Composable Enhancements for Gradual Assurances

A DISSERTATION PRESENTED
BY
LUCAS REED WAYE
TO
THE HARVARD JOHN A. PAULSON SCHOOL OF ENGINEERING AND APPLIED SCIENCES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS

NOVEMBER 2017

Dissertation Advisor: Professor Stephen Chong Lucas Reed Waye

Composable Enhancements for Gradual Assurances

Abstract

This dissertation presents three enhancements to software components that increase the trustworthiness and usability of the systems they comprise. The enhancements are *composable*: they are local to the components they enhance and are self-contained. Since they are self-contained and local, composable enhancements can be partially deployed in a system without adversely affecting system behavior. The assurances provided by the enhancements are *gradual*: as the number of enhanced components in the system increases, the overall assurances provided by the entire system also increases. This dissertation investigates these enhancements in three different settings. In each setting, it identifies an inherent security or functional correctness problem with the components in that setting and provides a composable enhancement that remedies the problem gradually.

In the first setting, this dissertation identifies that although modern services expose interfaces that are higher-order in spirit, the simplicity of the network protocols used forces services to rely on brittle encodings. To bridge the semantic gap, this dissertation presents Whip, a higher-order contract system that allows programmers to detect when services do not live up to their advertised higher-order interfaces. As more services use Whip, it becomes easier to identify errant services in the system.

In the second setting, this dissertation identifies that in the Disjunction Category label model, capability-like privileges are used to downgrade information, which when used inappropriately can

compromise security. To ensure privileges are used as intended, this dissertation presents restricted privileges, an enhancement to privileges to control downgrading based on a specification of security conditions for when they can be legitimately used. As more privileges are restricted, information in the system becomes more protected from the accidental or malicious exercise of privileges to downgrade more information than intended.

In the third setting, this dissertation identifies that information-flow control (IFC) programs often need to interact with a key-value store that can also be accessed by non-IFC programs. These non-IFC programs may inadvertently (or maliciously) fail to respect the policies enforced by the IFC programs. In order to ensure the information protected by the IFC programs is not exfiltrated or corrupted by non-IFC programs through the key-value store, this dissertation presents Clio, an extension to a popular IFC language that transparently incorporates cryptography for data on the untrustworthy key-value store. As more IFC programs use Clio, the more information is protected from non-IFC programs in the system.

Contents

Listing of figures

To my late mother Barbara Joan Waye
for getting me my first computer, instead of the guitar.

Acknowledgments

The research process broadly seeks to produce new knowledge or deepen the understanding of a topic. At its core, it is neither an art nor a science but a non-empirical craft. From its medieval roots, this craft has been taught through an apprenticeship. I have been fortunate enough to have a master teach me the craft, my advisor Stephen Chong. Under his guidance I have been able to hone my research skills as his apprentice through relentless supervised practice. For teaching me the craft, and in particular the knack of storytelling, I will be forever grateful. Thank you!

The process is not solitary; it is entirely a social endeavor. My committee members Eddie Kohler and Jim Waldo provided expert feedback on my research and my interactions with them proved to be invaluable for my research.

Many of these ideas in this dissertation would not have come to fruition without my collaborators. Many thanks to Owen Arden, Pablo Buiras, Christos Dimoulas, Dan King, Alejandro Russo, Deian Stefan, and Marco Vassena for putting up with me for the betterment of the work.

The members of the Harvard Programming Languages group were great to bounce research ideas off of and also to talk about anything (and everything). The conversations were always stimulating and made my time at Harvard much more enjoyable. Thanks Aaron Bembenek, David Darais, Anitha Gollamudi, Dan Huang, Andrew Johnson, Gregory Malecha, Scott Moore, and Adam Petcher. And thanks to my friends *not* in the lab for also being there for me, particularly Eric Buehl, Gautam Kamath, Weilin Meng, Shane Moriah, Jake Rowley, Tim Sidle, and Peter Smillie.

Thanks to the members of 410 Dryden Rd. for always staying in touch: Maxim Belomestnykh, Roman Goloborodko, Li Guo, Andy Hirschl, Shane McMorrow, Ryan Musa, and Sasha Naydich. May your cups be forever filled with foam.

Last, but certainly not least, my family has given me the strength and courage to persevere. I cannot thank my parents enough for all that they have done, so instead I will attempt to live my life by the honorable example they have set for me.

Most of all, thanks to the newest part of my family, Katherine Terracciano. Your patience, continued support, and love has made this all possible. We may have met at the same time I started graduate school, but unlike graduate school our time together will have no end date.

1 Introduction

Software development is an increasingly distributed and collaborative effort. Programmers write components that implement basic functionality and compose them to form large and complicated software systems. Due to the near-ubiquitous access to the internet and the benefits it provides, in many cases the mechanism of composition is a network protocol that facilitates inter-component communication. As a result, the computations and data storage of these software systems are distributed to geographically remote and untrusted locations.

At the same time, society has increasingly expected more from these online software systems as they have become more integrated into our world. We rely on them to communicate, perform financial transactions, learn, and manage our lives. From health care support systems to social networking web sites, the information handled and produced by these systems have intricate dependencies. Further, the data used by these systems often contains sensitive personal, corporate, and government information.

The expectations on the information used and produced by the systems are encoded in detailed and complicated requirements, often derived from legislation, industry standards, and organizational guidelines. These requirements can take the form of information security requirements or functional correctness requirements. Abstractly, the goal of information security is to ensure that all computations on information obey a given *security policy* while functional correctness asserts that the functional behavior of all computations satisfy a given *functional correctness property*. Security policies address the confidentiality, integrity, and availability of data a system uses and produces.

That is, the data is imbued with additional meaning; for example, the data also declares (either explicitly or implicitly) who may read it. Functional correctness properties address the structure (i.e., its data type) and decidable properties on the input and output data of the system.

Some of these properties can be automatically checked in a closed system. For example, security policies can be automatically enforced by Information-Flow Control (IFC) systems. But even with precise specifications for these systems, current methodologies for building them provide little assurance that the requirements are satisfied in the presence of untrusted or unvalidated components in the system. Due to the highly entangled composition of components, insecurity or functional incorrectness in a single component can lead to system-wide security and functional correctness issues. The reason for this is that enforcement of these requirements at component boundaries is not clearly connected to the overall system requirements. Since unvalidated or untrusted components do not ensure that they interact with the rest of the system in a secure or functionally correct way, the overall properties of the system become unclear. As a result, reasoning that a system correctly enforces a security or functional correctness requirement may require reasoning about all of the components in the system. In sum, the overall security and functional correctness of the system is in part influenced by the behavior of unvalidated or untrustworthy components in the system.

This work demonstrates how more precise specifications at component boundaries, together with their enforcement, can help to build more trustworthy and usable systems. Further, this work provides a clear migration path towards more usable and trustworthy systems by ensuring that the enforcement of these requirements is self-contained. We achieve this by creating enforcement mechanisms that operate on individual components instead of on entire systems. These per-component enforcement mechanisms serve as the *enhancements* to the components. Since components can be independently deployed in large online software systems, a full-scale deployment of the enhancements is often infeasible in practice. Thus, in order to be pragmatic and usable, the enforcement mechanisms should function under partial deployment.

Contributions and Outline This dissertation presents three enhancements to software components that provide gradual information security and functional correctness guarantees. The first enhancement is a higher-order contract system that allows programmers to detect services that do not live up to their advertised higher-order interfaces, presented in Chapter 2. In particular, the work provides a runtime monitor that is composable and practical, as well as a contract language that can capture many common behavioral properties unique to modern services.

Chapter 3 is not a contribution but instead provides background for Chapters 4 and 5. It describes LIO [77], an existing Information Flow Control Haskell library and Disjunction Category

(DC) labels [76], the label format used by LIO that specifies the information security policies to enforce on the data.

Chapter 4 presents the second enhancement: a restriction to privileges used by DC labels to control their accidental or malicious misuse to downgrade more information than intended. There are two kinds of restricted privileges: *bounded privileges*, which impose upper and lower bounds on the DC labels of data that is declassified or endorsed using that privilege, and *robust privileges* which provide a property known as robustness [60, 88].

Finally, Chapter 5 presents the third enhancement: Clio, an extension to LIO that transparently incorporates cryptography for data on an untrustworthy key-value store. Clio protects information on the store that can be accessed by non-IFC programs. Clio is formalized with a computational model of how it uses cryptography. With that model, a novel proof technique is used that incorporates the guarantees provided by IFC and cryptosystems together to show an even stronger property that accounts for attackers more powerful than those typically considered by IFC or cryptosystems alone.

All three enhancements include prototype implementations with various empirical evaluations to show their usability and feasibility in a real system.

The material in Chapter 2 is joint work with Christos Dimoulas and Stephen Chong. The material in Chapter 4 is joint work with Dan King, Pablo Buiras, Stephen Chong, and Alejandro Russo. The material in Chapter 5 is joint work with Pablo Buiras, Owen Arden, Alejandro Russo, and Stephen Chong.

2

Higher-Order Behavioral Contracts for Modern Services

2.1 Introduction

The documentation of popular services is rife with descriptions of non-trivial properties. For instance, the documentation of the Thrift API of the popular note-taking service Evernote[1] states that the listLinkedNotebooks operation returns (among other data) a noteStoreURL, the endpoint of a service that implements the NoteStore interface. This is not a trivial property of the listLinkedNotebooks operation as it describes how another server, denoted by noteStoreURL, behaves. That is, the server at noteStoreURL implements the NoteStore interface. Indeed, this is a *higher-order* property.

Thrift and other simple serialization formats with a few simple types cannot capture such a property, let alone enforce it. As a result, the documentation describes it only informally. It is up to the developers to add code to check whether the property holds and to figure out the problem when the property does not hold.

Despite this complication, the reliance on lightweight protocols comes with benefits. In fact, Evernote's API is just an instance of a general trend in software development, which we refer to as modern service-orientation. Modern services, dubbed *microservices*, opt out of complex shared message protocols, and encourage the use of different implementation languages and the independent update and re-deployment of services. Earlier service-oriented architectures compose services us-

[1] https://dev.evernote.com/doc/

ing sophisticated interfaces via middleware, such as CORBA [61] or Enterprise Service Buses, but they impose complex message protocols on developers and large software shops have found that it quickly becomes a productivity bottleneck [29]. Modern service-orientation makes development and deployment faster and has been employed at software companies—including Netflix, Google, Amazon, and Twitter—to construct massive and widely-used products [29, 72]. The success of modern services is succinctly summarized with the slogan "smart endpoints and dumb pipes" [29].

However, as we hint at with the Evernote example, implementation errors and incorrect compositions of components are more likely, as the simple message protocols—the "dumb pipes"—make it easy to misuse a component's interface or fail to implement an interface correctly. For example, in Evernote's API, a noteStoreURL may be a syntactically valid URL but the endpoint denoted by it may implement a different interface than expected (e.g., possibly it provides some other Evernote service).

The inability of low-level specifications to express and check such properties leads developers to inject brittle defensive checks in their code. Misplaced or incorrect checks complicate the debugging of services. This is exacerbated when services pass unchecked (and possibly incorrect) data from messages they receive from other services. When a service eventually discovers a problem, the source of the invalid data may be multiple message hops away.

To address this problem, we present Whip, a software contract system that bridges the semantic gap between the low-level interfaces of modern services and the high-level application-specific properties services need.

Inspired by Design by Contract [52–54], Whip associates each service with a *contract*: a precise and enforceable specification of its expectations of and promises to other components. Whip contracts embed predicates written in a full-fledged programming language in a *domain-specific contract language* tailored to the needs of modern services. Whip checks these contracts when components run. Thus Whip contracts make it easy for programmers to state and enforce precise conditions on the correct use of a service, and they eliminate the need for defensive code. Moreover, Whip facilitates the debugging of modern service-oriented applications, including legacy services, by providing *correct blame assignment*: blame information pinpoints accurately the services whose code is the source of the bug (i.e., behavior that deviates from the contract).

Modern services are higher-order in nature and so should be the Whip contracts that describe their interfaces. To that end, Whip's contract language is *higher-order* and, for instance, can express that noteStoreURL refers to a service that adheres to the NoteStore contract. Even though contracts for higher-order functions [26] have been extensively studied over the last fifteen years, adapting these results for modern services is not straightforward. Modern services exchange data serialized as

streams of bits rather than closures or objects. Thus, the Whip contract language gives to programmers specialized linguistic tools to associate serialized data with a higher-order entity of a particular interface. For instance, a Whip contract can express that (i) some bits from a message encode a pointer to the code portion of a closure; (ii) some other bits from the message encode the closure's environment; and (iii) the closure returns a list of maximum n notes when given an appropriate authorization token and a non-negative integer n.

In addition to an expressive specialized contract language, to be useful in practice, Whip must meet a demanding set of requirements, derived from the high degree of autonomy and independence that service owners have in the implementation and deployment of their services [29]:

- Whip must operate under *partial deployment*, since there may be some service owners that choose not to use Whip.
- Whip must be *transparent*: it must make no changes to communication patterns between services, so that services unaware of Whip continue to operate.
- Whip must be *language agnostic*. Services in the same application may be implemented in many different programming languages. Indeed, the source code of services may not be available to modify or even read, since application programmers wire together (possibly third-party) remote services. Thus, in contrast to a contract system for a programming language, Whip cannot depend on the runtime of a component's language to enforce contracts.
- Whip should accommodate the *loose coupling of modern services* and be *extensible with message formats*. The simple network serialization formats used by modern services (e.g., Thrift, SOAP, JSON, and Google Protocol Buffers) enable loose coupling, but evolve over time and new ones are designed frequently.

Whip meets all of these requirements. Whip can be deployed on a service-by-service basis and is backwards compatible with non-Whip services. Whip is language agnostic, taking a black-box approach to contract checking by inspecting only the messages that services send and receive. Whip is also designed to be modular with message formats, allowing it to be extended to support additional protocols and message formats. Whip already supports popular interface technologies such as Thrift, WSDL, and REST.

Due to the above requirements and the domain specific nature of its contract language, Whip is a complex and subtle system distinct from other contract systems. As any other contract system though, Whip aims to provide programmers with accurate information upon contract violations to

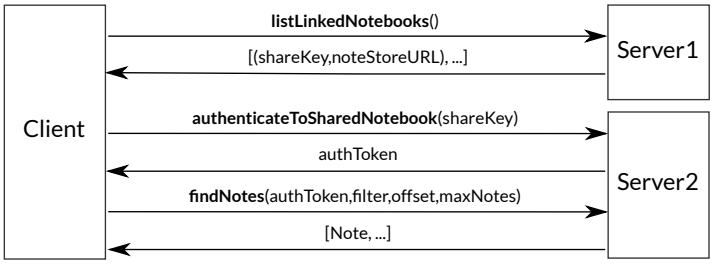

Figure 2.1: Evernote: access to a shared notebook

cut down the debugging space and speed up fixes. Indeed, Whip *provably assigns correct blame* [23]. To establish this key metatheoretical property of Whip and capture formally Whip's unique setting, we describe its behavior with a custom model, *WM* (Whip Model).

In the remainder of the chapter, we first describe Whip's contract language (Section 2.2). Then, we provide a complete but high-level description of the runtime of Whip (Section 2.3). Section 2.4 introduces *WM* to make the informal description of the previous two sections precise. In Section 2.5, we use *WM* to show that Whip assigns correct blame. We have implemented Whip and used it to harden the interfaces of a variety of off-the-shelf services (Section 2.6), and evaluated the performance impact of Whip (Section 2.7).

2.2 The Whip Contract Language

We present Whip's contract language by demonstrating how it can specify precise properties of the Evernote API we discussed briefly in the previous section.

Evernote provides cloud-based storage of notes, organized into notebooks. Each user's notes and notebooks are stored in a distributed database called a *note store*. Client-side services implement tools for users to manage their notes and notebooks. Moreover, Evernote allows users to share notebooks. Thus, a note store contains the notebooks a user has created, a list of *shared notebooks* she has shared with other users, and a list of *linked notebooks* that other users have shared with her.

Accessing a shared notebook may require contacting a different note store than the one the user contacts for her own notes. Figure 2.1 depicts some of the steps that a client must take to access a note from a notebook shared with the client's user by another user.[1] Each box represents a service: Client is the client-side service that interacts with Evernote services; Server1 is the Evernote service that implements the user's note store; and Server2 is the note store where a shared notebook resides.

[1] We present a simplified version of the API for clarity. Irrelevant arguments and return values of the operations we consider are elided.

Arrows indicate requests (left to right, annotated with operation and arguments) and replies (right to left, annotated with returned values).

To access a shared notebook, the client retrieves a list of linked notebooks from its note store (operation listLinkedNotebooks), and uses the information from the list to contact Server2 and authenticate to the shared notebook (authenticateToSharedNotebook). The client authenticates by presenting the particular shareKey for Server2. The client can then access the shared notebook, by, for example, calling the findNotes operation to search for particular notes, passing as one of the arguments the authToken returned by authenticateToSharedNotebook.

```
1  service NoteStore {
2    NoteList findNotes(1: string authToken,
3                       2: NoteFilter filter,
4                       3: i32 offset,
5                       4: i32 maxNotes)
6     throws (1: Errors.EDAMUserException userException),
7    ...
8
9    list<Types.LinkedNotebook> listLinkedNotebooks()
10    throws (1: Errors.EDAMUserException userException),
11   ...
12
13   AuthenticationResult authenticateToSharedNotebook(
14                       1: string shareKey)
15    throws (1: Errors.EDAMUserException userException),
16   ...
17 }
```

Figure 2.2: The NoteStore Thrift API

Figure 2.2 displays the portion of Evernote's Thrift API that corresponds to operations that play a part in this work flow. For each operation the Thrift API describes the data types of arguments and results together with the data types of any exceptions the operation may throw.

We focus on two first-order and two higher-order properties in this work flow that are necessary to access notes from a user's shared notebooks but are beyond the capabilities of Thrift's interface description language and are stated only informally in the documentation.

First-order Properties In contrast to its Thrift specification, the findNotes operation does not accept any 32-bit integer as its offset argument. The offset is the smallest numeric index of the notes included in the result of the operation. Thus, the documentation of the API explains, an offset has to be a non-negative integer. This is a *first-order function* property that a contract system for a programming language can capture with a pre-condition predicate.

The second property is a *dependent first-order function* property; the findNotes operation returns a list with length at most maxNotes (another argument to the function). Thus, it corresponds to a post-condition that states a property of the result of a function call in relation to the arguments of the call.

```
1  service NoteStore {
2    findNotes(authToken,filter,offset,maxNotes)
3    @requires « offset >= 0 »
4    @ensures  « length(result) <=  maxNotes »
5    ...
6
7    listLinkedNotebooks()
8    @foreach (noteStoreUrl, shareKey) in « result »
9      identifies NoteStore at « noteStoreUrl »
10     with index « shareKey »
11   ...
12
13   authenticateToSharedNotebook(shareKey)
14   @where index is « shareKey »
15   @ensures « type(result) != EDAMUserException »
16   ...
17 }
```

Figure 2.3: The NoteStore contract

Higher-order Properties The two higher-order properties of the Evernote API are the ones we mention in Section 2.1. First, the operation listLinkedNotebooks returns a list of pairs (noteStoreURL, shareKey) where noteStoreURL refers to a service that implements the interface of a note store. In terms of a programming language, this property can be expressed with a higher-order function contract that ascribes a contract for the services pointed to by the result of listLinkedNotebooks.

Second, the shareKey that is bundled up with each noteStoreURL in the result of listLinkedNotebooks has to be used as an argument for a successful call to authenticateToSharedNotebook on that service. This is a common pattern in the world of microservices due to the lack of proper abstractions such as closures and objects. Since programmers cannot properly encapsulate the environment of a piece of code, they manually follow call protocols and explicitly pass around relevant pieces of the environment of a service's operation when invoking the operation.

Whip's contract language can capture all these properties. The Whip contract language does not focus on syntactic specifications such as the data types of arguments and results of service operations (which, as Figure 2.2 demonstrates, interface description languages such as Thrift's already handle). Its features are tailored to the service contracts that Whip aims to express and enforce.

Figure 2.3 shows part of the service contract for a note store service expressed in Whip's Interface

Description Language (IDL). The keyword **service** defines a service contract that describes the interface of a service, and gives a name to the contract, in this case NoteStore. For each operation the service provides, the service contract contains an *operation contract*, that is a signature for the operation followed by tags that state properties about the operation's arguments and result. For example, the NoteStore service contract includes operation contracts for findNotes, listLinkedNotebooks, and authenticateToSharedNotebook.

First-order Operation Contracts The operation contract for findNotes (lines 2–4) expresses the two first-order properties from above: offset is a non-negative integer [1] and the length of the returned list is at most maxNotes. The tags **@requires** and **@ensures** define, respectively, a pre-condition and a post-condition, specified as Python code, i.e., code between « and » in the contract is Python. Code in pre- and post-conditions can refer to the operation's arguments (e.g., offset and maxNotes in the snippet above). Post-conditions also have access to a special variable **result** that is bound to the result of an operation call. In the snippet, **@requires** checks that offset is non-negative and **@ensures** specifies that the length of the **result** list is less than or equal to maxNotes.

Higher-order Operation Contracts Recall that the two higher-order properties of interest state that listLinkedNotebooks returns pairs (noteStoreURL, shareKey) where (i) noteStoreURL refers to a note store, and (ii) shareKey can be used to successfully call authenticateToSharedNotebook on that note store. Consider a single such pair (u, k). To express the properties, the Whip NoteStore contract must be able to express not only that u refers to a NoteStore service (say, Server2), but also that the operation authenticateToSharedNotebook on Server2 expects k as its argument.

To capture that u implements the NoteStore contract where k can be used to successfully call authenticateToSharedNotebook, we introduce *indexed service contracts*, a new kind of contract that handles this idiom of modern services. We treat NoteStore as a *family of service contracts*, indexed by a value.[2] Thus, the *indexed service contract* NoteStore$\langle k \rangle$ is an appropriate service contract for the service that u refers to.[3] The same service may, of course, satisfy other indexed service contracts from the same family, such as NoteStore$\langle k' \rangle$, where k' is a different shareKey.

Returning to the IDL, lines 9–10 present an example of a higher-order operation contract. The

[1] The Thrift API of Evernote already states this argument is a 32-bit integer. Thus, the Whip contract does not repeat this syntactic requirement.

[2] Indeed, every Whip contract is implicitly a family of contracts; if no index value is explicitly provided, a special default index value is used.

[3] We use "service contract" to refer to both service contract families and indexed service contracts when this is clear from the context.

operation contract specifies that the result of listLinkedNotebooks identifies multiple NoteStore services: for each pair in the returned list, noteStoreURL refers to a service that implements service contract NoteStore⟨shareKey⟩. Line 14 presents a *use* of the higher-order operation contract. This line indicates that authenticateToSharedNotebook is an operation of a service that implements NoteStore⟨shareKey⟩, where shareKey is the operation's argument.

2.3 The Whip Runtime, Informally

In this section we informally describe how the Whip contracts of Section 2.2 are enforced by a *Whip-enhanced service*. Section 2.3.1 describes the high-level design and deployment of an enhanced service. Section 2.3.2 explains how the enhanced service uses its internal state to enforce Whip contracts, and Section 2.3.3 describes the enhanced service's behavior in the event of a contract failure. Finally, Section 2.3.4 describes how Whip leverages a special enhanced protocol when both services involved in an operation are enhanced.

Although Whip targets distributed applications, we emphasize that we focus on functional correctness of service composition via higher-order contracts, and not on reliability or fault tolerance of distributed systems. Indeed, modern services are often chosen for organizational concerns such as loose coupling and scalability, rather than for reliability. Existing techniques to enhance the reliability of distributed systems are compatible with and orthogonal to Whip. That said, Whip assumes a communication layer (i.e., TCP), that can authenticate endpoints and does not corrupt messages. Whip does not rely on the order of message delivery in order, nor that message delivery is guaranteed. Although TCP provides both those guarantees, higher-level messaging layers do not (since, for example, they may close and re-open TCP connections).

2.3.1 The Whip Adapter

A *Whip-enhanced service* is a service deployed with an adapter. Figure 2.4 depicts a deployment of a Whip-enhanced service. The Whip network adapter intercepts all messages between the service and its peers. The network adapter uses its internal state to check messages against their corresponding contract. The adapter runs in its own OS process and intercepts raw TCP data to and from the service and is responsible for checking contracts. Whip treats all services as black boxes and does not require code modifications nor does it change a black-box service's view of interaction with other

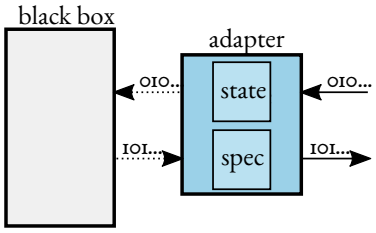

Figure 2.4: Diagram of a Whip adapter

services. If a contract check fails, Whip logs the details of the contract and message involved, including *blame labels* that are unique identifying names of adapters and help localize the fault at hand. An adapter's blame label should uniquely identify the deployment of the adapter's enhanced service. In some settings, this may be as simple as the hostname of the service.

Each adapter has local state that contains sufficient information to determine which messages to intercept and which contracts to enforce on these messages. The local state consists of a mutable *blame registry* and *confirmation mapping*.

- The blame registry tracks contract information about services and requests and who is responsible for this contract information. It maps *service entries* and *request entries* to *blame information*.
  - Service entries track contract information about services: a service entry is a pair of an endpoint and an indexed contract.
  - Request entries track contract information for requests made by and received by the Whip-enhanced service: a request entry is a pair of a unique request identifier and the service entry to which the request was made.
  - Blame information consists of blame labels that identify a set of adapters as the sources of the assumption that the relevant service should satisfy the indexed contract. When a contract violation occurs, blame information is used to generate log messages to help determine why the contract failed. The fine granularity of service entries and request entries allows Whip to precisely track blame information, and produce useful log messages.
- The confirmation mapping tracks whether a service has in fact agreed to implement a contract family k, or if this is merely asserted by a third party. The confirmation mapping maps service entries and request entries to their *confirmation status*. If a service entry or request entry is *confirmed*, then the adapter can correctly assume that the endpoint for the appropriate service has committed to the associated indexed contract.

The local state of each adapter is initialized with information about well-known service endpoints that the Whip-enhanced service might communicate with, and what contracts those communications should be held to. Any requests to those configured endpoints are intercepted and checked against the specified contract.

The local state is updated as the adapter processes messages it intercepts and learns about new services, observes requests, and finds out that services are confirmed. An adapter uses its state to lookup the contract family it believes a host adheres to and to assign blame in the event of a contract failure.

Configuring Whip We assume that all adapters have access to the same fixed Whip contract specification (as described in Section 2.2). That is, in our current version of Whip, adapters can learn about new services, but not about new contract families. This is not a fundamental restriction.

In addition, adapters are given a configuration file that describes, for each contract family, the low-level message protocol used and the syntactic representation of messages for this contract. This enables the adapter to parse the raw bytes comprising a message and extract values needed for contract checking and identifying new service entries.

The separation of the syntactic interface of a service and its contract allows Whip to handle multiple message protocols and RPC frameworks within a single distributed application, and even within a single Whip-enhanced service. This is a necessity due to the loosely-coupled and heterogeneous nature of modern services. Currently, Whip supports SOAP (defined by a WSDL), REST, and Thrift messages and support for more protocols can be added without modifications to the design of Whip or its contract language.

Although the design and implementation of this configuration and parsing information is one of the significant engineering challenges we encountered—and essential to developing a useful and practical tool—for the rest of the chapter we focus primarily on how we track contract and blame information and enforce contracts.

2.3.2 Enforcing Contracts

In this section we describe how a Whip adapter uses its local state to check the contracts described in Section 2.2.

When a service makes a request, the adapter intercepts the message if the destination endpoint matches a service entry's endpoint in the adapter's blame registry. Alternatively, if the service is making a reply, the adapter intercepts the message if the destination endpoint matches a request entry's endpoint in the adapter's blame registry. If a matching entry is found, then it is checked according to the contract family given in the blame registry for the entry. We discuss the behavior of each kind of contract check the adapter performs and then describe the behavior of the adapter when no matching entry is found.

First-order Contracts

The most basic Whip contracts consist of pre- and post-conditions that check first-order properties of application message data.

After the message is parsed according to the configured message protocol, the pre-condition check is made for request messages and the post-condition for reply messages. In the event of a contract failure for a pre-condition, the sending service is always blamed as it is the initiator of the request.

Post-condition violations occur when a service sends a reply to another service. Whip logs a contract error with the blame labels from the blame registry for the request entry. We describe in Section 2.3.3 which blame labels are used for a request entry in an adapter's blame registry, but intuitively, the blame labels are the sources of the assumption that the endpoint satisfies the associated indexed contract.

Higher-order Contracts

The second form of contract is a higher-order contract. From Section 2.2, these contracts contain an **identifies** tag.

For a message that invokes an operation that includes an **identifies** tag in its contract, the message contents may identify (zero or more) endpoints that should satisfy indexed contracts. For example, in our Evernote example, the reply message for the listLinkedNotebooks operation identified noteStoreURL as satisfying the indexed contract NoteStore⟨shareKey⟩.

The Whip adapter updates its blame registry to record the newly identified endpoint (and the indexed contract associated with it) so that future messages to and from the endpoint are intercepted and appropriate contract checks performed. That is, returning to our Evernote example, the adapter updates its blame registry to include a service entry for the endpoint at noteStoreURL that satisfies the indexed contract NoteStore⟨shareKey⟩.

Bypass Checks

If no matching service entry or request entry is found in the blame registry for an intercepted message, the adapter *bypasses* contract checks for the message. There are two reasons that there may be no matching entry in the blame registry: if no contract information is available, or there is *conflicting* contract information for the endpoint. There may be no contract information in the cases when we are interested in checking contracts for only some of the network communication performed by a black box. For example, we may choose to ignore web browser requests by the black box. The adapter may also bypass the checks if the local blame registry contains a *conflict* for an endpoint. A

conflict occurs when there is no confirmed service entry for the endpoint and multiple service entries that disagree on the contract family for the endpoint. We require that an endpoint implements at most one contract family, so at least one of the unconfirmed service entries is incorrect (but we do not necessarily know which one). (If we have a confirmed service entry, then the service definitely implements the specified contract family, so any unconfirmed service entries that disagree can be ignored.)

2.3.3 Tracking Blame

Central to Whip's contract checking mechanism is the blame information it provides upon a post-condition violation. The key intuitive idea is that in the event of a contract violation by a given endpoint, a Whip adapter should blame the service(s) that (from the adapter's perspective) are responsible for the initial association between the endpoint and the contract.

We describe how Whip adapters update their blame registry to record and track blame information in order to achieve this goal. We first introduce some terminology to help describe different ways an adapter updates its blame registry. Given an adapter $\mathcal{A}$, a service contract c, and a message m, *m identifies a service entry* if checking the relevant part of c against m results in the association of an endpoint with an indexed contract via the **identifies** tag of c. If the identified service entries are not already in the blame registry of $\mathcal{A}$, $\mathcal{A}$ extends its blame registry accordingly. Given an adapter $\mathcal{A}$ and a message m, *m invokes a service entry* if it is a request message to the endpoint of the service entry, and due to processing m, $\mathcal{A}$ checks m against the relevant part of the contract that the service entry associates the endpoint with. Note that the relevant service entry may not be in the blame registry of $\mathcal{A}$ before the invocation. In fact the invocation may cause $\mathcal{A}$ to add the service entry to its blame registry. For instance, in our Evernote example consider that the Client is enhanced and its adapter's blame registry maps Server2 to the indexed service contract NoteStore<k>. If the Client authenticates to Server2 using k′ (i.e., a shareKey other than k) then the invocation results in a new service entry in the blame registry of the Client's adapter that associates Server2 with NoteStore<k′>.

Service entries are created and updated in an adapter's blame registry only when the adapter processes messages that identify or invoke service entries. The adapter aims to use the most precise blame information when creating new entries in its blame registry. However this is not always possible. For example, when an enhanced service adds a service entry due to processing a request from a (seemingly) unenhanced service, the blame label of the unenhanced sender of the message is unknown. In these situations the special blame label "†" is used to represent unknown blame information. All other blame labels uniquely identify Whip adapters; blame label † can be thought of as

representing all services that an adapter believes are non-Whip-enhanced. Request entries are created when an adapter processes a request message (either as a sender or receiver) and their blame information is determined by the corresponding invoked service entry. Once created, request entries are not modified.[1] We describe further the various ways adapters extend their blame registries with service and request entries focusing on blame information.

- When an enhanced service sends a request message to an endpoint:
  - If the message invokes a service entry, the adapter of the enhanced service may create a new service entry in its blame registry as discussed above. The blame information for such a new service entry is the adapter's own label. Intuitively, this is because the black box of the enhanced service initiated the request, invoking a service entry not previously seen, and so it is solely responsible for the claim that the endpoint should be associated with the corresponding indexed contract.
  - If the message identifies a service entry that is not in the blame registry of the enhanced service's adapter, a new service entry is created. The blame information for such a service is the blame label of the enhanced service itself.
  - If the message identifies a service entry that is in the blame registry of the enhanced service's adapter, the blame information described above is merged (set union) with the existing blame information for the service entry.
  - If the message invokes a service entry, and the service entry is not confirmed, a new request entry is created, and its blame information is the same as the blame information for the invoked service entry. The blame labels for the request entry are used to assign blame in the event of a post-condition contract violation (i.e., the corresponding reply violates its contract). For this reason, the request entry inherits its blame information from the invoked service entry. Put differently, the blame labels are those of the enhanced service(s) that asserted that the endpoint should satisfy the associated indexed contract. In many cases, the enhanced service that made the assertion is in fact the service itself (e.g., through its initial configuration). If the invoked service entry is confirmed then no request entry is created as the adapter does not check a post-condition on the reply message for the request (see Section 2.3.4, below).
- When an enhanced service receives a request message:

[1] In the implementation, request entries exist only for the duration of the request; in the formal model of Section 2.4, request entries are never removed from the blame registry.

  - If the message invokes a service entry, the adapter of the enhanced service may create a new service entry in its blame registry. If the message originates from another enhanced service's Whip adapter and contains enhanced information (see Section 2.3.4, below), the blame information for the new service entry is the same as in the sender's blame registry (since the sender's adapter has sent this blame information). Otherwise, since the sender of the message is unknown, the blame information is the special blame label "†".
  - If the message identifies a service entry that is not in the blame registry of the enhanced service's adapter, a new service entry is created. If the message originates from another enhanced service's Whip adapter and contains enhanced information, the blame information for the new service entry is the same as in the sender's blame registry (since the sender's adapter has sent this blame information). Otherwise, the blame information is the special blame label †.
  - If the message identifies a service entry that is in the blame registry of the enhanced service's adapter, the blame information described above is merged (set union) with the existing blame information for the service entry.
  - If the message invokes a service entry, a new request entry is created, and its blame information is the same as the blame information for the invoked service entry.

- When an enhanced service sends or receives a reply message, if the message identifies a new service entry, the service's adapter adds a new service entry to its blame registry whith the same blame information as the blame information of the corresponding request entry.

Back to our example from the beginning of this sub-section, due to Client's request to Server2, Client's adapter adds to its blame registry a service entry whose blame information is the blame label for Client. Since this service entry is unconfirmed, Client's adapter also adds to its blame registry a request entry with the label of Client as its blame information, the same blame information as the newly added service entry. Thus if the Client's adapter discovers that the reply from Server2 violates the corresponding post-condition from the noteStore contract, the adapter logs a contract violation blaming the Client.

Blame for higher-order contracts Readers may find it surprising that the blame information for an identified service entry introduced by a reply message is the same as the client's blame information for the invoked service entry. However, this is in keeping with the philosophy of higher-

order contracts in functional programming languages [23, 26]. This is because a service entry identified in a reply is analogous to a function f returning a closure g; in higher-order contracts for functional programming, the contract to enforce on g is derived from the contract for f, and so the blame labels for g are the same as the blame labels for f.

To make this design decision more concrete, consider a variant of the Evernote example from above. Suppose that Client sends a request to Server1 and from the reply identifies that, according to Server1's contract, Server2 should adhere to the contract NoteStore⟨k⟩. Moreover, assume that Client sends a request to Server2 that invokes the service entry that associates Server2 with NoteStore⟨k⟩, and Client detects that the reply violates the relevant post condition of the contract. Who should be blamed for the violation? Naively we may say Server2. However, from the perspective of Client, Server2 never agreed to adhere to contract NoteStore⟨k⟩. It is Server1 that made an error in asserting that Server2 implements NoteStore⟨k⟩. That is, Server1 did not live up to its contract since listLinkedNotebooks returns a service, Server2, that for whatever reason does not meet NoteStore⟨k⟩. Making sure that Server1 returns a service that meets NoteStore⟨k⟩ is the right fix to the problem at hand. After all, Server2's behavior may be what its other clients expect. The fact that this behavior triggered a contract violation is only because our Client relied on Server1 to live up to its contract. Thus, transitively, Whip blames whoever Client believes is responsible for the assumption that Server1 implements an appropriate contract for listLinkedNotebooks.

2.3.4 Leveraging Other Adapters

When the destination of a message is known to be Whip-enhanced, an adapter enriches the message with extra information to help the receiver adapter improve the accuracy of its blame information. This *enhanced interaction* differs from *unenhanced interaction* as enhanced interaction includes additional information understandable only by adapters, whereas unenhanced interaction is equivalent to the messages sent by the black boxes.

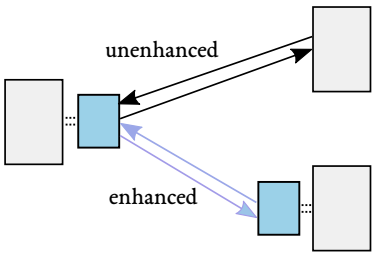

Figure 2.5: Whip-enhanced interaction

Enhanced messages are only sent between adapters, as we cannot assume that black boxes understand them. An enhanced message attaches to a black box message. Enhanced messages are translated back to their original black box messages before being passed to a black box.

Enhanced messages contain information from the sender's blame registry for relevant service

entries and request entries. This blame information is used as described in Section 2.3.3 above. Enhanced messages also contain confirmation status of service entries and request entries, to propagate knowledge of confirmed services (i.e., services known to be Whip-enhanced).

Whip favors enhanced interaction as this helps make blame information more precise. However, enhanced interaction can occur only when the sender knows (based on confirmation status) that the receiver is Whip-enhanced. Enhanced interaction may not be possible due to partial deployment (i.e., one of the black boxes does not have a Whip adapter) or because the two communicating Whip-enhanced services are not aware that the other is also Whip-enhanced (which may occur if the initial confirmation mapping in the adapter's local state did not confirm the other adapter, and previous enhanced messages in the system have not propagated confirmation of the other adapter). Figure 2.5 depicts enhanced interaction between two adapters (with blue arrows) together with unenhanced interaction with service that is not Whip-enhanced (with black arrows).

Additionally, when two Whip-enhanced services use enhanced interaction, they share the burden of contract checks; the sender of the request checks the pre-conditions, and the sender of the reply checks the post-conditions. Conversely, when adapters use unenhanced interaction, each adapter performs both pre- and post-condition checks, in case the other side is not enhanced.

2.4 Whip Formally

In this section, we gradually introduce *WM*. We first describe Core *WM*, a model of how modern services interact (Section 2.4.1). We extend Core *WM* to *WM*, a model for Whip (Section 2.4.2). We precisely describe the state of the adapter (Section 2.4.3), how it uses its state to intercept and check messages for contract violations (Section 2.4.4), and how it updates its state (Section 2.4.5). We formally show Whip assigns blame correctly in Section 2.5. *WM* demonstrates formally how Whip meets the requirements laid out in Section 2.1.

2.4.1 Core *WM*: Distributed Black Boxes

Figure 2.6 shows the syntax and reduction rules of Core *WM*, which captures communication between black-box services. Processes P and Q represent black-box services and the messages they exchange. There are two kinds of basic processes: black boxes $\boxed{a}$ and messages m. A compound process $P \parallel Q$ represents the parallel composition of processes P and Q. We assume standard structural equivalence.

A black box $\boxed{a}$ represents a service with host name a. To capture the lack of access to a service's source code, black boxes are opaque and we can observe only messages they send and receive. For

$$
\begin{array}{lrcl}
\textbf{Processes} & P, Q & ::= & \boxed{a} \mid m \texttt{ to } a \mid P \parallel Q \\
\textbf{Messages} & m & ::= & \texttt{req } \#n \texttt{ from } a : s \\
 & & \mid & \texttt{reply } \#n \texttt{ from } a : s \\
\textbf{Host names} & a, b & \in & \mathcal{A} \\
\textbf{Identifiers} & n & \in & \mathcal{N} \\
\textbf{Bit strings} & s & \in & \{0,1\}^* \\
\textbf{Contexts} & \mathcal{P} & ::= & [\cdot] \mid \mathcal{P} \parallel P
\end{array}
$$

Send-Request

$$
\frac{n \text{ fresh} \qquad a \neq b}{\mathcal{P}[\boxed{a}] \longrightarrow \mathcal{P}[\boxed{a} \parallel \texttt{req } \#n \texttt{ from } a : s \texttt{ to } b]}
$$

Send-Reply

$$
\frac{a \neq b}{\mathcal{P}[\boxed{a}] \longrightarrow \mathcal{P}[\boxed{a} \parallel \texttt{reply } \#n \texttt{ from } a : s \texttt{ to } b]}
$$

Receive-Message

$$
\frac{}{\mathcal{P}[\boxed{a} \parallel m \texttt{ to } a] \longrightarrow \mathcal{P}[\boxed{a}]}
$$

Figure 2.6: Core *WM* syntax and reduction rules

example, the client and two note stores from Section 2.2 are modeled as black boxes, which we refer to as `client`, `server1`, and `server2`.

Messages are of the form m `to` a where a denotes the name of the recipient of the message. In Core *WM* there are two types of messages: *requests* and *replies*. Request messages are of the form `req` $\#n$ `from` $a : s$ and reply messages are of the form `reply` $\#n$ `from` $a : s$, where a is the originator of the message, s is the payload of the message and n is the *request identifier*. A request identifier uniquely matches a request with its reply message.

In Core *WM*, processes evolve when black boxes consume and spawn messages. The reduction rules are of the form $P \rightarrow P'$. Rule Send-Request in Figure 2.6 shows that $\boxed{a}$ can produce a request message `req` $\#n$ `from` $a : s$ using a *fresh*[1] request identifier n. Rule Send-Reply shows that $\boxed{a}$ can spawn a reply message for any request identifier n. In practice, though, a meaningful request identifier in a reply message would come from a previous request it received. Finally, rule Receive-Message shows how a black box with host name a can consume request and reply mes-

[1] To ensure that request identifiers are locally unique, the black box draws in succession fresh identifiers from a local enumerable set. Locally unique request identifiers together with globally unique host names guarantee that each request is globally unique.

Processes	$P ::= \ldots \mid \texttt{mon}^{l}(\sigma, P^{a}) \mid \widehat{m}\ \texttt{to}\ a$
Base Processes	$P^{a} ::= \boxed{a} \mid P^{a} \parallel m\ \texttt{to}\ b$
Enhanced Messages	$\widehat{m} ::= m\,\texttt{with}\,\{\texttt{se-blame:=}\widetilde{l};\ \texttt{id-blame:=}\widetilde{l};\ \texttt{id-conf:=}c\}$
Log Entries	$le ::= \texttt{Pre}(se, l) \mid \texttt{Post}(se, \widetilde{l})$
Service Entries	$se ::= a\,\texttt{satisfies}\,k\langle \texttt{v} \rangle$
Request Entries	$re ::= \#n\ \texttt{from}\ a\,\texttt{expects}\,se$
Contract Names	$k \in \mathcal{K}$
Blame Labels	$l \in \mathcal{L}$
Contract Indices	$\texttt{v} \in \mathcal{V}$

Figure 2.7: *WM* syntax

sages.

Returning to the example from Section 2.2, we show the trace of the first request for listLinkedNotebooks between client and server1 (2.1), the client making the request (2.2), server1 consuming the request (2.3), server1 responding to the request (2.4), and client consuming the request (2.5):

$$\boxed{\texttt{client}} \parallel \boxed{\texttt{server1}} \rightarrow \tag{2.1}$$

$$\boxed{\texttt{client}} \parallel \texttt{req}\ \#n_1\ \texttt{from client : `list...' to server1} \parallel \boxed{\texttt{server1}} \rightarrow \tag{2.2}$$

$$\boxed{\texttt{client}} \parallel \boxed{\texttt{server1}} \rightarrow \tag{2.3}$$

$$\boxed{\texttt{client}} \parallel \boxed{\texttt{server1}} \parallel \texttt{reply}\ \#n_1\ \texttt{from server1 : `[share...' to client} \rightarrow \tag{2.4}$$

$$\boxed{\texttt{client}} \parallel \boxed{\texttt{server1}} \tag{2.5}$$

2.4.2 Adding Whip Adapters

Equipped with a model of modern service interaction, we extend *WM* with Whip adapters. Conceptually, an adapter wraps around a black-box service. In *WM* (and Whip), adapters are mutually-trusted. To reflect the independent deployability of modern services, we do not model adapters with access to a shared state and instead model it explicitly. Adapters piggyback on service messages to update each other's local state. Though it adds complexity, we formally model this explicit state transfer as it is necessary to efficiently and precisely blame contract violators.

Figure 2.7 presents the syntax of *WM*, extending the syntax of Core *WM*. An adapter process

$\texttt{mon}^l(\sigma, P^a)$ wraps a *base process* P^a, which consists of a black box, and a (possibly empty) list of Core *WM* messages that the black box has just sent and have yet to be processed by the adapter, or that the adapter has forwarded to the black box. Each adapter has a unique label l and local state σ that contains information for checking contracts and assigning blame. Back to the example from Section 2.2, we can model the interaction of a Whip-enhanced client with a Whip server1, and a non Whip-enhanced server2 as: $\texttt{mon}^{l_c}(\sigma_c, \boxed{\texttt{client}}) \parallel \texttt{mon}^{l_s}(\sigma_s, \boxed{\texttt{server1}}) \parallel \boxed{\texttt{server2}}$.

2.4.3 Adapter State

Section 2.3.1 presents informally the state of Whip adapters, which we map here to their formal counterparts in *WM*. Each adapter maintains local state σ, for which the syntax is given in Figure 2.8. Please ignore portions of the Figure shaded in gray; they will be discussed in Section 2.5.

State	σ	$::= (S, B, C, \widetilde{le}, V)$
Specification	S	$: k \mapsto (e, e)$
Predicates	e	$: m \mapsto \text{bool}$
Blame Registry	B	$: (se \mapsto \widetilde{l}) \cup (re \mapsto \widetilde{l})$
Confirmation Map	C	$: (se \mapsto c) \cup (re \mapsto c)$
Confirmation	c	$::= \checkmark \mid \text{✗}$
Provenance Map	V	$::= se \mapsto pe$
Provenance Entry	pe	$::= a$ **intro** $\mid se$ **intro** $\mid$ **learned**

Figure 2.8: *WM* store syntax

State σ is a tuple $(S, B, C, \widetilde{le})$, consisting of its specification S, blame registry B, confirmation map C and error log $\widetilde{le}$. For brevity, we write $\texttt{spec}_\sigma$, $\texttt{blame}_\sigma$, $\texttt{conf}_\sigma$, and $\texttt{errors}_\sigma$ to project each element, respectively, of state σ.

The first piece of state is the specification $\texttt{spec}_\sigma = S$ that maps contract names k to pre- and post-condition for a service's operation. For simplicity, we assume that every service offers one operation. Formally, S maps a contract name k to a pair (e_{pre}, e_{post}). We leave the syntax of predicates e unspecified, but require them to be total (i.e., terminating) single argument boolean functions. Specifications do not change during execution, nor are transferred between adapters. Moreover we assume specifications contain information about *all* contract names used by an adapter.

The second piece of state is the blame registry $\texttt{blame}_\sigma = B$, which maps service entries and request entries to blame information. A service entry $se = a\ \texttt{satisfies}\ k\langle \texttt{v} \rangle$ indicates that the service at host a should implement indexed contract $k\langle \texttt{v} \rangle$. We assume that a service host implements at most one contract family. A request entry $re = \#n\ \texttt{from}\ a\ \texttt{expects}\ se$ indicates that host a made a request with request identifier n to a service with service entry se. Entries in the blame registry of a

Whip-enhanced service a correspond to assumptions that a has about other services, and that clients of a have about a. Blame information $\widetilde{l}$ is the set of labels of adapters that introduced the relevant service. In Section 2.4.5, we describe formally how blame information is propagated to assign blame in the event of contract violations.

An adapter records the confirmation status of each service entry and request entry in its confirmation mapping $\texttt{conf}_\sigma = C$. If a service entry, a `satisfies` $k\langle \texttt{v} \rangle$, or a request entry, $\#n$ `from` b `expects` a `satisfies` $k\langle \texttt{v} \rangle$, is confirmed (i.e., confirmation status is ✓), then a is enhanced (i.e., is wrapped by an adapter), and the adapter state of a also associates the service it offers with the contract name k. A confirmation status ✗ means that the host a is not definitely known to be Whip-enhanced or that the adapter state of a does not associate the service it offers with the contract k.

Finally, an adapter's local state contains a set of log entries $\texttt{errors}_\sigma = \widetilde{le}$. Each log entry records the failure of a contract. A pre-condition log entry $\texttt{Pre}(a \texttt{ satisfies } k\langle \texttt{v} \rangle, l)$ indicates that the black box wrapped with the adapter with label l violated the pre-condition for indexed contract $k\langle \texttt{v} \rangle$ while making a request to a. A post-condition log entry $\texttt{Post}(a \texttt{ satisfies } k\langle \texttt{v} \rangle, \widetilde{l})$ indicates that black box a sent a reply for a request it received but the request failed to meet the post-condition of indexed contract $k\langle \texttt{v} \rangle$, and that the adapters with a label $l \in \widetilde{l}$ are to blame, i.e., they are responsible for the assumption that a would satisfy the indexed contract.

2.4.4 Adapter Message Interception and Contract Checking

Before delving into the semantics of *WM*, we first introduce formally *enhanced messages*, the messages adapters exchange in enhanced interaction (see Section 2.3.4). An enhanced message m `with` { `se-blame`:=$\widetilde{l}$; `id-conf`:=c; `id-blame`:=$\widetilde{l_{id}}$ } attaches to a black box message m the additional information that the receiver should hold $\widetilde{l}$ responsible if m does not live up to its contract. The enhanced message also contains blame information $\widetilde{l_{id}}$ and confirmation status c for the identified service in the message. For example, server1's reply to the client,

```
reply #n from server1 : `[shareKey...' to client
```

would identify `server2 satisfies NoteStore⟨shareKey⟩` and the confirmation status c in the enhanced message would indicate whether server2 is known to satisfy contract `NoteStore⟨shareKey⟩` and further that server2 is Whip-enhanced. To simplify the model, each message's payload identifies a single service.

The reduction semantics of *WM* include the reductions of Core *WM* (which allow black boxes

Enhanced Interaction

ENHANCED-SEND

$$\frac{(k, \checkmark) = \texttt{contract\_for}_\sigma(b, m) \quad \sigma', \widehat{m} = \texttt{lift}_\sigma(k, \checkmark, m, l)}{\mathcal{P}[\texttt{mon}^l(\sigma, P^a \parallel m \ \texttt{to}\ b)] \to \mathcal{P}[\texttt{mon}^l(\sigma', P^a) \parallel \widehat{m} \ \texttt{to}\ b]}$$

ENHANCED-RECEIVE

$$\frac{(k, \checkmark) = \texttt{contract\_for}_\sigma(a, \widehat{m}) \quad \sigma', m = \texttt{lower}_\sigma(k, \checkmark, \widehat{m})}{\mathcal{P}[\texttt{mon}^l(\sigma, P^a) \parallel \widehat{m} \ \texttt{to}\ a] \to \mathcal{P}[\texttt{mon}^l(\sigma', P^a \parallel m \ \texttt{to}\ a)]}$$

Unenhanced Interaction

UNENHANCED-SEND

$$\frac{(k, \text{✗}) = \texttt{contract\_for}_\sigma(b, m) \quad \sigma', \widehat{m} = \texttt{lift}_\sigma(k, \text{✗}, m, l) \quad \sigma'', m = \texttt{lower}_{\sigma'}(k, \text{✗}, \widehat{m})}{\mathcal{P}[\texttt{mon}^l(\sigma, P^a \parallel m \ \texttt{to}\ b)] \to \mathcal{P}[\texttt{mon}^l(\sigma'', P^a) \parallel m \ \texttt{to}\ b]}$$

UNENHANCED-RECEIVE

$$\frac{(k, c) = \texttt{contract\_for}_\sigma(a, m) \quad \sigma', \widehat{m} = \texttt{lift}_\sigma(k, \text{✗}, m, \dagger) \quad \sigma'', m = \texttt{lower}_{\sigma'}(k, \text{✗}, \widehat{m})}{\mathcal{P}[\texttt{mon}^l(\sigma, P^a) \parallel m \ \texttt{to}\ a] \to \mathcal{P}[\texttt{mon}^l(\sigma'', P^a \parallel m \ \texttt{to}\ a)]}$$

Bypass Adapter

BYPASS-SEND

$$\frac{\texttt{contract\_for}_\sigma(b, m) \text{ undefined}}{\mathcal{P}[\texttt{mon}^l(\sigma, P^a \parallel m \ \texttt{to}\ b)] \to \mathcal{P}[\texttt{mon}^l(\sigma, P^a) \parallel m \ \texttt{to}\ b]}$$

BYPASS-RECEIVE

$$\frac{\texttt{contract\_for}_\sigma(a, m) \text{ undefined}}{\mathcal{P}[\texttt{mon}^l(\sigma, P^a) \parallel m \ \texttt{to}\ a] \to \mathcal{P}[\texttt{mon}^l(\sigma, P^a \parallel m \ \texttt{to}\ a)]}$$

Figure 2.9: *WM* reduction rules

to consume and spawn messages). All messages sent to or from a wrapped black box are intercepted by the adapter, which processes the message (possibly performing contract checks and/or updating the adapter's internal state) and then forwards the message onwards (possibly transforming an unenhanced message to an enhanced message, or vice-versa).

There are six rules shown in Figure 2.9, in three groups: (1) Enhanced Interaction (cf. Section 2.3.4); (2) Unenhanced Interaction (cf. Section 2.3.4); and (3) Bypass Adapter (cf. Section 2.3.2). Within each group, there is one rule for when the wrapped black box is sending a message, and one rule for when the wrapped black box is receiving a message. The decision for which of the six rules to use is based on the internal state of the adapter, the destination of the message, and whether the message is enhanced or not. We discuss each of the three groups in turn.

Metafunction `contract_for`[1] provides convenient access to confirmation status, and is how the adapter chooses to use one of these three groups. Conceptually, `contract_for` searches the state for a service entry or request entry whose host matches the destination of the message and returns the contract family and confirmation status of that entry. It is possible that `contract_for` is undefined; we return to this last case in Section 2.4.4.

[1] Defined formally in Appendix A.1.

```
1:  function lift_σ(k, c, m, l)
2:    (se, re, se_id) = entries(m, k)
3:    update-lift(m, c, se, re, se_id, l)      ▷ State update
4:    (pre, post) = spec_σ(k)                  ▷ Fetch contract
5:    if type(m) == req then                   ▷ Request sent
6:      if ¬pre(m) then errors_σ += Pre(se, l)
7:    else                                     ▷ Reply sent
8:      l̄ = blame_σ(re)    ▷ Fetch recorded blame for re
9:      if ¬post(m) then errors_σ += Post(se, l̄)
10:   return m with {se-blame = blame_σ(se);
                     id-blame = blame_σ(se_id);
                     id-conf = conf_σ(se_id)}

11: function lower_σ(k, c, m̂)
12:   (se, re, se_id) = entries(raw(m̂), k)
13:   update-lower(m̂, c, se, re, se_id)        ▷ State update
14:   return raw(m̂)                            ▷ Return unenhanced

15: function entries(m, k)
16:   if type(m) == req then                   ▷ Is request message
17:     (a, b) = (to(m), from(m))              ▷ To a, from b
18:   else                                     ▷ Is reply message
19:     (a, b) = (from(m), to(m))              ▷ From a, to b
20:   return (a satisfies k⟨index(m)⟩,
            reqid(m) from b expects
                a satisfies k⟨index(m)⟩,
            identified(m))
```

Figure 2.10: `lift` and `lower` metafunctions.

Enhanced Interaction

The rules in this group apply when the recipient of the message is known to be Whip-enhanced, i.e., $\texttt{conf}_\sigma$ contains a mapping for a service entry for the message recipient that maps to a confirmed ✓ status. This is captured by metafunction `contract_for` returning $(k, ✓)$ where k is the contract name for the confirmed service. Returning to the example of Section 2.2, if `client` sent a message to `server1`, i.e., $\texttt{mon}^{l_c}(\sigma_c,$ `client` $\| \; m$ `to server1`$)$, the enhanced send rule would be used if server1 was known by client's adapter to be Whip-enhanced: $\texttt{contract\_for}_{\sigma_c}(\texttt{server1}, m) = (\texttt{NoteStore}, ✓)$.

When the recipient of the message is known to be Whip-enhanced (i.e., confirmed), the adapter transforms the message by "lifting" it to an enhanced message via the `lift` function. On the receiving end, the recipient adapter will "lower" the enhanced message it received back to an unenhanced message via the `lower` metafunction. Figure 2.10 presents the definitions of `lift` and `lower` as pseudocode. Both metafunctions imperatively update the adapter's state σ as a result of contract checking, confirmation propagation, and blame propagation. (We defer explaining the state update functions until Section 2.4.5.) We first describe `lift` and then `lower`.

Metafunction $\texttt{lift}_\sigma$ takes as arguments: the name of the contract to check the message against (k); confirmation status of the other communication party (c); the intercepted message to "lift" (m); and the label of the adapter of the sender of the message (l).[1] It returns a tuple $(\sigma', \widehat{m})$ where σ' is the implicitly returned final updated state of the adapter, and $\widehat{m}$ is the transformed enhanced message. We now describe each sub-task of `lift`.

[1] For Enhanced-Send, this is the adapter's own label.

Extract contract information `lift` extracts the contract information from the message it is processing using the `entries` metafunction to produce the service entry for the message, the request entry, and service entry of the identified service in the message. Internally it uses helper metafunctions to parse the message: `from`(m) extracts the origin of the message m and `to`(m) extracts the destination of the message m. For simplicity we assume two *message parsing* metafunctions that can parse Whip-specific information from a message payload: `index`(m) is the index for the contract family and corresponds to the tag **where index is** in the Whip IDL; `identified` corresponds to the result of the **identifies** tag, and indicates that the payload *identifies* a service that should implement a certain indexed contract. Returning to the running example of Section 2.2, server1's reply message to client's request for listlinkedNotebooks is parsed as:

$$\begin{aligned}&\texttt{entries}_{\sigma_s}(\texttt{reply}\ \#n\ \texttt{from server1 : `[(share...', NoteStore)} = (\\ &\qquad \texttt{server1 satisfies NoteStore}\langle \texttt{v}\rangle,\\ &\qquad \#n\ \texttt{from client expects server1 satisfies NoteStore}\langle \texttt{v}\rangle,\\ &\qquad \texttt{server2 satisfies NoteStore}\langle \texttt{shareKey}\rangle)\end{aligned}$$

In this example v is the index used when the client called listLinkedNotebooks on server1.

Pre-condition checks Upon a contract violation, the `lift` metafunction constructs a log entry deriving blame from its arguments. For rule Enhanced-Send (Figure 2.9), these arguments come from the label of the adapter. That is, for failure of a pre-condition, the service sending the request (hereafter the client) is always blamed.

Post-condition checks Post-condition violations occur for the service sending the reply (hereafter the server) to the client. Whip logs a contract error with the blame labels $\widetilde{l}$ from the server's blame registry for the request entry *re*. We discuss in more detail formally how blame information is transferred in Section 2.4.5.

Message transformation The final step of `lift` is to construct a new enhanced message (line 10) to transfer information from the local adapter's state to the receiving adapter. The enhanced message includes the client's blame information for the service entry, blame labels for the service entry that was identified in m, and the client's confirmation status recorded for the identified service. This enhanced message is then sent to the receiving adapter using the Enhanced-Send rule. The

enhanced message is processed by the Enhanced-Receive rule and transformed back to an unenhanced message via the `lower` function, which we now describe.

Metafunction $\texttt{lower}_\sigma$ takes the following arguments: the name of the contract to check the message against (k); whether the sender of the message is confirmed (c); and the intercepted enhanced message to "lower" ($\widehat{m}$). It returns a tuple (σ', m) where σ' is the implicitly returned final updated state of the adapter, and m is the transformed unenhanced message.

Whereas `lift` performed contract checks and introduced new blame for service entries, `lower` only updates its internal state based on the state transferred by the sending adapter. After the adapter state is updated, the enhanced message is transformed to its unenhanced counterpart via the `raw` function which simply discards the enhanced message metadata, i.e., if $(\sigma', \widehat{m}) = \texttt{lift}_\sigma(k, c, m, l)$ then $\texttt{raw}(\widehat{m}) = m$.

Unenhanced Interaction

Rule Unenhanced-Send applies when an adapter intercepts a message sent by the black box it wraps to host b where $\texttt{contract\_for}_\sigma(b, m) = (k, \text{✗})$, meaning the destination of the message is not known to be enhanced due to its confirmation status ✗.[1] Rule Unenhanced-Receive fires when the adapter intercepts an unenhanced message intended for the the black box it wraps, and the black box's host b should implement contract family k.

In either case, the adapter takes a best-effort approach to perform contract checking and provide as precise as possible blame information. Specifically, the adapter performs the contract checking and blame propagation that the adapters of the source and the destination of a message *would have* performed if they had opted for enhanced interaction. Thus, in both rules, we see that the adapter uses both metafunctions `lift` and `lower`, to emulate enhanced interaction. The confirmation status argument is ✗ so that the adapter will send an unenhanced reply for an unenhanced request, as the receiver may not be able to interpret an enhanced message.

One key difference between rule Unenhanced-Receive and the two enhanced rules, is that we use the unknown label † as the label for the sender when calling the two metafunctions, instead of the adapter's own label. This is because the adapter, as the recipient of the message, is *not* responsible for its contents. As mentioned in Section 2.3.3, the base label † is thus used as the label for all non-Whip-enhanced services, and when the label of the sender's adapter is not known.

[1] Note that it may be the case that the other host is Whip-enhanced, but this fact is not locally known.

```
1:  function update-lift_σ(m, c, se, re, se_id, l)
2:    if type(m) == req then                          ▷ Request sent
3:      blame_σ[se] ←? {l}                            ▷ Init blame for se
4:      blame_σ[se_id] ←? {l}                         ▷ Init blame for invoked
5:      prov_σ[se] ←? from(m) intro
6:      prov_σ[se_id] ←? from(m) intro
7:      conf_σ[re] ← c                                ▷ Record confirmation
8:    else                                            ▷ Reply sent
9:      blame_σ[se_id] ←? blame_σ(re)                 ▷ Init blame for id
10:     prov_σ[se_id] ←? se intro
11:   conf_σ[se_id] ←? ✗                              ▷ Initialize confirmation for id

12: function update-lower_σ(m̂, c, se, re, se_id)
13:   blame_σ[se_id] ←? ∅                              ▷ Init blame for id
14:   blame_σ[se_id] ← blame_σ(se_id) ∪ m̂.id-blame
15:   prov_σ[se_id] ←? learned
16:   if m̂.id-conf == ✓ then                          ▷ Id is confirmed
17:     conf_σ[se_id] ← ✓                             ▷ Promote to confirmed
18:   else                                            ▷ Id not known to be confirmed
19:     conf_σ[se_id] ←? ✗                            ▷ Initialize unconfirmed
20:   if type(m̂) == req then                          ▷ Receive request
21:     blame_σ[re] ← m̂.se-blame                      ▷ Record blame
22:     conf_σ[re] ← c                                ▷ Record confirmation
```

Figure 2.11: State update functions

Bypass Adapters

These rules apply in cases where the adapter chooses not to intercept messages, and so the messages bypass the adapter. This occurs when a Whip-enhanced service communicates with a host b for which there is either no contract information available or conflicting contract information, i.e., $\texttt{contract\_for}_\sigma(b, m)$ is undefined. The messages are not intercepted by the adapter. Messages bypass the adapter rather than getting stuck so that the presence of the adapter does not disturb traffic to and from the black box.

2.4.5 Updating State and Assigning Blame

In this section we make formal the discussion from Section 2.3.3 about how adapters update their state when they receive messages and how they keep track of blame information.

All parts of the state (except the immutable specification) are updated as the adapter intercepts and receives messages. State is updated via the `update-lift` and `update-lower` metafunctions, called from the `lift` and `lower` metafunctions, respectively. The update functions take the following inputs as arguments: the message intercepted m or $\widehat{m}$, the confirmation status c of the other communication party, the service entry of the request se, the request entry re, and the service identified in the message se_{id}. In the case of `update-lift`, it also contains the blame label of the sender of the message.

The update functions add new mappings using notation $f_\sigma[k] \leftarrow v$ where f is one of the store projection functions. For example, the result of $\texttt{blame}_\sigma[se] \leftarrow v$ is a modified state σ' where the blame registry contains a mapping $se \mapsto v$. The final updated state σ' is implicitly returned as

the first part of the result. We also use an optional assignment syntax $f_\sigma[k] \xleftarrow{?} v$ that adds mapping $f_\sigma[k \mapsto v]$ only if $k \notin f_\sigma$. Much of the subtlety of the `lift` and `lower` metafunctions is due to the propagation of blame information which is necessary for correct and precise blame assignment. We first describe `update-lift` and then `update-lower`.

Figure 2.11 describes the behaviors of the `update-lift` and `update-lower` metafunctions, given as pseudocode[1]. Conceptually, `update-lift` introduces blame information. In essence, it implements the informal description of tracking blame information in Section 2.3.3. There are two ways a service entry can be introduced into an adapter's blame registry. First, if the black box is sending a request, then both the invoked service entry and the identified service entry may be new to the adapter. That is, the adapter has not previously associated these services with indexed contracts. If so, the adapter uses its own label as the blame label for the new service entries. This can be seen in lines 3–4.

Second, if the black box is sending a response, then the identified service entry may be new to the adapter (lines 8–9). As discussed in Section 2.3.3, the blame labels for the identified service entry are the blame labels for the corresponding request entry.

Additionally, `update-lift` records the confirmation status for the request entry (line 7). That is, the adapter will send an enhanced reply if and only if it received an enhanced request.

Whereas `update-lift` introduces new blame for service entries, `update-lower` only records and merges the blame and confirmation information from the enhanced message it received. In particular, `update-lower` performs the following sub-tasks:

- The blame labels for the identified service entry from the sender's enhanced message are merged with any existing blame information the receiver had for the identified service entry (lines 13–14).
- Confirmation status for the identified service is merged. If the sender knows the identified service is confirmed then the message will contain a confirmation status of ✓ and the receiver will update its state to record that confirmation (lines 16–19).
- When processing a request, the server creates a request entry for the client request, recording the client's service entry blame (which is equal to the client's request entry blame). Confirmation information for the request entry is also recorded so that the `contract_for` metafunc-

[1] Similar to previous figures, the shaded parts of the diagram relate to metatheoretic properties of Whip and will be discussed in Section 2.5.

tion will be defined when the server sends a reply message (lines 20–22), and ensuring that the reply message will be enhanced if and only if the request message was enhanced.

- The enhanced message is transformed to its original unenhanced message the sending black box created via the `raw` function, which simply discards the enhanced message metadata (line 14).

2.5 Correct Blame

In this section, we establish the key metatheoretic result of *WM*: *correct blame assignment*.[1] The pragmatic value of a contract system depends on the correctness of blame assignment. Informally, Dimoulas et al. [23] define that a contract system assigns blame correctly if, given a value that violates a contract, it blames the component that vouched that the value meets that contract. They extend their contract calculus with provenance information and prove that the blame label reported upon a contract violation matches the provenance of the value that triggered the violation. The provenance information is not used in contract enforcement, but provides a sound basis for specifying blame correctness.

We use the same approach and extend the semantics of *WM* to track provenance. This extension is straightforward, and the tracking of provenance is meant to be obviously correct. The tracking of provenance is independent of the creation and propagation of blame information, and provides a sound basis to specify correct blame. Due to the black-box nature of services in *WM*, we cannot use the provenance tracking mechanism of Dimoulas et al. Instead, each adapter records in its local state provenance information about service entries that reflects how the adapter's registry was updated. Provenance information allows us to easily detect which adapters are responsible for introducing which service entries, and, transitively, for service entries introduced due to the reply from a service with service entry *se*, which adapters are responsible for introducing *se*.

The portions of Figures 2.8 and 2.11 shaded in gray extend *WM* to track provenance. The local state of an adapter is extended to include *provenance map V*, which maps service entries to *provenance entries*. Intuitively, an adapter's provenance map records for each service entry in the registry how the service entry was added to the registry. If the adapter learned about the service entry *se* from another service, then $V(se) = \textbf{learned}$. If information about *se* was introduced because host *a* sent a request, then $V(se) = a\ \textbf{intro}$. (Host *a* is typically the host wrapped by the adapter, but due to interaction with non-Whip-enhanced services, it may differ; see rule Unenhanced-Receive.)

[1] Complete formalisms and proofs are in Appendix A.2.

If service entry se_{id} was introduced due to service entry se identifying se_{id} in its reply then $V(se_{id}) = se$ **intro**. The three provenance entries mirror the three ways that service entries can enter an adapter's blame registry, described in Section 2.4.4. (i.e., learned, introduced by a request, or introduced by a reply).

We designed *WM* so that local blame information is, in essence, a summary of provenance information. We express this via a *blame consistent with provenance* judgment $P \Vdash \textbf{blame}\ l\ \textbf{for}\ se$. Intuitively, if there is a post-condition violation that involves a service entry se, and $P \Vdash \textbf{blame}\ l\ \textbf{for}\ se$ holds, then the provenance of se ultimately goes back to l, and so blaming l is consistent with the provenance information.

More generally, if $P \Vdash \textbf{blame}\ l\ \textbf{for}\ se$ holds, then either l is responsible for introducing se, or se was introduced by a response from a service with service entry se' and $P \Vdash \textbf{blame}\ l\ \textbf{for}\ se'$. The following rules for the judgment are the base cases for, respectively enhanced interaction and unenhanced interaction, corresponding to host a introducing service entry se due to sending a request:

$$\frac{\mathtt{prov}_\sigma(se) = a\ \textbf{intro}}{\mathcal{P}[\mathtt{mon}^l(\sigma, P^a)] \Vdash \textbf{blame}\ l\ \textbf{for}\ se} \qquad \frac{\mathtt{prov}_\sigma(se) = b\ \textbf{intro} \quad b \neq a}{\mathcal{P}[\mathtt{mon}^l(\sigma, P^a)] \Vdash \textbf{blame}\ \dagger\ \textbf{for}\ se}$$

The second base case corresponds to when unenhanced interaction means that an adapter can not know the precise provenance (or blame) for service entry se. This is the only case where Whip introduces non-precise blame. In any other case blame information pinpoints accurately a set of black boxes that if a programmer inspects, she will detect the source of the bug.

The inductive case involves blame assigned to a service entry se_{id} that was introduced by the reply from a service entry se. Intuitively, we should blame whoever introduced se, as se_{id} is part of the higher-order result of se. This intuition is captured by the following rule:

$$\frac{\mathtt{prov}_\sigma(se_{id}) = se\ \textbf{intro} \quad \mathcal{P}[\mathtt{mon}^l(\sigma, P^a)] \Vdash \textbf{blame}\ l'\ \textbf{for}\ se}{\mathcal{P}[\mathtt{mon}^l(\sigma, P^a)] \Vdash \textbf{blame}\ l'\ \textbf{for}\ se_{id}}$$

Blame consistency with provenance, together with certain reasonable assumptions on the initial state of adapters, forms a well-formedness predicate which we show is *preserved* as the process evolves. Well-formedness of *WM* requires that all blame information in an adapter's registry is consistent with provenance, which is sufficient to define *correct blame*. Intuitively, if a well-formed process P takes a step and this step results in an adapter in P detecting a contract violation, then (1) for a violation of a pre-condition, *WM* blames the sender of the request message that caused the violation since they previously "agreed" to the contract by using it; (2) for a violation of a post-condition, *WM* blames consistently with provenance. Formally, our theorem is:

Correct Blame. If well-formed $P_1 = \mathcal{P}_1[\mathtt{mon}^l(\sigma_1, P_1^a)]$ and $P_1 \rightarrow P_2$ and $P_2 = \mathcal{P}_2[\mathtt{mon}^l(\sigma_2, P_2^a)]$ and $\mathtt{errors}_{\sigma_2} = \{le\} \cup \mathtt{errors}_{\sigma_1}$ then

1. if $le = \mathtt{Pre}(se, l_e)$, then
   (a) if $P_1 = \mathcal{P}[\mathtt{mon}^l(\sigma_1, P^a \parallel m \ \mathtt{to}\ b)]$ and $P_2 = \mathcal{P}[\mathtt{mon}^l(\sigma_2, P^a) \parallel m' \ \mathtt{to}\ b]$ then $l_e = l$
   (b) if $P_1 = \mathcal{P}[\mathtt{mon}^l(\sigma_1, P^a) \parallel m \ \mathtt{to}\ a]$ and $P_2 = \mathcal{P}[\mathtt{mon}^l(\sigma_2, P^a \parallel m \ \mathtt{to}\ a)]$ then $l_e = \dagger$
2. if $le = \mathtt{Post}(se, \widetilde{l})$, then $\forall l \in \widetilde{l}.\ P_2 \Vdash \textbf{blame}\ l\ \textbf{for}\ se$.

2.6 Whip in Practice

We have developed a prototype implementation of Whip. It consists of the adapter described in Section 2.3 (and formalized in Section 2.4), and an interposition library for redirecting TCP connections through the adapter. The adapter is about 3,800 non-empty lines of Python and the interposition library is about 250 non-empty lines of C.

As described in Section 2.3, before deployment, users configure Whip adapters with information that describes: (i) what is the contract of the Whip-enhanced service the adapter enhances; and (ii) what are the contracts for other well-known services whose interaction with the Whip-enhanced service the adapter should monitor. Upon deployment, a Whip-enhanced service is linked with the interposition library. At run time, the library intercepts connect system calls from the service and contacts the adapter to check whether a new connection should bypass the adapter or not (based on the adapter's blame registry). The adapter's local state (see Section 2.3) is stored in a disk-backed permanent store, with an in-memory cache for performance. When a cache miss occurs, the requested data is fetched to memory if found. We leave garbage collection of on-disk adapter state as future work but note it can be added without significant changes to Whip's design: the lifetime of the information in the adapter state can be determined from user-provided configuration directives or with additional constructs in the Whip IDL that specify the scope of a contract.

Whip supports any message format given an appropriate message format plugin. We have implemented plugins for Thrift (in 150 lines of code), REST (100 lines) and SOAP (400 lines). To check contracts on encrypted communications (i.e., a service using TLS), the adapter and the service it enhances can share certificates or use a mutually trusted certificate authority to allow the adapter to decrypt messages for the black box.

We have used Whip to harden the interfaces of three real-world off-the-shelf services: Evernote (from Section 2.2), the Twitter API, and an online correspondence chess service. We discuss the most interesting aspects of the case studies in the remainder of this section, and discuss performance in Section 2.7.

2.6.1 Evernote

The Evernote case study showcases four of the aspects of Whip's runtime we discuss in context in Section 2.1: (i) Whip treats the Evernote server and its clients as black-boxes; (ii) Whip is partially deployed as we cannot enhance the Evernote servers; (iii) Whip does not change communication patterns between the Evernote server and its clients so as not to disrupt their operation; and (iv) Whip operates both on top of Evernote's Thrift-based API and its simpler HTTPS protocol for OAuth authentication requests.

As in Section 2.2, we designed Whip contracts for Evernote's API based on its informal documentation. We use *first-order Whip contracts* to express a variety of first-order properties similar to the two first-order properties from Section 2.2: non-empty strings, bounds checks on integers, malformed GUIDs, strings that are too long, missing parameters that could not be marked as required due to Thrift limitations, and strings not matching certain patterns (e.g., valid MIME type). We use *indexed higher-order Whip contracts* to express properties about the correct use of a multitude of tokens (similar to the second higher-order property in Section 2.2) despite some of these tokens originating from OAuth rather than Thrift services.

2.6.2 Twitter

Twitter's REST API[1] allows access to a user's tweets and followers, and is representative of many REST APIs. Its documentation has a series of examples that highlight key properties of the API. We use Whip contracts to turn these examples into a precise and executable specification. Beyond the Evernote case study, the Twitter case study showcases that (i) Whip is compatible with the most popular message format for microservices, REST; and (ii) the Whip contract language allows programmers to write precise contracts with minimum effort reusing code from Python libraries.

First-order Contracts for Well-formed Data. We employed first-order Whip contracts to express a variety of properties of arguments and results of operations of the Twitter API. For example, the operation to fetch tweets must consume either a user ID or a screen name. We encode this disjunctive requirement with a pre-condition. Few API libraries actually defensively check this

[1] `https://dev.twitter.com`

requirement but instead rely on the server to report back an error message. In addition, we used a post-condition to capture that the result of the operation should be a list of length equal to one of the arguments of the operation (or at most 200 elements).

Some of the properties required careful syntactic checks. Instead of performing these checks ourselves, we leveraged third-party Python libraries to perform the data validation. The Whip contract language allows importing packages via a familiar Python syntax **from** X **import** Y where X is the package name and Y is the name to import. In one case, dates needed to conform to the RFC 822 standard, so the contract imports the parsedate_tz function from the rfc822 Python package.

The following Whip contract language snippet exemplifies how we expressed these properties:

```
from rfc822 import parsedate_tz
service Twitter {
  /1.1/statuses/user_timeline(req)
  @requires « 'user_id' in req.args or 'screen_name' in req.args »
  @ensures «
    assert type(result) == list
    assert ('count' not in req.args or length(result) <= max(200, req.args.count))
    for tweet in result: assert parsedate_tz(tweet.created_at) != None
  »
  ...
}
```

A Higher-order Contract for Valid Tweet IDs. Outside the correct use of OAuth tokens, the correct behavior of Twitter operations depends on the correct use of unique tokens that denote other types of data, such as tweets. We discuss here an example of such a requirement; retweets should involve tokens that correspond to actual tweets. That is, for a request /1.1/statuses/retweet/<id>.json(req), id should be the unique token of an actual tweet. Consequently, a retweet request should use only an id retrieved from a request /1.1/statuses/user_timeline(req) or similar whose reply contains a list of tokens for actual tweets. The following Whip contract expresses this requirement:

```
service Twitter {
    /1.1/statuses/user_timeline(req)
    @foreach tweet in « result» identifies Twitter at receiver with index « 'tweet:' + tweet.id »
    ...

    /1.1/statuses/retweet/<id>.json(req)
    @where index is « 'tweet:' + id »
    @ensures « 'does not exist' not in result.get('errors') »
    ...
}
```

The result of the user_timeline operation identifies that the **receiver** service, i.e., the service that receives a request for this operation, implements contract Twitter⟨'tweet:' + t.id⟩, where t ranges over the tokens in the result of the operation. For a retweet operation, the receiver service must implement contract Twitter⟨'tweet:' + id⟩, where id is part of the request URL. Otherwise, if the post-condition of the operation fails, Whip blames the client for incorrectly claiming that the retweet involved an actual tweet.

A Higher-order Contract for Valid OAuth Tokens. Twitter, like Evernote, uses the OAuth protocol for authentication. The API describes that OAuth tokens passed as arguments should originate from an appropriate OAuth service request. We express the validity of OAuth tokens in a similar manner to the two higher-order properties of Evernote's API from Section 2.2:

```
from urlparse import parse_qs
service TwitterOAuth {
    /oauth/access_token(req)
    @identifies Twitter at receiver with index « 'oauth:' + parse_qs(result.content).get('oauth_token') »
}
service Twitter {
    /1.1/status/user_timeline.json(request)
    @where index is « 'oauth:' + request['headers'].get('Authorization') »
}
```

We use the parse_qs function from the urlparse package to parse the querystring of the resulting OAuth access token request in order to retrieve the OAuth access token *t*. The access token is used

to identify an indexed contract Twitter⟨'oauth:' + *t*⟩, which is later used in a subsequent request for the user_timeline operation.

2.6.3 Xfcc Correspondence Chess

Xfcc is a popular web service (WSDL) specification for correspondence chess.[1] The specification offers a standard for server implementations that manage chess games recognized by the World Chess Federation (FIDE). The specification describes two operations: GetMyGames returns the status of all games the user is playing in, and MakeAMove performs a game action (e.g., move a piece, offer a draw). Beyond the Evernote and Twitter case studies, the Xfcc case study showcases that (i) Whip is compatible with the standard message format for traditional web services, SOAP (WSDL); and (ii) Whip is compatible with a diversity of service implementations; (iii) Whip can detect specification violations in both servers and clients; and (iv) indexed contracts can encode complex conditions for the successful call of a service operation.

A First-order Contract for Valid PGN Moves. The GetMyGames operation of Xfcc returns a data structure that represents the status of a game. This data structure includes a moves field that specifies the history of the moves of the game in Portable Game Notation (PGN) format. Similar to the validity of dates in the Twitter case study, we used a third-party library to check the validity of the moves field. The read_game function from the chess.pgn package parses a string containing the list of moves in PGN format and returns a Python structure representing the game. When the parsing fails it throws an exception. With this function in hand, we wrote a contract that ensures that all games are in valid PGN format. The contract succeeds if the read_games function terminates without throwing an exception:

```
from chess.pgn import read_game
service Chess {
    GetMyGames(username, password)
    @ensures «
        for game in result:
            try: read_game(game['moves'])
            except: return False »
    ...
}
```

[1] http://xfcc.org/

A Higher-order Contract for Valid Game IDs. The documentation of Xfcc states that when a client provides an invalid game ID to MakeAMove the server should return error code InvalidGameID. Whip can express this property with a contract analogous to the contract for valid tweet IDs in the Twitter case study. We discovered that two popular Xfcc servers return an database error page rather than the documented correct error code. We also found that a popular client was unable to interpret the return code, making an invalid move look successful to its user.

A Higher-order Contract for Accepting a Draw Only When Allowed. The documentation states that draw offers are active only for one move and a player can accept a draw only for a game with an active draw offer. To make a draw offer to an opponent, a player passes **True** as the offerDraw argument of the MakeAMove operation of Xfcc. To accept the draw, the opponent passes **True** as the acceptDraw argument of their immediate next MakeAMove invocation. If a player does not follow the protocol for accepting a draw, the service should return the NoDrawWasOffered error code. Whip can express the compliance of players with the draw protocol with a higher-order indexed contract. This use of indexed contracts differs from those we have seen so far. While in the Evernote and Twitter case studies, we used indices to pair the code of an operation with its "environment," in the Xfcc case study we used indices to check a property of this "environment." In more detail, the Chess contract describes that the result of GetMyGames identifies that GetMyGames's **receiver** service implements contracts Chess⟨(g['gameId'], movecount(g), **False**)⟩ where gameId is the game ID of each game g in the result of the operation. Additionally, if a game's drawOffered flag is **True** (i.e., the opponent has offered a draw), the **receiver** service of GetMyGames also implements contract Chess⟨(g['gameId'], movecount(g), **True**)⟩. The following snippet puts these pieces together:

```
service Chess {
    GetMyGames(username, password)
    @foreach g in « result » identifies Chess at receiver with index « (g['gameId'], movecount(g), False) »
    @foreach g in « result » identifies Chess at receiver with index « (g['gameId'], movecount(g), True) »
        when « g['drawOffered'] == True »
    ...
}
```

The fact that a service implements Chess⟨(gameId,moveCount,True)⟩ indicates the existence of a draw offer for the game with ID gameId while the opposite indicates the absence of a draw offer. Moreover, the indices include the number of moves so far in a game, movecount(g), as the "timestamp" of a draw offer. Thus indexed contracts give us a way to express and enforce draw offers: a client

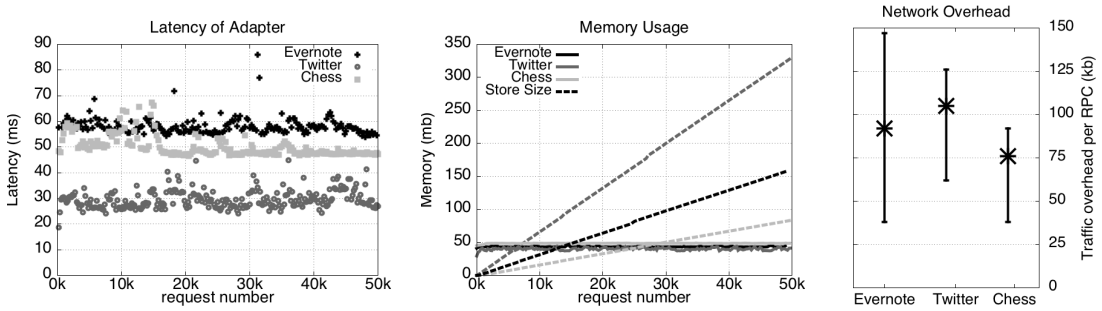

Figure 2.12: The charts show the time, memory, and network overhead for each case study. The left chart shows the latency of the adapter as the number of requests increases. Each point is the average of the 250 requests around it. The middle chart shows the resident set size of the adapter and the dashed lines show the sizes of the store on disk. The right chart shows the average amount of adapter traffic per operation call. Vertical bars indicate 95% confidence intervals.

can accept a draw at a given time in a game (i.e., acceptDraw is **True** and the game has ID gameId and moveCount moves so far), only if the **receiver** service of MakeAMove implements Chess⟨(gameId,moveCount,True)⟩:

```
service Chess {
    ...
    MakeAMove(gameId, resign, acceptDraw, movecount, offerDraw, ...)
    @where index is « (gameId, movecount, acceptDraw) »
    @ensures « result != "NoDrawWasOffered" »
}
```

In the event that the player's opponent has not offered a draw for the game with their last move, the player attempts to accept a draw, and the post-condition of MakeAMove fails, Whip blames the client for deviating from the draw protocol.

2.7 Performance

To evaluate how Whip impacts the performance of services it enhances, we analyze the time, memory, and network overhead due to Whip on the case studies from Section 2.6. We developed a test suite for each case study which exercises all the contracts from Section 2.6. All services are Whip-enhanced to maximize adapter traffic. Operations that identify a service entry always introduce the service entry (i.e., always use a new contract index and thus create new service entries in the adapters' local state, which maximizes local state size). We do not use the actual third-party services for our

experiments but instead *mock* their behavior, i.e., we simulate their behavior with pre-computed responses for each request. This is for two reasons: (1) mocking services removes several sources of measurement noise, like service latency variation from background request load, and (2) performing our experiments on third-party production servers violates their terms of use.

We collect the following measurements for each test. First, we record the time to perform each request in the test suite and receive a reply for (1) the test client alone, and (2) the client enhanced with an adapter. The difference between these two measurements yields the latency due to the client's adapter per request (adapter latency). Second, we record the amount of memory (RAM and hard-disk) used by the client's adapter. Finally, we measure the adapter-to-adapter traffic (not including the original request or reply) in the TCP stream. We measure only the client's adapter as it is the hub for all communication in each experiment.

We ran our experiments on a 3 GHz Intel Core i7 processor with 2 GB of DDR3 memory with loopback communication. Figure 2.12 shows the experimental results for each test suite. Average adapter latencies for Twitter, Chess, and Evernote are 22ms, 55ms, and 59ms respectively. To place these measurements in context, the production versions of the case studies' services have latencies approximately 20 times greater than the average adapter latency.[1] The rate of increase in the disk-backed store for Twitter, Chess, and Evernote is 6.8kb per request, 1.7kb per request, and 3.3kb per request respectively. While disk-backed storage will increase without limit, the size of the in-memory cache is capped. The experimental results show that a cache size of just a few hundred megabytes suffices to cache adapter information for tens of thousands of requests. Average network overhead for Twitter, Chess, and Evernote is 50 bytes, 54 bytes, and 70 bytes per request respectively. Variance in network overhead comes from the invoked operations identifying different service entries.

The network overhead and the rate of increase in store size depend on how many services each contract identifies. However, neither latency nor memory usage degrade as the number of requests increases despite an increasing network overhead and store size. Moreover, network overhead and store size do not have a dominant effect on latency; Twitter has the largest store and highest average network overhead yet the lowest latency. Instead, latency depends largely on the efficiency of the network plugin; the REST plugin uses a more efficient marshaller and handles sockets more performantly than the other plugins. Finally, all services in the experiments have a definite finite scope (according to their documentation) and so could be safely garbage collected at some point as discussed in Section 2.6.

[1] For example, see `https://dev.twitter.com/overview/status`.

Existing frameworks for composing services can enforce higher-order behavioral contracts similar to Whip's but assume that services are written and deployed in a particular manner. For example, CORBA [61], BPEL [40], and Java RMI [83] require all services to use their libraries. Whip supports compositions of services that do not or only partly use these middlewares with appropriate message format plugins.

Behavioral Interface Specification Languages (BISLs), such as JML [45], have extensions for specifying and enforcing higher-order behavioral contracts for communicating components. However, these languages are tightly coupled with particular component-implementation languages or families of languages. For instance, JML is designed for Java programs and has specific features to handle inheritance. Also, tools based on these languages re-write programs to insert checking probes. Thus BISLs and their contract checking tools are not language-agnostic. In contrast, Whip and its IDL are language-agnostic and do not modify services' code. Some features of Whip's IDL, such as pre- and post-conditions, are common with most BISLs. Others, such as **identifies**, are unique to Whip. Runtime verification tools, such as Monitor-Oriented Programming (MOP) [15], can in principle enforce higher-order behavioral contracts for communicating components in a language agnostic manner (with appropriate plugins). However, building a contract system on top of them requires first solving the semantic issues Whip solves.

Many techniques exist to enhance the reliability of distributed systems (e.g., [37, 42, 62]) and are compatible with and orthogonal to Whip. Indeed, modern services are often chosen for organizational concerns such as loose coupling and scalability, rather than reliability. We focus on functional correctness of modern service composition. We briefly discuss how Whip affects the failure model of distributed applications in Section 2.3.

A wide range of frameworks enforce synchronization protocols of communicating components. For example, finite state machines can constrain the order of WSDL-defined interactions [47] and web browsing [34] in a manner complementary to that of Whip. BPEL [40] is an expressive specification language for the orchestration of web services. Enforcement of BPEL, though, relies on a centralized communication bus for all services in an application. Multi-party session types [36] assume a global coordination protocol that is broken into locally and statically enforceable pieces. Further extensions marry multi-party session types with Design by Contract [13]. In general, dynamic monitoring of multi-party session types shares the same motivation as Whip [38]. Even though, in principal at least, the combination of session types with contracts and dynamic monitoring leads to specifications that subsume those of Whip, runtime verification of session types depends on an-

notating the source code of or using particular libraries by all components involved in a protocol. In contrast, the black-box treatment of (legacy) services and partial deployment are key aspects of Whip.

Closer to Whip, the work of Jia et al. [39] describes the theory of a runtime monitor with precise blame for higher-order session types. Besides the fact that higher-order session types alone do not subsume Whip's higher-order contracts, there are important differences between the mechanism of Jia et al. and Whip. First, for correct blame, many operations of Jia et al.'s model (e.g., cut and forwarding) affect the topology of communication introducing indirect message queues. In contrast, Whip only affects the communication topology locally to each service and introduces no intermediate indirection to service communication. Second, for precise blame, the mechanism of Jia et al. requires that monitors have access to shared state. Whip adapters have access only to local state. Furthermore, Whip adapters do not need to exchange any extra messages to keep their local states in sync; new information is inferred from or piggybacked onto messages that services exchange. Third, it is unclear how their mechanism handles legacy services. For example, the use of explicit direction shift messages due to polarized session types is incompatible with existing message protocols. Whip is compatible with any protocol built on top of TCP.

2.9 Conclusion

Whip enhances modern services with higher-order behavioral contracts to bridge the semantic gap between simple network protocols and the higher-order properties of services. Whip comes with a higher-order contract language tailored to the needs of modern services. Moreover, Whip is transparent, suitable for partial deployment, and compatible with popular message formats. Thus, Whip promotes the correct composition of modern service-oriented applications, including legacy services, and with correct blame assignment facilitates their debugging and maintenance.

3

Background on LIO and DC Labels

In this chapter we describe the necessary background for Chapters 4 and 5. Restricted privileges and Clio are based on the information flow control language LIO [77] which uses DC labels [76] as its policy language.

3.1 DC Labels

We call the set of all positive propositional formulas in conjunctive normal form CNF; we use the term *formula* to range over CNF. The primitive propositions of formulas are *principals* in the system. Principals are application-specific entities; for example they may refer to users in the system. DC labels [76] are pairs of confidentiality and integrity formulas. Confidentiality formulas describe who may learn information. Integrity formulas describe who takes responsibility or vouches for information. Both confidentiality and integrity formulas are in CNF. We assume that operations on formulas always reduce their results to conjunctive normal form.

$$
\begin{aligned}
C_1 \sqsubseteq^{\mathrm{C}} C_2 &\iff C_2 \Rightarrow C_1 \\
C_1 \sqcup^{\mathrm{C}} C_2 &\iff C_1 \wedge C_2 \\
C_1 \sqcap^{\mathrm{C}} C_2 &\iff C_1 \vee C_2 \\
\bot^{\mathrm{C}} \equiv \mathit{True} &\qquad \top^{\mathrm{C}} \equiv \mathit{False}
\end{aligned}
$$

Figure 3.1: Confidentiality lattice

$$
\begin{aligned}
I_1 \sqsubseteq^{\mathrm{I}} I_2 &\iff I_1 \Rightarrow I_2 \\
I_1 \sqcup^{\mathrm{I}} I_2 &\iff I_1 \vee I_2 \\
I_1 \sqcap^{\mathrm{I}} I_2 &\iff I_1 \wedge I_2 \\
\bot^{\mathrm{I}} \equiv \mathit{False} &\qquad \top^{\mathrm{I}} \equiv \mathit{True}
\end{aligned}
$$

Figure 3.2: Integrity lattice

$$
\begin{array}{rcl}
\langle C_1, I_1\rangle \sqsubseteq \langle C_2, I_2\rangle & \iff & C_1 \sqsubseteq^{\mathrm{C}} C_2 \text{ and } I_1 \sqsubseteq^{\mathrm{I}} I_2 \\
\langle C_1, I_1\rangle \sqcup \langle C_2, I_2\rangle & \equiv & \langle C_1 \sqcup^{\mathrm{C}} C_2, I_1 \sqcup^{\mathrm{I}} I_2\rangle \\
\langle C_1, I_1\rangle \sqcap \langle C_2, I_2\rangle & \equiv & \langle C_1 \sqcap^{\mathrm{C}} C_2, I_1 \sqcap^{\mathrm{I}} I_2\rangle \\
\mathbb{C}(\langle C, I\rangle) \equiv C & & \mathbb{I}(\langle C, I\rangle) \equiv I
\end{array}
$$

Figure 3.3: Security lattice for DC labels

$$\langle \mathsf{Alice}, \mathsf{Charlie}\rangle \not\sqsubseteq \langle \mathsf{Alice}, \mathsf{Charlie} \wedge \mathsf{Alice}\rangle$$

Figure 3.4: Downgrading integrity

$$\langle \mathsf{Alice} \wedge \mathsf{Bob}, \mathsf{Charlie}\rangle \not\sqsubseteq \langle \mathsf{Bob}, \mathsf{Charlie}\rangle$$

Figure 3.5: Downgrading confidentiality

Both confidentiality formulas and integrity formulas form lattices—see Figures 3.1 and 3.2 for their formal definitions. We interpret $C_1 \sqsubseteq^{\mathrm{C}} C_2$ as: C_2 is at least as confidential as C_1. For instance, $\mathsf{Alice} \vee \mathsf{Bob} \sqsubseteq^{\mathrm{C}} \mathsf{Alice}$, which means that data readable by either Alice or Bob is less confidential than data readable only by Alice. Conjunctions of principals represent the multiple interest of principals to protect the data. Conversely, disjunctions of principals represent groups wherein any member may learn the information. The integrity lattice is dually defined [11]; we interpret $I_1 \sqsubseteq^{\mathrm{I}} I_2$ as: I_1 is at least as trustworthy as I_2. For example, $\mathsf{Alice} \wedge \mathsf{Bob} \sqsubseteq^{\mathrm{I}} \mathsf{Alice}$, which indicates that data vouched for by $\mathsf{Alice} \wedge \mathsf{Bob}$ is more trustworthy than data vouched for only by Alice. In this case, conjunctions of principals represent groups whose members are independently responsible for the information. For example, data with integrity $\mathsf{Alice} \wedge \mathsf{Bob}$ means that Alice is completely responsible for the data, and so is Bob. Conversely, disjunctions of principals represent groups that collectively take responsibility for the information, however, no principal takes sole responsibility. For example, data with integrity $\mathsf{Alice} \vee \mathsf{Bob}$ means that Alice and Bob collectively are responsible for the data, i.e., both may have contributed to, or influenced the computation of the data.

Formally, a DC label is a pair of a confidentiality formula C and an integrity formula I, written $\langle C, I\rangle$. DC labels form a product lattice given in Figure 3.3. The $\sqsubseteq$ relation is called the *can-flow-to* relation because it describes information flows that respect confidentiality and integrity formulas. We write $\mathbb{C}(\cdot)$ and $\mathbb{I}(\cdot)$ for the projection of confidentiality and integrity components, respectively.

Downgrading

In the DC label model, information from one security label is *downgraded* to another security label if the relabeling does not satisfy the can-flow-to relation. Consider the pair of security labels in Figure 3.4. The first security label enforces the policy that data is vouched for by Charlie. The second

$$\langle C_1, I_1\rangle \sqsubseteq_p \langle C_2, I_2\rangle \iff C_1 \sqsubseteq^{\mathrm{C}}_p C_2 \text{ and } I_1 \sqsubseteq^{\mathrm{I}}_p I_2$$
$$\text{where} \quad C_1 \sqsubseteq^{\mathrm{C}}_p C_2 \iff C_1 \sqsubseteq^{\mathrm{C}} C_2 \sqcup^{\mathrm{C}} p$$
$$I_1 \sqsubseteq^{\mathrm{I}}_p I_2 \iff I_1 \sqcap^{\mathrm{I}} p \sqsubseteq^{\mathrm{I}} I_2$$

Figure 3.6: Relation can-flow-to-with-privilege-p

security label enforces the policy that data is vouched for by Charlie and Alice, therefore a secure system cannot permit data to flow from the sources protected by the first policy to sinks protected by the second policy. This downgrade is an *endorsement*, since it downgrades only integrity, i.e., it makes a value more trustworthy. Dually, a *declassification* downgrades only confidentiality, i.e., it makes a value less confidential. Consider the pair of security labels in Figure 3.5: The first security label enforces the policy that data is confidential to Alice $\wedge$ Bob. The second security label enforces that data is confidential to Bob. Permitting data to flow from a source protected by the first policy to a sink protected by the second policy violates the confidentiality expectations of the source. And so, downgrading is considered an *unsafe* operation as the relabeling may violate the the security policy of the information. In practice, downgrading is not permitted during normal execution of the program and can only be done under special privileged circumstances.

Privileges

Practical systems must permit some downgrading. The DC label model controls downgrading with *privileges*, where every principal has an associated privilege, and a principal's privilege enables downgrading. More precisely, given principal p, the *can-flow-to-with-privilege-p* relationship, written $\sqsubseteq_p$, describes the information flows permitted with p's privilege—see Figure 3.6. Observe that both downgrading examples from the previous section are now permitted by the can-flow-to-with-privilege relationship for the principal Alice, i.e., $\langle \mathsf{Alice}, \mathsf{Charlie}\rangle \sqsubseteq_{\mathsf{Alice}} \langle \mathsf{Alice}, \mathsf{Charlie} \wedge \mathsf{Alice}\rangle$ and $\langle \mathsf{Alice} \wedge \mathsf{Bob}, \mathsf{Charlie}\rangle \sqsubseteq_{\mathsf{Alice}} \langle \mathsf{Bob}, \mathsf{Charlie}\rangle$.

3.1.1 Floating label systems

DC labels are usually part of *floating label systems* like LIO [77], Hails [30], and COWL [79]. Such systems associate a *current label*, L_{pc}, with every computational task—this label plays a role similar to the *program counter* (PC) in more traditional language-based IFC approaches [69]. The current label denotes the fact that a computation depends only on data with labels bounded above by L_{pc}. When a task with current label L_{pc} observes information with label L_A, the current label after observation, L'_{pc}, must "float" above both the previous current label and the observed information's label, i.e., $L'_{pc} = L_{pc} \sqcup L_A$. Importantly, and to respect the security lattice, the current label restricts

the subsequent writes to communication channels. Specifically, a task with current label L_{pc} is prevented from writing to channels protected by formula L_A if $L_{pc} \not\sqsubseteq L_A$.

Floating-label systems typically use some run-time representation of principals' privilege, and downgrading operations require the run-time representation of a principal p's privilege to be presented in order to use the can-flow-to-with-privilege-p relation, $\sqsubseteq_p$. Thus, the run-time representation of a principal's privilege acts like a capability to downgrade that principal's information. We write $p^\sharp$ for the run-time representation of the privilege of principal p, and refer to this value as a *raw privilege* (to contrast it with the restricted privileges that are introduced in Chapter 4).

3.2 LIO

LIO is a dynamic, floating-label approach to language-based information flow control [77]. LIO uses Haskell features to control how sensitive information is used and to restrict I/O side-effects. In particular, it implements an embedded language and a runtime monitor based on the notion of a *monad*, an abstract data type that represents sequences of actions (also known as *computations*) that may perform side-effects. The basic interface of a monad consists in the fundamental operations return and $(\gg\!=)$ (read as "bind"). The expression return x denotes a computation that returns the value denoted by x, performing no side-effects. The function $(\gg\!=)$ is used to *sequence* computations. Specifically, $t \gg\!= \lambda x.t'$ takes the result produced by t and applies function $\lambda x.t'$ to it (which allows computation t' to depend on the value produced by t). In order to be useful, monads are usually extended with additional primitive operations to selectively allow the desired side-effects. The *LIO* monad is a specific instance of this pattern equipped with IFC-aware operations that enforce security.

LIO, like many dynamic IFC approaches (e.g., [16, 67, 91]), employs a floating label. Security concerns are represented by DC labels. The runtime monitor of LIO maintains as part of its the state the current label of computation. LIO operations adjust this label when sensitive information enters the program and use it to validate (or reject) outgoing flows.

Once the current label within a given computation is raised, it can never be lowered. This can be very restrictive, since, for example, as soon as confidential data is accessed by a computation, the computation will be unable to output any public data. To address this limitation, the toLabeled operation allows evaluation of an LIO computation m in a separate *compartment*: toLabeled l m will run m to completion, and produce a *labeled value* $\langle v : l \rangle$, where v is the result of computation m, and l is an over-approximation of the final current label of m. Note that the current label of the enclosing computation is not affected by executing toLabeled l m. In general, given a labeled value $\langle v : l \rangle$, label

$$
\begin{array}{rrl}
\text{Ground Value:} & \underline{v} & ::= \mathsf{true} \mid \mathsf{false} \mid () \mid l \mid (\underline{v}, \underline{v}) \\
\text{Value:} & v & ::= \underline{v} \mid (v, v) \mid x \mid \lambda x.t \mid t^{\mathsf{LIO}} \mid \langle v : l \rangle \\
\text{Term:} & t & ::= v \mid (t, t) \mid t\ t \mid \mathsf{fix}\ t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \\
 & & \mid t_1 \sqcup t_2 \mid t_1 \sqcap t_2 \mid t_1 \sqsubseteq t_2 \\
 & & \mid \mathsf{return}\ t \mid t \gg\!= t \\
 & & \mid \mathsf{label}\ t\ t \mid \mathsf{labelOf}\ t \mid \mathsf{unlabel}\ t \\
 & & \mid \mathsf{getLabel} \mid \mathsf{getClearance} \mid \mathsf{lowerClearance}\ t \\
 & & \mid \mathsf{toLabeled}\ t\ t \mid {}^{l}_{l}\{^{l}\ t\ \} \\
 & & \mid \mathsf{store}\ t\ t \mid \mathsf{fetch}_{\tau}\ t\ t \\
\text{Ground Type:} & \underline{\tau} & ::= \mathsf{Bool} \mid () \mid \mathsf{Label} \mid (\underline{\tau}, \underline{\tau}) \\
\text{Type:} & \tau & ::= \underline{\tau} \mid (\tau, \tau) \mid \tau \to \tau \mid \mathsf{LIO}\ \tau \mid \mathsf{Labeled}\ \tau
\end{array}
$$

Figure 3.7: Syntax for LIO values, terms, and types.

l is an upper bound on the information conveyed by v. Labeled values can also be created from raw values using operation label, and a labeled value can be read into the current scope with operation unlabel. Creating a labeled value with label l can be regarded as writing into a channel at security level l. Similarly, observing (i.e., unlabeling) a labeled value at l is analogous to reading from a channel at l.

LIO SECURITY GUARANTEES LIO provides a termination-insensitive *noninterference-based* security guarantee [31]. Intuitively, if a program is noninterfering with respect to confidentiality, then the public outputs of a program reveal nothing about the confidential inputs. More precisely, an attacker $\mathcal{A}$ that can observe inputs and outputs with confidentiality label at most $l_{\mathcal{A}}$ learns nothing about any input to the program with label l such that $l \not\sqsubseteq l_{\mathcal{A}}$. Similarly, a program is noninterfering for integrity if an attacker that can control untrusted inputs cannot influence trusted outputs.

3.2.1 LIO Calculus

This section provides a description of the LIO language and its formalization.

LIO CALCULUS LIO is formalized as a typed λ-calculus with call-by-name evaluation, in the same style as Stefan et al. [77]. Figure 3.7 gives the syntax of LIO values, terms, and types.

Security labels have type Label and labeled values have type Labeled τ. Computation on labeled

values occur in the LIO monad using the return and $(\gg\!=)$ monadic operators. The nonterminals t^{LIO} and $\frac{l}{l'}\{^{l''}\ t\}$ are generated only by intermediate reduction steps and are not valid source-level syntax. For convenience, we also distinguish values that can be easily serialized as *ground values*, $\underline{v}$. Ground values are all values except functions and LIO computations.

Static type checking is performed in the standard way. We elide the typing rules $\vdash t : \tau$ since they are mostly standard[1]. LIO enforces information flow control dynamically, so it does not rely on its type system to provide security guarantees.

The semantics is given by a small-step reduction relation $\longrightarrow$ among LIO configurations (shown in Figure 3.8)[2]. Configurations are of the form $\langle l_{\text{cur}}, l_{\text{clr}} \mid t\rangle$, where l_{cur} is the current label and t is the LIO term being evaluated. Label l_{clr} is the *current clearance* and is an upper bound on the current label l_{cur}. The clearance allows a programmer to specify an upper bound for information that a computation is allowed to access. We write $c \longrightarrow c'$ to express that configuration c can take a reduction step to configuration c'. We define $\longrightarrow^*$ as the reflexive and transitive closure of $\longrightarrow$. Given configuration $c = \langle l_{\text{cur}}, l_{\text{clr}} \mid t\rangle$ we write $\text{PC}(c)$ for l_{cur}, the current label of c.

Rules RETURN and BIND encode the core monadic operations. In rule LABEL, the operation label $l\ \underline{v}$ returns a labeled value with label l holding $\underline{v}$ ($\langle \underline{v} : l\rangle$), provided that the current label flows to l ($l_{\text{cur}} \sqsubseteq l$) and l flows to the current clearance ($l \sqsubseteq l_{\text{clr}}$). Note that we force the second argument to be a ground value, i.e. it should be fully normalized. Rule UNLABEL expresses that, given a labeled value lv with label l, the operation unlabel lv returns the value stored in lv and updates the current label to $l_{\text{cur}} \sqcup l$, to capture the fact that a value with label l has been read, provided that this new label flows to the current clearance ($l \sqsubseteq l_{\text{clr}}$). The operations getLabel and getClearance can be used to retrieve the current label and clearance respectively.

Rules TOLABELED and RESET deserve special attention. To evaluate toLabeled $l_2\ t$, we first check that l_2 is a valid target label ($l_{\text{cur}} \sqsubseteq l_2 \sqsubseteq l_{\text{clr}}$) and then wrap t in a compartment using the special syntactic form $\frac{l_{\text{cur}}}{l_{\text{clr}}}\{^{l_2}\ t\}$, recording the current label and clearance at the time of entering toLabeled and the target label of the operation, l_2. Evaluation proceeds by reducing t in the context of the compartment to a value of the form $t_{\text{I}}{}^{\mathsf{LIO}}$. Next, the rule RESET evaluates the term $\frac{l_{\text{cur}}}{l_{\text{clr}}}\{^{l_2}\ t_{\text{I}}{}^{\mathsf{LIO}}\}$, first checking that the current label flows to the target of the current toLabeled ($l_{\text{cur}} \sqsubseteq l_2$). Finally, the compartment is replaced by a normal label operation and the current label and clearance are restored to their saved values.

[1] Complete definitions given in Appendix C.1.3.

[2] The rest can be found in Appendix C.1.4.

LabelOf

$$\overline{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{labelOf}\ (\langle t : l_1 \rangle)\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid l_1\rangle}$$

Return

$$\overline{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{return}\ t\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid t^{\mathsf{LIO}}\rangle}$$

Bind

$$\overline{\langle l_{\text{cur}}, l_{\text{clr}} \mid t_1{}^{\mathsf{LIO}} \gg\!= t_2\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid (t_2\ t_1)\rangle}$$

Label

$$\frac{l_{\text{cur}} \sqsubseteq l_1 \qquad l_1 \sqsubseteq l_{\text{clr}}}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{label}\ l_1\ \underline{v}\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{return}\ (\langle \underline{v} : l_1 \rangle)\rangle}$$

Unlabel

$$\frac{l_{\text{cur}} \sqcup l_1 = l_2 \qquad l_2 \sqsubseteq l_{\text{clr}}}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{unlabel}\ (\langle t_2 : l_1 \rangle)\rangle \longrightarrow \langle l_2, l_{\text{clr}} \mid \mathsf{return}\ t_2\rangle}$$

ToLabeled

$$\frac{l_{\text{cur}} \sqsubseteq l_2 \qquad l_2 \sqsubseteq l_{\text{clr}}}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{toLabeled}\ l_2\ t\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid {}^{l_{\text{cur}}}_{l_{\text{clr}}}\{^{l_2}\ t\ \}\rangle}$$

Reset

$$\frac{l_{\text{cur}} \sqsubseteq l_2}{\langle l_{\text{cur}}, l_{\text{clr}} \mid {}^{l_1}_{l_3}\{^{l_2}\ t^{\mathsf{LIO}}\ \}\rangle \longrightarrow \langle l_1, l_3 \mid \mathsf{label}\ l_2\ t\rangle}$$

Figure 3.8: LIO language semantics (selected rules).

Label Format LIO is parametric in the label format, but for the purposes of this disseratation, we use DC labels [76] extended with a third component to model availability policies. Availability policies also form a lattice, defined in Figure 3.9. A label $\langle l_c, l_i, l_a\rangle$ represents a policy with confidentiality l_c, integrity l_i, and availability l_a. Information labeled with $\langle l_c, l_i, l_a\rangle$ can be read by l_c, is vouched for by l_i (as described in the previous section), and is hosted and made available by l_a. We write $\mathbb{A}(l)$ for the projections of the availability component in l. Each component is a conjunction of disjunctions of principal names, i.e., a formula in conjunctive normal form. In terms of availability, a disjunction $\mathcal{A} \vee B$ means that one of $\mathcal{A}$ or B can deny access to the data. Conjunction $\mathcal{A} \wedge B$ means that $\mathcal{A}$ and B can jointly deny access to the data together for availability.

$$\begin{aligned} \mathcal{A}_1 \sqsubseteq^{\mathcal{A}} \mathcal{A}_2 &\iff \mathcal{A}_1 \Rightarrow \mathcal{A}_2 \\ \mathcal{A}_1 \sqcup^{\mathcal{A}} \mathcal{A}_2 &\iff \mathcal{A}_1 \vee \mathcal{A}_2 \\ \mathcal{A}_1 \sqcap^{\mathcal{A}} \mathcal{A}_2 &\iff \mathcal{A}_1 \wedge \mathcal{A}_2 \\ \bot^{\mathcal{A}} \equiv \mathit{False} &\qquad \top^{\mathcal{A}} \equiv \mathit{True} \end{aligned}$$

Figure 3.9: Availability lattice

Data may flow between differently labeled entities, but only those with more *restrictive* policies: those readable, vouched for, or hosted by fewer entities. A label $\langle l_c, l_i, l_a\rangle$ can flow to any label where the confidentiality component is at least as sensitive than l_c, the integrity component is at

least as untrustworthy as l_i, and the availability is no more available than l_a, i.e. $l_1 \sqsubseteq l_2$ if and only if $\mathbb{C}(l_2) \implies \mathbb{C}(l_1), \mathbb{I}(l_1) \implies \mathbb{I}(l_2)$, and $\mathbb{A}(l_1) \implies \mathbb{A}(l_2)$. We use logical implication because it matches the intuitive meaning of disjunctions and conjunctions, e.g., data readable by $A \vee B$ is less confidential than data readable only by A, and data vouched for by $A \wedge B$ is more trustworthy than data vouched for only by A.

In Chapter 4, we consider computations that work on labels that are pairs of confidentiality and integrity labels, whereas in Chapter 5, we consider labels that are triples of of confidentiality, integrity, and availability.

4

Restricted Privileges for Downgrading

4.1 Introduction

Information-flow control (IFC) systems track the flow of information by associating *labels* with data. Disjunction Category Labels (DC labels) are a practical and expressive label format that can capture the security concerns of principals. IFC systems and DC labels can provide strong, expressive, and practical information security guarantees, preventing exploitation of, for example, cross-site scripting and code injection vulnerabilities [30, 41, 69, 79, 90].

IFC systems often need to *downgrade* information: *declassification* downgrades confidentiality, and *endorsement* downgrades integrity. Downgrading of DC labels occurs via operations that require unforgeable capability-like tokens known as *privileges*. Unfortunately, DC labels offer no methodology to protect developers from the *discretionary* (i.e., unrestricted) exercise of privileges—even a minor mistake in handling privileges can compromise the whole system's security. For example, we found a one-line vulnerability in an existing DC label application written by experts that enabled confidential information to be inappropriately released, thus violating the application's intended security properties.

To address this, we introduce *restricted privileges*: privileges that are limited in their ability to declassify and endorse information. By declaratively restricting the use of privileges, developers can reason about the security properties of the system, regardless of the code that may possess or use the restricted privileges. Thus, the developer's local declaration of restrictions enables the enforcement of global information security guarantees.

We present two kinds of restricted privileges: *bounded privileges* and *robust privileges*. A bounded privilege imposes upper and lower bounds on the DC labels of data that is declassified or endorsed using that privilege. Robust privileges avoid the accidental or malicious use of privileges to declassify or endorse more information than intended, achieving a property known as *robustness* [60, 88].

Bounded Privileges. A bounded privilege wraps an unrestricted privilege with two *immutable* labels that indicate upper and lower bounds for downgrading. DC labels form a lattice structure (described in Section 3.1), and thus a bounded privilege restricts where in the lattice downgrading may occur. A bounded privilege also has a *mode*, indicating whether the bounded privilege may be used for declassification, endorsement, or both declassification and endorsement.

In terms of confidentiality, the upper bound limits the confidentiality of information that can be declassified using the privilege, and the lower bound limits the confidentiality of the information after declassification. For example, suppose principal `fb.com` passes a bounded privilege to `gogl.com`. If the lower bound of the bounded privilege is the label "`gogl.com`" then the privilege can be used to declassify information only from `fb.com` to `gogl.com`. Even if `gogl.com` passes the bounded privilege to another domain, say `evil.com`, the bounded privilege cannot be used to declassify information from `fb.com` to `evil.com`.

In terms of integrity, the upper bound of a bounded privilege indicates the least trustworthy level of information the privilege can be used to endorse, and the lower bound limits the integrity of the information after endorsement. For example, by setting the upper bound appropriately, `fb.com` can create a bounded privilege that can be used to endorse data only from `gogl.com`, and cannot be used to endorse other data, say from `evil.com`.

Robust Privileges. The security of a system might be at risk if an attacker is able to influence the decision to declassify or endorse information, or can influence what information is declassified. For example, consider a routine that receives a secret pair `(username,password)` and uses a privilege to declassify the first component of the pair. If an attacker (from another system component) can influence the pair to be `(password,username)` and trigger the declassification, the password will be leaked.

Robust declassification [88] and *qualified robustness* [60] are end-to-end semantic security guarantees that ensure that attackers are unable to inappropriately influence what information is revealed to them. These security conditions can be enforced by restricting declassification and endorsement operations. A robust privilege wraps a privilege and ensures that it is used only in declassification and endorsement operations that satisfy appropriate robustness checks.

This chapter makes the following contributions:

(i) We introduce bounded and robust privileges to limit the exercise of privileges for declassification and endorsement. (ii) We present a semantic characterization of how bounded privileges and robust privileges restrict declassification and endorsement operations. (iii) We define run-time security checks for bounded privileges and robust privileges that soundly and completely enforce the semantic characterization of restricted downgrading operations. The run-time checking for robust downgrading is effectively a weakening of the underlying unrestricted privilege: a surprisingly simple characterization of robustness. (iv) We illustrate the applicability of bounded and robust privileges via a case study. Moreover, use of restricted privileges identified a vulnerability in an existing DC label-based application.

This chapter is organized as follows. Section 4.2 characterizes downgrading operations that use restricted privileges, and Section 4.3 provides the corresponding enforcement. Section 4.4 describes security properties in the presence of multiple restricted privileges. Case studies are given in Section 4.5. Section 4.6 examines related work and Section 4.7 concludes.

4.2 Security Definitions

If a system contains raw privilege $p^\sharp$, then downgrading of data with policies involving p depends entirely on how $p^\sharp$ is used in the system. Reasoning about what downgrading occurs may require reasoning about global properties of the system. Indeed, we found a vulnerability in a Hails example application [30] of a web-based rock-paper-scissors game where use of a raw privilege was localized to one component, but arbitrary data could be passed to this component to be downgraded. This motivates our work to restrict privileges, and enable local reasoning about downgrading that may occur in a system.

A *restricted privilege* is a raw privilege "wrapped" with limitations on its use. These limitations enable sound reasoning about the downgrading that may be performed using the restricted privilege, even if arbitrary code uses the restricted privilege. Thus, local reasoning that ensures $p^\sharp$ is always appropriately restricted provides global guarantees about the downgrading that can occur with respect to policies involving p.

We present two kinds of restricted privileges, *bounded privileges* and *robust privileges*, which provide simple declarative limitations on the use of raw privileges.

Bounded Privileges

A bounded privilege wraps a raw privilege with *downgrading bounds* and a downgrading mode. A downgrading bound is a pair of security lattice labels L_{high} and L_{low} that provide upper and lower

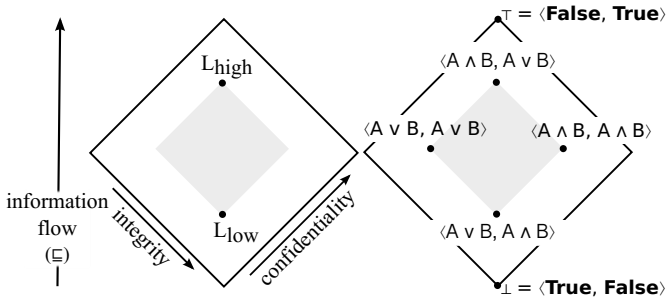

Figure 4.1: Bounded downgrading

bounds on downgrading, and the mode indicates whether the bounded privilege can be used to both declassify and endorse, only to declassify, or only to endorse.

Definition 1 (Downgrading bounds)**.** *An operation that downgrades from security policy L_{from} to security policy L_{to} in a computational context with current label L_{pc} satisfies downgrading bounds L_{high} and L_{low} if and only if $(L_{from} \sqcup L_{pc}) \sqsubseteq L_{high}$ and $L_{low} \sqsubseteq (L_{to} \sqcup L_{pc})$*

Definition 2 (Bounded privileges)**.** *A bounded privilege with bounds L_{high} and L_{low} and mode m on privilege $p^\sharp$, written ${}^{m}[p^\sharp]_{L_{low}}^{L_{high}}$, can be used only for downgrading operations with privilege $p^\sharp$ that satisfy downgrading bounds L_{high} and L_{low}. Mode m is one of* de*,* d*, or* e*. Declassification operations are permitted only if the mode is* de *or* d*; endorsement operations are permitted only if the mode is* de *or* e*.*

Figure 4.1 shows a visualization of bounded downgrading. The security lattice on the left is overlaid with a visualization of where bounded downgrading can occur (shaded) with respect to bounds L_{high} and L_{low}. The security lattice on the right shows an example of what labeled values can be declassified (shaded) with a bounded declassification privilege ${}^{\mathsf{d}}[p^\sharp]_{L_{low}}^{L_{high}}$ on privilege $p^\sharp$ with bounds $L_{high} = \langle \mathsf{A} \wedge \mathsf{B}, \mathsf{A} \vee \mathsf{B} \rangle$ and $L_{low} = \langle \mathsf{A} \vee \mathsf{B}, \mathsf{A} \wedge \mathsf{B} \rangle$.

Information typically only flows according to the safe information flow relation $\sqsubseteq$. However, downgrading introduces new flows. Bounded privileges, however, limit the number of new flows introduced. As a result, in essence, the confidentiality lattice has collapsed $\mathbb{C}(L_{high})$ and $\mathbb{C}(L_{low})$ and all points in between: information that has confidentiality up to $\mathbb{C}(L_{high})$ may be declassified to confidentiality $\mathbb{C}(L_{low})$—all other points in the confidentiality lattice are not affected. Guarantees for endorsement with respect to bounded privileges are similar, but for integrity instead of confidentiality.

Policy: Only Bob controls Alice's privilege: Principal Alice allows Bob to declassify her data provided that Bob vouches for the data and the decision to declassify. In other words,

information labeled with Alice can be declassified only after endorsement by Bob. This property can be captured by a bounded privilege with mode d and bounds: $L_{high} = \langle \top^{\mathsf{C}}, \mathsf{Bob} \rangle$, $L_{low} = \langle \bot^{\mathsf{C}}, \mathsf{Bob} \rangle$. If the privilege is used to declassify information that is not endorsed by Bob or in a context where the current label is not endorsed by Bob, then the declassification fails. In general, data must be vouched for by Bob (e.g., by using $\mathsf{Bob}^\sharp$ or another restricted privilege) before the bounded privilege for Alice can be used. For example, if a computational task has a current label $L_{pc} = \langle \mathsf{Alice}, \mathsf{Bob} \vee \mathsf{Charlie} \rangle$, the current label must be endorsed by Bob first. By endorsing the current label, Bob effectively vouches for any influence Charlie may have had on the computational task.

Policy: "An anonymous source said...": The bounded privilege $^{\mathsf{d}}[\mathsf{Alice}^\sharp]^{\langle \top^{\mathsf{C}}, \top^{\mathsf{I}} \rangle}_{\langle \bot^{\mathsf{C}}, \top^{\mathsf{I}} \rangle}$ requires that the integrity of data being declassified is $\top^{\mathsf{I}}$, i.e., data that no principal takes responsibility for. Alice may wish to impose this restriction on declassification involving data confidential to her to ensure that she has plausible deniability regarding the source of the data released. That is, the bounded privilege can not be used to declassify data for which Alice is explicitly responsible.

Robust Privileges

Robustness [60, 88] is a semantic security condition that limits downgrading based on which principals might benefit from the downgrading, and which principals have influenced the data to downgrade and the decision to downgrade.

Consider a declassification of information from a source protected by label L_{from} to a sink protected by label L_{to}. A formula $\mathcal{A}$ (representing a principal or party of principals) will benefit from the declassification if $\mathcal{A}$ cannot read from the source, but can read the sink, i.e.,

$$\mathbb{C}(L_{from}) \not\sqsubseteq^{\mathsf{C}} \mathcal{A} \text{ and } \mathbb{C}(L_{to}) \sqsubseteq^{\mathsf{C}} \mathcal{A}$$

A robust declassification does not permit any principal that benefits from it to influence either the decision to declassify or the data to declassify. $\mathcal{A}$ influences the decision to declassify if $\mathcal{A} \sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{pc})$, and $\mathcal{A}$ influences the data to declassify if $\mathcal{A} \sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{from})$.

Definition 3 (Robust declassification)**.** *A robust declassification using privilege $p^\sharp$ from a source protected by L_{from} to a sink protected by L_{to}, in a computational context with current label L_{pc} is a declassification (i.e., $\mathbb{C}(L_{from}) \sqsubseteq^{\mathsf{C}}_p \mathbb{C}(L_{to})$) where $\forall \mathcal{A} \in CNF. \mathbb{C}(L_{to}) \sqsubseteq^{\mathsf{C}} \mathcal{A} \wedge \mathbb{C}(L_{from}) \not\sqsubseteq^{\mathsf{C}} \mathcal{A} \Rightarrow \mathcal{A} \not\sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{pc}) \wedge \mathcal{A} \not\sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{from})$.*

For endorsement, a principal benefits if it may be held responsible for information from the

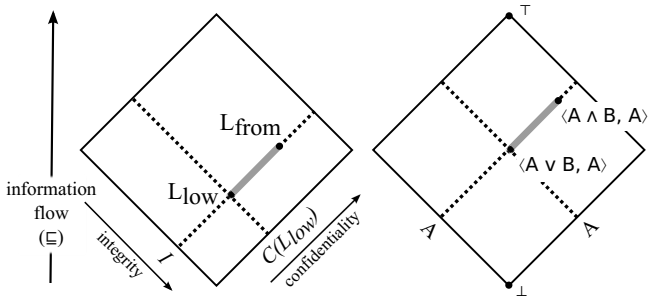

Figure 4.2: Robust declassification

source but is not held responsible for information from the sink. In other words, A benefits from an endorsement if A gets absolved of responsibility for a value, i.e., $A \sqsubseteq^{\mathbb{I}} \mathbb{I}(L_{from}) \wedge A \not\sqsubseteq^{\mathbb{I}} \mathbb{I}(L_{to})$. Robust endorsement does not permit principals that benefit from it to influence the decision to endorse.

Definition 4 (Robust endorsement)**.** *A robust endorsement using privilege $p^\sharp$ from a source protected by L_{from} to a sink protected by L_{to}, in a computational context with current label L_{pc} is an endorsement (i.e., $\mathbb{I}(L_{from}) \sqsubseteq^{\mathbb{I}}_{p} \mathbb{I}(L_{to})$) where $\forall A \in CNF. A \sqsubseteq^{\mathbb{I}} \mathbb{I}(L_{from}) \wedge A \not\sqsubseteq^{\mathbb{I}} \mathbb{I}(L_{to}) \Rightarrow A \not\sqsubseteq^{\mathbb{I}} \mathbb{I}(L_{pc})$.*

A *robust privilege* is a privilege that can only be used for robust downgrading operations.

Definition 5 (Robust privilege)**.** *A robust privilege with mode m on privilege $p^\sharp$, written $rbst^m\{p^\sharp\}$, restricts downgrading operations where it is used to those that are robust for $p^\sharp$. Mode m is one of* de*,* d*, or* e*. Declassification operations are permitted only if the mode is* de *or* d*; endorsement operations are permitted only if the mode is* de *or* e*.*

The definitions of robust declassification and endorsements both quantify over all formulas A in the (possibly infinite) set CNF. In Section 4.3, we consider how to implement efficient checks that do not use universal quantification.

Figure 4.2 shows a visualization of where robust declassification is allowed for a given robust privilege. The security lattice on the left is overlaid with a visualization of where a value with label L_{from} can be declassified to (shaded line) using a robust declassification privilege. (Note that the current label L_{pc} is not included in the diagram for brevity.) The dashed line at I represents the boolean formula for the integrity of the labeled value, that covers points L_{low} and L_{from}. L_{low} is one of the lowest points where L_{from} can be declassified to while still being a robust declassification, i.e., $L_{low} \sqsubseteq L_{to}$.

Note that with robust declassification, the integrity is re-interpreted in terms of confidentiality. That is, the integrity of the label of the value for declassification (together with the integrity of the current label of the process) is used as a lower bound for declassification. Intuitively, those who influence a declassification should not learn from it.

In the right hand side of Figure 4.2, the shaded line indicates to where a robust privilege may declassify the labeled value $\langle \mathsf{A} \wedge \mathsf{B}, \mathsf{A} \rangle$. The declassification is robust if A is not able to learn from the declassification. As a result, the value could not be declassified to $\langle \mathsf{A} \vee \mathsf{B}, \mathsf{A} \rangle$ as A would learn from a declassification that it influenced. In contrast, it is robust to declassify it to $\langle \mathsf{B}, \mathsf{A} \rangle$.

4.3 Enforcement for robust privileges

In this section we describe enforcement mechanisms for restricted privileges that satisfy their semantic characterizations described in Section 4.2. We have implemented these mechanisms in LIO and use them in our case study (see Section 4.5).

When a bounded privilege (Definition 2) is used at run time, it is simple to check that the downgrading operation satisfies the appropriate bounds, since the labels relevant to the downgrading (L_{from}, L_{to}, and L_{pc}) are all available at run time, and the label ordering relation can be easily checked dynamically.

Robust privileges (Definition 5) impose restrictions on downgrading operations which quantify over formulas $\mathcal{A}$. However, attempting to explicitly check each possible formula $\mathcal{A}$ at run time is not feasible. We can however, derive simple and efficient run-time checks that are sound and complete with respect to their semantic characterizations. These checks are inspired by Chong and Meyers [18], who provide run-time checks for robustness that are sound but not complete.

The following theorem shows that the semantic characterization of robust declassification (Definition 3) is equivalent to two confidentiality-policy comparisons involving only L_{from}, L_{to}, and L_{pc}.

Theorem 6 (Robust declassification check)**.** *A declassification using privilege $p^\sharp$ from a source protected by L_{from} to a sink protected by L_{to} in a computational context with current label L_{pc} is robust if and only if $\mathbb{C}(L_{from}) \sqsubseteq^{\mathsf{c}}_{p} \mathbb{C}(L_{to})$, $\mathbb{C}(L_{from}) \sqsubseteq^{\mathsf{c}} \mathbb{C}(L_{to}) \sqcup^{\mathsf{c}} \mathbb{I}(L_{pc})$, and $\mathbb{C}(L_{from}) \sqsubseteq^{\mathsf{c}} \mathbb{C}(L_{to}) \sqcup^{\mathsf{c}} \mathbb{I}(L_{from})$.*

The run-time check ensures that if there is any formula $\mathcal{A}$ that benefits from the declassification ($\mathbb{C}(L_{from}) \not\sqsubseteq^{\mathsf{c}} \mathcal{A}$ and $\mathbb{C}(L_{to}) \sqsubseteq^{\mathsf{c}} \mathcal{A}$) then $\mathcal{A} \not\sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{pc})$ (or, equivalently, $\mathbb{I}(L_{pc}) \not\sqsubseteq^{\mathsf{c}} \mathcal{A}$), and similarly that $\mathcal{A} \not\sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{from})$. Thus, the run-time check converts a comparison of integrity policies to a comparison of integrity policies that does not involve $\mathcal{A}$. Further, note that this check is *complete*: the check will pass if and only if the declassification is robust.

The next theorem describes a simple run-time check for robust endorsement.[1]

Theorem 7 (Robust endorsement check)**.** *An endorsement using privilege $p^\sharp$ from a source protected by L_{from} to a sink protected by L_{to} in a computational context with current label L_{pc} is robust (Definition 4) if and only if $\mathbb{I}(L_{from}) \sqsubseteq^{\mathrm{I}}_{p} \mathbb{I}(L_{to})$, and $\mathbb{I}(L_{pc}) \sqcap^{\mathrm{I}} \mathbb{I}(L_{from}) \sqsubseteq^{\mathrm{I}} \mathbb{I}(L_{to})$.*

The run-time check that all formulas $\mathcal{A}$ that may be responsible for either the current label ($\mathcal{A} \sqsubseteq^{\mathrm{I}} \mathbb{I}(L_{pc})$) or the data itself ($\mathcal{A} \sqsubseteq^{\mathrm{I}} \mathbb{I}(L_{from})$) should also be responsible for the data after endorsement ($\mathcal{A} \sqsubseteq^{\mathrm{I}} \mathbb{I}(L_{to})$).

Alternative formulation

In DC labels, privileges can be arbitrary formulas, which can be stronger or weaker than privileges for individual principals. For example, a privilege for $\mathsf{A} \wedge \mathsf{B}$ can downgrade more information than a privilege for A or B alone, whereas a privilege for $\mathsf{A} \vee \mathsf{B}$ can downgrade less information than a privilege for A or B alone. Leveraging this feature, we show how robust downgrading can be seen (and enforced) as normal downgrading operations that use a weakened privilege. That is, the privilege used in a downgrading operation is weakened so as to permit all and only robust downgrading operations.

The next corollaries follow from Theorems 6 and 7 and the definition for the *can-flow-to-with-privilege-p* relation.[2]

Corollary 8. *A declassification using raw privilege $p^\sharp$ from a source protected by L_{from} to a sink protected by L_{to} in a computational context with current label L_{pc} is robust (Definition 3) if and only if $\mathbb{C}(L_{from}) \sqsubseteq^{\mathrm{C}}_{p \vee \mathbb{I}(L_{from}) \vee \mathbb{I}(L_{pc})} \mathbb{C}(L_{to})$.*

This indicates that robust declassification can be achieved by simply weakening privilege $p^\sharp$ with the integrity labels of the current label and the data to be released, i.e., $p \vee \mathbb{I}(L_{from}) \vee \mathbb{I}(L_{pc})$. Robust endorsement has a similar corollary.

Corollary 9. *An endorsement using raw privilege $p^\sharp$ from a source protected by L_{from} to a sink protected by L_{to} in a computational context with current label L_{pc} is robust (Definition 3) if and only if $\mathbb{I}(L_{from}) \sqsubseteq^{\mathrm{I}}_{p \vee \mathbb{I}(L_{pc})} \mathbb{I}(L_{to})$.*

The current implementation of DC labels [76] provides the ability to infer appropriate L_{to} labels of downgrading operations given a privilege p. By expressing the runtime checks for robust downgrading operations as a standard downgrading operation with a weakened privilege, we can take

[1] Proofs of Theorems 6 and 7 are in Appendix B.

[2] The proof of Corollary 8 is in Appendix B; the proof of Corollary 9 is similar.

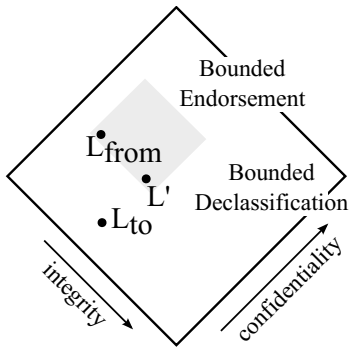

Figure 4.3: Multiple bounds.

advantage of this feature and automatically infer a suitable L_{to} label if one exists. This reduces the burden on the programmer.

4.4 Interaction among restricted privileges

We can extend restricted privileges to allow them to be composed, i.e., by allowing bounded privileges and robust privileges to wrap around other restricted privileges, as well as raw privileges. The guarantee provided by the composition of restricted privileges is the intersection of their individual guarantees. For example, a bounded privilege composed with another bounded privilege will require that downgrading operations satisfy the bounds of both privileges. A bounded privilege composed with a robust privilege (and vice-versa) requires the downgrading both to be robust and satisfy the downgrading bounds. Robust privileges are idempotent: a robust privilege composed with a robust privilege will simply require all downgrade operations to be robust.

Privileges might also interact because a system has multiple privileges available. Unlike composed privileges (which further restrict possible information flows), multiple privileges enable additional information flows. In the remainder of the section, we discuss the guarantees that result from the use of multiple restricted privileges. In the accompanying figures, bounded privileges are depicted as a shaded rectangle corresponding to their bounds. Robust declassification privileges are depicted as a pair of dashed lines: one line represents the integrity of the source and the other line represents the lower bound to which data may be declassified. Labels are depicted as points along with their names.

Bounded declassification and bounded endorsement Figure 4.3 depicts two bounded privileges, one for declassification and one for endorsement, as well as a label, L_{from} that is outside

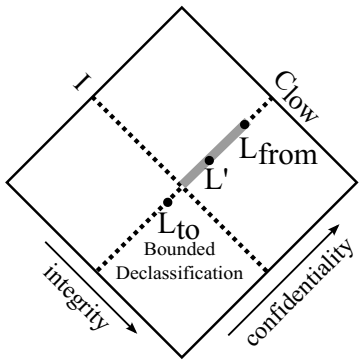

Figure 4.4: Bounded and robust declassification.

the bounds of the declassification privilege. Because the bounds of the privileges overlap, data can transitively flow from L_{from} to L_{to}. The endorsement privilege enables data from L_{from} to be endorsed to L'. The bounded declassification privilege can then declassify data from L' to L_{to}.

Bounded declassification and robust declassification Figure 4.4 depicts two declassification privileges, one robust and one bounded, and a label that is outside the bounds of the bounded declassification privilege. Neither privilege alone permits a flow from L_{from} to L_{to}. However, when used together, the robust declassification privilege permits declassification of data from L_{from} to L' and the bounded declassification permits a flow from L' to L_{to}, completing a flow from L_{from} to L_{to}.

Endorsement and robust declassification In a system with unrestricted endorsement, robust declassification provides almost no protection against attackers influencing what they learn. Intuitively, the endorsement of data by p can make the data trustworthy enough to make a subsequent declassification robust. Consider a declassification of a value from label $L_{from} = \langle \mathsf{A} \wedge \mathsf{B}, \mathsf{A} \rangle$ to $L = \langle \mathsf{A}, \mathsf{A} \rangle$ using the robust privilege $rbst^{\mathsf{d}}\{\mathsf{B}^{\sharp}\}$. This declassification is not robust: principal A, who benefits from this declassification, may be held responsible for the value, i.e., A may have decided what gets declassified. However, an unrestricted endorsement privilege $\mathsf{B}^{\sharp}$ could be used to endorse the value—effectively endorsing any possible influence by A. In other words, $\langle \mathsf{A} \wedge \mathsf{B}, \mathsf{A} \rangle$ can be endorsed to $\langle \mathsf{A} \wedge \mathsf{B}, \mathsf{B} \rangle$, and a subsequent declassification from $\langle \mathsf{A} \wedge \mathsf{B}, \mathsf{B} \rangle$ to $\langle \mathsf{A}, \mathsf{B} \rangle$ is robust.

Bounded endorsement effectively limits the aforementioned deletrious effects of unrestricted endorsement to the bounded area of the lattice, Figure 4.5 depicts this situation. Besides mitigating the effects of unrestricted endorsement, bounded endorsement is useful to relax robust declassifica-

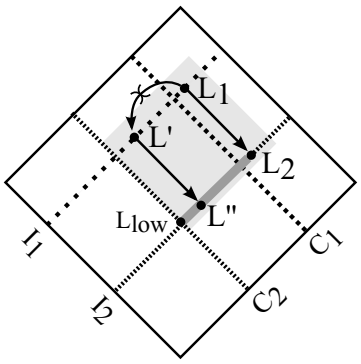

Figure 4.5: Bounded endorsement and robust declassification.

tion so that it succeeds for principals collaborating in achieving a common goal—see, for example, Section 4.5.

Bounded and robust declassification Figure 4.4 shows the guarantees when a robust declassification-only privilege (i.e., $rbst^{\mathsf{d}}\{p^{\sharp}\}$) and a bounded declassification-only privilege (i.e., ${}^{\mathsf{d}}[p^{\sharp}]_{L_{low}}^{L_{high}}$) for the same principal are both available in the system. Intuitively, p's information can be declassified from L_{from} to L' using the robust privilege. The information can then be declassified again to L_{to} using the bounded privilege, even though L_{from} is *above* the threshold imposed by robust declassification (i.e., the lowest possible label that robust declassification could declassify label L_{from}). Thus, the presence of a bounded declassification-only privilege can bypass (some of) the guarantees provided by robust declassification.

Several bounded privileges Multiple robust privileges for the same principal do not add any additional complexity, as all robust privileges are equivalent (up to their modes). Bounded privileges, however, may differ on the bounds they impose. The presence of multiple bounded privileges in a system for principal p collapses the label lattice for principal p in complex ways. For instance, the diagram of Figure 4.3 illustrates an example where there is a bounded endorsement-only privilege and a bounded declassification-only privilege with different bounds. It may be possible for a value labeled L_{from} to be relabeled to L_{to} via an endorsement to L' followed by a declassification. Thus, labels between L_{from} and L' and between L' and L_{to} are effectively collapsed, since the bounded privileges allow a value with any of these labels to be relabeled to any other of these labels. More generally, as more overlapping bounded privileges exist for a given principal, data can be downgraded in more possible ways.

4.5 Case studies

In this section, we explore the security guarantees provided by restricted privileges through a case study program we developed.

4.5.1 Calendar Case Study

We have extended LIO [77] with support for bounded privileges and robust privileges, and used them to develop a Calendar application to explore and illustrate the utility of restricted privileges. The application allows users to view their appointments, and schedule appointments with each other. DC label principals are the calendar users. A user's appointments are confidential to that user.

We consider a setting where principals belong to groups and a principal is willing to disclose her availability to all and only members of her groups. For example, if Bob wants to schedule an appointment with Alice at time t, the application will check Alice's calendar and inform Bob whether Alice is available at that time. This operation, which declassifies Alice's availability at time t to Bob, should succeed only if Alice and Bob are in the same group.

Each user A has a robust declassification privilege $\mathit{rbst}^{\mathsf{d}}\{\mathsf{A}^{\sharp}\}$, and, for each group G that A belongs to, a bounded endorsement privilege ${}^{\mathsf{e}}[\mathsf{A}^{\sharp}]_{\langle \bot^{\mathsf{c}}, \bot^{\mathsf{I}} \rangle}^{\langle \top^{\mathsf{c}}, G \rangle}$, where G is the disjunction of all users in the group. These are the only privileges available in the system for user A, and thus all endorsements must be bounded appropriately, and all declassifications must be robust.

Joint scheduling between A and B works as follows:

1. User B sends a scheduling request for time t labeled $\langle \mathsf{B}, \mathsf{B} \rangle$ to user A.
2. User A computes her availability for time t. Because the context that computes the availability reads data labeled $\langle \mathsf{A}, \mathsf{A} \rangle$ and $\langle \mathsf{B}, \mathsf{A} \rangle$, the label of the availability result is $\langle \mathsf{A} \wedge \mathsf{B}, \mathsf{A} \vee \mathsf{B} \rangle$.
3. If A and B are both in some group G, then A uses her bounded privilege to endorse the availability result to $\langle \mathsf{A} \wedge \mathsf{B}, \mathsf{A} \rangle$, since she is prepared to take sole responsibility for the availability result. Since both A and B are in the same group, the endorsement satisfies the bounds (i.e., $\mathsf{A} \vee \mathsf{B} \sqsubseteq^{\mathsf{I}} G$). If there is no group for which both A and B are members, then A has no bounded endorsement privilege for which the bounds will be satisfied.
4. User A uses her robust privilege to declassify the availability result to $\langle \mathsf{B}, \mathsf{A} \rangle$. The declassification is robust.
5. User A sends the declassified value to B.

Because all downgrading in the system relevant to user A must use A's restricted privileges, we obtain strong system-wide guarantees, even if A's restricted privileges manage to escape from the scheduling component, and even if if B sends malicious scheduling requests. Section 4.4 (Figure 4.5)

discusses in more detail the system-wide guarantees that hold when both a bounded endorsement privilege and a robust declassification privilege are available.

4.5.2 Restricted Privileges in Existing Applications

Using our restricted privileges, we found a security vulnerability in an application written using Haskell Automatic Information Labeling System (Hails) [30]. Hails is a web framework built on LIO that extends the traditional Model-View-Controller paradigm to Model-Policy-View-Controller. The policy module specifies all models and describes the labels for data fetched from the database. When data is stored in the database, Hails checks labels against the policy module to ensure appropriate data integrity. The policy module has access to a privilege that can declassify all models. As a design pattern, policy modules export functions that perform declassification for untrusted applications using the privilege; untrusted applications never have direct access to the privilege.

Rock-Paper-Scissors[1] is a Hails application that contains a security vulnerability due to misuse of the policy privilege, despite being written by security experts who developed Hails.

The policy module includes a function to get the outcome of a match given a particular move by a player. This function can be exploited to reveal the opponent's move before the player has actually committed to a move by submitting it to the database. As a result, a player can always win a match by exploiting this function to determine which move will win, and then committing to that winning move. When we replaced the policy module's raw privilege with a robust privilege, the robust declassification check signalled a potential security vulnerability. To fix the vulnerability, we added code that checks whether a player had committed to a move (i.e., the move is in the database), and, if so, endorses the submitted move. This endorsement allows the robust declassification check to succeed. Endorsing only when the player has committed to his move fixes the security vulnerability.

4.6 Related Work

Declassification can be characterized into different dimensions: *who*, *what*, *where*, and *when* [70]. Our work can be considering as restricting *where* in the security lattice downgrading may occur (bounded downgrading) and *who* may influence downgrading (robustness). Almeida Matos and Boudol [2] introduce a construct **flow** $p \prec q$ **in** c to indicate *where* additional information flows are allowed within a lexical scope. Intransitive noninterference [50, 66, 80] posits a non-transitive information flow ordering which describes *what* downgrading operations are permitted. Mantel and Sands [50] combine intransitive noninterference with language techniques that use declassification

[1] https://github.com/scslab/hails/tree/master/examples/hails-rock

annotations to explicitly identify non-transitive information flows. In our bounded declassification mechanism, violation of the normal ordering of security levels is tied to a runtime value, and not lexically scoped or marked by annotations.

In Jif [57], declassifications may explicitly state where in the security lattice the declassification occurs. By contrast, our bounded mechanisms declare this restriction on the run-time value that authorizes downgrading. Jif uses a form of access control to restrict which code may downgrade information, coined *selective declassification* by Pottier and Conchon [65]. Specifically, a downgrading operation that may compromise the security of principal p may only occur in code that has been (statically or dynamically) authorized by p. Similarly, the authority to declassify or endorse information in Asbestos [25], HiStar [90], Flume [41], and COWL [79] must come from the creator of the exercised privileges. By contrast, LIO associates the authority to declassify or endorse a principal's information with a run-time value. This capability-like approach to authorizing downgrading enables our local declarative approach to restrict downgrading. Birgisson et al. [12] use capabilities to restrict the ability to read and write memory locations, but do not consider the use of capabilities to restrict downgrading.

Zdancewic and Myers [88] introduce the semantic security condition of *robust declassification*, and Myers et al. [60] enforce robust declassification with a security type system [69, 82], and introduce *qualified robustness*, which extends the concept to reason about endorsement. Askarov and Myers [5] subsequently present a semantic framework for downgrading, and present a crisper version of qualified robustness. Chong and Myers [18] extend the notion of robust declassification to the Decentralized Label Model [58, 59]. The run-time checks used in this work to enforce robustness are analogous to the run-time checks Chong and Myers introduce for the DLM. In other work, Chong and Myers [17] note that the semantic security condition for robust declassification applies to information flow of confidential information generally, including, for example, information erasure, and is more general than just declassification. If the only privilege for p available in the system is a robust privilege with mode mode d then the system will be robust for p. If the privilege for that mode is de (i.e., robust declassification operations and robust endorsement operations are possible), then the end-to-end security guarantee is *qualified robustness* [5, 60]. A system satisfies qualified robustness if the only way an attacker can influence what information is released to it is via robust endorsement operations.

Foley et al. incorporate bounds constraints on a system with relabeling operations on objects [27]. Our model performs relabeling based on the use of capability-like tokens rather than with respect to a particular subject. Bound restrictions can be placed per privilege rather than on all relabeling operations, so the guarantees of this work are more dependent on what sorts of privileges are available for

use, but do not require changes to the trusted computing base.

The system HiStar [90] provides the notion of gates: entities designed to encapsulate privileges so that processes can safely switch their current label by exercising them through the gate. Gates have a clearance component which imposes an upper bound on the label that results from using it. Gates can be leveraged to restrict the use of privileges similar to upper bounds in bounded privileges. Similar to our approach, Flume[41] distinguishes privileges used for declassification (symbol $-$) and endorsement (symbol $+$).

4.7 Conclusion

Restricted privileges are a new mechanism to control declassification and endorsement in DC labels that is simple and intuitive yet expresses a rich set of desirable policies. Bounded privileges impose upper and lower bounds on data that is declassified or endorsed. Robust privileges help prevent the accidental or malicious exercise of privileges to downgrade more information than intended, and can provide the end-to-end security guarantees of robustness and qualified robustness. We provide sound and complete efficient security checks for downgrading using restricted privileges. We note that robust downgrading operations can be viewed as privileged downgrading with a weakened privilege. We explore the guarantees provided by combining the use of bounded and robust privileges as well as their composition in a case study. This work establishes a basis for better design of IFC systems that use privileges for downgrading information.

5

Cryptographically Secure Information-Flow Control for Key-Value Stores

5.1 Introduction

Cryptography is critical for applications that securely store and transmit data. It enables the authentication of remote hosts, authorization of privileged operations, and the preservation of confidentiality and integrity of data. However, applying cryptography is a subtle task, often involving setting up configuration options and low-level details that users must get right; even small mistakes can lead to major vulnerabilities [55, 71]. A common approach to address this problem is to raise the level of abstraction. For example, many libraries provide high-level interfaces for establishing TLS [22] network connections (e.g., OpenSSL[1]) that are very similar to the interfaces for establishing unencrypted connections. These libraries are useful (and popular) because they abstract many configuration details, but they also make several assumptions about certificate authorities, valid protocols, and client authentication. Due in part to these assumptions, the interfaces are designed for experienced cryptography programmers and as a result can be used incorrectly by non-experts in spite of their high level of abstraction [85]. Indeed, crypto library misuse is a more prevelant security issue than Cross-Site Scripting (XSS) and SQL Injection [1].

Information flow control (IFC) is an attractive approach to building secure applications because it addresses some of these issues. There has been extensive work in developing expressive informa-

[1] https://www.openssl.org/

tion flow policy languages [4, 59, 76] that help clarify a programmer's intent. Furthermore, many semantic guarantees offered by IFC languages are inherently compositional from a security point of view [31, 88]. However, existing IFC languages (e.g., [21, 35, 57, 68, 77, 79, 87]) generally assume that critical components of the system, such as persistent storage, are trustworthy—the components must enforce the policies specified by the language abstraction. This assumption makes most current IFC systems a poor fit for many of the use-cases that cryptographic mechanisms are designed for.

To remedy this issue, it is tempting to extend IFC guarantees to work with untrustworthy data storage by simply "plugging in" cryptography. However, the task is not simple: the threat model of an IFC system extended with cryptography differs from both the standard cryptographic threat models and from standard IFC threat models. Unlike most IFC security models, an attacker in this scenario may have low-level abilities to access signatures and ciphertexts of sensitive data, and the ability to deny access to data by corrupting it (e.g., flipping bits in ciphertexts).

Attackers also have indirect access to the private cryptographic keys through the trusted runtime. An attacker may craft and run programs that have access to the system's cryptographic keys in order to trick the system into inappropriately decrypting or signing information. Cryptographic security models often account for the high-level actions of attackers using *oracles* that mediate what information an active attacker can learn through interactions with the cryptosystem. These oracles abstractly represent implementation artifacts that could be used by the attacker to distinguish ciphertexts. Ensuring that an actual implementation constrains its behavior to that modeled by an oracle is typically left to developers.

An attacker's actual interactions with a system often extends beyond the semantics of specific cryptographic primitives and into application-specific runtime behavior such as how a server responds when a message fails to decrypt or a signature cannot be verified. If an attacker can distinguish this behavior, it may provide them with information about secrets. Building real implementations that provide no additional information to attackers beyond that permitted by the security model can be very challenging.

Therefore, to give developers better tools for building secure applications, we need to ensure that the system security is not violated by combining attackers' low-level abilities and their ability to craft their own programs. This requires extending the attacker's power beyond that typically considered by IFC models, and representing the attacker's interactions with the system more precisely than typical cryptographic security models.

This chapter presents Clio, a programming language that reconciles IFC and cryptography models to provide guarantees on both ephemeral data within Clio applications and persistent data on

an untrusted key-value store. Clio extends the IFC-tool LIO [77] with *store* and *fetch* operations for interacting with a persistent key-value store. Like LIO, Clio expresses confidentiality and integrity requirements using *security labels*: flows of information are controlled throughout the execution of programs to ensure the policies represented by the labels are enforced. Clio encrypts and signs data as it leaves the Clio runtime, and decrypts and verifies as it enters the system. These operations are done automatically according to the security labels—thus avoiding both the mishandling of sensitive data and the misuse of cryptographic mechanisms.

Clio transparently maps security labels to cryptographic keys and leverages the underlying IFC mechanisms to ensure that keys are not misused within the program. Since we consider attackers capable of denying access to information by corrupting data, Clio extends LIO labels with an availability policy that tracks who can deny access to information (i.e., who may corrupt the data).

Figure 5.1 presents an overview of the Clio threat model. At a high-level, a Clio program may be a malicious program written by the attacker. Attackers may also perform low-level `fetch` and `store` operations directly on the key-value store. In addition, all interactions between the runtime and the store are visible to the attacker. Only the (trusted) Clio runtime has access to the keys used to protect information from the attacker, but the attacker may have access to other "low" keys. The Clio runtime never exposes keys directly to program code: they are only used implicitly to protect or verify data as it leaves or enters the Clio runtime. Therefore, Clio's information flow control mechanisms mediate the attacker's ability to discover new information or modify signed values by interacting with a Clio program through `fetch`s and `store`s to a Clio store.

This chapter makes the following contributions:

- A formalization of the *ideal* semantics of Clio, which models its security without cryptography, and a *real* semantics, which enforces security cryptographically.
- A novel proof technique that combines standard programming language and cryptographic proof techniques. Using this approach, we have characterized the interaction between the high-level security guarantees provided by information flow control and the low-level guarantees offered by the cryptographic mechanisms. For confidentiality, we have formalized these guarantees as *chosen-term attack* security, an extension of *chosen-plaintext attack* security to systems where an attacker may choose arbitrary programs that encrypt and decrypt information. Similarly, for integrity we have defined *leveraged existential forgery*, an extension of *existential forgery* to systems where an attacker may choose and execute a program to produce signed values.

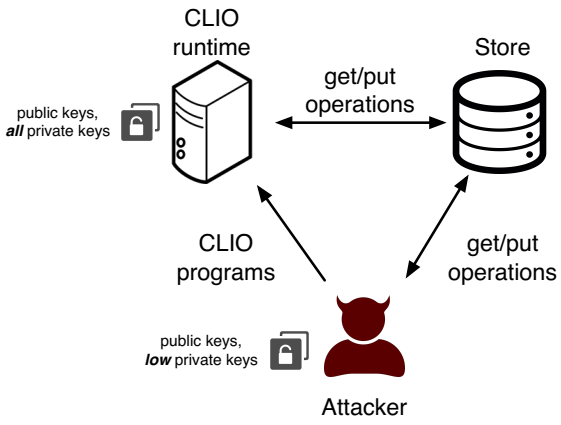

Figure 5.1: Clio threat model. Attackers write Clio programs, read from and write to the store, and observe the runtime's interactions.

- A prototype Clio implementation in the form of a Haskell library extending LIO. Our prototype system employs the DC labels model [76], previously used in practical systems (e.g., Hails [30] and COWL [79]). Our implementation extends DC labels with an availability component, both of which may be applicable to these existing systems as well.

Although our results are specific to Clio, we expect our approach to be useful in proving the security of cryptographic extensions of other information flow languages.

The rest of the chapter is structured as follows. Section 5.2 describes the extensions to it in order to interact with an untrusted store. Section 5.3 describes the computational model of Clio with cryptography, and Section 5.4 shows the model's formal security properties. Section 5.5 describes the prototype implementation of Clio along with a case study. And finally Section 5.6 discusses related work.

5.2 Interacting with an Untrusted Store

Clio extends LIO with a key-value store. The language is extended with two new commands: store t_k t_v puts a labeled value t_v in the store with key t_k; fetch$_\tau$ t_k t_v command fetches the entry with key t_k and if it cannot be fetched, returns the labeled value t_v. In both commands, t_k must evaluate to a ground value and the labeled value t_v must evaluate to a labeled ground value with type τ.

Semantics for fetch and store are shown in Figure 5.2. We modify the semantics to be a labeled transition system, where the step relation $\xrightarrow{\alpha}$ is annotated with *store events* α. A store event α is:

STORE

$$\frac{l_{\text{cur}} \sqsubseteq \ell_{store} \qquad l_{\text{cur}} \sqsubseteq l_1 \qquad \alpha = \mathsf{put}\ \langle \underline{v} : l_1 \rangle\ \mathsf{at}\ \underline{v}_k}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{store}\ \underline{v}_k\ \langle \underline{v} : l_1 \rangle \rangle \xrightarrow{\alpha} \langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{return}\ () \rangle}$$

FETCH-VALID

$$\frac{\begin{array}{c}\mathbb{A}(\ell_{store}) \sqsubseteq^{\mathcal{A}} \mathbb{A}(l_d) \\ \alpha = \mathsf{got}_{\tau}\ \langle \underline{v} : l \rangle\ \mathsf{at}\ \underline{v}_k \qquad l \sqsubseteq l_d\end{array}}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{fetch}_{\tau}\ \underline{v}_k\ \langle \underline{v}_d : l_d \rangle \rangle \xrightarrow{\alpha} \langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{return}\ \langle \underline{v} : l_d \rangle \rangle}$$

FETCH-INVALID

$$\frac{\begin{array}{c}\mathbb{A}(\ell_{store}) \sqsubseteq^{\mathcal{A}} \mathbb{A}(l_d) \\ (\alpha = \mathsf{nothing\text{-}at}\ \underline{v}_k)\ \text{or}\ (\alpha = \mathsf{got}_{\tau}\ \langle \underline{v} : l \rangle\ \mathsf{at}\ \underline{v}_k\ \text{and}\ l \not\sqsubseteq l_d)\end{array}}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{fetch}_{\tau}\ \underline{v}_k\ \langle \underline{v}_d : l_d \rangle \rangle \xrightarrow{\alpha} \langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{return}\ \langle \underline{v}_d : l_d \rangle \rangle}$$

Figure 5.2: Clio language semantics (store and fetch rules).

- skip (representing no interaction with the store, i.e., an internal step; we typically elide skip for clarity),
- put $\langle \underline{v} : l \rangle$ at $\underline{v}_k$ (representing putting a labeled ground value $\langle \underline{v} : l \rangle$ indexed by $\underline{v}_k$),
- got$_{\tau}$ $\langle \underline{v} : l \rangle$ at $\underline{v}_k$ (representing reading a labeled value from the store indexed by $\underline{v}_k$), or
- nothing-at $\underline{v}_k$ representing an attempt to read at index $\underline{v}_k$ for which there is no value.

Labeling transitions with store events allows us to cleanly factor out the implementation of the store, enabling us to easily use either an idealized (non-cryptographic) store, or a store that uses cryptography to help enforce security guarantees. We describe the semantics of store events in both these settings later.

We associate a label ℓ_{store} with the store. Intuitively, store level ℓ_{store} describes how trusted the store is: it represents the inherent protections provided by the store and the trust the store places on Clio that a Clio computation must respect. For example, the store may be behind an organization's firewall so data is accessible only to organization members due to an external access control mechanism (i.e., the firewall), so Clio can safely store the organization's information there. Dually, there may be integrity requirements that Clio is trusted to uphold when writing to the store. For example, the store may be used as part of a larger system that uses the store to perform important operations (e.g., ship customer orders). Thus the integrity component of the store label is a bound on the trustworthiness of information that Clio should write to the store (e.g., Clio should not put unendorsed shipping requests in the store). The availability component of the store label specifies

a bound on who is able to corrupt information in the store (and thus make it unavailable). (Note that we are concerned with *information availability* rather than *system availability*.) In general, this would describe all the principals who have direct and indirect write-access to the store.

Rule STORE (Figure 5.2) is used to put a labeled value $\langle \underline{v} : l \rangle$ in the store, indexed by key $\underline{v_k}$. We require that the current label l_{cur} is bounded above by store level ℓ_{store}. In terms of confidentiality, this means that any information that may be revealed by performing the store operation (i.e., l_{cur}) is permitted to be learned by users of the store. For integrity, the decision to place this value in the store (possibly overwriting a previous value) should not be influenced by information below the integrity requirements of the store. For availability, the information should not have come from less available sources than the store's availability level to ensure the store's availability level accurately reflects who could have corrupted stored information.

Additionally, we require the current label to flow to l_1, the label of the value that is being stored (i.e., $l_{\text{cur}} \sqsubseteq l_1$). Intuitively, this is because an entity that learns the labeled value also learns that the labeled value was put in the store. Current label l_{cur} is an upper bound on the information that lead to the decision to perform the store, and l_1 bounds who may learn the labeled value.

For a command $\mathsf{fetch}_{\tau}\ \underline{v_k}\ \langle \underline{v_d} : l_d \rangle$, the labeled value $\langle \underline{v_d} : l_d \rangle$ serves double duty. First, if the store cannot return a suitable value (e.g., because there is no value indexed by key $\underline{v_k}$, or because cryptographic signature verification fails), then the fetch command evaluates to the default labeled value $\langle \underline{v_d} : l_d \rangle$ (which might be an error value or a suitable default). Second, label l_d specifies an upper bound on the label of any value that may be returned by the fetch command: if the store wants to return a labeled value $\langle \underline{v} : l \rangle$ where $l \not\sqsubseteq l_d$, then the fetch command evaluates to $\langle \underline{v_d} : l_d \rangle$. This allows a programmer to specify a bound on information they are willing to read from the store.

Rule FETCH-VALID is used when a labeled value is successfully fetched from the store. Store event $\mathsf{got}_{\tau}\ \langle \underline{v} : l \rangle\ \mathsf{at}\ \underline{v_k}$ indicates that the store was able to return labeled value $\langle \underline{v} : l \rangle$ indexed by the key $\underline{v_k}$. Rule FETCH-INVALID is used when a labeled value cannot be found indexed at the index requested or it does not safely flow to the default labeled value (i.e., it is too secret, too untrustworthy or not available enough), and causes the fetch to evaluate to the specified default labeled value. Since the label of the default value l_d will be used for the label of the fetched value in general, the *availability* of the store level should be bounded above by the availabiltiy of the label of the default value (i.e., $\mathbb{A}(\ell_{store}) \sqsubseteq^{\mathcal{A}} \mathbb{A}(l_d)$) in both rules, as the label of the fetched value should reflect the fact that anyone from the store could have corrupted the value.

5.2.1 Ideal Store Behavior

In this section we informally desecribe the *ideal* behavior of an untrusted store from the perspective of a Clio program.[1] The ideal store semantics provides a specification of the behavior that a real Clio implementation should strive for, and allows the programmer to focus on functionality and security properties of the store rather than the details of cryptographic enforcement of the labeled values. In Section 5.3 we describe how we use cryptography to enforce these ideal security properties.

We use a small-step relation $\langle c, \sigma\rangle \rightsquigarrow \langle c', \sigma'\rangle$ where $\langle c, \sigma\rangle$ and $\langle c', \sigma'\rangle$ are pairs of a Clio configuration c and an ideal store σ. An ideal store σ maps ground values $\underline{v_k}$ to labeled ground values $\langle \underline{v} : l\rangle$. If a store doesn't contain a mapping for an index $\underline{v_k}$, we represent that as mapping it to the distinguished value $\bot$.

Store events emitted by the Clio configurations are used to communicate with the store. When a put $\langle \underline{v} : l\rangle$ at $\underline{v_k}$ event is emitted, the store is updated appropriately. When the Clio computation issues a fetch command, the store provides values appropriately (i.e., either requirement event nothing-at $\underline{v_k}$ or providing $\langle \underline{v} : l\rangle$ for event got$_\tau$ $\langle \underline{v} : l\rangle$ at $\underline{v_k}$). For Clio computation steps that don't interact with the store, store event skip is emitted, and the store is not updated.

5.2.2 Non-Clio Interaction: Threat Model

We assume that programs other than Clio computations may interact with the store and may try to actively or passively subvert the security of Clio programs. Our threat model for these adversarial programs is as follows (and uses store level ℓ_{store} to characterize some of the adversaries' abilities).

- All indices of the key-value store are public information, and an adversary can probe any index of the store and thus notice any and all updates to the store.
- An adversary can read labeled values $\langle \underline{v} : l_1\rangle$ in the store where the confidentiality level of label l_1 is no more confidential than the store level ℓ_{store} (i.e., $\mathbb{C}(l_1) \sqsubseteq^C \mathbb{C}(\ell_{store})$).
- An adversary can put labeled values $\langle \underline{v} : l_1\rangle$ in the store (with arbitrary ground value index $\underline{v_k}$) provided the integrity level of store level ℓ_{store} is at least as trustworthy as the integrity of label l_1 (i.e., $\mathbb{I}(\ell_{store}) \sqsubseteq^I \mathbb{I}(l_1)$).

An adversary can adaptively interact with the store. That is, the behavior of the adversary may depend upon changes the adversary detects or values in the store.

We make the following restrictions on adversaries.

[1] Complete formal definitions in Appendix C.1.6.

Low-Step

$$\frac{\langle c, \bar{I}(\sigma)\rangle \rightsquigarrow \langle c', \sigma'\rangle \qquad \mathrm{PC}(c) \sqsubseteq \ell_{store} \qquad \mathrm{PC}(c') \sqsubseteq \ell_{store}}{(\langle c, \sigma\rangle, \bar{I}) \curvearrowright \langle c', \sigma'\rangle}$$

Low-to-High-to-Low-Step

$$\frac{\langle c, \bar{I}(\sigma)\rangle \rightsquigarrow \langle c_0, \sigma_0\rangle \qquad \langle c_0, \sigma_0\rangle \rightsquigarrow \ldots \rightsquigarrow \langle c_j, \sigma_j\rangle \qquad \forall_{0 \le i < j}.\ \mathrm{PC}(c_i) \not\sqsubseteq \ell_{store} \qquad \mathrm{PC}(c_j) \sqsubseteq \ell_{store}}{(\langle c, \sigma\rangle, \bar{I}) \curvearrowright \langle c_j, \sigma_j\rangle}$$

$$\begin{aligned}
\bar{I} ::= \;& I \cdot \bar{I} \mid I \\
I ::= \;& \mathsf{skip} = \lambda\sigma.\sigma \\
\mid \;& \mathsf{put}\ \langle \underline{v} : l_1 \rangle\ \mathsf{at}\ \underline{v}' = \lambda\sigma.\ \sigma[\underline{v}' \mapsto \langle \underline{v} : l_1 \rangle] \text{ s.t. } \mathbb{I}(\ell_{store}) \sqsubseteq^{I} \mathbb{I}(l_1) \\
\mid \;& \mathsf{corrupt}\ \underline{v}_1, \ldots \underline{v}_n = \lambda\sigma.\sigma[\underline{v}_1 \mapsto \bot; \ldots\ \underline{v}_n \mapsto \bot]
\end{aligned}$$

Figure 5.3: Adversary interactions and low steps

- The adversary does not have access to timing information. That is, it cannot observe the time between updates to the store. We defer to orthorgonal techniques to mitigate the impact of timing channels [8]. For example, Clio could generate store events on a fixed schedule.
- The adversary cannot observe termination of a Clio program, including abnormal termination due to a failed label check. This assumption can be satisfied by requiring that all Clio programs do not diverge and are written defensively to avoid abnormal termination, e.g., by using getLabel to check the label of a labeled value before unlabeling it. Program analysis can ensure these conditions, and in the rest of the chapter we consider only Clio programs that terminate normally.

We formally model the non-Clio interactions with the store using sequences of *adversary interactions* I, given in Figure 5.3. Adversary interactions are skip, put $\langle \underline{v} : l_1 \rangle$ at $\underline{v}'$ and corrupt $\underline{v}_1, \ldots \underline{v}_n$, which, respectively: do nothing; put a labeled value in the store; and delete the mappings for entries at indices $\underline{v}_1$ to $\underline{v}_n$. For storing labeled values, we restrict the integrity of the labeled value stored by non-Clio interactions to be at most as trustworthy as the integrity of the level of the store. Sequences of interactions $I_1 \cdot \ldots \cdot I_n$ are notated as $\bar{I}$.

To model the adversary actively updating the store, we define a step semantics $\curvearrowright$ that includes the adversary interactions $\bar{I}$. We restrict interactions to occur only at *low steps*, i.e., when the current label of the Clio computation is less than or equal to the store level ℓ_{store}. (By contrast, a *high step* is

when the current label can not flow to ℓ_{store}.) Rules Low-Step and Low-To-High-To-Low-Step express adversary interactions occurring only at low steps.

5.3 Realizing Clio

In this section we describe how Clio uses cryptography to enforce the policies on the labeled values through a formal model, called the real Clio store semantics. This model serves as the basis on which to establish strong, formally proven, computational guarantees of the Clio system. We first describe how DC labels are enforced with cryptographic mechanisms (Section 5.3.1), and then describe the real Clio store semantics (Section 5.3.2).

5.3.1 Cryptographic DC Labeled Values

Clio, like many systems that use cryptography, identifies security principals with the *public key* of a cryptographic key pair, and associates the authority to act as a given principal with possession of the corresponding *private key*. At a high level, we will ensure that only those with access to a principal's private key can access information confidential to that principal and vouch for information on behalf of that principal.

Clio tracks key pairs in a *keystore*. Formally, a keystore is a mapping $\mathcal{P} : p \mapsto (\{0,1\}^*, \{0,1\}^*_\perp)$, where p is the principal's well-known name, and the pair of bit strings contains the public and private keys for the principal. In general, the private key for a principal may not be known—represented by $\perp$—which corresponds to knowing the identity of a principal, but not possessing its authority. Keystores are the basis of authority and identity for Clio computations. We use meta-functions on keystores to describe the authority of a keystore in terms of DC labels.[1] Conceptually, a keystore can access and vouch for any information for a principal for which it has the principal's private key. Meta-function $\mathsf{authorityOf}(\mathcal{P})$ returns a label where each component (confidentiality, integrity, and availability) is the conjunction of all principals for which the keystore $\mathcal{P}$ has the private key. We also use the keystore is to determine the starting label of a Clio program ($\mathsf{Start}(\mathcal{P})$, which is the most public, trusted, and available label possible given the keystore's authority), and the least restrictive clearance for a Clio computation ($\mathsf{Clr}(\mathcal{P})$, which is the most confidential (the conjunction of principals in the keystore), least trustworthy (the disjunction of principals in the keystore), and least available (the disjunction of principals in the keystore) data that the computation can compute on given the keystore's authority).

[1] Complete definitions for these functions are in Appendix C.1.8.

Using the principal keystore as a basis for authority and identity for principals, Clio derives a cryptographic protocol that enforces the security policies of safe information flows defined by DC labels.

In the DC label model, labels are made up of triples of *formulas*. Formulas are conjunctions of *categories* $C_1 \wedge \ldots \wedge C_n$. Categories are disjunctions of principals $p_1 \vee \ldots \vee p_n$. Any principal in a category it is a member of can read (for confidentiality) and vouch for (for integrity) information bounded above by the level of the category. We enforce that ability cryptographically by ensuring that only principals in the category have access to the private key for that category. Clio achieves this through the use of *category keys*.

A category key *ck* serves as the cryptographic basis of authority and identity for a category. A category public key is readable by all principals, while the category private key is only readable by members of the category. Category keys are created lazily by Clio as needed and placed in the store. A category key is created using a randomized meta-function[1] parameterized by the keystore. The generated category private key is encrypted for each member of the category separately using each member principal's public key. To prevent illegitimate creation of category keys, a category key is signed using the private key of one of the category members.[2] When a category key is created and placed in the store, it can be fetched by anyone but decrypted only by the members of the category. When a Clio computation fetches a category key, it verifies the signature of the category key to ensure that a category member actually created it. (Failing to verify the signature would allow an adversary to trick a Clio computation into using a category key that is accessible to non-category members.)

A Clio computation encrypts data confidential to a formula $C_1 \wedge \ldots \wedge C_n$ by chaining the encryptions of the value. It first encrypts using C_1's category public key and then encrypts the resulting ciphertext for formula $C_2 \wedge \ldots \wedge C_n$. This form of layered encryption relies on a canonical ordering of categories; we use a lexicographic ordering of principals to ensure a canonical ordering of encryptions and decryptions.

A Clio computation signs data for a formula by signing the data with each category's private key and then concatenating the signatures together. Verification succeeds only if every category signature can be verified.

Equipped with a mechanism to encrypt and sign data for DC labels that conceptually respects safe information flows in Clio, we use this mechanism to serialize and deserialize labeled values to

[1] Defined formally in Appendix C.1.7.

[2] The Clio runtime ensures that the first time a category key for a given category is required, it will be because data confidential to the category or vouched for by the category is being written to the store, and thus the computation has access to at least one category member's private key. Note that any computation with the authority of a category member has the authority of the category.

the store. Give a labeled ground value $\langle\langle l_c, l_i, l_a\rangle : \underline{v}\rangle$, the value $\underline{v}$ is signed according to formula l_i. The value and signature are encrypted according to formula l_c, and the resulting bitstring is the serialization of the labeled value. Deserialization performs decryption and then verification. If deserialization fails, then Clio treats it like a missing entry, and the fetch command that triggered the deserialization would evaluate to the default labeled value.

Replay Attacks Unfortunately, using just encryption and signatures does not faithfully implement the ideal store semantics: the adversary is able to swap entries in the store, or re-use a previous valid serialization, and thus in a limited way modify high-integrity labeled values in the store. We prevent these attacks by requiring that the encryption of the ground value and signature also includes the index value (i.e., the key used to store the labeled value) and a *version number*. The real Clio semantics keeps track of the last seen version of a labeled value for each index of the store. When a value is serialized, the version of that index is incremented before being put in the store. When the value is deserialized the version is checked to ensure that the version is not before a previously used version for that index. In practice, this version counter could be implemented as a vector clock between Clio computations to account for concurrent access to the store. However, for simplicity, we model the version as a natural number in the real Clio store semantics.

5.3.2 Clio Store Semantics

In this section we describe the real Clio store semantics in terms of a small-step probabilistic relation $\rightsquigarrow_p$. The relation models a step taken from a real Clio configuration to a real Clio configuration with probability p. A real Clio configuration is a triple $\langle c, \mathbb{R}, \mathbf{V}\rangle$ of a Clio configuration c, a *distribution* of *sequences* of *real interactions* $\mathbb{R}$, and a version map $\mathbf{V}$. Note that the interactions includes both adversary interactions and interactions made by Clio. The version map tracks version numbers for the store to prevent replay attacks, as described above. For technical reasons, instead of the configuration representing the key-value store as a map, we use the history of store interactions (which includes interactions made both by the Clio computation and the adversary). The sequence of interactions applied to the initial store gives the current store. Because the real Clio store semantics are probabilistic (due to the use of a probabilistic cryptosystem and cryptographic-style probabilistic polynomial-time adversaries), configurations contain distributions over sequences of store interactions.

Real interactions R (and their sequences $\overline{R}$) are defined in Figure 5.4 and are similar to adversary interactions with the ideal store. However, instead of labeled values containing ground values, they

LOW-STEP

$$\frac{\begin{array}{c}\mathbb{R}' = \{\!|\ \overline{R}_A \cdot \overline{R} \mid \overline{R}_A \leftarrow \mathbb{R}_A;\ \overline{R} \leftarrow \mathbb{R}\ |\!\} \\ \langle c, \mathbb{R}', \mathbf{V}\rangle \rightsquigarrow_p \langle c', \mathbb{R}'', \mathbf{V}'\rangle \\ \mathrm{PC}(c) \sqsubseteq \mathbb{C}(\ell_{store}) \qquad \mathrm{PC}(c') \sqsubseteq \mathbb{C}(\ell_{store})\end{array}}{(\langle c, \mathbb{R}, \mathbf{V}\rangle, \mathbb{R}_A) \curvearrowright_p \langle c', \mathbb{R}'', \mathbf{V}'\rangle}$$

LOW-TO-HIGH-TO-LOW-STEP

$$\frac{\begin{array}{c}\mathbb{R}' = \{\!|\ \overline{R}_A \cdot \overline{R} \mid \overline{R}_A \leftarrow \mathbb{R}_A;\ \overline{R} \leftarrow \mathbb{R}\ |\!\} \\ \langle c, \mathbb{R}', \mathbf{V}\rangle \rightsquigarrow_{p_0} \langle c_0, \mathbb{R}_0, \mathbf{V})\rangle \\ \langle c_0, \mathbb{R}_0, \mathbf{V}_0\rangle \rightsquigarrow_{p_1} \ldots \rightsquigarrow_{p_j} \langle c_j, \mathbb{R}_j, \mathbf{V}_j\rangle \\ \forall_{0 \le i < j}.\ \mathrm{PC}(c_i) \not\sqsubseteq \ell_{store} \qquad \mathrm{PC}(c_j) \sqsubseteq \ell_{store} \qquad p = \Pi_{0 \le i \le j}\, p_i\end{array}}{(\langle c, \mathbb{R}, \mathbf{V}\rangle, \mathbb{R}_A) \curvearrowright_p \langle c_j, \mathbb{R}_j, \mathbf{V}_j\rangle}$$

$$\begin{array}{rrl} \text{Interactions:} & R ::= & \mathsf{skip} = \lambda\sigma.\ \sigma \\ & \mid & \mathsf{put}\ ck\ \mathsf{at}\ C = \lambda\sigma.\ \sigma[C \mapsto ck] \\ & \mid & \mathsf{put}\ \langle s : l\rangle\ \mathsf{at}\ \underline{v}_k = \lambda\sigma.\ \sigma[\underline{v}_k \mapsto \langle s : l\rangle] \\ \text{Strategies:} & \mathcal{S} & :\ \mathbb{R} \to \mathbb{R} \end{array}$$

$$\begin{array}{rcl} \mathbf{step}^{\mathcal{P}}_{\ell_{store}}(c_0, \mathcal{S}, 1) & = & \left\{ \left(\langle c_1, \mathbb{R}_1, \mathbf{V}_1\rangle, p_0 \cdot p_1\right) \;\middle|\; (\langle c_0, \{(\mathsf{skip}, 1)\}, \Sigma_0\rangle, \mathcal{S}(\{(\mathsf{skip}, 1)\})) \curvearrowright_{p_1} \langle c_1, \mathbb{R}_1, \mathbf{V}_1\rangle \right\} \\ \mathbf{step}^{\mathcal{P}}_{\ell_{store}}(c_0, \mathcal{S}, j+1) & = & \left\{ \left(\langle c_2, \mathbb{R}_2, \mathbf{V}_2\rangle, p_0 \cdot p_1\right) \;\middle|\; \left(\langle c_1, \mathbb{R}_1, \mathbf{V}_1\rangle, p_0\right) \in \mathbf{step}^{\mathcal{P}}_{\ell_{store}}(c_0, \mathcal{S}, j); \right. \\ & & \left. (\langle c_1, \mathbb{R}_1, \mathbf{V}_1\rangle, \mathcal{S}(\mathbb{R}_1)) \curvearrowright_{p_1} \langle c_2, \mathbb{R}_2, \mathbf{V}_2\rangle \right\} \end{array}$$

Figure 5.4: Real Clio low step semantics

contain bitstrings b (expressing the low-level details of the cryptosystem and the ability of the adversary to perform bit-level operations).

To express probability distributions, we use notation

$$\{\!|\, f(X_1, \ldots, X_n) \mid X_1 \leftarrow D_1;\ \ldots X_n \leftarrow D_n\ |\!\}$$

to describe the distribution over the function f with inputs of random variables $X_1, \ldots, X_n$ where X_i is distributed according to distribution D_i for $1 \le i \le n$.

Figure 5.5 presents the inference rules for $\rightsquigarrow_p$. Internal steps do not affect the interactions or versions. For storing (rule STORE), the version of the entry is incremented using the increment function and the real Clio configuration uses a new distribution of interactions $\mathbb{R}'$ containing the interactions to store the labeled value. The new distribution contains the original interactions (distributed according to the original distribution of interactions) along with a concatenation of labeled ciphertexts and any new category keys (distributed according to the distribution given by serialization function). The configuration steps with probability 1 as the STORE rule will be used for all store

operations.

When fetching a labeled value, there are three possible rules that can be used depending on the current state of the store. The premise, $(\sigma, p) \in \{\!|\, \overline{R}(\emptyset) \mid \overline{R} \leftarrow \mathbb{R} \,|\!\}$ means that store σ has probability p of being produced (by drawing interaction sequence $\overline{R}$ from distribution $\mathbb{R}$ and applying $\overline{R}$ to the empty store $\emptyset$ to give store σ).

Which rule is used for a fetch operation depends on the state of the store, and so the transitions may have probability mass less than one. Rule Fetch-Exists is used when the sequence of interactions drawn produces a store that has a serialized labeled value indexed by $\underline{v_k}$ that can be correctly deserialized and whose version is not less than the last version seen at this index. Rule Fetch-Missing is used when the sequence of interactions drawn produces a store that either does not have an entry indexed by $\underline{v_k}$, or has an entry that cannot be correctly deserialized. Finally, Fetch-Replay rule is used when the sequences of interactions drawn produce a store where an adversary has attempted to replay an old value. More precisely, the store has a labeled value that can be deserialized correctly, but whose recorded index is not the same as the index requested by the Clio computation or whose version is less than the version last seen by the Clio computation.

Similar to the ideal store semantics, we also use a low step relation $\curvearrowright_p$ to model adversary interactions, shown in Figure 5.4. The low step relation is also probabilistic as it is based on the probabilistic single step relation $\rightsquigarrow_p$. Additionally, we use a distribution of sequences of adversarial interactions $\mathbb{R}_A$ to model an adversary that behaves probabilistically. In rules Low-Step and Low-To-High-To-Low-Step a new distribution of interactions, $\mathbb{R}'$ is created by concatenating interaction sequences drawn from the existing distribution of interactions $\mathbb{R}$ and the adversary distribution $\mathbb{R}_A$. This is analogous to the application of adversary interactions to the current store in the ideal semantics. The rest of the definitions of the rules follow the same pattern as the ideal Clio low step store semantics.

With the low step relation, we use metafunction $\mathbf{step}$ to describe the distributions of real Clio configurations resulting from taking j low steps from configuration c_o, formally defined in Figure 5.4. The $\mathbf{step}$ function is parameterized by the keystore $\mathcal{P}$ and store level ℓ_{store}. To provide a source of adversary interactions while running the program, the $\mathbf{step}$ function also takes as input a *strategy* $\mathcal{S}$ which is a function from distributions of interactions to distributions of interactions, representing the probabilities of interactions an active adversary would perform. Before each low step, the strategy is invoked to produce a distribution of interactions that will affect the store that the Clio computation is using. That is, strategy models the probabilistic behavior of the interactions made by the adversary.

INTERNAL-STEP

$$\frac{c \longrightarrow c'}{\langle c, \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_{\mathrm{I}} \langle c', \mathbb{R}, \mathbf{V}\rangle}$$

STORE

$$\frac{\begin{array}{c} c \xrightarrow{\text{put } \langle \underline{v} : l_1 \rangle \text{ at } \underline{v}_k} c' \\ n = \text{increment}(\mathbf{V}(\underline{v}_k)) \qquad \mathbf{V}' = \mathbf{V}[\underline{v}_k \mapsto n] \\ \mathbb{R}' = \left\{\!\!\left| \begin{array}{l} \text{put } \langle s : l_1 \rangle \text{ at } \underline{v}_k \cdot \overline{R'} \cdot \overline{R} \\ (\overline{R'}, \langle s : l_1 \rangle) \leftarrow \text{serialize}_{\mathcal{P}}(\sigma, \langle (\underline{v}, \underline{v}_k, n) : l_1 \rangle) \end{array} \right| \overline{R} \leftarrow \mathbb{R}; \right\}\!\!\right| \end{array}}{\langle c, \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_{\mathrm{I}} \langle c', \mathbb{R}', \mathbf{V}'\rangle}$$

FETCH-EXISTS

$$\frac{\begin{array}{c} c \xrightarrow{\text{got } \tau \ \langle \underline{v} : l_1 \rangle \text{ at } \underline{v}_k} c' \qquad n \not< \mathbf{V}(\underline{v}_k) \\ (\sigma, p) \in \left\{\!\!\left| \overline{R}(\emptyset) \mid \overline{R} \leftarrow \mathbb{R} \right|\!\!\right\} \qquad p > 0 \\ \langle (\underline{v}, \underline{v}_k, n) : l_1 \rangle = \text{deserialize}_{\mathcal{P}}(\sigma, \sigma(\underline{v}_k), \tau) \end{array}}{\langle c, \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_{p} \langle c, \mathbb{R}, \mathbf{V}\rangle}$$

FETCH-MISSING

$$\frac{\begin{array}{c} c \xrightarrow{\text{nothing-at } \underline{v}_k} c' \\ (\sigma, p) \in \left\{\!\!\left| \overline{R}(\emptyset) \mid \overline{R} \leftarrow \mathbb{R} \right|\!\!\right\} \qquad p > 0 \\ \text{deserialize}_{\mathcal{P}}(\sigma, \sigma(\underline{v}_k), \tau) \text{ undefined} \end{array}}{\langle c, \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_{p} \langle c, \mathbb{R}, \mathbf{V}\rangle}$$

FETCH-REPLAY

$$\frac{\begin{array}{c} c \xrightarrow{\text{nothing-at } \underline{v}_k} c' \qquad \underline{v}'_k \neq \underline{v}_k \text{ or } n < \mathbf{V}(\underline{v}_k) \\ (\sigma, p) \in \left\{\!\!\left| \overline{R}(\emptyset) \mid \overline{R} \leftarrow \mathbb{R} \right|\!\!\right\} \qquad p > 0 \\ \langle (\underline{v}, \underline{v}'_k, n) : l_1 \rangle = \text{deserialize}_{\mathcal{P}}(\sigma, \sigma(\underline{v}_k), \tau) \end{array}}{\langle c, \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_{p} \langle c, \mathbb{R}, \mathbf{V}\rangle}$$

Figure 5.5: Real Clio semantics

5.4 Formal Properties

In this section, we describe two formal properties of the real Clio store semantics.[1] We first show that if the cryptosystem is secure under *chosen-plaintext attacks*, then Clio is secure under *chosen-term attacks* (Section 5.4.1). A chosen-term attack is similar to a chosen-plaintext attack but more general as the adversary can choose a term (i.e., a program) to run rather than just a plaintext to encrypt. We then show that if the cryptosystem is secure against *existential forgery* under *chosen-message attacks* then Clio is secure against *leveraged forgery* (Section 5.4.2). Leveraged forgery is similar to existential

[1] Complete defintions and proofs are in Appendices C.1 and C.2.

forgery under chosen-message attacks, however it is similarly more general as the adversary can also provide a term to run to potentially produce a message with a valid signature.

5.4.1 Indistinguishability

A cryptosystem is *semantically secure* if, informally, the ciphertexts of messages of equal lengths are *computationally indistinguishable* from one another. Two sequences of probability distributions are computationally indistinguishable (written $\{X_n\}_n \approx \{Y_n\}_n$) if for all non-uniform probabilistic polynomial time (ppt) algorithms $\mathcal{A}$,

$$\left| \Pr[\mathcal{A}(x) = 1 \mid x \leftarrow X_n] - \Pr[\mathcal{A}(y) = 1 \mid y \leftarrow Y_n \,] \right|$$

is *negligible* in n [33].

In modern cryptosystems, semantic security is defined as indistinguishability under chosen-plaintext attacks (CPA), defined below [64].

Definition 10 (Indistinguishability under Chosen-Plaintext Attack)**.** *Let the random variable* $\mathrm{IND}_b(\mathcal{A}, n)$ *denote the output of the experiment, where* $\mathcal{A}$ *is non-uniform ppt,* $n \in \mathbb{N}$*,* $b \in \{0, 1\}$ *:*

$$\begin{aligned} \mathrm{IND}_b(\mathcal{A}, n) = {} & (pk, sk) \leftarrow \mathsf{Gen}(1^n); \\ & m_0, m_1, \mathcal{A}_2 \leftarrow \mathcal{A}(pk) \text{ s.t. } |m_0| = |m_1|; \\ & c \leftarrow \mathsf{Enc}(pk, m_b); \\ & \text{Output } \mathcal{A}_2(c) \end{aligned}$$

$\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is Chosen-Plaintext Attack (CPA) secure if for all non-uniform ppt* $\mathcal{A}$*:*

$$\left\{ \mathrm{IND}_0(\mathcal{A}, n) \right\}_n \approx \left\{ \mathrm{IND}_1(\mathcal{A}, n) \right\}_n$$

This definition of indistinguishability phrases the security of the cryptosystem in terms of a game. In this game, an adversary receives the public key and then produces two plaintext messages of equal length. The game then chooses one of the two messages to encrypt and passes the resulting ciphertext back to the adversary. The cryptosystem is CPA Secure if no adversary exists that can produce substantially different distributions of output based on the choice of message. In other words, no computationally-bounded adversary is able to effectively distinguish which message was encrypted.

Clio relies on a semantically secure cryptosystem. However, this is not enough to show that Clio protects the confidentiality of secret information. This is because CPA Security provides guarantees only for individually chosen plaintext messages. In contrast, in our setting we consider *terms* (i.e.,

programs) chosen by an adversary. There are also many principals and as a result many keys in a real system, so Clio must protect arbitrarily many principals' information from the adversary. Additionally, the adversary may already have access to some of the keys. Finally, the adversary is active: it can see interactions with the store and issue new interactions adaptively while the program is running. It may be the case that an adversary can leverage a Clio computation to illegitimately produce a value it should not have. It may, through the course of interacting with the store, trick the Clio system into leaking secret information, especially when considering that the program itself may be untrusted.

With these considerations in mind, we define indistinguishability under a new form of attack: *chosen-term attacks* (CTA).

Definition 11 (Indistinguishability under Chosen-Term Attack)**.** *Let the random variable* $\mathrm{IND}_b(\mathcal{P}, \mathcal{A}, \tilde{p}, j, n)$ *denote the output of the following experiment, where* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Sign}, \mathsf{Verify})$*,* $\mathcal{A}$ *is non-uniform ppt,* $n \in \mathbb{N}$*,* $b \in \{0, 1\}$*:*

$$
\begin{aligned}
&\mathrm{IND}_b(\mathcal{P}_0, \mathcal{A}, \tilde{p}, j, n) = \\
&\quad \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}'; \\
&\quad t, v_0, v_1, \mathcal{S}, \mathcal{A}_2 \leftarrow \mathcal{A}(\mathrm{pub}(\mathcal{P}))\ \textit{such that}\ v_0 =^{C}_{\ell_{store}} v_1 \\
&\qquad\qquad\qquad\qquad \textit{and}\ \vdash t : \mathsf{Labeled}\ \tau \to \mathsf{LIO}\ \tau' \\
&\qquad\qquad\qquad\qquad \textit{and}\ \vdash v_0 : \mathsf{Labeled}\ \tau \\
&\qquad\qquad\qquad\qquad \textit{and}\ \vdash v_1 : \mathsf{Labeled}\ \tau \\
&\qquad\qquad\qquad\qquad \textit{and}\ \ell_{store} = \mathsf{authorityOf}(\mathcal{P}_0); \\
&\quad \langle c, \mathbb{R}_b,\ \boldsymbol{V}' \rangle \leftarrow \boldsymbol{step}^{\mathcal{P}}_{\ell_{store}}(\langle \mathsf{Start}(\mathcal{P}), \mathsf{Clr}(\mathcal{P}) \mid (t\ v_b) \rangle, \mathcal{S}, j); \\
&\quad \overline{R_b} \leftarrow \mathbb{R}_b;\ \mathrm{Output}\ \mathcal{A}_2(\overline{R_b})
\end{aligned}
$$

Clio using Π *is CTA Secure if for all non-uniform ppt* $\mathcal{A}$*,* $j \in \mathbb{N}$*, keystores* $\mathcal{P}$*, and principals* $\tilde{p}$*:*

$$
\left\{ \mathrm{IND}_0(\mathcal{P}, \mathcal{A}, \tilde{p}, j, n) \right\}_n \approx \left\{ \mathrm{IND}_1(\mathcal{P}, \mathcal{A}, \tilde{p}, j, n) \right\}_n
$$

This version of the game follows the same structure as the CPA game. In addition, though, we allow the adversary to know certain information (by fixing it in the game), including some part of the keystore ($\mathcal{P}_0$), the set of principals that Clio is protecting ($\tilde{p}$), and the number of low steps the program takes (j).

In this game setup, $\mathsf{Gen}(\tilde{p}, 1^n)$ generates a new keystore $\mathcal{P}'$ containing private keys for each of the principals in $\tilde{p}$, using the underlying cryptosystem's Gen function for each keypair. Then, the adversary receives all the public keys of the keystore $\mathrm{pub}(\mathcal{P})$ and now, instead of returning two plaintext

mesasges (in CPA), it instead returns three well-typed Clio terms: a function t, and two program inputs to the function v_0 and v_1 that must be confidentiality-only low equivalent $=^{C}_{\ell_{store}}$ (i.e., they may differ only on secret values)[1]. It also returns a strategy $\mathcal{S}$ that it can use to interact with the store while it is running. In practice the strategy would not be explicitly used, but instead in this formal game $\mathcal{S}$ is a a non-uniform probabilistic polynomial-time function (as it comes from the $\mathcal{A}$ function which is also polynomial-time) that captures all of the possible adversary interactions and the probabilities with which it would make those interactions. The program t_0 is run with one of the inputs v_0 or v_1 for j steps. The adversary receives the interactions resulting from a run of the program and needs to use that information to determine which secret input the program was run with.

Being secure under a chosen-term attack means that the sequences of interactions between two low-equivalent programs are indistinguishable and hence an adversary does not learn any secret information from the store despite actively interacting with it while the program it chose is running. Note that the adversary receives the full trace of interactions on the store (including its own interactions); this gives it enough information to reconstruct the final state of the store and any intermediate state. For any set of principals, and any adversary store level, the interactions with the store contain no efficiently extractable secret information for all well-typed terminating programs.

We can show that Clio satisfies this security guarantee.

Theorem 12 (CTA Security)**.** *If* Π *if CPA Secure, then Clio using* Π *is CTA Secure.*

We prove this theorem in part by induction over the low step relation $\curvearrowright_p$, to show that two low equivalent configurations will produce low equivalent configurations, including computationally indistinguishable distributions over sequences of interactions. A subtlety is that we must strengthen the inductive hypothesis to show that sequences of interactions satisfy a stronger syntactic relation (rather than being just computationally indistinguishable).

More concretely, the proof follows three high-level steps. First, we show how a relation $\asymp$ on families of distributions of sequences of interactions preserves computational indistinguishability. That is, if $\mathbb{R}_1 \asymp \mathbb{R}_2$ and Π is CPA secure, then $\mathbb{R}_1 \approx \mathbb{R}_2$. Second, we show that as two low equivalent configurations step using the low step relation $\curvearrowright_p$, low equivalence is preserved and the interactions they produce satisfy the relation $\asymp$. Third, we show that the use of the **step** metafunction on two low equivalent configurations will produce computationally indistinguishable distributions over distributions of sequences of interactions. Each step of the proof relies on the previous step and the first step relies on the underlying assumptions on the cryptosystem. We now describe each step of the proof in more detail.

[1] Complete definition of low equivalence is in Appendix C.1.5.

Step 1: Interactions Relation We consider pairs of arbitrary distributions of sequences of interactions and show that, if they are both of a certain syntactic form then they are indistinguishable. Importantly, the indistinguishability lemmas do not refer to the Clio store semantics, i.e., they merely describe the form of arbitrary interactions that may or may not have come from Clio. The invariants on pairs of indistinguishable distributions of interactions implicitly require low equivalence of the programs that generated them, and low equivalence circularly requires indistinguishable distributions of interactions. As a result, we describe the lemmas free from the Clio store semantics to break the circularity.

We progressively define the relation $\asymp$ on a pair of interactions. Initially, distributions of interactions only contain secret encryptions so that we can appeal to a standard cryptographic argument of multi-message security. Formally, for all keystores $\mathcal{P}_0$, and $l_1, \ldots, l_k$, such that $\mathbb{C}(l_i) \sqsubseteq^C \mathbb{C}(\mathsf{authorityOf}(\mathcal{P}))$, and for all $m^1_{\{1,2\}} \ldots m^n_{\{1,2\}}$ and all principals $\tilde{p}$, if $|m^i_1| = |m^i_2|$ for all $1 \leq i \leq k$ and Π is CPA Secure, then

$$\Big\{ \mathsf{put} \langle b^1_1 : l^1 \rangle \ \mathsf{at} \ \underline{v}^1 \cdot \ \ldots \ \cdot \mathsf{put} \langle b^k_1 : l^k \rangle \ \mathsf{at} \ \underline{v}^k \ \Big| \\ \mathcal{P} \leftarrow \mathsf{Gen}(1^n);\ (pk^i, sk^i) \in rng(\mathcal{P});\ b^i_1 \leftarrow \mathsf{Enc}(pk^i, m^i_1);\ 1 \leq i \leq k \Big\}_n \\ \asymp \\ \Big\{ \mathsf{put} \langle b^1_2 : l^1 \rangle \ \mathsf{at} \ \underline{v}^1 \cdot \ \ldots \ \cdot \mathsf{put} \langle b^k_2 : l^k \rangle \ \mathsf{at} \ \underline{v}^k \ \Big| \\ \mathcal{P} \leftarrow \mathsf{Gen}(1^n);\ (pk^i, sk^i) \in rng(\mathcal{P});\ b^i_2 \leftarrow \mathsf{Enc}(pk^i, m^i_2);\ 1 \leq i \leq k \Big\}_n$$

Using multi-message security as a basis for indistinguishability, we then expand the relation to contain readable encryptions (i.e., ones for which the adversary has the private key to decrypt) where the values encrypted are the same. In the complete definiton of $\asymp$, we expand it to also contain interactions from a strategy, forming the final relationship on interactions captured by the $\asymp$ relation.

We establish an invariant that must hold between pairs in the relation in order for them to be indistinguishable. For example, in the first definition, the lengths of each corresponding message between the pair must be the same. Each intermediate definition of $\asymp$ is used to show that a ppt can simulate the extra information in the more generalized definition (thus providing no distinguishing power). For the first definition of the relation containing only secret encryptions, a hybrid argument is used similar to showing multi-message CPA security [64].

Step 2: Preservation of Low Equivalence We show that as two low equivalent programs t and t' progress, they simultaneously preserve low equivalence $t =^C_{\ell_{store}} t'$ and the distributions of sequences of interactions they produce $\mathbb{R}$ and $\mathbb{R}'$ are in the relation $\asymp$.

We first show that if $c_0 \xrightarrow{\alpha} c'_0$ and $c_1 \xrightarrow{\alpha} c'_1$ and $c_0 =^C_{\ell_{store}} c_1$ then $c'_0 =^C_{\ell_{store}} c'_1$. This proof takes advantage of the low equivalence preservation proofs for LIO in all cases except for the storing

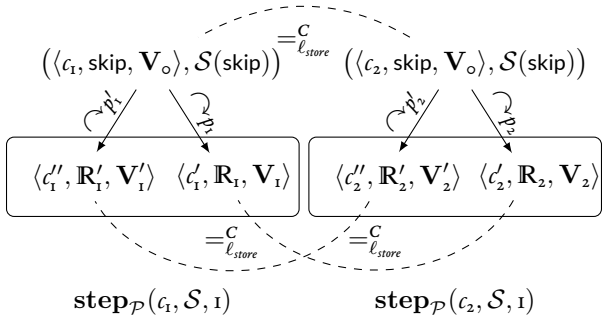

Figure 5.6: Low equivalence is preserved in $\mathbf{step}_{\mathcal{P}}$ for two low equivalent configurations c_1 and c_2 and a strategy $\mathcal{S}$.

and fetching rules. For store events, since all values being stored will have the same type (due to type soundness), and will be ground values, serialized values will have the same message lengths.

We then show that if $(\langle c_o, \mathbb{R}_o, \mathbf{V}_o\rangle, \mathbb{R}) \curvearrowright_p \langle c'_o, \mathbb{R}'_o, \mathbf{V}'_o\rangle$ and $(\langle c_1, \mathbb{R}_1, \mathbf{V}_1\rangle, \mathbb{R}) \curvearrowright_p \langle c'_1, \mathbb{R}'_1, \mathbf{V}'_1\rangle$ and $c_o =^{C}_{\ell_{store}} c_1$ and $\mathbf{V}_o = \mathbf{V}_1$ and $\mathbb{R}_o \asymp \mathbb{R}_1$ then $c'_o =^{C}_{\ell_{store}} c'_1$ and $\mathbf{V}'_o = \mathbf{V}'_1$ and $\mathbb{R}'_o \asymp \mathbb{R}'_1$. The proof on $\curvearrowright_p$ relies on the previous preservation proof on $\xrightarrow{\alpha}$ and the indistinguishability results on $\asymp$.

Step 3: Indistinguishability of the step metafunction We show that the $\mathbf{step}$ metafunction preserves low equivalence. More formally, we show that if $c_o =^{C}_{\ell_{store}} c_1$ and $\mathbf{V}_o = \mathbf{V}_1$ and $\mathbb{R}_o \asymp \mathbb{R}_1$ then

$$\{(\mathbb{R}'_o, p_o \cdot \ldots \cdot p) \mid \langle c_o, \mathbb{R}_o, \mathbf{V}_o\rangle \curvearrowright_{p_o} \ldots \curvearrowright_p \langle c'_o, \mathbb{R}'_o, \mathbf{V}'_o\rangle\}_n$$
$$\approx$$
$$\{(\mathbb{R}'_1, p'_o \cdot \ldots \cdot p') \mid \langle c_1, \mathbb{R}_1, \mathbf{V}_1\rangle \curvearrowright_{p'_o} \ldots \curvearrowright_{p'} \langle c'_1, \mathbb{R}'_1, \mathbf{V}'_1\rangle\}_n$$

We prove this by showing that the probabilities of traces taken by two low equivalent configurations are equal with all but negligible probability. As an example, Figure 5.6 shows graphically how one step of the trace is handled. We examine the result of $\mathbf{step}_{\mathcal{P}}(c_1, \mathcal{S}, 1)$ and $\mathbf{step}_{\mathcal{P}}(c_2, \mathcal{S}, 1)$ where $c_1 =^{C}_{\ell_{store}} c_2$. (Note that this setup matches the instantiation of the CTA game where $j = 1$.) The left rectangle shows the resulting distribution over distributions of configurations after one step of the c_1 configuration. The right circle shows the resulting distribution over distributions of configurations after one step of the c_2 configuration. Due to the results from Step 2, we can reason that $c'_1 =^{C}_{\ell_{store}} c'_2$ and that $c''_1 =^{C}_{\ell_{store}} c''_2$. We can also conclude that $\mathbb{R}_1 \asymp \mathbb{R}_2$ and that $\mathbb{R}'_1 \asymp \mathbb{R}'_2$. The final step of the proof is to show that the interactions from the resulting two distributions (i.e., the top circle and

bottom circle) are computationally indistinguishable. That is, we show that p_1 is equal to p_2 and also p_1' is equal to p_2' with all but negligible probability.

5.4.2 Leveraged Forgery

Whereas in the previous subsection we considered the security of encryptions, in this case we consider the security of the signatures. We show that an adversary cannot leverage a Clio computation to illegitimately produce a signed value.

A digital signature scheme is secure if it is difficult to forge signatures of messages. Clio requires its digital signature scheme to be secure against *existential forgery* under a *chosen-message attack*, where the adversary is a non-uniform ppt in the size of the key. Often stated informally in the literature [32], a digital signature scheme is secure against *existential forgery* if no adversary can succeed in forging the signature of one message, not necessarily of his choice. Further, the scheme is secure under a *chosen-message attack* if the adversary is allowed to ask the signer to sign a number of messages of the adversary's choice. The choice of these messages may depend on previously obtained signatures.

Parallel to CPA and CTA, we adapt the definition of existential forgery for Clio, which we call *leveraged forgery*. Intuitively, it should not be the case that a high integrity signature can be produced for a value when it is influenced by low integrity information. We capture this intuition in the following theorem:

Theorem 13 (Leveraged Forgery)**.** *For a principal p and all keystores $\mathcal{P}_0$, non-uniform ppts $\mathcal{A}$, and labels l_1, integers j, j', where $\ell_{store} = \mathsf{authorityOf}(\mathcal{P}_0)$ and $\mathbb{I}(l_1) \sqsubseteq^I p$, if Π is secure against existential forgery under chosen-message attacks, then*

$$
\begin{array}{l}
\boldsymbol{Pr}\Big[\ \langle b : l_1 \rangle \in \text{Values}_{\mathcal{P}}(\overline{R'}) \text{ and } \langle b : l_1 \rangle \notin \text{Values}_{\mathcal{P}}(\overline{R}) \\
\quad \mid\ \mathcal{P}' \leftarrow \mathsf{Gen}(\{p\}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}'; \\
\quad\ t, \mathcal{S}, \mathcal{A}_2 \leftarrow \mathcal{A}(\text{pub}(\mathcal{P})); \\
\quad\ \langle c, \mathbb{R},\ \boldsymbol{V}\rangle \leftarrow \boldsymbol{step}^{\mathcal{P}}_{\ell_{store}}(\langle \mathsf{Start}(\mathcal{P}), \mathsf{Clr}(\mathcal{P}) \mid t\rangle, \mathcal{S}, j); \\
\quad\ \overline{R} \leftarrow \mathbb{R}; \\
\quad\ t', \mathcal{S}' \leftarrow \mathcal{A}_2(\overline{R}); \\
\quad\ \langle c', \mathbb{R}',\ \boldsymbol{V}'\rangle \leftarrow \boldsymbol{step}^{\mathcal{P}}_{\ell_{store}}(\langle \mathsf{Start}(\mathcal{P}_0), \mathsf{Clr}(\mathcal{P}) \mid t'\rangle, \mathcal{S}', j); \\
\quad\ \overline{R'} \leftarrow \mathbb{R}'\Big]
\end{array}
$$

Intuitively, the game is structured as follows. First, an adversary chooses a term t and strategy $\mathcal{S}$

that will be run with high integrity (i.e., $\mathsf{Start}(\mathcal{P})$ where $\mathcal{P}$ has p's authority). The adversary sees the interactions $\overline{R}$ produced by the high integrity computation (which in general will include high integrity signatures).

With that information, the adversary constructs a new term t' and new strategy $\mathcal{S}'$ that will be run with low integrity (i.e., $\mathsf{Start}(\mathcal{P}_\circ)$). Note that the strategy may internally encode high integrity signatures learned from the high integrity run that it can place in the store.

The interactions produced by this low integrity computation should not contain any high integrity signatures (i.e., are signed by p). The adversary succeeds if it produces a new valid labeled bitstring $\langle b : l_1 \rangle$ that did not exist in the first run. In the experiment, the $\mathrm{Values}_{\mathcal{P}}$ metafunction extracts the set of valid labeled bitstrings (i.e., can be deserialized correctly) using the parameterized keystore $\mathcal{P}$ to perform the category key decryptions.

The proof of this theorem is in two parts. First we show that the label of a value being stored by a computation is no more trustworthy than the current label of computation. Second, we show that the current label never becomes more trustworthy than the starting label. This means that a low integrity execution (i.e., starting from $\langle \mathsf{Start}(\mathcal{P}_\circ), \mathsf{Clr}(\mathcal{P}) \mid t \rangle$) cannot produce a high integrity value (i.e., a labeled value $\langle b : l \rangle$ such that $\mathbb{I}(l) \sqsubseteq^I p$).

5.5 Clio in Practice

In this section, we describe our prototype implementation and a case study using Clio.

5.5.1 Implementation

We implemented a Clio prototype as a Haskell library, in the same style as LIO. Building on the LIO code base, the Clio library consists in an API for defining and running Clio programs embedded in Haskell. The library also implements a monitor that oversees the execution of the program and orchestrates three interdependent tasks:

- **Information-flow control** Clio executes the usual LIO IFC enforcement mechanism; in particular, it adjusts the current label and clearance and checks that information flows according to the DC labels lattice.
- **External key-value store** Clio handles all interactions with the store, realized as an external Redis database.[1] This is accomplished by using the hedis Haskell library,[2] which im-

[1] http://redis.io/
[2] http://hackage.haskell.org/package/hedis

plements a Redis client.

- **Cryptography** Clio takes care of managing and handling cryptographic keys as well as invoking cryptographic operations to protect the security of the principals' data as it crosses the system boundary into/back from the untrusted store. Instead of implementing our own cryptographic primitives, we leverage the third-party cryptonite library.[1]

Clio uses standard cryptographic schemes to protect the information in the store. In particular, for efficiency reasons we use a hybrid scheme that combines asymmetric cryptography with symmetric encryption. The category keys in the store are encrypted and signed with asymmetric schemes, while the entries stored by Clio programs are encrypted with symmetric encryption and signed with an asymmetric signature scheme.

Asymmetric cryptography We use cryptonite's implementation of RSA, specifically OAEP mode for encryption/decryption and PSS for signing/verification, both with 1024-bit keys and using SHA256 as a hash. We get around the message size limitation by chunking the plaintext and encrypting the chunks separately.

Symmetric encryption We use cryptonite's implementation of AES, specifically AES256 in Counter (CTR) mode for symmetric encryption. We use randomized initialization vectors (IVs) for each encryption. The library can use AESNI if the architecture supports it.

Randomness We use cryptonite's key generation functions and its random-number generator for initialization vectors. We have not evaluated how good these functions are as a source of randomness, but we remark that cryptonite can use RDRAND if the CPU supports it.

Serialization In order to avoid problems with improperly escaped strings, we encode every bitstring in base64.

Storing and Fetching

For each category used in the program we generate a symmetric key and two RSA key pairs: an encryption/decryption key pair and a signing/verification key pair. This information is stored in the database after being asymmetrically encrypted and then signed as described in Section 5.3.1. Category key generation relies on the RSA key pairs for each principal involved, which should be supplied by the user in the form of an initial keystore when the Clio computation starts.

After the relevant IFC effects have been performed, storing a labeled value involves fetching the symmetric key for each category in its confidentiality clause as well as the signature keys that corre-

[1] http://hackage.haskell.org/package/cryptonite

spond to each category in its integrity clause, potentially generating these on the fly. The labeled value is serialized to a bitstring, then RSA-PSS signed by at least one principal per integrity category, and finally AES256-CTR onion-encrypted using the symmetric key for each confidentiality category. Fetching involves the dual operations, i.e., symmetric decryption and RSA-PSS signature verification.

User API

Our library provides all the Clio operations described in the chapter, plus a few extra functions that are necessary to glue Clio code with the rest of the program. Here are some of the most important ones.

Clio code can be run using the `evalCLIO` function. This function takes two arguments: a record `initialState` of type `CLIOState` and a Clio computation m. The record `initialState` provides initial values for the current label, the current clearance, the keystore, the version map and the store label. The function simply establishes a connection with a Redis server and executes m using that database as the store and `initialState` as the local state.

In order to generate keystores, we provide the utility function `initializeKeyMapIO`. This function takes a list of principals as argument, and produces a keystore with fresh asymmetric key pairs for all of them. Our prototype does not provide means to store these keystores beyond the execution of the program, but it would be straightforward for users to implement this functionality in their own programs, or using a suitable PKI.

5.5.2 Case Study

We have implemented a simple case study to illustrate how our prototype Clio implementation can be used to build an application. In this case, we have built a system that models a tax preparation tool and its interactions with a customer (the taxpayer) and the tax reporting agency, communicating via a shared untrusted store. We model these three components as principals C (the customer), P (the preparer) and IRS (the tax reporting agency). The actions of each of these three principals are modeled as separate Clio computations `customerCode`, `preparerCode` and `irsCode`, respectively. We assume that the store level ℓ_{store} restricts writes to the store in confidential contexts, i.e. $\ell = \langle \bot, \top, S \rangle$, where S is the principal running as the store.

The customer C initially makes a record with his/her personal information, including his/her name, social security number (SSN), declared income and bank account details, modeled as the type `TaxpayerInfo`. Figure 5.7 shows the customer code on the left, modeled as a function that takes

```
customerCode :: TaxpayerInfo → LIO ()
customerCode tpi = do
    info ← label ⟨C ∨ P ∨ IRS, C, S⟩ tpi
    store taxpayer_info info
    return ()
```

```
preparerCode :: LIO ()
preparerCode = do
    d ← label ⟨P ∨ IRS, P ∨ C, S⟩ notFound
    info ← fetch τ taxpayer_info d
    r ← toLabeled ⟨P ∨ IRS, P ∨ C, S⟩ $ do
        i ← unlabel info
        return (prepareTaxes i)
    store tax_return r
```

```
irsCode :: LIO Bool
irsCode = do
    let l = ⟨IRS, P ∨ C ∨ IRS, S⟩
    d ← label l emptyTR
    lv ← fetch τ tax_return d
    tr ← unlabel lv
    return (verifyReturn tr)
```

Figure 5.7: Customer code (left), Preparer code (middle), and IRS code (right)

this record as an argument, `tpi`. The first step is to label `tpi` with the label $\langle C \vee P \vee \text{IRS}, C, S\rangle$. The confidentiality component is a disjunction of all the principals in the system, reflecting the fact that the customer expects both the preparer and the IRS to be able to read the data. The integrity component is just C since this data can be vouched for only by the customer at this point, while the availability is trusted since these values haven't been exposed to (and possibly corrupted by) the adversary in the store yet. The final step of the customer is to store their labeled `TaxpayerInfo` at key `"taxpayer_info"` for the preparer to see. Note that in practice this operation creates a category key for $C \vee P \vee \text{IRS}$, stores it in the database and uses it to encrypt the data, which gets signed by C.

The next step is to run the preparer code, shown in the middle of Figure 5.7. The preparer starts by fetching the taxpayer data at key `"taxpayer_info"`, using a default empty record labeled with $l_1 = \langle P \vee \text{IRS}, P \vee C, S\rangle$. The entry in the database is labeled with $l_2 = \langle C \vee P \vee \text{IRS}, C, S\rangle$, but the operation succeeds because $l_1 \sqsubseteq l_2$ and the availability is properly tracked in l_2, i.e., it reflects the fact that the adversary might have corrupted this data. The code then starts a `toLabeled` subcomputation to securely manipulate the labeled taxpayer record without raising its current label. In the subcomputation, we unlabel this labeled record and use function `prepareTaxes` to prepare the tax return. Since we are only concerned with the information-flow aspects of the example, we elide the details of how this function works; our code includes a naive implementation but it would be straightforward to extend it to implement a real-world tax preparation operation. The `toLabeled` block wraps the result in a labeled value r with label l_1, the argument to `toLabeled`. Finally, the preparer stores the labeled tax return r at key `"tax_return"`. Note that this operation would fail if we had not used `toLabeled`, since in that case the current label, raised by the `unlabel` operation, would not flow to ℓ, the label of the adversary.

Figure 5.7 shows the tax agency code on the right. This code fetches the tax return made by the preparer and stored at key `"tax_return"`. Analogously to the preparer code, we use the d value of the fetch operation to specify the target label of the result, namely $\langle \text{IRS}, P \vee C \vee \text{IRS}, S\rangle$, which in this case is once again more restrictive than what is stored in the database. Thereafter the labeled

tax return gets unlabeled and the information is audited in function `verifyReturn`, which returns a boolean that represents whether the declaration is correct. In a more realistic application, this auditing would be performed inside a `toLabeled` block too, but since we are not doing any further `store` operations we let the current label get raised for simplicity.

These three pieces of code are put together in the main function of the program, which we elide for brevity. This function simply generates suitable keystores for the principals involved (using the Clio library function `initializeKeyMapIO`) and then runs the code for each principal using the `evalCLIO` function. As the Clio computations run, it's possible to play the role of the adversary and interact with the database to corrupt or attempt to read the results.

5.6 Related Work

Language-based approaches. Combining cryptography and IFC languages is not new. Vaughan and Zdancewic [81] (and later Smith and Alpízar [74]) consider a version of the *decentralized label model* (DLM) with a symbolic treatment of cryptographic operations, assuming Dolev-Yao style attackers. While such an attacker model does not cover all real-world attacks, the authors establish a correspondence between labels and cryptographic keys. Chothia et al. [20] provide language-level cryptographic primitives as an extension to the DLM, called the keyed-DLM, but do not provide any security properties for their extended system. Their language has two primitives that encrypt (and implicitly declassify) and decrypt values, whereas in Clio values are implicitly encrypted and decrypted when interacting with the store.

Askarov et al. [6] introduce a possibilistic noninterference semantic condition that accounts for *cryptographically-masked flows*: covert information-flow channels due to the cryptosystem (e.g., an observer may distinguish different ciphertexts for the same message). This work ignores the probability distributions for ciphertexts, which might compromise security in some scenarios [51]. In later work, Laud [44] establishes conditions under which secure programs with cryptographically-masked flows are computationally secure, i.e., they satisfy *computational noninterference* [43]. Fournet and Rezk [28] describe a language that directly embeds cryptographic primitives and provide a language-based model of correctness, where cryptographic games are encoded in the language itself so that security can range from symbolic correctness to computational noninterference.

Availability has not been extensively examined as an information flow property. Li et al. [46] discuss the relationship between availability and integrity as information flow properties, and state a (termination- and progress-insensitive) noninterference property for availability. Zheng and Myers [92] extend the DLM with availability policies, which express which principals are trusted to make

data available. They present a semantics that is strict with respect to unavailable values, and prove a noninterference guarantee for availability. In their setting, availability is, in essence, the integrity of progress [7]: low-integrity inputs should not affect the availability of high-availability outputs. In our work, availability tracks the successful verification of signatures and decryption of ciphertexts, and has analogies with Zheng and Myers' approach. Chang et al. [14] also consider how low-integrity inputs affect availability: they use a static dependency analysis to find "inputs of coma" for C programs, i.e., inputs that will cause the program to loop forever.

Systems. Only a few existing IFC tools use cryptography to protect confidentiality and integrity of data, including DStar [91] and Fabric [48] (and its predecessor Jif/Split [89, 93]). DStar is a protocol to extend decentralized IFC in a distributed system. Every DStar node has an *exporter* that is responsible for communicating over the network. Exporters also establish the security categories trusted by a node via private/public keys. Fabric is a platform and statically-checked fine-grained IFC language. Fabric supports the secure transfer of data as well as code [3] through, in part, the use of cryptographic mechanisms. In contrast to Fabric, Clio provides coarse-grained IFC and uses DC labels instead of the DLM. In contrast to both DStar and Fabric, this work establishes a formal basis for security of the use of cryptography in the system. The lack of a formal proof in both DStar and Fabric is not surprising, given that they target more ambitious and complex scenarios (i.e., decentralized information-flow control for distributed systems).

Remote storage. While data can be stored and fetched cryptographically, information can be still leaked through *access patterns*. Private Information Retrieval protocols aim to avoid such leaks by hiding queries and answers from a potentially malicious server [19] similar to Clio's threat model. For performance reasons [63, 73], some approaches rely on a small trusted execution environment provided by hardware [24, 84] that provides the cryptographic support needed to obliviously query the data store [9, 75, 86]. This technique can be seen in oblivious computing [49], online advertising [10], and credit networks [56] for clients which are benign or follow an strict access protocol. If clients are malicious, however, attacker's code may leak information though its access patterns. Besides forcing communication to occur in non-sensitive contexts (as we do in this work), our language-based techniques could be extended to guarantee that untrusted code follows an oblivious protocol.

5.7 Conclusion

Clio is a computationally secure coarse-grained dynamic information-flow control library that uses cryptography to protect the confidentiality and integrity of data. The use of cryptography is hidden

from the language operations and is controlled instead through familiar language constructs in an existing IFC library, LIO. Clio ensures that the system security is not violated by combining attackers' low-level capabilities (i.e., accessing raw bit strings and observing store access patterns) together with their ability to craft their own programs to run on the system. These assurances are shown formally in a computational model through a novel proof technique that combines proof techniques from cryptography and programming languages theory. We provide a prototype implementation of Clio in Haskell, which includes a decentralized cryptographic encoding of DC labels. We see Clio as being essential for providing a foundation for real-world secure IFC systems that, in part, use cryptography as a mechanism to enforce information security guarantees.

6 Conclusion

This dissertation presents three enhancements to software components of large internet-connected applications. These per-component enforcement mechanisms enhance components by performing additional actions that modify the components' behavior. Even if only some of the components are enhanced due to partial deployment, there is still some benefit to the overall system as there are fewer unchecked illegitimate behaviors.

This dissertation takes the approach of enhancing the existing components of systems rather than developing new systems. In doing so we can focus on particular weaknesses in existing systems. In the setting of Whip (Chapter 2), we focus on functional correctness. In the setting of restricted privileges (Chapter 4), we focus on unsafe use of downgrading operations. In the setting of Clio (Chapter 5), we focus on unsafe use of cryptography.

Additionally, the enhancements are adaptable to many existing systems. In the setting of Whip, a Whip adapter supports many message formats and deployment strategies. In the setting of restricted privileges, the restricted privileges can be adapted to any IFC language that uses capability-like privileges to downgrade information (e.g., COWL [79] and Hails [78]). In the setting of Clio, the cryptographic mechanisms can be adapted to any IFC language that interacts with a key-value store.

This dissertation focuses on how programmers write large internet-connected applications. Some of the challenges programmers in this setting face include: ensuring their users' data secrecy and integrity is upheld despite interacting with an untrusted store, and interacting with other components over simple network protocols that do not ensure their desired application-specific behaviors.

The goal of this dissertation is not to show how to solve the exact instances of problems faced by programmers when writing these applications, but instead to give them the linguistic tools to make their job easier and more systematic. The dissertation is not a survey on bug-finding or code patching, though some bugs are presented. Instead, this dissertation presents enhancements that make the process of bug finding and code patching easier for programmers. In that way, this dissertation answers the question of how programmers can write large-scale internet-connected applications despite the complications that arise from dealing with untrusted participants on the internet.

The enhancements were created through a process of first determining the common aspects of componentization that are relevant to the problem, and then inventing a mechanism that solves the problem at those component boundaries. By following that methodology, only the problematic parts of a system are affected in contrast to developing an entirely new system. Further, the solution is applicable to many systems as the enhancements are designed around the common aspects of the component boundaries and not on system-specific details.

The enhancements go a step further to develop solutions that can also operate under partial deployment. The support for partial deployment is particularly important for large internet-connected applications where the governance of components is decentralized. In this setting, it is easier to convince a portion of component owners to subscribe to an (initially partial) improvement to the system than to convince the entire group to subscribe to a new system. When changes to components can not be made in a centralized and coordinated manner, support for partial deployment is required in order to support a feasible migration path for solving the problem in a deployed system.

Instead of changing the way programmers develop large internet-connected applications, this dissertation instead advocates to enhance the existing components programmers already use. In doing so programmers can effectively solve or isolate the particularly hard problems they face without substantially changing the way they write software. Programmers can apply the enhancements where they need them to solve the particular problems they are faced with. As a result, programmers can focus on writing code more efficiently and ultimately build applications that are more trustworthy and usable with little additional effort.

References

[1] *State of Software Security*, volume 6. Veracode, 2015.

[2] Ana Almeida Matos and Gerard Boudol. On declassification and the non-disclosure policy. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 226–240, 2005.

[3] Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. Sharing mobile code securely with information flow control. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 191–205, 2012.

[4] Owen Arden, Jed Liu, and Andrew C. Myers. Flow-limited authorization. In *Proceedings of the IEEE 28th Computer Security Foundations Symposium*, pages 569–583, 2015.

[5] Aslan Askarov and Andrew Myers. A semantic framework for declassification and endorsement. In *Proceedings of the 19th European Symposium on Programming*, 2010.

[6] Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. Cryptographically-masked flows. In *Proceedings of the 13th International Static Analysis Symposium*, August 2006.

[7] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, 2008.

[8] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[9] D. Asonov. *Querying Databases Privately: A New Approach to Private Information Retrieval*. Springer, 2005.

[10] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. Obliviad: Provably secure and practical online behavioral advertising. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[11] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, MITRE Corporation, 1977.

[12] Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. Capabilities for information flow. In *Proceedings of the 6th Workshop on Programming Languages and Analysis for Security*, 2011.

[13] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *Proceedings of the 21st International Conference on Concurrency Theory*, pages 162–176, 2010.

[14] Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, pages 186–199, 2009.

[15] Feng Chen and Grigore Roşu. MOP: An efficient and generic runtime verification framework. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, pages 569–588, 2007.

[16] Winnie Cheng, Dan R.K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in Aeolus. In *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 139–151, 2012.

[17] Stephen Chong and Andrew C. Myers. Language-based information erasure. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 241–254, June 2005.

[18] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, pages 242–256, 2006.

[19] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, 1995.

[20] Tom Chothia, Dominic Duggan, and Jan Vitek. Type-based distributed access control. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, pages 170–186, June 2003.

[21] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Flowfox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM Conference on Computer and communications security*, 2012.

[22] Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, 2008. URL https://rfc-editor.org/rfc/rfc5246.txt.

[23] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: No more scapegoating. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages*, pages 215–226, 2011.

[24] Xuhua Ding, Yanjiang Yang, Robert H. Deng, and Shuhong Wang. A new hardware-assisted PIR with O(n) shuffle cost. *International Journal of Information Security*, 9(4), 2010.

[25] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[26] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, 2002.

[27] Simon Foley, Li Gong, and Xiaolei Qian. A security model of dynamic labeling providing a tiered approach to verification. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 142–158, 1996.

[28] Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 323–335, 2008.

[29] Martin Fowler and James Lewis. Microservices, 2014. URL http://martinfowler.com/articles/microservices.html.

[30] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, October 2012.

[31] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[32] Shafi Goldwasser and Mihir Bellare. *Lecture Notes on Cryptography*, chapter 10. 2001.

[33] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 365–377, 1982.

[34] Sylvain Hallé, Taylor Ettema, Chris Bunch, and Tevfik Bultan. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 235–244, 2010.

[35] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th ACM Symposium on Applied Computing*, 2014.

[36] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 273–284, 2008.

[37] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[38] Raymond Hu, Rumyana Neykova, Nobuko Yoshida, Romain Demangeon, and Kohei Honda. Practical interruptible conversations - distributed dynamic verification with session types and Python. In *Runtime Verification - 4th International Conference*, pages 130–148, 2013.

[39] Limin Jia, Hannah Gommerstadt, and Frank Pfenning. Monitors and blame assignment for higher-order session types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 582–594, 2016.

[40] Matjaz B. Juric. *Business Process Execution Language for Web Services BPEL and BPEL4WS 2nd Edition*. 2006.

[41] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st Symposium on Operating Systems Principles*, October 2007.

[42] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2): 133–169, May 1998.

[43] Peeter Laud. Semantics and program analysis of computationally secure information flow. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, 2001.

[44] Peeter Laud. On the computational soundness of cryptographically masked flows. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 337–348, 2008.

[45] Gary T Leavens, Albert L Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *Software Engineering Notes*, 31(3):1–38, 2006.

[46] Peng Li, Yun Mao, and Steve Zdancewic. Information integrity policies. In *Proceedings of the Workshop on Formal Aspects in Security and Trust*, 2003.

[47] Zheng Li, Yan Jin, and J. Han. A runtime monitoring and validation framework for web service interactions. In *Proceedings of the Australian Software Engineering Conference*, pages 10–79, 2006.

[48] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 321–334, 2009.

[49] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: Practical oblivious computation in a secure processor. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*, 2013.

[50] Heiko Mantel and David Sands. Controlled declassification based on intransitive noninterference. In *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems*, volume 3303, pages 129–145, November 2004.

[51] John McLean. Security models and information flow. In *Proceedings of the IEEE Symposium On Security And Privacy*, pages 180–187, 1990.

[52] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[53] Bertrand Meyer. Design by contract. In *Advances in Object-Oriented Software Engineering*, pages 1–50. 1991.

[54] Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, 1992.

[55] Michael Mimoso. D-Link accidentally leaks private code-signing keys. https://threatpost.com/d-link-accidentally-leaks-private-code-signing-keys/114727/, September 2015.

[56] Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Kim Pecina. Privacy preserving payments in credit networks: Enabling trust with privacy in online marketplaces. In *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[57] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java Information Flow. Software release. http://www.cs.cornell.edu/jif, 2001–.

[58] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 129–142, 1997.

[59] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1998.

[60] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, April 2006.

[61] Object Management Group. CORBA component model. Specification Version 3.3, 2012. URL http://www.omg.org/spec/CORBA/3.3/.

[62] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 8–17, 1988.

[63] Femi G. Olumofin and Ian Goldberg. Revisiting the computational practicality of private information retrieval. In *Proceedings of the 15th International Conference on Financial Cryptography and Data Security*, March 2011.

[64] Rafael Pass and Abhi Shelat. *A Course in Cryptography*, chapter 7. 3rd edition, 2010.

[65] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, pages 46–57, 2000.

[66] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, 1999.

[67] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[68] Alejandro Russo. Functional Pearl: Two can keep a secret, if one of them uses Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pages 280–288, 2015.

[69] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[70] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.

[71] Rafia Shaikh. Microsoft accidentally leaks Xbox Live keys, user data at risk of man-in-the-middle attacks. http://wccftech.com/microsoft-accidentally-leaks-xbox-live-keys/, December 2015.

[72] Randy Shoup. Service architecture at scale: Lessons from Google and eBay. 2015.

[73] Radu Sion and Bogdan Carbunar. On the computational practicality of private information retrieval. In *Proceedings of the Network and Distributed Systems Security Symposium. Stony Brook Network Security and Applied Cryptography Lab Tech Report*, 2007.

[74] Geoffrey Smith and Rafael Alpízar. Secure information flow with random assignment and encryption. In *Proceedings of the 4th ACM Workshop on Formal Methods in Security*, pages 33–44, 2006.

[75] S. W. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM Systems Journal*, 40(3), March 2001.

[76] Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In *Proceedings of the 16th Nordic Conference on Security IT Systems*, pages 223–239, October 2011.

[77] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM symposium on Haskell*, pages 95–106, 2011.

[78] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, September 2012.

[79] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting users by confining JavaScript with COWL. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*, October 2014.

[80] Ron van der Meyden. What, indeed, is intransitive noninterference? In *Proceedings of the 12th European Symposium On Research In Computer Security*, volume 4734, pages 235–250, September 2007.

[81] Jeffrey A. Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 192–206, 2007.

[82] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[83] Jim Waldo. Remote procedure calls and Java remote method invocation. *IEEE Concurrency*, 6(3):5–7, 1998.

[84] Shuhong Wang, Xuhua Ding, Robert H. Deng, and Feng Bao. Private information retrieval using trusted hardware. In *Proceedings of the 11th European Symposium on Research in Computer Security*, pages 49–64, 2006.

[85] Alma Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proceedings of the 8th Conference on USENIX Security Symposium*, 1999.

[86] Peter Williams and Radu Sion. Usable PIR. In *Proceedings of the Network and Distributed System Security Symposium*, 2008.

[87] Alexander Yip, Neha Narula, Maxwell Krohn, and Robert Morris. Privacy-preserving browser-side scripting with BFlow. In *EuroSys*, 2009.

[88] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 15–23, 2001.

[89] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 1–14, October 2001.

[90] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 263–278, 2006.

[91] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 293–308, 2008.

[92] Lantian Zheng and Andrew C. Myers. End-to-end availability policies and noninterference. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 272–286, June 2005.

[93] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 236–250, May 2003.

A

Whip Definitions and Proofs

A.1 Remaining Definitions

$$\mathtt{contract\_for}(\sigma, b) = (n, \checkmark) \text{ if } \exists ee \mapsto \widetilde{l}\checkmark \in \mathtt{blame}_\sigma(\sigma). \\ \mathtt{ep}(ee) = b.y \wedge \mathtt{nm}(ee) = n$$

$$\mathtt{contract\_for}(\sigma, b) = (n, \text{✗}) \text{ if } (\exists ee \mapsto \widetilde{l}\text{✗} \in \mathtt{blame}_\sigma(\sigma). \\ \mathtt{ep}(ee) = b.y \wedge \mathtt{nm}(ee) = n) \\ \wedge (\forall ee' \mapsto \widetilde{l'}c \in \mathtt{blame}_\sigma(\sigma). \text{ if } \mathtt{ep}(ee') = b \\ \text{then } \mathtt{nm}(ee) = k \text{ and } c = \text{✗})$$

$$\mathtt{ep}(a.r\ \mathtt{satisfies}\ n\langle \mathsf{v} \rangle) = a.r$$
$$\mathtt{ep}(a.i\ \mathtt{expects}\ se) = a.i$$
$$\mathtt{nm}(a.r\ \mathtt{satisfies}\ n\langle \mathsf{v} \rangle) = n$$
$$\mathtt{nm}(a.i\ \mathtt{expects}\ se) = \mathtt{nm}(se)$$

$$\frac{}{\mathtt{names}(\boxed{a}) = \{a\}} \qquad \frac{}{\mathtt{names}(\mathtt{mon}^l(\sigma, P^a)) = \{a\}}$$

$$\frac{}{\mathtt{names}(m\ \mathtt{to}\ a) = \emptyset} \qquad \frac{}{\mathtt{names}(\hat{m}\ \mathtt{to}\ a) = \emptyset} \qquad \frac{\widetilde{a_1} = \mathtt{names}(P) \qquad \widetilde{a_2} = \mathtt{names}(Q)}{\mathtt{names}(P \,\|\, Q) = \widetilde{a_1} \cup \widetilde{a_2}}$$

$$\frac{}{\mathtt{labels}(\boxed{a}) = \emptyset} \qquad \frac{}{\mathtt{labels}(\mathtt{mon}^l(\sigma, P^a)) = \{l\}}$$

$$\frac{}{\mathtt{labels}(m\ \mathtt{to}\ a) = \emptyset} \qquad \frac{}{\mathtt{labels}(\hat{m}\ \mathtt{to}\ a) = \emptyset} \qquad \frac{\widetilde{l_1} = \mathtt{labels}(P) \qquad \widetilde{l_2} = \mathtt{labels}(Q)}{\mathtt{labels}(P \,\|\, Q) = \widetilde{l_1} \cup \widetilde{l_2}}$$

A.1.1 Congruence Relation

$$\frac{}{P = P} \qquad \frac{}{P \,\|\, Q = Q \,\|\, P} \qquad \frac{P = P' \qquad Q = Q'}{P \,\|\, Q = P' \,\|\, Q'}$$

$$\frac{\forall (a\ \texttt{satisfies}\ k\langle v\rangle \mapsto \checkmark) \in C.\ \mathcal{C}(a) = k}{C \preceq \mathcal{C}}$$

$$\frac{P \vdash^{\mathcal{C}}_{S} P}{\vdash^{\mathcal{C}}_{S} P\ \mathsf{wf}}$$

$$\frac{\begin{array}{cc} P \vdash^{\mathcal{C}}_{S} Q & P \vdash^{\mathcal{C}}_{S} R \\ \texttt{names}(Q) \cap \texttt{names}(R) = \emptyset & \texttt{labels}(Q) \cap \texttt{labels}(R) = \emptyset \end{array}}{P \vdash^{\mathcal{C}}_{S} Q \parallel R}$$

$$\frac{\begin{array}{ccc} \texttt{conf}_\sigma \preceq \mathcal{C} & l \neq \dagger & S = \texttt{spec}(\sigma) \\ \multicolumn{3}{c}{\forall se.\ se \in dom(\texttt{blame}_\sigma)\ \text{iff}\ se \in dom(\texttt{prov}_\sigma)} \\ \multicolumn{3}{c}{\forall se \mapsto \widetilde{l} \in \texttt{blame}_\sigma,\ l \in \widetilde{l}.\ P \Vdash \textbf{blame}\ l\ \textbf{for}\ se} \\ \multicolumn{3}{c}{\forall \#n\ \ \texttt{from}\ \ b\ \texttt{expects}\ se \mapsto \widetilde{l} \in \texttt{blame}_\sigma,\ l \in \widetilde{l}.\ P \Vdash \textbf{blame}\ l\ \textbf{for}\ se} \\ \multicolumn{3}{c}{\forall a \mapsto n \in \mathcal{C}.\ a\ \texttt{satisfies}\ k\langle \texttt{v}\rangle \mapsto \checkmark \in \texttt{conf}_\sigma} \end{array}}{P \vdash^{\mathcal{C}}_{S} \texttt{mon}^{l}(\sigma, P^{a})}$$

$$\frac{}{P \vdash^{\mathcal{C}}_{S} \boxed{a}} \qquad \frac{\begin{array}{c} m = \texttt{req}\ \ \#n\ \ \texttt{from}\ \ b\ :\ s\ \ \texttt{to}\ \ a\ \text{or} \\ m = \texttt{reply}\ \ \#n\ \ \texttt{from}\ \ b\ :\ s\ \ \texttt{to}\ \ a \end{array}}{P \vdash^{\mathcal{C}}_{S} m\ \ \texttt{to}\ \ a}$$

$$\frac{\begin{array}{cc} k = \mathcal{C}(a) & \{\texttt{identified}(m) \mapsto c\} \preceq \mathcal{C} \\ \forall l \in \widetilde{l}.\ P \Vdash \textbf{blame}\ l\ \textbf{for}\ \texttt{from}(a)\ \texttt{satisfies}\ k\langle \texttt{index}(m)\rangle & \forall l \in \widetilde{l_{id}}.\ P \Vdash \textbf{blame}\ l\ \textbf{for}\ \texttt{identified}(m) \end{array}}{P \vdash^{\mathcal{C}}_{S} m\ \texttt{with}\ \{\texttt{se-blame:=}\widetilde{l};\ \texttt{id-conf:=}c;\ \texttt{id-blame:=}\widetilde{l_{id}}\}\ \ \texttt{to}\ \ b}$$

First, note that we define well-formedness judgment $P \vdash^{\mathcal{C}}_{S} P'$, where P is the entire process, and P' is some subprocess of P. For brevity, we used wellformedness judgment $\vdash^{\mathcal{C}}_{S} P\ \mathsf{wf}$ where P was the entire process, and so treat that as equivalent to $P \vdash^{\mathcal{C}}_{S} P$.

Lemma 1 (Preservation of Well-formedness over congruence)**.** *If $P \vdash^{\mathcal{C}}_{S} P'$ and $P = P''$ and $P' = P'''$ then $P'' \vdash^{\mathcal{C}}_{S} P'''$.*

Proof. The congruence relation only changes the order of the processes. The well-formedness relation is not sensitive to ordering of the processes. □

Lemma 2 (`lift` Preserves Store Well-formedness)**.** If $\mathcal{P}[\texttt{mon}^{l}(\sigma, P^{a})] \vdash^{\mathcal{C}}_{S} \texttt{mon}^{l}(\sigma, P^{a})$ and $\sigma', m\ \texttt{with}\ \{\texttt{se-blame:=}\widetilde{l};\ \texttt{id-conf:=}c;\ \texttt{id-blame:=}\widetilde{l_{id}}\}\ \ \texttt{to}\ \ b = \texttt{lift}_\sigma(k, c, m, l)$ and (if $\texttt{from}(m) \neq a$ then $l = \dagger$), then

$\forall se \mapsto \widetilde{l} \in \texttt{blame}_{\sigma'},\ l' \in \widetilde{l}.\ \mathcal{P}[\texttt{mon}^{l'}(\sigma, P^{a})] \Vdash \textbf{blame}\ l'\ \textbf{for}\ se$ and
$\forall (\#n\ \texttt{from}\ c\ \texttt{expects}\ se) \mapsto \widetilde{l} \in \texttt{blame}_{\sigma'},\ l' \in \widetilde{l}.\ \mathcal{P}[\texttt{mon}^{l'}(\sigma, P^{a})] \Vdash$ **blame** l' **for** se .

Proof. By inspection of the `lift` rule given, there are two entries that can be updated: the service endpoint, and se_{id}. We first show $\forall l' \in \widetilde{l_s}.\ \mathcal{P}[\texttt{mon}^{l}(\sigma', P^{a})] \Vdash \textbf{blame}\ l'\ \textbf{for}\ se$, then $\forall l' \in \widetilde{l_{id}}.\ \mathcal{P}[\texttt{mon}^{l}(\sigma', P^{a})] \Vdash \textbf{blame}\ l'\ \textbf{for}\ se_{id}$.

For the well-formedness of se, we take cases on if $se \in dom(\texttt{blame}_\sigma)$:

- **Case** $se \notin dom(\textbf{blame}_\sigma)$:

  It is the case then that $\widetilde{l_s} = \{l\}$. Due to well-formedness, we have that $\forall se.\ se \in dom(\texttt{blame}_\sigma)$ iff $se \in dom(\texttt{prov}_\sigma)$. It will be the case then that $se \notin dom(\texttt{prov}_\sigma)$ so $pe_s = b$ **intro**. If $b = a$ we can form the following derivation:

  $$\frac{\texttt{prov}_{\sigma'}(se) = a \textbf{ intro}}{\mathcal{P}[\texttt{mon}^l(\sigma', P^a)] \Vdash \textbf{blame } l \textbf{ for } se}$$

  Otherwise we are in a partially deployed case (based on the reduction rules that call `lift`). As a result, in these cases $l_a = \dagger$. So we can form the partial deployment derivation:

  $$\frac{\texttt{prov}_{\sigma'}(se) = b \textbf{ intro}}{\mathcal{P}[\texttt{mon}^l(\sigma', P^a)] \Vdash \textbf{blame } \dagger \textbf{ for } se}$$

- **Case** $se \in dom(\textbf{blame}_\sigma)$:

  Due to well-formedness, $\forall se \in \texttt{blame}_\sigma$, then $\forall l' \in \widetilde{l_s}.\ \mathcal{P}[\texttt{mon}^l(\sigma, P^a)] \Vdash \textbf{blame } l' \textbf{ for } se$. Additionally, since no provenance or blame or provenance information is overwritten the derivation must hold for σ', so, $\forall l' \in \widetilde{l_s}.\ \mathcal{P}[\texttt{mon}^l(\sigma', P^a)] \Vdash \textbf{blame } l' \textbf{ for } re$.

Next we show the well-formedness of se_{id}. We also perform a case analysis on the type of the message together with if $se_{id} \in dom(\texttt{blame}_\sigma)$.

- **Case** $\textbf{type}(m) == \textbf{req} \wedge se_{id} \in dom(\textbf{blame}_\sigma)$: Due to well-formedness of the store, we have that $\forall se \mapsto \widetilde{l} \in \texttt{blame}_\sigma,\ l \in \widetilde{l}.\ \mathcal{P}[\texttt{mon}^l(\sigma, P^a)] \Vdash \textbf{blame } l \textbf{ for } se$, so it must be the case that $\forall l' \in \widetilde{l_{id}}.\ \mathcal{P}[\texttt{mon}^l(\sigma, P^a)] \Vdash \textbf{blame } l' \textbf{ for } se_{id}$. As a result, since the entry is replaced with itself along with its provenance (i.e., no information is replaced), the well-formedness condition still holds.

- **Case** $\textbf{type}(m) == \textbf{req} \wedge se_{id} \notin dom(\textbf{blame}_\sigma)$: It is the case then that $\widetilde{l_{id}} = \{l\}$. Due to well-formedness, we have that $\forall se.\ se \in dom(\texttt{blame}_\sigma)$ iff $se \in dom(\texttt{prov}_\sigma)$. As a result $se_{id} \notin dom(\texttt{prov}_\sigma)$.

  Next we consider if $b = a$. It will be the case that $\texttt{prov}_{\sigma'}(se_{id}) = a$ **intro**. So the derivation for this case is:

  $$\frac{\texttt{prov}_{\sigma'} = a \textbf{ intro}}{\mathcal{P}[\texttt{mon}^l(\sigma', P^a)] \Vdash \textbf{blame } l \textbf{ for } se_{id}}$$

  Otherwise, if $b \neq a$, we are in a partially deployed case (based on the reduction rules that call `lift`). As a result, in these cases $l_a = \dagger$. So we can form the partial deployment derivation:

$$\frac{\texttt{prov}_{\sigma'} = b \textbf{ intro}}{\mathcal{P}[\texttt{mon}^l(\sigma', P^a)] \Vdash \textbf{blame } \dagger \textbf{ for } se_{id}}$$

- **Case** $\texttt{type}(m) == \textbf{reply} \wedge se_{id} \in dom(\textbf{blame}_\sigma)$:

  The same reasoning applies for the case if $\texttt{type}(m) == \texttt{req}$ and $se_{id} \in dom(\texttt{blame}_\sigma)$ as the entry is looked up in the same fashion for requests and replies.

- **Case** $\texttt{type}(m) == \textbf{reply} \wedge se_{id} \notin dom(\textbf{blame}_\sigma)$:

  It is the case then that $\widetilde{l_{id}} = \widetilde{l_s}$. From well-formedness of the store, $\forall l' \in \widetilde{l_s}.\ \mathcal{P}[\texttt{mon}^l(\sigma, P^a)] \Vdash \textbf{blame } l' \textbf{ for } se$. Also from store well-formedness, it must be the case that $se_{id} \notin dom(\texttt{blame}_\sigma)$ as $\forall se.\ se \in dom(\texttt{blame}_\sigma)$ iff $se \in dom(\texttt{prov}_\sigma)$. Because of this, we know that $\texttt{prov}_{\sigma'}(se_{id}) = se \textbf{ intro}$.

  We can directly set up the derivation for an unconfirmed service identification for each $l' \in \widetilde{l_s}$:

$$\frac{\texttt{prov}_{\sigma'}(se_{id}) = (se \textbf{ intro}) \qquad \frac{\textit{(from well-formedness)}}{\mathcal{P}[\texttt{mon}^l(\sigma, P^a)] \Vdash \textbf{blame } l' \textbf{ for } se}}{\mathcal{P}[\texttt{mon}^l(\sigma, P^a)] \Vdash \textbf{blame } l' \textbf{ for } se_{id}}$$

□

Lemma 3 (`lower` Preserves Store Well-formedness)**.** If
$P \vdash^{\mathcal{C}}_{S} \texttt{mon}^l(\sigma', P^a) \,||\, m \,\texttt{with}\, \{\texttt{se-blame:=}\widetilde{l}; \texttt{ id-conf:=}c; \texttt{ id-blame:=}\widetilde{l_{id}}\} \,\texttt{to}\, b$ and
$\sigma', m = \texttt{lower}_\sigma(k, c, m \,\texttt{with}\, \{\texttt{se-blame:=}\widetilde{l}; \texttt{ id-conf:=}c; \texttt{ id-blame:=}\widetilde{l_{id}}\} \,\texttt{to}\, b)$ then
$\forall se \mapsto \widetilde{l} \in \texttt{blame}_{\sigma'},\ l' \in \widetilde{l}.\ \mathcal{P}[\texttt{mon}^{l'}(\sigma, P^a)] \Vdash \textbf{blame } l' \textbf{ for } se$ and
$\forall (\#n \,\texttt{from}\, c \,\texttt{expects}\, se) \mapsto \widetilde{l} \in \texttt{blame}_{\sigma'},\ l' \in \widetilde{l}.\ \mathcal{P}[\texttt{mon}^{l'}(\sigma, P^a)] \Vdash$
blame l' **for** se .

Proof. Since the message is well-formed, there will be blame consistent with provenance for the reply and service identified. Additionally, the provenance information in the receiving adapter is not overwritten so any existing derivations of blame consistent with provenance will still hold. As a result, the services added to the store will still have a valid derivation of blame consistent with provenance. □

Lemma 4 (`lift` and `lower` preserve state)**.** If $\mathcal{P}[\texttt{mon}^l(\sigma, P^a)] \vdash^{\mathcal{C}}_{S} \texttt{mon}^l(\sigma', P^a)$ and either $\sigma', \hat{m} = \texttt{lift}_\sigma(k, c, m, l)$ or $\sigma', m = \texttt{lower}_\sigma(k, c, \hat{m})$, then

1. $S = \texttt{spec}(\sigma')$
2. $\forall se.\ se \in dom(\texttt{blame}_{\sigma'})$ iff $se \in dom(\texttt{prov}_{\sigma'})$
3. $\forall a \mapsto k \in \mathcal{C}.\ a \,\texttt{satisfies}\, k\langle \texttt{v} \rangle \mapsto \checkmark \in \texttt{conf}_{\sigma'}$

Proof. For the first case, $S = \texttt{spec}(\sigma')$, it is clear from the store updates in the metafunctions that the store is not changed.

For the second case, $\forall se.\ se \in dom(\texttt{blame}_{\sigma'})$ iff $re \in dom(\texttt{prov}_{\sigma'})$, it is clear by inspection that the updates to the blame registry and provenance registry affect the same domain of services.

For the third case, $\forall a \mapsto n \in \mathcal{C}.\ a\ \texttt{satisfies}\ k\langle \mathsf{v} \rangle \mapsto \checkmark \in \texttt{conf}_{\sigma'}$, services added to the blame registry do not ever become unconfirmed after becoming confirmed (i.e., note the usage and definition of the `confirmed_or` function). As a result, if σ had this property, then no replacement could make the service become unconfirmed so it will still hold for σ'.

□

Theorem 14 (Preservation of Well-formedness)**.** *If* $P \vdash^{\mathcal{C}}_{S} P$ $P \rightarrow P'$ *then* $P' \vdash^{\mathcal{C}}_{S} P'$.

Proof. By case analysis on step used.

- **Case Black-box Send Request**
  From the definition of the rule, we have

$$\frac{n \text{ fresh} \qquad a \neq b}{\mathcal{P}[\boxed{a}] \longrightarrow \mathcal{P}[\boxed{a} \parallel \texttt{request}\ \#n\ \texttt{from}\ a\ \texttt{containing}\ s\ \texttt{to}\ b]}$$

  where $P = \mathcal{P}[\boxed{a}]$ and $P' = \mathcal{P}[\boxed{a} \parallel \texttt{request}\ \#n\ \texttt{from}\ a\ \texttt{containing}\ s\ \texttt{to}\ b]$. We also have from our congruence rules that $\mathcal{P}[\boxed{a} \parallel \texttt{request}\ \#n\ \texttt{from}\ a\ \texttt{containing}\ s\ \texttt{to}\ b] = \mathcal{P}[\boxed{a}] \parallel \texttt{request}\ \#n\ \texttt{from}\ a\ \texttt{containing}\ s\ \texttt{to}\ b$. We also have that $P \vdash^{\mathcal{C}}_{S} P$. We can re-use the proof derivation of that to construct $P' \vdash^{\mathcal{C}}_{S} \mathcal{P}[\boxed{a}]$. P' can be used in place of P on the left-hand side as it contains the same relevant processes to the blame-consistent-with-provenance relation. We can now construct a new derivation for P'. We can use the base message well-formed rule along with the composition rule to get:

$$\frac{\dfrac{\dots}{P' \vdash^{\mathcal{C}}_{S} \mathcal{P}[\boxed{a}]} \qquad \dfrac{}{P' \vdash^{\mathcal{C}}_{S} \texttt{request}\ \#n\ \texttt{from}\ a\ \texttt{containing}\ s\ \texttt{to}\ b} \quad \begin{array}{c}\texttt{names}(\mathcal{P}[\boxed{a}]) \cap \texttt{names}(\texttt{request}\ \#n\ \texttt{from}\ a\ \texttt{containing}\ s\ \texttt{to}\ b) = \emptyset \\ \texttt{labels}(\mathcal{P}[\boxed{a}]) \cap \texttt{labels}(\texttt{request}\ \#n\ \texttt{from}\ a\ \texttt{containing}\ s\ \texttt{to}\ b) = \emptyset\end{array}}{P' \vdash^{\mathcal{C}}_{S} \mathcal{P}[\boxed{a}] \parallel \texttt{request}\ \#n\ \texttt{from}\ a\ \texttt{containing}\ s\ \texttt{to}\ b}$$

- **Case Black-box Send Reply**
  From the definition of the rule, we have

$$\frac{a \neq b}{\mathcal{P}[\boxed{a}] \longrightarrow \mathcal{P}[\boxed{a} \parallel \texttt{reply}\ \#n\ \texttt{from}\ a\ \texttt{containing}\ s\ \texttt{to}\ b]}$$

where $P = \mathcal{P}[\boxed{a}]$ and $P' = \mathcal{P}[\boxed{a} \parallel \texttt{reply}\ \#n\ \texttt{from}\ a\ \texttt{containing}\ s\ \texttt{to}\ b]$. We also have from our congruence rules that $\mathcal{P}[\boxed{a} \parallel \texttt{reply}\ \#n\ \texttt{from}\ a\ \texttt{containing}\ s\ \texttt{to}\ b] = \mathcal{P}[\boxed{a}] \parallel \texttt{reply}\ \#n\ \texttt{from}\ a\ \texttt{containing}\ s\ \texttt{to}\ b$. We also have that $P \vdash^{\mathcal{C}}_{S} P$. We can re-use the proof derivation of that to construct $P' \vdash^{\mathcal{C}}_{S} \mathcal{P}[\boxed{a}]$. P' can be used in place of P on the left-hand side as it contains the same relevant processes to the blame-consistent-with-provenance relation. We can now construct a new derivation for P'. We can use the base message well-formed rule along with the composition rule to get:

$$\frac{\dfrac{\dots}{P' \vdash^{\mathcal{C}}_{S} \mathcal{P}[\boxed{a}]} \qquad \dfrac{}{P' \vdash^{\mathcal{C}}_{S} \texttt{reply}\ \#n\ \texttt{from}\ a\ \texttt{containing}\ s\ \texttt{to}\ b} \quad \texttt{names}(\mathcal{P}[\boxed{a}]) \cap \texttt{names}(\texttt{reply}\ \#n\ \texttt{from}\ a\ \texttt{containing}\ s\ \texttt{to}\ b) = \emptyset \quad \texttt{labels}(\mathcal{P}[\boxed{a}]) \cap \texttt{labels}(\texttt{reply}\ \#n\ \texttt{from}\ a\ \texttt{containing}\ s\ \texttt{to}\ b) = \emptyset}{P' \vdash^{\mathcal{C}}_{S} \mathcal{P}[\boxed{a}] \parallel \texttt{reply}\ \#n\ \texttt{from}\ a\ \texttt{containing}\ s\ \texttt{to}\ b}$$

- **Case Black-box Receive**
  From the definition of the rule, we have

$$\frac{}{\mathcal{P}[\boxed{a} \parallel m\ \texttt{to}\ a] \rightarrow \mathcal{P}[\boxed{a}]}$$

  where $P = \mathcal{P}[\boxed{a} \parallel m\ \texttt{to}\ a]$ and $P' = \mathcal{P}[\boxed{a}]$. We also have that $P \vdash^{\mathcal{C}}_{S} P$. We can re-use the proof derivation of that to construct $P' \vdash^{\mathcal{C}}_{S} \mathcal{P}[\boxed{a}]$ by removing the part of the derivation that included the rule used for the received message. P' can be used in place of P on the left-hand side as it contains the same relevant processes to the blame-consistent-with-provenance relation. With this, we can now construct a new derivation for P' directly to get $P' \vdash^{\mathcal{C}}_{S} \mathcal{P}[\boxed{a}]$.

- **Case Adapter Send Enhanced**
  From the definition of the rule, we have

$$\frac{(k, \checkmark) = \texttt{contract\_for}_{\sigma}(b, m) \qquad \sigma', \hat{m} = \texttt{lift}_{\sigma}(k, \checkmark, m, l)}{\mathcal{P}[\texttt{mon}^{l}(\sigma, P^{a} \parallel m\ \texttt{to}\ b)] \rightarrow \mathcal{P}[\texttt{mon}^{l}(\sigma', P^{a}) \parallel \hat{m}\ \texttt{to}\ b]}$$

  where $P = \mathcal{P}[\texttt{mon}^{l}(\sigma, P^{a} \parallel m\ \texttt{to}\ b)]$ and $P' = \mathcal{P}[\texttt{mon}^{l}(\sigma', P^{a}) \parallel \hat{m}\ \texttt{to}\ b]$. From Lemmas 2 and 4, we can show that $\mathcal{P}[\texttt{mon}^{l}(\sigma', P^{a})] \vdash^{\mathcal{C}}_{S} \texttt{mon}^{l}(\sigma', P^{a})$. We note that we can use P' in place of $\mathcal{P}[\texttt{mon}^{l}(\sigma', P^{a})]$ as it contains everything but the extra enhanced message, so all derivations will still hold for P'. Since the labels came from the store which is well-formed, along with the confirmation status, the blame premises for the message hold. That is, $\forall l \in \widetilde{l}.\ P \Vdash \textbf{blame}\ l\ \textbf{for}\ b\ \texttt{satisfies}\ k\langle\texttt{index}(m)\rangle$ and $\forall l \in \widetilde{l_{id}}.\ P \Vdash \textbf{blame}\ l\ \textbf{for}\ \texttt{identified}(m)$
  We can now set up the following derivation.

$$\frac{\begin{array}{c}\hat{m} = P \vdash^{\mathcal{C}}_{S} m \texttt{ with } \{\texttt{se-blame}:=\overline{l};\ \texttt{id-conf}:=c;\ \texttt{id-blame}:=\overline{l_{id}}\} \texttt{ to } b \\ k = \mathcal{C}(b) \qquad \texttt{type}(m) == \texttt{req} \\ \{\texttt{identified}(m) \mapsto c\} \preceq \mathcal{C} \qquad \forall l \in \overline{l}.\ P \Vdash \textbf{blame}\ l\ \textbf{for}\ b\ \texttt{satisfies}\ k\langle\texttt{index}(m)\rangle \\ \forall l \in \overline{l_{id}}.\ P \Vdash \textbf{blame}\ l\ \textbf{for}\ \texttt{identified}(m)\end{array}}{P' \vdash^{\mathcal{C}}_{S} \hat{m} \texttt{ to } b} \qquad \frac{}{P' \vdash^{\mathcal{C}}_{S} \texttt{mon}^{l}(\sigma', P^{a})} \qquad \texttt{names}(\hat{m} \texttt{ to } b) \cap \texttt{names}(\texttt{mon}^{l}(\sigma', P^{a})) = \emptyset$$
$$\texttt{labels}(\hat{m} \texttt{ to } b) \cap \texttt{labels}(\texttt{mon}^{l}(\sigma', P^{a})) = \emptyset$$
$$P' \vdash^{\mathcal{C}}_{S} \mathcal{P}[\texttt{mon}^{l}(\sigma', P^{a})] \parallel \hat{m} \texttt{ to } b$$

- **Case Adapter Receive Enhanced**
  From the definition of the rule, we have

$$\frac{(k, \checkmark) = \texttt{contract\_for}_{\sigma}(b, \hat{m}) \qquad \sigma', m = \texttt{lower}_{\sigma}(k, \checkmark, m)}{\mathcal{P}[\texttt{mon}^{l}(\sigma, P^{a}) \parallel \hat{m} \texttt{ to } a] \rightarrow \mathcal{P}[\texttt{mon}^{l}(\sigma', P^{a} \parallel m \texttt{ to } a)]}$$

  where $P = \mathcal{P}[\texttt{mon}^{l}(\sigma, P^{a}) \parallel \hat{m} \texttt{ to } a]$ and $P' = \mathcal{P}[\texttt{mon}^{l}(\sigma', P^{a} \parallel m \texttt{ to } a)]$. Since the enhanced message is well-formed, we can use Lemmas 3 and 4 to get that the new adapter state σ' is well-formed. Additionally, all unenhanced messages are well-formed, so we can use the parallel composition rule to get

$$\frac{\frac{}{P' \vdash^{\mathcal{C}}_{S} m \texttt{ to } a} \qquad \frac{}{P' \vdash^{\mathcal{C}}_{S} \texttt{mon}^{l}(\sigma', P^{a})} \qquad \texttt{names}(\texttt{mon}^{l}(\sigma', P^{a})) \cap \texttt{names}(m \texttt{ to } a) = \emptyset}{\texttt{labels}(\texttt{mon}^{l}(\sigma', P^{a})) \cap \texttt{labels}(m \texttt{ to } a) = \emptyset}$$
$$P' \vdash^{\mathcal{C}}_{S} \mathcal{P}[\texttt{mon}^{l}(\sigma', P^{a}) \parallel m \texttt{ to } a]$$

- **Case Adapter Send Unenhanced**
  From the definition of the rule, we have

$$\frac{(k, \boldsymbol{\times}) = \texttt{contract\_for}_{\sigma}(b, m) \qquad \sigma', \hat{m} = \texttt{lift}_{\sigma}(\boldsymbol{\times}, k, m, l) \qquad \sigma'', m = \texttt{lower}_{\sigma'}(k, \boldsymbol{\times}, \hat{m})}{\mathcal{P}[\texttt{mon}^{l}(\sigma, P^{a} \parallel m \texttt{ to } b)] \rightarrow \mathcal{P}[\texttt{mon}^{l}(\sigma'', P^{a}) \parallel m \texttt{ to } b]}$$

  where $P = \mathcal{P}[\texttt{mon}^{l}(\sigma, P^{a} \parallel m \texttt{ to } b)]$ and $P' = \mathcal{P}[\texttt{mon}^{l}(\sigma'', P^{a}) \parallel m \texttt{ to } b]$. We can use Lemmas 2 to 4 to get that the new adapter state σ'' is well-formed. Additionally, all unenhanced messages are well-formed, so we can use the composition rule to get

$$\frac{\frac{}{P' \vdash^{\mathcal{C}}_{S} m \texttt{ to } b} \qquad \frac{}{P' \vdash^{\mathcal{C}}_{S} \texttt{mon}^{l}(\sigma', P^{a})} \qquad \texttt{names}(\texttt{mon}^{l}(\sigma', P^{a})) \cap \texttt{names}(m \texttt{ to } b) = \emptyset}{\texttt{labels}(\texttt{mon}^{l}(\sigma', P^{a})) \cap \texttt{labels}(m \texttt{ to } b) = \emptyset}$$
$$P' \vdash^{\mathcal{C}}_{S} \mathcal{P}[\texttt{mon}^{l}(\sigma', P^{a}) \parallel m \texttt{ to } b]$$

- **Case Adapter Receive Unenhanced**
  From the definition of the rule, we have

$$\frac{(k, c) = \texttt{contract\_for}_{\sigma}(a, m) \quad \sigma', \hat{m} = \texttt{lift}_{\sigma}(\textbf{x}, m, \dagger) \quad \sigma'', m = \texttt{lower}_{\sigma'}(k, \textbf{x}, \hat{m})}{\mathcal{P}[\texttt{mon}^{l}(\sigma, P^{a}) \parallel m \ \texttt{to} \ a] \rightarrow \mathcal{P}[\texttt{mon}^{l}(\sigma'', P^{a} \parallel m \ \texttt{to} \ a)]}$$

where $P = \mathcal{P}[\texttt{mon}^{l}(\sigma, P^{a}) \parallel m \ \texttt{to} \ a]$ and $P' = \mathcal{P}[\texttt{mon}^{l}(\sigma'', P^{a} \parallel m \ \texttt{to} \ a)]$. We can use Lemmas 2 to 4 to get that the new adapter state σ'' is well-formed. Additionally, all unenhanced messages are well-formed, so we can use the composition rule to get

$$\frac{\dfrac{}{P' \vdash^{\mathcal{C}}_{S} m \ \texttt{to} \ a} \quad \dfrac{}{P' \vdash^{\mathcal{C}}_{S} \texttt{mon}^{l}(\sigma'', P^{a})} \quad \texttt{names}(\texttt{mon}^{l}(\sigma'', P^{a})) \cap \texttt{names}(m \ \texttt{to} \ a) = \emptyset \quad \texttt{labels}(\texttt{mon}^{l}(\sigma'', P^{a})) \cap \texttt{labels}(m \ \texttt{to} \ a) = \emptyset}{P' \vdash^{\mathcal{C}}_{S} \mathcal{P}[\texttt{mon}^{l}(\sigma'', P^{a} \parallel m \ \texttt{to} \ a)]}$$

- **Case Adapter Bypass Send and Receive**
  The adapter state is not changed and all unenhanced messages are well-formed so we can simply use the composition rule to form the new derivation of well-formedness.

With all rules of the reduction relation satisfying preservation of well-formedness, the proof is complete.

□

Theorem 15 (Correct Blame)**.** If well-formed $P_1 = \mathcal{P}_1[\texttt{mon}^{l}(\sigma_1, P_1^{a})]$ and $P_1 \rightarrow P_2$ and $P_2 = \mathcal{P}_2[\texttt{mon}^{l}(\sigma_2, P_2^{a})]$ and $\texttt{errors}_{\sigma_2} = \{le\} \cup \texttt{errors}_{\sigma_1}$ then

1. if $le = \texttt{Pre}(se, l_e)$, then
   (a) if $P_1 = \mathcal{P}[\texttt{mon}^{l}(\sigma_1, P^{a} \parallel m \ \texttt{to} \ b)]$ and $P_2 = \mathcal{P}[\texttt{mon}^{l}(\sigma_2, P^{a}) \parallel m' \ \texttt{to} \ b]$ then $l_e = l$
   (b) if $P_1 = \mathcal{P}[\texttt{mon}^{l}(\sigma_1, P^{a}) \parallel m \ \texttt{to} \ a]$ and $P_2 = \mathcal{P}[\texttt{mon}^{l}(\sigma_2, P^{a} \parallel m \ \texttt{to} \ a)]$ then $l_e = \dagger$
2. if $le = \texttt{Post}(se, \widetilde{l})$, then $\forall l \in \widetilde{l}.\ P_2 \Vdash \textbf{blame}\ l\ \textbf{for}\ se$.

Proof. The first case (precondition error) is a direct result of the reduction rules for receiving a message and the `lift` metafunction for contract checking. The second case (postcondition error) is a direct result of well-formedness (specifically $\forall se \mapsto \widetilde{lc} \in \texttt{blame}_{\sigma'},\ l \in \widetilde{l}.\ P \Vdash \textbf{blame}\ l\ \textbf{for}\ se$) and the `lift` metafunction for contract checking. □

B

Restrictied Privileges Proofs

The integrity and confidentiality lattices are dual lattices, as described below. This fact is used in subsequent proofs.

$$\begin{array}{rcccl} I_1 \sqsubseteq^{\mathrm{I}} I_2 & = & I_1 \Rightarrow I_2 & = & I_2 \sqsubseteq^{\mathrm{C}} I_1 \\ I_1 \sqcup^{\mathrm{I}} I_2 & = & I_1 \vee I_2 & = & I_1 \sqcap^{\mathrm{C}} I_2 \\ I_1 \sqcap^{\mathrm{I}} I_2 & = & I_1 \vee I_2 & = & I_1 \sqcup^{\mathrm{C}} I_2 \end{array}$$

Recall the robust declassification check from Theorem 6:

$$\mathbb{C}(L_{from}) \sqsubseteq^{\mathrm{C}} \mathbb{C}(L_{to}) \sqcup^{\mathrm{C}} \mathbb{I}(L_{pc}) \tag{B.1}$$

and

$$\mathbb{C}(L_{from}) \sqsubseteq^{\mathrm{C}} \mathbb{C}(L_{to}) \sqcup^{\mathrm{C}} \mathbb{I}(L_{from}) \tag{B.2}$$

Proof of Theorem 6. As a first step, we adapt the goal in Definition 3 to $\sqsubseteq^{\mathrm{C}}$ instead of $\sqsubseteq^{\mathrm{I}}$ by way of the lattice duality.

Soundness: We prove that (B.1) and (B.2) implies Definition 3. For that, we assume (B.1), (B.2), and

$$\forall \mathcal{A} \in \mathrm{CNF}\,, \mathbb{C}(L_{to}) \sqsubseteq^{\mathrm{C}} \mathcal{A} \text{ and } \mathbb{C}(L_{from}) \not\sqsubseteq^{\mathrm{C}} \mathcal{A} \tag{B.3}$$

having to prove that

$$\mathbb{I}(L_{pc}) \not\sqsubseteq^{\mathrm{C}} \mathcal{A} \tag{B.4}$$

and

$$\mathbb{I}(L_{from}) \not\sqsubseteq^{\mathrm{C}} \mathcal{A} \tag{B.5}$$

We only prove (B.4) by contradiction (the proof for (B.5) follows analogously). We have that,

$\mathbb{I}(L_{pc}) \sqsubseteq^{\mathrm{C}} \mathcal{A}$	contradiction supposition
$\mathbb{I}(L_{pc}) \sqsubseteq^{\mathrm{C}} \mathcal{A}$ and $\mathbb{C}(L_{to}) \sqsubseteq^{\mathrm{C}} \mathcal{A}$	obtained from (B.3) $\Longrightarrow$
$\mathbb{C}(L_{to}) \sqcup^{\mathrm{C}} \mathbb{I}(L_{pc}) \sqsubseteq^{\mathrm{C}} \mathcal{A}$	least upper bound
$\mathbb{C}(L_{from}) \sqsubseteq^{\mathrm{C}} \mathcal{A}$	from (B.1) and transitivity

Completeness: We show that Definition 3 implies (B.1) and (B.2). Again, we prove only the left side of the conjunct, the right side follows analogously. Consider the contrapositive of the robust declassification condition (see Definition 3) with $\mathcal{A} = \mathbb{C}(L_{to}) \sqcup \mathbb{I}(L_{pc})$, where we disregard the $\mathbb{I}(L_{from})$ half of the antecedent:

$$\begin{array}{l} \text{if } \mathbb{I}(L_{pc}) \sqsubseteq^{\mathrm{C}} \mathbb{I}(L_{pc}) \sqcup^{\mathrm{C}} \mathbb{C}(L_{to}) \\ \text{then } \mathbb{C}(L_{to}) \not\sqsubseteq^{\mathrm{C}} \mathbb{I}(L_{pc}) \sqcup^{\mathrm{C}} \mathbb{C}(L_{to}) \\ \quad \text{or } \mathbb{C}(L_{from}) \sqsubseteq^{\mathrm{C}} \mathbb{I}(L_{pc}) \sqcup^{\mathrm{C}} \mathbb{C}(L_{to}) \end{array}$$

The left side of the disjunction is false by the definition of least upper bound. We conclude that

$\mathbb{C}(L_{from}) \sqsubseteq^{\mathsf{C}} \mathbb{I}(L_{pc}) \sqcup^{\mathsf{C}} \mathbb{C}(L_{to})$ which proves the left side of the conjunct of the robust declassification check.

□

Proof of Theorem 7. Soundness: We show that the robust endorsement condition (see Definition 4) implies the robust endorsement check. Consider the contrapositive of the robust endorsement condition:

$$\begin{aligned}\forall \mathcal{A} \in \text{CNF}.\ \text{if } q \sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{pc}) \\ \text{then } q \sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{to}) \text{ or } q \not\sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{from})\end{aligned}$$

Setting $\mathcal{A} = \mathbb{I}(L_{pc}) \sqcap^{\mathsf{I}} \mathbb{I}(L_{from})$ allows us to conclude that the robustness condition,

$$\mathbb{I}(L_{pc}) \sqcap^{\mathsf{I}} \mathbb{I}(L_{from}) \sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{to})$$

is true because the right side of the disjunction must be false by the greatest upper bound property. *Completeness:* Now we show that the robust endorsement check implies the robust endorsement condition. We take as suppositions the robust endorsement check and the antecedent of the robust endorsement condition:

$$\begin{aligned}&\mathbb{I}(L_{pc}) \sqcap^{\mathsf{I}} \mathbb{I}(L_{from}) \sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{to}) \\ &\forall \mathcal{A} \in \text{CNF}\,,\ \mathcal{A} \not\sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{to}) \text{ and } \mathcal{A} \sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{from})\end{aligned}$$

The proof is by contradiction.

$$\begin{aligned}&\mathcal{A} \sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{pc}) && \text{contradiction supposition} \\ &\mathcal{A} \sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{pc}) \text{ and } \mathcal{A} \sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{from}) \implies && \\ &\quad \mathcal{A} \sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{pc}) \sqcap^{\mathsf{I}} \mathbb{I}(L_{from}) && \text{greatest lower bound} \\ &\mathcal{A} \sqsubseteq^{\mathsf{I}} \mathbb{I}(L_{to}) && \text{transitivity}\end{aligned}$$

Which contradicts the antecedent of the robustness condition. □

C

Clio Definitions and Proofs

C.1 Remaining Definitions

C.1.1 Security Lattice Orderings and Operators

We use three lattices (Confidentiality, Integrity, and Availability) whose domains (named C, I, and A) are formulas of principals in conjuctive normal form. We define an information flow ordering $\sqsubseteq$ (read as "can flow to"). We give the definitions of each below, where $\Rightarrow, \wedge, \vee$ are the usual classical logical connectives:

Confidentiality

$$\begin{aligned} C \sqsubseteq^{C} C' &\iff C' \Rightarrow C \\ C \sqsubset^{C} C' &\iff C \sqsubseteq^{C} C' \text{ and } C \neq C' \\ C \sqcup^{C} C' &\iff C \wedge C' \\ C \sqcap^{C} C' &\iff C \vee C' \\ \bot_C \equiv \mathit{True} & \qquad \top_C \equiv \mathit{False} \end{aligned}$$

Integrity

$$\begin{aligned} I \sqsubseteq^{I} I' &\iff I \Rightarrow I' \\ I \sqsubset^{I} I' &\iff I \sqsubseteq^{I} I' \text{ and } I \neq I' \\ I \sqcup^{I} I' &\iff I \vee I' \\ I \sqcap^{I} I' &\iff I \wedge I' \\ \bot_I \equiv \mathit{False} & \qquad \top_I \equiv \mathit{True} \end{aligned}$$

Availability

$$\begin{aligned} A \sqsubseteq^{A} A' &\iff A \Rightarrow A' \\ A \sqsubset^{A} A' &\iff A \sqsubseteq^{A} A' \text{ and } A \neq A' \\ A \sqcup^{A} A' &\iff A \vee A' \\ A \sqcap^{A} A' &\iff A \wedge A' \\ \bot_A \equiv \mathit{False} & \qquad \top_A \equiv \mathit{True} \end{aligned}$$

We define the security lattice of DC labels as a product lattice of the three individual lattices to form a security lattice whose domain is a triple of principal formulas ($l = \langle C, I, A \rangle$) and whose ordering is based on safe information flows. We also define a trust ordering $\preceq$ (read as "at least as trustworthy as"):

$$
\begin{array}{rcl}
\langle C, I, A\rangle \sqsubseteq \langle C', I', A'\rangle & \iff & C \sqsubseteq^{C} C' \text{ and } I \sqsubseteq^{I} I' \text{ and } A \sqsubseteq^{A} A' \\
\langle C, I, A\rangle \sqsubset \langle C', I', A'\rangle & \iff & C \sqsubset^{C} C' \text{ or } I \sqsubset^{I} I' \text{ or } A \sqsubset^{A} A' \\
\langle C, I, A\rangle \preceq \langle C', I', A'\rangle & \iff & C \sqsubseteq^{C} C' \text{ and } I' \sqsubseteq^{I} I \text{ and } A' \sqsubseteq^{A} A \\
\langle C, I, A\rangle \sqcup \langle C', I', A'\rangle & \iff & \langle\, C \sqcup^{\mathrm{c}} C' \,,\, I \sqcup^{\mathrm{I}} I' \,,\, A \sqcup^{A} A' \,\rangle \\
\langle C, I, A\rangle \sqcap \langle C', I', A'\rangle & \iff & \langle\, C \sqcap^{\mathrm{c}} C' \,,\, I \sqcap^{\mathrm{I}} I' \,,\, A \sqcap^{A} A' \,\rangle \\
\bot \equiv \langle \bot_{C}, \bot_{I}, \bot_{A}\rangle & & \top \equiv \langle \top_{C}, \top_{I}, \top_{A}\rangle
\end{array}
$$

For convenience of avoiding pattern matching over the components of a label when needing to inspect an individual component, we define the following projection functions:

$$
\begin{aligned}
\mathbb{C}(\langle C, I, A\rangle) &= C \\
\mathbb{I}(\langle C, I, A\rangle) &= I \\
\mathbb{A}(\langle C, I, A\rangle) &= A
\end{aligned}
$$

C.1.2 Cryptography Background Definitions

- **Distribution Ensemble**: An ensemble of probability distributions is a sequence $\{X_n\}_{n\in\mathcal{N}}$ of probability distributions.
- **Negligible Function**: A negligible function is a function $f(x) : \mathbb{N} \to \mathbb{R}$ such that for every positive integer $c \in \mathbb{Z}^+$ there exists an integer N_c such that for all $x > N_c$:

$$\left|f(x)\right| < \frac{1}{x^c}$$

- **Computationally Indistinguishable**: Let $\{X_n\}_n$ and $\{Y_n\}_n$ be distribution ensembles. Then we say that they are computationally indistinguishable $\approx$ if for any non-uniform PPT $\mathcal{A}$ the following function is negligible in n:

$$\left|\Pr[\mathcal{A}(x) = 1 \,|\, x \leftarrow X_n] - \Pr[\mathcal{A}(y) = 1 \,|\, y \leftarrow Y_n]\right|$$

.

- **Hybrid Argument**: Let $X_1, \ldots, X_m$ be a sequence of probability distributions, where m is polynomial in n. Suppose there exists a distinguisher D that distinguishes X_1 and X_m with probability ϵ. Then there exists $i \in [1, m-1]$ such that D distinguishes X_i and X_{i+1} with probability $\frac{\epsilon}{m}$. (Proof uses Triangle Inequality)
- **Cryptosystem**: We consider a asymmetric encryption system and signature scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Sign}, \mathsf{Verify})$ such that $\mathsf{Gen} : 1^n \to (\{0,1\}^n, \{0,1\}^n)$ representing the public key and private key to use in the assymetric cryptographic functions.
- **Correctness of Cryptosystem** For correctness of our encryption system, we require that if $p \in \{0,1\}^*$ and $(pk, sk) \leftarrow \mathsf{Gen}(1^n)$ and $c = \mathsf{Enc}(pk, p)$ then $p = \mathsf{Dec}(sk, c)$.

  For correctness of our signature scheme, we require that if $p \in \{0,1\}^*$ and $(pk, sk) \leftarrow \mathsf{Gen}(1^n)$ and $s = \mathsf{Sign}(sk, p)$ then $1 = \mathsf{Verify}(pk, s)$.

  To simplify the notation while mainting easy-to-prove security properties, we require that the encryption and signature functions operate on independent parts of the key; that is, they internally use a key derivation function (e.g., encryption uses the first half, and signing uses the second half).
- **Security of Cryptosystem**: For our encryption functions, we assume the cryptosystem is CPA secure, defined as follows [64]. Let the random variable $\mathrm{IND}_b(\mathcal{A}, n)$ denote the

output of the experiment, where $\mathcal{A}$ is a non-uniform p.p.t., $n \in \mathbb{N}$, $b \in \{0,1\}$:

$$\begin{aligned}\mathrm{IND}_b(\mathcal{A}, n) = {} & (pk, sk) \leftarrow \mathsf{Gen}(1^n); \\ & m_0, m_1, \mathcal{A}_2 \leftarrow \mathcal{A}(pk) \text{ s.t. } |m_0| = |m_1|; \\ & c \leftarrow \mathsf{Enc}(pk, m_b); \\ & \text{Output } \mathcal{A}_2(c)\end{aligned}$$

Then we say that Π is CPA (Chosen-Plaintext Attack) secure if for all non-uniform p.p.t. $\mathcal{A}$:

$$\left\{ \mathrm{IND}_0(\mathcal{A}, n) \right\}_n \approx \left\{ \mathrm{IND}_1(\mathcal{A}, n) \right\}_n$$

Note that we consider the cryptographic functions themselves to be public.

For security of the signature scheme, we assume Π is secure against existential forgery.

- **Indistinguishability Corrollary**: The CPA definition may be difficult to understand to some as it is phrased in the form of a game. An alternative definition of security (that is a fairly direct consequence of CPA security) is: For all p_0, p_1, if $|p_0| = |p_1|$ then,

$$\left\{ \mathsf{Enc}(pk, p_0) \mid (pk, sk) \leftarrow \mathsf{Gen}(1^n) \right\}_n \approx \left\{ \mathsf{Enc}(pk, p_1) \mid (pk, sk) \leftarrow \mathsf{Gen}(1^n) \right\}_n$$

  Informally, the results of encrypting of equal-length plaintexts are computationally indistinguishable.

- **Digital Signature Forgery**: We require our digital signature scheme to be secure against *existential forgery* under a *Chosen-Message Attack*, where the adversary is a non-unfiorm ppt in the size of the key [32]:

  - *Existential Foregery*: The adversary succeeds in forging the signature of one message, not necessarily of his choice.
  - *Chosen-Message Attack*: The adversary is allowed to ask the signer to sign a number of messages of the adversary's choice. The choice of these messages may depend on previously obtained signatures. For example, one may think of a notary public who signs documents on demand.

C.1.3 LIO Complete Syntax and Typing Rules

$$
\begin{array}{rrl}
\text{Ground Value:} & \underline{v} & ::= \mathsf{true} \mid \mathsf{false} \mid () \mid l \mid (\underline{v}, \underline{v}) \\
\text{Value:} & v & ::= \underline{v} \mid (v, v) \mid x \mid \lambda x.t \mid t^{\mathsf{LIO}} \mid \langle v : l \rangle \\
\text{Term:} & t & ::= v \mid (t, t) \mid t\ t \mid \mathsf{fix}\ t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \\
 & & \mid t_1 \sqcup t_2 \mid t_1 \sqcap t_2 \mid t_1 \sqsubseteq t_2 \\
 & & \mid \mathsf{return}\ t \mid t \gg\!= t \\
 & & \mid \mathsf{label}\ t\ t \mid \mathsf{labelOf}\ t \mid \mathsf{unlabel}\ t \\
 & & \mid \mathsf{getLabel} \mid \mathsf{getClearance} \mid \mathsf{lowerClearance}\ t \\
 & & \mid \mathsf{toLabeled}\ t\ t \mid {}^{l}_{l}\{^{l}\ t\ \} \\
\text{Ground Type:} & \underline{\tau} & ::= \mathsf{Bool} \mid () \mid \mathsf{Label} \mid (\underline{\tau}, \underline{\tau}) \\
\text{Type:} & \tau & ::= \underline{\tau} \mid (\tau, \tau) \mid \tau \to \tau \mid \mathsf{LIO}\ \tau \mid \mathsf{Labeled}\ \tau
\end{array}
$$

BOOL
$$\frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash b : \mathsf{Bool}}$$

UNIT
$$\frac{}{\Gamma \vdash () : ()}$$

LABEL
$$\frac{}{\Gamma \vdash l : \mathsf{Label}}$$

PAIR
$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : (\tau_1, \tau_2)}$$

LABELED
$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \langle v : l \rangle : \mathsf{Labeled}\ \tau}$$

VAR
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

ABSTRACTION
$$\frac{\Gamma[x \mapsto \tau_1] \vdash t_1 : \tau_2}{\Gamma \vdash \lambda x.\, t_1 : \tau_1 \to \tau_2}$$

APP
$$\frac{\Gamma \vdash t_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ t_2 : \tau_1}$$

FIX
$$\frac{\Gamma \vdash t : (\tau_1 \to \tau_2) \to \tau_2}{\Gamma \vdash \mathsf{fix}\ t : \tau_1 \to \tau_2}$$

LIO
$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash t^{\mathsf{LIO}} : \mathsf{LIO}\ \tau}$$

IF
$$\frac{\Gamma \vdash t_1 : \mathsf{Bool} \qquad \Gamma \vdash t_2 : \tau \qquad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : \tau}$$

LABELOP
$$\frac{\otimes \in \{\sqcap, \sqcup, \sqsubseteq\} \qquad \Gamma \vdash t_1 : \mathsf{Label} \qquad \Gamma \vdash t_2 : \mathsf{Label}}{\Gamma \vdash t_1 \otimes t_2 : \mathsf{Label}}$$

GETLABEL
$$\frac{}{\Gamma \vdash \mathsf{getLabel} : \mathsf{Label}}$$

GETCLEARANCE
$$\frac{}{\Gamma \vdash \mathsf{getClearance} : \mathsf{Label}}$$

LOWERCLEARANCE
$$\frac{\Gamma \vdash t : \mathsf{Label}}{\Gamma \vdash \mathsf{lowerClearance}\ t : \mathsf{LIO}\ ()}$$

LABELOF
$$\frac{\Gamma \vdash t : \mathsf{Labeled}\ \tau}{\Gamma \vdash \mathsf{labelOf}\ t : \mathsf{Label}}$$

RETURN
$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathsf{return}\ t : \mathsf{LIO}\ \tau}$$

BIND
$$\frac{\Gamma \vdash t_1 : \mathsf{LIO}\ \tau_1 \qquad \Gamma \vdash t_2 : \tau_1 \to \mathsf{LIO}\ \tau_2}{\Gamma \vdash t_1 \gg\!= t_2 : \mathsf{LIO}\ \tau_2}$$

LABEL
$$\frac{\Gamma \vdash t_1 : \mathsf{Label} \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash \mathsf{label}\ t_1\ t_2 : \mathsf{Labeled}\ \tau}$$

UNLABEL
$$\frac{\Gamma \vdash t : \mathsf{Labeled}\ \tau}{\Gamma \vdash \mathsf{unlabel}\ t : \mathsf{LIO}\ \tau}$$

TOLABELED
$$\frac{\Gamma \vdash t_1 : \mathsf{Label} \qquad \Gamma \vdash t_2 : \mathsf{LIO}\ \tau}{\Gamma \vdash \mathsf{toLabeled}\ t_1\ t_2 : \mathsf{Labeled}\ \tau}$$

RESET
$$\frac{\Gamma \vdash t : \mathsf{LIO}\ \tau}{\Gamma \vdash {}^{l_1}_{l_3}\{^{l_2}\ t\} : \mathsf{Labeled}\ \tau}$$

STORE
$$\frac{\Gamma \vdash t_1 : \underline{\tau}' \qquad \Gamma \vdash t_2 : \mathsf{Labeled}\ \underline{\tau}}{\Gamma \vdash \mathsf{store}\ t_1\ t_2 : \mathsf{LIO}\ ()}$$

FETCH
$$\frac{\Gamma \vdash t_1 : \underline{\tau}' \qquad \Gamma \vdash t_2 : \mathsf{Labeled}\ \underline{\tau}}{\Gamma \vdash \mathsf{fetch}_{\underline{\tau}}\ t_1\ t_2 : \mathsf{LIO}\ (\mathsf{Labeled}\ \underline{\tau})}$$

C.1.4 LIO Remaining Step Rules

The program state is $c = \langle l_{\text{cur}}, l_{\text{clr}} \mid t \rangle$ where l_{cur} is the current label, l_{clr} is the current clearance. Computation is modeled as a small-step semantics $c \xrightarrow{\alpha} c'$. We use labels α to represent interaction with the store.

$$
\begin{aligned}
E ::= \; & [\cdot] \mid E\,t \mid \mathsf{if}\ E\ \mathsf{then}\ t\ \mathsf{else}\ t \mid E \gg\!= t \mid \mathsf{label}\ E\,t \\
& \mid E \sqcup t \mid l \sqcup E \mid E \sqcap t \mid l \sqcup E \mid E \sqsubseteq t \mid l \sqsubseteq E \\
& \mid \mathsf{label}\ \underline{v}\ E \mid \mathsf{labelOf}\ E \mid \mathsf{unlabel}\ E \\
& \mid \mathsf{lowerClearance}\ E \mid \mathsf{toLabeled}\ E\,t \mid {}^{l}_{l}\{^{l}\ E\,\} \\
& \mid \mathsf{store}\ E\,t \mid \mathsf{store}\ \underline{v}\ E \mid \mathsf{fetch}_{\tau}\ E\,t \mid \mathsf{fetch}_{\tau}\ \underline{v}\ E
\end{aligned}
$$

APP

$$\frac{}{\langle l_{\text{cur}}, l_{\text{clr}} \mid (\lambda x.t_1)\ t_2\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid [t_2/x]\ t_1\rangle}$$

FIX

$$\frac{}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{fix}\ (\lambda x.t)\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid [\mathsf{fix}\ (\lambda x.t)/x]\ t\rangle}$$

IFTRUE

$$\frac{}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid t_2\rangle}$$

GETLABEL

$$\frac{}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{getLabel}\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{return}\ l_{\text{cur}}\rangle}$$

IFFALSE

$$\frac{}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid t_3\rangle}$$

GETCLEARANCE

$$\frac{}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{getClearance}\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{return}\ l_{\text{clr}}\rangle}$$

LOWERCLEARANCE

$$\frac{l_{\text{cur}} \sqsubseteq l_1 \qquad l_1 \sqsubseteq l_{\text{clr}}}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{lowerClearance}\ l_1\rangle \longrightarrow \langle l_{\text{cur}}, l_1 \mid \mathsf{return}\ ()\rangle}$$

LABELOP

$$\frac{\otimes \in \{\sqcap, \sqcup, \sqsubseteq\} \qquad \underline{v} = l_1 \otimes l_2}{\langle l_{\text{cur}}, l_{\text{clr}} \mid l_1 \otimes l_2\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid \underline{v}\rangle}$$

STEP

$$\frac{\langle l_{\text{cur}}, l_{\text{clr}} \mid t\rangle \longrightarrow \langle l'_{\text{cur}}, l'_{\text{clr}} \mid t'\rangle}{\langle l_{\text{cur}}, l_{\text{clr}} \mid E[t]\rangle \longrightarrow \langle l'_{\text{cur}}, l'_{\text{clr}} \mid E[t']\rangle}$$

Where ℓ_{store} is the store adversary level.

C.1.5 Complete Low Equivalence Relation

Terms:

$$
\begin{array}{lll}
\underline{v}_1 & =_{\ell_{store}} \quad \underline{v}_2 & \text{if } \underline{v}_1 = \underline{v}_2 \\
\langle \underline{v}_1 : l_1 \rangle & =_{\ell_{store}} \quad \langle \underline{v}_2 : l_1 \rangle & \text{where} \\
\multicolumn{3}{l}{\quad \begin{cases} \underline{v}_1 = \underline{v}_2 & \text{if} \quad \mathbb{C}(l_1) \sqsubseteq^C \mathbb{C}(\ell_{store}) \\ & \quad \text{and } \mathbb{I}(l_1) \sqsubset^I \mathbb{I}(\ell_{store}) \\ & \quad \text{and } \mathbb{A}(l_1) \sqsubset^A \mathbb{A}(\ell_{store}), \\ \mathsf{typeOf}\ \underline{v}_1 = \mathsf{typeOf}\ \underline{v}_2 & \text{otherwise} \end{cases}}
\end{array}
$$

$$
\begin{array}{lll}
(v_1, v_2) & =_{\ell_{store}} \quad (v_1', v_2') & \text{if } v_1 =_{\ell_{store}} v_1' \text{ and } v_2 =_{\ell_{store}} v_2' \\
t_1\ t_2 & =_{\ell_{store}} \quad t_1'\ t_2' & \text{if } t_1 =_{\ell_{store}} t_1' \text{ and } t_2 =_{\ell_{store}} t_2' \\
\mathsf{fix}\ t & =_{\ell_{store}} \quad \mathsf{fix}\ t' & \text{if } t =_{\ell_{store}} t' \\
\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 & =_{\ell_{store}} \quad \mathsf{if}\ t_1'\ \mathsf{then}\ t_2'\ \mathsf{else}\ t_3' & \text{if } t_1 =_{\ell_{store}} t_1' \text{ and } t_2 =_{\ell_{store}} t_2' \text{ and } t_3 =_{\ell_{store}} t_3' \\
\mathsf{return}\ t & =_{\ell_{store}} \quad \mathsf{return}\ t' & \text{if } t =_{\ell_{store}} t' \\
t_1 \gg\!= t_2 & =_{\ell_{store}} \quad t_1' \gg\!= t_2' & \text{if } t_1 =_{\ell_{store}} t_1' \text{ and } t_2 =_{\ell_{store}} t_2' \\
\mathsf{label}\ t_1\ t_2 & =_{\ell_{store}} \quad \mathsf{label}\ t_1'\ t_2' & \text{if } t_1 =_{\ell_{store}} t_1' \text{ and } t_2 =_{\ell_{store}} t_2' \\
\mathsf{labelOf}\ t & =_{\ell_{store}} \quad \mathsf{labelOf}\ t' & \text{if } t =_{\ell_{store}} t' \\
\mathsf{unlabel}\ t & =_{\ell_{store}} \quad \mathsf{unlabel}\ t' & \text{if } t =_{\ell_{store}} t' \\
\mathsf{getLabel} & =_{\ell_{store}} \quad \mathsf{getLabel} & \\
\mathsf{getClearance} & =_{\ell_{store}} \quad \mathsf{getClearance} & \\
\mathsf{lowerClearance}\ t & =_{\ell_{store}} \quad \mathsf{lowerClearance}\ t' & \text{if } t =_{\ell_{store}} t' \\
\mathsf{toLabeled}\ t_1\ t_2 & =_{\ell_{store}} \quad \mathsf{toLabeled}\ t_1'\ t_2' & \text{if } t_1 =_{\ell_{store}} t_1' \text{ and } t_2 =_{\ell_{store}} t_2' \\
{}^{l_1}_{l_2}\{^{l_3}\ t\ \} & =_{\ell_{store}} \quad {}^{l_1'}_{l_2'}\{^{l_3}\ t'\ \} & \text{if } l_1 =_{\ell_{store}} l_1' \text{ and } l_2 =_{\ell_{store}} l_2' \text{ and } t =_{\ell_{store}} t' \\
\mathsf{store}\ t_1\ t_2 & =_{\ell_{store}} \quad \mathsf{store}\ t_1'\ t_2' & \text{if } t_1 =_{\ell_{store}} t_1' \text{ and } t_2 =_{\ell_{store}} t_2' \\
\mathsf{fetch}_\tau\ t_1\ t_2 & =_{\ell_{store}} \quad \mathsf{fetch}_\tau\ t_1'\ t_2' & \text{if } t_1 =_{\ell_{store}} t_1' \text{ and } t_2 =_{\ell_{store}} t_2'
\end{array}
$$

Configurations:

$$
\begin{array}{lll}
\langle l_{\text{cur}}, l_{\text{clr}} \mid t \rangle & =_{\ell_{store}} \quad \langle l_{\text{cur}}', l_{\text{clr}}' \mid t' \rangle & \text{if } t =_{\ell_{store}} t' \text{ and } l_{\text{cur}} = l_{\text{cur}}' \text{ and } l_{\text{clr}} = l_{\text{clr}}' \\
\langle l_{\text{cur}}, l_{\text{clr}} \mid t \rangle & =_{\ell_{store}} \quad \langle l_{\text{cur}}', l_{\text{clr}}' \mid t' \rangle & \text{if } (l_{\text{cur}} \not\sqsubseteq \ell_{store} \text{ and} l_{\text{cur}}' \not\sqsubseteq \ell_{store}) \text{ and } (l_{\text{clr}} \not\sqsubseteq \ell_{store} \\
 & & \text{and } l_{\text{clr}}' \not\sqsubseteq \ell_{store})
\end{array}
$$

Confidentiality Low Equivalence

Define $=^C_{\ell_{store}}$ to be the low equivalence relation with respect to only confidentiality. The integrity and availability parts of the label are ignored.

$$
\begin{array}{lll}
\underline{v}_1 & =^{C}_{\ell_{store}} \quad \underline{v}_2 & \text{if } \underline{v}_1 = \underline{v}_2 \\
\langle \underline{v}_1 : l_1 \rangle & =^{C}_{\ell_{store}} \quad \langle \underline{v}_2 : l_1 \rangle & \text{where} \\
\multicolumn{3}{l}{\quad \begin{cases} \underline{v}_1 = \underline{v}_2 & \text{if} \quad \mathbb{C}(l_1) \sqsubseteq^{C} \mathbb{C}(\ell_{store}) \\ \mathsf{typeOf}\ \underline{v}_1 = \mathsf{typeOf}\ \underline{v}_2 & \text{otherwise} \end{cases}}
\end{array}
$$

$$
\begin{array}{lll}
(v_1, v_2) & =^{C}_{\ell_{store}} \quad (v'_1, v'_2) & \text{if } v_1 =^{C}_{\ell_{store}} v'_1 \text{ and } v_2 =_{\ell_{store}} v'_2 \\
t_1\, t_2 & =^{C}_{\ell_{store}} \quad t'_1\, t'_2 & \text{if } t_1 =^{C}_{\ell_{store}} t'_1 \text{ and } t_2 =_{\ell_{store}} t'_2 \\
 & \quad \ldots &
\end{array}
$$

Configurations:

$$
\begin{array}{lll}
\langle l_{\text{cur}}, l_{\text{clr}} \mid t \rangle & =^{C}_{\ell_{store}} \quad \langle l'_{\text{cur}}, l'_{\text{clr}} \mid t' \rangle & \text{if } t =^{C}_{\ell_{store}} t' \text{ and } l_{\text{cur}} = l'_{\text{cur}} \text{ and } l_{\text{clr}} = l'_{\text{clr}} \\
\langle l_{\text{cur}}, l_{\text{clr}} \mid t \rangle & =^{C}_{\ell_{store}} \quad \langle l'_{\text{cur}}, l'_{\text{clr}} \mid t' \rangle & \text{if } \mathbb{C}(l_{\text{cur}}) \not\sqsubseteq^{C} \mathbb{C}(\ell_{store}) \text{ and } \mathbb{C}(l'_{\text{cur}}) \not\sqsubseteq^{C} \mathbb{C}(\ell_{store}) \\
 & & \text{and } \mathbb{C}(l_{\text{clr}}) \not\sqsubseteq^{C} \mathbb{C}(\ell_{store}) \text{ and } l'_{\text{clr}} \not\sqsubseteq^{C} \mathbb{C}(\ell_{store})
\end{array}
$$

C.1.6 Ideal Clio Complete Syntax and Semantics

The ideal Clio state is $\langle c, \sigma \rangle$ where c is the LIO configuration and σ is a mapping $\sigma : \underline{v} \to \text{‹}\underline{v} : l\text{›}_\perp$, where $\perp$ represents a corrupted entry.

Internal-Step

$$\frac{c \longrightarrow c'}{\langle c, \sigma \rangle \rightsquigarrow \langle c', \sigma \rangle}$$

Store

$$\frac{c \xrightarrow{\text{put ‹}\underline{v}:l_1\text{› at } \underline{v}_k} c' \qquad \sigma' = \sigma[\underline{v}_k \mapsto \text{‹}\underline{v} : l_1\text{›}]}{\langle c, \sigma \rangle \rightsquigarrow \langle c', \sigma' \rangle}$$

Fetch-Exists

$$\frac{c \xrightarrow{\text{got}_\tau \text{ ‹}\underline{v}:l_1\text{› at } \underline{v}_k} c' \qquad \text{‹}\underline{v} : l_1\text{›} = \sigma(\underline{v}_k)}{\langle c, \sigma \rangle \rightsquigarrow \langle c', \sigma \rangle}$$

Fetch-Missing

$$\frac{c \xrightarrow{\text{nothing-at } \underline{v}_k} c' \qquad \sigma(\underline{v}_k) = \perp}{\langle c, \sigma \rangle \rightsquigarrow \langle c', \sigma \rangle}$$

Ideal interactions I are given by the following syntax:

$$\begin{aligned}
\bar{I} ::= \;& I \cdot \bar{I} \mid I \\
I ::= \;& \mathsf{skip} = \lambda\sigma.\sigma \\
\mid \;& \mathsf{put}\ \text{‹}\underline{v}:l_1\text{›}\ \mathsf{at}\ \underline{v}' = \lambda\sigma.\ \sigma[\underline{v}' \mapsto \text{‹}\underline{v}:l_1\text{›}] \text{ s.t. } \mathbb{I}(\ell_{store}) \sqsubseteq^I \mathbb{I}(l_1) \\
\mid \;& \mathsf{corrupt}\ \underline{v}_1, \ldots \underline{v}_n = \lambda\sigma.\sigma[\underline{v}_1 \mapsto \perp; \ldots\ \underline{v}_n \mapsto \perp]
\end{aligned}$$

The low-step relation $\curvearrowright_{\bar{I}}$ from ideal Clio configurations to Clio configurations using adversary interaction $\bar{I}$.

Low-Step

$$\frac{\langle c, \bar{I}(\sigma) \rangle \rightsquigarrow \langle c', \sigma' \rangle \qquad \mathrm{PC}(c) \sqsubseteq \ell_{store} \qquad \mathrm{PC}(c') \sqsubseteq \ell_{store}}{(\langle c, \sigma \rangle, \bar{I}) \curvearrowright \langle c', \sigma' \rangle}$$

Low-to-High-to-Low-Step

$$\frac{\langle c, \bar{I}(\sigma) \rangle \rightsquigarrow \langle c_0, \sigma_0 \rangle \qquad \langle c_0, \sigma_0 \rangle \rightsquigarrow \ldots \rightsquigarrow \langle c_j, \sigma_j \rangle \qquad \forall 0 \le i < j.\ \mathrm{PC}(c_i) \not\sqsubseteq \ell_{store} \qquad \mathrm{PC}(c_j) \sqsubseteq \ell_{store}}{(\langle c, \sigma \rangle, \bar{I}) \curvearrowright \langle c_j, \sigma_j \rangle}$$

C.1.7 Complete Definitions for Labeled Value Serialization

- $\mathsf{initialize\_ck}_{\mathcal{P}}(\sigma, C) = (\mathsf{fetch\_ck}_{\mathcal{P}}(\sigma, C), \mathsf{skip})$ if $\mathsf{fetch\_ck}_{\mathcal{P}}(\sigma, C)$ defined

- $\mathsf{initialize\_ck}_{\mathcal{P}}(\sigma, C) = (\mathsf{fetch\_ck}_{\mathcal{P}}(R(\sigma), C), R)$ if $\mathsf{fetch\_ck}_{\mathcal{P}}(\sigma, C)$ undefined and $R = \mathsf{create\_ck}_{\mathcal{P}}(\sigma, C)$

- $\mathsf{create\_ck}_{\mathcal{P}}(\sigma, p_1 \vee \ldots \vee p_n) = \text{store } (p_1 \vee \ldots \vee p_i \vee \ldots \vee p_n)\ (pk, m_n, s)$ where
  $(pk, sk) \leftarrow \mathrm{Gen}(1^n)$
  $m_0 = \emptyset$
  for i from 1 to n:
    $(pk_i, sk_i) = \mathcal{P}(p_i)$
    $m_i \leftarrow m_{i-1}[p_i \mapsto \mathsf{Enc}(pk_i, sk)]$
    $s \leftarrow \mathsf{Sign}(sk_i, (pk, m))$ if $sk_i \neq \bot$

- $\mathsf{fetch\_ck}_{\mathcal{P}}(\sigma, p_1 \vee \ldots \vee p_i \vee \ldots \vee p_n) = (pk, sk)$ where
  $(pk, m, s) = \sigma(p_1 \vee \ldots \vee p_i \vee \ldots \vee p_n)$
  $(pk_i, sk_i) = \mathcal{P}(p_i)$ where i chosen s.t. $sk_i \neq \bot$
  $sk = \mathsf{Dec}(sk_i, m(p_i))$
  $\mathsf{Verify}(pk_j, (pk, m), s) = 1$ for some pk_j in the category.

- $\mathsf{Enc}_{\mathcal{P}}(\sigma_0, \langle C_1 \wedge \ldots \wedge C_i \wedge \ldots \wedge C_n, l_i\rangle, v_0) = (R_n \cdot \ldots \cdot R_1, v_n)$ where
  for i from 1 to n:
    $((pk_i, sk_i), R_i) \leftarrow \mathsf{initialize\_ck}_{\mathcal{P}}(\sigma_{i-1}, C_i)$
    $\sigma_i = R_i(\sigma_{i-1})$
    $v_i \leftarrow \mathsf{Enc}(pk_i, v_{i-1})$

- $\mathsf{Sign}_{\mathcal{P}}(\sigma_0, \langle l_c, C_1 \wedge \ldots \wedge C_n\rangle, v) = (R_n \cdot \ldots \cdot R_1, s_1, \ldots, s_n)$ where
  for i from 1 to n:
    $((pk_i, sk_i), R_i) \leftarrow \mathsf{initialize\_ck}_{\mathcal{P}}(\sigma_{i-1}, C_i)$
    $\sigma_i = R_i(\sigma_{i-1})$
    $s_i \leftarrow \mathsf{Sign}(sk_i, v)$

- $\mathsf{Dec}_{\mathcal{P}}(\sigma, \langle C_1 \wedge \ldots \wedge C_n, l_i\rangle, v_n) = v_0$ where
  for i from n to 1:
    $(pk_i, sk_i) = \mathsf{fetch\_ck}_{\mathcal{P}}(\sigma, C_i)$
    $v_{i-1} = \mathsf{Dec}(sk_i, v_i)$

- $\mathsf{Verify}_{\mathcal{P}}(\sigma, \langle l_c, C_1 \wedge \ldots \wedge C_n\rangle, v, s_1, \ldots, s_n) = 1$ if
  for i from n to 1:
    $(pk_i, sk_i) = \mathsf{fetch\_ck}_{\mathcal{P}}(\sigma, C_i)$
    $1 = \mathsf{Verify}(pk_i, v, s_i)$

Similar to the category key meta-functions, we also annotate the results of the meta-functions with the interactions made on the store so that we can track what actions are being taken on the crypto store.

With these cryptographic functions operating on labels, we are now ready to describe the meta-functions which convert a labeled value to a bit string and vice-versa.

- $\mathsf{serialize}_{\mathcal{P}}(\sigma, \langle \underline{v} : l \rangle) = \{\!|(\overline{R_2} \cdot \overline{R_1}, \langle s : l \rangle) \;\big|\; (\overline{R}, b) \leftarrow \mathsf{Enc}_{\mathcal{P}}(\overline{R_1}(\sigma), l, (\underline{v}, s_1, \ldots, s_n)); (\overline{R_1}, s_1, \ldots, s_n) \leftarrow \mathsf{Sign}_{\mathcal{P}}(\sigma, l, \underline{v})|\!\}$
- $\mathsf{deserialize}_{\mathcal{P}}(\sigma, \langle s : l \rangle, \tau) = \langle \underline{v} : l \rangle$ if $\mathsf{Verify}_{\mathcal{P}}(\sigma, l, \underline{v}, s_1, \ldots, s_n) = 1$ and $\mathsf{Dec}_{\mathcal{P}}(\sigma, l, b) = (\underline{v}, s_1, \ldots, s_n)$ and $\mathsf{typeOf}\ \underline{v} = \tau$

We use the convenience function $\mathrm{pub}(\mathcal{P})$ to represent the projection of the keystore that only contains the public key parts of the keystores, and no private keys.

C.1.8 Real Clio Complete Syntax and Semantics

$$
\begin{array}{rrcl}
\text{Keystores:} & \mathcal{P} & : & p \to (b, b_\perp) \\
\text{Bit strings:} & s & \in & \{0,1\}^* \\
\text{Stores:} & \sigma & : & (\underline{v} \to \langle s : l \rangle_\perp) \cup (C \to ck_\perp) \\
\text{Versions:} & \mathbf{V} & : & \underline{v} \to n \\
\text{Interactions:} & \overline{R} & ::= & R \cdot \overline{R} \mid R \\
 & R & ::= & \mathsf{skip} = \lambda\sigma.\ \sigma \\
 & & \mid & \mathsf{put}\ ck\ \mathsf{at}\ C = \lambda\sigma.\ \sigma[C \mapsto ck] \\
 & & \mid & \mathsf{put}\ \langle s : l \rangle\ \mathsf{at}\ \underline{v}_k = \lambda\sigma.\ \sigma[\underline{v}_k \mapsto \langle s : l \rangle] \\
\text{Strategies:} & \mathcal{S} & : & \mathbb{R} \to \mathbb{R} \\
\text{Category Keys:} & ck & ::= & (b,\ esk,\ s') \\
\text{Encrypted Keys:} & esk & : & p \to s
\end{array}
$$

keystore Label Functions

$$
\begin{array}{lcl}
\mathsf{min}(\mathcal{P}) & = & p_1 \vee \ldots \vee p_i \vee \ldots \vee p_n \text{ for all } p_i \in dom(\mathcal{P}) \\
\mathsf{max}(\mathcal{P}) & = & p_1 \wedge \ldots \wedge p_i \wedge \ldots \wedge p_n \text{ for all } p_i \in dom(\mathcal{P}) \\
 & & \text{and } sk_i \neq \perp \text{ where } \mathcal{P}(p_i) = (pk_i, sk_i) \\
\mathsf{Start}(\mathcal{P}) & = & \langle \mathsf{min}(\mathcal{P}), \mathsf{max}(\mathcal{P}), \mathsf{max}(\mathcal{P}) \rangle \rangle \\
\mathsf{Clr}(\mathcal{P}) & = & \langle \mathsf{max}(\mathcal{P}), \mathsf{min}(\mathcal{P}), \mathsf{min}(\mathcal{P}) \rangle \rangle \\
\mathsf{authorityOf}(\mathcal{P}) & = & \langle \mathsf{max}(\mathcal{P}), \mathsf{max}(\mathcal{P}), \mathsf{max}(\mathcal{P}) \rangle
\end{array}
$$

$$
\begin{array}{rrcl}
\text{Interaction:} & R & ::= & \mathsf{skip} = \lambda\sigma.\ \sigma \\
 & & \mid & \mathsf{put}\ \langle s : l_1 \rangle\ \mathsf{at}\ \underline{v}_k = \lambda\sigma.\ \sigma[\underline{v}_k \mapsto \langle s : l_1 \rangle] \\
 & & \mid & \mathsf{put}\ ck\ \mathsf{at}\ C = \lambda\sigma.\ \sigma[C \mapsto ck]
\end{array}
$$

The interaction concatenation operation $R \cdot R$ sequences interactions. We use the notation $\overline{R} = R \cdot \ldots \cdot R$ to denote a sequence of interactions.

The real Clio state is $\langle c, \mathbb{R}, \mathbf{V} \rangle$ where c is the LIO configuration, σ is the store.

INTERNAL-STEP

$$\frac{c \longrightarrow c'}{\langle c, \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_{\mathrm{I}} \langle c', \mathbb{R}, \mathbf{V}\rangle}$$

STORE

$$\frac{\begin{array}{c} c \xrightarrow{\text{put } \langle \underline{v} : l_1 \rangle \text{ at } \underline{v}_k} c' \\ n = \text{increment}(\mathbf{V}(\underline{v}_k)) \qquad \mathbf{V}' = \mathbf{V}[\underline{v}_k \mapsto n] \\ \mathbb{R}' = \left\{\!\middle|\ \text{put } \langle s : l_1 \rangle \text{ at } \underline{v}_k \cdot \overline{R'} \cdot \overline{R} \;\middle|\; \begin{array}{l} \overline{R} \leftarrow \mathbb{R}; \\ (\overline{R'}, \langle s : l_1 \rangle) \leftarrow \mathsf{serialize}_{\mathcal{P}}(\sigma, \langle (\underline{v}, \underline{v}_k, n) : l_1 \rangle) \end{array} \right.\!\right\} \end{array}}{\langle c, \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_{\mathrm{I}} \langle c', \mathbb{R}', \mathbf{V}'\rangle}$$

FETCH-EXISTS

$$\frac{\begin{array}{c} c \xrightarrow{\text{got } \tau \ \langle \underline{v} : l_1 \rangle \text{ at } \underline{v}_k} c' \qquad n \not< \mathbf{V}(\underline{v}_k) \\ (\sigma, p) \in \left\{\!\middle|\ \overline{R}(\emptyset) \;\middle|\; \overline{R} \leftarrow \mathbb{R}\ \right|\!\right\} \qquad p > 0 \\ \langle (\underline{v}, \underline{v}_k, n) : l_1 \rangle = \mathsf{deserialize}_{\mathcal{P}}(\sigma, \sigma(\underline{v}_k), \tau) \end{array}}{\langle c, \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_{p} \langle c, \mathbb{R}, \mathbf{V}\rangle}$$

FETCH-MISSING

$$\frac{\begin{array}{c} c \xrightarrow{\text{nothing-at } \underline{v}_k} c' \\ (\sigma, p) \in \left\{\!\middle|\ \overline{R}(\emptyset) \;\middle|\; \overline{R} \leftarrow \mathbb{R}\ \right|\!\right\} \qquad p > 0 \\ \mathsf{deserialize}_{\mathcal{P}}(\sigma, \sigma(\underline{v}_k), \tau) \text{ undefined} \end{array}}{\langle c, \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_{p} \langle c, \mathbb{R}, \mathbf{V}\rangle}$$

FETCH-REPLAY

$$\frac{\begin{array}{c} c \xrightarrow{\text{nothing-at } \underline{v}_k} c' \qquad \underline{v}'_k \neq \underline{v}_k \text{ or } n < \mathbf{V}(\underline{v}_k) \\ (\sigma, p) \in \left\{\!\middle|\ \overline{R}(\emptyset) \;\middle|\; \overline{R} \leftarrow \mathbb{R}\ \right|\!\right\} \qquad p > 0 \\ \langle (\underline{v}, \underline{v}'_k, n) : l_1 \rangle = \mathsf{deserialize}_{\mathcal{P}}(\sigma, \sigma(\underline{v}_k), \tau) \end{array}}{\langle c, \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_{p} \langle c, \mathbb{R}, \mathbf{V}\rangle}$$

The low-step relation $\curvearrowright_p$ from real Clio configurations and adversary interactions to Clio configurations with probability p.

Low-Step

$$\frac{\begin{array}{c}\mathbb{R}' = \{\!|\ \overline{R}_A \cdot \overline{R} \mid \overline{R}_A \leftarrow \mathbb{R}_A;\ \overline{R} \leftarrow \mathbb{R}\ |\!\} \\ \langle c, \mathbb{R}', \mathbf{V}\rangle \rightsquigarrow_p \langle c', \mathbb{R}'', \mathbf{V}'\rangle \\ \mathrm{PC}(c) \sqsubseteq \mathbb{C}(\ell_{store}) \qquad \mathrm{PC}(c') \sqsubseteq \mathbb{C}(\ell_{store})\end{array}}{(\langle c, \mathbb{R}, \mathbf{V}\rangle, \mathbb{R}_A) \curvearrowright_p \langle c', \mathbb{R}'', \mathbf{V}'\rangle}$$

Low-to-High-to-Low-Step

$$\frac{\begin{array}{c}\mathbb{R}' = \{\!|\ \overline{R}_A \cdot \overline{R} \mid \overline{R}_A \leftarrow \mathbb{R}_A;\ \overline{R} \leftarrow \mathbb{R}\ |\!\} \\ \langle c, \mathbb{R}', \mathbf{V}\rangle \rightsquigarrow_{p_0} \langle c_0, \mathbb{R}_0, \mathbf{V})\rangle \\ \langle c_0, \mathbb{R}_0, \mathbf{V}_0\rangle \rightsquigarrow_{p_1} \cdots \rightsquigarrow_{p_j} \langle c_j, \mathbb{R}_j, \mathbf{V}_j\rangle \\ \forall_{0 \le i < j}.\ \mathrm{PC}(c_i) \not\sqsubseteq \ell_{store} \qquad \mathrm{PC}(c_j) \sqsubseteq \ell_{store} \qquad p = \Pi_{0 \le i \le j}\, p_i\end{array}}{(\langle c, \mathbb{R}, \mathbf{V}\rangle, \mathbb{R}_A) \curvearrowright_p \langle c_j, \mathbb{R}_j, \mathbf{V}_j\rangle}$$

The step function encodes the distribution of real Clio states after taking j low steps:

Note that, we consider only configurations and strategies that can and always will take at least j low steps for all strategies. That is, there is no possibility for a trace to fail to make a low step before j low steps. As a result, the step function will always produce a distribution (i.e., their probabilities will add up to 1). The program should be written in such a way that it is defensively written to ensure that it can take at least j low steps.

C.2 Complete Theorems and Proofs

C.2.1 Clio Interaction Indistinguishability Lemmas

Lemma 5 (Round 1: Multi-Message Security)**.** *For all* $m^{1}_{\{1,2\}} ... m^{n}_{\{1,2\}}$ *and all principals* $\tilde{p}$*, if* $|m^{\circ}_{i}| = |m^{1}_{i}|$ *for all* $1 \leq i \leq k$*, and* Π *is CPA Secure, then*

$$\left\{ \mathsf{Enc}(pk^{1}_{1}, m^{1}_{1}),\ ...,\ \mathsf{Enc}(pk^{i}_{1}, m^{i}_{1}),\ ...,\ \mathsf{Enc}(pk^{k}_{1}, m^{k}_{1}) \;\middle|\; \mathcal{P} \leftarrow \mathsf{Gen}(\tilde{p}, 1^{n});\ (pk^{i}_{2}, sk^{i}_{2}) \in rng(\mathcal{P});\ 1 \leq i \leq k \right\}_{n}$$
$$\approx$$
$$\left\{ \mathsf{Enc}(pk^{1}_{2}, m^{1}_{2}),\ ...,\ \mathsf{Enc}(pk^{i}_{2}, m^{i}_{2}),\ ...,\ \mathsf{Enc}(pk^{k}_{2}, m^{k}_{2}) \;\middle|\; \mathcal{P} \leftarrow \mathsf{Gen}(\tilde{p}, 1^{n});\ (pk^{i}_{2}, sk^{i}_{2}) \in rng(\mathcal{P});\ 1 \leq i \leq k \right\}_{n}$$

Proof. We perform a proof by contradiction: we assume the consequent does not hold and construct a counter-example to the CPA-security of Π.

The lengths of the sequences of encryptions are equal by setup. The sequences are also polynomial in n as each low step produces a polynomial number of messages and the number of low steps is polynomial in n as k is fixed.

This setup is equivalent to the multi-message CPA security problem. We use the same general technique to show that single-message CPA security. gives rise to multi-message CPA security by using a hybrid argument.

We can define a hybrid sequence of messages

$$H_i = \left\{ m^{1}_{1};\ ...;\ m^{i}_{1};\ m^{i+1}_{2};\ ...;\ m^{k}_{2} \right\}$$

such that at the point i we switch from using the sequence of messages from the first run to using the sequence of messages from the second run. By the hybrid argument, there must exist an i that distinguishes H_i and H_{i+1} with non-negligible probability in n. We will fix on that i. So we can now construct the following CPA adversary:

1. By assumption of our proof by contradiction, there exists a Round 1 adversary that can distinguish H_i and H_{i+1} for a particular i and particular $m^{\{0,1\}}_{1}, ..., m^{\{0,1\}}_{n}$, and call it $\mathcal{A}_{R1}$.

2. In our construction, we generate a new keystore $\mathcal{P}$ as defined in the lemma statement and give the plaintext messages m^{i+1}_{2} and m^{i+1}_{1} to the CPA game and it will then provide us back a ciphertext c of one of the messages.

3. We create a new sequence of encrypted messages in the following way:

   $$H = \left\{ \mathsf{Enc}(pk^{1}_{1}, m^{1}_{1});\ ...;\ \mathsf{Enc}(pk^{i}_{1}, m^{i}_{1});\ c;\ \mathsf{Enc}(pk^{i+2}_{2}, m^{i+2}_{2});\ ...;\ \mathsf{Enc}(pk^{k}_{2}, m^{k}_{2}) \;\middle|\; pk^{o}_{\{1,2\}} \leftarrow \mathsf{Gen}(1^{n}); 1 \leq o \leq k \right\}$$

   In the case where the CPA game chose message m^{i+1}_{2} we have that $H = H_i$ and in the other case where m^{i+1}_{1} we have that $H = H_{i+1}$. Since $\mathcal{A}_{R1}$ can distinguish exactly this case and that

the choice of message encrypted determines which sequence of messages was used, we can as a result distinguish which plain-text message was chosen by the CPA game with non-negligible probability.

As a result of constructing a CPA adversary that can distinguish plain-text messages with non-negligible probability, we have shown a contradiction, and can conclude that the above sequences of encryptions is indistinguishable.

□

Definition 16 (Low Equivalent Interactions)**.** Let $L_{\mathcal{P}}(\overline{R})$ to be a fixed function (i.e., it does not change its behavior based on its inputs) from interactions to interactions such that the result contains the original sequence of interactions with *low interactions* added at statically fixed locations in the sequence. Let the resulting sequences of interactions be called *low equivalent interactions*.

A low interaction is a skip command or a put $\langle b : l_1 \rangle$ at $\underline{v}_k$ such that
$\mathbb{C}(l_1) \sqsubseteq^C \mathbb{C}(\mathsf{authorityOf}(\mathcal{P}))$ and $\langle (\underline{v}, \underline{v}_k, n) : l_1 \rangle = \mathsf{deserialize}_{\mathcal{P}}(\overline{R}, \langle b : l_1 \rangle)$ or put $\tilde{p}'$ at C.

Lemma 6 (Round 2: Secret and Low Equivalent Interactions)**.** *For all keystores $\mathcal{P}_\circ$, and $l_1, \ldots, l_k$, such that $\mathbb{C}(l_i) \sqsubseteq^C \mathbb{C}(\mathsf{authorityOf}(\mathcal{P}))$, and for all $m^1_{\{1,2\}} \ldots m^n_{\{1,2\}}$ and all principals $\tilde{p}$, if $|m^i_1| = |m^i_2|$ for all $1 \leq i \leq k$ and Π is CPA Secure, then*

$$\left\{ L_{\mathcal{P}_\circ}\left(\mathsf{put}\ \langle b^1_1 : l^1 \rangle\ \mathsf{at}\ \underline{v}^1 \cdot \ \ldots\ \cdot \mathsf{put}\ \langle b^k_1 : l^k \rangle\ \mathsf{at}\ \underline{v}^k\right) \mid \mathcal{P} \leftarrow \mathsf{Gen}(1^n);\ (pk^i, sk^i) \in rng(\mathcal{P});\ b^i_1 \leftarrow \mathsf{Enc}(pk^i, m^i_1);\ 1 \leq i \leq k \right\}_n$$
$$\approx$$
$$\left\{ L_{\mathcal{P}_\circ}\left(\mathsf{put}\ \langle b^1_2 : l^1 \rangle\ \mathsf{at}\ \underline{v}^1 \cdot \ \ldots\ \cdot \mathsf{put}\ \langle b^k_2 : l^k \rangle\ \mathsf{at}\ \underline{v}^k\right) \mid \mathcal{P} \leftarrow \mathsf{Gen}(1^n);\ (pk^i, sk^i) \in rng(\mathcal{P});\ b^i_2 \leftarrow \mathsf{Enc}(pk^i, m^i_2);\ 1 \leq i \leq k \right\}_n$$

Proof. We perform a reduction to the Round 1 adversary. That is, if there exists a Round 2 adversary $\mathcal{A}_{R2}$ then there also exists a Round 1 adversary, which will provide a contradiction.

We construct our adversary as follows. Given a sequence of secret encryptions (from Round 1), we can construct the input to the L function by constructing a constant set of labels l_1 to l_n arbitrarily so long as they flow to $\mathbb{C}(\mathsf{authorityOf}(\mathcal{P}))$, which is a static property and also arbitrary input keys. We can then also construct the entry keys v^i in the same static fashion. When then just apply the deterministic L function and pass that to $\mathcal{A}_{R2}$.

Since we have shown how to construct a Round 1 adversary from a Round 2 adversary, we have a contradiction of the Round 1 lemma, so we conclude our proof.

□

Definition 17 (Clio Interactions)**.** Let $m^1_{\{1,2\}}, \ldots, m^k_{\{1,2\}}$ be sequences of messages such that $|m^i_1| = |m^i_2|$. Further, let $\overline{R}^1_{\{1,2\}}, \ldots, \overline{R}^j_{\{1,2\}}$ be sequences of low equivalent interactions (from Definition 2) whose ciphertexts are based on slices of the underlying message. For example

$$\begin{aligned} \overline{R}^1_1 &= \mathsf{put}\ (b, \{p \mapsto \mathsf{Enc}(pk, m^1_1)\}, b_s)\ \mathsf{at}\ C \cdot \mathsf{put}\ \mathsf{Enc}(pk, m^2_1)\ \mathsf{at}\ \underline{v}_1 \\ \overline{R}^2_1 &= \mathsf{skip} \\ \overline{R}^3_1 &= \mathsf{put}\ \mathsf{Enc}(pk', m^3_1)\ \mathsf{at}\ \underline{v}_3 \\ &\ldots \\ \overline{R}^j_1 &= \mathsf{put}\ \mathsf{Enc}(pk'', m^k_1)\ \mathsf{at}\ \underline{v}_j \end{aligned}$$

Then we say two sequences of interactions are *Clio equivalent* $\asymp$ iff they are of the form, with all but negligible probability,

$$\begin{aligned} &\{\, \overline{R}^1_1 \cdot \overline{R}'^2_1 \cdot \ldots \cdot \overline{R}^j_1 \cdot \overline{R}'^j_1 \ \Big|\ \mathcal{P} \leftarrow \mathsf{Gen}(1^n);\ (pk^i, sk^i) \in rng(\mathcal{P});\ b^i_1 \leftarrow \mathsf{Enc}(pk^i, m^i_1);\ 1 \leq i \leq k; \\ &\quad t, v_{\{0,1\}}, \mathcal{S} \leftarrow \mathcal{A}(1^n);\ \overline{R}'^s_1 \leftarrow \mathcal{S}(\overline{R}^s_1 \cdot \ldots \cdot \overline{R}^j_1 \cdot \overline{R}'^j_1);\ 1 < s < j;\ \overline{R}'^j_1 \leftarrow \mathcal{S}(\mathsf{skip}) \,\}_n \\ &\asymp \\ &\{\, \overline{R}^1_2 \cdot \overline{R}'^2_2 \cdot \ldots \cdot \overline{R}^j_2 \cdot \overline{R}'^j_2 \ \Big|\ \mathcal{P} \leftarrow \mathsf{Gen}(1^n);\ (pk^i, sk^i) \in rng(\mathcal{P});\ b^i_2 \leftarrow \mathsf{Enc}(pk^i, m^i_2);\ 1 \leq i \leq k; \\ &\quad t, v_{\{0,1\}}, \mathcal{S} \leftarrow \mathcal{A}(1^n);\ \overline{R}'^s_2 \leftarrow \mathcal{S}(\overline{R}^s_2 \cdot \ldots \cdot \overline{R}^j_2 \cdot \overline{R}'^j_2);\ 1 < s < j;\ \overline{R}'^j_2 \leftarrow \mathcal{S}(\mathsf{skip}) \,\}_n \end{aligned}$$

Lemma 7 (Round 3: Clio Interactions Indistinguishability)**.** *For all families of distributions $\mathbb{R}_1$ and $\mathbb{R}_2$, if $\mathbb{R}_1 \asymp \mathbb{R}_2$ and Π is CPA secure, then $\mathbb{R}_1 \approx \mathbb{R}_2$.*

Proof. Similar to the previous rounds, we will reduce this problem to the Round 2 Secret and Low Equivalent Interactions indistinguishability problem. If there exists a Round 3 adversary $\mathcal{A}_{R_3}$ then there also exists a Round 2 adversary, which will provide a contradiction.

We construct our adversary as follows. Given a sequence of low equivalent interactions (from Round 2) $\overline{R}_1$ and $\overline{R}_2$, we subdivide the interactions into sequences of interactions $\overline{R}^1_{\{1,2\}} \ldots \overline{R}^j_{\{1,2\}}$. We also add the categories in storing category keys arbitrarily in a static way and also add the category key signatures (as it was not in the previous round) by just performing the signing process according to the $\mathsf{initialize_ck}_{\mathcal{P}}$function.

For the strategy interactions, we just perform the draws from the strategy starting from the end of the sequences of interactions, working backwards and place them in their corresponding positions, i.e., $\overline{R}'^s \leftarrow \mathcal{S}(\overline{R}^s \cdot \ldots \cdot \overline{R}'^j \cdot \overline{R}^j$ for $1 < s < j$. We note that, although the interactions may differ they are indistinguishable. That is because the first sequence of interactions $\overline{R}^j_{\{1,2\}}$ is a sub-problem of the Round 2 sequences of interactions. As a result, since the two sequences of interactions are computationally indistinguishable, then their corresponding draws from the strategy are also computationally indistinguishable.

As a result, we can then pass the final sequence of interactions to $\mathcal{A}_{R_2}$ to distinguish the distributions. Since we have shown how to construct a Round 2 adversary from a Round 3 adversary, we have a contradiction of the Round 2 lemma, so we conclude our proof.

□

Lemma 8 (Preservation of Low Equivalence)**.** *For all keystores $\mathcal{P}_0$ where $\ell_{store} =$ $\mathsf{authorityOf}(\mathcal{P}_0)$, LIO configurations c_1, c_2, strategies $\mathcal{S}$, and principals $\tilde{p}$, and $j \in \mathbb{N}$, if Π is CPA Secure and $c_1 =^{C}_{\ell_{store}} c_2$, then*

$$\boldsymbol{Pr}\Big[\langle c_1', \mathbb{R}_1, \boldsymbol{V}_1\rangle \neq^{C}_{\ell_{store}} \langle c_2', \mathbb{R}_2, \boldsymbol{V}_2\rangle \text{ or } \boldsymbol{V}_1 \neq \boldsymbol{V}_2 \;\Big|\; \begin{array}{l} \mathcal{P} \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P}_1 = \mathcal{P}_0 \uplus \mathcal{P};\ \langle c_1', \mathbb{R}_1, \boldsymbol{V}_1\rangle \leftarrow \boldsymbol{step}^{\mathcal{P}_1}_{\ell_{store}}(c_1, \mathcal{S}, j); \\ \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P}_2 = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c_2', \mathbb{R}_2, \boldsymbol{V}_2\rangle \leftarrow \boldsymbol{step}^{\mathcal{P}_2}_{\ell_{store}}(c_2, \mathcal{S}, j)\Big] \end{array}$$

is negligible in n, and

$$\left\{ \overline{R}_1 \;\middle|\; \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c_1', \mathbb{R}_1, \boldsymbol{V}_1\rangle \leftarrow \boldsymbol{step}^{\mathcal{P}}_{\ell_{store}}(c_1, \mathcal{S}, j);\ \overline{R}_1 \leftarrow \mathbb{R}_1 \right\}_n$$
$$\asymp$$
$$\left\{ \overline{R}_2 \;\middle|\; \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c_2', \mathbb{R}_2, \boldsymbol{V}_2\rangle \leftarrow \boldsymbol{step}^{\mathcal{P}}_{\ell_{store}}(c_2, \mathcal{S}, j);\ \overline{R}_2 \leftarrow \mathbb{R}_2 \right\}_n$$

Proof. We will prove this lemma in two steps: first by showing that the invariant is preserved across Clio steps $\rightsquigarrow^*_p$ and then using that fact, we can show that the invariant is also preserved across Clio low steps $\curvearrowright_p$.

Proof on Step relation $\rightsquigarrow_p$: We will perform induction on the derivation of the steps (which will be finite when used with the low-step rules, i.e., it is well-founded) with the number of steps being k being 1 less than the total number of (possibly high) steps in the context of a single low step.

Our inductive hypothesis will be if $\langle c_1, \mathsf{skip}, \mathbf{V}_1\rangle =^{C}_{\ell_{store}} \langle c_2', \mathsf{skip}, \mathbf{V}_2\rangle$ and Π is CPA Secure, then,

$$\mathbf{Pr}\Big[\langle c_1', \mathbb{R}_1, \mathbf{V}_1\rangle \neq^{C}_{\ell_{store}} \langle c_2', \mathbb{R}_2, \mathbf{V}_2\rangle \text{ or } \mathbf{V}_1 \neq \mathbf{V}_2 \;\Big|\; \begin{array}{l} \mathcal{P} \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P}_1 = \mathcal{P}_0 \uplus \mathcal{P};\ \langle c_1, \mathsf{skip}, \Sigma_0\rangle \rightsquigarrow_{p_1} \ldots \rightsquigarrow_{p_k} \langle c_1', \mathbb{R}_1, \mathbf{V}_1\rangle; \\ \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P}_2 = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c_2, \mathsf{skip}, \Sigma_0\rangle \rightsquigarrow_{p_1'} \ldots \rightsquigarrow_{p_{k'}'} \langle c_2', \mathbb{R}_2, \mathbf{V}_2\rangle\Big] \end{array}$$

is negligible in n, and

$$\left\{ \overline{R}_1 \;\middle|\; \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c_1, \mathsf{skip}, \Sigma_0\rangle \rightsquigarrow_{p_1} \ldots \rightsquigarrow_{p_k} \langle c_1', \mathbb{R}_1, \mathbf{V}_1\rangle;\ \overline{R}_1 \leftarrow \mathbb{R}_1 \right\}_n$$
$$\asymp$$
$$\left\{ \overline{R}_2 \;\middle|\; \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c_2, \mathsf{skip}, \Sigma_0\rangle \rightsquigarrow_{p_1'} \ldots \rightsquigarrow_{p_{k'}'} \langle c_2', \mathbb{R}_2, \mathbf{V}_2\rangle;\ \overline{R}_2 \leftarrow \mathbb{R}_2 \right\}_n$$

. We must show as our inductive step that if our inductive hypothesis is true, that the following is true:

$$\mathbf{Pr}\Big[\langle c_1'', \mathbb{R}_1', \mathbf{V}_1'\rangle \neq^{C}_{\ell_{store}} \langle c_2'', \mathbb{R}_2', \mathbf{V}_2'\rangle \;\Big|\; \mathcal{P} \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P}_1 = \mathcal{P}_0 \uplus \mathcal{P};\ \langle c_1, \mathsf{skip}, \Sigma_0\rangle \rightsquigarrow_{p_1} \ldots \rightsquigarrow_{p_k} \langle c_1', \mathbb{R}_1, \mathbf{V}_1\rangle \rightsquigarrow_{p_{k+1}} \langle c_1'', \mathbb{R}_1', \mathbf{V}_1'\rangle;\ \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P}_2 = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c_2, \mathsf{skip}, \Sigma_0\rangle \rightsquigarrow_{p_1'} \ldots \rightsquigarrow_{p_{k'}'} \langle c_2', \mathbb{R}_2, \mathbf{V}_2\rangle \rightsquigarrow_{p_{k'+1}'} \langle c_2'', \mathbb{R}_2', \mathbf{V}_2'\rangle\Big]$$

is negligible in n, and

$$\Big\{\ \overline{R}_1' \;\Big|\; \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c_1, \mathsf{skip}, \Sigma_0\rangle \rightsquigarrow_{p_1} \ldots \rightsquigarrow_{p_k} \langle c_1', \mathbb{R}_1, \mathbf{V}_1\rangle \rightsquigarrow_{p_{k+1}} \langle c_1'', \mathbb{R}_1', \mathbf{V}_1'\rangle;\ \overline{R}_1' \leftarrow \mathbb{R}_1'\ \Big\}_n$$
$$\asymp$$
$$\Big\{\ \overline{R}_2' \;\Big|\; \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c_2, \mathsf{skip}, \Sigma_0\rangle \rightsquigarrow_{p_1'} \ldots \rightsquigarrow_{p_{k'}'} \langle c_2', \mathbb{R}_2, \mathbf{V}_2\rangle \rightsquigarrow_{p_{k'+1}'} \langle c_2'', \mathbb{R}_2', \mathbf{V}_2'\rangle;\ \overline{R}_2' \leftarrow \mathbb{R}_2'\ \Big\}_n$$

The base case $k = 0$ is direct as the initial interactions are a special case of Clio interactions (i.e., $\mathsf{skip} = \mathsf{skip}$) and $\Sigma_0 = \Sigma_0$ and we already know by supposition that $c_1 =^{C}_{\ell_{store}} c_2$.

For the inductive case, we now consider the derivation rule used for the $k+1$'th step and show that it preserves the inductive hypothesis, assuming it for the k'th step.

We note that the single steps may take differing numbers of steps (i.e., k and k'). Due to the Low-to-High-to-Low step rule, though, these differences only occur when l_{cur} is high. As a result, the only invariant we need to preserve is confidentiality-only low equivalence between configurations as the high steps do not change the versions, stores, or interactions. We can appeal to the preservation of low equivalence of LIO proved by Stefan et al. [77] to conclude the preservation confidentiality-only low equivalence of the standard (i.e., non-store and non-fetch) LIO internal steps. We now consider the low steps that affect the non-standard parts of LIO (i.e., the store and fetch commands).

We also note that since there are only a polynomial number of steps that the resulting sequences of configurations from single steps that do not preserve low equivalent in each step will still together be negligible. As a result, we only need to show that the probability of each step not preserving low equivalence is negligible in n. To that end, we will ignore the traces of steps with negligible probabilities that are not low equivalent.

- **Case Fetch (Fetch-Exists, Fetch-Missing, or Fetch-Replay):**

  In this case, both configurations have a term with an evaluation context hole that is at a fetch command. That is, they are both attempting to fetch an entry from the store with key $\underline{v_k}$ and $\underline{v_k'}$. Due to low equivalence they are both fetching the same key, $\underline{v_k} = \underline{v_k'}$. As a result, they must each be using one of the following rules: Fetch-Exists, Fetch-Missing, or Fetch-Replay.

We know by our inductive hypothesis that the distributions of interactions are Clio equivalent. Because the distributions are equal, we can consider steps where the draws are equivalent for values in the erased distributions. We now consider now each case that c_1' transitions with.

- **Case** FETCH-EXISTS and $\mathbb{C}(l_1) \sqsubseteq^C \mathbb{C}(\ell_{store})$: In this case the labeled value will be deserialized the same way and the same labeled value will be fetched. Since the labeled value is readable by the adversary it must be syntactically equivalent with all but negligible probability, otherwise the interactions would be distinguishable which would be a counter-example to Lemma 7. As a result, low equivalence will be preserved and both configurations will transition in the same way with all but negligible probability.
- **Case** FETCH-EXISTS and $\mathbb{C}(l_1) \not\sqsubseteq^C \mathbb{C}(\ell_{store})$: In general the value fetched from the store will vary, or it may be the case that only some of the time a value can even be deserialized. In these cases, the configurations may transition using this rule and other interactions from the distribution may result in it using another rule. However, if c_2' transitions using another fetch rule (FETCH-MISSING or FETCH-REPLAY), the default value will be used. Since secret values can differ and still be low equivalent, the resulting two configurations will still be low equivalent. In each of these cases, no new interactions are produced so the resulting distributions of interactions are still valid Clio interactions by our inductive hypothesis (as they did not change). As a result, low equivalence and the valid interactions invariant is preserved.
- **Case** FETCH-MISSING or FETCH-REPLAY: In these cases, the default labeled value will be used, which by our inductive hypothesis is already low equivalent (due to the configurations being low equivalent). The other configurations will transition in a symmetric way described for the FETCH-EXISTS rule.

- **Case** STORE**:**

In this case the distribution of stores and interactions will change so we must show that they remain equivalent. That is, we must show that $\{ \overline{R}_1' \mid \overline{R}_{\leftarrow}' \mathbb{R}_1' \}_n \asymp \{ \overline{R}_2' \mid \overline{R}_2' \leftarrow \mathbb{R}_2' \}$ where

$$\mathbb{R}_1' = \left\{ \begin{array}{l} \mathsf{put}\ \langle s_1 : l_1 \rangle\ \mathsf{at}\ \underline{v}_k \cdot \overline{R'}_1 \cdot \overline{R}_1 \\ (\overline{R'}_1, \langle s_1 : l_1 \rangle) \leftarrow \mathsf{serialize}_{\mathcal{P}}(\sigma_1, \langle (\underline{v}, \underline{v}_k, n_1) : l_1 \rangle) \end{array} \middle| \begin{array}{l} \\ \overline{R}_1 \leftarrow \mathbb{R}_1; \\ \\ \end{array} \right\}$$

$$\mathbb{R}_2' = \left\{ \begin{array}{l} \mathsf{put}\ \langle s_2 : l_1 \rangle\ \mathsf{at}\ \underline{v}_k \cdot \overline{R'}_2 \cdot \overline{R}_2 \\ (\overline{R'}_2, \langle s_2 : l_1 \rangle) \leftarrow \mathsf{serialize}_{\mathcal{P}}(\sigma_2, \langle (\underline{v}, \underline{v}_k, n_2) : l_1 \rangle) \end{array} \middle| \begin{array}{l} \\ \overline{R}_2 \leftarrow \mathbb{R}_2; \\ \\ \end{array} \right\}$$

We first note that the entry keys are the same from low equivalence. The versions are equal from Clio low equivalence, i.e., $n_1 = n_2$. We also note that the distributions of interactions are valid Clio interactions from our inductive hypothesis. For readable labeled values, we can conclude that they are syntactically equivalent values from low equivalence. For non-readable

values, the types of the secret values will be the same due to low equivalence (and so the serialized plaintext message will have the same length). As a result the put $\langle b_{\{0,1\}} : l_1 \rangle$ at $\underline{v_k}$ will be a valid extension of valid Clio interactions.

We next consider the creation of category keys (i.e., $\overline{R'}_1$ and $\overline{R'}_2$). The initialization of category keys will behave the same way as described for fetching a labeled value: it will either create new keys (if they were corrupted or not there), or skip. It will do this in the same way as the resulting interactions are indistinguishable. For the contents of the category keys, we can divide the parts of the category into deterministic parts (i.e., from Lemma 6) and secret encryptions.

With these considerations, we conclude that with all but negligible probability the resulting interactions will be valid Clio interactions.

Finally, for the versions mappings, we note that they are both updated equivalently (i.e., incremented by the version in the mapping) and the versions mappings originally were equal, so the resulting versions are equal.

With all cases of the reduction shown to satisfy the proof obligation, we can conclude the inductive hypothesis is true for all steps used in the context of a single low step. We next show the low equivalence invariant on the low step relation.

Proof on Low-step relation $\curvearrowright_p$: by induction on the number of low steps j. Our inductive hypothesis will match our lemma. For all keystores $\mathcal{P}_0$ where $\ell_{store} = \mathsf{authorityOf}(\mathcal{P}_0)$, LIO configurations c_1, c_2, strategies $\mathcal{S}$, and principals $\tilde{p}$, and $j \in \mathbb{N}$, if Π is CPA Secure and $c_1 =^C_{\ell_{store}} c_2$, then

$$\mathbf{Pr}\Big[\langle c'_1, \mathbb{R}_1, \mathbf{V}_1\rangle \neq^C_{\ell_{store}} \langle c'_2, \mathbb{R}_2, \mathbf{V}_2\rangle \text{ or } \mathbf{V}_1 \neq \mathbf{V}_2 \;\Big|\; \mathcal{P} \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P}_1 = \mathcal{P}_0 \uplus \mathcal{P};\ \langle c'_1, \mathbb{R}_1, \mathbf{V}_1\rangle \leftarrow \mathbf{step}^{\mathcal{P}_1}_{\ell_{store}}(c_1, \mathcal{S}, j);\ \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P}_2 = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c'_2, \mathbb{R}_2, \mathbf{V}_2\rangle \leftarrow \mathbf{step}^{\mathcal{P}_2}_{\ell_{store}}(c_2, \mathcal{S}, j)\Big]$$

is negligible in n, and

$$\Big\{\ \overline{R}_1 \ \Big|\ \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c'_1, \mathbb{R}_1, \mathbf{V}_1\rangle \leftarrow \mathbf{step}^{\mathcal{P}}_{\ell_{store}}(c_1, \mathcal{S}, j);\ \overline{R}_1 \leftarrow \mathbb{R}_1\ \Big\}_n \asymp \Big\{\overline{R}_2 \ \Big|\mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c'_2, \mathbb{R}_2, \mathbf{V}_2\rangle \leftarrow \mathbf{step}^{\mathcal{P}}_{\ell_{store}}(c_2, \mathcal{S}, j);\ \overline{R}_2 \leftarrow \mathbb{R}_2\ \Big\}_n$$

- **Base Case: $j = 1$:** That is, we will prove the following:

$$\mathbf{Pr}\Big[\langle c'_1, \mathbb{R}_1, \mathbf{V}_1\rangle \neq^C_{\ell_{store}} \langle c'_2, \mathbb{R}_2, \mathbf{V}_2\rangle \text{ or } \mathbf{V}_1 \neq \mathbf{V}_2 \;\Big|\; \mathcal{P} \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P}_1 = \mathcal{P}_0 \uplus \mathcal{P};\ \langle c'_1, \mathbb{R}_1, \mathbf{V}_1\rangle \leftarrow \mathbf{step}^{\mathcal{P}_1}_{\ell_{store}}(c_1, \mathcal{S}, 1);\ \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P}_2 = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c'_2, \mathbb{R}_2, \mathbf{V}_2\rangle \leftarrow \mathbf{step}^{\mathcal{P}_2}_{\ell_{store}}(c_2, \mathcal{S}, 1)\Big]$$

is negligible in n, and

$$\left\{ \overline{R}_1 \;\middle|\; \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c_1', \mathbb{R}_1, \mathbf{V}_1 \rangle \leftarrow \mathbf{step}^{\mathcal{P}}_{\ell_{store}}(c_1, \mathcal{S}, 1);\ \overline{R}_1 \leftarrow \mathbb{R}_1 \right\}_n$$
$$\asymp$$
$$\left\{ \overline{R}_2 \;\middle|\; \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c_2', \mathbb{R}_2, \mathbf{V}_2 \rangle \leftarrow \mathbf{step}^{\mathcal{P}}_{\ell_{store}}(c_2, \mathcal{S}, 1);\ \overline{R}_2 \leftarrow \mathbb{R}_2 \right\}_n$$

We must show $\langle c_1', \mathbb{R}_1, \mathbf{V}_1 \rangle =^{C}_{\ell_{store}} \langle c_2', \mathbb{R}_2, \mathbf{V}_2 \rangle$ or $\mathbf{V}_1 \neq \mathbf{V}_2$. There are two cases we must consider in the low step relation, the Low-Step rule and the Low-to-High-to-Low-Step rule. In the Low-Step, we must show that the inductive hypothesis holds after a single Clio step $\rightsquigarrow_p$, and for the Low-to-High-to-Low-Step rule must hold for many (finite) Clio steps $\rightsquigarrow^*_p$. Note that the Low-Step rule is a special case of the Low-to-High-to-Low-Step rule so we only consider the more general case of preserving the invariant across many steps. To show this, we appeal to the previous proof made to show that the invariant is preserved across Clio steps.

Unlike the single step relation, this includes a strategy interaction on the distribution of stores. Since both interactions receive the indistinguishable distributions of interactions (from Lemma 7 and the inductive hypothesis) so the resulting distributions from the strategy will also be computationally indistinguishable. That is because if they were not, then the strategy itself could be used as a counter-example for Lemma 7. In sum, the resulting strategy invocation results in a valid sequence of Clio interactions.

From the previous proof on the single-step relation, we can conclude that $\langle c_1', \mathbb{R}_1, \mathbf{V}_1 \rangle =^{C}_{\ell_{store}} \langle c_2', \mathbb{R}_2, \mathbf{V}_2 \rangle$. As a result, we satisfy the inductive hypothesis.

- **Inductive Case** $j = k + 1$: That is, we will prove the following:

$$\mathbf{Pr}\Big[\langle c_1', \mathbb{R}_1, \mathbf{V}_1 \rangle \neq^{C}_{\ell_{store}} \langle c_2', \mathbb{R}_2, \mathbf{V}_2 \rangle \text{ or } \mathbf{V}_1 \neq \mathbf{V}_2 \;\Big|\; \mathcal{P} \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P}_1 = \mathcal{P}_0 \uplus \mathcal{P};\ \langle c_1', \mathbb{R}_1, \mathbf{V}_1 \rangle \leftarrow \mathbf{step}^{\mathcal{P}_1}_{\ell_{store}}(c_1, \mathcal{S}, k{+}1);\ \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P}_2 = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c_2', \mathbb{R}_2, \mathbf{V}_2 \rangle \leftarrow \mathbf{step}^{\mathcal{P}_2}_{\ell_{store}}(c_2, \mathcal{S}, k{+}1) \Big]$$

is negligible in n, and

$$\left\{ \overline{R}_1 \;\middle|\; \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c_1', \mathbb{R}_1, \mathbf{V}_1 \rangle \leftarrow \mathbf{step}^{\mathcal{P}}_{\ell_{store}}(c_1, \mathcal{S}, k{+}1);\ \overline{R}_1 \leftarrow \mathbb{R}_1 \right\}_n$$
$$\asymp$$
$$\left\{ \overline{R}_2 \;\middle|\; \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c_2', \mathbb{R}_2, \mathbf{V}_2 \rangle \leftarrow \mathbf{step}^{\mathcal{P}}_{\ell_{store}}(c_2, \mathcal{S}, k{+}1);\ \overline{R}_2 \leftarrow \mathbb{R}_2 \right\}_n$$

We now must show that, for any low equivalent configurations that the resulting single step will remain low equivalent. We can use the same reasoning from the base case to show that

the adversary interaction preserves equivalence on distributions of stores. After this adversary interaction, we can invoke the single-step lemma result here to conclude that $c_1'' =_{\ell_{store}} c_2''$.

With the single low step relation handled we now must consider the distribution of distributions of interactions from the **step** function. For example, it may be the case that a particular distribution of interactions generated from one trace of low steps may be much more likely than another distribution of interactions. However, we can use the preservation of low equivalence to reason about the probabilities of corresponding low equivalent distributions of interactions. We consider the distribution formed from the step function after 1 low step as a running example to make our arguments concrete, shown graphically in the main matter in Section 5.4.1.

From our inductive hypothesis we know that corresponding low equivalent configurations have indistinguishable distributions of interactions. We now must consider the relationship between the probabilities that led to the corresponding low equivalent configurations (e.g., from the diagram p_1 and p_2, and also p_1' and p_2'). If they are similar, then the resulting draws from the distributions will be similar (from low equivalence).

Consider the pairs of low equivalent configurations and the probabilities that led to those configurations (e.g., from the diagram $\langle c_1', \mathbb{R}_1, \mathbf{V}_1 \rangle$ with probability p_1 and $\langle c_2', \mathbb{R}_2, \mathbf{V}_2 \rangle$ with probability p_2). Consider the ways the configurations can differ probabilistically (e.g., from the diagram, how c_1 steps to both c_1' and c_1'' and how c_2 steps to both c_2' and c_2''). The low step relation is just the probability of the trace of single steps leading to the next low Clio configuration. The Store and Internal-Step rules take steps with probability 1 so they will not cause the low step to differ probabilistically.

Indeed, only the fetching rules Fetch-Exists, Fetch-Missing, and Fetch-Replay rules will cause the configurations to differ probabilistically. In particular they will differ based on the interactions drawn, and as a result differ on how those interactions affect the fetch: if the entry is missing or not deserializeable (Fetch-Missing), if the value can be deserialized but the version is old (Fetch-Replay), or if it was successfully deserialized and the version is not old (Fetch-Valid).

Due to our inductive hypothesis we know that the distributions of interactions are valid Clio interactions and as a result are indistinguishable from Lemma 7. For readable labeled values, the configurations will step with the same probability in lock-step with all but negligible probability, as the readable labeled values will be syntactically equivalent (as the distributions of erased stores are equivalent).

In the case where the label of the labeled value is not readable, the rules used to step may not be the same as they are the results of encrypted values. For example, in one configuration a labeled value may be successfully fetched (using Fetch-Valid) but not in the corresponding configuration (e.g., Fetch-Missing was used). However, as noted above and by our inductive hypothesis, the different rules used will all step to a low equivalent configuration. In addition, though, to the configurations being low equivalent, it is also the case that the sums of the probabilities of all steps taken will be equivalent with all but negligible probability. For example, if c_1 steps using Fetch-Missing with probability p_1, and Fetch-Valid with probability p_2, it is also the case that c_2 will use the same rules Fetch-Missing with probability p_1 and Fetch-Valid with probability p_2 due to indistin-

guishability of the interactions. That is because if it did not, then an adversary could be constructed to distinguish the interactions based on the proportions of rules used by the Clio semantics. Intuitively, the draws of indistinguishable interactions will produce distributions of indistinguishable steps.

With this reasoning, we conclude that the probabilities of each corresponding single step taking place will be equal (e.g., in the diagram above, $p_1 = p_2$ and $p_2 = p'_2$). So, the resulting distribution of distributions over interactions will be still be valid Clio interactions and so the $\asymp$ relation holds (and, by Lemma 7, they are also indistinguishable as a result).

□

Definition 18 (Chosen-Term Attack (CTA) Game)**.** *Let the random variable* $\mathrm{IND}_b(\mathcal{P}, \mathcal{A}, \tilde{p}, j, n)$ *denote the output of the following experiment, where* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Sign}, \mathsf{Verify})$*,* $\mathcal{A}$ *is a non-uniform ppt,* $n \in \mathbb{N}$*,* $b \in \{0, 1\}$*:*

$$
\begin{array}{l}
\mathrm{IND}_b(\mathcal{P}_0, \mathcal{A}, \tilde{p}, j, n) = \\
\quad \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}'; \\
\quad t, v_0, v_1, \mathcal{S}, \mathcal{A}_2 \leftarrow \mathcal{A}(\mathrm{pub}(\mathcal{P}))\ \textit{such that}\ v_0 =^{C}_{\ell_{store}} v_1 \\
\qquad\qquad\qquad\qquad\qquad \textit{and}\ \vdash t : \mathsf{Labeled}\ \tau \to \mathsf{LIO}\ \tau' \\
\qquad\qquad\qquad\qquad\qquad \textit{and}\ \vdash v_0 : \mathsf{Labeled}\ \tau \\
\qquad\qquad\qquad\qquad\qquad \textit{and}\ \vdash v_1 : \mathsf{Labeled}\ \tau \\
\qquad\qquad\qquad\qquad\qquad \textit{and}\ \ell_{store} = \mathsf{authorityOf}(\mathcal{P}_0); \\
\quad \langle c, \mathbb{R}_b, \boldsymbol{V}' \rangle \leftarrow \boldsymbol{step}^{\mathcal{P}}_{\ell_{store}}(\langle \mathsf{Start}(\mathcal{P}), \mathsf{Clr}(\mathcal{P}) \mid (t\ v_b) \rangle, \mathcal{S}, j); \\
\quad \overline{R_b} \leftarrow \mathbb{R}_b;\ \mathrm{Output}\ \mathcal{A}_2(\overline{R_b})
\end{array}
$$

We say that Clio using Π *is CTA (Chosen-Term Attack) Secure if for all non-uniform ppt* $\mathcal{A}$*,* $j \in \mathbb{N}$*, keystores* $\mathcal{P}$*, and principals* $\tilde{p}$*:*

$$
\left\{ \mathrm{IND}_0(\mathcal{P}, \mathcal{A}, \tilde{p}, j, n) \right\}_n \approx \left\{ \mathrm{IND}_1(\mathcal{P}, \mathcal{A}, \tilde{p}, j, n) \right\}_n
$$

.

Theorem 19 (Indistinguishability Theorem)**.** *If* Π *if CPA Secure, then Clio using* Π *is CTA Secure.*

Proof. Direct result of low equivalence (interactions are valid Clio interactions, i.e., they satisfy the $\asymp$ relation) and Lemma 7 (indistinguishability of valid Clio interactions).

□

C.2.4 Leveraged Forgery Lemmas

Lemma 9 (Starting Label is a Floor)**.** *For all keystores $\mathcal{P}_0$ and terms t and strategies $\mathcal{S}$ and principals $\tilde{p}$ and j,*

$$\boldsymbol{Pr}\Big[\ \mathbb{I}(\mathrm{PC}(c)) \sqsubset^I \mathbb{I}(\mathsf{Start}(\mathcal{P}_0)) \ \Big|\ \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c, \mathbb{R}, \boldsymbol{V}\rangle \leftarrow \boldsymbol{step}^{\mathcal{P}}_{\ell_{store}}(\langle \mathsf{Start}(\mathcal{P}_0), \mathsf{Clr}(\mathcal{P}_0) \mid t\rangle, \mathcal{S}, j)\ \Big] = 0$$

Proof. We will prove this lemma in two steps: first by showing that the invariant is preserved across Clio steps $\rightsquigarrow_p^*$ and then using that fact, we can show that the invariant is also preserved across Clio low steps $\curvearrowright_p *$. Our invariant will serve as our inductive hypothesis in both cases.

Proof on Step relation $\rightsquigarrow_p$: We will perform induction on the derivation of the steps (which will be finite when used with the low-step rules, i.e., it is well-founded) with the number of steps being k being 1 less than the total number of (possibly high) steps in the context of a single low step, and our inductive hypothesis will be if $\mathbb{I}(\mathrm{PC}(c)) \sqsubset^I \mathbb{I}(\mathsf{Start}(\mathcal{P}_0))$ and,

$$\mathbf{Pr}\Big[\ \mathbb{I}(\mathrm{PC}(c')) \sqsubset^I \mathbb{I}(\mathsf{Start}(\mathcal{P}_0)) \ \Big|\ \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c, \mathsf{skip}, \Sigma_0\rangle \rightsquigarrow_{p_1} \ldots \rightsquigarrow_{p_k} \langle c', \mathbb{R}', \mathbf{V}'\rangle\Big] = 0$$

then,

$$\mathbf{Pr}\Big[\ \mathbb{I}(\mathrm{PC}(c'')) \sqsubset^I \mathbb{I}(\mathsf{Start}(\mathcal{P}_0)) \ \Big|\ \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\ \langle c, \mathsf{skip}, \Sigma_0\rangle \rightsquigarrow_{p_1} \ldots \rightsquigarrow_{p_k} \langle c', \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_{p_{k+1}} \langle c'', \mathbb{R}', \mathbf{V}'\rangle\Big] = 0$$

The base case $k = 0$ is trivial as it is true by supposition.

For the inductive case, we now consider the derivation rule used for the $k+1$'th step and show that it preserves the inductive hypothesis, assuming it for the k'th step. We now perform a case analysis on the step used.

- **Case** Internal-Step**:**

  In this derivation we have that:

$$\frac{\textsc{internal step}\quad c' \longrightarrow c''}{\langle c', \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_1 \langle c'', \mathbb{R}', \mathbf{V}_1\rangle}$$

  By inspection of each of the LIO rules, the label is manipulated in the following ways:

– In UNLABEL, the current label is joined with the level of the labeled value, so the flows relation between the current label and the starting label is preserved.
– In RESET the label is returned to its original label. However, from the TOLABELED rule, the label is based on the current label. As a result, since the label is based on a previous step's current label, and the inductive hypothesis assumes it was true for that point, then the label it is reset to is also satisfies the flow relation to the starting label.
– In all other rules, the current label is not changed, which by supposition satisfies the flow relation.

Proof on Low-Step relation $\curvearrowright\mathbf{p}$ By induction on j.

- **Base Case:** $j = 1$: That is, we will prove the following:

$$\mathbf{Pr}\Big[\, \mathbb{I}(\mathrm{PC}(c)) \sqsubset^I \mathbb{I}(\mathsf{Start}(\mathcal{P}_\circ)) \;\Big|\; \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_\circ \uplus \mathcal{P}';\quad \langle c, \mathbb{R}, \mathbf{V}\rangle \leftarrow \mathbf{step}^{\mathcal{P}}_{\ell_{store}}(\langle \mathsf{Start}(\mathcal{P}), \mathsf{Clr}(\mathcal{P}) \mid t\rangle, \mathcal{S}, 1) \Big] = 0$$

There are two cases we must consider in the low step relation, the LOW-STEP rule and the LOW-TO-HIGH-TO-LOW-STEP rule. In the LOW-STEP, we must show that the inductive hypothesis holds after a single Clio step $\rightsquigarrow_p$, and for the LOW-TO-HIGH-TO-LOW-STEP rule must hold for many (finite) Clio steps $\rightsquigarrow_p^*$. Note that the LOW-STEP rule is a special case of the LOW-TO-HIGH-TO-LOW-STEP rule so we only consider the more general case of preserving the invariant across many steps. To show this, we appeal to the previous proof made to show that the invariant is preserved across Clio steps. As a result, we have that:

$$\mathbf{Pr}\Big[\, \mathbb{I}(\mathrm{PC}(c')) \sqsubset^I \mathbb{I}(\mathsf{Start}(\mathcal{P}_\circ)) \;\Big|\; \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_\circ \uplus \mathcal{P}';\ \langle c, \mathsf{skip}, \Sigma_\circ\rangle \rightsquigarrow_{p_1} \ldots \rightsquigarrow_{p_k} \langle c', \mathbb{R}, \mathbf{V}\rangle \Big] = 0$$

From the previous proof on the single-step relation, we can conclude that $\mathbb{I}(\mathrm{PC}(c)) \sqsubset^I \mathbb{I}(\mathsf{Start}(\mathcal{P}_\circ))$. As a result, we satisfy the inductive hypothesis.

- **Inductive Case:** j = k+1: That is, we will prove the following:

$$\mathbf{Pr}\Big[\, \mathbb{I}(\mathrm{PC}(c)) \sqsubset^I \mathbb{I}(\mathsf{Start}(\mathcal{P}_\circ)) \;\Big|\; \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_\circ \uplus \mathcal{P}';\quad \langle c, \mathbb{R}, \mathbf{V}\rangle \leftarrow \mathbf{step}^{\mathcal{P}}_{\ell_{store}}(\langle \mathsf{Start}(\mathcal{P}), \mathsf{Clr}(\mathcal{P}) \mid t\rangle, \mathcal{S}, k+1) \Big] = 0$$

We can expand the **step** metafunction to be

$$\begin{aligned}
&\left(\langle c'', \mathbb{R}'', \mathbf{V}' \rangle, p \cdot p'\right) \\
&\quad \text{where} \\
&\qquad \left(\langle c', \mathbb{R}, \mathbf{V} \rangle, p\right) \in \mathbf{step}^{\mathcal{P}}_{\ell_{store}}(c, \mathcal{S}, k); \\
&\qquad \mathbb{R}' = \mathcal{S}(\mathbb{R}); \\
&\qquad (\langle c', \mathbb{R}, \mathbf{V} \rangle, \mathbb{R}') \curvearrowright_{p'} \langle c'', \mathbb{R}'', \mathbf{V}' \rangle
\end{aligned}$$

The strategy on the stores does not affect the current label. After this adversary interaction, we can invoke the single-step lemma result here to conclude that $\mathbb{I}(\mathrm{PC}(c'')) \sqsubset^{I} \mathbb{I}(\mathsf{Start}(\mathcal{P}_\circ))$. As a result, the inductive hypothesis is true.

With all cases accounted for in the low step relation, and the single-step relation, we can conclude the proof.

□

C.2.5 Leveraged Forgery Security Proof

Definition 20 (Values function)**.** *Define the Values function as follows:*

$$\begin{array}{lcll} \mathsf{Values}_{\mathcal{P}}(\mathit{put}\ \langle b : l_1 \rangle\ \mathit{at}\ \underline{v_k}\ \cdot\ \overline{R}) & = & \mathit{put}\ \langle b : l_1 \rangle\ \mathit{at}\ \underline{v_k}\ \cdot\ \mathsf{Values}(\overline{R}) & \text{if } \langle (\underline{v}, \underline{v_k}, n) : l_1 \rangle = \mathsf{deserialize}_{\mathcal{P}}(\overline{R}, \langle b : l_1 \rangle, \tau) \\ \mathsf{Values}_{\mathcal{P}}(R\ \cdot\ \overline{R}) & = & \mathsf{Values}(\overline{R}) & \text{otherwise} \end{array}$$

Theorem 21 (Existential forgery under chosen message attack)**.** *For all keystores $\mathcal{P}_0$, principals p in principal sets $\tilde{p}$, and j if Π is secure against existential forgery under chosen message attacks, then*

for all $\mathbb{I}(l_2) \sqsubseteq^I \mathbb{I}(l_1) \sqsubseteq^I \mathbb{I}(\mathsf{authorityOf}(\mathcal{P}_0)) \wedge p$*,*

$$\boldsymbol{Pr}\Big[\ \langle b : l_1 \rangle \in \mathrm{Values}_{\mathcal{P}}(\overline{R'}) \text{ and } \langle b : l_1 \rangle \notin \mathrm{Values}_{\mathcal{P}}(\overline{R}) \\ \Big|\ \mathcal{P}' \leftarrow \mathsf{Gen}(\{p\}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}'; \\ t, \mathcal{S}, \mathcal{A}_2 \leftarrow \mathcal{A}(\mathrm{pub}(\mathcal{P})); \\ \langle c, \mathbb{R}, \boldsymbol{V} \rangle \leftarrow \boldsymbol{step}^{\mathcal{P}}_{\ell_{store}}(\langle \mathsf{Start}(\mathcal{P}), \mathsf{Clr}(\mathcal{P}) \mid t \rangle, \mathcal{S}, j); \\ \overline{R} \leftarrow \mathbb{R}; \\ t', \mathcal{S}' \leftarrow \mathcal{A}_2(\overline{R}); \\ \langle c', \mathbb{R}', \boldsymbol{V}' \rangle \leftarrow \boldsymbol{step}^{\mathcal{P}}_{\ell_{store}}(\langle \mathsf{Start}(\mathcal{P}_0), \mathsf{Clr}(\mathcal{P}) \mid t' \rangle, \mathcal{S}', j); \\ \overline{R'} \leftarrow \mathbb{R}'\Big]$$

is negligible in n.

Proof. We consider the level required to produce a valid signature during a store operation. The signature must be valid for a $p \in \tilde{p}$. By inspection of the Clio semantics, the only way to a valid signature would occur in the interaction is during the store operation, which uses the labeled value's label, or by the strategy.

According to the store operation, current label must be bounded above by the label of the labeled value (i.e., $l_{\mathsf{cur}} \sqsubseteq l_1$). For integrity, this means the current label's integrity component I must be at least as trustworthy as the principal p.

By the previous lemma, we can conclude that the current integrity label will never be at a level I' such that

$$I' \sqsubset^I \mathbb{I}(\mathsf{Start}(\mathcal{P}_0))$$

This means that the level of the Clio computation would need to be at least

$$I = \mathbb{I}(\mathsf{authorityOf}(\mathcal{P}_0 \uplus \{p \mapsto \mathsf{Gen}(1^n)\}))$$

By unpacking the definition of $\mathsf{Start}(\mathcal{P} \uplus \{p \mapsto \mathsf{Gen}(1^n)\})$, this integrity label satisfies the following relation

$$I \sqsubset^I \mathbb{I}(\mathsf{Start}(\mathcal{P}_0))$$

Which we have shown is impossible to reach. As a result, the current label's integrity level will never be at a level where it can sign the value using p's private signing key.

As a result, the only way a high integrity value could be in the challenge store σ' and not in the original store σ would require the strategy to forge a signature itself without Clio's assistance. Since this occurs with only negiglible probability, we have satisfied the proof obligation. □