then add corruption.

Lemma VI.3. *If $\epsilon \vdash w$, then $\langle \mathrm{src}(w), \rightarrow_{\mathrm{i}} \rangle \geq \langle w, \rightarrow_{\mathrm{i}} \rangle$.*

Proof. Statements removed by $\mathrm{src}(\cdot)$ only produce internal actions, which the simulator can recreate. Host annotations affect program behavior only by changing the source and destination of internal actions and actions generated by **declassify**/**endorse**; the simulator must recover the original host names before forwarding messages from/to the adversary.

The simulator maintains a public view of w and runs the adversary against this view. When the adversary steps w, the simulator steps $\mathrm{src}(w)$ only if the statement is preserved by $\mathrm{src}(\cdot)$; it does nothing otherwise. To handle **declassify**, whenever the simulator receives a message of the form $*\mathrm{Adv}v$, the simulator inspects its copy of w to determine the sending host h, and sends $h\mathrm{Adv}v$ to the adversary instead. Similarly, to handle **endorse**, the simulator replaces h with $*$ in messages $\mathrm{Adv}hv$ from the adversary. □

Lemma VI.4. *If $\epsilon \vdash w$, then $\langle w, \rightarrow_{\mathrm{i}} \rangle \geq \langle (\!|w|\!), \rightarrow_{\mathrm{i}} \rangle$.*

Proof. Corruption only removes statements at malicious hosts, however, these statements only generate internal actions: **input**/**output** expressions always step internally using ideal rules, and typing ensures **declassify**/**endorse** expressions step internally. This means $(\!|w|\!)$ and w have the same external behavior, except w takes extra internal steps. The simulator follows the control flow and acts like the adversary, but whenever the adversary schedules $(\!|w|\!)$, the simulator schedules w multiple times until the head statement is at a nonmalicious host; then, it schedules w again.

A small caveat: in $(\!|w|\!)$, all data from malicious hosts is explicitly replaced with 0, whereas malicious hosts may store arbitrary data in w. Since data from malicious hosts is untrusted, our type system ensures this data does not influence trusted data, which includes all output messages. Formally, we only require and maintain that $(\!|w|\!)$ and w agree on *trusted* values. □

B. Correctness of Sequentialization

Next, we show that a well-synchronized choreography stepping concurrently simulates itself stepping sequentially.

Theorem VI.5. *If $\epsilon \vdash w$, and $\Delta \Vdash w$ for some Δ, then $\langle (\!|w|\!), \rightarrow_{\mathrm{i}} \rangle \geq \langle (\!|w|\!), \rightarrow_{\mathrm{i}}^{\mathrm{c}} \rangle$.*

The adversary, interacting with the concurrent version of the choreography, can schedule a statement that is not next in program order. If the statement produces an externally visible action, the simulator must schedule the same statement. Since the simulator interacts with the sequential version, it must "unwind" the choreography by scheduling every statement leading up to the desired statement. Synchronization ensures unwinding does not fail due to a statement blocked on input (**input** or **endorse**), or a statement that performs a different visible action (**output** or **declassify**).

The concurrent and sequential choreographies necessarily fall out of sync during simulation: the adversary may schedule steps for the concurrent choreography that the simulator cannot immediately match, and the simulator might schedule steps for the sequential choreography while unwinding, steps the adversary did not schedule. Nevertheless, the two choreographies remain *joinable*: they can reach a common choreography via only internal actions. We prove choreographies are *confluent* [57, 58, 59], which ensures joinable processes remain joinable throughout the simulation.

Proof sketch for theorem VI.5. The simulator maintains a public view of the concurrent process, and runs the adversary against this view. When the adversary schedules an output, the simulator schedules the sequential process until it performs the same output; the simulator does nothing for input and internal actions. Well-synchronization guarantees the sequential program can perform the output. The primary invariant, that the concurrent and sequential processes remain joinable, is ensured by confluence. See §D for details. □

C. Correctness of Ideal Execution

A choreography stepping with the ideal rules is simulated by itself stepping with the real rules.

Theorem VI.6. *If $\epsilon \vdash w$, then $\langle (\!|w|\!), \rightarrow_{\mathrm{i}}^{\mathrm{c}} \rangle \geq \langle (\!|w|\!), \rightarrow_{\mathrm{r}}^{\mathrm{c}} \rangle$.*

The main difference between the two semantics is the interface with the adversary. In the real semantics, dishonest hosts actively leak data to the adversary (through **send** expressions and communication statements), and the adversary controls

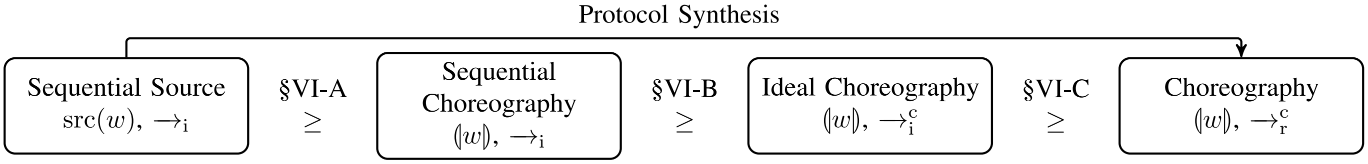


Figure 12. Intermediate simulation steps for proving the correctness of protocol synthesis.

all data coming from malicious hosts (through **receive**). In contrast, the ideal semantics interacts with the adversary only via **declassify** and **endorse**. In effect, the ideal semantics causes leakage and corruption to become coarse-grained. Additionally, by eliminating all blocking **receive** expressions (which communicate with the adversary), the ideal semantics can make progress in a manner independent of the adversary; this aids the sequentialization proof in §VI-B.

To bridge the gap between the real and ideal semantics, we show that the simulator can use **declassify** expressions to recreate all data no longer leaked through communication, and **endorse** expressions to corrupt all data no longer corruptible through **receive** expressions. This is possible because our type system ensures secrets are not directly sent to dishonest hosts (rules ℓ-SEND and ℓ-COMMUNICATE), and data from malicious hosts cannot directly influence trusted data (rule ℓ-RECEIVE).

Proof sketch for theorem VI.6. The simulator maintains a public view of the real process, and runs the adversary against this view. The simulator flips the input/output behavior of **declassify** and **endorse** expressions: when the ideal process *outputs* data through a **declassify** expression, the simulator inputs this data instead. Similarly, the simulator outputs data with **endorse** expressions, which it sends to the ideal process. The key invariant is that the simulator's version of the process matches the real one on *public values*, and the ideal process matches the real one on *trusted values*.

This invariant is strong enough to witness simulatability. Since the simulator's version of the process matches the real one on public values, the adversary in the real configuration has a view identical to the adversary running inside of the simulator (the adversary only sees public data). Similarly, since the real process matches the ideal one on trusted values, the environment has the same view in both (the environment is only sent trusted data).

Next, we argue that our simulator can preserve the invariant. For the simulator to have an accurate view of public values in the real process, the ideal process must output values through **declassify** expressions that match the values in the real process. The information-flow type system provides necessary restrictions. *Robust declassification* [34] guarantees only trusted values are declassified. Since the ideal process matches the real one on trusted values, data coming from **declassify** expressions is accurate. Dually, when the simulator sends data to the ideal process through **endorse** expressions, the values must match the ones in the real process. *Transparent endorsement* [34] guarantees only public values are endorsed.

Expressions	e	$::=$	$\ldots \mid \mathbf{receive}\ h \mid \mathbf{send}\ t\ \mathbf{to}\ h$
Statements	s	$::=$	$\ldots \mid \mathbf{case}\ (h_1 \rightsquigarrow h_2)\ \{v \mapsto s_v\}_{v \in V}$
Processes	w	$::=$	$\langle \{h \in \mathbb{H}\}, B, s \rangle$

Figure 13. Hybrid distributed syntax as an extension to source (fig. 5).

Thus, the result follows from the simulator holding accurate public values. See §E for details. □

VII. Endpoint Projection

The second stage of compilation, endpoint projection, transforms a choreography into a distributed program.

A. Hybrid Distributed Language

As for choreographies, fig. 13 specifies the syntax of hybrid distributed programs by extending the syntax of source programs. Hybrid distributed programs are configurations containing multiple processes, with each process acting on behalf of a single host $h \in \mathbb{H}$. These processes communicate data via **send/receive** and agree on control flow via **send** and **case**. The **case** statement, on receiving a value on the expected channel, steps to the specified branch.

Operational Semantics: Each process transitions using the real stepping rules ($\rightarrow_{\mathrm{r}}$), and the distributed configuration steps using the parallel composition rules in fig. 4.

B. Compiling to Distributed Programs

Given a choreography s and a host h, the endpoint projection $[\![s]\!]_h$ defines the local program that h should run. The distributed system $[\![s]\!]$ is derived by independently projecting onto each host in the choreography.

Our notion of endpoint projection is entirely standard, so we defer the formal definition to §F. Our proof is agnostic to how endpoint projection is defined, and only relies on its soundness and completeness, properties extensively studied in prior work [28, 35, 36, 43, 44].

C. Correctness of Endpoint Projection

Let $(\!|W|\!)$ remove from W all processes for malicious hosts.

Theorem VII.1. *If* $\epsilon \vdash w$, *then* $\langle (\!|w|\!), \rightarrow^{\mathrm{c}}_{\mathrm{r}} \rangle \geq \langle (\!|[\![w]\!]|\!), \rightarrow_{\mathrm{r}} \rangle$.

A choreography and its endpoint projection match each other action-for-action; once we prove this fact, showing simulation is trivial since we can pick $\mathcal{S} = \mathcal{A}$. This perfect correspondence between a choreography and its projection is studied extensively in the literature [28, 35, 36, 43, 44], and formalized as *soundness* and *completeness* of endpoint projection. However, the standard methods of proving soundness and completeness must be modified to handle malicious corruption and asynchronous communication. Existing work relates w to $[\![w]\!]$, which we must extend to relate $(\!|w|\!)$ to $(\!|[\![w]\!]|\!)$. This is trivial since $(\!|[\![w]\!]|\!) = [\![(\!|w|\!)]\!]$.

The presence of *asynchrony* breaks the perfect correspondence between the projected program and the choreography: a **send/receive** pair reduces in two steps in a projected program, but the corresponding communication statement reduces in only one. We follow prior work [53, 60] and add syntactic forms to choreographies for partially reduced **send/receive** pairs: messages that have been sent and buffered but not yet received. These run-time terms exist only to restore the correspondence, and are never generated by the compiler. An additional, simple simulation then shows that a choreography with these run-time terms simulates one without, removing the need to reason about run-time terms in other proof steps. For details, see §G.

VIII. Cryptographic Instantiation

Our simulation result is a necessary and novel first step toward constructing a verified, secure compiler for distributed protocols that use cryptography. We have abstracted all cryptographic mechanisms into idealized hosts (e.g., $\mathrm{MPC}(\mathrm{Alice}, \mathrm{Bob})$); thus, to achieve a full end-to-end security proof, these idealized hosts must be securely instantiated with cryptographic subprotocols (e.g., BGW [61] for multiparty computation). Such an instantiation would imply UC security for all compiled programs, in contrast to existing formalization efforts for individual protocols [62, 63, 64].

To this end, we show how the distributed protocols arising from our compilation correspond to hybrid protocols in the Simplified Universal Composability (SUC) framework [30]. Then, we show how to take advantage of the *composition* theorem in SUC to obtain secure, concrete instantiations of cryptographic protocols.

a) Simplified UC: Let s be a choreography with partitioning $[\![s]\!]$. We construct a corresponding SUC protocol $[\![s]\!]^{\mathsf{SUC}}$ which behaves identically to the partitioned choreography, with minor differences due to the differing computational models. Each host in s is either a *local* host (e.g., Alice), or an *idealized* host standing in for cryptography, such as $\mathrm{MPC}(\mathrm{Alice}, \mathrm{Bob})$. Local hosts map onto SUC parties, while idealized hosts map onto *ideal functionalities* in SUC.

Protocol execution in SUC happens through *activations* scheduled by the adversary: a party runs for some steps, delivers messages to a central *router*, and cedes execution to the adversary. To faithfully capture the behavior of host h in $[\![s]\!]$, the party/functionality for h in $[\![s]\!]^{\mathsf{SUC}}$ is essentially a *wrapper* around the projected host $[\![s]\!]_h$, who steps $[\![s]\!]_h$ accordingly and forwards correct messages to the router.

Each wrapper needs to explicitly model *corruption*, which framework captures by labels: if host h is semi-honest ($\mathbb{L}(h) \in \mathcal{P} \cap \mathcal{T}$), the wrapper for h allows the adversary to query h for its current message transcript so far. Similarly, if h is malicious ($\mathbb{L}(h) \notin \mathcal{T}$), the wrapper for h should enable the adversary to take complete control over h. By using labels to model corruption, we model *static* security in SUC.

b) Communication Model: In SUC, all messages between local hosts are fully public, while messages between hosts and functionalities contain *public headers* (e.g., the source/destination addresses) and *private content* (the message payload). In our system, we do not stratify message privacy along the party/functionalities axis, but rather along the information flow lattice: the adversary can read the messages intended for semi-honest hosts, and can forge messages from malicious hosts. Indeed, information flow policies allow more flexible security policies for communication.

However, we can encode our communication model into SUC with the aid of additional functionalities. To do so, we make use of a secure channel functionality $\mathcal{R}_{\mathsf{sec}}$, which guarantees in-order message delivery and enables secret communication between honest hosts. We can realize $\mathcal{R}_{\mathsf{sec}}$ in SUC via a standard subprotocol using a public key infrastructure.

For ideal functionalities in $[\![s]\!]^{\mathsf{SUC}}$, we need to ensure that they only communicate with local hosts, and not with other ideal functionalities. This property is preserved by compilation, so we only need to ensure that host selection produces a choreography s that has this property. Indeed, our synchronization judgment $\Delta \Vdash s$ makes it possible for choreographies to stay well-synchronized, even when the ideal hosts do not communicate with each other.

c) Adversaries and Environments: In our framework, we prove perfect security against non-probabilistic adversaries. However, allowing the adversary to use probability (as in SUC) does not weaken our simulation result.[2] Additionally, in UC/SUC, the environment is given by a concurrently running process that outputs a *decision bit*, whereas our model uses a trace semantics to model the environment. Security for the latter easily implies the former, since our simulation result proves equality of environment views between the two worlds.

A. Secure Instantiation of Cryptography

To securely instantiate cryptographic mechanisms, we appeal to the *composition* theorem in SUC, which says that ideal SUC-functionalities $\mathcal{F}$ may be substituted for SUC protocols that securely realize them. Concrete cryptographic protocols are obtained by applying this theorem iteratively to each ideal host.

Ideal hosts in our model correspond closely to the broad class of *reactive, deterministic straight-line* functionalities in SUC, including MPC [30, 65] and Zero-Knowledge Proofs (ZKP) [66]. The main difference is that our model allows the adversary to corrupt ideal functionalities (both semi-honestly and maliciously), while SUC functionalities are incorruptible. However, we guarantee that the adversary does not gain more power in our model by restricting the possible corruption models via authority labels for ideal hosts.

For example, we have $\mathrm{MPC}(\mathrm{Alice}, \mathrm{Bob})$ has label $\mathrm{A} \wedge \mathrm{B}$, meaning that $\mathrm{MPC}(\mathrm{Alice}, \mathrm{Bob})$ is semi-honest (resp. malicious) only if *both* Alice and Bob are semi-honest (resp.

[2]The dummy adversary theorem [26] implies that security against non-probabilistic adversaries guarantees security against probabilistic adversaries.

malicious). Thus, any power the adversary gains in corrupting $\mathrm{MPC}(\mathrm{Alice}, \mathrm{Bob})$ can be instead achieved using Alice and Bob alone. Similar security concerns for label-based host selection have been discussed for Viaduct [25]. We can formalize this intuition via a simulation of the form $\langle W, \rightarrow_{\mathrm{i}}\rangle \leq \langle W, \rightarrow'_{\mathrm{i}}\rangle$, where W uses $\mathrm{MPC}(\mathrm{Alice}, \mathrm{Bob})$, and $\rightarrow'_{\mathrm{i}}$ is modified so that corruption of $\mathrm{MPC}(\mathrm{Alice}, \mathrm{Bob})$ is impossible.

IX. Security Preservation

We use simulation to define the correctness of compilation, and show that it corresponds to a well-studied correctness criterion, *robust hyperproperty preservation* (RHP) [31]. RHP states that hyperproperties [37] satisfied by source programs under any context are also satisfied by target programs under any context. RHP is important because common notions of information-flow security such as termination-insensitive noninterference, observational determinism [67], and nonmalleable information flow control [34] are hyperproperties. With a compiler that satisfies RHP, one only needs to prove security of source programs; security of target programs immediately follows.

Definition IX.1 (Robust Hyperproperty Preservation (RHP))**.** Let $\downarrow$ be a compiler from a source program to a target program, $\bowtie$ be an operator that composes a program with its context, and $\mathbb{B}$ be a behavior function that returns the set of possible traces generated from a whole program (i.e., a program composed with a context). Then $\downarrow$ satisfies RHP over source program set $\mathbb{P}$, source context set $\mathbb{C}_{\mathrm{S}}$, and target context set $\mathbb{C}_{\mathrm{T}}$ when, given program $\mathrm{P} \in \mathbb{P}$, for all $\mathrm{C_T} \in \mathbb{C}_{\mathrm{T}}$ there exists $\mathrm{C_S} \in \mathbb{C}_{\mathrm{S}}$ such that $\mathbb{B}(\mathrm{C_S} \bowtie \mathrm{P}) = \mathbb{B}(\mathrm{C_T} \bowtie \mathrm{P}\!\downarrow)$.[3]

Patrignani et al. [38, 39] previously observed a correspondence between UC simulation and robust hyperproperty preservation; it also holds for our notion of simulation.

Theorem IX.2 (Simulation Implies RHP)**.** *Define* $\mathcal{A} \bowtie W = \mathcal{A} \parallel W$. *Then given behavior function* $\mathbb{B}(\cdot) = \mathbb{T}_{\mathrm{Env}}(\cdot)$ *and an operator* $\downarrow$ *between configurations such that* $W\!\downarrow \leq W$ *for any configuration* $W \in \mathbb{W}$*, we have that* $\downarrow$ *satisfies RHP over source program set* $\mathbb{W}$ *and source and target context set* $\{\mathcal{A}\}$*.*

Corollary IX.3 (Partitioning Satisfies RHP)**.** *The function* $\lambda w.\langle (\!|[\![w]\!]|\!), \rightarrow_{\mathrm{r}}\rangle$ *satisfies RHP over source and target context set* $\{\mathcal{A}\}$ *and source program set*

$$\{\langle \mathrm{src}(w), \rightarrow_{\mathrm{i}}\rangle \mid \epsilon \vdash w, \Delta \Vdash w\}$$

Proof. Theorem IX.2 follows from definitions III.2 and IX.1. Corollary IX.3 follows from theorems VI.1 and IX.2. □

X. Related Work

Secure Program Partitioning: Prior work on secure program partitioning [68, 20, 21, 25] focuses largely on the engineering effort of compiling security-typed source programs to distributed code with the aid of cryptography. Our compilation model is closest to that of Viaduct [25], because we also approximate security guarantees of cryptographic mechanism with information-flow labels, but our goal differs: to give formal guarantees to such compilers.

A long line of work [23, 24, 69, 70, 71, 72] focuses on enforcing computational noninterference for information-flow typed programs by using standard cryptographic mechanisms, such as encryption. However, computational noninterference guarantees little in the presence of downgrading. In contrast, our compiler enjoys *simulation-based security*, which guarantees preservation of *all* hyperproperties, even for programs using declassification and endorsement.

Liu et al. [73] give an informal UC simulation proof of a compiler limited to two party semi-honest MPC and oblivious RAM. They do not consider integrity.

Simulation-based Security: Simulation-based cryptographic frameworks, such as Universal Composability [26], Reactive Simulatability [42], and Constructive Cryptography [74], allow modular proofs of distributed cryptographic protocols, and Liao et al. [75] give a core language for formalizing UC protocols. We abstract away concrete cryptography, so we do not explicitly model some subtleties of these systems: probability, computational complexity, and cryptographic hardness assumptions. But our approach should be compatible with these frameworks.

Prior verification efforts [62, 76, 63] show simulation-based security for concrete cryptographic mechanisms. Our work is orthogonal: simulation-based security for compiler correctness, rather than proofs for individual protocols.

Secure Compilation: Standard notions of compiler correctness are derived from full abstraction and hyperproperty preservation [31]. Patrignani et al. [38, 39] argue that robust hyperproperty preservation and Universal Composability are directly analogous. We affirm this hypothesis by proving that our simulation-based security result guarantees RHP. To our knowledge, we are the first to make this connection formally.

Choreographies: The use of choreographies is central to our compilation process and to the proof of its correctness. The primary concern in the extensive literature on choreographies [28, 35, 36, 43, 44], is proving deadlock freedom; very little prior work considers security [77]. Our extension of choreographies with an information-flow type system, modeling semi-honest and malicious corruption, is novel.

XI. Conclusion and Future Work

This work presents a novel simulation-based security result for a compiler from sequential source programs to a distributed programs, using idealizations of cryptographic mechanisms. Our simulation result guarantees that the security properties of source programs are preserved in the compiled protocols.

This work opens up many opportunities for future research, such as fully integrating our result into the Universal Composability framework and instantiating our idealizations with concrete cryptographic protocols.

Our security result holds for a strong attacker model that precludes secret control flow. To allow it, weaker attacker models should be explored.

[3]This is the "property-free" definition of RHP as given by Abate et al. [31]. An equivalent but more direct definition is that $\downarrow$ satisfies RHP given that if for some hyperproperty HP it is the case that $\mathbb{B}(\mathrm{C_S} \bowtie \mathrm{P}) \in \mathrm{HP}$ for any $\mathrm{C_S} \in \mathbb{C}_{\mathrm{S}}$, then $\mathbb{B}(\mathrm{C_T} \bowtie \mathrm{P}\!\downarrow) \in \mathrm{HP}$ for any $\mathrm{C_T} \in \mathbb{C}_{\mathrm{T}}$.

We have focused on confidentiality and integrity properties; however, some protocols aim to provide availability as well [78], another interesting direction for exploration.

XII. Acknowledgments

This work was supported by the National Science Foundation under NSF grant 1704788 and by the IARPA HECTOR program under a subcontract from IBM. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any of these funders.

References

[1] L. Lamport, "The part-time parliament," *ACM Trans. on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998. [Online]. Available: http://doi.acm.org/10.1145/279227.279229

[2] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Trans. on Computer Systems*, vol. 20, p. 2002, 2002.

[3] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Workshop on Hardware and Architectural Support for Security and Privacy*, R. B. Lee and W. Shi, Eds. ACM, 2013, p. 10.

[4] A. Gollamudi, S. Chong, and O. Arden, "Information flow control for distributed trusted execution environments," in *32nd IEEE Computer Security Foundations Symp. (CSF)*. IEEE, 2019, pp. 304–318. [Online]. Available: https://doi.org/10.1109/CSF.2019.00028

[5] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *USENIX Security Symposium*, 2016.

[6] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in Android applications," in *ACM Conf. on Computer and Communications Security (CCS)*, 2013, pp. 73–84. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516693

[7] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *19th ACM Conf. on Computer and Communications Security (CCS)*, T. Yu, G. Danezis, and V. D. Gligor, Eds. ACM, 2012, pp. 38–49. [Online]. Available: https://doi.org/10.1145/2382196.2382204

[8] C. Decker and R. Wattenhofer, "Bitcoin transaction malleability and MtGox," in *19th European Symposium on Research in Computer Security*. Springer, 2014, pp. 313–326.

[9] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay - a secure two-party computation system," in *13th Usenix Security Symposium*, Aug. 2004, pp. 287–302. [Online]. Available: http://www.usenix.org/publications/library/proceedings/sec04/tech/malkhi.html

[10] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *25th ACM Symp. on Operating System Principles (SOSP)*. IEEE, 2015, pp. 359–376. [Online]. Available: https://doi.org/10.1109/SP.2015.29

[11] A. Aly, D. Cozzo, M. Keller, E. Orsini, D. Rotaru, P. Scholl, N. P. Smart, and T. Wood, *SCALE–MAMBA v1.6 : Documentation*, 2019. [Online]. Available: https://homes.esat.kuleuven.be/~nsmart/SCALE

[12] A. Rastogi, M. A. Hammer, and M. Hicks, "Wysteria: A programming language for generic, mixed-mode multiparty computations," in *IEEE Symp. on Security and Privacy*, May 2014, pp. 655–670.

[13] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *IEEE Symp. on Security and Privacy*. IEEE, 2013, pp. 238–252.

[14] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur, "Geppetto: Versatile verifiable computation," in *IEEE Symp. on Security and Privacy*. IEEE, 2015, pp. 253–270.

[15] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish, "Efficient RAM and control flow in verifiable outsourced computation," in *Network and Distributed System Security Symp.* The Internet Society, 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/efficient-ram-and-control-flow-verifiable-outsourced-computation

[16] A. Kosba, C. Papamanthou, and E. Shi, "xJsnark: A framework for efficient verifiable computation," in *IEEE Symp. on Security and Privacy*. IEEE, 2018, pp. 944–961.

[17] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. E. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: an optimizing compiler for fully-homomorphic neural-network inferencing," in *40th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 142–156.

[18] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *41st ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 546–561.

[19] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, "Porcupine: A synthesizing compiler for vectorized homomorphic encryption," in *42nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2021, pp. 375–389.

[20] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Secure program partitioning," *ACM Trans. on Computer Systems*, vol. 20, no. 3, pp. 283–328, Aug. 2002. [Online]. Available: http://www.cs.cornell.edu/andru/papers/sosp01/spp-tr.pdf

[21] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic, "Using replication and partitioning to build secure distributed systems," in *IEEE Symp. on Security and Privacy*, May 2003, pp. 236–250. [Online]. Available: http://www.cs.cornell.edu/andru/papers/sp03.pdf

[22] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure web applications via automatic partitioning," in *21st ACM Symp. on Operating System Principles (SOSP)*, Oct. 2007, pp. 31–44. [Online]. Available: http://www.cs.cornell.edu/andru/papers/swift-sosp07.pdf

[23] C. Fournet and T. Rezk, "Cryptographically sound implementations for typed information-flow security," in *35th ACM Symp. on Principles of Programming Languages (POPL)*, Jan. 2008, pp. 323–335. [Online]. Available: https://doi.org/10.1145/1328438.1328478

[24] C. Fournet, G. le Guernic, and T. Rezk, "A security-preserving compiler for distributed programs: From information-flow policies to cryptographic mechanisms," in *16th ACM Conf. on Computer and Communications Security (CCS)*, Nov. 2009, pp. 432–441. [Online]. Available: https://doi.org/10.1145/1653662.1653715

[25] C. Acay, R. Recto, J. Gancher, A. Myers, and E. Shi, "Viaduct: An extensible, optimizing compiler for secure distributed programs," in *42nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, Jun. 2021, pp. 740–755.

[26] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *42nd Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 2001, pp. 136–145.

[27] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003. [Online]. Available: http://www.cs.cornell.edu/andru/papers/jsac/sm-jsac03.pdf

[28] F. Montesi, *Introduction to Choreographies*. Cambridge: Cambridge University Press, 2023.

[29] A. Bakst, K. von Gleissenthall, R. G. Kıcı, and R. Jhala, "Verifying distributed programs via canonical sequentialization," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017.

[30] R. Canetti, A. Cohen, and Y. Lindell, "A simpler variant of universally composable security for standard multiparty computation," Cryptology ePrint Archive, Report 2014/553, 2014. [Online]. Available: https://eprint.iacr.org/2014/553

[31] C. Abate, R. Blanco, D. Garg, C. Hritcu, M. Patrignani, and J. Thibault, "Journey beyond full abstraction: Exploring robust property preservation for secure compilation," in *32nd IEEE Computer Security Foundations Symp. (CSF)*. IEEE Computer Society, 2019, pp. 256–271. [Online]. Available: https://doi.org/10.1109/CSF.2019.00025

[32] A. C. Yao, "Protocols for secure computations," in *23rd annual IEEE Symposium on Foundations of Computer Science*, 1982, pp. 160–164. [Online]. Available: https://doi.org/10.1109/SFCS.1982.38

[33] A. C. Myers, A. Sabelfeld, and S. Zdancewic, "Enforcing robust declassification," in *17th IEEE Computer Security Foundations Workshop (CSFW)*, Jun. 2004, pp. 172–186. [Online]. Available: http://www.cs.cornell.edu/andru/papers/csfw04.pdf

[34] E. Cecchetti, A. C. Myers, and O. Arden, "Nonmalleable information flow control," in *24th ACM Conf. on Computer and Communications Security (CCS)*. ACM, Oct. 2017, pp. 1875–1891. [Online]. Available: http://www.cs.cornell.edu/andru/papers/nmifc

[35] F. Montesi, "Choreographic programming," Ph.D. dissertation, IT University of Copenhagen, 2013. [Online]. Available: https://www.fabriziomontesi.com/files/choreographic-programming.pdf

[36] L. Cruz-Filipe and F. Montesi, "A core model for choreographic programming," *Theoretical Computer Science*, vol. 802, pp. 38–66, 2020.

[37] M. R. Clarkson and F. B. Schneider, "Hyperproperties," in *21st IEEE Computer Security Foundations Symp. (CSF)*, Jun. 2008, pp. 51–65.

[38] M. Patrignani, R. S. Wahby, and R. Künneman, "Universal composability is secure compilation," arXiv ePrint 1910.08634, 2019.

[39] M. Patrignani, R. Künnemann, and R. S. Wahby, "Universal composability is robust compilation," arXiv ePrint 1910.08634, 2022.

[40] Y. Lindell, "How to simulate it — A tutorial on the simulation proof technique," in *Tutorials on the Foundations of Cryptography*. Springer International Publishing, 2017, pp. 277–346.

[41] T. Dierks and E. Rescorla, "The transport layer security (TLS) protocol version 1.1," Internet RFC-4346, Apr. 2006.

[42] M. Backes, B. Pfitzmann, and M. Waidner, "The reactive simulatability (RSIM) framework for asynchronous systems," *Information and Computation*, vol. 205, no. 12, pp. 1685–1720, 2007.

[43] L. Cruz-Filipe, F. Montesi, and M. Peressotti, "A formal theory of choreographic programming," *CoRR*, vol. abs/2209.01886, 2022.

[44] A. K. Hirsch and D. Garg, "Pirouette: Higher-order typed functional choreographies," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–27, Jan. 2022.

[45] N. A. Lynch and M. R. Tuttle, "Hierarchical correctness proofs for distributed algorithms," in *6th ACM Symp. on Principles of Distributed Computing*, F. B. Schneider, Ed. ACM, 1987, pp. 137–151.

[46] W. Rafnsson and A. Sabelfeld, "Compositional information-flow security for interactive systems," in *27th IEEE Computer Security Foundations Symp. (CSF)*. IEEE Computer Society, 2014, pp. 277–292. [Online]. Available: https://doi.org/10.1109/CSF.2014.27

[47] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 4, pp. 410–442, Oct. 2000. [Online]. Available: http://www.cs.cornell.edu/andru/papers/iflow-tosem.pdf

[48] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell, "Disjunction category labels," in *Proceedings of the 16th Nordic conference on Information Security Technology for Applications*, ser. Lecture Notes in Computer Science, P. Laud, Ed., vol. 7161. Springer, 2011, pp. 223–239.

[49] O. Arden, J. Liu, and A. C. Myers, "Flow-limited authorization," in *28th IEEE Computer Security Foundations Symp. (CSF)*, Jul. 2015, pp. 569–583. [Online]. Available: http://www.cs.cornell.edu/andru/papers/flam

[50] D. Zagieboylo, G. E. Suh, and A. C. Myers, "Using information flow to design an ISA that controls timing channels," in *32nd IEEE Computer Security Foundations Symp. (CSF)*, Jun. 2019. [Online]. Available: https://www.cs.cornell.edu/andru/papers/hyperisa

[51] K. R. O'Neill, M. R. Clarkson, and S. Chong, "Information-flow security for interactive programs," in *19th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society, 2006, pp. 190–201. [Online]. Available: https://doi.org/10.1109/CSFW.2006.16

[52] D. Clark and S. Hunt, "Non-interference for deterministic interactive programs," in *5th Workshop on Formal Aspects in Security and Trust (FAST)*, ser. Lecture Notes in Computer Science, vol. 5491. Springer, 2008, pp. 50–66. [Online]. Available: https://doi.org/10.1007/978-3-642-01465-9_4

[53] L. Cruz-Filipe and F. Montesi, "On asynchrony and choreographies," in *ICE@DisCoTec*, ser. EPTCS, M. Bartoletti, L. Bocchi, L. Henrio, and S. Knight, Eds., vol. 261, 2017, pp. 76–90.

[54] M. Ishaq, A. Milanova, and V. Zikas, "Efficient MPC via program analysis: A framework for efficient optimal mixing," in *26th ACM Conf. on Computer and Communications Security (CCS)*. ACM, 2019, pp. 1539–1556.

[55] S. Nain and M. Y. Vardi, "Branching vs. linear time: Semantical perspective," in *ATVA*, ser. Lecture Notes in Computer Science, K. S. Namjoshi, T. Yoneda, T. Higashino, and Y. Okamura, Eds., vol. 4762. Springer, 2007, pp. 19–34.

[56] M. Hennessy and R. Milner, "Algebraic laws for nondeterminism and concurrency," *J. ACM*, vol. 32, no. 1, pp. 137–161, 1985.

[57] M. H. A. Newman, "On theories with a combinatorial definition of "equivalence"," *Annals of Mathematics*, vol. 43, no. 2, pp. 223–243, 1942.

[58] A. Church and J. B. Rosser, "Some properties of conversion," *Transactions of the American Mathematical Society*, vol. 39, no. 3, pp. 472–482, 1936.

[59] F. Baader and T. Nipkow, *Term rewriting and all that*. Cambridge University Press, 1998.

[60] J. Å. Pohjola, A. Gómez-Londoño, J. Shaker, and M. Norrish, "Kalas: A verified, end-to-end compiler for a choreographic language," in *ITP*, ser. LIPIcs, J. Andronick and L. de Moura, Eds., vol. 237. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 27:1–27:18.

[61] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation," in *20th Annual ACM Symposium on Theory of Computing*, J. Simon, Ed., 1988, pp. 1–10. [Online]. Available: https://doi.org/10.1145/62212.62213

[62] R. Canetti, A. Stoughton, and M. Varia, "EasyUC: Using EasyCrypt to mechanize proofs of universally composable

security," in *32nd IEEE Computer Security Foundations Symp. (CSF)*. IEEE, 2019, pp. 167–183.

[63] M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, and P. Strub, "Mechanized proofs of adversarial complexity and application to universal composability," in *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds. ACM, 2021, pp. 2541–2563. [Online]. Available: https://doi.org/10.1145/3460120.3484548

[64] J. Gancher, K. Sojakova, X. Fan, E. Shi, and G. Morrisett, "A core calculus for equational proofs of cryptographic protocols," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, pp. 866–892, 2023.

[65] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, "SoK: general purpose compilers for secure multi-party computation," in *IEEE Symp. on Security and Privacy*, 2019, pp. 1220–1237. [Online]. Available: https://doi.org/10.1109/SP.2019.00028

[66] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai, "Zero-knowledge from secure multiparty computation," in *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '07. New York, NY, USA: ACM, 2007, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/1250790.1250794

[67] S. Zdancewic and A. C. Myers, "Observational determinism for concurrent program security," in *16th IEEE Computer Security Foundations Workshop (CSFW)*, Jun. 2003, pp. 29–43. [Online]. Available: http://www.cs.cornell.edu/andru/papers/csfw03.pdf

[68] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Enhancing Java programs with distribution capabilities," *ACM Trans. Softw. Eng. Methodol.*, vol. 19, no. 1, pp. 1:1–1:40, Aug. 2009. [Online]. Available: http://doi.acm.org/10.1145/1555392.1555394

[69] A. Askarov, D. Hedin, and A. Sabelfeld, "Cryptographically-masked flows," *Theor. Comput. Sci.*, vol. 402, no. 2-3, pp. 82–101, 2008.

[70] P. Laud, "On the computational soundness of cryptographically masked flows," in *35th ACM Symp. on Principles of Programming Languages (POPL)*, G. C. Necula and P. Wadler, Eds. ACM, 2008, pp. 337–348.

[71] R. Küsters, T. Truderung, and J. Graf, "A framework for the cryptographic verification of Java-like programs," in *25th IEEE Computer Security Foundations Symp. (CSF)*, S. Chong, Ed. IEEE Computer Society, 2012, pp. 198–212.

[72] R. Küsters, T. Truderung, B. Beckert, D. Bruns, M. Kirsten, and M. Mohr, "A hybrid approach for proving noninterference of Java programs," in *28th IEEE Computer Security Foundations Symp. (CSF)*, C. Fournet, M. W. Hicks, and L. Viganò, Eds. IEEE Computer Society, 2015, pp. 305–319.

[73] C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks, "Automating efficient RAM-model secure computation," in *IEEE Symp. on Security and Privacy*. IEEE, 2014, pp. 623–638.

[74] U. Maurer, "Constructive cryptography – A new paradigm for security definitions and proofs," in *Theory of Security and Applications*, ser. Lecture Notes in Computer Science, S. Mödersheim and C. Palamidessi, Eds., vol. 6993. Springer, 2011, pp. 33–56.

[75] K. Liao, M. A. Hammer, and A. Miller, "ILC: a calculus for composable, computational cryptography," in *40th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun. 2019, pp. 640–654.

[76] A. Lochbihler, S. R. Sefidgar, D. Basin, and U. Maurer, "Formalizing constructive cryptography using CryptHOL," in *32nd IEEE Computer Security Foundations Symp. (CSF)*. IEEE, 2019, pp. 152–166.

[77] A. Lluch-Lafuente, F. Nielson, and H. R. Nielson, "Discretionary information flow control for interaction-oriented specifications," in *Logic, Rewriting, and Concurrency*, ser. Lecture Notes in Computer Science, N. Martí-Oliet, P. C. Ölveczky, and C. L. Talcott, Eds., vol. 9200. Springer, 2015, pp. 427–450.

[78] L. Zheng and A. C. Myers, "A language-based approach to secure quorum replication," in *9th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, Aug. 2014. [Online]. Available: http://www.cs.cornell.edu/andru/papers/plas14

A Details for Section IV (Specifying Security Policies)

Formally, an attack is specified by picking two sets of *principals*: public principals $P \subseteq \mathbb{P}$ and untrusted principals $Q \subseteq \mathbb{P}$. Some common-sense conditions must hold on the sets P and Q [34]. We state the conditions for P but they apply equally to Q. The adversary always controls the weakest principal, but never controls the strongest: $\mathbf{1} \in P$ and $\mathbf{0} \notin P$. If the adversary controls a principal, then it controls all weaker principals: if $p \in P$ and $p \Rightarrow q$, then $q \in P$. Attacking principals may collude: if $p, q \in P$, then $p \wedge q \in P$. Combining secret/trusted principals leads to secret/trusted principals: if $p \vee q \in P$, then either $p \in P$ or $q \in P$. Together, these conditions imply that P and Q are sensible truth assignments to elements $\mathbb{P}$: sensible in the sense that they play nicely with $\wedge$ and $\vee$.[4]

We can derive the set of public/secret and trusted/untrusted *labels* from P and Q:

$$\mathcal{P} = \{\langle p, q\rangle \in \mathbb{L} \mid p \in P\} \qquad \mathcal{S} = \mathbb{L} \setminus \mathcal{P} \qquad \mathcal{T} = \{\langle p, q\rangle \in \mathbb{L} \mid q \notin Q\} \qquad \mathcal{U} = \mathbb{L} \setminus \mathcal{T}.$$

Additionally, we require that attacks compromise at least as much confidentiality as integrity.

Definition A.1 (Valid Attack)**.** Attack $\langle P, Q\rangle$ is valid if all untrusted principals are public: $Q \subseteq P$.

Theorem IV.2. *Under a valid attack, if* $\blacktriangledown\ell$*, then we have* $\ell \notin \mathcal{S} \cap \mathcal{U}$*.*

Proof. Let $\langle P, Q\rangle$ be a valid attack and $\ell = \langle p, q\rangle$. Assume $\blacktriangledown\ell$ and $\ell \in \mathcal{U}$. By definition, we have $q \in Q$. Unfolding $\blacktriangledown\ell$, we have $q \Rightarrow p$, and since Q is upward closed, we have $p \in Q$. Finally, $p \in Q \subseteq P$, so $\ell \in \mathcal{P}$. □

B Details for Section V-C (Operational Semantics of Choreographies)

Figure 14 gives the full set ideal, real, and concurrent stepping rules for expressions and statements. Figure 15 gives buffer and process stepping rules. Concurrent stepping rules refer to *evaluation contexts*—statements containing a single hole—and the function $\mathrm{hosts}(\cdot)$, which returns the set of hosts that appear in an evaluation context. Rule s-Delay allows skipping over **let**, communication, and selection statements to step a statement in the middle of a program. Rule s-If-Delay allows stepping the body of an **if** statement without resolving the conditional as long as both branches step with the same action. Both rules require the actor of the performed action to be different from the hosts of the statements being skipped over, matching the behavior of target programs where code running on a single host is single-threaded.

C Properties of the Choreography Language

A. Typing and Synchronization

Typing ensures robust declassification and transparent endorsement, which guarantee that declassified values are always trusted, and that endorsed values are always public.

Lemma C.1 (Robust Declassification)**.** *If* $\Gamma \vdash \mathbf{declassify}(t, \ell_\mathrm{f} \to \ell_\mathrm{t}) : h.\ell$*,* $\ell_\mathrm{f} \notin \mathcal{P}$*, and* $\ell_\mathrm{t} \in \mathcal{P}$*, then* $\ell_\mathrm{f} \in \mathcal{T}$*.*

Lemma C.2 (Transparent Endorsement)**.** *If* $\Gamma \vdash \mathbf{endorse}(t, \ell_\mathrm{f} \to \ell_\mathrm{t}) : h.\ell$*,* $\ell_\mathrm{f} \notin \mathcal{T}$*, and* $\ell_\mathrm{t} \in \mathcal{T}$*, then* $\ell_\mathrm{f} \in \mathcal{P}$*.*

Typing has standard properties.

Definition C.3 (Refinement)**.** Define

- $\Gamma_1 \sqsubseteq \Gamma_2$ if $(x : h.\ell_2) \in \Gamma_2$ implies $(x : h.\ell_1) \in \Gamma_1$ for some ℓ_1 such that $\ell_1 \sqsubseteq \ell_2$.
- $\Delta_1 \sqsubseteq \Delta_2$ if $\Delta_1(h_1 h_2) \sqsubseteq \Delta_2(h_1 h_2)$ for all $h_1, h_2 \in H$.

Lemma C.4 (Subsumption)**.** *We have*

1) *If* $\Gamma \vdash e : h.\ell$*,* $\Gamma' \sqsubseteq \Gamma$*, and* $\ell \sqsubseteq \ell'$*, then* $\Gamma' \vdash e : h.\ell'$*.*
2) *If* $\Gamma \vdash s$ *and* $\Gamma' \sqsubseteq \Gamma$*, then* $\Gamma' \vdash s$*.*
3) *If* $\Delta \Vdash s$ *and* $\Delta' \sqsubseteq \Delta$*, then* $\Gamma' \Vdash s$*.*

Lemma C.5 (Substitution)**.** *Substitution preserves typing:*

1) *If* $(\Gamma, x : h'.\ell') \vdash e : h.\ell$*, then* $\Gamma \vdash e[v/x] : h.\ell$*.*
2) *If* $(\Gamma, x : h.\ell) \vdash s$*, then* $\Gamma \vdash s[v/x]$*.*

A well-typed program remains well typed under execution and all corruption.

Lemma C.6 (Type Preservation)**.** *If* $\Gamma \vdash s$ *and* $s \xrightarrow{a} s'$*, then* $\Gamma \vdash s'$*.*

Lemma C.7 (Robust Typing)**.** *If* $\Gamma \vdash s$*, then* $\Gamma \vdash (\!|s|\!)$*.*

A well-synchronized program remains well synchronized under execution and all corruption.

[4]For those familiar with order theory, P and Q must be *prime filters* of $\mathbb{P}$.

$$\boxed{h.e \xrightarrow{a}_{\mathrm{i}} v}$$

e-OPERATOR
$$\frac{v = \mathrm{eval}(f, v_1, \ldots, v_n)}{h.f(v_1, \ldots, v_n) \xrightarrow{!hh0}_{\mathrm{i}} v}$$

e-DECLASSIFY
$$\frac{\ell_{\mathrm{f}} \notin \mathcal{P} \qquad \ell_{\mathrm{t}} \in \mathcal{P}}{h.\mathbf{declassify}(v, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) \xrightarrow{!h\mathrm{Adv}v}_{\mathrm{i}} v}$$

e-DECLASSIFY-SKIP
$$\frac{\ell_{\mathrm{f}} \in \mathcal{P} \vee \ell_{\mathrm{t}} \notin \mathcal{P}}{h.\mathbf{declassify}(v, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) \xrightarrow{!hh0}_{\mathrm{i}} v}$$

e-ENDORSE
$$\frac{\ell_{\mathrm{f}} \notin \mathcal{T} \qquad \ell_{\mathrm{t}} \in \mathcal{T}}{h.\mathbf{endorse}(v, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) \xrightarrow{?\mathrm{Adv}hv'}_{\mathrm{i}} v'}$$

e-ENDORSE-SKIP
$$\frac{\ell_{\mathrm{f}} \in \mathcal{T} \vee \ell_{\mathrm{t}} \notin \mathcal{T}}{h.\mathbf{endorse}(v, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) \xrightarrow{!hh0}_{\mathrm{i}} v}$$

e-INPUT
$$\frac{\mathbb{L}(h) \in \mathcal{T}}{h.\mathbf{input} \xrightarrow{?\mathrm{Env}hv}_{\mathrm{i}} v}$$

e-INPUT-MALICIOUS
$$\frac{\mathbb{L}(h) \notin \mathcal{T}}{h.\mathbf{input} \xrightarrow{!hh0}_{\mathrm{i}} 0}$$

e-OUTPUT
$$\frac{\mathbb{L}(h) \in \mathcal{T}}{h.\mathbf{output}\ v \xrightarrow{!h\mathrm{Env}v}_{\mathrm{i}} 0}$$

e-OUTPUT-MALICIOUS
$$\frac{\mathbb{L}(h) \notin \mathcal{T}}{h.\mathbf{output}\ v \xrightarrow{!hh0}_{\mathrm{i}} 0}$$

e-RECEIVE
$$h.\mathbf{receive}\ h' \xrightarrow{!hh0}_{\mathrm{i}} 0$$

e-SEND
$$h.\mathbf{send}\ v\ \mathbf{to}\ h' \xrightarrow{!hh0}_{\mathrm{i}} 0$$

$$\boxed{s \xrightarrow{a}_{\mathrm{i}} s'}$$

s-LET
$$\frac{h.e \xrightarrow{a}_{\mathrm{i}} v}{\mathbf{let}\ h.x = e; s \xrightarrow{a}_{\mathrm{i}} s[v/x]}$$

s-COMMUNICATE
$$h_1.v \rightsquigarrow h_2.x; s \xrightarrow{!h_1h_10}_{\mathrm{i}} s[v/x]$$

s-SELECT
$$h_1[v] \rightsquigarrow h_2; s \xrightarrow{!h_1h_10}_{\mathrm{i}} s$$

s-IF
$$\frac{i = \text{ if } v \neq 0 \text{ then } 1 \text{ else } 2}{\mathbf{if}(h.v, s_1, s_2) \xrightarrow{!hh0}_{\mathrm{i}} s_i}$$

(a) Ideal stepping rules for expressions and statements.

$$\boxed{h.e \xrightarrow{a}_{\mathrm{r}} v}$$

e-DECLASSIFY-REAL
$$\frac{\ell_{\mathrm{f}} \notin \mathcal{P} \qquad \ell_{\mathrm{t}} \in \mathcal{P}}{h.\mathbf{declassify}(v, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) \xrightarrow{!hh0}_{\mathrm{r}} v}$$

e-ENDORSE-REAL
$$\frac{\ell_{\mathrm{f}} \notin \mathcal{T} \qquad \ell_{\mathrm{t}} \in \mathcal{T}}{h.\mathbf{endorse}(v, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) \xrightarrow{!hh0}_{\mathrm{r}} v}$$

e-RECEIVE-REAL
$$h.\mathbf{receive}\ h' \xrightarrow{?h'hv}_{\mathrm{r}} v$$

e-SEND-REAL
$$h.\mathbf{send}\ v\ \mathbf{to}\ h' \xrightarrow{!hh'v}_{\mathrm{r}} 0$$

$$\boxed{s \xrightarrow{a}_{\mathrm{r}} s'}$$

s-COMMUNICATE-REAL
$$h_1.v \rightsquigarrow h_2.x; s \xrightarrow{!h_1h_2v}_{\mathrm{r}} s[v/x]$$

s-SELECT-REAL
$$h_1[v] \rightsquigarrow h_2; s \xrightarrow{!h_1h_2v}_{\mathrm{r}} s$$

s-CASE
$$\mathbf{case}\ (h_1 \rightsquigarrow h_2)\ \{v \mapsto s, \ldots\} \xrightarrow{?h_1h_2v}_{\mathrm{r}} s$$

(b) Real stepping rules for expressions and statements. These override the rules in fig. 14a.

$$\boxed{s \xrightarrow{a}{}^{\mathrm{c}}_{\alpha} s'}$$

s-SEQUENTIAL
$$\frac{s \xrightarrow{a}_{\alpha} s'}{s \xrightarrow{a}{}^{\mathrm{c}}_{\alpha} s'}$$

s-DELAY
$$\frac{s \xrightarrow{a}{}^{\mathrm{c}}_{\alpha} s' \qquad \mathrm{actor}(a) \notin \mathrm{hosts}(E)}{E[s] \xrightarrow{a}{}^{\mathrm{c}}_{\alpha} E[s']}$$

s-IF-DELAY
$$\frac{s_1 \xrightarrow{a}{}^{\mathrm{c}}_{\alpha} s_1' \qquad s_2 \xrightarrow{a}{}^{\mathrm{c}}_{\alpha} s_2' \qquad \mathrm{actor}(a) \neq h}{\mathbf{if}(h.t, s_1, s_2) \xrightarrow{a}{}^{\mathrm{c}}_{\alpha} \mathbf{if}(h.t, s_1', s_2')}$$

$$\boxed{E} \qquad \boxed{\mathrm{hosts}(E) = H}$$

$$\text{Evaluation Contexts} \quad E \quad ::= \quad \mathbf{let}\ h.x = e; [\cdot] \mid h_1.t \rightsquigarrow h_2.x; [\cdot] \mid h_1[v] \rightsquigarrow h_2; [\cdot]$$

$$\mathrm{hosts}(\mathbf{let}\ h.x = e; [\cdot]) = \{h\} \qquad \mathrm{hosts}(h_1.t \rightsquigarrow h_2.x; [\cdot]) = \{h_1, h_2\} \qquad \mathrm{hosts}(h_1[v] \rightsquigarrow h_2; [\cdot]) = \{h_1, h_2\}$$

(c) Concurrent lifting of ideal/real stepping rules.

Figure 14. Ideal, real, and concurrent stepping rules for expressions and statements.

$\boxed{B \xrightarrow{a} B'}$

B-INPUT
$$B[c_1c_2 := V] \xrightarrow{?c_1c_2v} B[c_1c_2 := V \cdot v]$$

B-OUTPUT
$$B[c_1c_2 := v \cdot V] \xrightarrow{!c_1c_2v} B[c_1c_2 := V]$$

$\boxed{w \xrightarrow{a} w'}$

w-INPUT
$$\frac{c_1 \notin H \qquad c_2 \in H \qquad B \xrightarrow{?c_1c_2v} B'}{\langle H, B, s\rangle \xrightarrow{?c_1c_2v} \langle H, B', s\rangle}$$

w-DISCARD
$$\frac{c_1 \in H \vee c_2 \notin H}{\langle H, B, s\rangle \xrightarrow{?c_1c_2v} \langle H, B, s\rangle}$$

w-INTERNAL
$$\frac{B \xrightarrow{!c_1c_2v} B' \qquad s \xrightarrow{?c_1c_2v} s'}{\langle H, B, s\rangle \xrightarrow{!c_2c_20} \langle H, B', s'\rangle}$$

w-OUTPUT
$$\frac{s \xrightarrow{!m} s'}{\langle H, B, s\rangle \xrightarrow{!m} \langle H, B, s'\rangle}$$

Figure 15. Stepping rules for buffers and processes.

Lemma C.8 (Synchrony Preservation)**.** *If $\Delta \Vdash s$ and $s \xrightarrow{a} s'$, then $\Delta \Vdash s'$.*

Lemma C.9 (Robust Synchrony)**.** *If $\Delta \Vdash s$, then $\Delta \Vdash (\!|s|\!)$.*

A host can only output if it is synchronized with all previous external actions.

Lemma C.10 (Output Synchronization)**.** *If $\Delta \Vdash s$ and $s \xrightarrow{!m}$ with m external, then* $\mathrm{synched}(\Delta, h)$.

B. Operational Semantics

Below, we write $\rightarrow$ to stand for any of $\rightarrow_{\mathrm{i}}$, $\rightarrow_{\mathrm{r}}$, $\rightarrow^{\mathrm{c}}_{\mathrm{i}}$, or $\rightarrow^{\mathrm{c}}_{\mathrm{r}}$.
Processes never refuse input.

Lemma C.11 (Input Totality)**.** *For all w and m, there exists w' such that $w \xrightarrow{?m} w'$.*

The stepping judgments are nondeterministic since inputs are externally controlled (different input values lead to different states), and, for concurrent judgments, outputs and internal actions are independent across hosts. However, processes are fully deterministic when the action is fixed.

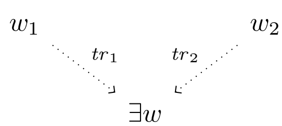
Lemma C.12 (Internal Determinism)**.** *If $w \xrightarrow{a} w_1$ and $w \xrightarrow{a} w_2$, then $w_1 = w_2$.*

Lemma C.13 (Output Determinism)**.** *If $w \xrightarrow{!m_1} w_1$, $w \xrightarrow{!m_2} w_2$, and $\mathrm{actor}(!m_1) = \mathrm{actor}(!m_2)$, then $m_1 = m_2$.*

These results lift to configurations W as long as the configuration does not contain duplicate hosts.

D DETAILS FOR SECTION VI-B (CORRECTNESS OF SEQUENTIALIZATION)

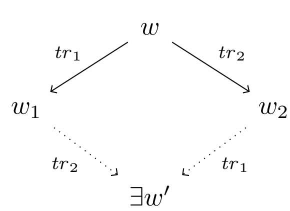
Definition D.1 (Joinable Processes)**.** We write $w_1 \downarrow w_2$ if there exist traces tr_1 and tr_2 containing only internal actions such that $w_1 \xrightarrow{tr_1}{}^{\mathrm{c}}_{\mathrm{i}} w$ and $w_2 \xrightarrow{tr_2}{}^{\mathrm{c}}_{\mathrm{i}} w$ for some w. Diagrammatically:



We prove confluence through a *diamond* lemma, which allows reordering *independent* actions.

Definition D.2 (Independent Actions)**.** Actions a_1 and a_2 are independent, written $a_1 \perp\!\!\!\perp a_2$, if one is an input while the other is an output, or they are on different channels. We write $tr_1 \perp\!\!\!\perp tr_2$ if $a_1 \perp\!\!\!\perp a_2$ for all $a_1 \in tr_1$ and $a_2 \in tr_2$.

Lemma D.3 (Diamond for Processes)**.** *If $w \xrightarrow{tr_1}{}^{\mathrm{c}}_{\mathrm{i}} w_1$, $w \xrightarrow{tr_2}{}^{\mathrm{c}}_{\mathrm{i}} w_2$, and $tr_1 \perp\!\!\!\perp tr_2$, then $w_1 \xrightarrow{tr_2}{}^{\mathrm{c}}_{\mathrm{i}} w'$ and $w_2 \xrightarrow{tr_1}{}^{\mathrm{c}}_{\mathrm{i}} w'$ for some w'. Diagrammatically:*



Lemma D.3 does the heavy lifting when proving multiple confluence results below, and requires quite a bit of work to show. We first prove a diamond lemma for statements, and then lift it to processes.

Lemma D.4 (Half Diamond for Statements)**.** *If $s \xrightarrow{a_1}_{\mathrm{i}} s_1$, $s \xrightarrow{a_2}{}^{\mathrm{c}}_{\mathrm{i}} s_2$, and $a_1 \perp\!\!\!\perp a_2$, then $s_1 \xrightarrow{a_2}{}^{\mathrm{c}}_{\mathrm{i}} s'$ and $s_2 \xrightarrow{a_1}_{\mathrm{i}} s'$ for some s'.*

Proof. By case analysis on $s \xrightarrow{a_2}{}^{\mathrm{c}}_{\mathrm{i}} s_2$.

- Case s-SEQUENTIAL. Contradicts $a_1 \perp\!\!\!\perp a_2$.
- Case s-DELAY. By case analysis on the evaluation context followed by inversion on $s \xrightarrow{a_1}_{\mathrm{i}} s_1$. The step for a_1 involves only the head statement and ignores all future statements, whereas the step for a_2 ignores the head statement and involves only a statement in the future. Thus, they can be performed in sequence in either order without changing the end result.
- Case s-IF-DELAY. The step for a_2 steps both branches of the **if** statement, whereas the step for a_1 selects a branch. They can be performed in sequence in either order. ☐

Lemma D.5 (Diamond for Statements)**.** *If $s \xrightarrow{a_1}{}^{\mathrm{c}}_{\mathrm{i}} s_1$, $s \xrightarrow{a_2}{}^{\mathrm{c}}_{\mathrm{i}} s_2$, and $a_1 \perp\!\!\!\perp a_2$, then $s_1 \xrightarrow{a_2}{}^{\mathrm{c}}_{\mathrm{i}} s'$ and $s_2 \xrightarrow{a_1}{}^{\mathrm{c}}_{\mathrm{i}} s'$ for some s'.*

Proof. By induction on the derivations of both stepping judgments. If either is by rule s-SEQUENTIAL, we conclude by lemma D.4. Otherwise, both steps ignore the head of s using the same delay rule. We appeal to the induction hypothesis, and use the same delay rule to get a complete derivation. ☐

Proof of lemma D.3. We prove the statement when tr_1 and tr_2 are single actions; the more general statement follows straightforwardly by induction on tr_1 followed by induction on tr_2.

We proceed by case analysis on both stepping judgments.

- Both steps are input (rules w-INPUT and w-DISCARD). Since $tr_1 \perp\!\!\!\perp tr_2$, the input messages are added at the end of two different queues. Both actions can be performed in either order without affecting the end result.
- One step is input, the other is by rule w-INTERNAL. The input step adds a message to a queue, while rule w-INTERNAL pops a message from a queue and feeds it to the choreography. The queues must be different since $tr_1 \perp\!\!\!\perp tr_2$, so the steps are independent.
- One step is input, the other is by rule w-OUTPUT. The input step only affects the queue, and the output step only affects the choreography, so the steps are independent.
- Both steps are output (rules w-INTERNAL and w-OUTPUT). If either step is by rule w-INTERNAL, then we use $tr_1 \perp\!\!\!\perp tr_2$ as before to show we pull messages out of different queues. This allows reordering changes to the buffer. Lemma D.5 finishes the proof. ☐

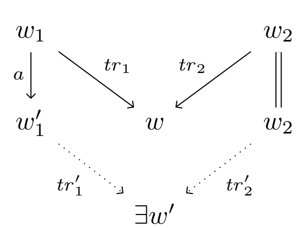
The proof of lemma D.3 reasons generically about buffers, and appeals to a diamond lemma for statements in a black-box manner. This means we can generalize lemma D.3 to arbitrary (combinations of) stepping relations without extra work as long as a diamond property for the same relations holds for statements.

Lemma D.6 (Generalized Diamond for Processes)**.** *Assume the diamond property holds for statements with stepping relations $\longrightarrow_1$ and $\longrightarrow_2$. If $w \xrightarrow{tr_1}_1 w_1$, $w \xrightarrow{tr_2}_2 w_2$, and $tr_1 \perp\!\!\!\perp tr_2$, then $w_1 \xrightarrow{tr_2}_2 w'$ and $w_2 \xrightarrow{tr_1}_1 w'$ for some w'.*

Proof. Same as the proof of lemma D.3. ☐

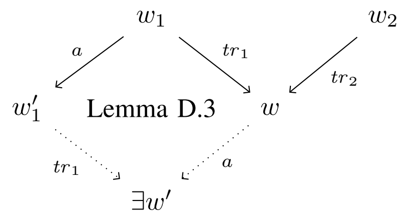
Processes remain joinable after taking internal or matching steps.

Lemma D.7 (Internal Action)**.** *If $w_1 \downarrow w_2$ and $w_1 \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{i}} w_1'$ for a internal, then $w_1' \downarrow w_2$. Diagrammatically:*

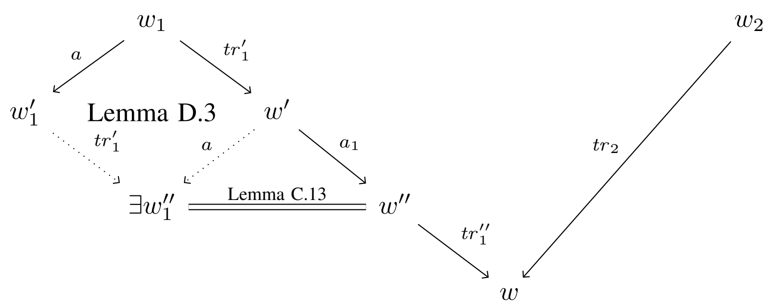


Proof. Since w_1 and w_2 are joinable, there exist w and internal tr_1, tr_2 such that $w_1 \xrightarrow{tr_1}{}^{\mathrm{c}}_{\mathrm{i}} w$ and $w_2 \xrightarrow{tr_2}{}^{\mathrm{c}}_{\mathrm{i}} w$. We case on whether $a \perp\!\!\!\perp tr_1$.

- Case $a \perp\!\!\!\perp tr_1$. Lemma D.3 gives w' such that $w'_1 \xrightarrow{tr_1}{}^{\mathrm{c}}_{\mathrm{i}} w'$ and $w \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{i}} w'$. Since a and tr_2 are internal, so is $tr_2 \cdot a$. Thus, $w'_1 \downarrow w_2$ through tr_1 and $tr_2 \cdot a$.

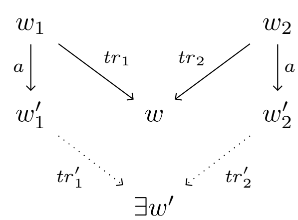


- Case $a \not\perp\!\!\!\perp tr_1$. Let a_1 be the first action in tr_1 such that $a \not\perp\!\!\!\perp a_1$, that is, $tr_1 = tr'_1 \cdot a_1 \cdot tr''_1$ with $a \perp\!\!\!\perp tr'_1$. We have, $w_1 \xrightarrow{tr'_1}{}^{\mathrm{c}}_{\mathrm{i}} w' \xrightarrow{a_1}{}^{\mathrm{c}}_{\mathrm{i}} w'' \xrightarrow{tr''_1}{}^{\mathrm{c}}_{\mathrm{i}} w$. Lemma D.3 gives w''_1 such that $w'_1 \xrightarrow{tr'_1}{}^{\mathrm{c}}_{\mathrm{i}} w''_1$ and $w' \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{i}} w''_1$. We now have $w' \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{i}} w''_1$ and $w' \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{i}} w''$, however, the stepping judgment is deterministic on dependent internal actions, thus lemma C.13 gives $w''_1 = w''$.[5] Finally, $w'_1 \downarrow w_2$ through $tr'_1 \cdot tr''_1$ and tr_2.

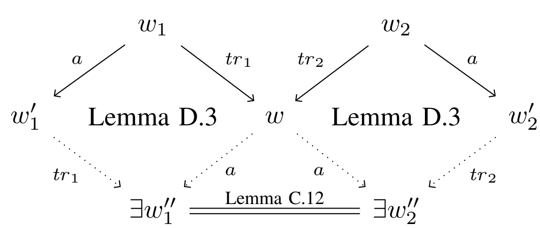


□

Lemma D.8 (Matching Actions)**.** *If $w_1 \downarrow w_2$, $w_1 \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{i}} w'_1$, and $w_2 \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{i}} w'_2$, then $w'_1 \downarrow w'_2$. Diagrammatically:*



Proof. Since w_1 and w_2 are joinable, there exist w and internal tr_1, tr_2 such that $w_1 \xrightarrow{tr_1}{}^{\mathrm{c}}_{\mathrm{i}} w$ and $w_2 \xrightarrow{tr_2}{}^{\mathrm{c}}_{\mathrm{i}} w$. If a is internal, then the result follows by two applications of lemma D.7. Otherwise, $a \perp\!\!\!\perp tr_1$ and $a \perp\!\!\!\perp tr_2$. The result follows from two applications of lemma D.3, and one application of lemma C.12:



□

If a well-typed, well-synchronized program can take an output step concurrently, then it can take the same step sequentially (after taking the series of internal steps leading up to the output). We write $w \overset{!m}{\twoheadrightarrow}_{\mathrm{i}} w'$ if $w \xrightarrow{tr \,\cdot\, !m}_{\mathrm{i}} w'$ for some internal tr.

Lemma D.9 (Sequential Execution)**.** *If $\epsilon \vdash w$, $\Delta \Vdash w$, and $w \xrightarrow{!m}{}^{\mathrm{c}}_{\mathrm{i}}$ for m external, then $w \overset{!m}{\twoheadrightarrow}_{\mathrm{i}}$.*

Proof sketch. By induction on the stepping relation. If the step is by rule s-SEQUENTIAL, then $w \xrightarrow{!m}_{\mathrm{i}}$ and we are done. Otherwise, it must be by rule s-DELAY. We need to show that we can take a sequential internal step by casing on the evaluation

[5]More specifically, a and a_1 are internal actions, which are represented as outputs. Since $a \not\perp\!\!\!\perp a_1$, we have $\mathrm{actor}(a) = \mathrm{actor}(a_1)$, so lemma C.13 applies.

context E. Note that the top statement in E must be internal, otherwise we get a contradiction by lemma C.10. Since w has no free variables, it can take an internal step. ☐

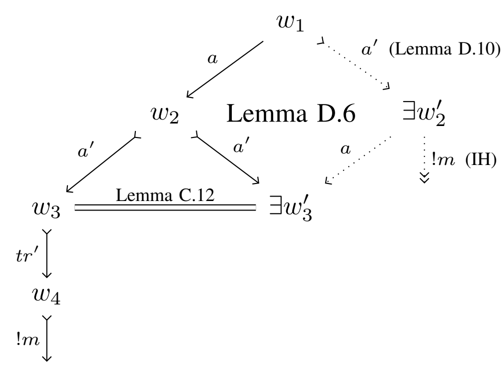
Lemma D.10 (Step Over)**.** *If* $w_1 \xrightarrow{a_1}{}^{c}_{i} w_2 \xrightarrow{a_2}_{i}$*, then either* $w_1 \xrightarrow{a_1}_{i} w_2$*, or* $w_1 \xrightarrow{a_2}_{i}$ *and* $a_1 \perp\!\!\!\perp a_2$*.*

Proof. Assume the first step cannot be done sequentially (otherwise we are done). Then, the first step uses a delay rule (rules s-DELAY and s-IF-DELAY), which ignores the head statement of w_1. The second step is sequential so it only depends on the head statement of w_2, which is the same as the head statement of w_1, thus, w_1 can step with a_2. We get that $a_1 \perp\!\!\!\perp a_2$ from the side conditions on delay rules, which require $\mathrm{actor}(a_1)$ to be different from $\mathrm{actor}(a_2)$. ☐

Lemma D.11 (Delayed Step)**.** *If* $w_1 \xrightarrow{a}{}^{c}_{i} w_2 \xrightarrow{!m}\!\!\!\!\rightarrow_{i}$ *for* a *internal, then* $w_1 \xrightarrow{!m}\!\!\!\!\rightarrow_{i}$*.*

Proof. Unfolding the definition of $\twoheadrightarrow_i$ gives $w_2 \xrightarrow{tr}_{i} w_3 \xrightarrow{!m}_{i}$ for some tr and w_3. We proceed by induction on tr. In either case, we are done if $w_1 \xrightarrow{a}_{i} w_2$, so assume w_1 cannot perform a sequentially.

- Case $tr = \epsilon$. We have $w_1 \xrightarrow{a}{}^{c}_{i} w_2 \xrightarrow{!m}_{i}$. Lemma D.10 gives $w_1 \xrightarrow{!m}_{i}$, and thus $w_1 \xrightarrow{!m}\!\!\!\!\rightarrow_{i}$.
- Case $tr = a' \cdot tr'$. We have $w_1 \xrightarrow{a}{}^{c}_{i} w_2 \xrightarrow{a'}_{i} w_3 \xrightarrow{tr'}_{i} w_4 \xrightarrow{!m}_{i}$. Lemma D.10 gives $w_1 \xrightarrow{a'}_{i} w_2'$ for some w_2'. Lemma D.6 with lemma D.4 then gives $w_1 \xrightarrow{a'}_{i} w_2' \xrightarrow{a}{}^{c}_{i} w_3'$ for some w_3', and lemma C.12 shows $w_3' = w_3$. We use the induction hypothesis on $w_2' \xrightarrow{a}{}^{c}_{i} w_3 \xrightarrow{tr'}_{i} w_4 \xrightarrow{!m}_{i}$ to get $w_2' \xrightarrow{!m}\!\!\!\!\rightarrow_{i}$, and combined with $w_1 \xrightarrow{a'}_{i} w_2'$, we get $w_1 \xrightarrow{!m}\!\!\!\!\rightarrow_{i}$.
  Diagrammatically (we use tails to denote sequential steps):



☐

If two processes are joinable and one of them can concurrently perform an output action, then the other can perform the same action sequentially (after unwinding).

Lemma D.12 (Matching Outputs)**.** *Let* w_1 *and* w_2 *be such that* $w_1 \downarrow w_2$*,* $\epsilon \vdash w_2$*, and* $\Delta \Vdash w_2$*. If* $w_1 \xrightarrow{!m}{}^{c}_{i}$ *for* m *external, then* $w_2 \xrightarrow{!m}\!\!\!\!\rightarrow_{i}$*.*

Proof. Since w_1 and w_2 are joinable, there exist w and internal tr_1, tr_2 such that $w_1 \xrightarrow{tr_1}{}^{c}_{i} w$ and $w_2 \xrightarrow{tr_2}{}^{c}_{i} w$. Because m is external and tr_1 is internal, we have $!m \perp\!\!\!\perp tr_1$, so lemma D.3 gives $w \xrightarrow{!m}{}^{c}_{i}$. Now we have $w_2 \xrightarrow{tr_2 \cdot !m}{}^{c}_{i}$, and we want to show $w_2 \xrightarrow{!m}\!\!\!\!\rightarrow_{i}$. We proceed by induction on tr_2. When tr_2 is empty, lemma D.9 completes the proof. When $tr_2 = a \cdot tr_2'$, the induction hypothesis gives $w_2 \xrightarrow{a}{}^{c}_{i} w_2' \xrightarrow{!m}\!\!\!\!\rightarrow_{i}$, and lemma D.11 gives the desired result. ☐

Proof of theorem VI.5. The simulator maintains a public view of the concurrent process, and runs the adversary against this view. When the adversary schedules an output action, the simulator schedules the sequential process until it performs the same output; the simulator does nothing for input and internal actions. Lemma D.12 guarantees the sequential program can perform the output. The primary invariant, that the concurrent and sequential processes remain joinable, is ensured by lemmas D.7 and D.8.

More formally, we show UC simulation as follows.

Simulator The simulator has the form $\mathcal{S}(\mathcal{A} \parallel w_1', w_2')$ where w_1' is the public view of the concurrent process, and w_2' is the public view of the sequential process. When the simulator receives an input from the environment or a **declassify** message from the sequential process, it feeds the message to $\mathcal{A}$, w_1', and w_2'. When the adversary outputs a value (for the environment or for an **endorse** expression), the simulator feeds it to w_1' and w_2', and outputs the same value. When the simulator receives an internal message from the sequential process (which indicates the sequential process has taken a step), it steps w_2'. The simulator only allows an output step for the sequential process if w_1' can perform same output.

Bisimulation Relation Let $\mathcal{A} \parallel w_1 \; R \; \mathcal{S}(\mathcal{A}' \parallel w_1', w_2') \parallel w_2$ if: (1) $\mathcal{A} = \mathcal{A}'$, (2) $w_1 =_{\mathcal{P}} w_1'$, (3) $w_2 =_{\mathcal{P}} w_2'$, (4) $w_1 \downarrow w_2$, and (5) $\epsilon \vdash w_2$ and $\Delta \Vdash w_2$ for some Δ.

Simulation We claim R is a weak bisimulation.

Conditions (4) and (5) ensure lemma D.12 is applicable, which in turn ensures the external behavior of both systems is the same.

Condition (1) is preserved since w_1' is an accurate public view of w_1 (condition (2)). Conditions (2) and (3) are preserved since messages from **declassify** are sufficient to maintain a public view. Lemmas D.7 and D.8 ensure condition (4) is preserved. Lemmas C.6 and C.8 ensure condition (5) is preserved. □

E Details for Section VI-C (Correctness of Ideal Execution)

Consider choreography w and its corruption $(\!|w|\!)$ when Alice is malicious:

// w	// $(\!|w|\!)$
let Alice.x = **input**;	
Alice.$x \rightsquigarrow$ Bob.y;	**let** Bob.y = **receive** Alice;
Alice.$x \rightsquigarrow$ Chuck.z;	**let** Chuck.z = **receive** Alice;
let Bob.$y' = y + 1$;	**let** Bob.$y' = y + 1$;
Bob.$y' \rightsquigarrow$ Alice.x';	**let** Bob._ = **send** y' **to** Alice;

The function $(\!|\cdot|\!)$ erases all code on Alice (the first **let** statement) since a malicious host does not follow the choreography and has arbitrary behavior. Additionally, it replaces all communication statements involving Alice with **receive/send** statements, capturing the fact that Alice need not use the variables specified in the choreography (x and x'). In particular, even though the original choreography specifies Alice sends *the same* value to Bob and Chuck, a malicious Alice can send *different* values. Giving Alice the power to *equivocate* in this manner can compromise security, for instance, Alice could cause Bob and Chuck to disagree on control flow if x is used as a conditional guard. Information-flow checking prevents Alice from exploiting this power.

Information-flow checking ensures untrusted data (from malicious hosts) cannot influence trusted data (of nonmalicious hosts). We formalize this intuition by erasing all data from malicious hosts in the ideal semantics: instead of receiving the value of y from Alice (i.e., the adversary), Bob simply assigns 0 to y (Chuck does the same for z). The adversary cannot possibly have any control over trusted data if all data coming from the adversary is replaced with 0. Note that erasing untrusted data can change the adversary's view. In the example, Bob sends $y' = y + 1$ to Alice, which is different from sending $0 + 1$. The simulator can compute the correct value in this case since y comes from the adversary (which the simulator has access to), and 1 is a fixed constant. In the general case, the simulator can compute all public values, and our type system ensures only public values are sent to dishonest hosts (rules ℓ-Send and ℓ-Communicate).

In addition to preventing the adversary from corrupting trusted values, we must prevent the adversary from learning secrets. In the real semantics, the adversary witnesses all communication and can read any message if at least one endpoint is dishonest. Information-flow checking ensures the adversary does not learn anything new by reading these messages. We formalize this intuition by erasing communication: in the ideal semantics, communication statements step internally. The simulator must again recreate these hidden messages for the adversary, which is possible since our type system ensures the messages the adversary can read are public.

Simply discarding all untrusted data and hiding all secret data weakens the adversary in the ideal semantics too much. We bridge the gap between the real and ideal semantics through downgrade expressions. An **endorse** expression indicates that some untrusted data should be treated as trusted, so in the ideal semantics, an **endorse** inputs data from the adversary. Dually, a **declassify** expression indicates some secret data should be treated as public, so a **declassify** outputs data to the adversary. Explicit **declassify/endorse** expressions capture programmer intent. Going back to the example, our type system requires Bob and Chuck to **endorse** x before using it in a trusted context. If Bob and Chuck separately **endorse** x, then they might get two different values. If there is only one **endorse** (e.g., a separate trusted host performs the **endorse** and shares the result), then there can only be one value.

The core of the simulation result is showing that the simulator can use **declassify** expressions to recreate all data no longer leaked through communication, and **endorse** expressions to influence all data no longer corruptible through **receive** expressions. For this to work, we need to ensure the ideal choreography outputs the correct value to the simulator when performing a **declassify**, and we need to ensure the simulator can input the correct value to the ideal choreography for an **endorse**. For example, when the ideal choreography performs **declassify** x, we must ensure the value of x is the same in the real and ideal choreographies. This is nontrivial since the ideal semantics replaces all untrusted data with 0. *Robust declassification* requires only trusted data is declassified, and type checking ensures untrusted data does not influence trusted data. Thus, x is trusted and erased values cannot influence its value. Similarly, when the ideal choreography performs **endorse** x, the

$\boxed{h.e \xrightarrow{a}_{\text{sim}} v}$

$$\dfrac{\ell_{\mathrm{f}} \notin \mathcal{P} \qquad \ell_{\mathrm{t}} \in \mathcal{P}}{h.\mathbf{declassify}(v, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) \xrightarrow{?h\mathrm{Adv}v'}_{\text{sim}} v'}\ e\text{-DECLASSIFY-SIMULATOR} \qquad \dfrac{\ell_{\mathrm{f}} \notin \mathcal{T} \qquad \ell_{\mathrm{t}} \in \mathcal{T}}{h.\mathbf{endorse}(v, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) \xrightarrow{!\mathrm{Adv}hv'}_{\text{sim}} v}\ e\text{-ENDORSE-SIMULATOR}$$

Figure 16. Stepping rules used internally by the simulator. These override fig. 14b.

simulator must compute the value x would have in the real choreography and send that to the ideal choreography. *Transparent endorsement* requires only public data is endorsed, and the simulator can recreate all public data.

The simulator maintains a public view of the real process, and runs the adversary against this view. It uses the rules in fig. 16, which flip the roles of **declassify** and **endorse**. We maintain the invariant that the simulator's version of the choreography matches the real one on *public values*, and the ideal choreography matches the real one on *trusted values*. Next, we define what it means for two terms to agree on public/trusted values.

Definition E.1 (Closing Substitution)**.** A closing substitution $\sigma : \Gamma$ is a mapping from variables to values $\sigma : \mathrm{dom}(\Gamma) \to \mathbb{V}$.

Definition E.2 (Channel Label)**.** A channel's label derives from the labels of its endpoints:

$$\mathbb{L}(c_1 c_2 v) = \mathbb{L}(c_1 c_2) = \mathbb{L}(c_1) \vee \mathbb{L}(c_2) \qquad\qquad \mathbb{L}(?m) = \mathbb{L}(!m) = \mathbb{L}(m).$$

We let $\mathbb{L}(\mathrm{Adv}) = \mathbf{0}^{\leftarrow}$ and $\mathbb{L}(\mathrm{Env}) = \mathbf{0}$, which leads to $\mathbb{L}(\mathrm{Env}\, c) = \mathbb{L}(c\, \mathrm{Env}) = \mathbb{L}(c)$ and $\mathbb{L}(\mathrm{Adv}\, c) = \mathbb{L}(c\, \mathrm{Adv}) = \mathbb{L}(c)^{\leftarrow}$ (communication with the adversary is public, and is trusted only if the other endpoint is).

Definition E.3 (Syntactic L-Equivalence)**.** For a set of labels $L \subseteq \mathbb{L}$, define $=_L$ as follows.

- $c_1 c_2 v_1 =_L c_1 c_2 v_2$ if $\mathbb{L}(c_1 c_2) \in L$ implies $v_1 = v_2$.
- $?m_1 =_L ?m_2$ and $!m_1 =_L !m_2$ if $m_1 =_L m_2$.
- $s_1 =_L s_2$ if there exist Γ_1, Γ_2, and s with $(\Gamma_1, \Gamma_2) \vdash s$, and substitutions $\sigma_1, \sigma_2 : \Gamma_2$ such that $\sigma_1(s) = s_1$ and $\sigma_2(s) = s_2$. Additionally, for $(x : \ell.h) \in (\Gamma_1, \Gamma_2)$, we require $\mathbb{L}(h) \Rightarrow \ell$, and for $(x : \ell.h) \in \Gamma_2$, we require $\ell \notin L$.
- $B_1 =_L B_2$ if $B_1(c_1 c_2) = B_2(c_1 c_2)$ for all c_1 and c_2 such that $\mathbb{L}(c_1 c_2) \in L$.
- $\langle H, B_1, s_1 \rangle =_L \langle H, B_2, s_2 \rangle$ if $B_1 =_L B_2$ and $s_1 =_L s_2$.

We instantiate definition E.3 with $L = \mathcal{P}$ for agreement on public values, and with $L = \mathcal{T}$ for agreement on trusted values. Definition E.3 requires the two terms to have the same structure, but allows some values $v \in \mathbb{V}$ (those with labels *not* in L) to differ between them. For example, two messages can only be equivalent if they are on the same channel. Additionally, they must carry the same value if the channel is public and we are considering public equality ($=_{\mathcal{P}}$); they are allowed to carry different values otherwise. Action and buffer equivalence simply lift the definition for messages. Equivalence for statements demands further explanation.

Values are fixed constants and can be assigned any label. It is therefore not immediate which values should be allowed to differ between statements. For example, consider the following statements that have the same structure but differ in the value of x:

// s_1	// s_2
let Alice.$x = 0$;	**let** Alice.$x = 1$;
Alice.$x \rightsquigarrow$ Bob.y;	Alice.$x \rightsquigarrow$ Bob.y;

The intuition behind definition E.3 is that s_1 and s_2 are equivalent if x can be treated as secret/untrusted. To check that, definition E.3 abstracts out values where the two statement differ to find a common statement, and type-checks the generalized statement in a context where all introduced variables are marked as secret/untrusted. For example, we could pick s as follows

// s
let Alice.$x = x'$;
Alice.$x \rightsquigarrow$ Bob.y;

along with substitutions $\sigma_1 = \{x' \mapsto 0\}$ and $\sigma_2 = \{x' \mapsto 1\}$. If s can be typed under a context where x' is considered secret, then $s_1 =_{\mathcal{P}} s_2$. However, if Bob has a public label (is dishonest), for example, then there is no such context.

Definition E.3 splits the context into Γ_1 and Γ_2, with the substitutions only assigning values for variables in Γ_2. Context Γ_1 is added to allow relating open terms, which is needed for inductive cases of some proofs.

For the rest of this section, we assume $L = \mathcal{P}$ or $L = \mathcal{T}$. Moreover, whenever **receive** h or **send** t **to** h appears in a program, we assume $\mathbb{L}(h) \notin \mathcal{T}$ (which is ensured by $(\!|\cdot|\!)$).[6]

[6]Our results hold for more general L, but we do not need this generality.

Equivalent choreographies remain equivalent given equivalent inputs and after producing outputs on the same host.

Lemma E.4 (Equivalence Preservation)**.** *Assume $s_1 =_L s_2$ and $s_1 \xrightarrow{a_1}{}^{\mathrm{c}}_{\mathrm{r}} s_1'$ without using rule e-*DECLASSIFY-REAL *or rule e-*ENDORSE-REAL*.*

- *If $a_1 = ?m_1$, then $s_2 \xrightarrow{?m_2}{}^{\mathrm{c}}_{\mathrm{r}} s_2'$ with $s_1' =_L s_2'$ for* all *$m_2 =_L m_1$.*
- *If $a_1 = !m_1$, then $s_2 \xrightarrow{!m_2}{}^{\mathrm{c}}_{\mathrm{r}} s_2'$ with $s_1' =_L s_2'$ for* some *m_2 with* $\text{actor}(!m_2) = \text{actor}(!m_1)$.

Proof. By definition E.3, there exists s such that $\Gamma \vdash s$, $\sigma_1(s) = s_1$, and $\sigma_2(s) = s_2$ for some $\Gamma = (\Gamma_1, \Gamma_2)$ and $\sigma_1, \sigma_2 : \Gamma_2$. We proceed by induction on the stepping judgment. In all cases, stepping on s_1 forces certain atomic expressions t to be values v (as opposed to variables x); the same expressions in s_2 must also be values since σ_2 substitutes for the same variables as σ_1. We appeal to this fact implicitly.

- Case s-LET. We have

$$s_1 = \textbf{let } h.x = e_1; s_1'' \qquad s_2 = \textbf{let } h.x = e_2; s_2'' \qquad s = \textbf{let } h.x = e; s''$$

  and

$$h.e_1 \xrightarrow{a_1}_{\mathrm{r}} v_1 \qquad h.e_2 \xrightarrow{a_2}_{\mathrm{r}} v_2$$

  with $\text{actor}(a_1) = \text{actor}(a_2) = h$. Inversion on $\Gamma \vdash s$ gives

$$\frac{\Gamma \vdash e : h.\ell \qquad \mathbb{L}(h) \Rightarrow \ell \qquad \Gamma, x : h.\ell \vdash s''}{\Gamma \vdash s}\ \text{RULE } \ell\text{-LET}$$

  In each case, we either prove $v_1 = v_2$ or $\ell \notin \mathcal{P}$. When $v_1 = v_2$, we define $s' = s''[v/x]$. We then have $\Gamma \vdash s'$ by lemma C.5, $\sigma_1(s') = s_1''[v/x] = s_1'$, and $\sigma_2(s') = s_2''[v/x] = s_2'$, so $s_1' =_L s_2'$.
  When $\ell \notin \mathcal{P}$, we define $\Gamma_2' = (\Gamma_2, x : h.\ell)$, $\sigma_1' = \sigma_1 \cup \{x \mapsto v_1\}$, $\sigma_2' = \sigma_2 \cup \{x \mapsto v_2\}$, which ensures $\sigma_1'(s'') = s_1''[v_1/x] = s_1'$ and $\sigma_2'(s'') = s_2''[v_2/x] = s_2'$. Note that Γ_2 satisfies the requirements of definition E.3, and $\sigma_1', \sigma_2' : \Gamma_2'$, so $s_1' =_L s_2'$.
  We case on the expression stepping relation to show one of the requirements.
  - Case e-OPERATOR. We have

$$e_1 = f(t_1^1, \ldots, t_1^n) \qquad e_2 = f(t_2^1, \ldots, t_2^n) \qquad e = f(t^1, \ldots, t^n).$$

    If all t^i are values, then $t_1^i = t^i = t_2^i$ and $v_1 = v_2$. Otherwise, let $t^i = x^i$. Inversion on $\Gamma \vdash e : h.\ell$ gives $(x^i : h.\ell') \in \Gamma_2$ for $\ell' \notin L$ and $\ell' \sqsubseteq \ell$, which implies $\ell \notin L$.
  - Case e-DECLASSIFY-REAL. Deliberately excluded; handled by lemma E.9.
  - Case e-DECLASSIFY-SKIP. Same as the case for rule e-OPERATOR.
  - Case e-ENDORSE-REAL. Deliberately excluded; handled by lemma E.10.
  - Case e-ENDORSE-SKIP. Same as the case for rule e-OPERATOR.
  - Case e-INPUT. We have

$$e_1 = e_2 = e = \textbf{input}.$$

    Assume $\ell \in L$ since we are done otherwise. Inversion on $\Gamma \vdash e : h.\ell$ gives $\mathbb{L}(h) \sqsubseteq \ell$, so $\mathbb{L}(h) \in L$, which means $\mathbb{L}(\text{Env}h) = \mathbb{L}(h) \in L$. Thus, $?\text{Env}hv_1 = a_1 = a_2 = ?\text{Env}hv_2$ and $v_1 = v_2$.
  - Case e-INPUT-MALICIOUS. We have $v_1 = 0 = v_2$.
  - Case e-OUTPUT. We have $v_1 = 0 = v_2$.
  - Case e-OUTPUT-MALICIOUS. We have $v_1 = 0 = v_2$.
  - Case e-RECEIVE-REAL. We have

$$e_1 = e_2 = e = \textbf{receive } h'.$$

    We have $\mathbb{L}(h') \notin \mathcal{T}$ by assumption, and $\mathbb{L}(h') \in \mathcal{P}$ by definition A.1.
    If $L = \mathcal{P}$, then $\mathbb{L}(h') \in L$ so $\mathbb{L}(h'h) \in L$. This gives $?h'hv_1 = a_1 = a_2 = ?h'hv_2$, so $v_1 = v_2$.
    If $L = \mathcal{T}$, then inversion on $\Gamma \vdash e : h.\ell$ gives $\mathbb{L}(h')^{\leftarrow} \sqsubseteq \ell$. Since $\mathbb{L}(h') \notin \mathcal{T} = L$, we have $\ell \notin L$.
  - Case e-SEND-REAL. We have $v_1 = 0 = v_2$.
- Case s-COMMUNICATE-REAL. We have

$$s_1 = h_1.v_1 \rightsquigarrow h_2.x; s_1'' \qquad s_2 = h_1.v_2 \rightsquigarrow h_2.x; s_2'' \qquad s = h_1.t \rightsquigarrow h_2.x; s''$$

and

$$s_1 \xrightarrow{!h_1 h_2 v_1}_{\mathrm{r}} s_1''[v_1/x] \qquad\qquad s_2 \xrightarrow{!h_1 h_2 v_2}_{\mathrm{r}} s_2''[v_2/x].$$

By inversion on $\Gamma \vdash s$, we have

$$\frac{\Gamma \vdash t : h_1.\ell \qquad \mathbb{L}(h_2) \Rightarrow \ell \qquad \Gamma, x : h_2.\ell \vdash s''}{\Gamma \vdash h_1.t \rightsquigarrow h_2.x; s''}\ \text{RULE } \ell\text{-COMMUNICATE}$$

We case on t. If $t = v$ for some v, then $v_1 = \sigma_1(v) = v = \sigma_2(v) = v_2$. Additionally, $\Gamma \vdash s''[v/x]$ by lemma C.5, $\sigma_1(s''[v/x]) = s_1''[v/x]$, and $\sigma_2(s''[v/x]) = s_2''[v/x]$, so $s_1''[v_1/x] =_L s_2''[v_2/x]$.
Otherwise, $t = x'$ for some $x' \in \mathrm{dom}(\Gamma_2)$. By inversion on $\Gamma \vdash t : h_1.\ell$, we have $(x : h_1.\ell') \in \Gamma_2$ for some $\ell' \sqsubseteq \ell$. Since $\ell' \notin L$ and $\ell' \sqsubseteq \ell$, we have $\ell \notin L$. Define $\Gamma_2' = (\Gamma_2, x : h_2.\ell)$, $\sigma_1' = \sigma_1 \cup \{x \mapsto v_1\}$, $\sigma_2' = \sigma_2 \cup \{x \mapsto v_2\}$, which ensures $\sigma_1'(s'') = s_1''[v_1/x]$ and $\sigma_2'(s'') = s_2''[v_2/x]$. Note that Γ_2 satisfies the requirements of definition E.3, and $\sigma_1', \sigma_2' : \Gamma_2'$, so $s_1''[v_1/x] =_L s_2''[v_2/x]$.

- Case s-SELECT-REAL. We have

$$s_1 = h_1[v_1] \rightsquigarrow h_2; s_1' \qquad s_2 = h_1[v_2] \rightsquigarrow h_2; s_2' \qquad s = h_1[v] \rightsquigarrow h_2; s'$$

and

$$s_1 \xrightarrow{!h_1 h_2 v_1}_{\mathrm{r}} s_1' \qquad\qquad s_2 \xrightarrow{!h_1 h_2 v_2}_{\mathrm{r}} s_2'.$$

$\Gamma \vdash s'$ (by inversion on $\Gamma \vdash s$), $\sigma_1(s') = s_1'$, and $\sigma_2(s') = s_2'$, thus $s_1' =_L s_2'$.

- Case s-IF. We have

$$s_1 = \mathbf{if}(h.v_1, s_1^1, s_1^2) \qquad s_2 = \mathbf{if}(h.v_2, s_2^1, s_2^2) \qquad s = \mathbf{if}(h.t, s^1, s^2)$$

and

$$s_1 \xrightarrow{!hh0}_{\mathrm{r}} s_1^i \qquad\qquad s_2 \xrightarrow{!hh0}_{\mathrm{r}} s_2^j.$$

Inversion on $\Gamma \vdash s$ (which must be by rule ℓ-IF) gives $\Gamma \vdash t : h.\mathbf{0}^{\leftarrow}$. Additionally, $\sigma_1(t)$ and $\sigma_2(t)$ are values, so $\mathrm{free}(t) \subseteq \Gamma_2$, meaning $\Gamma_2 \vdash t : h.\mathbf{0}^{\leftarrow}$. Since $\mathbf{0}^{\leftarrow} \in L$ for all attacks (recall definition A.1), and Γ_2 only contains variables with labels not in L, t must be a value, that is, $t = v$ for some v. Then, $v_1 = \sigma_1(v) = v = \sigma_2(v) = v_2$, so $i = j$. Finally, we have $s_1^i =_L s_2^j$ since $\Gamma \vdash s_i$ (by inversion on $\Gamma \vdash s$), $\sigma_1(s^i) = s_1^i$, and $\sigma_2(s^i) = s_2^i = s_2^j$.

- Case s-CASE. Impossible by inversion on $\Gamma \vdash s$.
- Case s-SEQUENTIAL. Immediate by the induction hypothesis.
- Case s-DELAY. We have

$$s_1 = E_1[s_1''] \qquad s_2 = E_2[s_2''] \qquad s = E[s'']$$

and

$$\frac{s_1'' \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{r}} s_1''' \qquad \mathrm{actor}(a) \notin \mathrm{hosts}(E_1)}{s_1 \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{r}} E_1[s_1''']} \qquad\qquad \frac{s_2'' \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{r}} s_2''' \qquad \mathrm{actor}(a) \notin \mathrm{hosts}(E_1)}{s_2 \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{r}} E_2[s_2''']}$$

Note that $(\Gamma_1, \Gamma_1', \Gamma_2) \vdash s''$ where Γ_1' are the variables defined by E (which must be the same as the ones defined by E_1 and E_2). Thus, $s_1'' =_L s_2''$ through s'', σ_1, and σ_2, and we can apply induction hypothesis to get $s_1''' =_L s_2'''$. This then gives $s_1' = E_1[s_1'''] =_L E_2[s_2''']s_2'$.

- Case s-IF-DELAY. Using the induction hypotheses similar to rule s-DELAY. ☐

Public-equivalent choreographies produce public-equivalent outputs.

Lemma E.5 (Public Outputs)**.** *If* $s_1 =_{\mathcal{P}} s_2$, $s_1 \xrightarrow{!m_1}{}^{\mathrm{c}}_{\mathrm{r}}$, *and* $s_2 \xrightarrow{a_2}{}^{\mathrm{c}}_{\mathrm{r}}$ *with* $\mathrm{actor}(!m_1) = \mathrm{actor}(a_2)$, *then* $!m_1 =_{\mathcal{P}} a_2$.

Proof. By definition E.3, there exists s such that $\Gamma \vdash s$, $\sigma_1(s) = s_1$, and $\sigma_2(s) = s_2$ for some $\Gamma = (\Gamma_1, \Gamma_2)$ and $\sigma_1, \sigma_2 : \Gamma_2$. We proceed by induction on the two stepping judgments, which must be by the same rule since s_1 and s_2 have the same structure.

Cases for rules s-DELAY and s-IF-DELAY follow from the induction hypotheses. Cases for rules e-OPERATOR, e-DECLASSIFY-SKIP, e-ENDORSE-SKIP, e-OUTPUT-MALICIOUS, s-IF, e-DECLASSIFY-REAL and e-ENDORSE-REAL are immediate since both actions are internal, i.e., $!m_1 = !hh0 = a_2$ for some h, which implies $!m_1 =_{\mathcal{P}} a_2$. We detail the remaining cases.

- Case s-LET. We have

$$s_1 = \textbf{let } h.x = e_1; s_1' \qquad s_2 = \textbf{let } h.x = e_2; s_2' \qquad s = \textbf{let } h.x = e; s'$$

and

$$h.e_1 \xrightarrow{!hcv_1}_{\mathrm{r}} \qquad h.e_2 \xrightarrow{!hcv_2}_{\mathrm{sim}} .$$

Inversion on $\Gamma \vdash s$ gives $\Gamma \vdash e : h.\ell$ for $\mathbb{L}(h) \Rightarrow \ell$. We case on the expression stepping relations.

  - Case e-OUTPUT. We have $c = \mathrm{Env}$, $\mathbb{L}(h) \in \mathcal{T}$, and

$$e_1 = \textbf{output } v_1 \qquad e_2 = \textbf{output } v_2 \qquad e = \textbf{output } t.$$

If $\mathbb{L}(h) \notin \mathcal{P}$, then $!m_1 = !h\mathrm{Env}v_1 =_{\mathcal{P}} !h\mathrm{Env}v_2 = a_2$ immediately, so assume $\mathbb{L}(h) \in \mathcal{P}$. Inversion on $\Gamma \vdash e : h.\ell$ gives $\Gamma \vdash t : h.\mathbb{L}(h)$. Since $\mathbb{L}(h) \in \mathcal{P}$, $t = v$ for some v, meaning $v_1 = \sigma_1(t) = v = \sigma_2(t) = v_2$, so $!m_1 = !h\mathrm{Env}v = !h\mathrm{Env}v = a_2$, and $!m_1 =_{\mathcal{P}} a_2$.

  - Case e-SEND-REAL. We have $c = h'$ and

$$e_1 = \textbf{send } v_1 \textbf{ to } h' \qquad e_2 = \textbf{send } v_2 \textbf{ to } h' \qquad e = \textbf{send } t \textbf{ to } h'.$$

If $t = v$ for some v, then $v_1 = v_2$ and we are done, so assume $t = x'$ for some x'. Inversion on $\Gamma \vdash e : h.\ell$ gives $\Gamma \vdash t : h.\mathbb{L}(h')^{\rightarrow}$. We then have $(x' : \ell'.h) \in \Gamma_2$ with $\mathbb{L}(h) \Rightarrow \ell'$, $\ell' \notin \mathcal{P}$, and $\ell' \sqsubseteq \mathbb{L}(h')^{\rightarrow}$. Then,

$$\ell' \notin \mathcal{P} \wedge \mathbb{L}(h) \Rightarrow \ell' \implies \mathbb{L}(h) \notin \mathcal{P}$$
$$\ell' \sqsubseteq \mathbb{L}(h')^{\rightarrow} \wedge \ell' \notin \mathcal{P} \implies \mathbb{L}(h')^{\rightarrow} \notin \mathcal{P} \implies \mathbb{L}(h') \notin \mathcal{P}.$$

Thus, $\mathbb{L}(hh') \notin \mathcal{P}$ and $!m_1 = !hh'v_1 =_{\mathcal{P}} !hh'v_2 = a_2$.

- Case s-COMMUNICATE-REAL. We have

$$s_1 = h_1.v_1 \rightsquigarrow h_2.x; s_1'' \qquad s_2 = h_1.v_2 \rightsquigarrow h_2.x; s_2'' \qquad s = h_1.t \rightsquigarrow h_2.x; s''$$

and

$$s_1 \xrightarrow{!h_1h_2v_1}_{\mathrm{r}} \qquad s_2 \xrightarrow{!h_1h_2v_2}_{\mathrm{sim}} .$$

By inversion on $\Gamma \vdash s$, we have

RULE ℓ-COMMUNICATE
$$\frac{\Gamma \vdash t : h_1.\ell \qquad \mathbb{L}(h_2) \Rightarrow \ell \qquad \Gamma, x : h_2.\ell \vdash s''}{\Gamma \vdash h_1.t \rightsquigarrow h_2.x; s''}$$

We case on t. If $t = v$ for some v, then $v_1 = \sigma_1(v) = v = \sigma_2(v) = v_2$. So $a_1 = !h_1h_2v = a_2$ and $a_1 =_{\mathcal{P}} a_2$. Otherwise, $t = x'$ for some $x' \in \mathrm{dom}(\Gamma_2)$. By inversion on $\Gamma \vdash t : h_1.\ell$, we have $(x : h_1.\ell') \in \Gamma_2$ for some $\ell' \sqsubseteq \ell$. Then,

$$\ell' \notin \mathcal{P} \wedge \ell' \sqsubseteq \ell \implies \ell \notin \mathcal{P}$$
$$\ell' \notin \mathcal{P} \wedge \mathbb{L}(h_1) \Rightarrow \ell' \implies \mathbb{L}(h_1) \notin \mathcal{P}$$
$$\ell \notin \mathcal{P} \wedge \mathbb{L}(h_2) \Rightarrow \ell \implies \mathbb{L}(h_2) \notin \mathcal{P}$$
$$\mathbb{L}(h_1) \notin \mathcal{P} \wedge \mathbb{L}(h_2) \notin \mathcal{P} \implies \mathbb{L}(h_1h_2) \notin \mathcal{P}.$$

Since $\mathbb{L}(h_1h_2) \notin \mathcal{P}$, $a_1 = !h_1h_2v_1 =_{\mathcal{P}} !h_1h_2v_2 = a_2$.

- Case s-SELECT-REAL. We have

$$s_1 = h_1[v_1] \rightsquigarrow h_2; s_1' \qquad s_2 = h_1[v_2] \rightsquigarrow h_2; s_2' \qquad s = h_1[v] \rightsquigarrow h_2; s'$$

and

$$s_1 \xrightarrow{!h_1h_2v_1}_{\mathrm{r}} \qquad s_2 \xrightarrow{!h_1h_2v_2}_{\mathrm{sim}} .$$

Note that selection statements do not allow variables to be communicated, so s sending v (rather than t) is not a mistake. Thus, we have $v_1 = \sigma_1(v) = v = \sigma_2(v) = v_2$, which means $a_1 = !h_1h_2v = a_2$, which in turn means $a_1 =_{\mathcal{P}} a_2$. ☐

Trusted-equivalent choreographies produce trusted-equivalent outputs *for the environment*. The statement does not apply to intermediate messages between hosts because untrusted values can be sent on trusted channels (e.g., a trusted third party can process untrusted values from other hosts).

Lemma E.6 (Trusted Outputs)**.** *If* $s_1 =_{\mathcal{P}} s_2$*,* $s_1 \xrightarrow{!h\mathrm{Env}v_1}{}^{\mathrm{c}}_{\mathrm{r}}$*, and* $s_2 \xrightarrow{a_2}{}^{\mathrm{c}}_{\mathrm{r}}$ *with* $\mathrm{actor}(a_2) = h$*, then* $!h\mathrm{Env}v_1 =_{\mathcal{P}} a_2$*.*

Proof. By induction on the stepping relations. Inductive cases are handled similarly to lemma E.5. The only other relevant case is under rule s-LET with rule e-OUTPUT. The argument is similar to the case in lemma E.5, but holds because only trusted values can be **output** to nonmalicious hosts. ☐

The simulator's view of the real choreography stays accurate on public values.

Lemma E.7 (Matching Steps for Public Equivalence). *Assume $s_1 =_{\mathcal{P}} s_2$ and $s_1 \xrightarrow{a_1}{}^{\mathrm{c}}_{\mathrm{r}} s_1'$ without using rule e-DECLASSIFY-REAL or e-ENDORSE-REAL.*

- *If $a_1 = ?m_1$, then $s_2 \xrightarrow{?m_2}{}^{\mathrm{c}}_{\mathrm{sim}} s_2'$ with $s_1' =_{\mathcal{P}} s_2'$ for* all *$m_2 =_{\mathcal{P}} m_1$.*
- *If $a_1 = !m_1$, then $s_2 \xrightarrow{!m_2}{}^{\mathrm{c}}_{\mathrm{sim}} s_2'$ with $s_1' =_{\mathcal{P}} s_2'$ for* some *$m_2 =_{\mathcal{P}} m_1$.*

In addition, the statement holds with the roles of $\rightarrow^{\mathrm{c}}_{\mathrm{r}}$ and $\rightarrow^{\mathrm{c}}_{\mathrm{sim}}$ reversed (excluding rules e-DECLASSIFY-SIMULATOR and e-ENDORSE-SIMULATOR instead).

Proof. Follows immediately from lemmas E.4 and E.5 since $\rightarrow^{\mathrm{c}}_{\mathrm{sim}}$ is equivalent to $\rightarrow^{\mathrm{c}}_{\mathrm{r}}$ except for rules e-DECLASSIFY-REAL and e-ENDORSE-REAL, which we exclude. ☐

The ideal choreography stays accurate to the real choreography on trusted values.

Lemma E.8 (Matching Steps for Trusted Equivalence). *Assume $s_1 =_{\mathcal{T}} s_2$ and $s_1 \xrightarrow{a_1}{}^{\mathrm{c}}_{\mathrm{r}} s_1'$ without using rule e-DECLASSIFY-REAL or e-ENDORSE-REAL.*

- *If $a_1 = ?\mathrm{Env}hv_1$, then $s_2 \xrightarrow{a_2}{}^{\mathrm{c}}_{\mathrm{i}} s_2'$ with $s_1' =_{\mathcal{T}} s_2'$ for* all *$a_2 =_{\mathcal{T}} a_1$.*
- *If $a_1 = !h\mathrm{Env}v_1$, then $s_2 \xrightarrow{a_2}{}^{\mathrm{c}}_{\mathrm{i}} s_2'$ with $s_1' =_{\mathcal{T}} s_2'$ for* some *$a_2 =_{\mathcal{T}} a_1$.*
- *Otherwise, $s_2 \xrightarrow{!hh0}{}^{\mathrm{c}}_{\mathrm{i}} s_2'$ with $s_1' =_{\mathcal{T}} s_2'$ and $\mathrm{actor}(a_1) = h$.*

In addition, the statement holds with the roles of $\rightarrow^{\mathrm{c}}_{\mathrm{r}}$ and $\rightarrow^{\mathrm{c}}_{\mathrm{i}}$ reversed (excluding rules e-DECLASSIFY and e-ENDORSE instead).

Proof. Follows from lemmas E.4 and E.6. judgment $\rightarrow^{\mathrm{c}}_{\mathrm{r}}$ behaves the same as $\rightarrow^{\mathrm{c}}_{\mathrm{r}}$ except it replaces some output messages with internal steps. Since this does not affect the resulting choreographies (only the actions), lemma E.4 applies and shows that the resulting choreographies are equivalent. The one exception to this rules e-RECEIVE and e-RECEIVE-REAL, where the ideal choreography proceeds with 0 instead of receiving a value; this value is treated as untrusted so the choreographies still agree on trusted values as required. ☐

Assume the simulator's view agrees with the real choreography on public values, and the ideal choreography agrees with the real choreography on trusted values. If the ideal choreography declassifies a value and we feed that value to the simulator, then all three choreographies remain in agreement. Only trusted values are declassified, so the ideal choreography outputs the correct value to the simulator.

Lemma E.9 (Equivalence After Declassify). *Let $s_1 =_{\mathcal{P}} s_2$ and $s_1 =_{\mathcal{T}} s_3$. If $s_1 \xrightarrow{!hh0}{}^{\mathrm{c}}_{\mathrm{r}} s_1'$, $s_2 \xrightarrow{?h\mathrm{Adv}v}{}^{\mathrm{c}}_{\mathrm{sim}} s_2'$, and $s_3 \xrightarrow{!h\mathrm{Adv}v}{}^{\mathrm{c}}_{\mathrm{i}} s_3'$, then $s_1' =_{\mathcal{P}} s_2'$ and $s_1' =_{\mathcal{T}} s_3'$.*

Proof. By definition E.3, there exist s_{p} and s_{t} such that $\Gamma_{\mathrm{p}} \vdash s_{\mathrm{p}}$, $\sigma_1(s_{\mathrm{p}}) = s_1$, $\sigma_2(s_{\mathrm{p}}) = s_2$, and $\Gamma_{\mathrm{t}} \vdash s_{\mathrm{t}}$, $\sigma_2'(s_{\mathrm{t}}) = s_2$, $\sigma_3(s_{\mathrm{t}}) = s_3$ for $\Gamma_{\mathrm{p}} = (\Gamma_1, \Gamma_2)$, $\sigma_1, \sigma_2 : \Gamma_2$, $\Gamma_{\mathrm{t}} = (\Gamma_3, \Gamma_4)$, and $\sigma_2', \sigma_3 : \Gamma_4$.

We proceed by induction on the stepping relations. Inductive cases (rules s-DELAY and s-IF-DELAY) are handled similarly to lemma E.7. The only remaining case is when the steps are by rules e-DECLASSIFY, e-DECLASSIFY-REAL and e-DECLASSIFY-SIMULATOR, respectively. We have

$$s_1 = \mathbf{let}\ h.x = \mathbf{declassify}(v_1, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}); s_1'' \qquad s_{\mathrm{p}} = \mathbf{let}\ h.x = \mathbf{declassify}(t_{\mathrm{p}}, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}); s_{\mathrm{p}}''$$

$$s_2 = \mathbf{let}\ h.x = \mathbf{declassify}(v_2, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}); s_2'' \qquad s_{\mathrm{t}} = \mathbf{let}\ h.x = \mathbf{declassify}(t_{\mathrm{t}}, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}); s_{\mathrm{t}}''$$

$$s_3 = \mathbf{let}\ h.x = \mathbf{declassify}(v, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}); s_3''$$

where $\ell_{\mathrm{f}} \notin \mathcal{P}$ and $\ell_{\mathrm{t}} \in \mathcal{P}$, and

$$s_1' = s_1''[v_1/x] \qquad s_2' = s_2''[v/x] \qquad s_3' = s_3''[v/x].$$

We claim $v_1 = v$ (v_2 is ignored by s_2, so it is irrelevant). By lemma C.1 and inversion on $\Gamma_{\mathrm{t}} \vdash s_{\mathrm{t}}$, we have $\ell_{\mathrm{f}} \in \mathcal{T}$. Assume for contradiction that $t_{\mathrm{t}} = x_{\mathrm{t}}$ for some x_{t}. Then, $(x_{\mathrm{t}} : h.\ell) \in \Gamma_4$ for $\ell \sqsubseteq \ell_{\mathrm{f}}$. However, $\ell \notin \mathcal{T}$ so $\ell_{\mathrm{f}} \notin \mathcal{T}$, which is a contradiction. Thus, $t_{\mathrm{t}} = v_{\mathrm{t}}$ for some v_{t}. Then, $v_1 = \sigma_2'(t_{\mathrm{t}}) = \sigma_2'(v_{\mathrm{t}}) = v_{\mathrm{t}} = \sigma_3(v_{\mathrm{t}}) = \sigma_3(t_{\mathrm{t}}) = v$.

Finally, let $s_{\mathrm{p}}' = s_{\mathrm{p}}''[v/x]$ and $s_{\mathrm{t}}' = s_{\mathrm{t}}''[v/x]$. We have $s_1' =_{\mathcal{P}} s_2'$ since $\Gamma_{\mathrm{p}} \vdash s_{\mathrm{p}}'$ (inversion on $\Gamma_{\mathrm{p}} \vdash s_{\mathrm{p}}$ followed by lemma C.5), $\sigma_1(s_{\mathrm{p}}') = s_1'$, and $\sigma_2(s_{\mathrm{p}}') = s_2'$; and we have $s_2' =_{\mathcal{T}} s_3'$ since $\Gamma_{\mathrm{t}} \vdash s_{\mathrm{t}}'$ (inversion on $\Gamma_{\mathrm{t}} \vdash s_{\mathrm{t}}$, then lemma C.5), $\sigma_2'(s_{\mathrm{t}}') = s_2'$, and $\sigma_3(s_{\mathrm{t}}') = s_3'$. ☐

Similarly, if the simulator recreates a value that the ideal choreography endorses, all three choreographies remain in agreement. Only public values are endorsed, so the simulator outputs the correct value to the ideal choreography.

Lemma E.10 (Equivalence After Endorse)**.** *Let* $s_1 =_{\mathcal{P}} s_2$ *and* $s_1 =_{\mathcal{T}} s_3$*. If* $s_1 \xrightarrow{!hh0}{}^{c}_{r} s_1'$, $s_2 \xrightarrow{!\mathrm{Adv}hv}{}^{c}_{\mathrm{sim}} s_2'$, *and* $s_3 \xrightarrow{?\mathrm{Adv}hv}{}^{c}_{i} s_3'$, *then* $s_1' =_{\mathcal{P}} s_2'$ *and* $s_1' =_{\mathcal{T}} s_3'$.

Proof. Dual to lemma E.9, but focusing on $s_1 =_{\mathcal{P}} s_2$ and using lemma C.2. ☐

Lemmas E.7 and E.8 straightforwardly lift from choreographies s to processes w. Lemma E.7 needs an additional condition on buffer equivalence: for $B_1 =_{\mathcal{P}} B_2$, we require $|B_1(c_1c_2)| = |B_2(c_1c_2)|$ when $\mathbb{L}(c_1c_2) \notin \mathcal{P}$. That is, the buffers must agree exactly on public channels, and agree on the number of messages on secret channels. This condition allows the simulator to keep track of messages on secret channels even though it cannot read message contents.

Proof sketch for theorem VI.6. We prove simulation as follows.

Simulator The simulator has the form $\mathcal{S}(\mathcal{A} \parallel w)$ where w is a public view of the real process. The simulator runs w against $\mathcal{A}$ for all internal messages. The simulator forwards inputs from Env to $\mathcal{A}$ and w, and forwards messages from $\mathcal{A}$ destined for Env to Env. When the ideal process outputs data through a **declassify** expression, the simulator inputs this data to w. Similarly, when w outputs data through an **endorse** expressions, the simulator forwards this data to the ideal process.[7]

Bisimulation Relation We maintain the invariant that the simulator's version of the process matches the real one on *public values*, and the ideal process matches the real one on *trusted values*. More concretely, we define $\mathcal{A} \parallel w_1 \; R \; \mathcal{S}(\mathcal{A}' \parallel w) \parallel w_2$ if: (1) $\mathcal{A} = \mathcal{A}'$, (2) $w_1 =_{\mathcal{P}} w$, and (3) $w_1 =_{\mathcal{T}} w_2$.

Simulation We claim R is a weak bisimulation.

Since the simulator's version of the process matches the real one on public values (condition (2)), the adversary in the real configuration has a view identical to the adversary running inside of the simulator (the adversary only sees public data). Similarly, since the real process matches the ideal one on trusted values, the environment has the same view in both (the environment is only sent trusted data).

Condition (1) is preserved since w is an accurate public view of w_1 (condition (2)). When there are no downgrade actions, lemma E.7 ensures condition (2) is preserved, and lemma E.8 ensures condition (3) is preserved. Lemmas E.9 and E.10 cover the cases with downgrades. ☐

F Details for Section VII (Endpoint Projection)

Figure 17 formalizes projecting onto a host h. Projection keeps **let** statements assigned to h, and removes ones assigned to other hosts. Communication statements become a **send** or a **receive**, or are entirely removed depending on whether h is the sending host, the receiving host, or neither. Selection statements follow the same logic, but are projected as a **send** expression or a **case** statement with a single branch.

The most interesting case is **if** statements. If the **if** statement is placed at h, we perform the **if** as usual and project the branches. Otherwise, h does not store the conditional and cannot determine which branch should be taken. In this case, the projections of the two branches must be compatible, formalized by a merge function. Merging requires the two branches to have the same syntactic structure, but allows **case** statements to have disjoint branches, which are combined into one. We elide most cases of the merge function, since the proof is agnostic to the details.

We lift projection to processes: projecting a buffer onto h keeps only messages destined for h, and projecting processes is done componentwise. The *projection* $[\![w]\!]$ of process w is the configuration formed by projecting onto each host in w:

$$[\![w = \langle H, \_, \_\rangle]\!] = \Big\|_{h \in H} [\![w]\!]_h.$$

G Details for Section VII-C (Correctness of Endpoint Projection)

A choreography and its endpoint projection match each other action-for-action; once we prove this fact, showing simulation is trivial since we can pick $\mathcal{S} = \mathcal{A}$. The choreographic programming literature [28, 35, 36, 43, 44] extensively studies this perfect correspondence between a choreography and its projection, and formalizes the correspondence as strong bisimulation.

To prove that a choreography w is bisimilar to its endpoint projection $[\![w]\!]$, we must define a relation R between an arbitrary configuration and process, $W_1 \; R \; w_2$, and show that R is a bisimulation. The obvious approach is to define $W_1 \; R \; w_2$ if $W_1 = [\![w_2]\!]$, but this idea fails because R is not preserved under stepping.

Lemma G.1. *Define* $W_1 \; R \; w_2$ *if* $W_1 = [\![w_2]\!]$*. We claim* R *is* not *a bisimulation.*

[7] The simulator needs to step the ideal process an additional time so that the ideal process pulls the message from its buffer. This is due to how we define operational rules for processes. This extra step forces us to use weak bisimulation instead of strong bisimulation.

$\boxed{[\![s]\!]_h = s'}$

$$[\![\textbf{let } h'.x = e; s]\!]_h = \begin{cases} \textbf{let } h'.x = e; [\![s]\!]_h & h' = h \\ [\![s]\!]_h & \text{o/w} \end{cases}$$

$$[\![h_1.t \rightsquigarrow h_2.x; s]\!]_h = \begin{cases} \textbf{let } h_1.\_ = \textbf{send } t \textbf{ to } h_2; [\![s]\!]_h & h_1 = h \\ \textbf{let } h_2.x = \textbf{receive } h_1; [\![s]\!]_h & h_2 = h \\ [\![s]\!]_h & \text{o/w} \end{cases}$$

$$[\![h_1[v] \rightsquigarrow h_2; s]\!]_h = \begin{cases} \textbf{let } h_1.\_ = \textbf{send } v \textbf{ to } h_2; [\![s]\!]_h & h_1 = h \\ \textbf{case } (h_1 \rightsquigarrow h_2)\ \{v \mapsto [\![s]\!]_h\} & h_2 = h \\ [\![s]\!]_h & \text{o/w} \end{cases}$$

$$[\![\textbf{if}(h'.t, s_1, s_2)]\!]_h = \begin{cases} \textbf{if}(h'.t, [\![s_1]\!]_h, [\![s_2]\!]_h) & h' = h \\ \text{merge}([\![s_1]\!]_h, [\![s_2]\!]_h) & \text{o/w} \end{cases}$$

$$[\![\textbf{skip}]\!]_h = \textbf{skip}$$

$\boxed{\text{merge}(s_1, s_2) = s}$

$$\begin{aligned} \text{merge}(s_1, s_2) &= \textbf{case } (h_1 \rightsquigarrow h_2)\ \{v \mapsto s_v\}_{v \in V_1 \cup V_2} \\ &\quad \text{where } s_1 = \textbf{case } (h_1 \rightsquigarrow h_2)\ \{v \mapsto s_v\}_{v \in V_1} \\ &\quad\quad\quad\ \ s_2 = \textbf{case } (h_1 \rightsquigarrow h_2)\ \{v \mapsto s_v\}_{v \in V_2} \\ &\quad\quad\quad\ \ V_1 \text{ and } V_2 \text{ disjoint} \\ \text{merge}(s_1, s_2) &= \textbf{let } h.x = e; \text{merge}(s_1', s_2') \\ &\quad \text{where } s_1 = \textbf{let } h.x = e; s_1' \\ &\quad\quad\quad\ \ s_2 = \textbf{let } h.x = e; s_2' \end{aligned}$$

$\boxed{[\![B]\!]_h = B'}$ $\boxed{[\![w]\!]_h = w'}$

$$[\![B]\!]_h(c_1 c_2) = \begin{cases} B(c_1 c_2) & c_1 \neq h \wedge c_2 = h \\ \epsilon & \text{otherwise} \end{cases}$$

$$[\![\langle H, B, s\rangle]\!]_h = \langle H \cap \{h\}, [\![B]\!]_h, [\![s]\!]_h\rangle$$

Figure 17. Endpoint projection: statements, buffers, processes.

Proof. Consider the following choreography and its projection:

// Choreography
$w_2 = \textbf{if}(\text{Alice}.1, \text{Alice}[\text{Bob}] \rightsquigarrow 1; s_1, \text{Alice}[\text{Bob}] \rightsquigarrow 0; s_2)$
// Alice
$[\![w_2]\!]_{\text{Alice}} = \textbf{if}(\text{Alice}.1, \textbf{send } 1 \textbf{ to } \text{Bob}; [\![s_1]\!]_{\text{Alice}}, \textbf{send } 0 \textbf{ to } \text{Bob}; [\![s_2]\!]_{\text{Alice}})$
// Bob
$[\![w_2]\!]_{\text{Bob}} = \textbf{case } (\text{Alice} \rightsquigarrow \text{Bob})\ \{1 \mapsto [\![s_1]\!]_{\text{Bob}}, 0 \mapsto [\![s_2]\!]_{\text{Bob}}\}$

Let $W_1 = [\![w_2]\!]$; we have $W_1\ R\ w_2$. Now, host Alice can reduce the **if** statement with an internal step in both W_1 and w_2, which gives:

// Choreography
$w_2' = \text{Alice}[\text{Bob}] \rightsquigarrow 1; s_1$
// Alice
$W_1'(\text{Alice}) = \textbf{send } 1 \textbf{ to } \text{Bob}; [\![s_1]\!]_{\text{Alice}}$
// Bob
$W_1'(\text{Bob}) = \textbf{case } (\text{Alice} \rightsquigarrow \text{Bob})\ \{1 \mapsto [\![s_1]\!]_{\text{Bob}}, 0 \mapsto [\![s_2]\!]_{\text{Bob}}\}$

$$\begin{array}{llll} \text{Statements} & s & ::= & \dots \\ & & | & h_1 \rightsquigarrow h_2[v/x]; s \\ & & | & h_1 \rightsquigarrow h_2[v]; s \end{array}$$

Figure 18. The syntax of asynchronous choreographies (extends fig. 6).

$\boxed{s \xrightarrow{a}_{\mathrm{a}} s'}$

s-Communicate-Send
$$h_1.v \rightsquigarrow h_2.x; s \xrightarrow{!h_1 h_2 v}_{\mathrm{a}} h_1 \rightsquigarrow h_2[v/x]; s$$

s-Communicate-Receive
$$h_1 \rightsquigarrow h_2[v/x]; s \xrightarrow{!h_2 h_2 0}_{\mathrm{a}} s[v/x]$$

s-Select-Send
$$h_1[v] \rightsquigarrow h_2; s \xrightarrow{!h_1 h_2 v}_{\mathrm{a}} h_1 \rightsquigarrow h_2[v]; s$$

s-Select-Receive
$$h_1 \rightsquigarrow h_2[v]; s \xrightarrow{!h_2 h_2 0}_{\mathrm{a}} s$$

Figure 19. Stepping rules for asynchronous choreographies. These override the rules for $\longrightarrow_{\mathrm{r}}$ in fig. 14b.

Note that the process for Bob in W_1' does not match $[\![w_2']\!]_{\text{Bob}}$, which is

$$[\![w_2']\!]_{\text{Bob}} = \textbf{case}\ (\text{Alice} \rightsquigarrow \text{Bob})\ \{1 \mapsto [\![s_1]\!]_{\text{Bob}}\}$$

(there is no case for 0).

Thus, we have $W_1\ R\ w_2$, $W_1 \xrightarrow{!\text{AliceAlice}0} W_1'$, $w_2 \xrightarrow{!\text{AliceAlice}0} w_2'$, but it is not the case that $W_1'\ R\ w_2'$. Lemma C.12 implies W_1' is uniquely determined, so there is no other W_1'' related to w_2' that W_1 can step to. Therefore, R is not a bisimulation. ☐

Intuitively, when a choreography reduces an **if** statement, the branch that is not taken disappears in one step for all hosts. However, in the projected program, each host reduces its corresponding **case** statement separately, which results in extraneous dead branches during simulation. This is a known issue in the choreography literature [28], and it does *not* break bisimilarity, but we need to be smarter about how to define R.

The solution to the issue raised by lemma G.1 is to ignore extraneous branches when defining R. Even though the configuration W_1' has "leftover" branches that projecting the choreography w_2' does not create, we know that these branches will never be taken. So we can ignore these branches when defining R.

Following Montesi [28], we define $w_1 \succeq w_2$ if w_1 and w_2 are structurally identical, except w_1 has at least as many branches in **case** statements as w_2. We lift $\succeq$ pointwise to configurations. Now, we define $W_1\ R\ w_2$ if $W_1 \succeq [\![w_2]\!]$. We claim R is a bisimulation. Further, the proof is split into showing the *soundness* and *completeness* of endpoint projection.

Lemma G.2 (Soundness of Endpoint Projection)**.** *If $W \succeq [\![w]\!]$ and $w \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{r}} w'$, then $W \xrightarrow{a}_{\mathrm{r}} W'$ for some $W' \succeq [\![w']\!]$.*

Lemma G.3 (Completeness of Endpoint Projection)**.** *If $W \succeq [\![w]\!]$ and $W \xrightarrow{a}_{\mathrm{r}} W'$, then $w \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{r}} w'$, for some w' with $W' \succeq [\![w']\!]$.*

A. Handling Asynchronous Communication

We follow prior work [53, 60] and add syntactic forms to choreographies to represent partially reduced **send/receive** pairs given in fig. 18. We extend endpoint projection so that the new syntactic forms are projected as a **receive** statement and a message on the receiver's buffer. For example, while $\text{Alice}.v \rightsquigarrow \text{Bob}.x; s$ becomes a **send** on Alice and a **receive** on Bob, $\text{Alice} \rightsquigarrow \text{Bob}[v/x]; s$ becomes a **receive** on Bob and a message (from Alice) in Bob's buffer. We update the stepping rules for communication and selection statements so that they reduce to the corresponding run-time terms, which in turn reduce to their continuations. Figure 19 gives the updated rules.

These run-time terms are sufficient to restore perfect correspondence, and make lemmas G.2 and G.3 go through. We refer to Cruz-Filipe and Montesi [53] for details.

Lemma G.4. *If $\epsilon \vdash w$, then $\langle (\!| w |\!), \rightarrow^{\mathrm{c}}_{\mathrm{a}} \rangle \geq \langle (\!| [\![w]\!] |\!), \rightarrow_{\mathrm{r}} \rangle$.*

Proof. Let $\mathcal{S} = \mathcal{A}$. Lemmas G.2 and G.3 immediately give a strong bisimulation. ☐

B. Restoring Original Choreography Syntax

Lemma G.4 proves the correctness of endpoint projection for *extended* choreographies that have run-time terms. Next, we show a simple simulation that a choreography with run-time terms simulates one without, removing the need to reason about run-time terms in other proof steps.

Lemma G.5. *If $\epsilon \vdash w$, then $\langle (\!| w |\!), \rightarrow^{\mathrm{c}}_{\mathrm{r}} \rangle \geq \langle (\!| w |\!), \rightarrow^{\mathrm{c}}_{\mathrm{a}} \rangle$.*

Proof. The simulator follows the control flow by maintaining a public view of the extended choreography $\langle (\!|w|\!), \longrightarrow^{c}_{a} \rangle$ and runs the adversary against this view. The simulator behaves the same as the adversary, except when the adversary schedules a run-time term, the simulator takes an internal step (and does not schedule the original choreography). □

Proof of theorem VII.1. By lemmas G.4 and G.5 using the transitivity of UC simulation. □