PROVABLY CORRECT COMPILATION FOR DISTRIBUTED CRYPTOGRAPHIC APPLICATIONS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Coşku Acay

August 2023

PROVABLY CORRECT COMPILATION FOR DISTRIBUTED CRYPTOGRAPHIC APPLICATIONS

Coşku Acay, Ph.D.

Cornell University 2023

Developing secure distributed systems is difficult, and even harder when advanced cryptography must be used to achieve security goals. We present Viaduct, a compiler that transforms high-level programs into secure, efficient distributed realizations. Instead of implementing a system of communicating processes, the Viaduct programmer implements a *centralized, sequential* program which is automatically compiled into a secure distributed version that uses cryptography. Viaduct programs specify security policies declaratively using information-flow labels, and need not mention cryptographic primitives.

Unlike prior compilers for cryptographic libraries, Viaduct is general and extensible: it can efficiently and automatically combine local computation with multiple advanced cryptographic primitives such as commitments, zero-knowledge proofs, secure multiparty computation, and fully homomorphic encryption.

We develop a modular security proof for Viaduct that abstracts away from the details of cryptographic mechanisms. Our proof relies on a novel unification of simulation-based security, information-flow control, choreographic programming, and sequentialization techniques for concurrent programs. To our knowledge, this is the first security proof that simultaneously addresses subtleties essential for robust applications, such as multiple cryptographic mechanisms, malicious corruption, and asynchronous communication. Our approach offers a clear path toward leveraging Universal Composability to obtain end-to-end security with fully instantiated cryptographic mechanisms.

BIOGRAPHICAL SKETCH

Coşku "Josh" Acay was born in Balıkesir, Turkey in 1993. He moved to Ardahan, and then to İstanbul due to his father's career in the military. He graduated from Üsküdar American Academy in 2012. He then attended Carnegie Mellon University, where he earned a Bachelor's degree in Computer Science. After graduating, he worked as a software engineer at WhatsApp for a year. He realized he would be happier doing research, so he joined Cornell to pursue his Ph.D. in 2017.

For my brother, Can.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Andrew Myers. Andrew's unwavering faith in my abilities and constant motivation have been the guiding forces behind this work. I sincerely appreciate Andrew's patience, guidance, and refusal to tell me what to do, even when I insisted on it. He afforded me an incredible degree of freedom, and allowed me figure things out for myself.

Furthermore, I would like to thank my committee members, Elaine Shi and Dexter Kozen, for their invaluable feedback and insightful suggestions which have significantly contributed to improving the quality of my research.

A great many people deserve special thanks. My collaborators, Rolph Recto, Joshua Gancher, and Silei Ren, greatly shaped my research. My friends, Drew Zagieboylo, Danny Adams, Shir Maimon, and Makis Arsenis, supported me throughout my journey. My colleagues, Siqiu Yao, Haobin Ni, Ethan Cecchetti, Andrew Hirsch, Tom Magrino, Isaac Sheff, Mae Milano, Yizhou Zhang, Ian Wilkie Tomasik, and Vivian Ding, have been an exceptional source of new insights and feedback. I would not have finished my Ph.D. without them.

My parents and my brother, Nadide, Coşkun, and Can, have been wonderfully loving and supportive. Thank you for helping me from several continents away.

The work in this dissertation was funded in part by the National Science Foundation.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Developing modern distributed applications is difficult and error-prone, as such applications consist of multiple independent components that communicate and coordinate to appear as a single coherent entity to the end user. Developers of distributed systems must program each component separately, and design communication protocols to keep global state up to date. To make matters worse, a functioning system is not necessarily (or usually) a secure system: distributed systems cross administrative boundaries and involve parties that do not fully trust each other. A correct application must guarantee confidentiality and integrity for all parties.

Developers may turn to programming languages and related tools to help deal with the complexity of building distributed systems; unfortunately, languages and tools leave a lot to be desired. Most programming languages, type systems, compilers, and debuggers are designed for code that runs on a single machine. This means programmers get very little to no help for the most challenging part of designing distributed systems: coordinating multiple components. Even a simple client–server application requires developers to maintain two separate code bases, manage network connections, serialize/deserialize data, match sends and receives, and more. Modern applications go well beyond the simple client–server model: nowadays, there are multiple heterogenous servers, independent cloud providers, peer-to-peer applications, and so on. The complexity of these systems only highlights the weaknesses in programming tools.

The situation is even worse as distributed applications must guarantee not only functional correctness, but also security and reliability. Producing the correct output when all components behave as prescribed is not enough. The system must maintain the confidentiality and integrity of data even when some parties deviate from the

protocol (due to bugs or with malicious intent) or try to glean extra information. For example, in a server-based video game, players have incentive to cheat, so the client is untrusted. To maintain integrity, the server must handle safety-critical computations such as checking that the player has enough energy or in-game currency to perform an action. In a different setting, a privacy-sensitive client may not trust the server to not use or sell their data without their consent [45]. To preserve the confidentiality of client data, as much computation as possible should be performed on the client's machine. Finally, there are settings with mutual distrust between all parties. A client wishing to compare their passwords against a list of compromised passwords should not send all their passwords to the server, which might get hacked. Similarly, the server should not reveal the list to the client, who might be a hacker in disguise attempting to learn weak passwords. In this setting, there is *no party* trusted to perform the computation.

When there is no party trusted to read all input or write all output, the only way to defend security is by employing sophisticated mechanisms such as complex distributed protocols [61, 21], trusted hardware [71, 49, 28], or advanced cryptography. For the compromised passwords example, the client and server can engage in a secure multi-party computation (MPC) protocol [96] that reveals to client if any of their passwords are compromised without revealing any other information to either party. These technologies add significant complexity to software development and require expertise to use successfully [40, 46, 37]. The application developer not only has to implement a distributed system, but correctly and securely employ cryptography, an activity so full of pitfalls that it garnered the maxim "don't roll your own crypto."

This dissertation presents an approach that simplifies the development of distributed systems that use cryptography. Our approach integrates two previously separate lines of work: *program partitioning* and *cryptographic compilation*. Compilers that

perform program partitioning automatically derive a distributed system, but do not support advanced cryptographic mechanisms. Cryptographic compilers are designed to make advanced cryptographic mechanisms easier to use, but are limited to individual primitives. By uniting these lines of work, we get a partitioning compiler that supports secure combinations of multiple cryptographic mechanisms.

The goal of program partitioning is to let programmers write a single program that describes the global behavior of the entire system, and have a compiler automatically derive a correct-by-construction distributed implementation. This is accomplished by "projecting" the global program for each host, that is, given the global program and a host, the compiler determines which parts of the program that host needs to execute, and inserts the necessary communication. In choreographic programming [73, 72, 32, 33, 53], the programmer explicitly places each statement on a host, and is therefore responsible for ensuring this placement does not compromise confidentiality and/or integrity. Jif/split [100, 101] and Swift [25] let programmers specify high-level security policies using information-flow annotations [90], and automatically compute a secure host assignment. Neither Jif/split nor existing work on choreographic programming can incorporate advanced cryptographic mechanisms.

Advanced cryptographic mechanisms such as secure multiparty computation (MPC), zero-knowledge proofs (ZKP), and fully homomorphic encryption (FHE) all expect computation to be represented as circuits. Since programming circuits directly is tedious and error prone, prior work leverages compilers that translate high-level programs into low-level circuits. Unfortunately, most compilers only target a single mechanism—Fairplay [69], ObliVM [66], SCALE-MAMBA [3], Wysteria [85] target MPC; Pinocchio [79], Geppetto [29], Buffet [95], xjSNARK [58] target ZKP; CHET [35], EVA [34], Porcupine [30] target FHE—and thus do not support secure combinations of mecha-

nisms. Efficient applications must use cheaper mechanisms such as local computation, replication, and commitments if possible to avoid the huge overheads associated with advanced cryptographic mechanisms [13].

We present Viaduct, a system that makes it easier for non-expert programmers to develop secure distributed programs that employ cryptography. The key idea of Viaduct is to treat cryptographic mechanisms such as MPC and ZKP as trusted third parties, and to partition programs onto actual hosts as well as hosts simulated by cryptography. Viaduct's *security-typed* language allows developers to annotate programs with information-flow labels to specify fine-grained security policies regarding the confidentiality and integrity of data. An inference algorithm allows these annotations to be lightweight, and enables Viaduct to reject inherently insecure programs. Viaduct then enforces these policies by compiling high-level source code to secure distributed programs, automatically choosing efficient use of cryptography without sacrificing security. The compiler supports a range of cryptographic protocols whose security guarantees are characterized using information-flow labels. New protocols can be added to Viaduct by specifying their security properties and by implementing well-defined interfaces.

The only way to trust cryptography is to prove it correct, and because Viaduct is a compiler, we need to prove the correctness of not a specific protocol, but any protocol generated by Viaduct. Additionally, Viaduct is an extensible compiler that does not bake in a fixed set of cryptographic mechanisms. Our proof must similarly be modular so adding new mechanisms does not necessitate a new proof. We split our correctness into two halves. The first half treats cryptographic mechanisms such as MPC and ZKP uniformly as implementing idealized hosts. We then use information-flow labels to describe the security guarantees of these hosts, and derive a generic proof of correctness for program partitioning. In the second half, we connect our result

to Universal Composability (UC) [17], a framework for showing the correctness of cryptographic mechanisms. This paves the way for instantiating the idealized hosts with actual cryptographic mechanisms.

To our knowledge, by providing a unified abstraction to both specify security policies of programs and to specify security guarantees of cryptographic mechanisms, Viaduct is the first system to compile secure, distributed programs with an *extensible* suite of cryptography, and the first system of its kind with a modular security proof.

1.1 Dissertation Outline

The remainder of this dissertation proceeds as follows. Chapter 2 introduces the Viaduct system, and discusses information-flow type inference and protocol selection. Chapter 3 develops a correctness proof for a model of the Viaduct compiler. Chapter 4 concludes with future directions.

Chapter 2 is based on joint work with Rolph Recto, Joshua Gancher, Andrew Myers, Elaine Shi [2], and Silei Ren. Chapter 3 is based on joint work with Joshua Gancher, Rolph Recto, and Andrew Myers.

CHAPTER 2

SECURE PROGRAM PARTITIONING WITH CRYPTOGRAPHIC PROTOCOLS

Recent efforts from the cryptography community have pushed advanced cryptographic mechanisms from theory to practical deployment [13], but a gap remains: they still require too much expertise to use successfully [40, 46, 37]. In this chapter, we introduce Viaduct, a system that makes it easier for non-expert programmers to develop secure distributed programs that employ cryptography. Viaduct programs declaratively specify security policies regarding the confidentiality and integrity of data and computation using information-flow labels. Viaduct then enforces these policies by compiling high-level source code to secure distributed programs, automatically choosing efficient use of cryptography without sacrificing security.

We make the following contributions:

- An algorithm to infer minimum consistent security requirements of data storage and computation for programs written in a security-typed language. (section 2.3)
- A technique to compile secure distributed programs, deploying an extensible set of cryptographic protocols while minimizing a customizable notion of cost. (section 2.5)
- An extensible runtime system for running compiled programs. Cryptographic mechanisms are added as plug-ins to the runtime. (section 2.6, section 2.7)
- An evaluation that shows that the Viaduct compiler can synthesize a wide variety of secure and efficient distributed programs, that the compilation technique is scalable, and that the annotation burden of the source language is minimal. (section 2.8)
- An open-source implementation of the Viaduct compiler and runtime system.[1]

[1] Available at `https://github.com/apl-cornell/viaduct`.

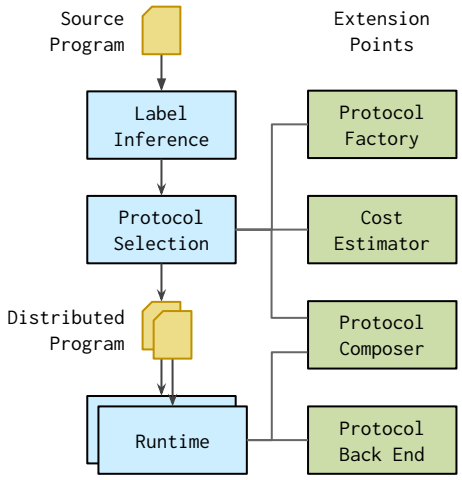


Figure 2.1: Architecture of Viaduct.

2.1 Overview

Figure 2.1 gives a high-level overview of Viaduct. Its compiler takes a high-level source program partially annotated with information-flow labels. The compiler infers labels consistent with programmer-supplied annotations to determine security requirements for all program components. Then for each component the compiler selects a protocol that matches these requirements, guiding the selection with a cost model. The output is a secure and efficient distributed program, which hosts execute using the Viaduct runtime system. The Viaduct architecture has a small set of well-defined extension points, allowing developers to add support for new protocols with relative ease.

We give two examples to motivate and describe the Viaduct compilation process.

Historical Millionaires' Problem Our first example is a slightly modified version of the "millionaires' problem" [96]. As in the classic formulation, two individuals, Alice and Bob, want to determine who has more money without revealing how much money they have to the other person. Rather than comparing their current wealth, in our "historical"

```
1  host Alice, Bob
2  assume Alice trusts Bob for integrity
3  assume Bob trusts Alice for integrity
4
5  fun historicalMillionaires() {
6    val a1, a2, a3 = Alice.input<int>()
7    val b1, b2, b3 = Bob.input<int>()
8    val a = min(a1, a2, a3)
9    val b = min(b1, b2, b3)
10   val b_richer = declassify(a < b, Alice ∨ Bob)
11   Alice.output(b_richer)
12   Bob.output(b_richer)
13 }
```

Figure 2.2: Implementation of the historical millionaires' problem in Viaduct. Viaduct uses MPC for the comparison `a < b`, but computes the minima locally.

variant Alice and Bob want to see who was richer at their *poorest*. Figure 2.2 shows an implementation of the historical millionaires' problem in Viaduct. The program compares Alice's lowest wealth with Bob's, and outputs the answer (`b_richer`) to both Alice and Bob.

Viaduct programs must specify the *hosts* that participate in the program along with any trust assumptions between them. Line 1 declares hosts Alice and Bob, and lines 2 to 3 declare that Alice and Bob trust each other for *integrity* (but not confidentiality), meaning Alice and Bob trust each other to execute the program correctly, but they do not trust each other with their secrets.

All security policies in Viaduct are represented using *security labels*, which are defined formally in section 2.2. Security labels capture both *confidentiality* and *integrity*. Each variable and expression in Viaduct carries a security label, which is derived from the possible flows of information in the program. The variables in lines 6 to 9 carry the same label as their respective hosts, since they only involve data local to that host. However, the comparison `a < b` involves *both* hosts' private data, so has the higher

security label Alice ∧ Bob. This label corresponds to data that is secret to and trusted by both principals. Since Alice ∧ Bob corresponds to secret data, we require an explicit *declassification* to Alice ∨ Bob, which describes data that both hosts can see.

During protocol selection (section 2.5), Viaduct chooses cryptographic protocols to securely and efficiently execute our example. The central idea that allows Viaduct to select protocols automatically is that the security guarantees of protocols can also be captured by labels. Neither Alice nor Bob alone has enough authority to be responsible for the comparison, so Viaduct generates the following distributed implementation: Alice and Bob compute their respective minima locally but perform the comparison `a < b` in semi-honest MPC. A semi-honest MPC protocol works here because Alice and Bob trust each other for integrity. Without that assumption, Viaduct is instead forced to select another protocol such as maliciously secure MPC.

There are typically multiple ways to assign protocols to a given program expression. For example, the computation of Alice's minimum on line 8 could be securely performed in MPC, but since the computation requires the authority of Alice alone, it is cheaper yet still secure to do the computation locally on Alice's machine. Using its cost estimator, Viaduct compiles the optimal program described above.

After protocol selection, Viaduct outputs a distributed program which captures the required cryptography to execute the source program. Hosts can execute this distributed program using Viaduct's runtime system.

Guessing Game Figure 2.3 presents a contrasting example where Alice and Bob do not trust each for integrity, modeling a *malicious* corruption scenario. Since they do not trust each other to execute the program correctly, semi-honest MPC is not applicable. In the *guessing game*, Bob inputs a number `n` and Alice has five attempts to guess

```
host Alice, Bob

fun guessingGame() {
  val n = endorse(Bob.input<int>(), Alice)
  var tries = 5
  var win = false
  while (0 < tries & !win) {
    val guess = declassify(Alice.input<int>(), Bob)
    val tguess = endorse(guess, Bob)
    win = declassify(n == tguess, Alice)
    tries -= 1
  }
  Alice.output(win)
  Bob.output(win)
}
```

Figure 2.3: Guessing game. Alice attempts to guess Bob's secret. Viaduct uses zero-knowledge proofs so Alice learns nothing more than whether her guesses are correct.

the number. Since Bob's input is not trusted by Alice in this setting, it must first be *endorsed* to raise its integrity and prevent Bob from unilaterally modifying the value. This endorsement requires a cryptographic mechanism to protect the integrity and secrecy of variable `n` throughout program execution.

Viaduct synthesizes a program in which Bob commits to `n` so that its value remains secret to Alice but Bob cannot later lie about the committed value. The statement `n == tguess` is computed by having Bob send a zero-knowledge proof (ZKP) to Alice, so that Alice can trust the outcome but learns no additional information. All other variables are replicated in plaintext across the two hosts.

These examples show that Viaduct is general, as it treats protocols such as MPC and ZKP uniformly.

2.1.1 Label Inference

Viaduct selects a protocol for every piece of data and computation in the program based on their authority requirements, represented as labels. Intuitively, program components must be executed by protocols with enough authority to defend the confidentiality of host inputs and the integrity of host outputs. These authority requirements are captured formally by a type system (section 2.3.1), and Viaduct uses a novel inference algorithm (section 2.4) to compute for all program components the minimum-authority labels that still respect the information-flow constraints on the program.

The only required label annotations on Viaduct programs are on `declassify`/`endorse` expressions—all labels on variables can be elided, making annotation burden low. As we show in our evaluation, these required annotations are enough to capture programmer intent: minimally annotated programs compile to the same distributed programs as their fully annotated versions.

2.1.2 Protocol Selection

After label inference, Viaduct performs *protocol selection*, which assigns a protocol to compute and store each subexpression and variable. Protocols encompass storage and computation performed "in the clear" as well as cryptographic mechanisms such as commitments, zero-knowledge proofs, and secure multiparty computation.

Each protocol r carries an associated authority label $\mathbb{L}(r)$, which approximates the security guarantees the protocol provides. Given a program component with minimum authority requirement ℓ, protocol selection only assigns r to execute that component if $\mathbb{L}(r) \Rightarrow \ell$—that is, if r meets the authority requirement for the program component.

Protocol	Confidentiality	Integrity
Local(h)	h	h
Replication(H)	$\bigvee_{h \in H} h$	$\bigwedge_{h \in H} h$
Commitment($h_\mathrm{p}, h_\mathrm{v}$)	h_p	$h_\mathrm{p} \wedge h_\mathrm{v}$
ZKP($h_\mathrm{p}, h_\mathrm{v}$)	h_p	$h_\mathrm{p} \wedge h_\mathrm{v}$
MAL-MPC(H)	$\bigwedge_{h \in H} h$	$\bigwedge_{h \in H} h$
SH-MPC(H)	$\bigwedge_{h \in H} h$	$\bigvee_{h \in H} h$

Figure 2.4: Example protocols and security labels that represent their authority.

Intuitively, given a program s and protocol r, we may imagine an *ideal functionality* r^s (in the style of UC [17]) which executes the program fragments of s that are assigned to r. The fragments of s that are assigned to r may depend on the computational abilities of r. For example, if r is a commitment protocol, then r^s is only able to store values but not perform any computations. If r is an MPC protocol, then r^s can execute computations that can be translated into circuits—the standard interface for MPC implementations.

Functionality r^s guarantees that the storage and computation it performs are protected at label $\mathbb{L}(r)$. In particular, the adversary cannot observe storage or computation performed by r^s unless its confidentiality is at least $\mathbb{L}(r)$; dually, the adversary cannot influence storage or computation performed by r^s unless its integrity is at least $\mathbb{L}(r)$.

Examples of protocols and their corresponding authority labels are given in fig. 2.4. Following the above intuition for the security of functionalities r^s, the authority label of protocols are determined to be the least authority required of the adversary to corrupt the protocol (in confidentiality or integrity). We explain the example protocols below:

Local(h) No cryptography is performed: data is stored and computations performed on host h in the clear. It provides exactly the authority of h.

Replication(H) Data and computations are replicated on all hosts in set H, and replicated data is checked for equality when necessary. This protocol provides low

confidentiality since all hosts hold the plaintext value. It provides high integrity since all hosts must corrupt their local values for the value to be globally corrupted.

Commitment($h_\mathrm{p}, h_\mathrm{v}$) Data is stored on h_p and commitments are placed on h_v. Commitments are computationally inexpensive but usually no computations can be performed with them. Commitments increase integrity without sacrificing confidentiality. Its confidentiality is h_p since only h_p holds the plaintext value, while h_v only holds a commitment. Its integrity is $h_\mathrm{p} \wedge h_\mathrm{v}$ for the same reason as for replication.

ZKP($h_\mathrm{p}, h_\mathrm{v}$) A zero-knowledge proof protocol where h_p is the prover and h_v is the verifier. The prover computes over its private data and sends the result to the verifier, along with a *proof* that attests the value computed is correct. The proof reveals nothing about the private data except what can be gleaned from the result itself. Zero-knowledge proofs provide the same authority as commitments, for essentially the same reason: the prover holds all secret information and performs all computation, while the verifier only holds information which allows it to believe in the correctness of the result, but nothing more.

MAL-MPC(H) A corrupt-majority, maliciously secure multiparty computation protocol [48, 19, 16] performed by hosts H. The protocol allows hosts to jointly perform a computation over their private inputs, keeping these inputs secret to the other hosts and revealing only the result. The label $\bigwedge_{h \in H} h$ reflects that the confidentiality (resp., integrity) of data computed in MPC is compromised only if *all* participating hosts have compromised confidentiality (resp. integrity).

SH-MPC(H) A corrupt-majority, semi-honest secure multiparty computation protocol performed by hosts H. The protocol provides high confidentiality similar to MAL-MPC(H). The low integrity $\bigvee_{h \in H} h$ indicates that the MPC computation

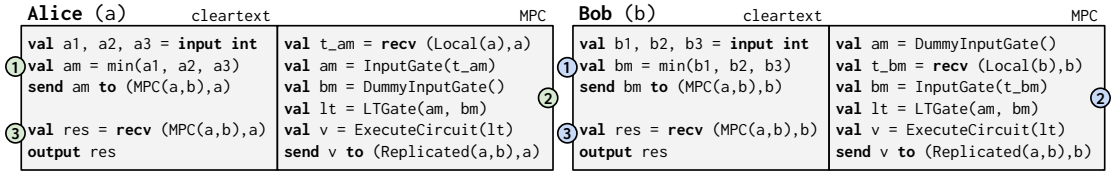


Figure 2.5: Execution of the compiled distributed program for the historical millionaires' problem using a cleartext back end and an MPC back end. Sends and receives are over protocol–host pairs (r, h). These messages are processed by the back end for protocol r at host h.

may be compromised if *any* host behaves maliciously. Effectively, the low integrity means the protocol is only applicable if all hosts H trust each other for integrity.

2.1.3 Runtime

Viaduct provides a modular runtime system for executing compiled distributed programs, implemented as an interpreter. All hosts run the interpreter with the same compiled program, which then executes each host's portion of the program. During execution, the interpreter calls out to back ends implementing the cryptographic mechanisms used in the program. Back ends translate computations in the source language into their cryptographic realizations. For instance, the back ends for MPC and ZKP in our implementation build a circuit representation of the program as it executes.

Protocol back ends can send data to and receive data from each other, supporting the composition of protocols. Source-level declassification and endorsement induce this communication. For example, in fig. 2.2 on line 10, the computation `a < b` is declassified from Alice $\wedge$ Bob to Alice $\vee$ Bob. This declassification causes the MPC protocol between Alice and Bob to execute its stored circuit for this comparison, and to output the result in cleartext.

Figure 2.5 shows the execution of the program compiled by Viaduct for the historical millionaires' problem. The program runs as follows.

- First, the cleartext back ends on Alice and Bob's machines receive input locally and compute their respective minima. The cleartext back ends send the minima as secret inputs to their respective MPC back ends, which create input gates for these values.
- Next, the MPC back ends on Alice and Bob's machines each create an operation gate that compares Alice and Bob's secret inputs. The back ends jointly execute the circuit with the comparison result as output, which they send to their respective cleartext back ends.
- Finally, the cleartext back ends on Alice and Bob's machines both receive from their MPC back ends and output the result.

2.1.4 Threat Model

Compiled programs run in a distributed setting in which each host executes a single thread concurrently with other hosts. Hosts communicate via message passing over secure, private, asynchronous channels. There is no shared memory that spans multiple hosts. We assume the attacker cannot observe wall-clock timing. Additionally, we are not concerned with availability, so the attacker can halt execution at any time.

In the setting of Viaduct, there is no single notion of an attacker. For example, in the historical millionaires problem, neither Alice nor Bob fully trust the other. To Alice, Bob is a potential attacker; Alice expects her security requirements to be met as long as her assumptions about Bob are correct (that is, Bob follows the protocol but may try to

leak data). Conversely, to Bob, Alice is a potential attacker. Hence, we are concerned with security versus all possible attackers.

Corruption happens at the level of hosts. Hosts can be *honest*, *semi-honest*, or *malicious*. Malicious hosts are fully controlled by the adversary; semi-honest hosts follow the protocol, but leak all their data to the adversary [64]. Corruption must be consistent with the trust assumptions between hosts. For example, if Bob trusts Alice with confidentiality and Alice is semi-honest, then Bob is also semi-honest (if the adversary can read data on Alice, and Bob is willing to let Alice see his data, then the adversary can see Bob's data). Similarly, if Bob trusts Alice with integrity and Alice is malicious, then so is Bob.

In the historical millionaires' problem (fig. 2.2), there are five interesting corruption scenarios: both are honest; Alice is semi-honest; Bob is semi-honest; both are semi-honest; both are malicious. Malicious corruption of a single host is not possible because the hosts trust each other for integrity.

2.2 Information-Flow Control

To capture the adversary's power to *read* and *write*, we use a label model that can describe confidentiality and integrity simultaneously [75, 90, 4].

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 the set of hosts $\mathbb{H} = \{\text{Alice}, \text{Bob}, \text{Chuck}, \ldots\}$.

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:

$$\langle p, q \rangle^{\rightarrow} = \langle p, \mathbf{1} \rangle \qquad\qquad \langle p, q \rangle^{\leftarrow} = \langle \mathbf{1}, q \rangle.$$

Like DLM [75] and FLAM [4], we use the authority ordering on principals to define information flow. Intuitively, information flow policies become more restrictive as they become more secret and less trusted:

$$\langle p_1, q_1 \rangle \sqsubseteq \langle p_2, q_2 \rangle \iff p_2 \Rightarrow p_1 \text{ and } q_1 \Rightarrow q_2 \qquad \text{(flows to)}$$

$$\langle p_1, q_1 \rangle \sqcup \langle p_2, q_2 \rangle = \langle p_1 \wedge p_2, q_1 \vee q_2 \rangle \qquad \text{(join)}$$

$$\langle p_1, q_1 \rangle \sqcap \langle p_2, q_2 \rangle = \langle p_1 \vee p_2, q_1 \wedge q_2 \rangle \qquad \text{(meet)}$$

The flows-to relation $\ell_1 \sqsubseteq \ell_2$ orders information flow policies: it means label ℓ_1 is more permissive about the use of information than ℓ_2. The join $\ell_1 \sqcup \ell_2$ is more restrictive than both ℓ_1 and ℓ_2, and the meet $\ell_1 \sqcap \ell_2$ is more permissive than either ℓ_1 or ℓ_2. The least restrictive policy is $\mathbf{0}^{\leftarrow}$ ("public trusted"), describing information that can be used anywhere, while the most restrictive is $\mathbf{0}^{\rightarrow}$ ("secret untrusted").

2.2.1 Capturing Attacks with Labels

To reason about the security of a system, we need a clear definition of what attackers can do. Formally, an attack is specified by picking two sets of *principals*: public principals

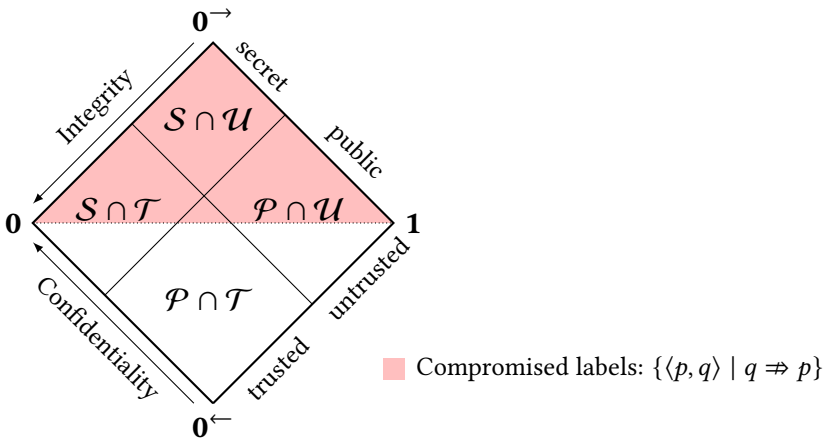


Figure 2.6: Information flow lattice. Attacks are valid when $\mathcal{S} \cap \mathcal{U}$ is entirely compromised.

$P \subseteq \mathbb{P}$ and untrusted principals $Q \subseteq \mathbb{P}$. Some common-sense conditions must hold on the sets P and Q [23]. We state the conditions for P but they apply equally to Q.

- The attacker always controls the weakest principal, but never the strongest: $\mathbf{1} \in P$ and $\mathbf{0} \notin P$.
- If the attacker controls a principal, then it controls all weaker principals: if $p \in P$ and $p \Rightarrow q$, then $q \in P$.
- Attacker-controlled principals may collude: if $p, q \in P$, then $p \wedge q \in P$.
- If the attacker controls the the common power of two principals, then it controls at least one of the 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 $\vee$.[2]

Given the sets of attacker-controlled principals P and Q, we can partitioning labels $\mathbb{L}$ across the two axes: public/secret and trusted/untrusted. We denote these sets as $\mathcal{P}/\mathcal{S}$

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

Delegation	d	::=	$p \Rightarrow q$
Principal Delegations	θ	::=	$d_1, \ldots, d_2$
Label Delegations	Θ	::=	$\langle \theta_{\rightarrow}, \theta_{\leftarrow} \rangle$

Figure 2.7: Syntax of delegations.

and $\mathcal{T}/\mathcal{U}$, respectively. The partitioning is depicted in fig. 2.6 and defined as follows:

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

For a host h, define $\mathbb{L}(h) = \langle h, h\rangle$. Recalling the threat model, 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 restricting attacks to *valid attacks*.

Definition 2.2.1 (Valid Attack)**.** Attack $\langle P, Q\rangle$ is valid if all untrusted principals are public: $Q \subseteq P$.

Similar to hosts, protocols with secret, untrusted labels do not make sense. We rule out such protocols by requiring protocol labels to be *uncompromised* [97]. A valid attack never classifies an uncompromised label as secret and untrusted.

Definition 2.2.2 (Uncompromised 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 2.2.3. *If $\langle P, Q\rangle$ is a valid attack and $\blacktriangledown\ell$, then $\ell \notin \mathcal{S} \cap \mathcal{U}$.*

$\boxed{P \models d}$ $\qquad$ $\boxed{P \models \theta}$

$$\frac{p \in P \implies q \in P}{P \models p \Rightarrow q} \qquad \frac{\forall d \in \theta \,.\, P \models d}{P \models \theta}$$

$\boxed{\theta \models p \Rightarrow q}$

$$\frac{\forall P \,.\, (P \models \theta \implies P \models p \Rightarrow q)}{\theta \models p \Rightarrow q}$$

Figure 2.8: Semantics of delegations.

2.2.2 Delegation

We capture trust assumptions between hosts as sets of *delegations*. Figure 2.7 gives the syntax of delegations. A delegation is an assumption on the acts-for relation: the assumption $p \Rightarrow q$ means principal q delegates its authority to principal p (i.e., q trusts p). A principal-delegation context θ is a set of delegations. Since labels are pairs of principals, a label-delegation context Θ is two sets of delegations: $\theta_{\rightarrow}$ collecting confidentiality delegations and $\theta_{\leftarrow}$ collecting integrity delegations. For example, the assumptions in fig. 2.2 (lines 2 to 3) are captured with the context $\langle \epsilon, \{\text{Bob} \Rightarrow \text{Alice}, \text{Alice} \Rightarrow \text{Bob}\}\rangle$.

Since delegations are *assumptions*, we model them as limitations on the sets of attacks we consider. For example, with the assumption Alice $\Rightarrow$ Bob in the system, we are no longer interested in attackers that control Alice but not Bob. We formally capture this insight in fig. 2.8. We write $P \models \theta$ when an attack P is consistent with every delegation in θ. We say p acts for q under the assumptions θ, written $\theta \models p \Rightarrow q$, when any attacker consistent with θ that controls p necessarily controls q.[3]

[3]A delegation Alice $\Rightarrow$ Bob has an effect beyond the principals explicitly involved (i.e., Alice and Bob). For example, it implies Alice $\wedge$ Chuck $\Rightarrow$ Bob $\wedge$ Chuck.

When comparing labels, we use two sets of delegations $\Theta = \langle \theta_{\rightarrow}, \theta_{\leftarrow} \rangle$: $\theta_{\rightarrow}$ for the confidentiality component and $\theta_{\leftarrow}$ for the integrity component. For example,

$$\langle \theta_{\rightarrow}, \theta_{\leftarrow} \rangle \models \langle p_1, q_1 \rangle \sqsubseteq \langle p_2, q_2 \rangle \iff \theta_{\rightarrow} \models p_2 \Rightarrow p_1 \text{ and } \theta_{\leftarrow} \models q_1 \Rightarrow q_2.$$

The fact that we have different delegations for confidentiality and integrity leads to a problem when defining uncompromised labels, where we compare the integrity component of a label to its confidentiality component. Namely, which context do we use? We resolve the problem by using an intermediary principal.

Definition 2.2.4 (Uncompromised Label Under Delegations)**.** $\ell = \langle p, q \rangle$ in uncompromised under delegations $\Theta = \langle \theta_{\rightarrow}, \theta_{\leftarrow} \rangle$, written $\Theta \models \blacktriangledown \ell$, if there exists $p' \in \mathbb{P}$ such that $\theta_{\leftarrow} \models q \Rightarrow p'$ and $\theta_{\rightarrow} \models p' \Rightarrow p$.

Delegations in Viaduct are defined globally and fixed for the entire program. Therefore, we fix a delegation context Θ for the rest of our development, and suppress contexts in judgments.

2.3 Source Language

The syntax for Viaduct's source language, a simplified version of the *surface* language, is given in fig. 2.9. The language supports an abstract set of values and operators over them. We distinguish between fully evaluated atomic expressions t, and expressions e that evaluate to values and may have side effects. The **declassify** expression marks locations where private data is explicitly allowed to flow to public data, while the **endorse** expression marks locations where untrusted data is explicitly allowed to influence trusted data. The **input**/**output** expressions allow programs to interact with hosts.

$$
\begin{array}{lrcl}
\text{Principals} & p & \in & \mathbb{P} \\
\text{Labels} & \ell & \in & \mathbb{L} = \mathbb{P} \times \mathbb{P} \\
\multicolumn{4}{l}{\text{Variables } x \in \mathbb{X} \qquad \text{Hosts } h \in \mathbb{H} \qquad \text{Protocols } r \in \mathbb{R}} \\
\text{Values} & v & \in & \mathbb{V} \ni 0 \\
\text{Operators} & f & \in & \mathbb{F} \\
\text{Atomic Expressions} & t & ::= & v \mid x \\
\text{Expressions} & e & ::= & f(t_1, \ldots, t_n) \\
 & & \mid & \textbf{declassify}(t, p) \mid \textbf{endorse}(t, p) \\
 & & \mid & \textbf{input}\ h \mid \textbf{output}\ t\ \textbf{to}\ h \\
\text{Statements} & s & ::= & \textbf{let}\ x = e; s \\
 & & \mid & \textbf{if}\ t\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \\
 & & \mid & \mu X.s \mid X \\
 & & \mid & s_1; s_2 \mid \textbf{skip}
\end{array}
$$

Figure 2.9: Abstract syntax of Viaduct's source language

Since declassification does not affect integrity, only the *change* to confidentiality must be specified. Symmetrically, endorsement changes integrity but not confidentiality. Therefore, **declassify**/**endorse** expressions specify a single label component (a principal) p instead of a full label ℓ.

Statements consist of **let**-bindings, conditionals, recursive statements, and sequential composition. Surface-level `val` statements are represented as **let** statements. For simplicity, we do not model mutable variables (`var` declarations); they are easy to add to the language. We require all intermediate computations to be **let**-bound, enforcing a variant of *A-normal form* [41]. We use the more general recursive statements instead of the more traditional **for**/**while** loops, simplifying the conversion to A-normal form. Loops are recovered easily, for example:

$$\textbf{while}\ t\ \textbf{do}\ s \triangleq \mu X.\textbf{if}\ t\ \textbf{then}\ (s; X)\ \textbf{else skip}.$$

$\boxed{\Gamma \vdash t : \ell}$ $\qquad$ $\boxed{\Gamma; pc \vdash e : \ell}$

$$\frac{}{\Gamma \vdash v : \ell} \qquad \frac{\Gamma(x) \sqsubseteq \ell}{\Gamma \vdash x : \ell} \qquad \frac{\forall i \,.\, \Gamma \vdash t_i : \ell}{\Gamma; pc \vdash f(t_1, \ldots, t_n) : \ell} \qquad \frac{pc \sqsubseteq \ell \qquad \Gamma \vdash t : \ell_{\mathrm{f}} \qquad \blacktriangledown \ell_{\mathrm{f}} \qquad \ell_{\mathrm{f}} \sqcap \langle p, \mathbf{1} \rangle \sqsubseteq \ell}{\Gamma; pc \vdash \mathbf{declassify}(t, p) : \ell}$$

$$\frac{pc \sqsubseteq \ell \qquad \Gamma \vdash t : \ell_{\mathrm{f}} \qquad \blacktriangledown \ell_{\mathrm{f}} \qquad \ell_{\mathrm{f}} \sqcap \langle \mathbf{0}, p \rangle \sqsubseteq \ell}{\Gamma; pc \vdash \mathbf{endorse}(t, p) : \ell} \qquad \frac{pc \sqsubseteq \mathbb{L}(h) \qquad \mathbb{L}(h) \sqsubseteq \ell}{\Gamma; pc \vdash \mathbf{input}\ h : \ell} \qquad \frac{pc \sqsubseteq \mathbb{L}(h) \qquad \Gamma \vdash t : \mathbb{L}(h)}{\Gamma; pc \vdash \mathbf{output}\ t\ \mathbf{to}\ h : \ell}$$

$\boxed{\Gamma; pc \vdash s}$

$$\frac{\Gamma; pc \vdash e : \ell \qquad pc \sqsubseteq \ell \qquad (\Gamma, x : \ell); pc \vdash s}{\Gamma; pc \vdash \mathbf{let}\ x = e; s} \qquad \frac{pc \sqsubseteq pc' \qquad \Gamma \vdash t : pc' \qquad \Gamma; pc' \vdash s_1 \qquad \Gamma; pc' \vdash s_2}{\Gamma; pc \vdash \mathbf{if}\ t\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2} \qquad \frac{pc \sqsubseteq pc' \qquad (\Gamma, X : pc'); pc' \vdash s}{\Gamma; pc \vdash \mu X.s}$$

$$\frac{pc \sqsubseteq \Gamma(X)}{\Gamma; pc \vdash X} \qquad \frac{\Gamma; pc \vdash s_1 \qquad \Gamma; pc \vdash s_2}{\Gamma; pc \vdash s_1; s_2} \qquad \frac{}{\Gamma; pc \vdash \mathbf{skip}}$$

Figure 2.10: Information flow checking rules for expressions and statements.

2.3.1 Label Checking

Viaduct's type system enforces secure information flow in a standard way. The type system serves two purposes. First, it helps programmers ensure there are no unintended information flows: secrets are not leaked to and data is not corrupted by unauthorized principals. Second, it specifies what labels can be assigned to variables and expressions that the user did not explicitly annotate.

Figure 2.10 presents label checking rules for expressions and statements. Expressions are checked by the judgment $\Gamma; pc \vdash e : \ell$, which means that e has label ℓ under the context on the left. Here, Γ is a finite partial map from variables to labels:

$$\text{Label Contexts} \quad \Gamma \quad ::= \quad \epsilon \mid \Gamma, x : \ell \mid \Gamma, X : \ell$$

The *program counter* label *pc* is a standard way to prevent implicit flows of information via control flow [88]. The rules for **input**/**output** expressions differ from those in standard security-typed languages in that they also include premises with *pc* checks. These checks are required because hosts must be able to follow control flow to respond to **input**/**output** requests. Prior work that targets the distributed setting contains similar checks to control *read channels* [100].

The rules for **declassify**/**endorse** expressions enforce NMIFC [23]. They require that the data being downgraded has an uncompromised label [97]. They combine the label of the input expression with the specified *change p* to compute the output label: the **declassify** expression weakens confidentiality by p (the output is more public), and the **endorse** expression strengthens integrity by p (the output is more trusted).

Statement checking rules have the form $\Gamma; pc \vdash s$; they are largely standard [88]. Because we assume attackers cannot observe timing nor analyze traffic, the rule for conditional statements does not require branches to have the same timing behavior or effects (e.g., **input**/**output**).

Nonmalleable Information Flow Control

Information flow type systems typically aim to enforce a compositional security property such as *noninterference* [47]. Noninterference is a strong property but it is too restrictive for practical applications, which usually have a more nuanced policy for secure information flow. Hence, like most languages supporting information flow control (e.g., [74, 84, 14]), Viaduct allows programmers to signify the exceptions to a noninterference policy through *downgrading* expressions.

```
host Client, Server
assume Client trusts Server

val info: int{Server}, pw: int{Server}, guess: int{Client}
if (declassify (pw == guess) to {Client})
  Client.output(declassify info to {Client})
```

Figure 2.11: A program that violates robust declassification.

Downgrading enables information flows that would violate noninterference, so it can be dangerous. This is especially true in the distributed setting, where storage and computation can be performed by hosts that one does not fully trust. Downgrading confidentiality (declassification) allows secret information to be treated as public information—a necessity for many applications, but doing so might allow a corrupted host to control when information is released or what information is released. Downgrading integrity (endorsement) allows untrusted information to be treated as trusted information, but might enable a corrupted host to trick an honest one into accepting mauled secrets.

The property of *nonmalleable information flow control* (NMIFC) [23] prevents both of these abuses of downgrading by combining two properties: *robust declassification* [99] and *transparent endorsement* [23]. Robust declassification requires that principals to which data is declassified could not have influenced either the decision to declassify or the data itself. Meanwhile, transparent endorsement prevents trusting mauled secrets by ensuring that information can only be endorsed if the providing principal can read it.

The declassification and endorsement rules in fig. 2.10 enforce NMIFC by preventing the program from downgrading information with *compromised labels* [97]. These rules generate authority requirements that prevent the Viaduct compiler from placing data and computation on insufficiently trustworthy hosts.

Consider the program in fig. 2.11, where a server releases secret information to a client when the client guesses the correct password. This program violates robust declassification, because the decision to declassify `info` depends on (low-integrity) `guess`. Without the restrictions on downgrading, Viaduct could compile the program to store the guard `pw == guess` (with label Client) on Client. Client could simply claim to the server that its guess is correct! For this program to type-check with NMIFC, endorsement is needed to make the guard high-integrity. A naive programmer might think to endorse the entire guard, but this (nontransparent) endorsement could still be compiled in a way that lets an untrusted host supply its value. The correct solution is to explicitly endorse `guess` before declassifying the comparison; since `guess` is not secret, the endorsement is transparent. The resulting labels correctly force Viaduct to put the comparison on the server.

2.4 Label Inference

Checking secure information flow is not enough; for protocol selection, the compiler also needs the labels of all expressions. We present an algorithm to infer these labels.

As in prior work on inferring information flow labels [74, 84], information flow checking reduces to a system of flows-to ($\sqsubseteq$) constraints over label constants and label variables. Type inference collects these premises from fig. 2.10, and generates fresh label variables for labels that appear in a premise of a rule but not its conclusion (e.g., ℓ_f in the rule for **declassify** expressions and pc' in the rule for **if** statements). The inference algorithm finds a label-variable assignment that satisfies all the constraints, if possible.

The algorithm computes the *minimum-authority* solution, the choice of labels requiring the least amount of confidentiality and integrity for each component. Minimum-

Label Constants	ℓ		
Label Variables	Y		
Authority Projections	π	$\in$	$\{\rightarrow, \leftarrow\}$
Label Expressions	L	::=	$\ell \mid Y \mid L^{\pi} \mid L_1 \sqcup L_2 \mid L_1 \sqcap L_2 \mid L_1 \vee L_2 \mid L_1 \wedge L_2$
Label Constraints	C	::=	$L_1 \sqsubseteq L_2 \mid \blacktriangledown L$

Figure 2.12: Syntax of label constraints.

authority labels are desirable because higher authority is achieved only through more trust or costly cryptography.

2.4.1 Label Constraints

Figure 2.12 gives the syntax of the label constraint language. Expressions in the constraint language include label constants and label variables, as well as standard lattice operations like join and meet. Additionally, we have authority projections L^{π} that allow isolating the confidentiality ($\rightarrow$) or the integrity ($\leftarrow$) component of an expression L.

Constraints have two forms: we can assert that an expression flows to another, or we can assert that an expression is uncompromised. The premises in fig. 2.10 can be readily translated to constraints in the label constraint language.

Recall that a label ℓ is a pair of principals $\langle p, q \rangle$, meaning label expressions and label constraints are essentially syntactic sugar for manipulating pairs. Instead of solving label constraints directly, we translate constraints over labels to constraints over principals.

$$
\begin{array}{lll}
\text{Principal Constants} & p & \\
\text{Principal Variables} & Y^{\pi} & \\
\text{Principal Expressions} & P^{\pi} & ::= \quad p \mid Y^{\pi} \mid P_1^{\pi} \vee P_2^{\pi} \mid P_1^{\pi} \wedge P_2^{\pi} \\
 & & \quad\;\; \mid \quad p_1 \rightarrow P_2^{\pi} \mid \min_{\pi'}(P^{\pi'}) \\
\text{Principal Constraints} & D & ::= \quad P_1^{\pi} \Rightarrow_{\pi} P_2^{\pi}
\end{array}
$$

Figure 2.13: Syntax of principal constraints.

2.4.2 Principal Constraints

Figure 2.13 gives the syntax of the principal constraint language. Since labels are tuples of principal components, the syntax includes a principal variable Y^{π} for each combination of label variable Y and projection π. That is, $Y^{\rightarrow}$ represents the confidentiality component of Y, and $Y^{\leftarrow}$ represents the integrity component of Y.

We index expressions P^{π} by the component π they represent. This prevents mixing confidentiality and integrity components in the same expression, for example, by writing $Y_1^{\rightarrow} \wedge Y_2^{\leftarrow}$. Expressions include principal constants and principal variables, as well as principal-lattice operations $\vee$ and $\wedge$. The operation $\rightarrow$ is called the *relative pseudocomplement* of $\wedge$: $p_1 \rightarrow p_2$ is defined as the weakest principal p such that $p_1 \wedge p \Rightarrow p_2$. We use $\rightarrow$ to solve constraints of the form $Y^{\pi} \wedge p_1 \Rightarrow_{\pi} P_2^{\pi}$.[4] The operation $\min_{\pi}(p)$ is the strongest principal p' equivalent to p according to the delegations θ_{π} for the π component. This operation allows mixing integrity and confidentiality components; we use it when solving for labels that must be uncompromised.

Constraints have the form $P_1^{\pi} \Rightarrow_{\pi} P_2^{\pi}$, which asserts that expression P_1^{π} acts for P_2^{π} under delegations θ_{π}. Note that we must specify π since we might have different delegations for confidentiality and integrity.

[4] A lattice that supports the $\rightarrow$ operation is called a *Heyting algebra* [87]. Any free distributive lattice, such as our lattice of principals, is a Heyting algebra.

$\boxed{[\![L]\!]_\pi = P^\pi}$

$$
\begin{aligned}
[\![\langle p, q\rangle]\!]_\pi &= \begin{cases} p & \text{if } \pi = \rightarrow \\ q & \text{if } \pi = \leftarrow \end{cases} \\
[\![Y]\!]_\pi &= Y^\pi \\
[\![L^{\pi'}]\!]_\pi &= \begin{cases} [\![L]\!]_\pi & \text{if } \pi = \pi' \\ \mathbf{1} & \text{if } \pi \neq \pi' \end{cases} \\
[\![L_1 \sqcup L_2]\!]_\pi &= [\![(L_1 \wedge L_2)^{\rightarrow} \wedge (L_1 \vee L_2)^{\leftarrow}]\!]_\pi \\
[\![L_1 \sqcap L_2]\!]_\pi &= [\![(L_1 \vee L_2)^{\rightarrow} \wedge (L_1 \wedge L_2)^{\leftarrow}]\!]_\pi \\
[\![L_1 \vee L_2]\!]_\pi &= [\![L_1]\!]_\pi \vee [\![L_2]\!]_\pi \\
[\![L_1 \wedge L_2]\!]_\pi &= [\![L_1]\!]_\pi \wedge [\![L_2]\!]_\pi
\end{aligned}
$$

$\boxed{[\![C]\!] = D_1, \ldots, D_n}$

$$
\begin{aligned}
[\![L_1 \sqsubseteq L_2]\!] &= [\![L_2]\!]_\rightarrow \Rightarrow_\rightarrow [\![L_1]\!]_\rightarrow,\ [\![L_1]\!]_\leftarrow \Rightarrow_\leftarrow [\![L_2]\!]_\leftarrow \\
[\![\blacktriangledown L]\!] &= [\![L]\!]_\leftarrow \Rightarrow_\leftarrow \min_\rightarrow([\![L]\!]_\rightarrow)
\end{aligned}
$$

Figure 2.14: Translating label constraints to principal constraints.

Figure 2.14 gives rules for translating label constraints to principal constraints. First, we define a function $[\![L]\!]_\pi$, which returns a principal expression representing the confidentiality or the integrity component (depending on π) of the label expression L. The definition of $[\![L]\!]_\pi$ is a straightforward reading of the rules in section 2.2. Using $[\![L]\!]$, we translate a flows-to ($\sqsubseteq$) constraint to multiple acts-for ($\Rightarrow$) constraints, one for each label component, following section 2.2. The constraint $\blacktriangledown L$ requires the integrity of L to act for its confidentiality, but this raises a question: should we use confidentiality or integrity delegations when deciding what acts-for means? Since we have different delegations for confidentiality and integrity, the confidentiality and integrity components of L intuitively "live in different universes," and therefore cannot be directly compared. We solve this problem by using the operation $\min_\pi(\cdot)$: we

$$\frac{P_1^\pi \Rightarrow_\pi P^\pi \qquad P_2^\pi \Rightarrow_\pi P^\pi}{P_1^\pi \vee P_2^\pi \Rightarrow_\pi P^\pi} \qquad \frac{(P_1^\pi \wedge P_3^\pi) \vee (P_2^\pi \wedge P_3^\pi) \Rightarrow_\pi P^\pi}{(P_1^\pi \vee P_2^\pi) \wedge P_3^\pi \Rightarrow_\pi P^\pi} \qquad \frac{P_1^\pi \Rightarrow_\pi (p_2 \to P^\pi)}{P_1^\pi \wedge p_2 \Rightarrow_\pi P^\pi}$$

Figure 2.15: Rules for simplifying principal constraints.

compute acts-for in the integrity domain ($\Rightarrow_{\leftarrow}$), and compute a strongest representative for the confidentiality component of L in the confidentiality domain ($\min_{\rightarrow}(L)$).

2.4.3 Solving Principal Constraints

We are now ready to discuss how we solve constraints over principals. Our constraint solver requires the left-hand side of each constraint to be atomic (a constant or a variable), that is, we only work with constraints of the form $p_1 \Rightarrow_\pi P_2^\pi$ and $Y_1^\pi \Rightarrow_\pi P_2^\pi$. We apply the rules in fig. 2.15, as well as the associativity, commutativity, and idempotence of $\vee$ and $\wedge$, until no left-hand side of any constraint can be simplified any further. This process always terminates, and ensures that the left-hand side of each constraint either is atomic or contains a meet ($\wedge$).[5]

Constraint solving fails if the left-hand side of any constraint contains a meet, since such constraint systems do not have unique solutions. For example, the system $Y_1 \wedge Y_2 \Rightarrow$ Alice has no minimal solution: we can assign $\{Y_1 \mapsto \text{Alice}, Y_2 \mapsto \mathbf{1}\}$ or $\{Y_1 \mapsto \mathbf{1}, Y_2 \mapsto \text{Alice}\}$, but neither solution is better than the other. The typing rules we give in fig. 2.10 never lead to constraint systems with meets on the left (after simplification). However, our implementation of the Viaduct compiler supports label polymorphic functions that allow arbitrary constraints as side conditions, which might lead to unsolvable system. The Viaduct compiler fails with an informative error message

[5]Translation rules in fig. 2.14 never generate constraints with $\rightarrow$ or $\min_\pi(\cdot)$ on the left-hand side, and the rules in fig. 2.15 eliminate all joins ($\vee$) on the left.

that points at the constraint that generates a meet on the left-hand side when this happens.

Assume the simplification process succeeds and produces a constraint system with only atomic expressions on the left-hand side of constraints. We adapt the algorithm of Rehof and Mogensen [86] for iteratively solving semilattice constraints. Fix a delegation context $\Theta = \langle \theta_{\rightarrow}, \theta_{\leftarrow} \rangle$. We initialize all principal variables to $\mathbf{1}$, and use unsatisfied constraints to update variables repeatedly until a fixed point is reached, using the following rule:

$$\text{given} \quad Y^{\pi} \Rightarrow_{\pi} P^{\pi}, \quad \text{set} \quad Y^{\pi} := Y^{\pi} \wedge \text{current-value}(\Theta, P^{\pi}),$$

where current-value(Θ, P^{π}) is the value of P^{π} according to the current assignment. Note that the update rule only uses Θ when computing $\min_{\pi}(\cdot)$; in particular, lattice operations $\vee$, $\wedge$, and $\rightarrow$ are computed using the underlying lattice, completely ignoring delegations. Additionally, constraints that have principal constants p on the left-hand side are ignored during the fixed point computation. Once we reach a fixed point solution, we perform the following check for each constraint with a constant left-hand side:

$$\text{given} \quad p \Rightarrow_{\pi} P^{\pi}, \quad \text{check} \quad \theta_{\pi} \models p \Rightarrow \text{current-value}(\Theta, P^{\pi}).$$

We have a minimal-authority solution if all such constraints are satisfied; otherwise, there is no valid solution to the constraint system.

2.5 Protocol Selection

The protocol selection phase of Viaduct assigns a protocol to each program component. Formally, a *protocol assignment* associates a protocol to each variable declaration (**let**

$\boxed{\Pi \models t : r}$ $\boxed{\Pi \models e : r}$

$$\frac{}{\Pi \models v : r} \qquad \frac{\mathbb{L}(x)^{\rightarrow} \sqsubseteq \mathbb{L}(r)^{\rightarrow} \qquad \mathrm{comm}(\Pi(x), r)}{\Pi \models x : r} \qquad \frac{\forall i \,.\, \Pi \models t_i : r}{\Pi \models f(t_1, \ldots, t_n) : r}$$

$$\frac{\Pi \models t : r}{\Pi \models \textbf{declassify}(t, p) : r} \qquad \frac{\Pi \models t : r}{\Pi \models \textbf{endorse}(t, p) : r} \qquad \frac{}{\Pi \models \textbf{input}\ h : \mathrm{Local}(h)}$$

$$\frac{\Pi \models t : \mathrm{Local}(h)}{\Pi \models \textbf{output}\ t\ \textbf{to}\ h : \mathrm{Local}(h)}$$

$\boxed{\Pi \models s}$

$$\frac{\begin{array}{c}\mathbb{L}(r) \Rightarrow \mathbb{L}(x) \\ \Pi \models e : r \qquad \Pi, x : r \models s\end{array}}{\Pi \models \textbf{let}\ r.x = e; s} \qquad \frac{\begin{array}{c}H = \mathrm{hosts}(s_1) \cup \mathrm{hosts}(s_2) \\ \forall h \in H \,.\, \mathbb{L}(x)^{\rightarrow} \sqsubseteq \mathbb{L}(h)^{\rightarrow} \\ \forall h \in H \,.\, \mathbb{L}(\Pi(x))^{\leftarrow} \sqsubseteq \mathbb{L}(h)^{\leftarrow} \\ \forall h \in H \,.\, \text{local-export}(\Pi(x), h) \\ \Pi \models s_1 \qquad \Pi \models s_2\end{array}}{\Pi \models \textbf{if}\ x\ \textbf{then}\ s_1\ \textbf{else}\ s_2}$$

$$\frac{\Pi \models s_1 \qquad \Pi \models s_2}{\Pi \models \textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2} \qquad \frac{\Pi \models s}{\Pi \models \mu X.s} \qquad \frac{}{\Pi \models X} \qquad \frac{\Pi \models s_1 \qquad \Pi \models s_2}{\Pi \models s_1; s_2} \qquad \frac{}{\Pi \models \textbf{skip}}$$

Figure 2.16: Rules for validating a protocol assignment.

statement), which we denote by annotating the source program. An annotated statement **let** $r.x = e; s$ means r executes e and stores the result in x; other protocols in s that refer to x receive the value of x from r.

We first give a declarative specification of when a protocol assignment is correct; we then explain how to search for such assignments.

$\boxed{\text{protocols}(s) = R}$

$$
\begin{aligned}
\text{protocols}(\textbf{let } r.x = e; s) &= \{r\} \cup \text{protocols}(s) \\
\text{protocols}(\textbf{if } t \textbf{ then } s_1 \textbf{ else } s_2) &= \text{protocols}(s_1) \cup \text{protocols}(s_2) \\
\text{protocols}(\mu X.s) &= \text{protocols}(s) \\
\text{protocols}(X) &= \text{protocols}(\mu X.s) \\
&\quad \text{where } \mu X.s \text{ is the original definition of } X \\
\text{protocols}(s_1; s_2) &= \text{protocols}(s_1) \cup \text{protocols}(s_2) \\
\text{protocols}(\textbf{skip}) &= \emptyset
\end{aligned}
$$

$\boxed{\text{hosts}(s) = H}$

$$\text{hosts}(s) = \bigcup_{r \in \text{protocols}(s)} \text{hosts}(r)$$

Figure 2.17: Protocols and hosts involved in the execution of a statement. Here, $\text{hosts}(r)$ is the set of hosts that protocol r runs on, specified individually for each protocol.

2.5.1 Validity of Protocol Assignments

Figure 2.16 outlines the conditions under which a protocol assignment is valid. The judgment $\Pi \models e : r$ means that expression e can be executed by protocol r. Similarly, the judgment $\Pi \models s$ means that s has a valid protocol assignment. The context Π maps variables to the protocols storing them:

$$\text{Protocol Contexts} \quad \Pi \quad ::= \quad \epsilon \mid \Pi, x : r$$

We now describe the rules for validity. The rule for variables states that x can only be read by protocol r if r has enough confidentiality to read x, and the protocol storing x can communicate with r, written $\text{comm}(r_x, r)$. Not all pairs of protocols can communicate; the customizable *protocol composer*, discussed further in section 2.6.1, defines the valid set of protocol compositions. Other rules restrict where certain expressions can be executed. The **input**/**output** expressions must be executed locally on the relevant host.

The rule for **let** statements ensures that the protocol selected for a variable has enough authority to securely store it. Formally, the label $\mathbb{L}(r)$ of the protocol storing x must act for ($\Rightarrow$) the minimum required authority label $\mathbb{L}(x)$ computed for x in section 2.4. Labels $\mathbb{L}(r)$ are the ones explained in fig. 2.4.

The rule for conditional statements ensures that the control flow is public to and trusted by all hosts involved in the execution of a conditional statement (fig. 2.17). The first premise requires that involved hosts have enough confidentiality to read the value of the conditional guard. The second premise requires that the protocol storing the guard has enough integrity to supply the guard to the involved hosts. The third premise ensures that the protocol storing the guard can forward it to the involved hosts, written local-export($\Pi(x), h$). These premises are trivially satisfied when the guard is a constant expression.

The judgment local-export(r, h) holds when protocol r stores values in a way that allows host h to receive the cleartext value from r without communicating with any other hosts. For instance, we have local-export(Replication(H), h) for all $h \in H$, and local-export(Commitment(h_p, h_v), h_p).

Where necessary, the Viaduct compiler removes these guard visibility constraints by multiplexing [69] conditional statements into straight-line code. This allows, for example, the compilation of conditionals with secret guards that require execution in MPC.

2.5.2 Cost of Protocol Assignments

There can be many valid protocol assignments that securely realize a source program. To select an optimal assignment, Viaduct attributes a cost to each assignment using an

$$
\begin{aligned}
\text{cost}(\textbf{let } r.x = e; s) &= \text{c}_{\text{exec}}(r, e) + \textstyle\sum_{r' \in \text{readers}(x,s)} \text{c}_{\text{comm}}(r, r) + \text{cost}(s) \\
\text{cost}(\textbf{if } t \textbf{ then } s_1 \textbf{ else } s_2) &= \max(\text{cost}(s_1), \text{cost}(s_2)) \\
\text{cost}(\mu X.s) &= W_{\text{loop}} \times \text{cost}(s) \\
\text{cost}(X) &= 0 \\
\text{cost}(s_1; s_2) &= \text{cost}(s_1) + \text{cost}(s_2) \\
\text{cost}(\textbf{skip}) &= 0
\end{aligned}
$$

Figure 2.18: Abstract cost model.

abstract cost model, shown in fig. 2.18. Developers can instantiate the abstract model by modifying the customizable *cost estimator*, which specifies $\text{c}_{\text{exec}}(r, s)$, the cost of executing statement s in protocol r; $\text{c}_{\text{comm}}(r_1, r_2)$, the cost of communicating between r_1 and r_2; and the global constant W_{loop}, the number of times a loop is assumed to execute when its iteration count is not statically known.

Our implementation configures c_{exec} to assign a small cost to executing "in the clear" and a large cost to the use of cryptography, so the compiler avoids the use of cryptography except when required for security. We also configure the communication cost c_{comm} to minimize data movement. For example, a frequently accessed public variable would be replicated on two hosts so that each host has a local copy. Placing the variable only on one of the hosts could reduce storage cost but entails frequently sending its value to the other host.

2.5.3 Computing an Optimal Protocol Assignment

To compute an optimal protocol assignment given a program s, the Viaduct compiler constructs a constrained optimization problem over the following sets of variables:

Assignment variables (α_i) These represent the protocols that execute **let**-bindings or declarations.

Cost variables (β_i) These represent the cost of executing **let**-bindings or declarations.

Participating host variables ($\gamma_{i,j}$) These are true if host j is participating in the execution of a statement i.

The compiler generates a set of constraints $\{\phi_1, \ldots, \phi_n\}$ over these assignment, cost, and participating host variables, as well as an expression β_s capturing the cost of s as in fig. 2.18. These constraints are drawn from a grammar consisting of logical connectives, an equality predicate between assignment variables and protocols, and an equality predicate between cost variables and cost expressions. The compiler uses an off-the-shelf constraint solver [36] to compute a variable assignment such that all constraints $\{\phi_1, \ldots, \phi_n\}$ are satisfied and β_s is minimized. Define $\mathrm{VA}(s)$ to be the set of valid protocol assignments for s, that is,

$$\mathrm{VA}(s) = \{s' \mid \epsilon \models s' \text{ and } s' \equiv s \text{ ignoring annotations}\}.$$

Then, this solution for the assignment variables corresponds to a protocol assignment s_{opt} such that

$$s_{\mathrm{opt}} = \operatorname*{arg\,min}_{s' \in \mathrm{VA}(s)} \mathrm{cost}(s').$$

Protocol Factory

To construct the optimization problem, the compiler draws the set of available protocols from the customizable *protocol factory*. Developers wishing to add new protocols to Viaduct must extend the protocol factory so that the compiler can generate assignments with these protocols during protocol selection.

The protocol factory maps each variable to a set of viable protocols that can execute the **let** statement for that variable. This allows developers to specify limitations regarding the use of particular protocols. For example, commitment protocols may be unable to compute over commitments. Other protocols may lack support for certain operators.

2.6 Runtime System

Once it has computed a protocol assignment, the Viaduct compiler outputs a program where every **let**-binding is annotated with the protocol that will execute it. This annotated program can be executed by the Viaduct runtime, which consists of an extensible interpreter that interacts with a set of *protocol back ends*, each of which implement a set of protocols. The interface for protocol back ends is straightforward: back ends must implement methods to execute **let**-binding, and methods to communicate with other protocol back ends.

Each host runs a copy of the interpreter with the annotated program as input. For each statement, the interpreter checks whether the host participates in its execution, as defined by hosts$(\cdot)$—if not, the statement is treated like **skip**. If a host participates in executing a **let**-binding, the interpreter calls the back end for the protocol assigned to the statement. To execute a conditional, the host retrieves the cleartext value of the guard from the protocol back end that stores it, and executes the appropriate branch. The validity rules for protocol assignments ensure the host is allowed to see the cleartext value, and that it is able to retrieve it.

2.6.1 Protocol Composition

The protocol back end executing a **let** statement must send the computed value to back ends executing statements that read the bound variable. How one back end sends a value to another depends on the protocols involved. For example, a statement executed in Replication(Alice, Bob) reading a variable computed in SH-MPC(Alice, Bob) corresponds to executing an MPC circuit and revealing the output to the hosts. On the other hand, a variable computed in Local(Chuck) might not meaningfully be read by a statement executed under SH-MPC(Alice, Bob) as it is unclear how the MPC back end should read local data from an unrelated host.

Viaduct uses the customizable *protocol composer* to define the set of source and destination protocols that can communicate. The composer translates communication between two protocols to a set of messages between hosts participating in the protocols. Developers who want to extend Viaduct with support for a new protocol must enumerate the set of allowed compositions for the protocol and ensure that such compositions are secure.

Formally, the protocol composer is given a source and a destination protocol r_1, r_2 and a host $h \in \text{hosts}(r_1) \cup \text{hosts}(r_2)$. It returns code that h must execute to perform the communication $r_1 \rightsquigarrow r_2$. Host h is allowed to send messages to and receive messages from the other hosts in $\text{hosts}(r_1) \cup \text{hosts}(r_2)$. The Viaduct runtime handles the delivery of these messages.

Recalling the example from before, when SH-MPC(Alice, Bob) sends a value to Replication(Alice, Bob), the MPC back ends at Alice and Bob jointly execute a circuit in an MPC protocol. The MPC back end at Alice then sends the revealed circuit output to the cleartext back end (which implements Replication$(\cdot)$) at Alice. There is a corre-

sponding message between the MPC and cleartext back ends at Bob. Step (3) in fig. 2.5, which depicts execution of the historical millionaires' problem, shows this protocol composition in the context of a larger program.

We describe a select subset of important compositions next. We write Cleartext(H) to denote either Local(h) or Replication(H), depending on whether H contains a single host or multiple hosts, respectively.

Cleartext(H_1) $\rightsquigarrow$ Cleartext(H_2) We wish to communicate a value replicated across one set of hosts to a different set of hosts while preserving security. We first look at some special cases.

Example 2.6.1. Replication(Alice, Bob) $\rightsquigarrow$ Local(Chuck). In order to preserve integrity, Alice and Bob *both* send the value to Chuck; Chuck checks that the two values are the same, and aborts if they differ. With this protocol, Alice and Bob must *both* be malicious to trick Chuck into accepting a corrupted value. If, for example, only Alice is malicious, Alice can cause Chuck to abort, but it cannot cause Chuck to exhibit incorrect behavior.

Example 2.6.2. Local(Alice) $\rightsquigarrow$ Replication(Bob, Chuck). Alice sends the value to Bob and to Chuck. Next, Bob sends the value it receives from Alice to Chuck, and Chuck does the same for Bob. Finally, Bob and Chuck verify that all values they received are the same, and abort otherwise. In this protocol, Bob and Chuck ensure Alice does not *equivocate*, that is, send different values to Bob and Chuck. Removing this check would be disastrous. For example, Bob and Chuck could diverge arbitrarily if the value is used as a guard in an **if** statement.

Example 2.6.3. Replication(Alice, Bob) $\rightsquigarrow$ Replication(Bob, Chuck). Since Bob already has the value, it does not need to receive it from Alice. Therefore, this

case is essentially the same as Replication(Alice, Bob) $\rightsquigarrow$ Local(Chuck): Alice and Bob both send the value to Chuck, and Chuck verifies that they are the same. Bob performs a local copy.

We are now ready to tackle the general case: Cleartext(H_1) $\rightsquigarrow$ Cleartext(H_2). Hosts that already store the value do not need to receive it (example 2.6.3), so define $H_r = H_2 \setminus H_1$, the set of hosts that need the value. Each host in H_1 sends the value to each host in H_r. Each host in H_r ensures all values they received from hosts H_1 are the same (example 2.6.1). Each host in H_r sends the value they have to every other host in H_r. Each host in H_r performs an equivocation check (example 2.6.2).

Local(h_1) $\rightsquigarrow$ SH-MPC(h_1, h_2) Host h_1 creates a secret input gate while h_2 creates a dummy input gate. We do not commit to how the back end implements secret inputs, but this could corresponds to h_1 creating a secret sharing of its value and sending one of the shares to h_2.

Replication(h_1, h_2) $\rightsquigarrow$ SH-MPC(h_1, h_2) Hosts h_1 and h_2 create cleartext input gates.

SH-MPC(h_1, h_2) $\rightsquigarrow$ Local(h_1) Both hosts create a secret output gate for h_1, and execute the MPC circuit. Host h_1 receives a cleartext output but h_2 does not.

SH-MPC(h_1, h_2) $\rightsquigarrow$ Replication(h_1, h_2) Both hosts create a public output gate and execute the MPC circuit. Both hosts receive a cleartext output.

Local(h_p) $\rightsquigarrow$ Commitment(h_p, h_v) Host h_p creates a commitment and sends it to h_v. Host h_p stores both the cleartext value and the information necessary to later open the commitment; h_v stores the commitment.

Commitment(h_p, h_v) $\rightsquigarrow$ Local(h_v) Host h_p sends to h_v the information necessary to open the previously created commitment. Host h_v opens the commitment and receives a cleartext value.

ZKP(h_p, h_v) $\rightsquigarrow$ Local(h_v) Host h_p sends to h_v a cleartext value and a zero-knowledge proof that the value is computed correctly.

The creation of a commitment and its opening; the execution of an MPC circuit and the revealing of its output; a prover sending a zero-knowledge proof to a verifier—all of these are captured by a composition of one protocol with another. This wide range of behaviors we can capture as composition illustrates our insight that protocol composition is a general abstraction to represent the use of cryptographic mechanisms.

2.7 Implementation

We implemented the Viaduct compiler in about 20 KLoC of Kotlin code, which includes code for the parser, the label constraint solver, protocol selection, and the runtime system. The code written against the compiler's extension points—the protocol factory, the protocol composer, the cost estimator, and the protocol back ends—runs to about 4 KLoC. Viaduct uses the Z3 SMT solver [36] to solve the optimization problem generated during protocol selection.

The compiler supports the more liberal surface syntax seen in figs. 2.2 and 2.3, as well as functions with bounded polymorphism on parameter labels. The compiler specializes functions based on label arguments to label polymorphic functions, allowing different compiled implementations for the same function.

We implemented four protocol back ends for Viaduct:

Local/Replication The cleartext back end executes code in Local and Replicated protocols. It maintains a store for objects that directly represent the temporaries

and assignables of the source program. Computations performed by the cleartext back end are executed directly.

SH-MPC This back end links Viaduct to ABY [38], a library for two-party semi-honest MPC. It maintains a store of gate objects that represent circuit components executed by ABY. Computations performed by the back end build gate objects that represent the operation performed (e.g., an addition in the source program creates an ADD gate).

The ABY framework supports execution of circuits in three different schemes—arithmetic sharing, boolean sharing, and Yao's garbled circuits—as well as conversions between these, allowing for execution of mixed-protocol circuits. Viaduct represents each scheme as a separate protocol, but all three are implemented by a single back end. To generate efficient mixed circuits, we follow Demmler et al. [38] and Ishaq et al. [55] and estimate inputs to the cost estimator by measuring execution time of individual operations under a particular scheme and conversions between schemes. We perform measurements for two settings: low-latency, high-bandwidth (LAN), and high-latency, low-bandwidth (WAN).[6] Thus the cost estimator has two modes, each of which optimizes compiled programs for a specific network environment.

Commitment This back end manages commitments, implemented using SHA-256 hashes of data along with a nonce. The back end for the commitment creator maintains a store of cleartext values along with metadata for commitments. The back end for the commitment receiver maintains the set of commitments, as hashes. The commitment back end cannot support computation.

[6]Existing work such as Büscher et al. [15] and Ishaq et al. [55] focus on optimizing mixed circuits for ABY specifically, and as such these employ more sophisticated reasoning about cost for ABY circuits. We consider it future work to incorporate such techniques into Viaduct.

ZKP This back end links to libsnark [89], a library for zkSNARKs (zero-knowledge Succinct Non-interactive ARguments of Knowledge). This back end maintains a store of circuit gate objects. The prover and verifier both manage cleartext values for the public inputs to the proof, while only the prover manages cleartext values for the secret inputs. To ensure the prover cannot modify secret inputs mid-execution, all secret inputs are "committed" by sending their hash to the verifier. All proofs that use a secret input then include a clause that equates the input to the pre-image of the hash held by the verifier.

The libsnark library requires proving and verifying keys to be generated for each unique circuit before the protocol is executed. The current prototype requires a "dummy" run of the compiled program to generate these keys.

2.8 Evaluation

To evaluate Viaduct, we address these research questions:

1. Is Viaduct expressive enough?
2. Is its compilation performance acceptable?
3. Does it generate efficient distributed programs?
4. How much does label inference reduce the annotation burden for programmers?
5. What is the overhead of the runtime system?

Experiments used Dell OptiPlex 7050 machines with an 8-core Intel Core i7 7th Gen CPU and 16 GB of RAM. Note that for experiments involving time measurements (items 2, 3 and 5), the numbers reported are over 5 trials and the relative standard error is at most 6% of the sample mean.

Benchmark	Description	LoC	Ann
battleship	Model of the board game	79	12
bet	C bets who wins hist. millionaires b/w A & B	79	7
biometric match	Min distance b/w sample & database [15]	40	8
guessing game	Same as in fig. 2.3	16	6
HHI score	Market concentration index [94]	22	3
historical millionaires	Same as in fig. 2.2 but with arrays	17	3
interval	A & B compute interval of combined points; C attests point is in interval	45	9
k-means	Cluster secret points from A & B [15]	82	3
k-means (unrolled)	k-means w/ 3 unrolled iterations	174	3
median	Median of A & B's lists [57]	36	6
rock-paper-scissors	A & B commit to moves then reveal	56	6
two-round bidding	A & B bid for a list of items	34	4

Table 2.1: Benchmark programs. **Ann** is the minimum number of label annotations needed to write the program.

	Protocols	Selection	
Benchmark	LAN / WAN	Vars	Time (s)
battleship	RZ / RZ	1022	1.0
bet	CLRY / CLRY	1022	1.0
biometric match	ALRY / ALRY	708	2.0
guessing game	RZ / RZ	193	0.4
HHI score	ALRY / LRY	285	1.1
historical millionaires	LRY / LRY	187	0.7
interval	RYZ / RYZ	660	2.8
k-means	ARY / RY	1684	7.9
k-means (unrolled)	ARY / RY	3629	29.0
median	RY / RY	386	1.0
rock-paper-scissors	CR / CR	741	1.0
two-round bidding	LRY / LRY	575	1.7

Table 2.2: Protocol selection for benchmark programs. **Protocols** give the protocols used in the compiled program for either the LAN or WAN setting. Legend for protocols used: **A, B, Y**–ABY arithmetic/boolean/Yao sharing; **C**–Commitment; **L**–Local; **R**–Replicated; **Z**–ZKP. **Selection** gives the number of symbolic variables and run time for protocol selection, averaged across five runs.

2.8.1 Expressiveness

Table 2.1 shows the benchmarks used for the experiments, and table 2.2 shows the cryptography synthesized by Viaduct for each benchmark. Several are from prior work, rewritten in the Viaduct source language. Host configurations are either semi-honest, as in fig. 2.2, where hosts A and B trust each other for integrity; mutually distrusting as in fig. 2.3; or are "hybrid" configurations where A and B trust each other but host C is trusted by neither.

Our benchmarks show that Viaduct can compile programs whose security demands a variety of cryptographic mechanisms. With hybrid configurations (interval, bet), Viaduct combines MPC and ZKP to implement different components of a single distributed program.

2.8.2 Scalability of Compilation

The two main phases of the Viaduct compiler are label inference and protocol selection. The overhead of label inference is negligible: at most several hundred milliseconds. As seen in table 2.2, the overhead for protocol selection is more significant, but still on the order of several seconds for most benchmarks. The longest running benchmark, k-means, performs most of its computations in MPC. In this case, it may be harder to converge to the optimal solution since the solver generates a large mixed circuit, choosing between the three MPC schemes supported by ABY.

Benchmark	Bool	YAO	Opt-LAN	Opt-WAN
biometic match	3.6	2.8	**1.0**	**1.0**
HHI score	0.8	0.5	**0.3**	**0.3**
historical millionires	1.0	0.6	**0.3**	**0.3**
k-means	56.5	44.4	**17.7**	44.4
median	11.5	12.8	**0.7**	**0.7**
2-R bidding	17.3	17.8	**3.1**	**3.1**

(a) Run time (in seconds) in the LAN setting.

Benchmark	Bool	YAO	Opt-LAN	Opt-WAN
biometic match	95.9	7.1	**2.2**	**2.2**
HHI score	9.7	1.6	1.1	**0.9**
historical millionires	90.6	1.6	**0.7**	**0.7**
k-means	696.1	117.4	**35.8**	117.4
median	1098.7	35.4	**31.7**	**31.7**
2-R bidding	184.7	184.5	**155.5**	**155.5**

(b) Run time (in seconds) in the WAN setting.

Benchmark	Bool	YAO	Opt-LAN	Opt-WAN
biometic match	56.0	52.3	**3.9**	**3.9**
HHI score	7.0	2.7	**0.5**	0.6
historical millionires	4.8	3.1	**0.005**	**0.005**
k-means	1273.1	1051.3	**180.0**	1051.3
median	197.1	327.8	**1.0**	**1.0**
2-R bidding	233.0	233.0	**4.7**	**4.7**

(c) Communication (in MB) in the LAN/WAN setting.

Table 2.3: Run time and communication of select benchmark programs, averaged across five runs. **Bool** and **Yao** are naive assignments using boolean sharing and Yao sharing, respectively. **Opt-LAN** and **Opt-WAN** are optimal assignments generated by Viaduct for the LAN and WAN setting, respectively. Best values are highlighted in **bold**.

2.8.3 Performance of Compiled Programs

To show that Viaduct can compile efficient distributed programs, we chose a subset of our benchmarks requiring the use of MPC and compared the execution of optimal programs generated by Viaduct—for each benchmark, one optimized for local area networks (LAN) and another for wide area networks (WAN)—with naive protocol assignments that perform all computation in MPC. The naive ABY assignments use either boolean sharing or Yao garbled circuits, since arithmetic sharing can only perform arithmetic operations. We measured executions in a 1 Gbit/s LAN and simulated WAN (100 Mbit/s bandwidth and 50 ms latency). We configured ABY to use 32-bit integers and set its security parameter to 128 bits.

Table 2.3 summarizes our results. For some benchmarks (HHI score, historical millionaires, median, two-round bidding), computation can be securely moved from MPC to cleartext protocols, making execution much more efficient. Even for benchmarks that require computations to be almost entirely in MPC (biometric match, k-means), Viaduct chooses efficient mixed circuits that perform much better than the naive assignments entirely in boolean sharing or Yao circuits. Viaduct replicates the result in Büscher et al. [15] (which specifically targets the ABY framework) in choosing a mix of arithmetic and Yao circuits as optimal assignments for the two benchmarks from that paper, with the exception of the k-means benchmark in the WAN setting.

2.8.4 Annotation Burden of Security Labels

Security-typed languages add some annotation burden when writing programs. In practice, host delegation assumptions and labels on downgrading operations suffice to specify intended security policies in Viaduct programs. To substantiate this claim, we

	LAN		WAN	
Benchmark	**Time (s)**	**Slowdown (%)**	**Time (s)**	**Slowdown (%)**
biometric match	0.4	150	1.5	50
HHI score	0.3	0	1.0	10
historical millionaires	0.3	0	0.7	0
k-means	1.2	1380	4.1	770
median	0.5	40	31.5	0
2-R bidding	1.6	90	154.7	0

Table 2.4: Run time of LAN-optimized benchmarks hand-written to use ABY directly and the slowdown of running the same benchmarks through the Viaduct runtime in LAN and WAN settings.

created two versions of each benchmark program. In one, every variable has a label annotation; in the other, "erased" version, all such labels are omitted.

For all benchmarks, Viaduct generates the same compiled program for the fully labeled and the erased versions. The **Ann** column in Table 2.1 counts label annotations on erased programs. This is the minimum number of annotations needed to write the program: effectively, the number of downgrades plus the number of delegations between hosts. The table shows that the annotation burden is low: most benchmarks need only a few label annotations.

2.8.5 Overhead of Runtime System

The Viaduct runtime introduces some overhead compared to using cryptographic libraries like ABY directly. To measure this overhead, we translated Viaduct's LAN-optimized outputs for the MPC benchmarks in table 2.3 to directly use the ABY frame-

work's API. We then measured the performance of these hand-written programs in the LAN and WAN settings.[7]

Table 2.4 gives running times for the hand-written programs and the overhead of using the Viaduct runtime. For most benchmarks, the Viaduct runtime incurs an overhead of at most 150% in the LAN setting; the overhead is reduced to at most 50% in the WAN setting where network delay is a more significant factor. This overhead is due to the cost of interpretation and dynamic circuit generation, and can be eliminated by moving circuit generation to compile time [66, 15].

The markedly larger overhead of the k-means benchmark is due to Viaduct recomputing intermediate results. The benchmark has 8 outputs; while Viaduct evaluates 8 smaller MPC circuits each with one output, the hand-written version evaluates one larger circuit with 8 outputs, taking advantage of shared intermediate computations. The compiler could, with additional analysis, determine when output gates can be grouped and executed in the same circuit. We leave this to future work.

2.9 Related Work

Compilation to Cryptographic Protocols

The idea of compiling a high-level program to a cryptographic protocol has been explored in the context of multiparty computation [51] (e.g., Fairplay [69], SCVM [65], ObliVM [66], OblivC [98], Wysteria [85], HyCC [15], SCALE-MAMBA [3]), and that of zero-knowledge proofs (e.g., Pinocchio [79], Geppetto [29], Buffet [95], xjSNARK [58]).

[7] Running LAN-optimized programs in the WAN setting does not skew the results since table 2.3 shows that LAN-optimized programs perform roughly the same as WAN-optimized programs in the WAN setting.

Earlier work is generally limited to the domain of a particular fixed cryptographic task (e.g., MPC or ZKP); Viaduct's novelty is synthesizing efficient protocols *across* cryptographic tasks. Like SCVM [65], Viaduct can synthesize "hybrid" programs that perform computations locally, replicated between hosts, or under MPC. This is impossible in the simple two-point label model that many MPC compilers [3, 66] use, which only distinguish between public (low) and secret (high) information. Viaduct also does not fix the number of hosts in a program (unlike [66, 65, 69]), nor fix compiling programs only under a semi-honest or malicious setting (unlike [85, 66, 65, 58, 95, 79]).

Program Partitioning

Another line of related work [100, 101, 43, 42] describes distributed computations using sequential programs and captures security requirements using information-flow labels. The Jif/split compiler [100, 101] synthesizes simple cryptographic primitives such as cryptographic commitments to satisfy security constraints that would otherwise be impossible without relying on trusted principals. Unlike Viaduct, Jif/split is not extensible to new protocols. Later work [43, 42] proves computational soundness for a similar system under a strong attacker that controls the network and some of the hosts. However, this work does not support replicating computations (only *data* replication is supported), or the other protocols that Viaduct supports.

CHAPTER 3

PROVABLE SECURITY

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, security critical applications employ sophisticated mechanisms such as complex distributed protocols [61, 21], trusted hardware [71, 49, 28], and advanced uses of cryptography including multiparty computation [69, 66, 3, 85], zero-knowledge proofs [79, 29, 95, 58], and homomorphic encryption [35, 34, 30].

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, and thus do not support secure combinations of mechanisms. On the other hand, compilers that perform *secure program partitioning* [100, 101, 25, 43, 42, 2] do combine mechanisms, but come with limited or informal correctness guarantees.

In this chapter, we present 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 [17], which establishes a modular foundation for cryptographic protocol security. 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 achieve a proof of end-to-end security.

Programming-language techniques for establishing simulation-based security are still in their infancy [63, 20, 44, 80]. Our security proof requires the incorporation of

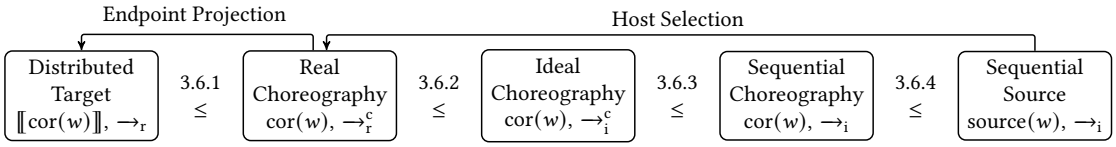


Figure 3.1: Overview of compilation and the correctness proof. Right-to-left arrows are compilation steps; $\leq$ are proof steps. Term w is a choreography, $[\![\cdot]\!]$ is endpoint projection, source$(\cdot)$ is the inverse of host selection, and cor$(\cdot)$ models corruption.

multiple techniques for simulation-based security: information-flow type systems [88] to define the security policy and guide partitioning, choreographies [73] to define global programs for distributed executions, and a novel information-flow guided technique for concurrent program sequentialization [8]. We make the following contributions:

- We formalize a variant of Simplified Universal Composability (SUC) [18] 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 as a sequence of type-preserving transformations between security-typed choreographies [73]. The source program models an idealized sequential execution on a single, trusted security domain, while the target program models 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.

```
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) Intermediate 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) Target distributed program derived by projecting choreography.

Figure 3.2: Compiling the Millionaires' Problem

3.1 Overview

Formalizing correctness requires being explicit about source, intermediate, and target languages, so we revisit the classic Millionaires' Problem [96] from section 2.1.

Figure 3.2a expresses the Millionaires' Problem as a source program in abstract syntax. As before, Alice and Bob engage in a protocol to 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). We use the original formulation of the problem instead of our "historical" variant, and drop the assumption that Alice and Bob trust each other for integrity, since these details are not relevant to our discussion.

3.1.1 Information Flow Control

Recall that our source language prevents insecure information flows through a security type system section 2.3.1. In our example, 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. The **endorse** expressions are necessary in this version of the problem since we drop the assumption that Alice and Bob trust each other for integrity.

Downgrade mechanisms require explicit *source* and *target* labels.[1] In fig. 3.2a, we suppress these labels for the **endorse** expressions, but show the **declassify** expression to move from $A \wedge B$ to $A \sqcap B$. The label $A \wedge B$ is too secret to allow Alice or Bob to see the value, while the label $A \sqcap B$ allows both parties to see the value.

3.1.2 Compilation

Given the source program, our compiler proceeds in two stages: first, *host selection* generates a *choreography* [73, 72, 32, 33, 53], a global program that represents a distributed system by making explicit statement placement and communication. In a choreography, each statement specifies a *host* where it is executed; and further, explicit *data movement* statements are used to transfer messages between hosts. Hosts may either represent *parties*, such as Alice and Bob, or *idealized functionalities* such as MPC(Alice, Bob), a (maliciously secure) multiparty computation protocol between Alice and Bob. Next, *endpoint projection* [73, 33] produces a distributed program, where each host in the choreography runs in parallel and interacts via message passing. Figure 3.1 depicts these compilation steps.

Figure 3.2b shows a choreography where Alice and Bob perform their respective **input** and **output** statements, while MPC(Alice, Bob) does the comparison. Additionally, the penultimate line has *synchronization* between Alice and Bob, which requires Bob to wait on an input from Alice before delivering his output. This synchronization step is necessary for the distributed program to match the sequential source program, where Bob's output happens after Alice's. Finally, fig. 3.2c shows the resulting distributed program corresponding to fig. 3.2b.

[1] The syntax we present in section 2.3 requires a single annotation that corresponds to the change in authority, and relies on inference to reconstruct source and target labels. We make these inferred labels explicit in our formalism.

3.1.3 Threat Model

Hosts can be *honest*, *semi-honest*, or *malicious*. Malicious hosts are fully controlled by the adversary; semi-honest hosts follow the protocol, but leak all their data to the adversary [64]. We say a host is *dishonest* if it is semi-honest or malicious, and *nonmalicious* if it is honest or semi-honest. The adversary controls all scheduling, even for honest hosts.

Deviating from section 2.1.4, we remove the assumption that channels between hosts are private. That is, the adversary can view all message *headers* (source and destination), even for messages between honest hosts and those involving the external environment. The adversary can only view message *content* when either the source or destination is dishonest. The adversary may not drop, duplicate, or modify messages between nonmalicious hosts. This abstraction of secure channels can be realized by standard techniques, such as TLS [39].

In our model as in most models of cryptographic protocols [17, 7], 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. Cryptographic applications must remove secret control flow through multiplexing [69], or other constant-time programming techniques [22, 9, 11, 93]. Supporting secret control flow requires significantly changing how attackers are modeled in cryptographic frameworks, which is beyond the scope of this work. However, we allow public control flow.

3.1.4 Correctness of Compilation

We formally guarantee secure compilation by drawing on the well established simulation paradigm from cryptography [64]: the target program is correct if attacks mounted by the adversary in the "real world" (the target program) can be *simulated* in the "ideal world" (source program). Simulation ensures that one only needs to reason about a weak adversary attacking the source program to understand possible real-world attacks by an adversary who may corrupt hosts and exploit concurrency by affecting scheduling.

We use transitivity of simulation to break the proof of correctness into simpler steps, depicted in fig. 3.1. Our proof follows compilation in reverse order: we first show the correctness of endpoint projection (distributed target programs simulate the choreography they are derived from), then we show the correctness of host selection (choreographies simulate their original source programs). Proving the correctness of endpoint projection allows reasoning using choreographies, which have useful properties. For instance, unlike arbitrary distributed programs, choreographies cannot have mismatched sends and receives, which means our proof need not consider such malformed programs.

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. We therefore split the correctness of host selection into intermediate steps, each dealing with a separate aspect. We first show choreographies simulate idealized choreographies, which handles the difference in the notion of security. Then, we show idealized choreographies simulate sequential choreographies, which shows that host selection inserts enough synchronization to preserve the sequential meaning of source programs. Finally, we show sequential choreographies simulate source programs, which is mainly

a bookkeeping step: choreographies have explicit host annotations for each statement and source programs do not.

Besides paving the way for end-to-end security, simulation also implies a well-studied notion of secure compilation, *robust hyperproperty preservation* (RHP) [1, 80]. Because the compiler satisfies RHP, security conditions (such as noninterference) satisfied by a source program automatically hold for the generated target program.

3.1.5 Outline

The rest of this chapter is structured as follows. Section 3.2 formalizes the syntax and semantics of our choreography language, section 3.3 details compilation steps, section 3.5 defines simulation based security, section 3.6 presents our proof, and section 3.7 briefly discusses how to connect our results to the UC framework.

3.2 Choreography Language

Compilation involves three languages: an interactive source language exposed to programmers (fig. 3.2a), an intermediate choreography language for high-level description of distributed systems (fig. 3.2b), and a target process language for implementing code running on each host (fig. 3.2c). We present a single unified syntax for all three languages; the source and target languages are subsets of the unified language.

Figure 3.3 gives the syntax of choreographies. The language supports an abstract set of values—which we assume includes 0—along with operators over them. We distinguish between pure, atomic expressions t, and expressions e that may have side effects. The

Variables $x \in \mathbb{X}$		Labels $\ell \in \mathbb{L}$		Hosts $h \in \mathbb{H}$
Values	v	$\in$	$\mathbb{V} \ni 0$	
Operators	f	$\in$	$\mathbb{F}$	
Atomic Expr.	t	::=	$v \mid x$	
Expressions	e	::=	$f(t_1, \ldots, t_n)$	
		$\mid$	$\textbf{declassify}(t, \ell_\mathrm{f} \to \ell_\mathrm{t})$	
		$\mid$	$\textbf{endorse}(t, \ell_\mathrm{f} \to \ell_\mathrm{t})$	
		$\mid$	$\textbf{input} \mid \textbf{output}\ t$	
		$\mid$	$\textbf{receive}\ h \mid \textbf{send}\ t\ \textbf{to}\ h$	‡
Statements	s	::=	$\textbf{let}\ h.x = e; s$	
		$\mid$	$h_1.t \rightsquigarrow h_2.x; s$	†
		$\mid$	$h_1[v] \rightsquigarrow h_2; s$	†
		$\mid$	$\textbf{if}\ h.t\ \textbf{then}\ s_1\ \textbf{else}\ s_2$	
		$\mid$	$\textbf{case}\ (h_1 \rightsquigarrow h_2)\ \{v \mapsto s_v\}_{v \in V}$	‡
		$\mid$	$\textbf{skip}$	
Channel Endpoints	c	$\in$	$\mathbb{C} = \{\mathrm{Adv}, \mathrm{Env}\} \cup \mathbb{H}$	
Buffers	B	$\in$	$\mathbb{C} \times \mathbb{C} \to \mathbb{V}^*$	
Processes	w	::=	$\langle H \subseteq \mathbb{H}, B, s \rangle$	
Configurations	W	::=	$w_1 \parallel \cdots \parallel w_n$	

Figure 3.3: Unified syntax of source, choreography, and target languages. † terms are choreography only, ‡ are target only.

declassify expression marks locations where private data is explicitly allowed to flow to public data. Similarly, **endorse** marks locations where untrustworthy data is explicitly allowed to influence trusted data. The **input**/**output** expressions allow programs to interact with the external *environment* [78, 27]. In contrast, **receive**/**send** expressions allow communicating with other *hosts*.

The **let** statement $\textbf{let}\ h.x = e; s$ represents performing the local computation e on host h, binding the result to variable x at h, and continuing as s. The computation can only refer to variables on h. The **if** statement represents conditional execution; like **let**, it names the host performing the computation. 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; intuitively, it represents a **send**/**receive** pair. The *selection* statement

$h_1[v] \rightsquigarrow h_2; s$ communicates control flow decisions, and is used to establish *knowledge of choice* [73, 72]. Finally, the **case** statement allows receiving a value from an external host and branching based on that value.

A process $\langle H, B, s \rangle$ is a choreography s along with an input buffer B and a set of hosts $H \subseteq \mathbb{H}$. Hosts H serve as the identifier for the process, and must contain all hosts mentioned in s. The buffer B stores incoming messages as a mapping from channels (pairs of endpoints $c_1 c_2$) to first-in-first-out queues of values. A *configuration* W is a set of *processes* composed in parallel (using operator $\|$). We require the processes in a configuration to have disjoint hosts.

Knowledge of Choice To understand why choreographies need selection statements $h_1[v] \rightsquigarrow h_2; s$, consider this choreography being executed by Alice and Bob:

> **let** Alice.x = **input**;
> **if** Alice.x **then** Alice$[1] \rightsquigarrow$ Bob; **let** Bob._ = **output** 1; **skip**
> **else** Alice$[0] \rightsquigarrow$ Bob; **skip**

Here, Alice branches on her local variable x, and within each branch, informs Bob of the branch taken using selection statements. Importantly, Bob performs an output in the **then** branch but not the **else** branch. Were the selection statements omitted, Bob could not determine whether to perform his output or not, rendering the choreography *unrealizable* as a distributed system.

Embedding Source and Target Programs Choreographies have a concurrent semantics where statements on different hosts can execute out of order. Source programs are special choreographies with a fully sequential semantics, and target programs represent a parallel composition of sequential choreographies.

$$\begin{array}{llll} \text{Messages} & m \in \mathbb{M} & ::= & c_1 c_2 v \\ \text{Actions} & a \in \mathbb{A} & ::= & ?m \mid !m \end{array}$$

$\boxed{\text{actor}(a) = c}$

$$\text{actor}(?c_1 c_2 v) = c_2 \qquad\qquad \text{actor}(!c_1 c_2 v) = c_1$$

Figure 3.4: Syntax of messages and actions.

Source programs are statements s that do not contain **receive**/**send** expressions, nor global communication, selection, or **case** statements. Source programs can be lifted to configurations consisting of a single process $\langle \{*\}, \epsilon, s \rangle$, where $*$ is the single locus of ideal program execution and ϵ is the initial, empty buffer.

Target programs are statements s_i for each host h_i. Target statements do not use global communication or selection statements.

$$\langle h_1, B_1, s_1 \rangle \parallel \langle h_2, B_2, s_2 \rangle \parallel \cdots \parallel \langle h_n, B_n, s_n \rangle$$

3.2.1 Operational Semantics

We give operational semantics to choreographies using labeled transition systems [73]. First, we define the syntax actions, then we define the transition relations.

Figure 3.4 gives the syntax of messages and actions. An action a is either the input $?m$ or 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. We define $\text{actor}(a)$ 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.

Following fig. 3.1, we define two transition relations: *ideal* stepping $\xrightarrow{a}_{\mathrm{i}}$ gives meaning to source programs and idealized choreographies, and *real* stepping $\xrightarrow{a}_{\mathrm{r}}$ gives meaning to target programs and choreographies that represent target programs. Additionally, we lift ideal and real stepping to concurrent versions, written $\xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{i}}$ and $\xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{r}}$, to capture the concurrent semantics of choreographies. We detail these relations next.

Ideal Semantics

Figure 3.5 defines the *ideal* stepping rules for source programs, given by the stepping relation $\rightarrow_{\mathrm{i}}$. To define the semantics of programs, we define the semantics of expressions and statements individually, and then lift these semantics generically to processes and configurations in 3.2.1.

For expressions, 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: they map a list of values of any size to a value. Partial operators (like division) can be made total using defaults. Formally, we give meaning to operator application assuming a denotation function $\mathrm{eval} : \mathbb{F} \times \mathbb{V}^* \rightarrow \mathbb{V}$. 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 secrecy/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 program and idealized choreographies, **receive**/**send** expressions are only used to communicate with malicious hosts; therefore, they take internal steps and always return 0.

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

$$\textsc{e-Operator}\ \frac{v = \mathrm{eval}(f, v_1, \ldots, v_n)}{h.f(v_1, \ldots, v_n) \xrightarrow{!hh0}_{\mathrm{i}} v} \qquad \textsc{e-Declassify}\ \frac{\ell_{\mathrm{f}} \notin \mathcal{P} \qquad \ell_{\mathrm{t}} \in \mathcal{P}}{h.\mathbf{declassify}(v, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) \xrightarrow{!h\mathrm{Adv}v}_{\mathrm{i}} v}$$

$$\textsc{e-Declassify-Skip}\ \frac{\ell_{\mathrm{f}} \in \mathcal{P} \vee \ell_{\mathrm{t}} \notin \mathcal{P}}{h.\mathbf{declassify}(v, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) \xrightarrow{!hh0}_{\mathrm{i}} v} \qquad \textsc{e-Endorse}\ \frac{\ell_{\mathrm{f}} \notin \mathcal{T} \qquad \ell_{\mathrm{t}} \in \mathcal{T}}{h.\mathbf{endorse}(v, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) \xrightarrow{?\mathrm{Adv}hv'}_{\mathrm{i}} v'}$$

$$\textsc{e-Endorse-Skip}\ \frac{\ell_{\mathrm{f}} \in \mathcal{T} \vee \ell_{\mathrm{t}} \notin \mathcal{T}}{h.\mathbf{endorse}(v, \ell_{\mathrm{f}} \to \ell_{\mathrm{t}}) \xrightarrow{!hh0}_{\mathrm{i}} v} \qquad \textsc{e-Input}\ \frac{\mathbb{L}(h) \in \mathcal{T}}{h.\mathbf{input} \xrightarrow{?\mathrm{Env}hv}_{\mathrm{i}} v} \qquad \textsc{e-Input-Malicious}\ \frac{\mathbb{L}(h) \notin \mathcal{T}}{h.\mathbf{input} \xrightarrow{!hh0}_{\mathrm{i}} 0}$$

$$\textsc{e-Output}\ \frac{\mathbb{L}(h) \in \mathcal{T}}{h.\mathbf{output}\ v \xrightarrow{!h\mathrm{Env}v}_{\mathrm{i}} 0} \qquad \textsc{e-Output-Malicious}\ \frac{\mathbb{L}(h) \notin \mathcal{T}}{h.\mathbf{output}\ v \xrightarrow{!hh0}_{\mathrm{i}} 0} \qquad \textsc{e-Receive}\ \frac{}{h.\mathbf{receive}\ h' \xrightarrow{!hh0}_{\mathrm{i}} 0}$$

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

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

$$\textsc{s-Let}\ \frac{h.e \xrightarrow{a}_{\mathrm{i}} v}{\mathbf{let}\ h.x = e; s \xrightarrow{a}_{\mathrm{i}} s[v/x]} \qquad \textsc{s-Communicate}\ \frac{}{h_1.v \rightsquigarrow h_2.x; s \xrightarrow{!h_1h_10}_{\mathrm{i}} s[v/x]} \qquad \textsc{s-Select}\ \frac{}{h_1[v] \rightsquigarrow h_2; s \xrightarrow{!h_1h_10}_{\mathrm{i}} s}$$

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

Figure 3.5: Ideal stepping rules for expressions and statements.

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

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

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

$$h.\mathbf{receive}\ h' \xrightarrow{?h'hv}_{\mathrm{r}} v \; e\text{-RECEIVE-REAL} \qquad h.\mathbf{send}\ v\ \mathbf{to}\ h' \xrightarrow{!hh'v}_{\mathrm{r}} 0 \; e\text{-SEND-REAL}$$

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

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

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

Figure 3.6: Real stepping rules for expressions and statements. These override the rules in fig. 3.5.

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.

Real Semantics

To give semantics to target programs, fig. 3.6 defines real stepping rules, $\rightarrow_{\mathrm{r}}$. These rules are defined by appropriate modifications of our ideal stepping rules (fig. 3.5). The **declassify**/**endorse** expressions always step internally instead of communicating with Adv. The **receive**/**send** expressions communicate a value with the specified host. Communication and selection statements step with a visible action instead of internally.

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

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

$$\frac{s_1 \xrightarrow{a}{}^{\mathrm{c}}_{\alpha} s_1' \qquad s_2 \xrightarrow{a}{}^{\mathrm{c}}_{\alpha} s_2' \qquad \mathrm{actor}(a) \neq h}{\textbf{if } h.t \textbf{ then } s_1 \textbf{ else } s_2 \xrightarrow{a}{}^{\mathrm{c}}_{\alpha} \textbf{if } h.t \textbf{ then } s_1' \textbf{ else } s_2'}\ \textsc{s-If-Delay}$$

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

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

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

$$\mathrm{hosts}(h_1[v] \rightsquigarrow h_2; [\cdot])\{h_1, h_2\}$$

Figure 3.7: Concurrent lifting of ideal/real stepping rules.

Finally, the **case** statement, on receiving a value on the expected channel, steps to the specified branch.

Concurrent Lifting for Choreographies

Figure 3.7 lifts an underlying statement-stepping judgment ($\rightarrow_{\mathrm{i}}$ or $\rightarrow_{\mathrm{r}}$) to a *concurrent* judgment, written $s \xrightarrow{a}{}^{\mathrm{c}}_{\alpha} s'$. 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 [73].

The rules refer to *evaluation contexts*, which are statements containing a single hole, and the function hosts$(\cdot)$, which returns the set of all 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. This matches the behavior of target programs where code running on a single host is single-threaded.

Synchronous vs. Asynchronous Choreographies In delay rules, requiring only the *actor* to be missing from the context leads to an *asynchronous* semantics [31]. In a *synchronous* setting, the side condition would require both endpoints to be missing: $\text{hosts}(a) \cap \text{hosts}(E) = \emptyset$. Consider the following program:

$$\textbf{let } \text{Alice}.x_1 = \textbf{input};$$
$$\text{Bob}.0 \rightsquigarrow \text{Alice}.x_2;$$

Here, Alice is waiting on an input from the environment, 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 ahead of the first.

Processes and Configurations

Figure 3.8 gives stepping rules for buffers, processes, and configurations. All rules in this figure are parameterized by an underlying stepping relation $s \longrightarrow_\alpha s'$, where $\alpha \in \{\text{i}, \text{r}\}$ may either refer to ideal or real stepping rules. 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

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

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

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

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

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

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

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

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

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

W-Input
$$\frac{\forall i \,.\, w_i \xrightarrow{?m}_\alpha w_i'}{w_1 \parallel \cdots \parallel w_n \xrightarrow{?m}_\alpha w_1' \parallel \cdots \parallel w_n'}$$

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

Figure 3.8: Stepping rules for buffers, processes, and configurations.

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. A configuration takes an input step by forwarding the input to all processes in the configuration. When a process in the configuration takes an output step, the output is fed to all other processes, and becomes the output of the configuration.

$\boxed{\text{source}(s) = s'}$

$$\text{source}(\textbf{let } h.x = e; s) = \begin{cases} \textbf{let } h.x = e; \text{source}(s) & \text{I/O}(e) \\ \textbf{let } *.x = e; \text{source}(s) & \neg\, \text{I/O}(e) \end{cases}$$
$$\text{source}(h_1.t \rightsquigarrow h_2.x; s) = \text{source}(s)[t/x]$$
$$\text{source}(h_1[v] \rightsquigarrow h_2; s) = \text{source}(s)$$
$$\text{source}(\textbf{if } h.t \textbf{ then } s_1 \textbf{ else } s_2) = \textbf{if } *.t \textbf{ then } \text{source}(s_1) \textbf{ else } \text{source}(s_2)$$
$$\text{source}(\textbf{skip}) = \textbf{skip}$$

$$\text{I/O}(e) = (e = \textbf{input}) \vee (\exists t \textbf{ . } e = \textbf{output } t)$$

Figure 3.9: Canonical source program from a choreography.

3.3 Compilation

Compilation consists of two stages: *host selection* turns a source program into a choreography, and *endpoint projection* turns a choreography into a target program.

3.3.1 Host Selection

This first compilation stage converts a source program into a choreography. Instead of committing to a specific host selection algorithm, we give validity criteria for the output of host selection in the form of an information-flow type system and a synchronization checking judgment. This generalizes our results beyond the algorithm we present in section 2.5.

Because a source program can be realized as many different choreographies, host selection cannot be modeled as a function from source programs to choreographies. Instead, we formalize the validity of host selection by considering a mapping from choreographies to source programs.

Definition 3.3.1 (Valid Host Selection)**.** Choreography s' is a valid result of host selection on source program s if $\text{source}(s') = s$, $\epsilon \vdash s'$, and $\Delta \Vdash s'$ for some Δ.

Figure 3.9 defines the function $\text{source}(\cdot)$, 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 judgement $\Gamma \vdash s$ denotes that choreography s has secure information flows, and $\Delta \Vdash s$ denotes s is well-synchronized. We define these judgments next.

Information-Flow Checking

First, we give a type system for choreographies based on information-flow control [100, 101, 25, 2] which validates that its corresponding host selection is secure.

Figure 3.10 gives the typing rules. A label context Γ maps a variable to its host and label:

$$\text{Label Contexts} \quad \Gamma \quad ::= \quad \epsilon \mid \Gamma, x : h.\ell$$

Expressions are checked at a particular host h with the judgment $\Gamma \vdash e : h.\ell$, which means that e has label ℓ under 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) [23] by requiring source and target labels to be uncompromised [97, 23]. 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 section 3.6.2.

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

$$\ell\text{-Value}\ \frac{}{\Gamma \vdash v : h.\ell} \qquad \ell\text{-Variable}\ \frac{\ell' \sqsubseteq \ell}{\Gamma, x : h.\ell' \vdash x : h.\ell} \qquad \ell\text{-Operator}\ \frac{\forall i \,.\, \Gamma \vdash t_i : h.\ell}{\Gamma \vdash f(t_1, \ldots, t_n) : h.\ell}$$

$$\ell\text{-Declassify}\ \frac{\Gamma \vdash t : h.\ell_f \qquad \ell_f^{\leftarrow} = \ell_t^{\leftarrow} \qquad \blacktriangledown \ell_f \qquad \blacktriangledown \ell_t \qquad \ell_t \sqsubseteq \ell}{\Gamma \vdash \mathbf{declassify}(t, \ell_f \rightarrow \ell_t) : h.\ell} \qquad \ell\text{-Endorse}\ \frac{\Gamma \vdash t : h.\ell_f \qquad \ell_f^{\rightarrow} = \ell_t^{\rightarrow} \qquad \blacktriangledown \ell_f \qquad \blacktriangledown \ell_t \qquad \ell_t \sqsubseteq \ell}{\Gamma \vdash \mathbf{endorse}(t, \ell_f \rightarrow \ell_t) : h.\ell} \qquad \ell\text{-Input}\ \frac{\mathbb{L}(h) \sqsubseteq \ell}{\Gamma \vdash \mathbf{input} : h.\ell}$$

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

$\boxed{\Gamma \vdash s}$

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

$$\ell\text{-Select}\ \frac{\mathbb{L}(h_1)^{\leftarrow} \sqsubseteq \mathbb{L}(h_2)^{\leftarrow} \qquad \Gamma \vdash s}{\Gamma \vdash h_1[v] \rightsquigarrow h_2; s} \qquad \ell\text{-If}\ \frac{\Gamma \vdash t : h.\mathbf{0}^{\leftarrow} \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \mathbf{if}\ h.t\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2} \qquad \ell\text{-Skip}\ \frac{}{\Gamma \vdash \mathbf{skip}}$$

Figure 3.10: Information-flow typing rules for expressions and statements.

In choreographies, **receive**/**send** expressions model communication with malicious hosts. Choreographies exclude code for malicious hosts, as these hosts exhibit arbitrary behavior; thus, we must approximate labels for **receive**/**send** expressions. 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.

$\boxed{\Delta \Vdash s}$

$$\textsc{Sync-External}\ \frac{\text{external}(e) \qquad \text{reset}(\Delta, h) \Vdash s \qquad \text{outputting}(e) \implies \text{synched}(\Delta, h)}{\Delta \Vdash \textbf{let}\ h.x = e; s}$$

$$\textsc{Sync-Internal}\ \frac{\text{internal}(e) \qquad \Delta \Vdash s}{\Delta \Vdash \textbf{let}\ h.x = e; s} \qquad \textsc{Sync-Communicate}\ \frac{\text{sync}(\Delta, h_1 \rightsquigarrow h_2) \Vdash s}{\Delta \Vdash h_1.t \rightsquigarrow h_2.x; s} \qquad \textsc{Sync-Select}\ \frac{\text{sync}(\Delta, h_1 \rightsquigarrow h_2) \Vdash s}{\Delta \Vdash h_1[v] \rightsquigarrow h_2; s}$$

$$\textsc{Sync-If}\ \frac{\Delta \Vdash s_1 \qquad \Delta \Vdash s_2}{\Delta \Vdash \textbf{if}\ h.t\ \textbf{then}\ s_1\ \textbf{else}\ s_2} \qquad \textsc{Sync-Skip}\ \frac{}{\Delta \Vdash \textbf{skip}}$$

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

$$\text{synched}(\Delta, h) = \forall h' \,.\, \Delta(h', h) \sqsubseteq \mathbb{L}(h') \vee \mathbb{L}(h)$$

$$\text{reset}(\Delta, h) = \Delta[h, * := \mathbf{1}][h, h := \mathbb{L}(h)]$$

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

Figure 3.11: Checking that a concurrent program acts like a sequential program.

Statement checking rules have the form $\Gamma \vdash s$; they are largely standard [88], 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, than 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.

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, consider the following choreography:

$$\textbf{let } \text{Alice}.x_1 = \textbf{endorse } x_{\text{guess}};$$
$$\textbf{let } \text{Bob}.x_2 = \textbf{declassify } x_{\text{secret}};$$

The **endorse** corresponds to the adversary committing to a guess, while the **declassify** corresponds to the program revealing a secret to the adversary. Even though the **endorse** is before the **declassify** in program order, these expressions are on different hosts and therefore may be reordered during execution. That is, the adversary may schedule Bob ahead of Alice, thus learning the secret before committing to a guess.

To enforce program order of these two statements, we *synchronize* the two hosts via communication, inserting the statement $\text{Alice}.0 \rightsquigarrow \text{Bob}.\_$; between the **endorse** and **declassify**. In turn, Bob only steps after receiving 0 from Alice, which Alice only does after performing the **declassify**.

Synchronization becomes more complex when taking *corruption* into account. For example, if Alice and Bob synchronize through another host h ($\text{Alice} \rightsquigarrow h \rightsquigarrow \text{Bob}$) 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 3.11 defines the synchronization-checking judgment $\Delta \Vdash s$. Here, the context Δ is an adjacency matrix $\Delta(h_1, h_2)$ mapping pairs of hosts to integrity labels. Intuitively, a choreography is well-synchronized when for any *external* (input or output) expression e, there exists a high-integrity communication path from e to all *output* expressions

that come after e in the program order.[2] The integrity of a communication path $h_1 \rightsquigarrow h_2 \rightsquigarrow \ldots \rightsquigarrow h_n$ can be computed from the hosts along the path:

$$\mathbb{L}(h_1 \rightsquigarrow h_2 \rightsquigarrow \ldots \rightsquigarrow h_n) = \mathbb{L}(h_1)^{\leftarrow} \vee \mathbb{L}(h_2)^{\leftarrow} \vee \cdots \vee \mathbb{L}(h_n)^{\leftarrow}.$$

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:

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

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

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. Additionally, if e is an output expression, we ensure h is synchronized with all hosts by checking the following condition, which ensures that if neither h_1 nor h_2 is malicious, there exists a communication path from h_1 to h_2 that could not have been influenced by the adversary:

$$\mathbb{L}(\text{paths}(h_1, h_2)) \sqsubseteq \mathbb{L}(h_1) \vee \mathbb{L}(h_2). \tag{3.1}$$

Rules Sync-Communicate and Sync-Select update Δ using $\text{sync}(\Delta, h_1 \rightsquigarrow h_2)$. The function captures the fact that if there is a path from some h to h_1, now there is a path from h to h_2 that goes through h_1. Additionally, all existing paths are still valid.

$\boxed{s \mathbin{@} H = s'}$

$$\mathbf{let}\ h.x = e; s \mathbin{@} H = \begin{cases} \mathbf{let}\ h.x = e; s \mathbin{@} H & h \in H \\ s \mathbin{@} H & \text{o/w} \end{cases}$$

$$h_1.t \rightsquigarrow h_2.x; s \mathbin{@} H = \begin{cases} h_1.t \rightsquigarrow h_2.x; s \mathbin{@} H & h_1, h_2 \in H \\ \mathbf{let}\ h_1.\_ = \mathbf{send}\ t\ \mathbf{to}\ h_2; s \mathbin{@} H & h_1 \in H \\ \mathbf{let}\ h_2.x = \mathbf{receive}\ h_1; s \mathbin{@} H & h_2 \in H \\ s \mathbin{@} H & \text{o/w} \end{cases}$$

$$h_1[v] \rightsquigarrow h_2; s \mathbin{@} H = \begin{cases} h_1[v] \rightsquigarrow h_2; s \mathbin{@} H & h_1, h_2 \in H \\ \mathbf{let}\ h_1.\_ = \mathbf{send}\ v\ \mathbf{to}\ h_2; s \mathbin{@} H & h_1 \in H \\ \mathbf{case}\ (h_1 \rightsquigarrow h_2)\ \{v \mapsto s \mathbin{@} H\} & h_2 \in H \\ s \mathbin{@} H & \text{o/w} \end{cases}$$

$$\mathbf{if}\ h.t\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \mathbin{@} H = \begin{cases} \mathbf{if}\ h.t\ \mathbf{then}\ s_1 \mathbin{@} H\ \mathbf{else}\ s_2 \mathbin{@} H & h \in H \\ \text{merge}(s_1 \mathbin{@} H, s_2 \mathbin{@} H) & \text{o/w} \end{cases}$$

$$\mathbf{skip} \mathbin{@} H = \mathbf{skip}$$

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

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

Figure 3.12: Endpoint projection and select merge rules.

3.3.2 Endpoint Projection

The second compilation stage, *endpoint projection*, is a standard procedure in choreographic programming for extracting a distributed system from a choreography [72, 32]. Given a choreography s and a host h, the endpoint projection $[\![s]\!]_h$ defines the local

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

program that h should run. The distributed system is derived by projecting onto each host in the choreography.

To define endpoint projection, we first define an auxiliary function in fig. 3.12, $s @ H$, *generalized projection*, for projecting statements onto a *set* of hosts. This extra generality allows modeling malicious corruption in section 3.5.1. Our definition matches the standard notion of endpoint projection [73] when the set of hosts is a singleton set.

The high-level idea of projecting onto the set of hosts H is as follows. If H contains *all* hosts involved in a statement, the statement stays as is. This is the first case for **let**, communication, and selection statements. If H contains *none* of the hosts involved, the statement is removed entirely. This is the last case for **let**, communication, and selection statements. Otherwise, H contains *some* of the involved hosts, and we project based on the role of H. Communication statements become either a **send** or a **receive**, depending on whether H contains the sending host or the receiving host. Selection statements are projected as a **send** expression, or as a **case** statement with a single branch.

The most interesting case is for **if** statements. If the host performing the **if** statement is in H, then we perform the **if** as usual and project the branches. Otherwise, hosts H do not store the conditional and cannot determine which branch should be taken. In this case, we require the projections of the two branches to be compatible with each other, which is formalized using 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 our proof is agnostic to the details. We only rely on the soundness and completeness of endpoint projection, which is extensively studied in prior work [73, 72, 32, 33, 53]. Additionally, we lift projection to processes. Projecting a buffer onto H keeps only the messages

destined for H, and projecting a process is done component-wise:

$$(B \mathbin{@} H)(c_1 c_2) = \begin{cases} B(c_1 c_2) & c_1 \notin H \wedge c_2 \in H \\ \epsilon & \text{otherwise} \end{cases}$$

$$\langle H', B, s \rangle \mathbin{@} H = \langle H \cap H', B \mathbin{@} H, s \mathbin{@} H \rangle.$$

Finally, we define endpoint projection $[\![w]\!]_h$ as projecting onto a single host:

$$[\![w]\!]_h = w \mathbin{@} \{h\}.$$

The *projection* $[\![w]\!]$ of process w is the configuration formed by projecting onto each host in w:

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

3.4 Properties of the Language

3.4.1 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 3.4.1 (Robust Declassification)**.** *If* $\Gamma \vdash \mathbf{declassify}(t, \ell_f \rightarrow \ell_t) : h.\ell$, $\ell_f \notin \mathcal{P}$, *and* $\ell_t \in \mathcal{P}$, *then* $\ell_f \in \mathcal{T}$.

Lemma 3.4.2 (Transparent Endorsement)**.** *If* $\Gamma \vdash \mathbf{endorse}(t, \ell_f \rightarrow \ell_t) : h.\ell$, $\ell_f \notin \mathcal{T}$, *and* $\ell_t \in \mathcal{T}$, *then* $\ell_f \in \mathcal{P}$.

Typing has standard properties.

Definition 3.4.3 (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 3.4.4 (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 3.4.5 (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 3.4.6 (Type Preservation)**.** *If* $\Gamma \vdash s$ *and* $s \xrightarrow{a} s'$, *then* $\Gamma \vdash s'$.

Lemma 3.4.7 (Robust Typing)**.** *If* $\Gamma \vdash s$, *then* $\Gamma \vdash \mathrm{cor}(s)$.

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

Lemma 3.4.8 (Synchrony Preservation)**.** *If* $\Delta \Vdash s$ *and* $s \xrightarrow{a} s'$, *then* $\Delta \Vdash s'$.

Lemma 3.4.9 (Robust Synchrony)**.** *If* $\Delta \Vdash s$, *then* $\Delta \Vdash \mathrm{cor}(s)$.

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

Lemma 3.4.10 (Output Synchronization)**.** *If* $\Delta \Vdash s$ *and* $s \xrightarrow{!m}$ *with* m *external, then* $\mathrm{synched}(\Delta, h)$.

3.4.2 Operational Semantics

Below, we write $\rightarrow$ to stand for any of $\rightarrow_{\mathrm{i}}$, $\rightarrow_{\mathrm{r}}$, $\rightarrow_{\mathrm{i}}^{\mathrm{c}}$, or $\rightarrow_{\mathrm{r}}^{\mathrm{c}}$.

Processes never refuse input.

Lemma 3.4.11 (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 3.4.12 (Internal Determinism). *If $w \xrightarrow{a} w_1$ and $w \xrightarrow{a} w_2$, then $w_1 = w_2$.*

Lemma 3.4.13 (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.

3.5 Simulation

Simulation determines when a configuration W_1 securely realizes configuration W_2: that is, if *every* adversary $\mathcal{A}$ interacting with W_1 can be simulated by *some* simulator $\mathcal{S}$ (of equal power) running against W_2 [17]. The simulator must make the two systems indistinguishable to *any* external environment. Since we compare source and target languages defined by differing operational semantics, our definition of simulation is between *pairs* of configurations and their respective operational semantics.

Definition 3.5.1 (Simulation)**.** $\langle W_1, \rightarrow_1 \rangle \leq \langle W_2, \rightarrow_2 \rangle$ (W_1 simulates W_2 under semantics $\rightarrow_1$ and $\rightarrow_2$) if

$$\forall \mathcal{A} \,.\, \exists \mathcal{S} \,.\, \mathbb{T}_{\text{Env}}(\langle \mathcal{A} \parallel W_1, \rightarrow_1 \rangle) = \mathbb{T}_{\text{Env}}(\langle \mathcal{S} \parallel W_2, \rightarrow_2 \rangle).$$

Here, $\mathbb{T}_{\text{Env}}(\langle W, \rightarrow \rangle)$ is the set of *traces*, $tr = a_1, \ldots, a_n$, generated by W restricted to communication with the environment:

$$\mathbb{T}_{\text{Env}}(\langle W, \rightarrow \rangle) = \{ tr_{|\text{Env}} \mid W \xrightarrow{tr} \}.$$

Restriction $tr_{|\text{Env}}$ removes all actions in tr where neither the source nor the destination is Env.

An adversary $\mathcal{A}$ or $\mathcal{S}$ is an arbitrary process given via a labeled transition system; the system uses the same actions a, but need not use our syntax for programs. The rules for running an adversary in parallel with a configuration are the same as rules W-Input and W-Output.

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 3.5.2 (Adversary Interface)**.** For all $\mathcal{A}$:

1. If $\mathbb{L}(c_1) \in \mathcal{S} \cap \mathcal{T}$ and $\mathbb{L}(c_2) \in \mathcal{S} \cap \mathcal{T}$, then $\mathcal{A} \xrightarrow{?c_1 c_2 v_1} \mathcal{A}'$ if and only if $\mathcal{A} \xrightarrow{?c_1 c_2 v_2} \mathcal{A}'$ for all v_1 and v_2.
2. If $\mathcal{A} \xrightarrow{!c_1 c_2 v}$, then either $c_1 = \text{Adv}$ or $\mathbb{L}(c_1) \notin \mathcal{T}$.

Our definition of simulation guarantees *perfect* (i.e., information-theoretic) security. In section 3.7, we discuss how to transfer our results to the SUC framework.

3.5.1 Modeling Malicious Hosts

In target programs, we entirely remove processes that correspond to malicious hosts, and allow the adversary to act in their stead. To reason about corruption in choreographies, we use the generalized projection function $w \mathbin{@} H$ to alter the choreography corresponding to the set of malicious hosts. Intuitively, if a host is malicious, we remove it from the choreography, as the adversary forges messages for it; and if the choreography has a global communication statement $h_1.t \rightsquigarrow h_2.x; s$ where h_1 is honest but h_2 is corrupt (or vice versa), then we turn the statement into a **send** (resp. **receive**) to communicate with the adversary. Formally, we corrupt the choreography by projecting along the set of nonmalicious hosts:

Definition 3.5.3 (Corrupted Choreography)**.** $\mathrm{cor}(w)$ is defined as

$$\mathrm{cor}(w) = w \mathbin{@} H \quad \text{where} \quad H = \{h \in \mathbb{H} \mid \mathbb{L}(h) \in \mathcal{T}\}.$$

Removing statements from a choreography to model corruption might seem odd, but this perfectly mirrors removing malicious hosts from target programs. Note that corrupted choreographies only show up in intermediate proof steps, in particular, we do not apply $\mathrm{cor}(\cdot)$ to source programs (refer back to fig. 3.1). This means end-to-end security can be stated and understood without reference to $\mathrm{cor}(\cdot)$, and more importantly, an incorrect definition cannot invalidate our results (it can only make our proof fail).

3.6 Correctness of Compilation

In this section, we prove our main result of simulation-based security for our compilation process. 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 $\mathrm{source}(w) = \langle H, B, \mathrm{source}(s)\rangle$.

Theorem 3.6.1. *If $\epsilon \vdash w$, and $\Delta \Vdash w$ for some Δ, then*

$$\langle [\![\mathrm{cor}(w)]\!], \rightarrow_{\mathrm{r}} \rangle \leq \langle \mathrm{source}(w), \rightarrow_{\mathrm{i}} \rangle.$$

From definition 3.5.1, it is immediate that simulation is transitive. Therefore, we can prove theorem 3.6.1 through a series of intermediate simulations, following fig. 3.1 from left to right. First, in section 3.6.1, we show that endpoint projection is correct, which allows reasoning using the choreography instead of the distributed program. Next, in section 3.6.2, we move from the real semantics $\rightarrow^{\mathrm{c}}_{\mathrm{r}}$ to the ideal semantics $\rightarrow^{\mathrm{c}}_{\mathrm{i}}$, which limits all adversarial corruption to **declassify** and **endorse** expressions. Then, we show in section 3.6.3 that our synchronization judgment ensures all externally visible actions happen in program order. Finally, in section 3.6.4, we prove that the choreography simulates the source program.

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 [76, 52] between the real world (adversary running against real configuration) and the ideal world (simulator running against ideal configuration).

3.6.1 Correctness of Endpoint Projection

First, we show that the result of partitioning a choreography simulates the choreography.

Theorem 3.6.2. *If $\epsilon \vdash w$, then $\langle [\![\mathrm{cor}(w)]\!], \rightarrow_{\mathrm{r}} \rangle \leq \langle \mathrm{cor}(w), \rightarrow^{\mathrm{c}}_{\mathrm{r}} \rangle$.*

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 $\mathcal{S} = \mathcal{A}$. The

choreographic programming literature [73, 72, 32, 33, 53] 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 3.6.3. *Define $W_1 \; R \; w_2$ if $W_1 = [\![w_2]\!]$. We claim R is* not *a bisimulation.*

Proof. Consider the following choreography and its projection:

// *Choreography*

$w_2 = \textbf{if}\ \text{Alice}.1\ \textbf{then}\ \text{Alice}[\text{Bob}] \rightsquigarrow 1; s_1\ \textbf{else}\ \text{Alice}[\text{Bob}] \rightsquigarrow 0; s_2$

// Alice

$[\![w_2]\!]_{\text{Alice}} = \textbf{if}\ \text{Alice}.1\ \textbf{then send}\ 1\ \textbf{to}\ \text{Bob}; [\![s_1]\!]_{\text{Alice}}\ \textbf{else send}\ 0\ \textbf{to}\ \text{Bob}; [\![s_2]\!]_{\text{Alice}}$

// 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

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

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 3.4.12 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 [73], and it does *not* break bisimilarity; we only need to be smarter about how we define R.

The solution to the issue raised by lemma 3.6.3 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 [73], 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. Following Montesi [73], the proof is split into showing the *soundness* and *completeness* of endpoint projection.

Lemma 3.6.4 (Soundness of Endpoint Projection)**.** *If* $W \succeq [\![w]\!]$ *and* $w \xrightarrow{a}_{\mathrm{r}}^{\mathrm{c}} w'$*, then* $W \xrightarrow{a}_{\mathrm{r}} W'$ *for some* $W' \succeq [\![w']\!]$.

Lemma 3.6.5 (Completeness of Endpoint Projection)**.** *If* $W \succeq [\![w]\!]$ *and* $W \xrightarrow{a}_{\mathrm{r}} W'$*, then* $w \xrightarrow{a}_{\mathrm{r}}^{\mathrm{c}} w'$*, for some* w' *with* $W' \succeq [\![w']\!]$.

$$
\begin{array}{rrcl}
\text{Statements} & s & ::= & \ldots \\
& & \mid & h_1 \rightsquigarrow h_2[v/x]; s \\
& & \mid & h_1 \rightsquigarrow h_2[v]; s
\end{array}
$$

Figure 3.13: The syntax of asynchronous choreographies (extends fig. 3.3).

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

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 3.14: Stepping rules for asynchronous choreographies. These override the rules for $\longrightarrow_{\mathrm{r}}$ in fig. 3.6.

Proving soundness and completeness is largely standard, except for the modifications needed to handle malicious corruption and asynchronous communication. Handling malicious corruption does not significantly affect the proof since we make coordinated changes to both the projected program and the choreography. To maintain the correspondence between projected programs and choreographies in the presence of malicious corruption, we remove processes running on malicious hosts from projected programs, and remove statements on malicious hosts in choreographies, both by using $\mathrm{cor}(\cdot)$. This transformation allows us to ignore malicious hosts, so we can invoke the soundness and completeness results from the literature, except on a smaller set of hosts.

Handling Asynchronous Communication

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 in the choreography reduces in only one. We follow prior work [31, 83] and add syntactic forms to choreographies

to represent partially reduced **send**/**receive** pairs (i.e., messages that have been sent and buffered but not yet received), given in fig. 3.13. These run-time terms only exist to restore perfect correspondence, and are never written by the programmer.

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 Alice.$v \rightsquigarrow$ Bob.$x; s$ becomes a **send** on Alice and a **receive** on Bob, Alice $\rightsquigarrow$ 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 3.14 gives the updated rules.

These run-time terms are sufficient to restore perfect correspondence, and make lemmas 3.6.4 and 3.6.5 go through. We refer to Cruz-Filipe and Montesi [31] for details.

Lemma 3.6.6. *If* $\epsilon \vdash w$, *then* $\langle [\![\mathrm{cor}(w)]\!], \longrightarrow_{\mathrm{r}} \rangle \leq \langle \mathrm{cor}(w), \longrightarrow_{\mathrm{a}}^{\mathrm{c}} \rangle$.

Proof. Let $\mathcal{S} = \mathcal{A}$. Lemmas 3.6.4 and 3.6.5 immediately give a strong bisimulation. □

Restoring Original Choreography Syntax

Lemma 3.6.6 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 later proof steps.

Lemma 3.6.7. *If* $\epsilon \vdash w$, *then* $\langle \mathrm{cor}(w), \longrightarrow_{\mathrm{a}}^{\mathrm{c}} \rangle \leq \langle \mathrm{cor}(w), \longrightarrow_{\mathrm{r}}^{\mathrm{c}} \rangle$.

Proof. The simulator follows the control flow by maintaining a public view of the extended choreography $\langle \mathrm{cor}(w), \longrightarrow_{\mathrm{a}}^{\mathrm{c}} \rangle$ 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 3.6.2. By lemmas 3.6.6 and 3.6.7 using the transitivity of UC simulation. □

3.6.2 Correctness of Ideal Execution

Next, we show that a choreography stepping with the real rules (fig. 3.6) simulates itself stepping with the ideal rules (fig. 3.5).

Theorem 3.6.8. *If* $\epsilon \vdash w$, *then* $\langle \mathrm{cor}(w), \longrightarrow_{\mathrm{r}}^{\mathrm{c}} \rangle \leq \langle \mathrm{cor}(w), \longrightarrow_{\mathrm{i}}^{\mathrm{c}} \rangle$.

The main difference between the two semantics is the interface with the adversary. With the real semantics, dishonest hosts actively leak data to the adversary (through **send** expressions and communication statements), and the adversary controls all data coming from malicious hosts (through **receive** expressions). In contrast, the ideal semantics performs all adversarial interaction via **declassify** and **endorse** expressions. In effect, the ideal semantics causes all leakage and corruption to become much more coarse grained. Additionally, by eliminating all blocking **receive** expressions (which communicate with the adversary), the ideal semantics is able to make progress in a manner independent of the adversary; this aids our sequentialization proof in section 3.6.3.

Consider choreography w and its representation $\mathrm{cor}(w)$ when Alice is malicious:

$// \, w$	$// \, \mathrm{cor}(w)$
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 $\mathrm{cor}(\cdot)$ 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

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

$$\frac{\ell_f \notin \mathcal{P} \qquad \ell_t \in \mathcal{P}}{h.\mathbf{declassify}(v, \ell_f \to \ell_t) \xrightarrow{?h\text{Adv}v'}_{\text{sim}} v'} \; e\text{-Declassify-Simulator}$$

$$\frac{\ell_f \notin \mathcal{T} \qquad \ell_t \in \mathcal{T}}{h.\mathbf{endorse}(v, \ell_f \to \ell_t) \xrightarrow{!\text{Adv}hv'}_{\text{sim}} v} \; e\text{-Endorse-Simulator}$$

Figure 3.15: Stepping rules used internally by the simulator. These override fig. 3.6.

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 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. 3.15, 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 3.6.9 (Closing Substitution)**.** A closing substitution $\sigma : \Gamma$ is a mapping from variables to values $\sigma : \text{dom}(\Gamma) \to \mathbb{V}$.

Definition 3.6.10 (Channel Label)**.** A channel's label derives from the labels of its endpoints:

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

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

Definition 3.6.11 (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 3.6.11 with $L = \mathcal{P}$ for agreement on public values, and with $L = \mathcal{T}$ for agreement on trusted values. Definition 3.6.11 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 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 3.6.11 is that s_1 and s_2 are equivalent if x can be treated as secret/untrusted. To check that, definition 3.6.11 abstract 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 3.6.11 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 $\mathbb{L}(h) \not\in \mathcal{T}$ (this is ensured by $\mathrm{cor}(\cdot)$).[3]

[3]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 3.6.12 (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* $\mathrm{actor}(!m_2) = \mathrm{actor}(!m_1)$.

Proof. By definition 3.6.11, 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

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

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

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

  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 3.4.5, $\sigma_1(s') = s_1''[v/x] = s_1'$, 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 3.6.11, 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 3.6.17.
- Case e-Declassify-Skip. Same as the case for rule e-Operator.
- Case e-Endorse-Real. Deliberately excluded; handled by lemma 3.6.18.
- 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 2.2.1.

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

$$s_1 \xrightarrow{!h_1h_2v_1}_{\mathrm{r}} s_1''[v_1/x] \qquad\qquad s_2 \xrightarrow{!h_1h_2v_2}_{\mathrm{r}} s_2''[v_2/x].$$

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

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

We case on t. If $t = v$ for some v, then $v_1 = \sigma_1(v) = v = \sigma_2(v) = v_2$. Additionally, $\Gamma \vdash s''[v/x]$ by lemma 3.4.5, $\sigma_1(s''[v/x]) = s_1''[v/x]$, 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 3.6.11, 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

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

$\Gamma \vdash s'$ (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

$$s_1 = \textbf{if } h.v_1 \textbf{ then } s_1^1 \textbf{ else } s_1^2 \qquad s_2 = \textbf{if } h.v_2 \textbf{ then } s_2^1 \textbf{ else } s_2^2$$

$$s = \textbf{if } h.t \textbf{ then } s^1 \textbf{ else } s^2$$

and

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

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 2.2.1), 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

$$\frac{s''_1 \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{r}} s'''_1 \qquad \mathrm{actor}(a) \notin \mathrm{hosts}(E_1)}{s_1 \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{r}} E_1[s'''_1]} \qquad \frac{s''_2 \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{r}} s'''_2 \qquad \mathrm{actor}(a) \notin \mathrm{hosts}(E_1)}{s_2 \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{r}} E_2[s'''_2]}$$

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 3.6.13 (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 3.6.11, 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

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

  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

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

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

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

We case on t. If $t = v$ for some v, then $v_1 = \sigma_1(v) = v = \sigma_2(v) = v_2$. 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 $\mathbb{L}(h_1 h_2) \notin \mathcal{P}$, $a_1 = !h_1 h_2 v_1 =_{\mathcal{P}} !h_1 h_2 v_2 = a_2$.

- 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

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

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 $a_1 = !h_1 h_2 v = a_2$, 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 3.6.14 (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 3.6.13. The only other relevant case is under rule *s*-Let with rule *e*-Output. The argument is similar to the case in lemma 3.6.13, 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 3.6.15 (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 $\longrightarrow^{\mathrm{c}}_{\mathrm{r}}$ *and* $\longrightarrow^{\mathrm{c}}_{\mathrm{sim}}$ *reversed (excluding rules e-*Declassify-Simulator *and e-*Endorse-Simulator *instead).*

Proof. Follows immediately from lemmas 3.6.12 and 3.6.13 since $\longrightarrow^{\mathrm{c}}_{\mathrm{sim}}$ is equivalent to $\longrightarrow^{\mathrm{c}}_{\mathrm{r}}$ 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 3.6.16 (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 $\longrightarrow^{\mathrm{c}}_{\mathrm{r}}$ *and* $\longrightarrow^{\mathrm{c}}_{\mathrm{i}}$ *reversed (excluding rules e-*Declassify *and e-*Endorse *instead).*

Proof. Follows from lemmas 3.6.12 and 3.6.14. judgment $\longrightarrow^{\mathrm{c}}_{\mathrm{r}}$ behaves the same as $\longrightarrow^{\mathrm{c}}_{\mathrm{r}}$ except it replaces some output messages with internal steps. Since this does not affect the resulting choreographies (only the actions), lemma 3.6.12 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 3.6.17 (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 3.6.11, 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 3.6.15. The only remaining case is when the steps are by rules *e*-Declassify, *e*-Declassify-Real and *e*-Declassify-Simulator, respectively. We have

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

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

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

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 3.4.1 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 3.4.5), $\sigma_1(s'_{\mathrm{p}}) = s'_1$, 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 3.4.5), $\sigma'_2(s'_{\mathrm{t}}) = s'_2$, and $\sigma_3(s'_{\mathrm{t}}) = s'_3$. □

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

Lemma 3.6.18 (Equivalence After Endorse)**.** *Let $s_1 =_{\mathcal{P}} s_2$ and $s_1 =_{\mathcal{T}} s_3$. If $s_1 \xrightarrow{!hh0}{}^{\mathrm{c}}_{\mathrm{r}} s'_1$, $s_2 \xrightarrow{!\mathrm{Adv}hv}{}^{\mathrm{c}}_{\mathrm{sim}} s'_2$, and $s_3 \xrightarrow{?\mathrm{Adv}hv}{}^{\mathrm{c}}_{\mathrm{i}} s'_3$, then $s'_1 =_{\mathcal{P}} s'_2$ and $s'_1 =_{\mathcal{T}} s'_3$.*

Proof. Dual to lemma 3.6.17, but focusing on $s_1 =_{\mathcal{P}} s_2$ and using lemma 3.4.2. □

Lemmas 3.6.15 and 3.6.16 straightforwardly lift from choreographies s to processes w. Lemma 3.6.15 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 3.6.8. 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.[4]

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 3.6.15 ensures condition (2) is preserved, and lemma 3.6.16 ensures condition (3) is preserved. Lemmas 3.6.17 and 3.6.18 cover the cases with downgrades. □

3.6.3 Correctness of Sequentialization

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

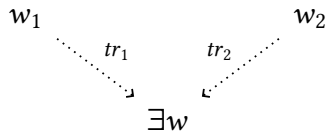
Theorem 3.6.19. *If* $\epsilon \vdash w$, *and* $\Delta \Vdash w$ *for some* Δ, *then* $\langle \mathrm{cor}(w), \rightarrow_{\mathrm{i}}^{\mathrm{c}} \rangle \leq \langle \mathrm{cor}(w), \rightarrow_{\mathrm{i}} \rangle$.

[4]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.

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* [77, 26, 6], which ensures joinable processes remain joinable throughout the simulation.

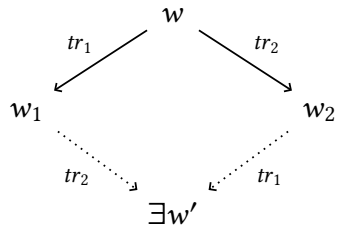
Definition 3.6.20 (Joinable Processes)**.** We write $w_1 \downarrow w_2$ if there exist traces tr_1 and tr_2 containing only internal actions such that $w_1 \xrightarrow{tr_1}{}^{\mathrm{c}}_{\mathrm{i}}\ w$ and $w_2 \xrightarrow{tr_2}{}^{\mathrm{c}}_{\mathrm{i}}\ w$ for some w. Diagrammatically:



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

Definition 3.6.21 (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 3.6.22 (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 3.6.22 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 3.6.23 (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 3.6.24 (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 3.6.23. 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 3.6.22. 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 3.6.24 finishes the proof. □

The proof of lemma 3.6.22 reasons generically about buffers, and appeals to a diamond lemma for statements in a black-box manner. This means we can generalize

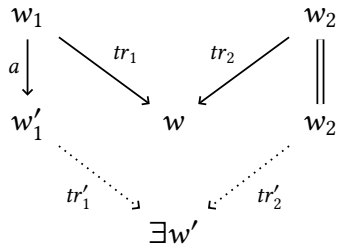
lemma 3.6.22 to arbitrary (combinations of) stepping relations without extra work as long as a diamond property for the same relations holds for statements.

Lemma 3.6.25 (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 3.6.22. □

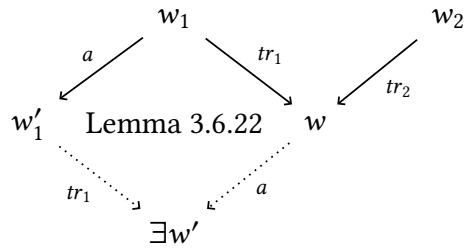
Processes remain joinable after taking internal or matching steps.

Lemma 3.6.26 (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:*



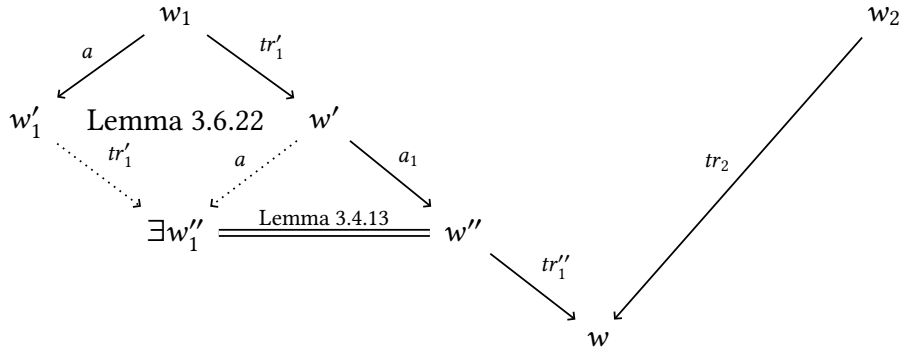
Proof. Since w_1 and w_2 are joinable, there exist w and internal tr_1, tr_2 such that $w_1 \xrightarrow{tr_1}{}^{\mathrm{c}}_{\mathrm{i}} w$ and $w_2 \xrightarrow{tr_2}{}^{\mathrm{c}}_{\mathrm{i}} w$. We case on whether $a \perp\!\!\!\perp tr_1$.

- Case $a \perp\!\!\!\perp tr_1$. Lemma 3.6.22 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 $tr_2 \cdot a$.



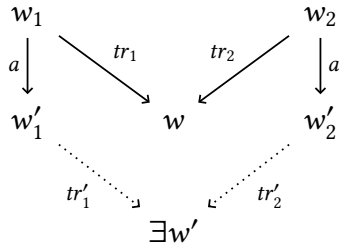
- Case $a \not\perp\!\!\!\perp tr_1$. Let a_1 be the first action in tr_1 such that $a \not\perp\!\!\!\perp a_1$, that is, $tr_1 = tr_1' \cdot a_1 \cdot tr_1''$ with $a \perp\!\!\!\perp tr_1'$. We have, $w_1 \xrightarrow{tr_1'}{}^{\mathrm{c}}_{\mathrm{i}} w' \xrightarrow{a_1}{}^{\mathrm{c}}_{\mathrm{i}} w'' \xrightarrow{tr_1''}{}^{\mathrm{c}}_{\mathrm{i}} w$. Lemma 3.6.22 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 3.4.13 gives $w_1'' = w''$.[5] Finally, $w_1' \downarrow w_2$ through $tr_1' \cdot tr_1''$ and tr_2.

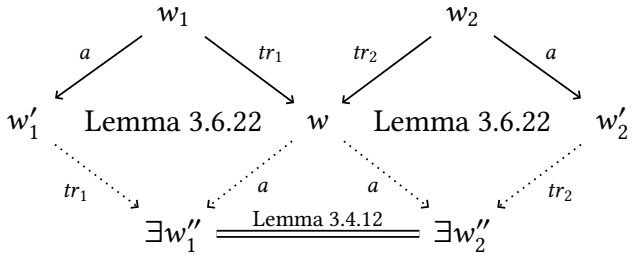


□

Lemma 3.6.27 (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 3.6.26. Otherwise, $a \perp\!\!\!\perp tr_1$ and $a \perp\!\!\!\perp tr_2$. The result follows from two applications of lemma 3.6.22, and one application of lemma 3.4.12:



[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 3.4.13 applies.

□

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 $w \xrightarrow{!m}\!\!\!\!\rightarrow_{\mathrm{i}} w'$ if $w \xrightarrow{tr \cdot !m}_{\mathrm{i}} w'$ for some internal tr.

Lemma 3.6.28 (Sequential Execution)**.** *If* $\epsilon \vdash w$, $\Delta \Vdash w$, *and* $w \xrightarrow{!m}_{\mathrm{i}}^{\mathrm{c}}$ *for* m *external, then* $w \xrightarrow{!m}\!\!\!\!\rightarrow_{\mathrm{i}}$.

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 context E. Note that the top statement in E must be internal, otherwise we get a contradiction by lemma 3.4.10. Since w has no free variables, it can take an internal step. □

Lemma 3.6.29 (Step Over)**.** *If* $w_1 \xrightarrow{a_1}_{\mathrm{i}}^{\mathrm{c}} w_2 \xrightarrow{a_2}_{\mathrm{i}}$, *then either* $w_1 \xrightarrow{a_1}_{\mathrm{i}} w_2$, *or* $w_1 \xrightarrow{a_2}_{\mathrm{i}}$ *and* $a_1 \perp\!\!\!\perp a_2$.

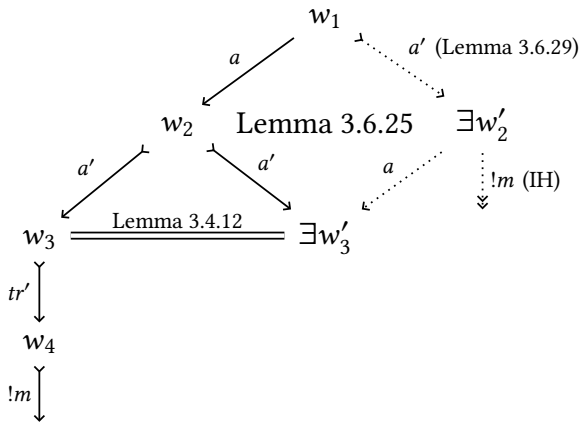
Proof. Assume the first step cannot be done sequentially (otherwise we are done). Then, the first step uses a delay rule (rules *s*-Delay and *s*-If-Delay), which ignores the head statement of w_1. The second step is sequential so it only depends on the head statement of w_2, which is the same as the 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 3.6.30 (Delayed Step)**.** *If* $w_1 \xrightarrow{a}_{\mathrm{i}}^{\mathrm{c}} w_2 \xrightarrow{!m}\!\!\!\!\rightarrow_{\mathrm{i}}$ *for* a *internal, then* $w_1 \xrightarrow{!m}\!\!\!\!\rightarrow_{\mathrm{i}}$.

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

- Case $tr = \epsilon$. We have $w_1 \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{i}} w_2 \xrightarrow{!m}{}_{\mathrm{i}}$. Lemma 3.6.29 gives $w_1 \xrightarrow{!m}{}_{\mathrm{i}}$, and thus $w_1 \xrightarrow{!m}{}\!\!\!\!\rightarrow_{\mathrm{i}}$.
- Case $tr = a' \cdot tr'$. We have $w_1 \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{i}} w_2 \xrightarrow{a'}{}_{\mathrm{i}} w_3 \xrightarrow{tr'}{}_{\mathrm{i}} w_4 \xrightarrow{!m}{}_{\mathrm{i}}$. Lemma 3.6.29 gives $w_1 \xrightarrow{a'}{}_{\mathrm{i}} w_2'$ for some w_2'. Lemma 3.6.25 with lemma 3.6.23 then gives $w_1 \xrightarrow{a'}{}_{\mathrm{i}} w_2' \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{i}} w_3'$ for some w_3', and lemma 3.4.12 shows $w_3' = w_3$. We use the induction hypothesis on $w_2' \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{i}} w_3 \xrightarrow{tr'}{}_{\mathrm{i}} w_4 \xrightarrow{!m}{}_{\mathrm{i}}$ to get $w_2' \xrightarrow{!m}{}\!\!\!\!\rightarrow_{\mathrm{i}}$, and combined with $w_1 \xrightarrow{a'}{}_{\mathrm{i}} w_2'$, we get $w_1 \xrightarrow{!m}{}\!\!\!\!\rightarrow_{\mathrm{i}}$.

  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 3.6.31 (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 $w_1 \xrightarrow{!m}{}^{\mathrm{c}}_{\mathrm{i}}$ for m external, then $w_2 \xrightarrow{!m}{}\!\!\!\!\rightarrow_{\mathrm{i}}$.*

Proof. Since w_1 and w_2 are joinable, there exist w and internal tr_1, tr_2 such that $w_1 \xrightarrow{tr_1}{}^{\mathrm{c}}_{\mathrm{i}} w$ and $w_2 \xrightarrow{tr_2}{}^{\mathrm{c}}_{\mathrm{i}} w$. Because m is external and tr_1 is internal, we have $!m \perp\!\!\!\perp tr_1$, so lemma 3.6.22 gives $w \xrightarrow{!m}{}^{\mathrm{c}}_{\mathrm{i}}$. Now we have $w_2 \xrightarrow{tr_2 \cdot !m}{}^{\mathrm{c}}_{\mathrm{i}}$, and we want to show $w_2 \xrightarrow{!m}{}\!\!\!\!\rightarrow_{\mathrm{i}}$. We proceed by induction on tr_2. When tr_2 is empty, lemma 3.6.28 completes the proof.

When $tr_2 = a \cdot tr_2'$, the induction hypothesis gives $w_2 \xrightarrow{a}{}^{\mathrm{c}}_{\mathrm{i}} w_2' \xrightarrow{!m}{}^{*}_{\mathrm{i}}$, and lemma 3.6.30 gives the desired result. □

Proof of theorem 3.6.19. 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 3.6.31 guarantees the sequential program can perform the output. The primary invariant, that the concurrent and sequential processes remain joinable, is ensured by lemmas 3.6.26 and 3.6.27.

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 3.6.31 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 3.6.26 and 3.6.27 ensure condition (4) is preserved. Lemmas 3.4.6 and 3.4.8 ensure condition (5) is preserved. □

3.6.4 Correctness of Host Selection

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

Theorem 3.6.32. *If* $\epsilon \vdash w$, *then* $\langle \mathrm{cor}(w), \rightarrow_{\mathrm{i}} \rangle \leq \langle \mathrm{source}(w), \rightarrow_{\mathrm{i}} \rangle$.

We break the result into simpler simulations. First, we recover statements removed by $\mathrm{cor}(\cdot)$.

Lemma 3.6.33. *If* $\epsilon \vdash w$, *then* $\langle \mathrm{cor}(w), \rightarrow_{\mathrm{i}} \rangle \leq \langle w, \rightarrow_{\mathrm{i}} \rangle$.

Proof. Expressions on malicious hosts only generate internal actions: stepping rules in fig. 3.5 ensure **input**/**output** expressions step internally; typing rules in fig. 3.10 ensure **declassify**/**endorse** expressions step internally. This means $\mathrm{cor}(w)$ and w have the same external behavior, except w takes extra internal steps. The simulator follows the control flow and acts the same as the adversary, but whenever the adversary schedules $\mathrm{cor}(w)$, the simulator schedules w multiple times until the head statement is at a nonmalicious host, then it schedules w once more.

A small caveat: in $\mathrm{cor}(w)$, 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 $\mathrm{cor}(w)$ and w agree on *trusted* values (formalized using $=_{\mathcal{T}}$). □

Next, we remove host annotations and explicit communication.

Lemma 3.6.34. *If* $\epsilon \vdash w$, *then* $\langle w, \longrightarrow_{\mathrm{i}} \rangle \leq \langle \mathrm{source}(w), \longrightarrow_{\mathrm{i}} \rangle$.

Proof. Statements removed by $\mathrm{source}(\cdot)$ only produce internal actions, which the simulator can recreate. Host annotations do not affect program behavior beyond changing the source and destination of internal actions and actions generated by **declassify**/**endorse** expressions; 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 $\mathrm{source}(w)$ only if the statement is preserved by $\mathrm{source}(\cdot)$; 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. □

Proof of theorem 3.6.32. Immediate from lemmas 3.6.33 and 3.6.34. □

3.7 Instantiating Cryptographic Hosts

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., MPC(Alice, Bob)); thus, to achieve a full end-to-end security proof, these idealized hosts must be securely instantiated with cryptographic subprotocols (e.g., BGW [12] for multiparty computation). Such an instantiation would imply UC security for all compiled programs, in contrast to existing formalization efforts for individual protocols [20, 10, 44].

To this end, we show how distributed protocols arising from compilation correspond to *hybrid* protocols in the Simplified Universal Composability (SUC) framework [18]. Then, we show how to take advantage of the *composition* theorem in SUC to obtain secure instantiations of cryptographic protocols.

Simplified UC Let s be a choreography with partitioning $[\![s]\!]$. We construct a corresponding SUC protocol $[\![s]\!]^{\mathrm{SUC}}$ 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 MPC(Alice, Bob). Local hosts map onto SUC parties, while idealized hosts map onto *ideal functionalities* in SUC.

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

Additionally, each wrapper needs to explicitly model *corruption*. In our framework, corruption is captured by labels: if host h is semi-honest ($\mathbb{L}(h) \in \mathcal{P} \cap \mathcal{T}$), then the

wrapper for h allows the adversary to query h for its current message transcript so far. Similarly, if h is malicious ($\mathbb{L}(h) \notin \mathcal{T}$), then 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.

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.

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.[6]

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.

3.7.1 Secure Instantiation of Cryptography

To securely instantiate cryptographic mechanisms, we appeal to the *composition* theorem in SUC, which states that ideal SUC-functionalities $\mathcal{F}$ may be substituted for SUC protocols that securely realize $\mathcal{F}$. To obtain a concrete cryptographic protocol, we may iterate the composition theorem to apply it 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 [18, 51] and Zero-Knowledge Proofs (ZKP) [54]. The main difference between our model and SUC is that our model allows the adversary to corrupt ideal functionalities (both semi-honestly and maliciously), while functionalities in SUC 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, the label of MPC(Alice, Bob) indicates that MPC(Alice, Bob) is semi-honest (resp. malicious) only if *both* Alice and Bob are semi-honest (resp. malicious).

[6]The dummy adversary theorem [17] implies that security against non-probabilistic adversaries guarantees security against probabilistic adversaries.

Thus, any power the adversary gains in corrupting MPC(Alice, Bob) can be instead achieved using Alice and Bob alone. Similar security concerns for label-based host selection have been discussed for Viaduct [2]. We can formalize this intuition via a simulation of the form $\langle W, \rightarrow_{\mathsf{i}} \rangle \leq \langle W, \rightarrow'_{\mathsf{i}} \rangle$, where W uses MPC(Alice, Bob), and $\rightarrow'_{\mathsf{i}}$ is modified so that corruption of MPC(Alice, Bob) is impossible.

3.8 Related Work

Secure Program Partitioning

Prior work on secure program partitioning [92, 100, 101, 2] focuses largely on the engineering effort on compiling security-typed source programs to distributed implementations with the aid of cryptography. Our compilation model and type system are closest to that of Viaduct [2], because we also approximate security guarantees of cryptographic mechanism with information-flow labels. The purpose of the present work is to give formal guarantees to such compilers.

A long line of work [43, 42, 5, 62, 60, 59] focuses on enforcing computational noninterference for information-flow typed programs by using standard cryptographic mechanisms, such as encryption. In contrast, our compiler enjoys *simulation-based security*, which guarantees preservation of *all* hyperproperties. Computational noninterference gives limited security guarantees in the presence of downgrading mechanisms. In contrast, we give novel security guarantees to declassifications and endorsements.

Liu et al. [65] 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 [17], Reactive Simulatability [7], and Constructive Cryptography [70], allow modular proofs of distributed cryptographic protocols. Additionally, Liao et al. [63] give a core language for formalizing UC protocols. Since we abstract away cryptographic mechanisms using ideal hosts, we do not explicitly model many subtleties of these systems, such as probability, computational complexity, and applications of cryptographic hardness assumptions. We expect our results to naturally be compatible with these frameworks.

Prior verification efforts [20, 68, 10] show simulation-based security for concrete cryptographic mechanisms. We develop simulation-based security for compiler correctness, showing the correctness of an entire *family* of protocols (those generated by the compiler), rather than an individual protocol. However, we appeal to such existing correctness proofs for individual mechanisms when instantiating idealized hosts, making our work orthogonal.

Secure Compilation

The standard notions of compiler correctness in the literature are derived from full abstraction and hyperproperty preservation [1]. Patrignani et al. [81, 80] 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.

There is an extensive literature on choreographies [73, 72, 32, 33, 53], but the primary concern in this literature is proving deadlock freedom, and very little prior work considers security [67]. We extend choreographies with an information-flow type system, and show how to model semi-honest and malicious corruption, which is novel.

CHAPTER 4

CONCLUSION

This dissertation presents Viaduct, a compiler that translates high-level, security-typed programs into efficient distributed programs that employ a variety cryptographic mechanisms to ensure security, and a general methodology for proving such compilers correct. The Viaduct compiler and its correctness proof are both agnostic to the set of available cryptographic mechanisms, making them easily extensible. Our simulation-based security result guarantees the compiler preserves *arbitrary* hyperproperties of source programs in the compiled protocol, allowing developers to perform *all* their reasoning at the source level.

Promising avenues for future work remain. We have focused on confidentiality and integrity properties, however, the label model could be extended with availability policies [102], guiding selection of fault-tolerant protocols like quorum replication [103] and MPC with guaranteed output delivery [50]. The Viaduct compiler can be extended with support for executing code on trusted execution environments like hardware enclaves [71, 56, 49], the use of special-purpose protocols like private set intersection [24, 82] and Oblivious RAM [91], and the incorporation of a more detailed and accurate cost model [55].

Using choreographies to reason about UC protocols can be of independent interest beyond program partitioning. Current approaches [17, 63] require specifying ideal functionalities as separate processes, which complicates whole-program reasoning. Fully integrating our result into the Universal Composability framework and showing how to instantiate our idealizations with concrete cryptographic protocols could pave the way for simpler UC proofs.

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

BIBLIOGRAPHY

[1] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *32nd IEEE Computer Security Foundations Symp. (CSF)*. IEEE Computer Society, 256–271. https://doi.org/10.1109/CSF.2019.00025

[2] Coşku Acay, Rolph Recto, Joshua Gancher, Andrew Myers, and Elaine Shi. 2021. Viaduct: An Extensible, Optimizing Compiler for Secure Distributed Programs. In *42nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 740–755. https://doi.org/10.1145/3453483.3454074

[3] Abdelrahaman Aly, Daniele Cozzo, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Nigel P. Smart, and Tim Wood. 2019. *SCALE–MAMBA v1.6 : Documentation*. https://homes.esat.kuleuven.be/~nsmart/SCALE

[4] Owen Arden, Jed Liu, and Andrew C. Myers. 2015. Flow-Limited Authorization. In *28th IEEE Computer Security Foundations Symp. (CSF)*. 569–583. https://doi.org/10.1109/CSF.2015.42

[5] Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. 2008. Cryptographically-masked flows. *Theor. Comput. Sci.* 402, 2-3 (2008), 82–101. https://doi.org/10.1016/j.tcs.2008.04.028

[6] Franz Baader and Tobias Nipkow. 1998. *Term rewriting and all that.* Cambridge University Press.

[7] Michael Backes, Birgit Pfitzmann, and Michael Waidner. 2007. The Reactive Simulatability (RSIM) Framework for Asynchronous Systems. *Information and*

Computation 205, 12 (2007), 1685–1720. https://doi.org/10.1016/j.ic.2007.05.002

[8] Alexander Bakst, Klaus von Gleissenthall, Rami Gökhan Kıcı, and Ranjit Jhala. 2017. Verifying Distributed Programs via Canonical Sequentialization. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 110 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133934

[9] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-Aided Cryptography. In *IEEE Symposium on Security and Privacy*. IEEE, 777–795. https://doi.org/10.1109/SP40001.2021.00008

[10] Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. 2021. 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*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 2541–2563. https://doi.org/10.1145/3460120.3484548

[11] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.* 4, POPL (2020), 7:1–7:30. https://doi.org/10.1145/3371075

[12] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *20^{th} Annual ACM Symposium on Theory of Computing*, Janos Simon (Ed.). 1–10. https://doi.org/10.1145/62212.62213

[13] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. 2009. Financial Cryptography and Data Security. Springer-Verlag, Berlin, Heidelberg, Chapter Secure Multiparty Computation Goes Live, 325–343. https://doi.org/10.1007/978-3-642-03549-4_20

[14] Niklas Broberg, Bart van Delft, and David Sands. 2013. Paragon for Practical Programming with Information-Flow Control. In *11th ASIAN Symposium on Programming Languages and Systems, APLAS 2013*. Springer, 217–232. https://doi.org/10.1007/978-3-319-03542-0_16

[15] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. 2018. HyCC: Compilation of Hybrid Protocols for Practical Secure Computation. In *25th ACM Conf. on Computer and Communications Security (CCS)*. ACM, New York, NY, USA, 847–861. https://doi.org/10.1145/3243734.3243786

[16] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology* (2000), 143–202. https://doi.org/10.1007/s001459910006

[17] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 136–145. https://doi.org/10.1109/SFCS.2001.959888

[18] Ran Canetti, Asaf Cohen, and Yehuda Lindell. 2014. A Simpler Variant of Universally Composable Security for Standard Multiparty Computation. Cryptology ePrint Archive, Report 2014/553. https://eprint.iacr.org/2014/553

[19] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. 2002. Universally composable two-party and multi-party secure computation. In *34th Annual ACM Symposium on Theory of Computing*. ACM, 494–503. https://doi.org/10.1145/509907.509980

[20] Ran Canetti, Alley Stoughton, and Mayank Varia. 2019. EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security. In *32nd IEEE Computer Security Foundations Symp. (CSF)*. IEEE, 167–183. https://doi.org/10.1109/CSF.2019.00019

[21] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. on Computer Systems* 20 (2002), 2002.

[22] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: a DSL for timing-sensitive computation. In *40th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 174–189. https://doi.org/10.1145/3314221.3314605

[23] Ethan Cecchetti, Andrew C. Myers, and Owen Arden. 2017. Nonmalleable Information Flow Control. In *24th ACM Conf. on Computer and Communications Security (CCS)* (Dallas, TX). ACM, 1875–1891. https://doi.org/10.1145/3133956.3134054

[24] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast Private Set Intersection from Homomorphic Encryption. In *24th ACM Conf. on Computer and Communications Security (CCS)*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). 1243–1255. https://doi.org/10.1145/3133956.3134061

[25] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure Web Applications via Automatic Partitioning. In *21st ACM Symp. on Operating System Principles (SOSP)*. 31–44. https://doi.org/10.1145/1323293.1294265

[26] Alonzo Church and J. B. Rosser. 1936. Some Properties of Conversion. *Trans. Amer. Math. Soc.* 39, 3 (1936), 472–482. https://doi.org/10.2307/1989762

[27] David Clark and Sebastian Hunt. 2008. Non-Interference for Deterministic Interactive Programs. In *5th Workshop on Formal Aspects in Security and Trust (FAST) (Lecture Notes in Computer Science, Vol. 5491)*. Springer, 50–66. https://doi.org/10.1007/978-3-642-01465-9_4

[28] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium*.

[29] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. 2015. Geppetto: Versatile verifiable computation. In *IEEE Symp. on Security and Privacy*. IEEE, 253–270. https://doi.org/10.1109/SP.2015.23

[30] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. 2021. Porcupine: A Synthesizing Compiler for Vectorized Homomorphic Encryption. In *42nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 375–389. https://doi.org/10.1145/3453483.3454050

[31] Luís Cruz-Filipe and Fabrizio Montesi. 2017. On Asynchrony and Choreographies. In *ICE@DisCoTec (EPTCS, Vol. 261)*, Massimo Bartoletti, Laura Bocchi, Ludovic

Henrio, and Sophia Knight (Eds.). 76–90. https://doi.org/10.4204/EPTCS.261.8

[32] Luís Cruz-Filipe and Fabrizio Montesi. 2020. A core model for choreographic programming. *Theoretical Computer Science* 802 (2020), 38–66. https://doi.org/10.1016/j.tcs.2019.07.005

[33] Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2022. A Formal Theory of Choreographic Programming. *CoRR* abs/2209.01886 (2022). https://doi.org/10.48550/arXiv.2209.01886 arXiv:2209.01886

[34] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation. In *41st ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 546–561. https://doi.org/10.1145/3385412.3386023

[35] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin E. Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing. In *40th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 142–156. https://doi.org/10.1145/3314221.3314628

[36] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary). Springer-Verlag, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[37] Christian Decker and Roger Wattenhofer. 2014. Bitcoin transaction malleability and MtGox. In *19th European Symposium on Research in Computer Security*. Springer, 313–326. https://doi.org/10.1007/978-3-319-11212-1_18

[38] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *Network and Distributed System Security Symp.* The Internet Society. https://doi.org/10.14722/ndss.2015.23113

[39] T. Dierks and E. Rescorla. 2006. The Transport Layer Security (TLS) Protocol Version 1.1. Internet RFC-4346.

[40] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *ACM Conf. on Computer and Communications Security (CCS)* (Berlin, Germany). 73–84. http://doi.acm.org/10.1145/2508859.2516693

[41] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling With Continuations. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (Albuquerque, New Mexico, USA) *(PLDI '93)*. 237–247. https://doi.org/10.1145/155090.155113

[42] Cédric Fournet, Guervan le Guernic, and Tamara Rezk. 2009. A Security-Preserving Compiler for Distributed Programs: From Information-Flow Policies to Cryptographic Mechanisms. In *16th ACM Conf. on Computer and Communications Security (CCS)*. 432–441. https://doi.org/10.1145/1653662.1653715

[43] Cédric Fournet and Tamara Rezk. 2008. Cryptographically sound implementations for typed information-flow security. In *35th ACM Symp. on Principles of Pro-*

gramming Languages (POPL). 323–335. https://doi.org/10.1145/1328438.1328478

[44] Joshua Gancher, Kristina Sojakova, Xiong Fan, Elaine Shi, and Greg Morrisett. 2023. A Core Calculus for Equational Proofs of Cryptographic Protocols. *Proc. ACM Program. Lang.* 7, POPL (2023), 866–892. https://doi.org/10.1145/3571223

[45] GDPR 2016. General Data Protection Regulation. https://gdpr-info.eu

[46] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The most dangerous code in the world: validating SSL certificates in non-browser software. In *19th ACM Conf. on Computer and Communications Security (CCS)*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM, 38–49. https://doi.org/10.1145/2382196.2382204

[47] Joseph A. Goguen and Jose Meseguer. 1982. Security Policies and Security Models. In *IEEE Symp. on Security and Privacy*. 11–20. https://doi.org/10.1109/SP.1982.10014

[48] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game. In *19th Annual ACM Symposium on Theory of Computing*, Alfred V. Aho (Ed.). 218–229. https://doi.org/10.1145/28395.28420

[49] Anitha Gollamudi, Stephen Chong, and Owen Arden. 2019. Information Flow Control for Distributed Trusted Execution Environments. In *32nd IEEE Computer Security Foundations Symp. (CSF)*. IEEE, 304–318. https://doi.org/10.1109/CSF.2019.00028

[50] S. Dov Gordon, Feng-Hao Liu, and Elaine Shi. 2015. Constant-Round MPC with Fairness and Guarantee of Output Delivery. , 63–82 pages. https://doi.org/10.1007/978-3-662-48000-7_4

[51] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. 2019. SoK: General Purpose Compilers for Secure Multi-Party Computation. In *IEEE Symp. on Security and Privacy*. 1220–1237. https://doi.org/10.1109/SP.2019.00028

[52] Matthew Hennessy and Robin Milner. 1985. Algebraic Laws for Nondeterminism and Concurrency. *J. ACM* 32, 1 (1985), 137–161. https://doi.org/10.1145/2455.2460

[53] Andrew K. Hirsch and Deepak Garg. 2022. Pirouette: Higher-Order Typed Functional Choreographies. *Proc. ACM Program. Lang.* 6, POPL (Jan. 2022), 1–27. https://doi.org/10.1145/3498684

[54] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2007. Zero-knowledge from Secure Multiparty Computation. In *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing* (San Diego, California, USA) *(STOC '07)*. ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/1250790.1250794

[55] Muhammad Ishaq, Ana Milanova, and Vassilis Zikas. 2019. Efficient MPC via Program Analysis: A Framework for Efficient Optimal Mixing. In *26^{th} ACM Conf. on Computer and Communications Security (CCS)*. ACM, 1539–1556. https://doi.org/10.1145/3319535.3339818

[56] David Kaplan, Jeremy Powell, and Tom Woller. 2016. *AMD memory encryption.* Technical Report.

[57] Florian Kerschbaum. 2011. Automatically Optimizing Secure Computation. In *18th ACM Conf. on Computer and Communications Security (CCS)*. https://doi.org/10.1145/2046707.2046786

[58] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. 2018. xJsnark: A Framework for Efficient Verifiable Computation. In *IEEE Symp. on Security and Privacy*. IEEE, 944–961. https://doi.org/10.1109/SP.2018.00018

[59] Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. 2015. A Hybrid Approach for Proving Noninterference of Java Programs. In *28th IEEE Computer Security Foundations Symp. (CSF)*, Cédric Fournet, Michael W. Hicks, and Luca Viganò (Eds.). IEEE Computer Society, 305–319. https://doi.org/10.1109/CSF.2015.28

[60] Ralf Küsters, Tomasz Truderung, and Juergen Graf. 2012. A Framework for the Cryptographic Verification of Java-Like Programs. In *25th IEEE Computer Security Foundations Symp. (CSF)*, Stephen Chong (Ed.). IEEE Computer Society, 198–212. https://doi.org/10.1109/CSF.2012.9

[61] Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. on Computer Systems* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[62] Peeter Laud. 2008. On the computational soundness of cryptographically masked flows. In *35th ACM Symp. on Principles of Programming Languages (POPL)*, George C. Necula and Philip Wadler (Eds.). ACM, 337–348. https://doi.org/10.1145/1328438.1328479

[63] Kevin Liao, Matthew A. Hammer, and Andrew Miller. 2019. ILC: A Calculus for Composable, Computational Cryptography. In *40th ACM SIGPLAN Conf. on*

Programming Language Design and Implementation (PLDI). 640–654. https://doi.org/10.1145/3314221.3314607

[64] Yehuda Lindell. 2017. How to Simulate It — A Tutorial on the Simulation Proof Technique. In *Tutorials on the Foundations of Cryptography*. Springer International Publishing, 277–346. https://doi.org/10.1007/978-3-319-57048-8_6

[65] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. 2014. Automating Efficient RAM-Model Secure Computation. In *IEEE Symp. on Security and Privacy*. IEEE, 623–638. https://doi.org/10.1109/SP.2014.46

[66] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *25th ACM Symp. on Operating System Principles (SOSP)*. IEEE, 359–376. https://doi.org/10.1109/SP.2015.29

[67] Alberto Lluch-Lafuente, Flemming Nielson, and Hanne Riis Nielson. 2015. Discretionary Information Flow Control for Interaction-Oriented Specifications. In *Logic, Rewriting, and Concurrency (Lecture Notes in Computer Science, Vol. 9200)*, Narciso Martí-Oliet, Peter Csaba Ölveczky, and Carolyn L. Talcott (Eds.). Springer, 427–450. https://doi.org/10.1007/978-3-319-23165-5_20

[68] Andreas Lochbihler, S Reza Sefidgar, David Basin, and Ueli Maurer. 2019. Formalizing constructive cryptography using CryptHOL. In *32nd IEEE Computer Security Foundations Symp. (CSF)*. IEEE, 152–166. https://doi.org/10.1109/CSF.2019.00018

[69] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay - A Secure Two-Party Computation System. In *13th Usenix Security Symposium*

(San Diego, CA). 287–302. http://www.usenix.org/publications/library/proceedings/sec04/tech/malkhi.html

[70] Ueli Maurer. 2011. Constructive Cryptography – A New Paradigm for Security Definitions and Proofs. In *Theory of Security and Applications (Lecture Notes in Computer Science, Vol. 6993)*, Sebastian Mödersheim and Catuscia Palamidessi (Eds.). Springer, 33–56. https://doi.org/10.1007/978-3-642-27375-9_3

[71] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Workshop on Hardware and Architectural Support for Security and Privacy*, Ruby B. Lee and Weidong Shi (Eds.). ACM, 10. https://doi.org/10.1145/2487726.2488368

[72] Fabrizio Montesi. 2013. *Choreographic Programming*. Ph. D. Dissertation. IT University of Copenhagen. https://www.fabriziomontesi.com/files/choreographic-programming.pdf

[73] Fabrizio Montesi. 2023. *Introduction to Choreographies*. Cambridge University Press, Cambridge.

[74] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *26th ACM Symp. on Principles of Programming Languages (POPL)*. 228–241. https://doi.org/10.1145/292540.292561

[75] Andrew C. Myers and Barbara Liskov. 2000. Protecting Privacy using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology* 9, 4 (Oct. 2000), 410–442. https://doi.org/10.1145/363516.363526

[76] Sumit Nain and Moshe Y. Vardi. 2007. Branching vs. Linear Time: Semantical Perspective. In *ATVA (Lecture Notes in Computer Science, Vol. 4762)*, Kedar S. Namjoshi,

Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura (Eds.). Springer, 19–34. https://doi.org/10.1007/978-3-540-75596-8_4

[77] M. H. A. Newman. 1942. On Theories with a Combinatorial Definition of "Equivalence". *Annals of Mathematics* 43, 2 (1942), 223–243. https://doi.org/10.2307/2269299

[78] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. 2006. Information-Flow Security for Interactive Programs. In *19th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society, 190–201. https://doi.org/10.1109/CSFW.2006.16

[79] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly practical verifiable computation. In *IEEE Symp. on Security and Privacy*. IEEE, 238–252. https://doi.org/10.1109/SP.2013.47

[80] Marco Patrignani, Robert Künnemann, and Riad S. Wahby. 2022. Universal Composability is Robust Compilation. arXiv:1910.08634 [cs.PL]

[81] Marco Patrignani, Riad S. Wahby, and Robert Künneman. 2019. Universal Composability is Secure Compilation. *CoRR* abs/1910.08634 (2019). https://doi.org/10.48550/ARXIV.1910.08634

[82] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2019. SpOT-Light: Lightweight Private Set Intersection from Sparse OT Extension. In *Advances in Cryptology – CRYPTO 2019*. Springer, 401–431. https://doi.org/10.1007/978-3-030-26954-8_13

[83] Johannes Åman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. 2022. Kalas: A Verified, End-To-End Compiler for a Choreographic Language. In *ITP (LIPIcs, Vol. 237)*, June Andronick and Leonardo de Moura

(Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:18. https://doi.org/10.4230/LIPIcs.ITP.2022.27

[84] François Pottier and Vincent Simonet. 2002. Information flow inference for ML. In *29th ACM Symp. on Principles of Programming Languages (POPL)*. 319–330. https://doi.org/10.1145/503272.503302

[85] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. 2014. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *IEEE Symp. on Security and Privacy*. 655–670. https://doi.org/10.1109/SP.2014.48

[86] Jakob Rehof and Torben Æ. Mogensen. 1996. Tractable Constraints in Finite Semilattices. In *3rd International Symposium on Static Analysis (Lecture Notes in Computer Science, 1145)*. Springer-Verlag, 285–300. https://doi.org/10.1007/3-540-61739-6_48

[87] Daniel Edwin Rutherford. 1965. *Introduction to Lattice Theory*. Oliver and Boyd.

[88] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (Jan. 2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

[89] SCIPR Lab and contributors. 2012. libsnark: a C++ library for zkSNARK proofs. https://github.com/scipr-lab/libsnark.

[90] Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. 2011. Disjunction Category Labels. In *Proceedings of the 16th Nordic conference on Information Security Technology for Applications (Lecture Notes in Computer Science, Vol. 7161)*, Peeter Laud (Ed.). Springer, 223–239. https://doi.org/10.1007/978-3-642-29615-4_16

[91] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *20th ACM Conf. on Computer and Communications Security (CCS)*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). 299–310. https://doi.org/10.1145/2508859.2516660

[92] Eli Tilevich and Yannis Smaragdakis. 2009. J-Orchestra: Enhancing Java Programs with Distribution Capabilities. *ACM Trans. Softw. Eng. Methodol.* 19, 1, Article 1 (Aug. 2009), 40 pages. https://doi.org/10.1145/1555392.1555394

[93] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. 2021. Automatically eliminating speculative leaks from cryptographic code with Blade. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434330

[94] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *ACM SIGOPS/EuroSys European Conference on Computer Systems*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). 3:1–3:18. https://doi.org/10.1145/3302424.3303982

[95] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. 2015. Efficient RAM and Control Flow in Verifiable Outsourced Computation. In *Network and Distributed System Security Symp.* The Internet Society. https://doi.org/10.14722/ndss.2015.23097

[96] Andrew C. Yao. 1982. Protocols for Secure Computations. In *23rd annual IEEE Symposium on Foundations of Computer Science*. 160–164. https://doi.org/10.1109/SFCS.1982.38

[97] Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. 2019. Using Information Flow to Design an ISA that Controls Timing Channels. In *32nd IEEE Computer Security Foundations Symp. (CSF)*. https://doi.org/10.1109/CSF.2019.00026

[98] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. *IACR Cryptol. ePrint Arch.* (2015). http://eprint.iacr.org/2015/1153

[99] Steve Zdancewic and Andrew C. Myers. 2001. Robust Declassification. In *14th IEEE Computer Security Foundations Workshop (CSFW)* (Cape Breton, Nova Scotia, Canada). 15–23. https://doi.org/10.1109/CSFW.2001.930133

[100] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. 2002. Secure Program Partitioning. *ACM Trans. on Computer Systems* 20, 3 (Aug. 2002), 283–328. https://doi.org/10.1145/566340.566343

[101] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. 2003. Using Replication and Partitioning to Build Secure Distributed Systems. In *IEEE Symp. on Security and Privacy* (Oakland, California). 236–250. https://doi.org/10.1109/SECPRI.2003.1199340

[102] Lantian Zheng and Andrew C. Myers. 2005. End-to-End Availability Policies and Noninterference. In *18th IEEE Computer Security Foundations Workshop (CSFW)*. 272–286. https://doi.org/10.1109/CSFW.2005.16

[103] Lantian Zheng and Andrew C. Myers. 2014. A Language-Based Approach to Secure Quorum Replication. In *9th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*. https://doi.org/10.1145/2637113.2637117