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
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.
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

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biographical Sketch</td>
<td>iii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>v</td>
</tr>
<tr>
<td>Contents</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>1 Introduction</td>
<td></td>
</tr>
<tr>
<td>1.1 Dissertation Outline</td>
<td>5</td>
</tr>
<tr>
<td>2 Secure Program Partitioning with Cryptographic Protocols</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Overview</td>
<td>7</td>
</tr>
<tr>
<td>2.1.1 Label Inference</td>
<td>11</td>
</tr>
<tr>
<td>2.1.2 Protocol Selection</td>
<td>11</td>
</tr>
<tr>
<td>2.1.3 Runtime</td>
<td>14</td>
</tr>
<tr>
<td>2.1.4 Threat Model</td>
<td>15</td>
</tr>
<tr>
<td>2.2 Information-Flow Control</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1 Capturing Attacks with Labels</td>
<td>17</td>
</tr>
<tr>
<td>2.2.2 Delegation</td>
<td>20</td>
</tr>
<tr>
<td>2.3 Source Language</td>
<td>21</td>
</tr>
<tr>
<td>2.3.1 Label Checking</td>
<td>23</td>
</tr>
<tr>
<td>2.4 Label Inference</td>
<td>26</td>
</tr>
<tr>
<td>2.4.1 Label Constraints</td>
<td>27</td>
</tr>
<tr>
<td>2.4.2 Principal Constraints</td>
<td>28</td>
</tr>
<tr>
<td>2.4.3 Solving Principal Constraints</td>
<td>30</td>
</tr>
<tr>
<td>2.5 Protocol Selection</td>
<td>31</td>
</tr>
<tr>
<td>2.5.1 Validity of Protocol Assignments</td>
<td>33</td>
</tr>
<tr>
<td>2.5.2 Cost of Protocol Assignments</td>
<td>34</td>
</tr>
<tr>
<td>2.5.3 Computing an Optimal Protocol Assignment</td>
<td>35</td>
</tr>
<tr>
<td>2.6 Runtime System</td>
<td>37</td>
</tr>
<tr>
<td>2.6.1 Protocol Composition</td>
<td>38</td>
</tr>
<tr>
<td>2.7 Implementation</td>
<td>41</td>
</tr>
<tr>
<td>2.8 Evaluation</td>
<td>43</td>
</tr>
<tr>
<td>2.8.1 Expressiveness</td>
<td>45</td>
</tr>
<tr>
<td>2.8.2 Scalability of Compilation</td>
<td>45</td>
</tr>
<tr>
<td>2.8.3 Performance of Compiled Programs</td>
<td>47</td>
</tr>
<tr>
<td>2.8.4 Annotation Burden of Security Labels</td>
<td>47</td>
</tr>
<tr>
<td>2.8.5 Overhead of Runtime System</td>
<td>48</td>
</tr>
<tr>
<td>2.9 Related Work</td>
<td>49</td>
</tr>
</tbody>
</table>
LIST OF TABLES

2.1 Benchmark programs. **Ann** is the minimum number of label annotations needed to write the program. ........................................ 44

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. ........................................ 44

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**. ........................................ 46

   a. Run time (in seconds) in the LAN setting. ........................................ 46
   b. Run time (in seconds) in the WAN setting. ........................................ 46
   c. Communication (in MB) in the LAN/WAN setting. ........................................ 46

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. ........................................ 48
LIST OF FIGURES

2.1 Architecture of Viaduct ........................................... 7
2.2 Implementation of the historical millionaires’ problem in Viaduct. Viaduct uses MPC for the comparison \( a < b \), but computes the minima locally ......................................................... 8
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 ................................................. 10
2.4 Example protocols and security labels that represent their authority .................. 12
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 \( h \) ..................... 14
2.6 Information flow lattice. Attacks are valid when \( S \cap U \) is entirely compromised ......................................................... 18
2.7 Syntax of delegations .................................................................. 19
2.8 Semantics of delegations .......................................................... 20
2.9 Abstract syntax of Viaduct’s source language ........................................... 22
2.10 Information flow checking rules for expressions and statements .......... 23
2.11 A program that violates robust declassification ........................................... 25
2.12 Syntax of label constraints ....................................................... 27
2.13 Syntax of principal constraints ....................................................... 28
2.14 Translating label constraints to principal constraints ............................... 29
2.15 Rules for simplifying principal constraints .............................................. 30
2.16 Rules for validating a protocol assignment .............................................. 32
2.17 Protocols and hosts involved in the execution of a statement. Here, hosts\((r)\) is the set of hosts that protocol \( r \) runs on, specified individually for each protocol ..................... 33
2.18 Abstract cost model .............................................................. 35
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, \( \lfloor \cdot \rfloor \) is endpoint projection, source\((\cdot)\) is the inverse of host selection, and cor\((\cdot)\) models corruption ......................................................... 52
3.2 Compiling the Millionaires’ Problem .............................................. 53
   a Source program with information-flow labels ........................................... 53
   b Intermediate choreography with explicit communication and synchronization ......................................................... 53
   c Target distributed program derived by projecting choreography ..................... 53
3.3 Unified syntax of source, choreography, and target languages. \( * \) terms are choreography only, \( \dagger \) are target only ......................................................... 59
3.4 Syntax of messages and actions ...................................................... 61
3.5 Ideal stepping rules for expressions and statements ...................................... 63
3.6 Real stepping rules for expressions and statements. These override the rules in fig. 3.5. ................................. 64
3.7 Concurrent lifting of ideal/real stepping rules. ......................... 65
3.8 Stepping rules for buffers, processes, and configurations. .......... 67
3.9 Canonical source program from a choreography. .................... 68
3.10 Information-flow typing rules for expressions and statements. ... 70
3.11 Checking that a concurrent program acts like a sequential program. 71
3.12 Endpoint projection and select merge rules. ......................... 74
3.13 The syntax of asynchronous choreographies (extends fig. 3.3). .... 84
3.14 Stepping rules for asynchronous choreographies. These override the rules for \( \rightarrow_r \) in fig. 3.6. ................................. 84
3.15 Stepping rules used internally by the simulator. These override fig. 3.6 89
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 heterogeneous 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
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”
host Alice, Bob
assume Alice trusts Bob for integrity
assume Bob trusts Alice for integrity

fun historicalMillionaires() {
val a1, a2, a3 = Alice.input<int>()
val b1, b2, b3 = Bob.input<int>()
val a = min(a1, a2, a3)
val b = min(b1, b2, b3)
val b_richer = declassify(a < b, Alice ∨ Bob)
Alice.output(b_richer)
Bob.output(b_richer)
}

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
**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 $L(r)$, which approximates the security guarantees the protocol provides. Given a program component with minimum authority requirement $\ell$, protocol selection only assigns $r$ to execute that component if $L(r) \Rightarrow \ell$—that is, if $r$ meets the authority requirement for the program component.
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 $L(r)$. In particular, the adversary cannot observe storage or computation performed by $r^s$ unless its confidentiality is at least $L(r)$; dually, the adversary cannot influence storage or computation performed by $r^s$ unless its integrity is at least $L(r)$.

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:

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Confidentiality</th>
<th>Integrity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local($h$)</td>
<td>$h$</td>
<td>$h$</td>
</tr>
<tr>
<td>Replication($H$)</td>
<td>$\sqrt{h \in H} h$</td>
<td>$\land_{h \in H} h$</td>
</tr>
<tr>
<td>Commitment($h_p, h_v$)</td>
<td>$h_p$</td>
<td>$h_p \land h_v$</td>
</tr>
<tr>
<td>ZKP($h_p, h_v$)</td>
<td>$h_p$</td>
<td>$h_p \land h_v$</td>
</tr>
<tr>
<td>MAL-MPC($H$)</td>
<td>$\land_{h \in H} h$</td>
<td>$\land_{h \in H} h$</td>
</tr>
<tr>
<td>SH-MPC($H$)</td>
<td>$\land_{h \in H} h$</td>
<td>$\lor_{h \in H} h$</td>
</tr>
</tbody>
</table>

Figure 2.4: Example protocols and security labels that represent their authority.
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** \((h_p, h_v)\) 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 \(h_p \land h_v\) for the same reason as for replication.

**ZKP** \((h_p, h_v)\) 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 \(\land_{h \in H} h\) 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 \(\lor_{h \in H} h\) 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 \(Alice \land Bob\) to \(Alice \lor Bob\). 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
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 $\mathbb{H} = \{ Alice, Bob, Chuck, \ldots \}$. 
The \textit{acts-for} relation \((\Rightarrow)\) orders principals by authority, and coincides with logical implication: for example, \(p \land q \Rightarrow p\) and \(q \Rightarrow p \lor q\). The most powerful principal is \(0\) and the least powerful, \(1\), so we have \(0 \Rightarrow p \Rightarrow 1\) for any principal \(p\).

We lift \(\land, \lor, \Rightarrow\) to labels in the obvious pointwise manner. Whenever appropriate, we write \(p\) for the security label \(\langle p, p \rangle\). Confidentiality and integrity projections \(\ell \to\) and \(\ell \leftarrow\) completely weaken the other component of a label:

\[
\langle p, q \rangle \to = \langle p, 1 \rangle \quad \langle p, q \rangle \leftarrow = \langle 1, q \rangle.
\]

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:

\[
\langle p_1, q_1 \rangle \subseteq \langle p_2, q_2 \rangle \iff p_2 \Rightarrow p_1 \text{ and } q_1 \Rightarrow q_2 \quad \text{(flows to)}
\]

\[
\langle p_1, q_1 \rangle \sqcup \langle p_2, q_2 \rangle = \langle p_1 \land p_2, q_1 \lor q_2 \rangle \quad \text{(join)}
\]

\[
\langle p_1, q_1 \rangle \sqcap \langle p_2, q_2 \rangle = \langle p_1 \lor p_2, q_1 \land q_2 \rangle \quad \text{(meet)}
\]

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

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

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

- The attacker always controls the weakest principal, but never the strongest: $1 \in P$ and $0 \notin P$.
- 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 $p \land q \in P$.
- If the attacker controls the the common power of two principals, then it controls at least one of the principals: if $p \lor q \in P$, then either $p \in P$ or $q \in P$.

Together, these conditions imply that $P$ and $Q$ are sensible truth assignments to elements $P$: sensible in the sense that they play nicely with $\land$ and $\lor$.

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

\footnote{For those familiar with order theory, $P$ and $Q$ must be prime filters of $P$.}
Delegation
\[ d ::= p \Rightarrow q \]
Principal Delegations
\[ \theta ::= d_1, \ldots, d_2 \]
Label Delegations
\[ \Theta ::= (\theta_{\rightarrow}, \theta_{\leftarrow}) \]

Figure 2.7: Syntax of delegations.

and \( T/U \), respectively. The partitioning is depicted in fig. 2.6 and defined as follows:

\[ \mathcal{P} = \{ \langle p, q \rangle \in L \mid p \in P \} \]
\[ \mathcal{S} = L \setminus \mathcal{P} \]
\[ \mathcal{T} = \{ \langle p, q \rangle \in L \mid q \notin Q \} \]
\[ \mathcal{U} = L \setminus \mathcal{T} \]

For a host \( h \), define \( \mathbb{L}(h) = \langle h, h \rangle \). Recalling the threat model, an honest host has a secret, trusted label (\( \mathbb{L}(h) \in S \cap T \)); a semi-honest host has a public, trusted label (\( \mathbb{L}(h) \in P \cap T \)); and a malicious host has a public, untrusted label (\( \mathbb{L}(h) \notin T \)). 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 \( \nabla \ell \), 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 \( \nabla \ell \), then \( \ell \notin S \cap U \).
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 \( \theta \) is a set of delegations. Since labels are pairs of principals, a label-delegation context \( \Theta \) 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 \( \text{Alice} \Rightarrow \text{Bob} \) in the system, we are no longer interested in attackers that control \( \text{Alice} \) but not \( \text{Bob} \). We formally capture this insight in fig. 2.8. We write \( P \models \theta \) when an attack \( P \) is consistent with every delegation in \( \theta \). We say \( p \) acts for \( q \) under the assumptions \( \theta \), written \( \theta \models p \Rightarrow q \), when any attacker consistent with \( \theta \) that controls \( p \) necessarily controls \( q \)\(^3\).

\(^3\)A delegation \( \text{Alice} \Rightarrow \text{Bob} \) has an effect beyond the principals explicitly involved (i.e., \( \text{Alice} \) and \( \text{Bob} \)). For example, it implies \( \text{Alice} \land \text{Chuck} \Rightarrow \text{Bob} \land \text{Chuck} \).
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,

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

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 $\Theta \models \triangleleft \ell$, 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 $\Theta$ 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.
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 \( \ell \).

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:

\[
\text{while } t \text{ do } s \equiv \mu X.\text{if } t \text{ then } (s;X) \text{ else skip.}
\]
\[
\begin{align*}
&\Gamma \vdash t : \ell & &\quad &\Gamma ; pc \vdash e : \ell \\
&\Gamma(x) \subseteq \ell & &\quad &\forall i . \Gamma \vdash t_i : \ell \\
&\Gamma \vdash v : \ell & &\quad &pc \subseteq \ell \\
&\Gamma \vdash x : \ell & &\quad &\nabla \ell_i \quad \ell_i \cap \langle 0, 1 \rangle \subseteq \ell \\
&pc \subseteq \ell & &\quad &pc \vdash e : \ell \\n&pc \subseteq L(h) & &\quad &pc \subseteq L(h) \\
&\Gamma ; pc \vdash endorse(t, p) : \ell & &\quad &pc \vdash input h : \ell \\
&pc \subseteq \Delta \langle h \rangle & &\quad &pc \vdash output t to h : \ell \\
&pc \subseteq \Gamma(X) & &\quad &pc \vdash \mu X. s \\
&pc \subseteq \Delta \langle h \rangle & &\quad &pc \vdash \mu X. s \\
&\Gamma ; pc \vdash s & &\quad &pc \vdash s_1 \\
&\Gamma ; pc \vdash \mu X. s & &\quad &pc \vdash \mu X. s \\
&\Gamma ; pc \vdash \mu X. s & &\quad &pc \vdash \mu X. s \\
&\Gamma ; pc \vdash skip & &\quad &pc \vdash skip
\end{align*}
\]

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 \( \ell \) under the context on the left. Here, \( \Gamma \) is a finite partial map from variables to labels:

\[
\text{Label Contexts} \quad \Gamma ::= \epsilon | \Gamma, x : \ell | \Gamma, X : \ell
\]
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.
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 maulled 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 \( \text{info} \) depends on (low-integrity) \( \text{guess} \). Without the restrictions on downgrading, Viaduct could compile the program to store the guard \( \text{pw} == \text{guess} \) (with label \( \text{Client} \)) on \( \text{Client} \). \( \text{Client} \) 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 \( \text{guess} \) before declassifying the comparison; since \( \text{guess} \) 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., \( \ell_f \) in the rule for \texttt{declassify} expressions and \( pc' \) in the rule for \texttt{if} statements). The inference algorithm finds a label-variable assignment that satisfies all the constraints, if possible.

The algorithm computes the \textit{minimum-authority} solution, the choice of labels requiring the least amount of confidentiality and integrity for each component. Minimum-
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 $L^\pi$ 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 $\ell$ is a pair of principals $(p, q)$, 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.
Principal Constants \( p \)
Principal Variables \( Y^\pi \)
Principal Expressions

\[
P^\pi := p \mid Y^\pi \mid P^\pi_1 \lor P^\pi_2 \mid P^\pi_1 \land P^\pi_2 \\
\mid p_1 \rightarrow P^\pi_2 \mid \min_{\pi'}(P^\pi')
\]

Principal Constraints \( D \)

\[
D := P^\pi_1 \Rightarrow \pi P^\pi_2
\]

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 \( Y^\pi \) for each combination of label variable \( Y \) and projection \( \pi \). That is, \( Y^\rightarrow \) represents the confidentiality component of \( Y \), and \( Y^\leftarrow \) represents the integrity component of \( Y \).

We index expressions \( P^\pi \) by the component \( \pi \) they represent. This prevents mixing confidentiality and integrity components in the same expression, for example, by writing \( Y_1^\rightarrow \land Y_2^\leftarrow \). Expressions include principal constants and principal variables, as well as principal-lattice operations \( \lor \) and \( \land \). The operation \( \rightarrow \) is called the relative pseudocomplement of \( \land \): \( p_1 \rightarrow p_2 \) is defined as the weakest principal \( p \) such that \( p_1 \land p \Rightarrow p_2 \). We use \( \rightarrow \) to solve constraints of the form \( Y^\pi \land p_1 \Rightarrow \pi P^\pi_2 \). The operation \( \min_{\pi'}(p) \) is the strongest principal \( p' \) equivalent to \( p \) according to the delegations \( \theta_{\pi} \) for the \( \pi \) component. This operation allows mixing integrity and confidentiality components; we use it when solving for labels that must be uncompromised.

Constraints have the form \( P^\pi_1 \Rightarrow \pi P^\pi_2 \), which asserts that expression \( P^\pi_1 \) acts for \( P^\pi_2 \) under delegations \( \theta_{\pi} \). Note that we must specify \( \pi \) since we might have different delegations for confidentiality and integrity.

\[\footnote{A lattice that supports the \( \rightarrow \) operation is called a Heyting algebra \cite{87}. Any free distributive lattice, such as our lattice of principals, is a Heyting algebra.}\]
\[\llbracket L \rrbracket_\pi = P_\pi\]

\[
\llbracket (p, q) \rrbracket_\pi = \begin{cases} 
p & \text{if } \pi = \rightarrow \\
q & \text{if } \pi = \leftarrow 
\end{cases}
\]

\[
\llbracket Y \rrbracket_\pi = Y^\pi
\]

\[
\llbracket L^\pi \rrbracket_\pi = \begin{cases} 
\llbracket L \rrbracket_\pi & \text{if } \pi = \pi' \\
1 & \text{if } \pi \neq \pi'
\end{cases}
\]

\[
\llbracket L_1 \sqcup L_2 \rrbracket_\pi = \llbracket (L_1 \land L_2) \rightarrow \land (L_1 \lor L_2) \leftarrow \rrbracket_\pi
\]

\[
\llbracket L_1 \sqcap L_2 \rrbracket_\pi = \llbracket (L_1 \lor L_2) \rightarrow \land (L_1 \land L_2) \leftarrow \rrbracket_\pi
\]

\[
\llbracket L_1 \lor L_2 \rrbracket_\pi = \llbracket L_1 \rrbracket_\pi \lor \llbracket L_2 \rrbracket_\pi
\]

\[
\llbracket L_1 \land L_2 \rrbracket_\pi = \llbracket L_1 \rrbracket_\pi \land \llbracket L_2 \rrbracket_\pi
\]

\[\llbracket C \rrbracket = D_1, \ldots, D_n\]

\[
\llbracket L_1 \sqsubseteq L_2 \rrbracket = \llbracket L_2 \rrbracket \Rightarrow \Rightarrow \llbracket L_1 \rrbracket, \llbracket L_1 \rrbracket \Leftarrow \Rightarrow \llbracket L_2 \rrbracket, \llbracket \nabla L \rrbracket = \llbracket L \rrbracket \Leftarrow \Rightarrow \min_\pi(\llbracket L \rrbracket)\]

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 \(\llbracket L \rrbracket_\pi\), which returns a principal expression representing the confidentiality or the integrity component (depending on \(\pi\)) of the label expression \(L\). The definition of \(\llbracket L \rrbracket_\pi\) is a straightforward reading of the rules in section 2.2. Using \(\llbracket L \rrbracket\), we translate a flows-to (\(\sqsubseteq\)) constraint to multiple acts-for (\(\Rightarrow\)) constraints, one for each label component, following section 2.2. The constraint \(\nabla L\) 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
compute acts-for in the integrity domain ($\Rightarrow_{\neg}$), 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 $p_1 \Rightarrow \pi p_2$ and $Y_1 \Rightarrow \pi Y_2$. We apply the rules in fig. 2.15, as well as the associativity, commutativity, and idempotence of $\lor$ and $\land$, 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 ($\land$).\(^5\)

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 $Y_1 \land Y_2 \Rightarrow Alice$ has no minimal solution: we can assign $\{Y_1 \mapsto Alice, Y_2 \mapsto 1\}$ or $\{Y_1 \mapsto 1, Y_2 \mapsto Alice\}$, 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 systems. The Viaduct compiler fails with an informative error message.

\[^5\]Translation rules in fig. 2.14 never generate constraints with $\rightarrow$ or $\min_{\rightarrow}(\cdot)$ on the left-hand side, and the rules in fig. 2.15 eliminate all joins ($\lor$) 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 1, and use unsatisfied
constraints to update variables repeatedly until a fixed point is reached, using the
following rule:

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

where current-value$(\Theta, P^\pi)$ is the value of $P^\pi$ according to the current assignment.

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
\[ \Pi \models t : r \quad \Pi \models e : r \]

\[ \Pi \models t : r \quad \Pi \models e : r \]

\[ \Pi \models v : r \quad \Pi \models x : r \quad \Pi \models f(t_1, \ldots, t_n) : r \]

\[ \Pi \models t : r \quad \Pi \models t : r \quad \Pi \models t : r \quad \Pi \models t : r \]

\[ \Pi \models \text{declassify}(t, p) : r \quad \Pi \models \text{endorse}(t, p) : r \quad \Pi \models \text{input} h : \text{Local}(h) \]

\[ \Pi \models t : \text{Local}(h) \quad \Pi \models \text{output} t \text{ to } h : \text{Local}(h) \]

\[ \Pi \models s \]

\[ H = \text{hosts}(s_1) \cup \text{hosts}(s_2) \]
\[ \forall h \in H. \mathbb{L}(x) \rightarrow \subseteq \mathbb{L}(h) \rightarrow \]
\[ \forall h \in H. \mathbb{L}(\Pi(x)) \leftarrow \subseteq \mathbb{L}(h) \leftarrow \]
\[ \forall h \in H. \text{local-export}(\Pi(x), h) \]

\[ \Pi \models x : r \models s \]

\[ \Pi \models e : r \quad \Pi \models x : r \models s \]

\[ \Pi \models \text{let } r. x = e ; s \]

\[ \Pi \models \text{if } x \text{ then } s_1 \text{ else } s_2 \]

\[ \Pi \models \text{if } v \text{ then } s_1 \text{ else } s_2 \quad \Pi \models s \quad \Pi \models s \quad \Pi \models s \quad \Pi \models \mu X. s \quad \Pi \models X \quad \Pi \models s_1 \quad \Pi \models s_2 \quad \Pi \models \text{skip} \]

Figure 2.16: Rules for validating a protocol assignment.

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

We first give a declarative specification of when a protocol assignment is correct; we then explain how to search for such assignments.
protocols(s) = R

protocols(let r.x = e; s) = \{r\} ∪ protocols(s)
protocols(if t then s₁ else s₂) = protocols(s₁) ∪ protocols(s₂)
protocols(µX.s) = protocols(s)
protocols(X) = protocols(µX.s)

where µX.s is the original definition of X
protocols(s₁; s₂) = protocols(s₁) ∪ protocols(s₂)
protocols(skip) = ∅

hosts(s) = H

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

Figure 2.17: Protocols and hosts involved in the execution of a statement. Here, 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 \( \Pi \) maps variables to the protocols storing them:

\[
\text{Protocol Contexts } \Pi ::= \epsilon \mid \Pi, x : r
\]

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 comm(rₓ, 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 \texttt{let} statements ensures that the protocol selected for a variable has enough authority to securely store it. Formally, the label $L(r)$ of the protocol storing $x$ must act for ($\Rightarrow$) the minimum required authority label $L(x)$ computed for $x$ in section 2.4. Labels $L(r)$ 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 $\text{local-export}(\Pi(x), h)$. These premises are trivially satisfied when the guard is a constant expression.

The judgment $\text{local-export}(r, h)$ 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 $\text{local-export}($Replication$(H), h)$ for all $h \in H$, and $\text{local-export}(\text{Commitment}(h_p, h_v), h_p)$.

Where necessary, the Viaduct compiler removes these guard visibility constraints by multiplexing 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
\[
\text{cost(let } r.x = e; s) = c_{\text{exec}}(r, e) + \sum_{r' \in \text{readers}(x,s)} c_{\text{comm}}(r, r') + \text{cost}(s)
\]
\[
\text{cost(if } t \text{ then } s_1 \text{ else } s_2) = \max(\text{cost}(s_1), \text{cost}(s_2))
\]
\[
\text{cost(}\mu X.s\text{)} = W_{\text{loop}} \times \text{cost}(s)
\]
\[
\text{cost}(X) = 0
\]
\[
\text{cost}(s_1; s_2) = \text{cost}(s_1) + \text{cost}(s_2)
\]
\[
\text{cost(}\text{skip}\text{)} = 0
\]

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 \(c_{\text{exec}}(r, s)\), the cost of executing statement \(s\) in protocol \(r\); \(c_{\text{comm}}(r_1, r_2)\), the cost of communicating between \(r_1\) and \(r_2\); and the global constant \(W_{\text{loop}}\), the number of times a loop is assumed to execute when its iteration count is not statically known.

Our implementation configures \(c_{\text{exec}}\) 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 \(c_{\text{comm}}\) 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 ($\alpha_i$) These represent the protocols that execute let-bindings or declarations.

Cost variables ($\beta_i$) 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 $\beta_s$ 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 $\beta_s$ is minimized. Define $\text{VA}(s)$ to be the set of valid protocol assignments for $s$, that is,

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

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

$$s_{opt} = \arg\min_{s' \in \text{VA}(s)} \text{cost}(s').$$

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(·)—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(·)) 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 \texttt{Replication(Alice, Bob) \rightarrow 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: \texttt{Cleartext(}\textit{H}_1\texttt{) \rightarrow Cleartext(}\textit{H}_2\texttt{)}. 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).

\texttt{Local(}\textit{h}_1\texttt{) \rightarrow SH-MPC(}\textit{h}_1,\textit{h}_2\texttt{)} 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 \(h_2\).

\texttt{Replication(}\textit{h}_1,\textit{h}_2\texttt{) \rightarrow SH-MPC(}\textit{h}_1,\textit{h}_2\texttt{)} Hosts \(h_1\) and \(h_2\) create cleartext input gates.

\texttt{SH-MPC(}\textit{h}_1,\textit{h}_2\texttt{) \rightarrow Local(}\textit{h}_1\texttt{)} 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.

\texttt{SH-MPC(}\textit{h}_1,\textit{h}_2\texttt{) \rightarrow Replication(}\textit{h}_1,\textit{h}_2\texttt{)} Both hosts create a public output gate and execute the MPC circuit. Both hosts receive a cleartext output.

\texttt{Local(}\textit{h}_p\texttt{) \rightarrow Commitment(}\textit{h}_p,\textit{h}_v\texttt{)} 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.

\texttt{Commitment(}\textit{h}_p,\textit{h}_v\texttt{) \rightarrow Local(}\textit{h}_v\texttt{)} 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\((h_p, h_v) \rightsquigarrow \text{Local}(h_v)\) 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). 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.

---

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

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Protocols</th>
<th>Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LAN / WAN</td>
<td>Vars</td>
</tr>
<tr>
<td>battleship</td>
<td>RZ / RZ</td>
<td>1022</td>
</tr>
<tr>
<td>bet</td>
<td>CLRY / CLRY</td>
<td>1022</td>
</tr>
<tr>
<td>biometric match</td>
<td>ALRY / ALRY</td>
<td>708</td>
</tr>
<tr>
<td>guessing game</td>
<td>RZ / RZ</td>
<td>193</td>
</tr>
<tr>
<td>HHI score</td>
<td>ALRY / LRY</td>
<td>285</td>
</tr>
<tr>
<td>historical millionaires</td>
<td>LRY / LRY</td>
<td>187</td>
</tr>
<tr>
<td>interval</td>
<td>RYZ / RYZ</td>
<td>660</td>
</tr>
<tr>
<td>k-means</td>
<td>ARY / RY</td>
<td>1684</td>
</tr>
<tr>
<td>k-means (unrolled)</td>
<td>ARY / RY</td>
<td>3629</td>
</tr>
<tr>
<td>median</td>
<td>RY / RY</td>
<td>386</td>
</tr>
<tr>
<td>rock-paper-scissors</td>
<td>CR / CR</td>
<td>741</td>
</tr>
<tr>
<td>two-round bidding</td>
<td>LRY / LRY</td>
<td>575</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Bool</th>
<th>YAO</th>
<th>Opt-LAN</th>
<th>Opt-WAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>biometric match</td>
<td>3.6</td>
<td>2.8</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>HHI score</td>
<td>0.8</td>
<td>0.5</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>historical millionaires</td>
<td>1.0</td>
<td>0.6</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>k-means</td>
<td>56.5</td>
<td>44.4</td>
<td>17.7</td>
<td>44.4</td>
</tr>
<tr>
<td>median</td>
<td>11.5</td>
<td>12.8</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>2-R bidding</td>
<td>17.3</td>
<td>17.8</td>
<td>3.1</td>
<td>3.1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Bool</th>
<th>YAO</th>
<th>Opt-LAN</th>
<th>Opt-WAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>biometric match</td>
<td>95.9</td>
<td>7.1</td>
<td>2.2</td>
<td>2.2</td>
</tr>
<tr>
<td>HHI score</td>
<td>9.7</td>
<td>1.6</td>
<td>1.1</td>
<td>0.9</td>
</tr>
<tr>
<td>historical millionaires</td>
<td>90.6</td>
<td>1.6</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>k-means</td>
<td>696.1</td>
<td>117.4</td>
<td>35.8</td>
<td>117.4</td>
</tr>
<tr>
<td>median</td>
<td>1098.7</td>
<td>35.4</td>
<td>31.7</td>
<td>31.7</td>
</tr>
<tr>
<td>2-R bidding</td>
<td>184.7</td>
<td>184.5</td>
<td>155.5</td>
<td>155.5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Bool</th>
<th>YAO</th>
<th>Opt-LAN</th>
<th>Opt-WAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>biometric match</td>
<td>56.0</td>
<td>52.3</td>
<td>3.9</td>
<td>3.9</td>
</tr>
<tr>
<td>HHI score</td>
<td>7.0</td>
<td>2.7</td>
<td>0.5</td>
<td>0.6</td>
</tr>
<tr>
<td>historical millionaires</td>
<td>4.8</td>
<td>3.1</td>
<td>0.005</td>
<td>0.005</td>
</tr>
<tr>
<td>k-means</td>
<td>1273.1</td>
<td>1051.3</td>
<td>180.0</td>
<td>1051.3</td>
</tr>
<tr>
<td>median</td>
<td>197.1</td>
<td>327.8</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2-R bidding</td>
<td>233.0</td>
<td>233.0</td>
<td>4.7</td>
<td>4.7</td>
</tr>
</tbody>
</table>

(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
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LAN</th>
<th>WAN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (s)</td>
<td>Slowdown (%)</td>
</tr>
<tr>
<td>biometric match</td>
<td>0.4</td>
<td>150</td>
</tr>
<tr>
<td>HHI score</td>
<td>0.3</td>
<td>0</td>
</tr>
<tr>
<td>historical millionaires</td>
<td>0.3</td>
<td>0</td>
</tr>
<tr>
<td>k-means</td>
<td>1.2</td>
<td>1380</td>
</tr>
<tr>
<td>median</td>
<td>0.5</td>
<td>40</td>
</tr>
<tr>
<td>2-R bidding</td>
<td>1.6</td>
<td>90</td>
</tr>
</tbody>
</table>

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.\footnote{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.}

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 \cite{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.

\section{2.9 Related Work}

\subsection*{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 \cite{51} (e.g., Fairplay \cite{69}, SCVM \cite{65}, ObliVM \cite{66}, OblivC \cite{98}, Wysteria \cite{85}, HyCC \cite{15}, SCALE-MAMBA \cite{3}), and that of zero-knowledge proofs (e.g., Pinocchio \cite{79}, Geppetto \cite{29}, Buffet \cite{95}, xjSNARK \cite{58}).
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.
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, 72, 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
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.

Figure 3.1: Overview of compilation and the correctness proof. Right-to-left arrows are compilation steps; ≤ are proof steps. Term \( w \) is a choreography, \( \llbracket \cdot \rrbracket \) is endpoint projection, source(\( \cdot \)) is the inverse of host selection, and cor(\( \cdot \)) models corruption.
let $a : A = \text{input Alice}$;  
let $b : B = \text{input Bob}$;  
let $x = \text{declassify(endorse } a < \text{endorse } b, A \land B \rightarrow A \sqcap B)$;  
output $x$ to Alice  
output $x$ to Bob

(a) Source program with information-flow labels.

let Alice.$a$ = input;  
Alice.$a$ $\rightsquigarrow$ MPC(Alice, Bob).$a'$;  
let Bob.$b$ = input;  
Bob.$b$ $\rightsquigarrow$ MPC(Alice, Bob).$b'$;  
let MPC(Alice, Bob).$x$ = declassify (endorse $a'$ < endorse $b'$);  
MPC(Alice, Bob).$x$ $\rightsquigarrow$ Alice.$x_1$;  
MPC(Alice, Bob).$x$ $\rightsquigarrow$ Bob.$x_2$;  
output $x_1$ to Alice  
Alice.$0$ $\rightsquigarrow$ Bob.; // Sync outputs  
output $x_2$ to Bob

(b) Intermediate choreography with explicit communication and synchronization.

// Alice  
let $a = \text{input}$;  
send $a$ to MPC(Alice, Bob)  
let $x_1 = \text{receive MPC(…)}$;  
output $x_1$  
send 0 to Bob // Sync  
// MPC(Alice, Bob)  
let $a'$ = receive Alice;  
let $b'$ = receive Bob;  
let $x = \text{declassify (endorse } a' < \text{endorse } b')$;  
send $x$ to Alice  
send $x$ to Bob

// Bob  
let $b = \text{input}$;  
send $b$ to MPC(Alice, Bob)  
let $x_2 = \text{receive MPC(…)}$;  
let _ = receive Alice; // Sync  
output $x_2$

(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 \cite{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 \ref{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. In fig. 3.2a, we suppress these labels for the endorse expressions, but show the declassify expression to move from $A \wedge B$ to $A \cap B$. The label $A \wedge B$ is too secret to allow Alice or Bob to see the value, while the label $A \cap B$ 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, \textit{robust hyperproperty preservation} (RHP) \cite{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 $x \in X$
Labels $\ell \in L$
Hosts $h \in H$

Values $v \in \mathbb{V} \ni 0$
Operators $f \in \mathbb{F}$

Atomic Expr. $t ::= v \mid x$
Expressions $e ::= f(t_1, \ldots, t_n)$
| declassify($t$, $\ell_1 \rightarrow \ell_1$)
| endorse($t$, $\ell_1 \rightarrow \ell_1$)
| input \mid output $t$
| receive $h \mid$ send $t$ to $h$

Statements $s ::= \text{let } h. x = e; s$
| $h_1.t \rightsquigarrow h_2.x; s$
| $h_1[v] \rightsquigarrow h_2; s$
| if $h.t$ then $s_1$ else $s_2$
| case $(h_1 \rightsquigarrow h_2) \{v \mapsto s_v\}_{v \in V}$
| skip

Channel Endpoints $c \in \mathbb{C} = \{\text{Adv, Env}\} \cup H$
Buffers $B \in \mathbb{C} \times \mathbb{C} \rightarrow \mathbb{V}^*$
Processes $w ::= \langle H \subseteq H, B, s \rangle$
Configurations $W ::= w_1 \parallel \cdots \parallel w_n$

Figure 3.3: Unified syntax of source, choreography, and target languages. $\dagger$ terms are choreography only, $\ddagger$ 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 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
$h_1[v] \leadsto h_2; s$ 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 $⟨H, B, s⟩$ 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 $c_1c_2$) 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 $h_1[v] \leadsto h_2; s$, consider this choreography being executed by Alice and Bob:

```plaintext
let Alice.x = input;
if Alice.x then Alice[1] \leadsto Bob; let Bob._ = output 1; skip
else Alice[0] \leadsto 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.
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 $\langle \{\star\}, \epsilon, s \rangle$, where $\star$ is the single locus of ideal program execution and $\epsilon$ 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 $\stackrel{a}{\rightarrow}_i$ gives meaning to source programs and idealized choreographies, and *real* stepping $\stackrel{a}{\rightarrow}_r$ gives meaning to target programs and choreographies that represent target programs. Additionally, we lift ideal and real stepping to concurrent versions, written $\stackrel{a}{\rightarrow}_i^c$ and $\stackrel{a}{\rightarrow}_r^c$, 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 $\rightarrow_i$. 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 $h.e \stackrel{a}{\rightarrow}_i v$ 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 $\text{eval} : F \times V^* \rightarrow V$. 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.
Figure 3.5: Ideal stepping rules for expressions and statements.
For statements, we write \( s \xrightarrow{a} s' \) 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, \( \rightarrow_r \). 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.
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 ($\rightarrow_i$ or $\rightarrow_r$) to a concurrent judgment, written $s \xrightarrow{a}_c s'$. 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: hosts(a) ∩ hosts(E) = Ø. Consider the following program:

```markdown
let Alice.x₁ = input;
Bob.0 ⇝ Alice.x₂;
```

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 →ₐ s', where a ∈ {i, 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 c₁c₂, but messages across different channels may be reordered. A process w forwards its input to its buffer if the
message is addressed to a relevant host; otherwise, w discards the message. A process takes an internal step when its buffer delivers a message to its statement, and an output step when its statement outputs. 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.
source(s) = s'

source(let h.x = e; s) = \begin{align*}
\text{let h.x = e; source(s)} & \quad \text{I/O(e)} \\
\text{let *.x = e; source(s)} & \quad \neg \text{I/O(e)}
\end{align*}

source(h_1,t \leadsto h_2.x; s) = source(s)[t/x]
source(h_1[v] \leadsto h_2; s) = source(s)
source(if h.t then s_1 else s_2) = if *.t then source(s_1) else source(s_2)
source(skip) = skip

I/O(e) = (e = \text{input}) \lor (\exists t . e = \text{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 $\Delta \models s'$ for some $\Delta$.

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 $\Delta \models s$ 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 $\Gamma$ maps a variable to its host and label:

Label Contexts $\Gamma ::= \epsilon \mid \Gamma, x : h.\ell$

Expressions are checked at a particular host $h$ with the judgment $\Gamma \vdash e : h.\ell$, which means that $e$ has label $\ell$ under the context $\Gamma$. Rule $\ell$-Variable ensures hosts only use variables they own. Rules $\ell$-Declassify and $\ell$-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.
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 $\ell$-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 $\ell$-Send ensures secret data is not sent to malicious hosts; it ignores integrity since malicious hosts are untrusted.
\[
\begin{align*}
\Delta \vdash s & \\
\textbf{Sync-External} & \\
\text{external}(e) & \quad \text{reset}(\Delta, h) \vdash s & \text{outputting}(e) \quad \Rightarrow \quad \text{synched}(\Delta, h) \\
\textbf{Sync-Internal} & \\
\text{internal}(e) & \quad \Delta \vdash s & \textbf{Sync-Communicate} & \quad \text{sync}(\Delta, h_1 \rightsquigarrow h_2) \vdash s & \textbf{Sync-Select} & \quad \text{sync}(\Delta, h_1 \rightsquigarrow h_2) \not\vdash s \\
\Delta \vdash \text{let } h.x = e; s & \\
\Delta \vdash h_1.t \rightsquigarrow h_2.x; s & \\
\Delta \vdash h_1[\nu] \rightsquigarrow h_2; s & \textbf{Sync-If} \quad \Delta \vdash \text{if } h.t \text{ then } s_1 \text{ else } s_2 & \textbf{Sync-Skip} & \Delta \vdash \text{skip} \\
\text{synched}(\Delta, h) & \quad \text{reset}(\Delta, h) = \Delta' & \text{sync}(\Delta, h_1 \rightsquigarrow h_2) = \Delta' \\
\end{align*}
\]

\[
\begin{align*}
\text{synched}(\Delta, h) = \forall h'. \Delta(h', h) \subseteq L(h') \lor L(h) \\
\text{reset}(\Delta, h) = \Delta[h, \ast := 1][h, h := L(h)] \\
\text{sync}(\Delta, h_1 \rightsquigarrow h_2) = \Delta[\ast, h_2 := \Delta(\ast, h_2) \land (\Delta(\ast, h_1) \lor L(h_2))] \\
\end{align*}
\]

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 [\textit{\ell-let}] and [\textit{\ell-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 [\textit{\ell-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 [\textit{\ell-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:

```plaintext
let Alice.x_1 = endorse x_guess;
let Bob.x_2 = declassify x_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 `Alice.0 \xrightarrow{} 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 $h$ (Alice $\xrightarrow{} h \xrightarrow{} Bob$) 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 $\Delta$ 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. The integrity of a communication path \( h_1 \leadsto h_2 \leadsto \ldots \leadsto h_n \) can be computed from the hosts along the path:

\[
\mathbb{L}(h_1 \leadsto h_2 \leadsto \ldots \leadsto h_n) = \mathbb{L}(h_1) \leftarrow \mathbb{L}(h_2) \leftarrow \cdots \leftarrow \mathbb{L}(h_n) \leftarrow.
\]

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

\[
\mathbb{L} \left( \text{paths}(h_1, h_2) \right) = \bigwedge_{\text{path} \in \text{paths}(h_1, h_2)} \mathbb{L}(\text{path}).
\]

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 \textbf{Sync-External} checks a \texttt{let} statement that executes an external expression \( e \) on \( h \). The continuation is checked under a context where the label of all paths \textit{from} \( h \) to any other host are set to 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:

\[
\mathbb{L}(\text{paths}(h_1, h_2)) \subseteq \mathbb{L}(h_1) \lor \mathbb{L}(h_2). \tag{3.1}
\]

Rules \textbf{Sync-Communicate} and \textbf{Sync-Select} update \( \Delta \) using \( \text{sync}(\Delta, h_1 \leadsto h_2) \). 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.
\[ s @ H = s' \]

\[
\text{let } h.x = e; s @ H = \begin{cases} 
& \text{let } h.x = e; s @ H \quad h \in H \\
& s @ H \quad \text{o/w}
\end{cases}
\]

\[
h_{1,t} \rightsquigarrow h_{2,x}; s @ H = \begin{cases} 
& h_{1,t} \rightsquigarrow h_{2,x}; s @ H \quad h_{1}, h_{2} \in H \\
& \text{let } h_{1,-} = \text{send } t \text{ to } h_{2}; s @ H \quad h_{1} \in H \\
& \text{let } h_{2,x} = \text{receive } h_{1}; s @ H \quad h_{2} \in H \\
& s @ H \quad \text{o/w}
\end{cases}
\]

\[
h_{1[v]} \rightsquigarrow h_{2}; s @ H = \begin{cases} 
& h_{1[v]} \rightsquigarrow h_{2}; s @ H \quad h_{1}, h_{2} \in H \\
& \text{let } h_{1,-} = \text{send } v \text{ to } h_{2}; s @ H \quad h_{1} \in H \\
& \text{case } (h_{1} \rightsquigarrow h_{2}) \{ v \mapsto s \} \quad h_{2} \in H \\
& s @ H \quad \text{o/w}
\end{cases}
\]

\[
\text{if } h.t \text{ then } s_{1} \text{ else } s_{2} @ H = \begin{cases} 
& \text{if } h.t \text{ then } s_{1} @ H \text{ else } s_{2} @ H \quad h \in H \\
& \text{merge}(s_{1} @ H, s_{2} @ H) \quad \text{o/w}
\end{cases}
\]

\[
\text{skip } @ H = \text{skip}
\]

merge(s_{1}, s_{2}) = s

merge(s_{1}, s_{2}) = \text{case } (h_{1} \rightsquigarrow h_{2}) \{ v \mapsto s_{v} \}_{v \in V_{1} \cup V_{2}}

\begin{align*}
& \text{where } s_{1} = \text{case } (h_{1} \rightsquigarrow h_{2}) \{ v \mapsto s_{v} \}_{v \in V_{1}} \\
& s_{2} = \text{case } (h_{1} \rightsquigarrow h_{2}) \{ v \mapsto s_{v} \}_{v \in V_{2}}
\end{align*}

V_{1} \text{ and } V_{2} \text{ disjoint}

merge(s_{1}, s_{2}) = \text{let } h.x = e; \text{merge}(s'_{1}, s'_{2})

\begin{align*}
& \text{where } s_{1} = \text{let } h.x = e; s'_{1}, s_{2} = \text{let } h.x = e; s'_{2}
\end{align*}

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

\[2\text{Input expressions are } \text{input} \text{ and } \text{endorse}; \text{output expressions are } \text{output} \text{ and } \text{declassify}.\]
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 \cite{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 \cite{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:

$$ (B \circ H)(c_1c_2) = \begin{cases} 
  B(c_1c_2) & c_1 \notin H \land c_2 \in H \\
  \epsilon & \text{otherwise} 
\end{cases} $$

$$ \langle H', B, s \rangle \circ H = \langle H \cap H', B \circ H, s \circ H \rangle. $$

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

$$ [w]_h = w \circ \{h\}. $$

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

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

### 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 $\Gamma \vdash \text{declassify}(t, \ell_t \rightarrow \ell_t) : h.\ell_t, \ell_t \not\in \mathcal{P}$, and $\ell_t \in \mathcal{T}$, then $\ell_t \in \mathcal{T}$.

**Lemma 3.4.2** (Transparent Endorsement). If $\Gamma \vdash \text{endorse}(t, \ell_t \rightarrow \ell_t) : h.\ell_t, \ell_t \not\in \mathcal{T}$, and $\ell_t \in \mathcal{T}$, then $\ell_t \in \mathcal{P}$.

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 \( \ell_1 \) such that \( \ell_1 \sqsubseteq \ell_2 \).
- \( \Delta_1 \sqsubseteq \Delta_2 \) if \( \Delta_1(h_1h_2) \sqsubseteq \Delta_2(h_1h_2) \) 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 \( \Gamma \vdash e[\ell/\ell'] : h.\ell \).
2. If \( (\Gamma, x : h.\ell) \vdash s \), then \( \Gamma \vdash s[\ell/\ell] \).

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 \( \Gamma \vdash \text{cor}(s) \).

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 \( \Delta \vdash \text{cor}(s) \).

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 \( \text{synched}(\Delta, h) \).
3.4.2 Operational Semantics

Below, we write \( \rightarrow \) to stand for any of \( \rightarrow_i \), \( \rightarrow_r \), \( \rightarrow^c_i \), or \( \rightarrow^c_r \).

Processes never refuse input.

**Lemma 3.4.11** (Input Totality). For all \( w \) and \( m \), there exists \( w' \) such that \( w \overset{?m}{\rightarrow} w' \).

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 \( w \overset{a}{\rightarrow} w_1 \) and \( w \overset{a}{\rightarrow} w_2 \), then \( w_1 = w_2 \).

**Lemma 3.4.13** (Output Determinism). If \( w \overset{!m_1}{\rightarrow} w_1 \), \( w \overset{!m_2}{\rightarrow} w_2 \), and \( \text{actor}(!m_1) = \text{actor}(!m_2) \), 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 \( A \) interacting with \( W_1 \) can be simulated by some simulator \( S \) (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

\[
\forall \mathcal{A} \exists S. \mathcal{T}_{\text{Env}}((\mathcal{A} \parallel W_1, \rightarrow_1)) = \mathcal{T}_{\text{Env}}((S \parallel W_2, \rightarrow_2)).
\]

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

\[
\mathcal{T}_{\text{Env}}((W, \rightarrow)) = \{ tr_{\parallel W} \mid W \rightarrow tr \}. 
\]

Restriction \( tr_{\parallel W} \) removes all actions in \( tr \) where neither the source nor the destination is \( \text{Env} \).

An adversary \( \mathcal{A} \) or \( S \) 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 \([W-\text{INPUT}]\) and \([W-\text{OUTPUT}]\).

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

Our definition of simulation guarantees \textit{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 \( w@H \) 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 \texttt{send} (resp. \texttt{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).** \( \text{cor}(w) \) is defined as

\[
\text{cor}(w) = w@H \quad \text{where} \quad H = \{ h \in \mathbb{H} \mid L(h) \in T \}.
\]

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 \( \text{cor}(\cdot) \) to source programs (refer back to fig. 3.1). This means end-to-end security can be stated and understood without reference to \( \text{cor}(\cdot) \), 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 \( \Delta \models w \) if \( \Gamma \vdash s \) and \( \Delta \models s \), respectively, and define \( \text{source}(w) = \langle H, B, \text{source}(s) \rangle \).
**Theorem 3.6.1.** If $\epsilon \vdash w$, and $\Delta \vdash w$ for some $\Delta$, then

$$\langle \llbracket \text{cor}(w) \rrbracket, \rightarrow_r \rangle \leq \langle \text{source}(w), \rightarrow_i \rangle.$$  

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 $\rightarrow^c_r$ to the ideal semantics $\rightarrow^c_i$, 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 $\langle \llbracket \text{cor}(w) \rrbracket, \rightarrow_r \rangle \leq \langle \text{cor}(w), \rightarrow^c_i \rangle$.

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 $S = \mathcal{A}$. 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 $\llbracket w \rrbracket$, 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 $W_1 = \llbracket w_2 \rrbracket$, but this idea fails because $R$ is not preserved under stepping.

**Lemma 3.6.3.** Define $W_1 R w_2$ if $W_1 = \llbracket w_2 \rrbracket$. We claim $R$ is not a bisimulation.

**Proof.** Consider the following choreography and its projection:

```plaintext
// Choreography
w_2 = if Alice.1 then Alice[Bob] \rightarrow 1; s_1 else Alice[Bob] \rightarrow 0; s_2

// Alice
\llbracket w_2 \rrbracket_{Alice} = if Alice.1 then send 1 to Bob; \llbracket s_1 \rrbracket_{Alice} else send 0 to Bob; \llbracket s_2 \rrbracket_{Alice}

// Bob
\llbracket w_2 \rrbracket_{Bob} = case (Alice \rightarrow Bob) \{ 1 \mapsto \llbracket s_1 \rrbracket_{Bob}, 0 \mapsto \llbracket s_2 \rrbracket_{Bob} \}
```

Let $W_1 = \llbracket w_2 \rrbracket$; 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:

```plaintext
// Choreography
w'_2 = Alice[Bob] \rightarrow 1; s_1

// Alice
W'_1(Alice) = send 1 to Bob; \llbracket s_1 \rrbracket_{Alice}

// Bob
W'_1(Bob) = case (Alice \rightarrow Bob) \{ 1 \mapsto \llbracket s_1 \rrbracket_{Bob}, 0 \mapsto \llbracket s_2 \rrbracket_{Bob} \}
```

Note that the process for Bob in $W'_1$ does not match $\llbracket w'_2 \rrbracket_{Bob}$, which is

```
\llbracket w'_2 \rrbracket_{Bob} = case (Alice \rightarrow Bob) \{ 1 \mapsto \llbracket s_1 \rrbracket_{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 $W_1 \succeq \llbracket w_2 \rrbracket$. 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 $W \succeq \llbracket w \rrbracket$ and $w \xrightarrow{t}^\xi w'$, then $W \xrightarrow{a}^t W'$ for some $W' \succeq \llbracket w' \rrbracket$.

**Lemma 3.6.5** (Completeness of Endpoint Projection). If $W \succeq \llbracket w \rrbracket$ and $W \xrightarrow{a}^t W'$, then $w \xrightarrow{a}^t w'$, for some $w'$ with $W' \succeq \llbracket w' \rrbracket$. 

83
Statements $s ::= \ldots$

| $h_1 \rightsquigarrow h_2[v/x]; s$
| $h_1 \rightsquigarrow h_2[v]; s$

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

$$s \xrightarrow{a} s'$$

**s-Communicate-Send**

$$h_1.v \rightsquigarrow h_2.x; s \xrightarrow{h_1h_2a} h_1 \rightsquigarrow h_2[v/x]; s$$

**s-Communicate-Receive**

$$h_1 \rightsquigarrow h_2[v/x]; s \xrightarrow{h_2h_20} s[v/x]$$

**s-Select-Send**

$$h_1[v] \rightsquigarrow h_2; s \xrightarrow{h_1h_2v} h_1 \rightsquigarrow h_2[v]; s$$

**s-Select-Receive**

$$h_1 \rightsquigarrow h_2[v]; s \xrightarrow{h_2h_20} s[v/x]$$

Figure 3.14: Stepping rules for asynchronous choreographies. These override the rules for $\rightarrow_r$ 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 $\text{cor}(\cdot)$. 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.\(v\) \(\rightarrow\) Bob.\(x\); \(s\) becomes a send on Alice and a receive on Bob, Alice \(\rightarrow\) 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 \(\langle \text{cor}(w), \rightarrow_r \rangle \leq \langle \text{cor}(w), \rightarrow^c_r \rangle\).

Proof. Let \(S = A\). Lemmas 3.6.4 and 3.6.5 immediately give a strong bisimulation. \(\square\)

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 \(\langle \text{cor}(w), \rightarrow^c_r \rangle \leq \langle \text{cor}(w), \rightarrow^c_r \rangle\).
Proof. The simulator follows the control flow by maintaining a public view of the extended choreography \( \langle \text{cor}(w), \rightarrow \rangle \) 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).

\[ \square \]

Proof of theorem 3.6.2. By lemmas 3.6.6 and 3.6.7 using the transitivity of UC simulation.

\[ \square \]

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 \( \langle \text{cor}(w), \rightarrow \rangle \leq \langle \text{cor}(w), \rightarrow^* \rangle \).

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 $\text{cor}(w)$ when Alice is malicious:

```plaintext
// w

let Alice.x = input;
Alice.x ⇝ Bob.y;
Alice.x ⇝ Chuck.z;
let Bob.y' = y + 1;
Bob.y' ⇝ Alice.x';

// cor(w)

let Bob.y = receive Alice;
let Chuck.z = receive Alice;
let Bob.y' = y + 1;
let Bob._ = send y' to Alice;
```

The function $\text{cor}(\cdot)$ 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 $\ell$-Send and $\ell$-Communicate).

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

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

The core of the simulation result is showing that the simulator can use declassify expressions to recreate all data no longer leaked through communication, and endorse expressions to influence all data no longer corruptible through receive expressions. For this to work, we need to ensure the ideal choreography outputs the correct value to
the simulator when performing a `declassify`, and we need to ensure the simulator can input the correct value to the ideal choreography for an `endorse`. For example, when the ideal choreography performs `declassify` \( x \), we must ensure the value of \( x \) is the same in the real and ideal choreographies. This is nontrivial since the ideal semantics replaces all untrusted data with 0. *Robust declassification* requires only trusted data is declassified, and type checking ensures untrusted data does not influence trusted data. Thus, \( x \) is trusted and erased values cannot influence its value. Similarly, when the ideal choreography performs `endorse` \( x \), the 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 \( \sigma : \text{dom}(\Gamma) \rightarrow \mathbb{V} \).
**Definition 3.6.10** (Channel Label). A channel’s label derives from the labels of its endpoints:

\[ \mathbb{L}(c_1c_2v) = \mathbb{L}(c_1c_2) = \mathbb{L}(c_1) \lor \mathbb{L}(c_2) \quad \mathbb{L}(?m) = \mathbb{L}(!m) = \mathbb{L}(m). \]

We let \( \mathbb{L}(\text{Adv}) = 0 \) and \( \mathbb{L}(\text{Env}) = 0 \), which leads to \( \mathbb{L}(\text{Env}c) = \mathbb{L}(c\text{Env}) = \mathbb{L}(c) \) and \( \mathbb{L}(\text{Adv}c) = \mathbb{L}(c\text{Adv}) = \mathbb{L}(c)^{\text{e}} \) (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.

- \( c_1c_2v_1 =_{L} c_1c_2v_2 \) if \( \mathbb{L}(c_1c_2) \in L \) 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 \( \Gamma_1, \Gamma_2 \), 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 \( \ell \not\in L \).
- \( B_1 =_{L} B_2 \) if \( B_1(c_1c_2) = B_2(c_1c_2) \) for all \( c_1 \) and \( c_2 \) such that \( \mathbb{L}(c_1c_2) \in L \).
- \( \langle H, B_1, s_1 \rangle =_{L} \langle H, B_2, s_2 \rangle \) 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 value of $x$:

```
// s1
let Alice.x = 0;
Alice.x ⇝ Bob.y;

// s2
let Alice.x = 1;
Alice.x ⇝ 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 abstracts out values where the two statements 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 ⇝ 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 $s_1 = s_2$. 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 $\Gamma_1$ and $\Gamma_2$, with the substitutions only assigning values for variables in $\Gamma_2$. Context $\Gamma_1$ 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 $L(h) \notin T$ (this is ensured by $\text{cor}()$).\footnote{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 \( s_1 \equiv_L s_2 \) and \( s_1 \overset{a_1 \xi}{\rightarrow} s_1' \) without using rule \( \textit{e-DECLASSIFY-REAL} \) or rule \( \textit{e-ENDORSE-REAL} \)

- If \( a_1 = ?m_1 \), then \( s_2 \overset{?m_2 \xi}{\rightarrow} s_2' \) with \( s_1' \equiv_L s_2' \) for all \( m_2 \equiv_L m_1 \).
- If \( a_1 = \!m_1 \), then \( s_2 \overset{!m_2 \xi}{\rightarrow} s_2' \) with \( s_1' \equiv_L s_2' \) for some \( m_2 \) with \( \text{actor}(!m_2) = \text{actor}(!m_1) \).

**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 \( \sigma_2 \) substitutes for the same variables as \( \sigma_1 \). We appeal to this fact implicitly.

- Case \( s\text{-LET} \) We have
  \[
s_1 = \text{let } h.x = e_1; s_1'' \quad s_2 = \text{let } h.x = e_2; s_2'' \quad s = \text{let } h.x = e; s''
  \]
  and
  \[
h.e_1 \overset{a_1}{\rightarrow} v_1 \\
h.e_2 \overset{a_2}{\rightarrow} v_2
  \]
  with \( \text{actor}(a_1) = \text{actor}(a_2) = h \). Inversion on \( \Gamma \vdash s \) gives
  \[
  \begin{array}{c}
  \text{RULE } \ell\text{-LET} \\
  \Gamma \vdash e : h.\ell \\
  \llbracket(h) \Rightarrow \ell \\
  \Gamma, x : h.\ell \vdash s'' \\
  \hline
  \end{array}
  \]
  \( \Gamma \vdash s \)
  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 \( s_1' \equiv_L s_2' \).
When \( \ell \not\in \mathcal{P} \), 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 \( \sigma'_1(s'') = s''[v_1/x] = s'_1 \) and \( \sigma'_2(s'') = s''[v_2/x] = s'_2 \). Note that \( \Gamma_2 \) satisfies the requirements of definition 3.6.11 and \( \sigma'_1, \sigma'_2 : \Gamma'_2 \), so \( s'_1 \equiv_L s'_2 \).

We case on the expression stepping relation to show one of the requirements.

- **Case** `e-Operator`. We have

\[
e_1 = f(t^1_1, \ldots, t^n_1) \quad e_2 = f(t^1_2, \ldots, t^n_2) \quad e = f(t^1, \ldots, t^n).
\]

If all \( t^i \) are values, then \( t^i_1 = t^i = t^i_2 \) 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 \( \ell' \not\in \mathcal{L} \) and \( \ell' \not\subseteq \ell \), which implies \( \ell \not\in \mathcal{L} \).

- **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

\[
e_1 = e_2 = e = \text{input}.
\]

Assume \( \ell \in \mathcal{L} \) since we are done otherwise. Inversion on \( \Gamma \vdash e : h.\ell \) gives \( \mathbb{L}(h) \subseteq \ell \), so \( \mathbb{L}(h) \in \mathcal{L} \), which means \( \mathbb{L}(\text{Env}h) = \mathbb{L}(h) \in \mathcal{L} \). 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

\[
e_1 = e_2 = e = \text{receive} h'.
\]

We have \( \mathbb{L}(h') \not\in \mathcal{T} \) by assumption, and \( \mathbb{L}(h') \in \mathcal{P} \) by definition 2.2.1

93
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 $\mathbb{L}(h') \nsubseteq \ell$. Since $\mathbb{L}(h') \notin \mathcal{T} = L$, we have $\ell \notin L$.

- **Case $\mathbf{e}$-\texttt{Send-Real}** We have $v_1 = 0 = v_2$.

  - **Case $\mathbf{s}$-\texttt{Communicate-Real}** We have
    \[
    s_1 = h_1.v_1 \leadsto h_2.x; s'_1 \\
    s_2 = h_1.v_2 \leadsto h_2.x; s'_2 \\
    s = h_1.t \leadsto h_2.x; s''
    \]
    and
    \[
    s_1 \xrightarrow{!h_1.h_2.v_1} \rightarrow_s s'_1[v_1/x] \\
    s_2 \xrightarrow{!h_1.h_2.v_2} \rightarrow_s s''_2[v_2/x].
    \]
    By inversion on $\Gamma \vdash s$, we have
    \[
    \text{(rule $\ell$-\texttt{Communicate})} \\
    \Gamma \vdash t : h_1.\ell \\
    \mathbb{L}(h_2) \Rightarrow \ell \\
    \Gamma, x : h_2.\ell \vdash s''
    \]
    \[
    \Gamma \vdash h_1.t \leadsto h_2.x; s''
    \]
    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 $3.4.5$ and $\sigma_1(s''[v/x]) = s''_1[v/x]$, 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 $x' \in \text{dom}(\Gamma_2)$. By inversion on $\Gamma \vdash t : h_1.\ell$, we have $(x : h_1.\ell') \in \Gamma_2$ for some $\ell' \subseteq \ell$. Since $\ell' \notin L$ and $\ell' \subseteq \ell$, 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 $\Gamma_2$ 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 $\mathbf{s}$-\texttt{Select-Real}** We have
    \[
    s_1 = h_1[v_1] \leadsto h_2; s'_1 \\
    s_2 = h_1[v_2] \leadsto h_2; s'_2 \\
    s = h_1[v] \leadsto h_2; s'
    \]
    and
    \[
    s_1 \xrightarrow{!h_1.h_2.v_1} \rightarrow_s s'_1 \\
    s_2 \xrightarrow{!h_1.h_2.v_2} \rightarrow_s s'_2.
    \]
Γ ⊨ s' (by inversion on Γ ⊨ s), σ₁(s′) = s′₁, and σ₂(s′) = s′₂, thus s′₁ =_L s′₂.
- Case **s-If** We have
  
  \[ s_1 = \text{if } h.v_1 \text{ then } s_1^1 \text{ else } s_2^2 \quad s_2 = \text{if } h.v_2 \text{ then } s_1^1 \text{ else } s_2^2 \]

  \[ s = \text{if } h.t \text{ then } s_1 \text{ else } s_2 \]

  and

  \[ s_1 \xrightarrow{\text{if}_h v_0} s_1^i \quad s_2 \xrightarrow{\text{if}_h v_0} s_2^j. \]

  Inversion on Γ ⊨ s (which must be by rule **s-If**) gives Γ ⊨ t : h.0\. Additionally, σ₁(t) and σ₂(t) are values, so free(t) ⊆ Γ₂, meaning Γ₂ ⊨ t : h.0\. Since 0\. ∈ L for all attacks (recall definition 2.2.1), and Γ₂ only contains variables with labels not in L, t must be a value, that is, t = v for some v. Then, v₁ = σ₁(v) = v = σ₂(v) = v₂, so i = j. Finally, we have s₁ =_L s₂ since Γ ⊨ s₁ (by inversion on Γ ⊨ s), σ₁(sᵢ) = s₁ᵢ, and σ₂(sᵢ) = s₂ᵢ = s₂ᵢ.

- Case **s-Case** Impossible by inversion on Γ ⊨ s.

- Case **s-Sequential** Immediate by the induction hypothesis.

- Case **s-Delay** We have
  
  \[ s_1 = E_1[s''_1] \quad s_2 = E_2[s''_2] \quad s = E[s''] \]

  and

  || \[ s''_1 \xrightarrow{a^c} s'''_1 \quad \text{actor}(a) \notin \text{hosts}(E_1) \] \[ s_1 \xrightarrow{a^c} E_1[s'''_1] \] || \[ s''_2 \xrightarrow{a^c} s'''_2 \quad \text{actor}(a) \notin \text{hosts}(E_1) \] \[ s_2 \xrightarrow{a^c} E_2[s'''_2] \]

  Note that (Γ₁, Γ'₁, Γ₂) ⊨ s'' when Γ'₁ are the variables defined by E (which must be the same as the ones defined by E₁ and E₂). Thus, s''₁ =_L s''₂ through s'', σ₁, and σ₂, and we can apply induction hypothesis to get s'''₁ =_L s'''₂. This then gives s₁ = E₁[s''₁] =_L E₂[s''₂] s₂.
Public-equivalent choreographies produce public-equivalent outputs.

**Lemma 3.6.13 (Public Outputs).** If \( s_1 \equiv \rho s_2 \), \( s_1 \xrightarrow{\ell m_1} r \), and \( s_2 \xrightarrow{\ell a_2} r \) with \( \text{actor}(!m_1) = \text{actor}(a_2) \), then \( !m_1 \equiv \rho a_2 \).

**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 \( s\text{-Delay} \) and \( s\text{-If-Delay} \) follow from the induction hypotheses. Cases for rules \( e\text{-Operator} \), \( e\text{-Declassify-Skip} \), \( e\text{-Endorse-Skip} \), \( e\text{-Output-Malicious} \), \( s\text{-If} \), \( e\text{-Declassify-Real} \) and \( e\text{-Endorse-Real} \) are immediate since both actions are internal, i.e., \( !m_1 = !hh0 = a_2 \) for some \( h \), which implies \( !m_1 \equiv \rho a_2 \). We detail the remaining cases.

- Case \( s\text{-Let} \) We have

\[
s_1 = \text{let } h.x = e_1; s'_1 \quad s_2 = \text{let } h.x = e_2; s'_2 \quad s = \text{let } h.x = e; s'
\]

and

\[
h.e_1 \xrightarrow{\ell hcv_1} r \quad \quad h.e_2 \xrightarrow{\ell hcv_2} \sim \ell.
\]

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

- Case \( e\text{-Output} \) We have \( c = \text{Env}, L(h) \in \mathcal{T} \), and

\[
e_1 = \text{output } v_1 \quad e_2 = \text{output } v_2 \quad e = \text{output } t.
\]

If \( L(h) \notin \mathcal{P} \), then \( !m_1 = !h\text{Env}v_1 = \rho !h\text{Env}v_2 = a_2 \) immediately, so assume \( L(h) \in \mathcal{P} \). Inversion on \( \Gamma \vdash e : h.\ell \) gives \( \Gamma \vdash t : h.L(h) \). Since \( L(h) \in \mathcal{P} \), \( t = v \) for some \( v \), meaning \( v_1 = \sigma_1(t) = v = \sigma_2(t) = v_2 \), so \( !m_1 = !h\text{Env}v = !h\text{Env}v = a_2 \), and \( !m_1 \equiv \rho a_2 \).
Case e-SEND-REAL We have $c = h'$ and

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

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 $\ell' \subseteq \mathbb{L}(h') \rightarrow$. Then,

$$\ell' \notin \mathcal{P} \land \mathbb{L}(h) \Rightarrow \ell' \Rightarrow \mathbb{L}(h) \notin \mathcal{P}$$

$$\ell' \subseteq \mathbb{L}(h') \rightarrow \land \ell' \notin \mathcal{P} \Rightarrow \mathbb{L}(h') \notin \mathcal{P} \Rightarrow \mathbb{L}(h') \notin \mathcal{P}.$$

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

Case s-COMMUNICATE-REAL We have

$$s_1 = h_1.v_1 \leadsto h_2.x; s_1'' \quad s_2 = h_1.v_2 \leadsto h_2.x; s_2'' \quad s = h_1.t \leadsto h_2.x; s''$$

and

$$s_1 \xrightarrow{!h_1h_2v_1} \tau \quad s_2 \xrightarrow{!h_1h_2v_2} \text{sim}.$$

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

$$\text{rule } \ell'-\text{COMMUNICATE}$$

$$\Gamma \vdash t : h_1.\ell \quad \mathbb{L}(h_2) \Rightarrow \ell \quad \Gamma, x : h_2.\ell \vdash s''$$

$$\Gamma \vdash h_1.t \leadsto h_2.x; s''$$

We case on $t$. If $t = v$ for some $v$, then $v_1 = \sigma_1(v) = v = \sigma_2(v) = v_2$. So $a_1 = !h_1h_2v = a_2$ and $a_1 = \rho \ a_2$.

Otherwise, $t = x'$ for some $x' \in \text{dom}(\Gamma_2)$. By inversion on $\Gamma \vdash t : h_1.\ell$, we have $(x : h_1.\ell') \in \Gamma_2$ for some $\ell' \subseteq \ell$. Then,

$$\ell' \notin \mathcal{P} \land \ell' \subseteq \ell \Rightarrow \ell \notin \mathcal{P}$$

$$\ell' \notin \mathcal{P} \land \mathbb{L}(h_1) \Rightarrow \ell' \Rightarrow \mathbb{L}(h_1) \notin \mathcal{P}$$

$$\ell \notin \mathcal{P} \land \mathbb{L}(h_2) \Rightarrow \ell \Rightarrow \mathbb{L}(h_2) \notin \mathcal{P}$$

$$\mathbb{L}(h_1) \notin \mathcal{P} \land \mathbb{L}(h_2) \notin \mathcal{P} \Rightarrow \mathbb{L}(h_1h_2) \notin \mathcal{P}.$$
Since $L(h_1h_2) \not\in P$, $a_1 = !h_1h_2v_1 = \rho !h_1h_2v_2 = a_2$.

- Case $s$-SELECT-REAL We have

\[ s_1 = h_1[v_1] \rightsquigarrow h_2; s'_1 \quad s_2 = h_1[v_2] \rightsquigarrow h_2; s'_2 \quad s = h_1[v] \rightsquigarrow h_2; s' \]

and

\[ s_1 \xrightarrow{!h_1h_2v_1} r \quad s_2 \xrightarrow{!h_1h_2v_2} \sim \]

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

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 $s_1 = \rho s_2$, $s_1 \xrightarrow{!hEnvv_1} c$, and $s_2 \xrightarrow{a_2} c$ with $\text{actor}(a_2) = h$, then $!hEnvv_1 = \rho a_2$.

**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 $s_1 = \rho s_2$ and $s_1 \xrightarrow{s_1} c$, $s'_1$ without using rule $e$-DECLASSIFY-REAL or $e$-ENDORSE-REAL.
• If \( a_1 = m_1 \), then \( s_2 \xrightarrow{m_2} \text{sim} s'_2 \) with \( s'_1 = \sim_1 s'_2 \) for all \( m_2 = \sim_1 m_1 \).

• If \( a_1 = !m_1 \), then \( s_2 \xrightarrow{m_2} \text{sim} s'_2 \) with \( s'_1 = \sim_1 s'_2 \) for some \( m_2 = \sim_1 m_1 \).

In addition, the statement holds with the roles of \( \text{→}_{\text{r}}^\xi \) and \( \text{→}_{\text{sim}}^\xi \) reversed (excluding rules \text{e-Declassify-Simulator} and \text{e-Endorse-Simulator} instead).

Proof. Follows immediately from lemmas 3.6.12 and 3.6.13 since \( \text{→}_{\text{sim}}^\xi \) is equivalent to \( \text{→}_{\text{r}}^\xi \) except for rules \text{e-Declassify-Real} and \text{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 \( s_1 =_T s_2 \) and \( s_1 \xrightarrow{a_1}^\xi s'_1 \) without using rule \text{e-Declassify-Real} or \text{e-Endorse-Real}.

• If \( a_1 = ?\text{Env}_{hv} \), then \( s_2 \xrightarrow{a_2}^\xi s'_2 \) with \( s'_1 =_T s'_2 \) for all \( a_2 =_T a_1 \).

• If \( a_1 = !h\text{Env}_{v} \), then \( s_2 \xrightarrow{a_2}^\xi s'_2 \) with \( s'_1 =_T s'_2 \) for some \( a_2 =_T a_1 \).

• Otherwise, \( s_2 \xrightarrow{a_2}^\xi s'_2 \) with \( s'_1 =_T s'_2 \) and \( \text{actor}(a_1) = h \).

In addition, the statement holds with the roles of \( \text{→}_{\text{r}}^\xi \) and \( \text{→}_{\text{i}}^\xi \) reversed (excluding rules \text{e-Declassify} and \text{e-Endorse} instead).

Proof. Follows from lemmas 3.6.12 and 3.6.14. Judgment \( \text{→}_{\text{r}}^\xi \) behaves the same as \( \text{→}_{\text{i}}^\xi \) 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 \text{e-Receive} and \text{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 \( s_1 \models \rho s_2 \) and \( s_1 \models \tau s_3 \). If \( s_1 \xrightarrow{\text{Adv} \cdot \tau} s'_1 \), \( s_2 \xrightarrow{\text{Adv} \cdot \text{sim}} s'_2 \), and \( s_3 \xrightarrow{\text{Adv} \cdot \text{sim}} s'_3 \), then \( s'_1 \models \rho s'_2 \) and \( s'_1 \models \tau s'_3 \).

**Proof.** By definition 3.6.11 there exist \( s_p \) and \( s_1 \) such that \( \Gamma_p \vdash s_p, \sigma_1(s_p) = s_1, \sigma_2(s_p) = s_2 \), and \( \Gamma_t \vdash s_1, \sigma_2'(s_1) = s_2, \sigma_3(s_1) = s_3 \) for \( \Gamma_p = (\Gamma_1, \Gamma_2), \sigma_1, \sigma_2 : \Gamma_2, \Gamma_1 = (\Gamma_3, \Gamma_4) \), and \( \sigma_2', \sigma_3 : \Gamma_4 \).

We proceed by induction on the stepping relations. Inductive cases (rules \( s\text{-DELAY} \) and \( s\text{-IF-Delay} \)) are handled similarly to lemma 3.6.15. The only remaining case is when the steps are by rules \( e\text{-DECLASSIFY} \), \( e\text{-DECLASSIFY-REAL} \), and \( e\text{-DECLASSIFY-SIMULATOR} \) respectively. We have

\[
\begin{align*}
s_1 &= \text{let } h.x = \text{declassify}(v_1, \ell_t \rightarrow \ell_t); s''_1, \quad s_p = \text{let } h.x = \text{declassify}(t_p, \ell_t \rightarrow \ell_t); s''_p \smallskip \\
s_2 &= \text{let } h.x = \text{declassify}(v_2, \ell_t \rightarrow \ell_t); s''_2, \quad s_t = \text{let } h.x = \text{declassify}(t_t, \ell_t \rightarrow \ell_t); s''_t \smallskip \\
s_3 &= \text{let } h.x = \text{declassify}(v, \ell_t \rightarrow \ell_t); s''_3
\end{align*}
\]

where \( \ell_t \not\in \mathcal{P} \) and \( \ell_t \in \mathcal{P} \), and

\[
\begin{align*}
s'_1 &= s''_1[v_1/x], \\
s'_2 &= s''_2[v/x], \\
s'_3 &= s''_3[v/x].
\end{align*}
\]

We claim \( v_1 = v \) (\( v_2 \) is ignored by \( s_2 \), so it is irrelevant). By lemma 3.4.1 and inversion on \( \Gamma_t \vdash s_1 \), we have \( t_t \in \mathcal{T} \). Assume for contradiction that \( t_t = x_t \) for some \( x_t \). Then, \( (x_t : h.t) \in \mathcal{G}_4 \) for \( t \subseteq \ell_t \). However, \( t \not\in \mathcal{T} \) so \( \ell_t \not\in \mathcal{T} \), which is a contradiction. Thus, \( t_t = v_t \) for some \( v_t \). Then, \( v_1 = \sigma_2'(t_t) = \sigma_2'(v_t) = v_t = \sigma_3(v_t) = \sigma_3(t_t) = v \).
Finally, let \( s'_p = s''_p[v/x] \) and \( s'_t = s''_t[v/x] \). We have \( s'_1 =_p s'_2 \) since \( \Gamma_p \vdash s'_p \) (inversion on \( \Gamma_p \vdash s_p \) followed by lemma 3.4.5), \( \sigma_1(s'_p) = s'_1 \), and \( \sigma_2(s'_p) = s'_2 \); and we have \( s'_2 =_r s'_3 \) since \( \Gamma_t \vdash s'_t \) (inversion on \( \Gamma_t \vdash s_t \), then lemma 3.4.5), \( \sigma'_2(s'_t) = s'_2 \), and \( \sigma'_3(s'_t) = s'_3 \). □

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 \( s_1 =_p s_2 \) and \( s_1 =_r s_3 \). If \( s_1 \xrightarrow{1_{hh0}} c s'_1 \), \( s_2 \xrightarrow{\text{Adv\_sim}} s'_2 \), and \( s_3 \xrightarrow{?_{Adv\_1}} c s'_3 \), then \( s'_1 =_p s'_2 \) and \( s'_1 =_r s'_3 \).

Proof. Dual to lemma 3.6.17 but focusing on \( s_1 =_p s_2 \) 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 \( B_1 =_p B_2 \), 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 \( S(\mathcal{A} \parallel w) \) 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.

**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 $\mathcal{A} \parallel w_1 R S (\mathcal{A}' \parallel w) \parallel w_2$ 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]).


\[ \square \]

### 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 $\Delta$, then $\langle \text{cor}(w), \neg \rightarrow_i \rangle \leq \langle \text{cor}(w), \neg \rightarrow_i \rangle$.

---

⁴The 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 \( w_1 \xrightarrow{tr_1} w \) and \( w_2 \xrightarrow{tr_2} w \) for some \( w \). Diagrammatically:

\[
\begin{array}{c}
  w_1 \\
  \overset{\text{tr}_1}{\downarrow} \in w \\
  \overset{\text{tr}_2}{\downarrow} \\
  w_2
\end{array}
\]

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 \( a_1 \perp a_2 \), if one is an input while the other is an output, or they are on different channels. We write \( tr_1 \perp tr_2 \) if \( a_1 \perp a_2 \) for all \( a_1 \in tr_1 \) and \( a_2 \in tr_2 \).
**Lemma 3.6.22** (Diamond for Processes). If \( w \xrightarrow{tr_1} c w_1, \ w \xrightarrow{tr_2} c w_2, \) and \( tr_1 \perp tr_2, \) then \( w \xrightarrow{tr_1} c w' \) and \( w \xrightarrow{tr_2} c w' \) for some \( w'. \) Diagrammatically:

![Diagram for Lemma 3.6.22]

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 \( s \xrightarrow{a_1} s_1, \ s \xrightarrow{a_2} c s_2, \) and \( a_1 \perp a_2, \) then \( s \xrightarrow{a_1} c s' \) and \( s \xrightarrow{a_2} c s' \) for some \( s'. \)

**Proof.** By case analysis on \( s \xrightarrow{a_2} c s_2. \)

- **Case s-SEQUENTIAL.** Contradicts \( a_1 \perp a_2. \)
- **Case s-DELAY.** By case analysis on the evaluation context followed by inversion on \( s \xrightarrow{a_1} s_1. \) 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. \( \square \)

**Lemma 3.6.24** (Diamond for Statements). If \( s \xrightarrow{a_1} c s_1, \ s \xrightarrow{a_2} c s_2, \) and \( a_1 \perp a_2, \) then \( s \xrightarrow{a_1} c s' \) and \( s \xrightarrow{a_2} c s' \) for some \( s'. \)
Proof. By induction on the derivations of both stepping judgments. If either is by rule \texttt{s-SEQUENTIAL}, we conclude by lemma \ref{lem: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. \hfill \Box

Proof of lemma \ref{lem: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 \texttt{w-INPUT} and \texttt{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 \texttt{w-INTERNAL}. The input step adds a message to a queue, while rule \texttt{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 \texttt{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 \texttt{w-INTERNAL} and \texttt{w-OUTPUT}). If either step is by rule \texttt{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 \ref{lem:3.6.24} finishes the proof. \hfill \Box

The proof of lemma \ref{lem: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 $\rightarrow_1$ and $\rightarrow_2$. If $w \xrightarrow{tr_1} w_1$, $w \xrightarrow{tr_2} w_2$, and $tr_1 \perp tr_2$, then $w_1 \xrightarrow{tr_2} w'$ and $w_2 \xrightarrow{tr_1} w'$ 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 $w_1 \xrightarrow{a_i} w_1'$ for a internal, then $w_1' \downarrow w_2$.

*Diagrammatically:*

\[
\begin{array}{c}
\xymatrix{ w_1' \ar[d]^a & w_1 \ar[ld]^{tr_1} \ar[rd]^{tr_2} & w_2 \\
\exists w' & w \ar[ld]^{tr_1'} \ar[rd]^{tr_2'} & w_2} \\
\end{array}
\]

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

- **Case $a \perp tr_1$.** Lemma 3.6.22 gives $w'$ such that $w_1' \xrightarrow{tr_1} w'$ and $w \xrightarrow{a_i} w'$. Since $a$ and $tr_2$ are internal, so is $tr_2 \cdot a$. Thus, $w_1' \downarrow w_2$ through $tr_1$ and $tr_2 \cdot a$.

\[
\begin{array}{c}
\xymatrix{ w_1' \ar[d]^a & w_1 \ar[ld]^{tr_1} \ar[rd]^{tr_2} & w_2 \\
\exists w' & w \ar[ld]^{tr_1'} \ar[rd]^{a} & w_2} \\
\end{array}
\]

- **Case $a \not\perp tr_1$.** Let $a_1$ be the first action in $tr_1$ such that $a \not\perp a_1$, that is, $tr_1 = tr_1' \cdot a_1 \cdot tr_1''$ with $a \perp tr_1'$. We have, $w_1 \xrightarrow{tr_1'} w' \xrightarrow{a_1} w'' \xrightarrow{tr_1''} w$. Lemma 3.6.22 gives
such that \( w' \xrightarrow{t_{r_1}} w'' \) and \( w' \xrightarrow{a} w_1' \). We now have \( w' \xrightarrow{a} w'' \) and \( w' \xrightarrow{a} w'' \), however, the stepping judgment is deterministic on dependent internal actions, thus lemma 3.4.13 gives \( w'' = w'' \).

Finally, \( w' \downarrow w_2 \) through \( t_{r_1} \cdot t_{r_2} \).

\[ \exists w'' \]

\[ \text{Lemma 3.6.22} \]

\[ \text{Lemma 3.4.13} \]

\[ \square \]

**Lemma 3.6.27 (Matching Actions).** If \( w_1 \downarrow w_2 \), \( w_1 \xrightarrow{a} w_1' \), and \( w_2 \xrightarrow{a} w_2' \), then \( w_1' \downarrow w_2 \). Diagrammatically:

\[ \exists w' \]

\[ \text{Lemma 3.6.22} \]

\[ \text{Lemma 3.6.22} \]

\[ \text{Lemma 3.4.12} \]

Proof. Since \( w_1 \) and \( w_2 \) are joinable, there exist \( w \) and internal \( t_{r_1}, t_{r_2} \) such that \( w_1 \xrightarrow{t_{r_1}} w \) and \( w_2 \xrightarrow{t_{r_2}} w \). If \( a \) is internal, then the result follows by two applications of lemma 3.6.26. Otherwise, \( a \perp t_{r_1} \) and \( a \perp t_{r_2} \). The result follows from two applications of lemma 3.6.22 and one application of lemma 3.4.12.

More specifically, \( a \) and \( a_1 \) are internal actions, which are represented as outputs. Since \( a \perp a_1 \), we have \( \text{actor}(a) = \text{actor}(a_1) \), 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 $w \xrightarrow{\text{m}}_1 w'$ if $w \xrightarrow{\text{tr} \cdot \text{m}}_1 w'$ for some internal $\text{tr}$.

**Lemma 3.6.28** (Sequential Execution). If $\epsilon \vdash w$, $\Delta \models w$, and $w \xrightarrow{\text{m}}_1$ for $m$ external, then $w \xrightarrow{\text{m}}_1$.

*Proof sketch.* By induction on the stepping relation. If the step is by rule $s$-Sequential, then $w \xrightarrow{\text{m}}_1$ 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 $w_1 \xrightarrow{a_1}_i w_2 \xrightarrow{a_2}_i$, then either $w_1 \xrightarrow{a_1}_i w_2$, or $w_1 \xrightarrow{a_2}_i$ and $a_1 \perp a_2$.

*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 $a_1 \perp a_2$ from the side conditions on delay rules, which require actor($a_1$) to be different from actor($a_2$). □

**Lemma 3.6.30** (Delayed Step). If $w_1 \xrightarrow{a_1}_i w_2 \xrightarrow{\text{m}}_1$ for a internal, then $w_1 \xrightarrow{\text{m}}_1$.

*Proof.* Unfolding the definition of $\xrightarrow{\text{m}}_1$ gives $w_2 \xrightarrow{\text{tr}}_1 w_3 \xrightarrow{\text{m}}_1$ for some $\text{tr}$ and $w_3$. We proceed by induction on $\text{tr}$. In either case, we are done if $w_1 \xrightarrow{a}_i w_2$, so assume $w_1$ cannot perform $a$ sequentially.
• Case $tr = \varepsilon$. We have $w_1 \xrightarrow{a_i^c} w_2 \xrightarrow{!m_i}$. Lemma 3.6.29 gives $w_1 \xrightarrow{!m_i}$, and thus $w_1 \xrightarrow{!m_i}$.

• Case $tr = a' \cdot tr'$. We have $w_1 \xrightarrow{a_i^c} w_2 \xrightarrow{a'_i} w_3 \xrightarrow{tr'} w_4 \xrightarrow{!m_i}$. Lemma 3.6.29 gives $w_1 \xrightarrow{a'_i} w_2'$ for some $w_2'$. Lemma 3.6.25 with lemma 3.6.23 then gives $w_1 \xrightarrow{a'_i} w_2' \xrightarrow{a_i^c} w_3'$ for some $w_3'$, and lemma 3.4.12 shows $w_3' = w_3$. We use the induction hypothesis on $w_2' \xrightarrow{a_i^c} w_3 \xrightarrow{tr'} w_4 \xrightarrow{!m_i}$ to get $w_2' \xrightarrow{!m_i}$, and combined with $w_1 \xrightarrow{a'_i} w_2'$, we get $w_1 \xrightarrow{!m_i}$.

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 $w_1 \downarrow w_2$, $\varepsilon \vdash w_2$, and $\Delta \vDash w_2$. If $w_1 \xrightarrow{!m_i}$ for $m$ external, then $w_2 \xrightarrow{!m_i}$.

**Proof.** Since $w_1$ and $w_2$ are joinable, there exist $w$ and internal $tr_1$, $tr_2$ such that $w_1 \xrightarrow{tr_1^c} w$ and $w_2 \xrightarrow{tr_2^c} w$. Because $m$ is external and $tr_1$ is internal, we have $!m \perp tr_1$, so lemma 3.6.22 gives $w \xrightarrow{!m_i}$. Now we have $w_2 \xrightarrow{tr_2 \cdot !m_i}$, and we want to show $w_2 \xrightarrow{!m_i}$.

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 \( w_2 \xrightarrow{a}^c i w'_2 \xrightarrow{m} \), and lemma 3.6.30 gives the desired result.

\[ \square \]

**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 \( S(\mathcal{A} \parallel w'_1, w'_2) \) 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 \( \mathcal{A} \parallel w_1 R S(\mathcal{A}' \parallel w'_1, w'_2) \parallel w_2 \) if: (1) \( \mathcal{A} = \mathcal{A}' \), (2) \( w_1 =_p w'_1 \), (3) \( w_2 =_p w'_2 \), (4) \( w_1 \downarrow w_2 \), and (5) \( e \vdash w_2 \) and \( \Delta \vdash w_2 \) for some \( \Delta \).

**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.

\[ \square \]

### 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 \(\langle \text{cor}(w), \rightarrow_i \rangle \leq \langle \text{source}(w), \rightarrow_i \rangle\).

We break the result into simpler simulations. First, we recover statements removed by cor(·).

**Lemma 3.6.33.** If \(\epsilon \vdash w\), then \(\langle \text{cor}(w), \rightarrow_i \rangle \leq \langle w, \rightarrow_i \rangle\).

**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 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 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 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 \( \text{cor}(w) \) and \( w \) agree on *trusted* values (formalized using \( =_T \)).

Next, we remove host annotations and explicit communication.

**Lemma 3.6.34.** If \( \in \vdash w \), then \( \langle w, \rightarrow_i \rangle \leq \langle \text{source}(w), \rightarrow_i \rangle \).

**Proof.** Statements removed by \( \text{source}() \) 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 \( \text{declassify}/\text{endorse} \) 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 \( \text{source}(w) \) only if the statement is preserved by \( \text{source}() \); it does nothing otherwise. To handle \( \text{declassify} \), whenever the simulator receives a message of the form \( \ast \text{Adv}v \), the simulator inspects its copy of \( w \) to determine the sending host \( h \), and sends \( h\text{Adv}v \) to the adversary instead. Similarly, to handle \( \text{endorse} \), the simulator replaces \( h \) with \( \ast \) in messages \( \text{Adv}hv \) 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 $\left\llbracket s \right\rrbracket$. We construct a corresponding SUC protocol $\left\llbracket s \right\rrbracket^{SUC}$ 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 $\left\llbracket s \right\rrbracket$, the party/functionality for $h$ in $\left\llbracket s \right\rrbracket^{SUC}$ is essentially a wrapper around the projected host $\left\llbracket s \right\rrbracket_h$, who steps $\left\llbracket s \right\rrbracket_h$ 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 ($\mathcal{L}(h) \in \mathcal{P} \cap \mathcal{T}$), 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 \( \mathcal{R}_{sec} \), which guarantees in-order message delivery and enables secret communication between honest hosts. We can realize \( \mathcal{R}_{sec} \) in SUC via a standard subprotocol using a public key infrastructure.

For ideal functionalities in \( \llbracket s \rrbracket_{SUC} \), 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 \( \Delta \models s \) 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 \( \text{MPC}(\text{Alice, Bob}) \) indicates that \( \text{MPC}(\text{Alice, Bob}) \) 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 \text{MPC}(\text{Alice, Bob}) can be instead achieved using Alice and Bob alone. Similar security concerns for label-based host selection have been discussed for Viaduct \cite{2}. We can formalize this intuition via a simulation of the form \langle W, \rightarrow_i \rangle \leq \langle W, \rightarrow'_i \rangle, \text{ where } W \text{ uses } \text{MPC}(\text{Alice, Bob}), \text{ and } \rightarrow'_i \text{ is modified so that corruption of } \text{MPC}(\text{Alice, Bob}) \text{ is impossible.}

### 3.8 Related Work

#### Secure Program Partitioning

Prior work on secure program partitioning \cite{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 \cite{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 \cite{43,42,5,62,60,59} focuses on enforcing computational non-interference 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. \cite{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 \cite{73, 72, 32, 33, 53}, but the primary concern in this literature is proving deadlock freedom, and very little prior work considers security \cite{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.


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


[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. 


