MAKING DISTRIBUTED COMPUTATION SECURE BY CONSTRUCTION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Lantian Zheng

January 2007

MAKING DISTRIBUTED COMPUTATION SECURE BY CONSTRUCTION

Lantian Zheng, Ph.D.

Cornell University 2007

This thesis introduces new techniques to build distributed applications that are secure by construction, satisfying strong, end-to-end policies for confidentiality, integrity and availability. The new techniques are designed to solve the problem of how to specify, analyze and enforce end-to-end availability policies in distributed settings, without jeopardizing the enforcement of confidentiality and integrity policies. This thesis also presents a correctness proof for these techniques.

BIOGRAPHICAL SKETCH

Lantian Zheng is a graduate student at the Computer Science Department of Cornell University. Lantian received his B.S. degree from Peking University and his M.S. degree from Cornell University.

To my parents

ACKNOWLEDGEMENTS

I have many people to thank for helping me complete this thesis. First and foremost is my advisor, Andrew Myers. I have learned a great deal from Andrew over the years. Without his guidance this thesis could not have been written.

I also owe a large debt to the other members of my thesis committee: Fred Schneider and Levent Orman. They provided invaluable advice in a timely fashion. I particularly want to thank Fred for his insightful feedback on drafts of this work.

I wish to express warm thanks to Steve Zdancewic, Nate Nystrom and Stephen Chong, who were my collaborators on the Jif/split project. Many of the ideas in this thesis were born out of the project.

Thanks go to Michael Clarkson for his feedback on early drafts of this thesis. Over the years, I have benefited from technical discussions with Andrei Sabelfeld, Lorenzo Alvisi, Heiko Mantel, Greg Morrisett, Wei Wei, Yong Yao, Jed Liu, Michael George, Krishnaprasad Vikram and Xin Qi. I would also like to thank Lidong Zhou for introducing me to the idea of quorum system.

I wish to thank my parents for all of their support during my never-ending education. And thanks to Yuan for providing much-needed distractions.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1
Introduction

Distributed computing systems are ubiquitous, yet it is currently difficult to make strong statements about the security provided by a distributed system as a whole, especially if some of the participants in a distributed computation do not trust other participants or the computing software and hardware they provide. Distributed systems serving mutually distrusting principals include clinical and financial information systems, business-to-business transactions, and joint military information systems.

This thesis proposes a unified approach (within a common framework of program analysis and transformation) to building distributed programs that enforce *end-to-end* confidentiality, integrity and availability policies, in a system with untrusted hosts.

Informally, an end-to-end confidentiality policy of data d specifies who can learn about d; an end-to-end integrity policy of d specifies who can affect d; an end-to-end availability policy of d specifies who can make d unavailable (d is *available* if the issuer of an authorized access request to d will eventually get the value of d). These policies regulate the behaviors of the whole system and can be viewed as an application of the end-to-end principle [74] to specifying security policies.

End-to-end confidentiality and integrity policies are also known as information flow policies, since they impose restrictions on how information is propagated throughout the system. Dynamic information flow control mechanisms, including mandatory access control (MAC) [9, 19], use run-time checks to ensure that information does not flow to a place protected by a weaker confidentiality policy or a stronger integrity policy. Although widely used in practice, those dynamic mechanisms suffer from high run-time overheads and covert exception channels associated with run-time security checks. Further, these dynamic mechanisms abort the programs that fail a run-time check, making it difficult to enforce availability policies. Denning [18] showed how to use static program analysis to ensure that programs do not violate its information flow policies, and this approach has been instantiated in a number of languages in which the type system implements a similar static analysis (e.g., [88, 34, 102, 70, 7, 73]). Although static information flow control does not have the shortcomings of those dynamic mechanisms, it remains a challenge to develop a *sound* static analysis for distributed programs running in a system with untrusted hosts.

End-to-end availability policies specify availability requirements in terms of which principal can make the concerned data unavailable. The expressiveness of availability policies thus depends on the expressiveness of principals, which can represent not only users but also hardware, attacks or defense mechanisms, as shown in the following examples:

- `power`: the main power supply of a system, whose failure may bring down the entire system.
- `hostset`(n): a host set containing n hosts. This principal can be used to specify the minimum number (n in this case) of host failures needed to bring down a system. This is a common way of specifying availability requirements [75].
- `puzzle`: the puzzle generated by a puzzle-based defense mechanism [41] for DoS attacks. This principal can be used to specify the availability requirement that the system tolerates DoS attacks from attackers who cannot feasibly solve the puzzle.

Intuitively, end-to-end availability policies prevent attackers from making data unavailable. However, these policies do not ensure data to be available eventually. For example, a system may contain an infinite loop (such as an event handler), and any output to be produced after the loop will not be available, even if there are no attacks. A theoretical implication is that enforcing such availability policies does not require solving the halting problem [38].

1.1 Security by construction

Following the idea of static information flow control, a straightforward approach to building a secure distributed program is to develop a static program analysis that can determine whether a distributed program enforces its security policies. However, this *analytic approach* is not appealing for distributed systems composed of hosts that are heterogeneously trusted by different principals. In such a system, the distribution of data and computation depends on the *trust configuration*, that is, the trust relationship between principals and hosts. For example, Alice's confidential data can only be distributed to hosts that are trusted by Alice to protect data confidentiality. Such security concerns may be orthogonal to application logic, increasing the burden of software development. In addition, with this approach, programmers have to figure out how to adapt a program to changing trust configurations, and thus the burden of software maintenance is also increased.

This thesis pursues a *constructive approach*, which allows programmers to write high-level source program as if the program would be run on a single trusted host, and uses a compiler to translate the source program into a secure distributed program with respect to a given trust configuration. Programmers can thus focus on the application logic. Moreover, if the trust configuration changes, a mere recompilation of the source code with respect to the new configuration will generate a new secure distributed program.

1.1.1 Example

The advantages of the constructive approach can be illustrated by a simple example. Suppose Alice tries to bid for something from Bob, and Bob has three items that satisfy Alice's requirement and offers a price for each item. The offers are stored in an ordered

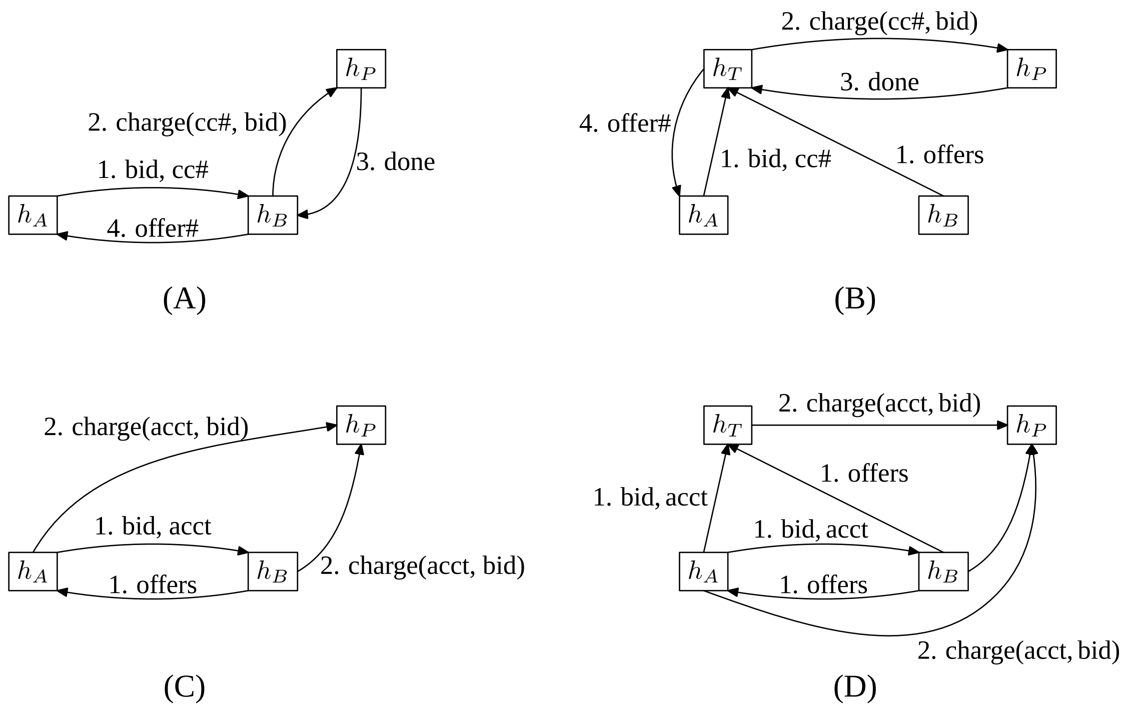


Figure 1.1: Distributed implementations of the bidding application

list, and Alice is forced to accept the first offer in the list that is lower than or equal to her bid. During the transaction, Alice and Bob should not be able to update the bid or the offers, and they should not be able to abort the transaction alone.

Using the analytic approach

Figure 1.1 shows several distributed implementations of the application. For simplicity, only network messages are shown, which are labeled with sequence numbers indicating their order of occurrence. The computations done at each local host are described below:

(A) Host h_A (a host fully controlled and trusted by Alice) sends Alice's credit card number and bid to host h_B (a host fully controlled and trusted by Bob). Then h_B compares the bid with Bob's offers and sends a payment request to host h_P, which handles charging Alice's credit card account. Once h_B gets a response from h_P, it sends the offer number to h_A. This implementation corresponds to the common scenario in which Alice places the bid on a web site managed by Bob.

There are several security problems with this implementation. First, Alice may not trust h_B to keep her credit card number confidential. Second, Bob has the full control of h_B, and is able to change the bid and the offers, or simply abort the transaction.

(B) Suppose host h_T is trusted by both Alice and Bob. Then h_A and h_B can send all the data to h_T, which compares Alice's bid with Bob's offers and sends the payment request to h_P. This implementation relies on the existence of a host fully trusted by all the participating principals.

(C) Suppose the system is composed of only h_A, h_B and h_P. There are still ways to improve security over implementation (A). First, Alice's credit card number is simply a reference to Alice's payment account, and it is possible to use a public identifier `acct` as the account reference, such as the email address in the PayPal service. Second, the data (including the bid and the offers) and computation can be replicated on h_A and h_B, which both send the payment request to h_P. If the requests from h_A and h_B are not the same, h_P can detect that some host is compromised and abort the transaction. Therefore, Alice or Bob cannot modify the bid and the offers without being detected. The main problem with this implementation is that either h_A and h_B can send a fabricated request and cause the transaction to be aborted.

(D) Suppose Alice and Bob do not fully trust h_T. Instead, they trust that at most one host among h_A, h_B and h_T might fail. Then Alice and Bob would be satisfied with the security assurance of this implementation, in which the data and computation is also replicated on h_A, h_B and h_T, and host h_P accepts the payment request if the request comes from both h_A and h_B or from h_T.

```
t := 0; a := -1;
while (t < 3) do
  if (bid >= offer[t]) then acct := acct - bid; a := t; break;
  else t := t + 1;
result := a;
```

Figure 1.2: Bidding program

As Figure 1.1 shows, determining how to securely distribute and replicate data and computation may be subtle and error-prone. Further, if the corresponding trust configuration changes, a distributed program may need to undergo significant modification to adapt to the new configuration. For example, in (B), if h_T is detected to be compromised, and there are no other hosts trusted by both Alice and Bob, then the program needs to be redesigned.

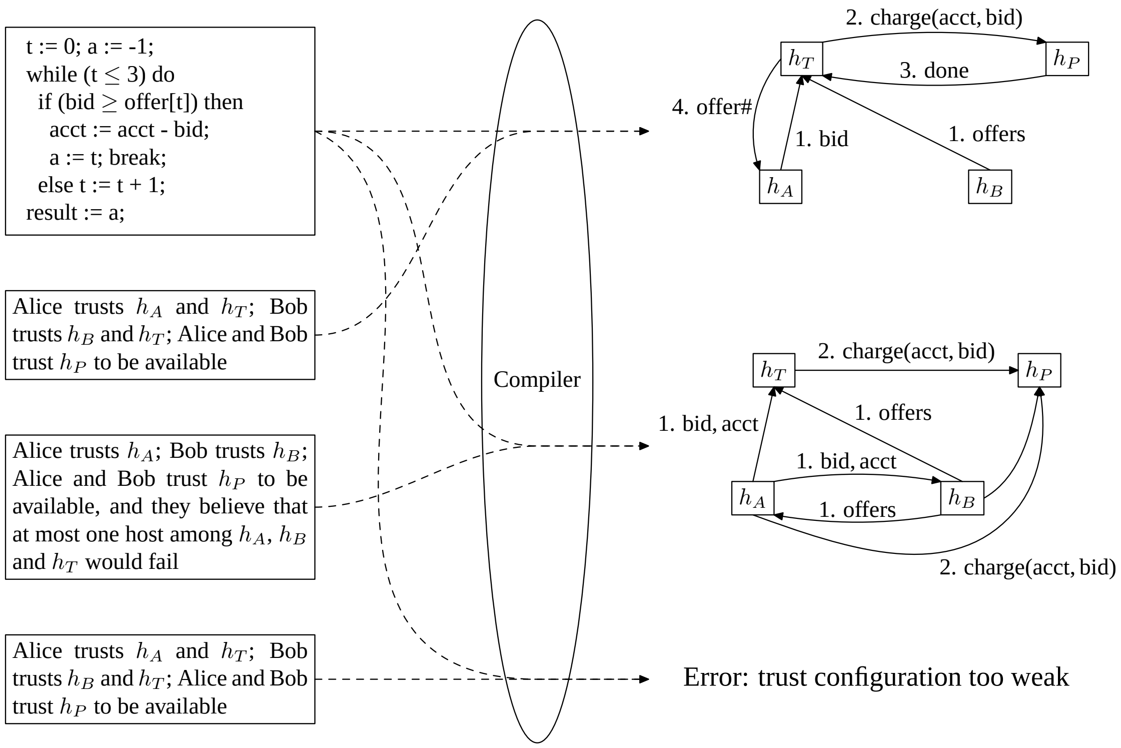
Using the constructive approach

Although the implementations in Figure 1.1 are different, they all try to perform the same computation, which can be described by the sequential program shown in Figure 1.2. Using the constructive approach, this simple program is all that needs to be written by programmers.

As Figure 1.3 shows, given the source program and a trust configuration, a compiler generates a secure distributed program or report an error when there are not enough trusted hosts in the system. With this approach, programmers can focus on application logic, since the trust configuration of the system is transparent to them. Further, only recompilation is needed to cope with changing trust configurations.

1.1.2 Secure program partitioning and replication

Earlier work on the Jif/split system [104, 105] explored the constructive approach, using end-to-end confidentiality and integrity policies to guide automatic partitioning and



t := 0; a := -1;
while (t ≤ 3) do
if (bid ≥ offer[t]) then
acct := acct - bid;
a := t; break;
else t := t + 1;
result := a;
Alice trusts h_A and h_T; Bob trusts h_B and h_T; Alice and Bob trust h_P to be available
Alice trusts h_A; Bob trusts h_B; Alice and Bob trust h_P to be available, and they believe that at most one host among h_A, h_B and h_T would fail
Alice trusts h_A and h_T; Bob trusts h_B and h_T; Alice and Bob trust h_P to be available
Compiler
2. charge(acct, bid)
3. done
4. offer#
1. bid
1. offers
h_T
h_P
h_A
h_B
2. charge(acct, bid)
1. bid, acct
1. offers
1. bid, acct
1. offers
2. charge(acct, bid)
Error: trust configuration too weak

Figure 1.3: Security by construction

replication of code and data onto a distributed system.

In the Jif/split system, source programs are written in Jif [62, 65], which extends Java with type-based static information flow control. The Jif/split compiler translates a source program into a distributed Java program that enforces the information flow policies specified as type annotations in the source.

1.1.3 What is new

The Jif/split system demonstrates the feasibility of the constructive approach to building secure distributed programs. However, the Jif/split system does not support end-to-end availability policies, and there is no correctness proof for the translation algorithm of the Jif/split compiler, partially because of the complexity of the Jif language.

In comparison to the work on the Jif/split system, this thesis makes two major contributions. First, this thesis proposes a way of analyzing and enforcing availability policies, based on the idea of static information flow control. Second, this thesis formalizes the core part of the Jif/split translation, extends it with support for availability, and proves the correctness of the translation.

1.2 Enforcing availability policies

Although availability is often considered one of the three key aspects of information security (along with confidentiality and integrity), availability assurance has been largely divorced from other security concerns. This thesis starts to bridge the gap by providing a unified way of specifying, analyzing and enforcing end-to-end confidentiality, integrity and availability policies.

End-to-end confidentiality and integrity policies can be enforced by ensuring that the system obey noninterference [31]. In general, an end-to-end availability policy on some data d specifies that principals $p_1, \ldots, p_n$ can make d unavailable, and such a policy can also be enforced by a form of noninterference: principals other than $p_1, \ldots, p_n$ do not interfere with the availability of d. This suggests that the idea of static information flow control can be applied to availability too. This thesis introduces a sequential language Aimp with a security type system that ensures a well-typed program satisfies the noninterference properties that enforce the end-to-end policies (including availability policies) specified as type annotations.

An interesting challenge in designing the Aimp type system is to analyze the dependencies between integrity and availability. Consider the code in Figure 1.2. Attackers can make the output `result` unavailable by compromising the integrity of `t` (making `t` always less than 3). Intuitively, the integrity policy of `t` needs to be as strong as the availability policy of `result`. To enable comparing an integrity policy with an availability policy, this thesis proposes a universal label model, in which confidentiality, integrity and availability labels have the same form and the same interpretation.

To enforce availability policies in a distributed setting, this thesis presents the DSR (Distributed Secure Reactors) language for describing distributed, concurrent computation on replicated hosts. The DSR language makes the following specific contributions related to enforcing availability policies:

- The DSR language supports quorum replication [30, 35], which is extended to be guided by explicit security policies. Voting replicas can enforce both integrity and availability policies.
- A novel timestamp scheme is used to coordinate concurrent computations running on different replicas, without introducing covert channels.

Applying the constructive approach, this thesis presents a translation from Aimp to DSR, which generates a secure distributed DSR program from an Aimp program and a trust configuration. The translation automatically figures out how to replicate data and code in quorum systems to enforce integrity and availability policies.

1.3 Proving correctness

In Aimp and DSR, security policies are explicitly defined using *labels* that annotate data items, computations, hosts and principals with security levels. In a system with mutually distrusted principals, attackers are treated as principals that have the power to affect certain behaviors of the system. The power of attackers is represented by a label $l_{\mathtt{A}}$. A label is *low-security* if it is lower than or equal to $l_{\mathtt{A}}$, and *high-security* otherwise. Intuitively, the policies specified by labels require that attackers cannot learn information about data with high-confidentiality labels, affect data with high-integrity labels, or make data with high-availability labels unavailable. Thus, a system is secure if the following three noninterference [31] properties are satisfied:

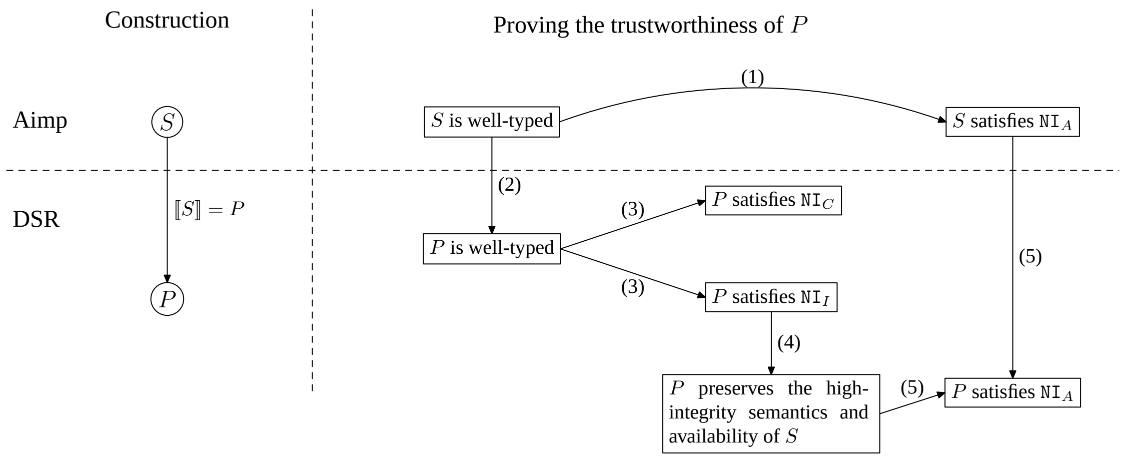

Construction
Proving the trustworthiness of P
Aimp
DSR
S
P
$[\![S]\!] = P$
S is well-typed
S satisfies $\texttt{NI}_A$
P is well-typed
P satisfies $\texttt{NI}_C$
P satisfies $\texttt{NI}_I$
P preserves the high-integrity semantics and availability of S
P satisfies $\texttt{NI}_A$
(1)
(2)
(3)
(3)
(4)
(5)
(5)

Figure 1.4: Trustworthiness by construction

- Confidentiality noninterference ($\texttt{NI}_C$): attackers cannot infer high-confidentiality data, or high-confidentiality inputs cannot interfere with low-confidentiality outputs that are observable to attackers.
- Integrity noninterference ($\texttt{NI}_I$): attackers cannot affect high-integrity data.
- Availability noninterference ($\texttt{NI}_A$): attackers cannot affect the availability of data with high-availability labels.

With these concepts, our goal is to prove that the Aimp-DSR translation generates distributed programs that satisfy $\texttt{NI}_C$, $\texttt{NI}_I$, and $\texttt{NI}_A$.

Figure 1.4 shows the proof strategy. A well-typed program S in Aimp is translated into a DSR program P, and the translation is denoted by $[\![S]\!] = P$. The proof that P satisfies the noninterference properties is done in the following steps.

(1) We show that the type system of Aimp ensures that a well-typed program satisfies the noninterference properties. Since S is well-typed, S satisfies $\texttt{NI}_A$.

(2) We show that the translation preserves sound typing. Thus, P is well-typed because S is well-typed.

(3) We show that the type system of DSR enforces the confidentiality and integrity

noninterference properties. Therefore, P satisfies $\texttt{NI}_C$ and $\texttt{NI}_I$, because P is well-typed.

(4) We show that P always produces the same high-integrity outputs as S despite attacks from low-integrity hosts, and P can achieve the same level of availability as S. In other words, if S produces a high-availability output, then P also produces that output, although the output may be different if it is low-integrity. The proof relies on the fact that P satisfies $\texttt{NI}_I$, which means attackers cannot affect the high-integrity outputs even if they can compromise some low-integrity hosts and launch attacks from them.

(5) We show that P satisfies $\texttt{NI}_A$, based on that P can achieve the same availability as S, and S satisfies $\texttt{NI}_A$.

1.4 Limitations

The Aimp/DSR instantiation of the constructive approach to building secure distributed programs has a few limitations. First, source language Aimp is a sequential language and cannot be used to express concurrent computation. However, many useful applications, including the bidding example in Figure 1.2, are sequential.

Second, the DSR type system does not deal with timing channels. Since Aimp is sequential, a target DSR program generated from an Aimp program does not cause race conditions and internal timing channels [103]. Attackers may still be able to infer confidential information by timing the network messages they can observe. However, this kind of timing channels is more noisy than internal timing channels, and it is largely an orthogonal issue, partially addressed by ongoing work [3, 72].

Third, in a distributed system, attackers can cause certain execution paths to diverge (going into an infinite loop or getting stuck) by compromising some hosts, and create termination channels. This issue is also not addressed in this thesis, partly because termination channels generally have low bandwidth.

Finally, in this thesis, the formal notion of availability glosses over another aspect of availability: timeliness. How soon does an output have to occur in order to be considered to be available? For real-time services, there may be hard time bounds beyond which a late output is useless. Reasoning about how long it takes to generate an output adds considerable complexity, and this is left for future work.

1.5 Outline

The remainder of this thesis is organized as follows. Chapter 2 describes the universal label model for specifying end-to-end security policies. Chapter 3 describes source language Aimp. Chapter 4 describes the features and mechanisms in DSR, which support performing secure distributed computation. Chapter 5 presents the formal semantics of DSR and proves the type system of DSR enforces confidentiality and integrity noninterference properties. Chapter 6 presents a translation from Aimp to DSR, and prove that the translation generates secure distributed programs. Chapter 7 concludes.

Chapter 2
Universal decentralized label model

A label model describes how to specify and analyze *security labels*, which are associated with data to describe the security levels of data and help characterize the restrictions on data generation and uses. Formally, a label model is a set of labels $\mathcal{L}$ with a partial order relation $\leq$. For example, $\{\texttt{public}, \texttt{secret}\}$ with $\leq$ being $\{(\texttt{public}, \texttt{public}),$ $(\texttt{public}, \texttt{secret}),$ $(\texttt{secret}, \texttt{secret})\}$ forms a simple confidentiality label model, in which `secret` represents a higher confidentiality level than `public`.

The *decentralized label model* (DLM) [64] allows different users to specify their own information flow policies in a security label. The DLM is suitable for a system with mutual distrust, in which different users might have different security requirements.

This chapter introduces a universal DLM, which extends the DLM with support for specifying and analyzing availability labels. The universal DLM is composed of a set of *decentralized labels*, each of which can be used as a confidentiality, integrity or availability label. This is possible due to a uniform semantics for decentralized labels, which interprets a label as a security assumption. Besides simplicity, the major benefit of the universal DLM is to support analyzing interactions between integrity and availability.

2.1 Security properties, labels and policies

This thesis focuses on the three core security properties on information: confidentiality, integrity and availability. In the security literature, the three properties (especially integrity and availability) have many different meanings [69, 12]. In this thesis, a security property ρ, which may represent the confidentiality, integrity or availability of data d, is written and interpreted as follows:

- *confidentiality*(d), meaning that attackers cannot learn about data d,

- *integrity*(d), meaning that attackers cannot affect the value of d, or
- *availability*(d), meaning that attackers cannot make d unavailable.

A security property is often treated as a predicate on the set of all traces of a system. For example, Zakinthinos and Lee [100] define a security (confidentiality) property as a predicate that a system satisfies if and only if for any trace τ of the system, the set of traces that attackers cannot distinguish from τ satisfy another specific predicate. In particular, their work shows that a noninterference property can be defined as such a predicate. The security properties considered in this thesis are also closely related to the notion of noninterference. Intuitively, a system enforces *confidentiality*(d) if and only if the value of d does not interfere with any data observable to attackers; a system enforces *integrity*(d) and *availability*(d) if and only if attackers cannot interfere with the value and availability of d, respectively. Therefore, a security property ρ can also be viewed as a predicate on the set of all traces of a system.

Security assumptions as labels

It is infeasible to enforce a security property if attackers have unconstrained power, and security rests on assumptions that constrain the power of attackers. For example, security commonly depends on a trusted computing base (TCB), which is assumed immune to attacks. Furthermore, security assumptions help define the restrictions on the use of data. For example, consider enforcing *confidentiality*(d) under the assumption that user `root` is trusted, which says that `root` is not controlled by attackers. According to the assumption, `root` may be allowed to learn about d, but not other principals. Defining the restrictions on the use of data is also the purpose of security labels. Thus, it is natural to specify security assumptions as labels. Formally, if a label l is associated with property ρ, then the semantics of l, written as $[\![l]\!]$, describes the security assumptions for enforcing ρ.

Security policies

A label l specified on property ρ defines a *security policy* $\langle \rho : l \rangle$, which is enforced by a system $\mathcal{S}$ if and only if $\mathcal{S}$ satisfies ρ under the assumption $[\![l]\!]$, or more formally, $[\![l]\!] \Rightarrow \rho(\mathcal{S})$, where $\rho(\mathcal{S})$ denotes that $\mathcal{S}$ satisfies ρ, and $\Rightarrow$ means "implies". For example, $\langle \textit{confidentiality}(d) : l \rangle$ is a confidentiality policy on d, and it is enforced if attackers cannot learn about d under the assumption $[\![l]\!]$. Enforcing the policy $\langle \rho : l \rangle$ is also called "enforcing l on ρ", or simply "enforcing l" if there is no ambiguity about ρ. For brevity, in a logical proposition, we write ρ for $\rho(\mathcal{S})$ if it is clear which system is under consideration. For example, $[\![l]\!] \Rightarrow \rho(\mathcal{S})$ can be written as $[\![l]\!] \Rightarrow \rho$ if there is no ambiguity about $\mathcal{S}$.

In this thesis, a system is considered *secure* if it does not violate any policy of the form $\langle \rho : l \rangle$ specified on its data.

2.2 Dependency analysis and noninterference

A computing system processes inputs and produces outputs, creating dependencies between security properties of those inputs and outputs. Such dependencies capture the system vulnerabilities that can be exploited by attackers to compromise security properties. For example, consider a system running the following pseudo-code:

```
while ($i$ > 0) skip;
$o$ := $i$;
```

This program assigns the input i to the output o if the value of i is not positive. Otherwise, the program diverges and the output is unavailable. Thus, the availability of o depends on the integrity of i. An attacker can try to exploit this dependency: making o unavailable by affecting the value of i to make it positive.

In a system $\mathcal{S}$, property ρ *depends* on $\rho_1 \vee \ldots \vee \rho_n$, written $\rho_1 \vee \ldots \vee \rho_n \rightsquigarrow \rho$, if the proposition $\neg(\rho_1(\mathcal{S}) \vee \ldots \vee \rho_n(\mathcal{S})) \Rightarrow \neg\rho(\mathcal{S})$ holds, that is, if all properties ρ_1 through ρ_n are *not* satisfied in $\mathcal{S}$, then ρ is *not* satisfied in $\mathcal{S}$. In general, the dependencies caused by a system can be identified by statically analyzing the code of a system. For example, by analyzing the code of the above system, we know that the value of o is computed using the value of i, and the availability of o is affected by the value of i. Therefore, the system causes the following dependencies:

$$\begin{aligned}
&\textit{confidentiality}(o) \rightsquigarrow \textit{confidentiality}(i)\\
&\textit{integrity}(i) \rightsquigarrow \textit{integrity}(o)\\
&\textit{availability}(i) \rightsquigarrow \textit{availability}(o)\\
&\textit{integrity}(i) \rightsquigarrow \textit{availability}(o)
\end{aligned}$$

We assume that an attacker can interact with a system only by affecting the inputs and observing the outputs. The interactions between attackers and inputs/outputs of a system always mean that some security properties are compromised: if attackers can observe an output, then the confidentiality of the output is compromised; if attackers can affect the value of an input, then the integrity of the input is compromised; and if attackers can affect the availability of an input, then the availability of the input is compromised.

In addition, we assume that a policy is enforced in a system if the enforcement of the policy is affected not by the system, but by the external environment. In particular, a system cannot affect how its inputs are generated and how its outputs are used, and thus the integrity and availability policies on inputs, and the confidentiality policies on outputs are assumed to be enforced. This is called the *safe environment* assumption. Based on this assumption, we have the following theorem, which gives a sufficient condition for ensuring that a system is secure.

Theorem 2.2.1 (Dependency). Let l_ρ be the label specified on property ρ. A system is secure if it satisfies the following condition:

$$\forall \rho, \rho_1, \ldots, \rho_n.\ (\rho_1 \vee \ldots \vee \rho_n \rightsquigarrow \rho) \Rightarrow (\llbracket l_\rho \rrbracket \Rightarrow \llbracket l_{\rho 1} \rrbracket \vee \ldots \vee \llbracket l_{\rho n} \rrbracket) \qquad \text{(DP)}$$

Proof. Suppose, by way of contradiction, that the system is not secure and violates a policy $\langle \rho : l_\rho \rangle$. Then $\llbracket l_\rho \rrbracket \Rightarrow \rho$ does not hold, which implies $\llbracket l_\rho \rrbracket \wedge \neg\rho$. Since $\llbracket l_\rho \rrbracket$ holds, by the safe environment assumption, attackers cannot compromise ρ directly without exploiting the system. Therefore, ρ is compromised because of some interactions between attackers and the system. As discussed above, those interactions mean that a set of properties $\rho_1, \ldots, \rho_n$ are compromised directly, and for all i in $\{1, \ldots, n\}$, the enforcement of policy $\langle \rho_i : l_{\rho_i} \rangle$ is not affected by the system. By the safe environment assumption, $\langle \rho_i : l_{\rho_i} \rangle$ is enforced, and $\llbracket l_{\rho_i} \rrbracket$ does not hold for all $i \in \{1, \ldots, n\}$. Since ρ is compromised due to those interactions, we have $\neg\rho_1 \wedge \ldots \wedge \neg\rho_1 \Rightarrow \neg\rho$, that is, $\rho_1 \vee \ldots \vee \rho_n \rightsquigarrow \rho$. By DP, we have $\llbracket l_\rho \rrbracket \Rightarrow \llbracket l_{\rho 1} \rrbracket \vee \ldots \vee \llbracket l_{\rho n} \rrbracket$, which contradicts the fact that $\llbracket l_\rho \rrbracket$ holds, and $\llbracket l_{\rho_i} \rrbracket$ does not hold for $i \in \{1, \ldots, n\}$. $\square$

By Theorem 2.2.1, a system is secure if it satisfies the following condition that is equivalent to DP:

$$\forall \rho_1, \ldots, \rho_n, \rho.\ (\llbracket l_\rho) \rrbracket \not\Rightarrow \llbracket l_{\rho 1} \rrbracket \vee \ldots \vee \llbracket l_{\rho n} \rrbracket) \Rightarrow \rho_1 \vee \ldots \vee \rho_n \not\rightsquigarrow \rho \qquad \text{(NI)}$$

The condition NI says that ρ does not depend on $\rho_1 \vee \ldots \vee \rho_n$, unless their labels satisfy $\llbracket l_\rho \rrbracket \Rightarrow \bigvee_{1 \leq i \leq n} \llbracket l_{\rho_i} \rrbracket$. The notion of independence between properties is often formalized a noninterference [31] property. In practice, DP helps construct a program dependency analysis and identify the label constraints that need to be verified, while NI is often used as a semantic security condition in proving the correctness of an enforcement mechanism. For both conditions, we only need to reason about the label constraints of the form $\llbracket l \rrbracket \Rightarrow \llbracket l_1 \rrbracket \vee \ldots \vee \llbracket l_n \rrbracket$ and the dependencies between security properties. In particular, we do not need to directly reason whether $\llbracket l \rrbracket \Rightarrow \rho$ holds.

2.3 Universal decentralized label model

In the security model discussed in Section 2.1, a security label is interpreted as a security assumption, regardless of what kind of security property that the label is specified on. As a result, it is possible to construct a universal label model applicable to confidentiality, integrity and availability simultaneously. This is desirable because it allows us to compare labels specified on different kinds of properties. For example, we can compare an integrity label with an availability label, and reason about dependencies between integrity and availability.

This section extends the DLM to make it universally applicable to confidentiality, integrity and availability. And a uniform semantics for labels is presented.

2.3.1 Owned labels

The DLM is designed to let users specify and manage their own confidentiality and integrity labels. Thus, it is necessary to be able to identify the owner of a label. To achieve this ability, the DLM is built on *owned labels*, in which label owners are specified explicitly. An owned label O has the form $u : p$, where user principal u is the owner of the label, and p is a principal, representing the system entities that u considers non-compromised (not falling under the control of attackers).

This section describes the syntax and semantics of owned labels. The formalization borrows some ideas from the access control calculus [1] designed by Abadi, Burrows, Lampson and Plotkin.

Principals

Formally, principals are specified using the following syntax:

$$\text{Principals} \quad u, p \quad ::= \quad a \;\mid\; * \;\mid\; p_1 \wedge p_2 \;\mid\; p_1 \vee p_2$$

The meta-variable a is an abstract name representing an atomic entity that may affect the behavior of a system. For example, a may be used to represent a user, a host, the power supply or other system components. Principal $*$ is a top principal who *acts for* every principal. In general, principal p_1 acts for principal p_2, written as $p_1 \succeq p_2$, if p_1 can act with the full authority of p_2, or in other words, any behavior by p_1 can be viewed as a behavior by p_2. The $\succeq$ relation is reflexive and transitive.

Another useful relation between principals is the *speaks-for* relation [1]: p_1 speaks for p_2 if any claim made by p_1 can be viewed as a claim made by p_2. Intuitively, the acts-for relation is stronger than the speaks-for relation, since making a claim is just one kind of behaviors that a principal may perform.

It is possible to construct more complex principals using conjunction and disjunction operators [1]: $\wedge$ and $\vee$. The composite principal $p_1 \wedge p_2$ is the principal with exactly the authority of both p_1 and p_2. Any behavior by both p_1 and p_2 is viewed as a behavior by $p_1 \wedge p_2$, and vice versa. It is clear that $p_1 \wedge p_2$ is the least upper bound of p_1 and p_2 with respect to the $\succeq$ ordering. More concretely, $p_1 \wedge p_2$ acts for p_1 and p_2; and if principal p acts for p_1 and p_2, then p also acts for $p_1 \wedge p_2$.

Another constructor $\vee$ is used to construct a group (disjunction): any behavior by p_1 or p_2 is considered a behavior of $p_1 \vee p_2$, and vice versa. There are other meaningful principal constructors such as "a *as* R" (the principal a in role R) [1, 25], and $\neg a$ (the *negative* principal of a), which represents the principal who has all the authorities that a does not have. The negative principal can be used to specify the *separation of duties* [14]. Suppose data d can be read only by principal a, and data d' only by $\neg a$. Then no principal other than the top principal can read both d and d'. This thesis only considers the conjunctive and disjunctive connectors because these two connectors are sufficient for specifying expressive end-to-end policies.

We assume that a principal is either compromised or non-compromised. A compromised principal is controlled by attackers, while a non-compromised principal is not. More formally, let $\mathtt{A}$ be the principal representing all the attackers. Then a principal p is compromised if and only if $\mathtt{A} \succeq p$.

Semantics

The owned label $u\!:\!p$ explicitly expresses the assumption by u that p is non-compromised. As the label owner, user u is able to affect the implications of this label by making claims about the acts-for relations or whether other principals are non-compromised. For example, u may claim $p' \succeq p$. Then label $u\!:\!p$ also implies that p' is non-compromised. In general, label $u:p$ conveys an implicit assumption that u is *honest*, meaning that every claim made by u is true.

Note that a non-compromised principal is not necessarily honest. Furthermore, we do not assume that a compromised principal is dishonest because the assumption is not essential, albeit intuitive.

Formally, a security assumption can be expressed by a proposition σ with the following syntax:

$$\sigma ::= \mathit{good}\, p \;\mid\; p\ \mathit{says}\ \sigma \;\mid\; \mathit{honest}\, p \;\mid\; p_1 \succeq p_2 \;\mid\; \sigma_1 \wedge \sigma_2 \;\mid\; \sigma_1 \vee \sigma_2$$

The interpretation is straightforward: $\mathit{good}\, p$ means that p is non-compromised; $p\ \mathit{says}\ \sigma$ means that p claims σ; $\mathit{honest}\, p$ means that p is honest ($\forall\sigma.\ p\ \mathit{says}\ \sigma \Rightarrow \sigma$); $p_1 \succeq p_2$ means that p_1 acts for p_2. The connectors $\wedge$ and $\vee$ are the standard propositional "and" and "or". With this language, the semantics of label $u\!:\!p$ is as follows:

$$[\![u\!:\!p]\!] = \mathit{honest}\, u \wedge \mathit{good}\, p$$

By the meaning of $p_1 \succeq p_2$, it is clear that $p_1 \succeq p_2$ implies $\mathit{good}\, p_2 \Rightarrow \mathit{good}\, p_1$, $p_1\ \mathit{says}\ \sigma \Rightarrow p_2\ \mathit{says}\ \sigma$, and $\mathit{honest}\, p_1 \Rightarrow \mathit{honest}\, p_2$. By the definition of $p_1 \wedge p_2$, we immediately have the following inference rules:

R1. $\mathit{good}\, p_1 \vee \mathit{good}\, p_2 \Leftrightarrow \mathit{good}\,(p_1 \wedge p_2)$

R2. $p_1\ \mathit{says}\,\sigma \wedge p_2\ \mathit{says}\,\sigma \Leftrightarrow p_1 \wedge p_2\ \mathit{says}\,\sigma$

Dually, we have the following rules with $p_1 \vee p_2$:

R3. $\mathit{good}\, p_1 \wedge \mathit{good}\, p_2 \Leftrightarrow \mathit{good}\,(p_1 \vee p_2)$

R4. $p_1\ \mathit{says}\,\sigma \vee p_2\ \mathit{says}\,\sigma \Leftrightarrow p_1 \vee p_2\ \mathit{says}\,\sigma$

By rule (R4), we can prove the following lemma:

Lemma 2.3.1. $\mathit{honest}\, p_1 \vee p_2 \Leftrightarrow \mathit{honest}\, p_1 \wedge \mathit{honest}\, p_2$

Proof.

$$\begin{array}{rl}
& \mathit{honest}\, p_1 \vee p_2 \\
\Leftrightarrow & \forall \sigma.\ p_1 \vee p_2\ \mathit{says}\,\sigma \Rightarrow \sigma \\
\Leftrightarrow & \forall \sigma.\ p_1\ \mathit{says}\,\sigma \vee p_2\ \mathit{says}\,\sigma \Rightarrow \sigma \\
\Leftrightarrow & \forall \sigma.\ (p_1\ \mathit{says}\,\sigma \Rightarrow \sigma) \wedge (p_2\ \mathit{says}\,\sigma \Rightarrow \sigma) \\
\Leftrightarrow & (\forall \sigma.\ p_1\ \mathit{says}\,\sigma \Rightarrow \sigma) \wedge (\forall \sigma.\ p_2\ \mathit{says}\,\sigma \Rightarrow \sigma) \\
\Leftrightarrow & \mathit{honest}\, p_1 \wedge \mathit{honest}\, p_2
\end{array}$$

□

2.3.2 Decentralized labels

Owned labels allow different principals to specify and manage their security requirements. Multiple owned labels $O_1, \ldots, O_n$ may be specified on the same security property ρ. A secure system needs to enforce all these labels $O_1, \ldots, O_n$ on ρ, which amounts to ensuring $\forall i \in \{1, \ldots, n\}.\ [\![O_i]\!] \Rightarrow \rho$, or equivalently, $(\bigvee_{1 \le i \le n} [\![O_i]\!]) \Rightarrow \rho$.

Based on this observation, we can write $O_1, \ldots, O_n$ together as a single label $l = \{O_1, \ldots, O_n\}$ and let $[\![l]\!] = \bigvee_{1 \le i \le n} [\![O_i]\!]$. Then specifying and enforcing $O_1, \ldots, O_n$ on ρ is equivalent to specifying and enforcing l on ρ. The label l is called a *decentralized* *label* because it incorporates security requirements of different principals that generally do not fall under a centralized authority. Now the security requirements with regard to a security property can be described by a single label, even in a distributed system with mutual distrust. This greatly simplifies security analysis.

2.3.3 Comparing labels

A label l_2 is as high as another label l_1, written as $l_1 \leq l_2$, if the enforcement of l_2 on any property ρ implies the enforcement of l_1 on ρ. Intuitively, if ρ is enforced under a weak assumption, then ρ is also enforced under a strong assumption. Therefore, $l_1 \leq l_2$ if and only if $[\![l_1]\!]$ is as strong as $[\![l_2]\!]$, or $[\![l_1]\!] \Rightarrow [\![l_2]\!]$. By the semantics of owned labels, the following rule for comparing owned labels immediately follows:

$$\frac{u_2 \succeq u_1 \quad p_2 \succeq p_1}{u_1\!:\!p_1 \leq u_2\!:\!p_2}$$

Consider two decentralized labels l_1 and l_2. Intuitively, if for any owned label O in l_1, there exists an owned label in l_2 that is as high as O, then l_2 is as high as l_1. Formally, it is easy to show that $(\forall O \in l_1.\ \exists O' \in l_2.\ O \leq O')$ implies $[\![l_1]\!] \Rightarrow [\![l_2]\!]$. Thus, we have the following inference rule for comparing decentralized labels:

$$\frac{\forall O \in l_1.\ \exists O' \in l_2.\ O \leq O'}{l_1 \leq l_2}$$

The set of all the decentralized labels form a lattice with the following *join* ($\sqcup$) and *meet* ($\sqcap$) operations:

$$\begin{aligned} &l_1 \sqcup l_2 = l_1 \cup l_2 \\ &l_1 \sqcap l_2 = \{u_1 \vee u_2\!:\!p_1 \vee p_2 \mid u_1\!:\!p_1 \in l_1 \wedge u_2\!:\!p_2 \in l_2\} \end{aligned}$$

The join and meet operations are well-defined because of the following theorem, which implies that $l_1 \sqcup l_2$ is the least upper bound of l_1 and l_2 with respect to the $\leq$ ordering, and $l_1 \sqcap l_2$ is the greatest lower bound of l_1 and l_2.

Theorem 2.3.1. $[\![l \sqcup l']\!] = [\![l]\!] \vee [\![l']\!]$ and $[\![l \sqcap l']\!] = [\![l]\!] \wedge [\![l']\!]$.

Proof. Suppose $l = \{O_1, \ldots, O_n\}$ and $l' = \{O'_1, \ldots, O'_m\}$, and $O_i = u_i\!:\!p_i$ and $O'_j = u'_j\!:\!p'_j$. Then $l \sqcup l' = \{O_1, \ldots, O_n, O'_1, \ldots, O'_m\}$. Thus,

$$[\![l \sqcup l']\!] = (\textstyle\bigvee_{1\le i\le n}[\![O_i]\!]) \vee (\bigvee_{1\le j\le m}[\![O'_j]\!]) = [\![l]\!] \vee [\![l']\!].$$

For $l \sqcap l'$, we have the following deduction:

$$\begin{aligned}
&[\![l \sqcap l']\!] = [\![\{u_i\vee u'_j\!:\!p_i\vee p'_j \mid 1 \le i \le n,\ 1 \le j \le m\}]\!]\\
&= \textstyle\bigvee_{1\le i\le n,\ 1\le j\le m}[\![u_i\vee u'_j\!:\!p_i\vee p'_j]\!]\\
&= \textstyle\bigvee_{1\le i\le n,\ 1\le j\le m} \textit{honest}\, u_i\vee u'_j \wedge \textit{good}\, p_i\vee p'_j\\
&= \textstyle\bigvee_{1\le i\le n,\ 1\le j\le m} \textit{honest}\, u_i \wedge \textit{honest}\, u'_j \wedge \textit{good}\, p_i \wedge \textit{good}\, p'_j \qquad \text{(By Lemma 2.3.1 and R3)}\\
&= \textstyle\bigvee_{1\le i\le n,\ 1\le j\le m}[\![u_i\!:\!p_i]\!] \wedge [\![u'_j\!:\!p'_j]\!]\\
&= \textstyle\bigvee_{1\le i\le n}([\![u_i\!:\!p_i]\!] \wedge \bigvee_{1\le j\le m}[\![u'_j\!:\!p'_j]\!])\\
&= \textstyle\bigvee_{1\le i\le n}([\![u_i\!:\!p_i]\!] \wedge [\![l']\!])\\
&= (\textstyle\bigvee_{1\le i\le n}[\![u_i\!:\!p_i]\!]) \wedge [\![l']\!]\\
&= [\![l]\!] \wedge [\![l']\!]
\end{aligned}$$

□

By the definition of the join operation, $\perp = \emptyset$ is the bottom of the decentralized label lattice, since $\forall l.\ \emptyset \sqcup l = l$. Intuitively, the bottom label represents the strongest security assumption, and thus $[\![\emptyset]\!]$ is the proposition `false`. The top element of the decentralized label lattice is $\top = \{*\!:\!*\}$, because for any owned label $u\!:\!p$, we have $u\!:\!p \le *\!:\!*$.

Having a lattice of labels supports static program analysis [18]. For example, consider an addition expression $e_1 + e_2$. Let $A(e_1)$ and $A(e_2)$ represent the availability labels of the results of e_1 and e_2. By condition DP, we have $A(e_1 + e_2) \le A(e_1)$ and $A(e_1 + e_2) \le A(e_2)$, since the result of $e_1 + e_2$ is available if and only if the results of e_1 and e_2 are both available. Because the labels form a lattice, $A(e_1 + e_2) = A(e_1) \sqcap A(e_2)$ is the least restrictive availability label we can assign to the result of $e_1 + e_2$. Similarly, $I(e_1 + e_2) = I(e_1) \sqcap I(e_2)$ is the least restrictive integrity label for the result of $e_1 + e_2$, where $I(e_1)$ and $I(e_2)$ are respectively the integrity labels of e_1 and e_2. Dually, if $C(e_1)$ and $C(e_2)$ are the confidentiality labels of e_1 and e_2, then $C(e_1) \leq C(e_1 + e_2)$ and $C(e_2) \leq C(e_1 + e_2)$. The least restrictive confidentiality label that can be assigned to the result of $e_1 + e_2$ is $C(e_1) \sqcup C(e_2)$.

In addition, with a lattice label model, the DP and NI conditions can be written as:

$$\forall \rho, \rho_1, \ldots, \rho_n.\ (\rho_1 \vee \ldots \vee \rho_n \rightsquigarrow \rho) \Rightarrow (l_\rho \leq l_{\rho_1} \sqcup \ldots \sqcup l_{\rho_n}) \qquad \text{(DP)}$$

$$\forall \rho, \rho_1, \ldots, \rho_n.\ (l_\rho \not\leq l_{\rho_1} \sqcup \ldots \sqcup l_{\rho_n}) \Rightarrow \rho_1 \vee \ldots \vee \rho_n \not\rightsquigarrow \rho \qquad \text{(NI)}$$

2.3.4 Information security labels

In general, a system will need to simultaneously enforce labels on confidentiality, integrity, and availability for the information it manipulates. These labels can be applied to information as a single security label, which have the following syntax:

$$\begin{array}{rrcl} \text{Property names} & \alpha & \in & \mathbf{P} \\ \text{Security labels} & \ell & ::= & \{\alpha_1 = l_1, \ldots, \alpha_n = l_n\} \end{array}$$

Essentially, an information security label $\ell = \{\alpha_1 = l_1, \ldots, \alpha_n = l_n\}$ incorporates labels on various security properties about a piece of information: names $\alpha_1, \ldots, \alpha_n$ from a name space $\mathbf{P}$ identify the security properties, and l_i is the decentralized label on the property identified by α_i. The security label ℓ is composed of decentralized labels, but does not belong to the DLM itself. To distinguish these two kinds of labels, we call a decentralized label l a *base label*. The name space $\mathbf{P}$ contains at least C, I and A, representing the confidentiality, integrity and availability properties, respectively. Label ℓ usually has the form $\{C = l_1,\ I = l_2,\ A = l_3\}$, but can also contain base labels for other security properties. For example, in a real-time system, we may want to enforce *timing integrity*, which means that attackers cannot affect when a piece of data is generated. Then we can use the name *TI* to represent the timing integrity property and specify a base label l on *TI* to prevent attackers from compromising timing integrity under the assumption $[\![l]\!]$.

Given a label ℓ, let $\alpha_i(\ell)$ denote the base label component corresponding to α_i. For example, $C(\ell)$, $I(\ell)$, and $A(\ell)$ represent the respective confidentiality, integrity, and availability components of ℓ.

It is convenient to have a single label to incorporate base labels on different properties. For example, this leads to a more succinct label constraint when analyzing information flows. An information flow from data d_1 to data d_2 means that the value of d_2 may depend on the value of d_1. The security implications are that the confidentiality of d_1 depends on the confidentiality of d_2, and the integrity of d_2 depends on the integrity of d_1. Let ℓ_1 and ℓ_2 be the labels of d_1 and d_2, respectively. The dependencies caused by the information flow impose the following label constraints:

$$C(\ell_1) \leq C(\ell_2) \qquad I(\ell_2) \leq I(\ell_1)$$

Based on the two constraints, we can define an information flow ordering ($\sqsubseteq$) on labels:

$$\frac{C(\ell_1) \leq C(\ell_2) \qquad I(\ell_2) \leq I(\ell_2)}{\ell_1 \sqsubseteq \ell_2}$$

Information flow from d_1 to d_2 is secure if and only if $\ell_1 \sqsubseteq \ell_2$. In addition, we can define a join ($\sqcup$) operation on security labels:

$$\ell_1 \sqcup \ell_2 = \{C = C(\ell_1) \sqcup C(\ell_2),\ I = I(\ell_1) \sqcap I(\ell_2),\ A = A(\ell_1) \sqcap A(\ell_2)\}$$

The join operation is useful in analyzing computation. For example, suppose expressions e_1 and e_2 have security labels ℓ_1 and ℓ_2, respectively. Then $\ell_1 \sqcup \ell_2$ is the least restrictive label that can be assigned to the result of $e_1 + e_2$, as discussed in the previous section.

It is easy to show that (1) $\ell_1 \sqsubseteq \ell_1 \sqcup \ell_2$ and $\ell_2 \sqsubseteq \ell_1 \sqcup \ell_2$; (2) $\ell_1 \sqsubseteq \ell$ and $\ell_2 \sqsubseteq \ell$ imply $\ell_1 \sqcup \ell_2 \sqsubseteq \ell$. Thus, $\ell_1 \sqcup \ell_2$ is a least upper bound of ℓ_1 and ℓ_2 with respect to the $\sqsubseteq$ ordering. Based on the definition of the join operation on information labels, the bottom label for the $\sqsubseteq$ ordering is $\bot = \{C = \bot,\ I = \top,\ A = \top\}$, which satisfies $\bot \sqcup \ell = \ell$ if $\ell = \{C = l_1,\ I = l_2,\ A = l_3\}$.

2.4 Example

Consider the example in Figure 1.2. Now we can assign formal security labels to variables accessed by the program:

$$\texttt{bid}, \texttt{offer}, \texttt{t}, \texttt{a}, \texttt{result} : \ \ell_0 \qquad\qquad \texttt{acct} : \ \ell_1$$

where

$\ell_0 = \{C = \texttt{A}\wedge\texttt{B}:\texttt{A}\vee\texttt{B},\ I = \texttt{A}\wedge\texttt{B}:(\texttt{A}\wedge\texttt{B})\vee(\texttt{B}\wedge\texttt{T})\vee(\texttt{A}\wedge\texttt{T}),\ A = l\}$

$\ell_1 = \{C = \texttt{A}:\texttt{A},\ I = \texttt{A}:\texttt{A}\vee(\texttt{B}\wedge\texttt{T}),\ A = l\}$

$l = \texttt{A}\wedge\texttt{B}:(\texttt{A}\wedge\texttt{B})\vee(\texttt{B}\wedge\texttt{T})\vee(\texttt{A}\wedge\texttt{T})\vee(\texttt{C1}\wedge\texttt{C2})\vee(\texttt{C1}\wedge\texttt{C3})\vee(\texttt{C2}\wedge\texttt{C3})$

The label of `bid` is ℓ_0, in which `A` represents Alice, `B` represents Bob, and `T` is a third party helping to mediate the transaction. The confidentiality label $\texttt{A}\wedge\texttt{B}:\texttt{A}\vee\texttt{B}$ means that both `B` and `A` can learn the values of these variables with label ℓ_0. The integrity label indicates that affecting the value in `bid` requires the cooperation of at least two parties. For example, $\texttt{A}\wedge\texttt{B}$ can cooperatively affect `bid`, since they are the two directly involved parties. If `A` and `B` disagree on the value of `bid`, the mediator `T` can keep the transaction going by agreeing with the value claimed by either `B` or `A`. As a result, both $\texttt{A}\wedge\texttt{T}$ and $\texttt{B}\wedge\texttt{T}$ can affect the value of `bid`.

The availability component $A(\ell_0)$ is l, which assumes $\texttt{B}\wedge\texttt{A}$, $\texttt{B}\wedge\texttt{T}$, $\texttt{A}\wedge\texttt{T}$, $\texttt{C1}\wedge\texttt{C2}$, $\texttt{C1}\wedge\texttt{C3}$ and $\texttt{C2}\wedge\texttt{C3}$ to be non-compromised, where `C1`, `C2` and `C3` represent three clusters of hosts, and hosts in the same cluster are supposed to share the same failure causes. Principal `C1` fails if and only if all the hosts in `C1` fail; the same holds for `C2` and `C3`. This label assumes that at most one cluster among `C1`, `C2` and `C3` would fail, effectively specifying a failure model. Because all the variables share the same availability label l, the availability label of `result` cannot be violated by making other variables unavailable. Since $l \leq I(\ell_0)$, the availability label cannot be violated by compromising the integrity of `t`.

The label of `acct` is ℓ_1, which has a confidentiality component $\texttt{A} : \texttt{A}$, meaning that

Alice allows only herself to learn about `acct`. The integrity label of `acct` is $\texttt{A:A}\vee(\texttt{B}\wedge\texttt{T})$, meaning that `A` as well as $\texttt{B}\wedge\texttt{T}$ can affect the value of `acct`, because the transaction can proceed to charge `acct` as long as `B` and `T` cooperate.

2.5 Related work

Security labels have been widely used in security models and mechanisms for controlling information flow. Such security models include the Bell-LaPadula model [10], the secure information flow model [16], and the multilevel security model [23]. More recent models for information flow have defined various security properties that ensure the absence of insecure information flows, such as noninterference [31] for deterministic systems, possibilistic extensions of noninterference including nondeducibility [82] and generalized noninterference [54], and probabilistic extensions of noninterference including the Flow Model [57] and P-restrictiveness [32].

According to Denning [16], information flow control mechanisms fall into four categories based on whether they support static or dynamic binding of objects to labels and whether they are run-time or compile-time mechanisms.

- Run-time static-binding mechanisms include the access control mechanism of the MITRE system [9], which checks static-binding labels at run time to enforce the “no read-up and no write-down” rule. The Data Mark Machine proposed by Fenton [24] has an interesting run-time enforcement mechanism, in which labels are static except for the program counter label that is associated with the program counter of a process and may be updated at run time to control implicit flows. Recently, run-time label checking has been formalized as explicit language structures, and type systems [86, 106] have been developed to analyze run-time label checks statically.

- Compile-time static-binding mechanisms include the program certification mechanism proposed by Denning and Denning [18], and type-based information flow analyses [88, 34, 101, 70, 7].
- Run-time dynamic-binding mechanisms update the label of an object according to the changes to the contents of the object. This type of mechanisms was used in ADEPT [94], and more recently the IX system [55], the Flask security architecture [80] and the Asbestos system [20].
- Compile-time dynamic-binding mechanisms have been studied recently. Amtoft and Banerjee [6] developed a Hoare-like logic to track the independence relations between variables. Given a variable x, the set of variables that x depend on can be viewed as the security label of x, which may be different at different program points. Hunt and Sands [39] proposed a flow-sensitive type system for analyzing information flows. In the flow-sensitive type system, a variable may be assigned different types (including security labels) at different program points.

Owned-retained access control (ORAC) [53] uses owner-retained ACLs to label objects and enables flexible label management by allowing the owner of an object to modify its own ACL about the object.

Myers and Liskov proposed the decentralized label model for specifying information flow policies [63]. This thesis generalizes the DLM to provide a unified framework for specifying confidentiality, integrity and availability policies.

Focardi and Gorrieri [27] provide a classification of security properties in the setting of a non-deterministic process algebra. In particular, the BNDC (*bisimulation-based non-deducibility on compositions*) property prevents attackers from affecting the availabilities of observable process actions. However, the BNDC property requires observational equivalence, making it difficult to separate the concerns for integrity and availability.

Yu and Gligor [99] develop a formal method for analyzing availability: a form of first-order temporal logic is used to specify safety and liveness constraints on the inputs and behaviors of a service, and then those constraints can be used to formally verify the availability guarantees of the service. The flexibility and expressiveness of first-order temporal logic come at a price: it is difficult to automate the verification process. The approach of formalizing and reasoning system constraints and guarantees in terms of logic resembles the rely-guarantee method [40], which was also applied to analyzing cryptographic protocols by Guttman et al. [33].

The formalization of owned labels is inspired by the access control calculus [1], which introduces the formula A *says* s (principal A says s), and a principal logic with conjunctive and disjunctive principals. The purpose of the access control calculus is to determine whether access requests should be granted given a set of access control policies formalized as formulas in the calculus. In comparison, the universal DLM focuses on comparing decentralized labels.

Chapter 3
The Aimp language

As discussed in the previous chapter, security policies can be enforced by noninterference. It is well known that noninterference in terms of confidentiality and integrity can be enforced by static, compile-time analysis of program text [88, 34, 102, 73]. The new challenge is to apply the same approach to availability. This chapter presents the Aimp language with a security type system enforcing noninterference for three security properties: availability, along with confidentiality and integrity.

3.1 Syntax

The Aimp language is a simple imperative language with assignments, sequential composition, conditionals, and loops. The syntax of Aimp is as follows:

$$\begin{array}{rrcl} \text{Values} & v & ::= & n \\ \text{Expressions} & e & ::= & v \mid\ !m \mid e_1 + e_2 \\ \text{Statements} & s & ::= & \texttt{skip} \mid m := e \mid S_1; S_2 \\ & & \mid & \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ e\ \texttt{do}\ S \end{array}$$

In Aimp, a value is an integer n. An expression may be a value v, a dereference expression $!m$, or an addition expression $e_1 + e_2$. A statement may be an empty statement `skip`, an assignment statement $m := e$, a sequential composition $S_1; S_2$, or an `if` or `while` statement.

A program of Aimp is just a statement, and the state of a program is captured by a memory M that maps memory references (memory locations) to values. For simplicity, we assume that memory is observable and use memory references to model I/O channels. A reference representing an input is called an *input reference*, and a reference representing an output is called an *output reference*. To model availability, a memory reference may be mapped to two special values:

- `none`, indicating that the value of the reference is not available, and
- `void`, indicating that the reference itself is not available.

Intuitively, if a reference m is mapped to `none`, then a dereference operation on m will cause the running program to *get stuck* (cannot be further evaluated); if m is mapped to `void`, then either a dereference or assignment operation will cause the running program to get stuck. In particular, an output reference mapped to `none` represents an unavailable (not yet produced) output, and an input reference mapped to `void` represents an unavailable (remaining so during execution) input.

3.2 Operational semantics

The small-step operational semantics of Aimp is given in Figure 3.1. Let M represent a memory that is a finite map from locations to values (including `none` and `void`), and let $\langle S,\, M\rangle$ be a machine configuration. Then a small evaluation step is a transition from $\langle S,\, M\rangle$ to another configuration $\langle S',\, M'\rangle$, written $\langle S,\, M\rangle \longmapsto \langle S',\, M'\rangle$.

The evaluation rules (S1)–(S7) are standard for an imperative language. Rules (E1)–(E3) are used to evaluate expressions. Because an expression causes no side-effects to memory, we use the notation $\langle e,\, M\rangle \Downarrow v$ to mean that evaluating e in memory M results in the value v. Rule (E1) is used to evaluate a dereference expression $!m$. In rule (E1), Let $M[m]$ represent the value that m is mapped to in M. Then $M(m)$ is computed as follows:

$$M(m) = \begin{cases} n & \text{if} \quad M[m] = n \\ \texttt{none} & \text{if} \quad M[m] = \texttt{none} \text{ or } M[m] = \texttt{void} \end{cases}$$

Dereferencing a reference that is mapped to `none` or `void` produces an unavailable value, represented by `none`.

$$\text{(E1)} \qquad \frac{M(m) = v}{\langle !m,\, M\rangle \Downarrow v}$$

$$\text{(E2)} \qquad \frac{\langle e_1,\, M\rangle \Downarrow v_1 \qquad \langle e_2,\, M\rangle \Downarrow v_2 \qquad v = v_1 \oplus v_2}{\langle e_1 + e_2,\, M\rangle \Downarrow v}$$

$$\text{(E3)} \qquad \langle v,\, M\rangle \Downarrow v$$

$$\text{(S1)} \qquad \frac{\langle e,\, M\rangle \Downarrow n \qquad M[m] \neq \texttt{void}}{\langle m := e,\, M\rangle \longmapsto \langle \texttt{skip},\, M[m \mapsto n]\rangle}$$

$$\text{(S2)} \qquad \frac{\langle S_1,\, M\rangle \longmapsto \langle S_1',\, M'\rangle}{\langle S_1; S_2,\, M\rangle \longmapsto \langle S_1'; S_2,\, M'\rangle}$$

$$\text{(S3)} \qquad \langle \texttt{skip}; S,\, M\rangle \longmapsto \langle S,\, M\rangle$$

$$\text{(S4)} \qquad \frac{\langle e,\, M\rangle \Downarrow n \qquad n > 0}{\langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\, M\rangle \longmapsto \langle S_1,\, M\rangle}$$

$$\text{(S5)} \qquad \frac{\langle e,\, M\rangle \Downarrow n \qquad n \leq 0}{\langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\, M\rangle \longmapsto \langle S_2,\, M\rangle}$$

$$\text{(S6)} \qquad \frac{\langle e,\, M\rangle \Downarrow n \qquad n > 0}{\langle \texttt{while}\ e\ \texttt{do}\ S,\, M\rangle \longmapsto \langle S; \texttt{while}\ e\ \texttt{do}\ S,\, M\rangle}$$

$$\text{(S7)} \qquad \frac{\langle e,\, M\rangle \Downarrow n \qquad n \leq 0}{\langle \texttt{while}\ e\ \texttt{do}\ S,\, M\rangle \longmapsto \langle \texttt{skip},\, M\rangle}$$

Figure 3.1: Operational semantics for Aimp

Rule (E2) evaluates addition expressions. Intuitively, the sum of two values v_1 and v_2 is unavailable if v_1 or v_2 is unavailable. Accordingly, if e_1 and e_2 are evaluated to v_1 and v_2, $e_1 + e_2$ is evaluated to $v_1 \oplus v_2$, which is computed using the following formula:

$$v_1 \oplus v_2 = \begin{cases} n_1 + n_2 & \text{if} \quad v_1 = n_1 \text{ and } v_2 = n_2 \\ \texttt{none} & \text{if} \quad v_1 = \texttt{none} \text{ or } v_2 = \texttt{none} \end{cases}$$

Rules (S1)–(S7) are mostly self-explanatory. In rule (S1), the assignment to m can be accomplished only if m does not fail ($M[m] \neq \texttt{void}$). Rules (S1), (S4)–(S7) show that if the evaluation of configuration $\langle S,\, M\rangle$ depends on the result of an expression e, it must be the case that $\langle e,\, M\rangle \Downarrow n$. In other words, if $\langle e,\, M\rangle \Downarrow \texttt{none}$, the evaluation of

```
(A) m_2:=!m_1;     m_o:= 1;
(B) while (!m_1) do skip;    m_o:=1;
(C) if (!m_1) then while (1) do skip; else skip;
    m_o:=1;
(D) if (!m_1) then m_o:=1 else skip;
    while (!m_2) do skip;
    m_o:=2;
```

Figure 3.2: Examples

$\langle S,\, M\rangle$ gets stuck.

3.3 Examples

The Aimp language focuses on the essentials of an imperative language. Figure 3.2 shows a few code segments that demonstrate various kind of availability dependencies, some of which are subtle. In all these examples, m_o represents an output, and its initial value is `none`. All other references represent inputs.

In code segment (A), if m_1 or m_2 is not available (mapped to `void`), execution gets stuck at the first assignment. Therefore, the availability of m_o depends on the availability of m_1 and m_2.

In code segment (B), the `while` statement gets stuck if the value of m_1 is not available. Moreover, it *diverges* (goes into an infinite loop) if the value of m_1 is positive. Thus, availability of m_o depends on both the availability and integrity of m_1.

In code segment (C), the `if` statement diverges if the value of m_1 is positive, so the availability of m_o depends on the integrity of m_1.

In code segment (D), m_o is assigned in one branch of the `if` statement, but not in the other. Therefore, when the `if` statement terminates, the availability of m_o depends on the value of m_1. Moreover, the program executes a `while` statement that may diverge before m_o is assigned the value 2. Therefore, for the whole program, the availability of

```
t := 0; a := -1;
while (!t < 3)
  if (!bid >= !offer[i]) then
    acct := !acct + !bid; a := t;
    t := 5
  else t := !t + 1;
result := !a;
```

Figure 3.3: Bidding example

m_o depends on the integrity of m_1. Similar to (B), the availability of m_o also depends on the availability and integrity of m_2.

The Aimp language is expressive enough to write the bidding program in Figure 1.2, as shown in Figure 3.3. The array `offer` can be viewed as syntactic sugar for three references `offer1`, `offer2` and `offer3` that are accessed based on the value of `t`. Dereferencing variables is now represented explicitly with the operator !.

3.4 Type system

Let l range over a lattice $\mathcal{L}$ of base labels, such as the set of combined owned labels discussed in Section 2.3.2. The top and bottom elements of $\mathcal{L}$ are represented by $\top$ and $\bot$, respectively. The syntax for types in Aimp is shown as follows:

$$\begin{array}{rrcl} \text{Base labels} & l & \in & \mathcal{L} \\ \text{Labels} & \ell, pc & ::= & \{C = l_1,\ I = l_2,\ A = l_3\} \\ \text{Types} & \tau & ::= & \texttt{int}_\ell \mid \texttt{int}_\ell\,\texttt{ref} \mid \texttt{stmt}_\mathcal{R} \end{array}$$

In Aimp, the only data type is $\texttt{int}_\ell$, an integer type annotated with security label ℓ, which contains three base labels as described in Section 2.3.4. Suppose τ is $\texttt{int}_\ell$. Then we use the notations $C(\tau)$, $I(\tau)$ and $A(\tau)$ to represent $C(\ell)$, $I(\ell)$ and $A(\ell)$, respectively.

A memory reference m has type $\texttt{int}_\ell\,\texttt{ref}$, indicating the value stored at m has type $\texttt{int}_\ell$. In Aimp, types of memory references are specified by a *typing assignment* Γ that maps references to types so that the type of m is τ `ref` if $\Gamma(m) = \tau$.

The type of a statement S has the form $\mathtt{stmt}_{\mathcal{R}}$ where $\mathcal{R}$ contains the set of unassigned output references when S terminates. Intuitively, $\mathcal{R}$ represents all the outputs that are still expected by users after S terminates.

The type system of Aimp is designed to ensure that any well-typed Aimp program satisfies noninterference. For confidentiality and integrity, the type system performs a standard static information flow analysis [18, 88]. For availability, the type system tracks the set of unassigned output references at each program point. And the availability of an unassigned output reference at a program point depends on whether execution gets stuck at that program point. Such dependency relations induce label constraints that the type system of Aimp enforces, as the DP condition of Section 2.2 requires.

To track unassigned output references, the typing environment for a statement S includes a component $\mathcal{R}$, which contains the set of unassigned output references before the execution of S. The typing judgment for statements has the form: $\Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash S : \mathtt{stmt}_{\mathcal{R}'}$, meaning that S has type $\mathtt{stmt}_{\mathcal{R}'}$ with respect to the typing assignment Γ, the set of unassigned output references $\mathcal{R}$, and *pc*, the program counter label [17] used to indicate security levels of the program counter. The typing judgment for expressions has the form $\Gamma\,;\mathcal{R} \vdash e : \tau$, meaning that e has type τ with respect to Γ and $\mathcal{R}$.

The typing rules are shown in Figure 3.4. Rules (INT) and (NONE) check constants. An integer n has type $\mathtt{int}_{\ell}$ where ℓ can be an arbitrary label. The value `none` represents an unavailable value, so it can have any data type. Since `int` is the only data type in Aimp, `none` has type $\mathtt{int}_{\ell}$.

Rule (REF) says that the type of a reference m is τ `ref` where $\tau = \Gamma(m)$. In Aimp, $\Gamma(m)$ is always an integer type.

Rule (DEREF) checks dereference expressions. It disallows dereferencing the references in $\mathcal{R}$, because they may be unassigned output references.

(INT) $$\Gamma\,;\mathcal{R} \vdash n : \mathtt{int}_\ell$$

(NONE) $$\Gamma\,;\mathcal{R} \vdash \mathtt{none} : \mathtt{int}_\ell$$

(REF) $$\frac{\Gamma(m) = \tau}{\Gamma\,;\mathcal{R} \vdash m : \tau\ \mathtt{ref}}$$

(DEREF) $$\frac{m \notin \mathcal{R} \qquad \Gamma(m) = \mathtt{int}_\ell}{\Gamma\,;\mathcal{R} \vdash !m : \mathtt{int}_\ell}$$

(ADD) $$\frac{\Gamma\,;\mathcal{R} \vdash e_1 : \mathtt{int}_{\ell_1} \qquad \Gamma\,;\mathcal{R} \vdash e_2 : \mathtt{int}_{\ell_2}}{\Gamma\,;\mathcal{R} \vdash e_1 + e_2 : \mathtt{int}_{\ell_1 \sqcup \ell_2}}$$

(SKIP) $$\Gamma\,;\mathcal{R}\,;pc \vdash \mathtt{skip} : \mathtt{stmt}_\mathcal{R}$$

(SEQ) $$\frac{\begin{array}{c}\Gamma\,;\mathcal{R}\,;pc \vdash S_1 : \mathtt{stmt}_{\mathcal{R}_1}\\ \Gamma\,;\mathcal{R}_1\,;pc \vdash S_2 : \mathtt{stmt}_{\mathcal{R}_2}\end{array}}{\Gamma\,;\mathcal{R}\,;pc \vdash S_1; S_2 : \mathtt{stmt}_{\mathcal{R}_2}}$$

(ASSIGN) $$\frac{\begin{array}{c}\Gamma\,;\mathcal{R} \vdash m : \mathtt{int}_\ell\ \mathtt{ref} \qquad \Gamma\,;\mathcal{R} \vdash e : \mathtt{int}_{\ell'}\\ C(pc) \sqcup C(\ell') \le C(\ell) \quad I(\ell) \le I(pc) \sqcap I(\ell')\\ A_\Gamma(\mathcal{R}) \le A(\ell') \sqcap A(\ell)\end{array}}{\Gamma\,;\mathcal{R}\,;pc \vdash m := e : \mathtt{stmt}_{\mathcal{R}-\{m\}}}$$

(IF) $$\frac{\begin{array}{c}\Gamma\,;\mathcal{R} \vdash e : \mathtt{int}_\ell \qquad A_\Gamma(\mathcal{R}) \le A(\ell)\\ \Gamma\,;\mathcal{R}\,;pc \sqcup \ell \vdash S_i : \tau \quad i \in \{1,2\}\end{array}}{\Gamma\,;\mathcal{R}\,;pc \vdash \mathtt{if}\ e\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2 : \tau}$$

(WHILE) $$\frac{\begin{array}{c}\Gamma \vdash e : \mathtt{int}_\ell \qquad \Gamma\,;\mathcal{R}\,;pc \sqcup \ell \vdash S : \mathtt{stmt}_\mathcal{R}\\ A_\Gamma(\mathcal{R}) \le I(\ell) \sqcap I(pc) \sqcap A(\ell)\end{array}}{\Gamma\,;\mathcal{R}\,;pc \vdash \mathtt{while}\ e\ \mathtt{do}\ S : \mathtt{stmt}_\mathcal{R}}$$

(SUB) $$\frac{\Gamma\,;\mathcal{R}\,;pc \vdash S : \tau \qquad \Gamma\,;\mathcal{R}\,;pc \vdash \tau \le \tau'}{\Gamma\,;\mathcal{R}\,;pc \vdash S : \tau'}$$

Figure 3.4: Typing rules for Aimp

Rule (ADD) checks addition expressions. As discussed in Section 2.3.4, the label of $e_1 + e_2$ is exactly $\ell_1 \sqcup \ell_2$ if e_i has the label ℓ_i for $i \in \{1, 2\}$.

Rule (SKIP) checks the `skip` statement, which does not have any effects. Thus, the unassigned output references are still $\mathcal{R}$ after executing `skip`.

Rule (SEQ) checks sequential statements. The premise $\Gamma\,;\mathcal{R}\,;pc \vdash S_1 : \mathtt{stmt}_{\mathcal{R}_1}$

means that $\mathcal{R}_1$ contains the set of unassigned output references after S_1 terminates and before S_2 starts. Therefore, the typing environment for S_2 is $\Gamma\,;\mathcal{R}_1\,;pc$. It is clear that S_2 and $S_1; S_2$ terminate at the same point. Thus, $S_1; S_2$ has the same type as S_2.

Rule (ASSIGN) checks assignment statements. The statement $m := e$ assigns the value of e to m, creating an explicit information flow from e to m and an implicit flow from the program counter to m. To control these information flows, this rule requires $C(\ell') \sqcup C(pc) \leq C(\Gamma(m))$ to protect the confidentiality of e and the program counter, and $I(\Gamma(m)) \leq I(pc) \sqcap I(\ell')$ to protect the integrity of m.

If the value of e is unavailable or the reference m fails, the assignment $m := e$ will get stuck. Therefore, rule (ASSIGN) has the premise $A_\Gamma(\mathcal{R}) \leq A(\ell') \sqcap A(\ell)$, where $A_\Gamma(\mathcal{R}) = \bigsqcup_{m\in\mathcal{R}} A(\Gamma(m))$, to ensure the availability labels of e and m is as high as the availability label of any unassigned output reference. For example, in the code segment (A) of Figure 3.2, the type system ensures that $A(\Gamma(m_o)) \leq A(\Gamma(m_1)) \sqcap A(\Gamma(m_2))$.

When the assignment $m := e$ terminates, m should be removed from the set of unassigned output references, and thus the statement has type $\mathtt{stmt}_{\mathcal{R}-\{m\}}$.

Rule (IF) checks `if` statements. Consider the statement: `if` e `then` S_1 `else` S_2. The value of e determines which branch is executed, so the program-counter labels for branches S_1 and S_2 subsume the label of e to protect e from implicit flows. As usual, the `if` statement has type τ if both S_1 and S_2 have type τ. As in rule (ASSIGN), the premise $A_\Gamma(R) \leq A(\ell)$ ensures that e has sufficient availability.

Rule (WHILE) checks `while` statements. In this rule, the premise $A_\Gamma(\mathcal{R}) \leq I(\ell) \sqcap I(pc) \sqcap A(\ell)$ can be decomposed into three constraints: $A_\Gamma(\mathcal{R}) \leq A(\ell)$, which ensures that e has sufficient availability, $A_\Gamma(\mathcal{R}) \leq I(\ell)$, which prevents attackers from making the `while` statement diverge by compromising the integrity of e, and $A_\Gamma(\mathcal{R}) \leq I(pc)$, which prevents attackers from affecting whether the control flow reaches the `while` statement, because a `while` statement may diverge without any interaction with attackers.

For example, consider the code segments (B) and (C) in Figure 3.2, in which $\mathcal{R} = \{m_o\}$. Suppose base label $l_{\mathtt{A}}$ represents the security level of attackers, and $A(\Gamma(m_o)) \not\leq l_{\mathtt{A}}$. In (B), the constraint $A_\Gamma(\mathcal{R}) \leq I(\ell)$ of rule (WHILE) ensures $I(\Gamma(m_1)) \not\leq l_{\mathtt{A}}$, so attackers cannot affect the value of m_1, or whether the `while` statement diverges. In (C), the constraint $A_\Gamma(\mathcal{R}) \leq I(pc)$ guarantees $I(pc) \not\leq l_{\mathtt{A}}$, and thus $I(\Gamma(m_1)) \not\leq l_{\mathtt{A}}$ holds because $I(pc) \leq I(\Gamma(m_1))$. Therefore, attackers cannot affect which branch of the `if` statement would be taken, or whether control reaches the `while` statement.

Rule (SUB) is the standard subsumption rule. Let $\Gamma\,;\mathcal{R}\,;pc \vdash \tau \leq \tau'$ denote that τ is a subtype of τ' with respect to the typing environment $\Gamma\,;\mathcal{R}\,;pc$. The type system of Aimp has one subtyping rule:

$$\text{(ST)}\qquad \frac{\begin{array}{c}\mathcal{R}' \subseteq \mathcal{R}'' \subseteq \mathcal{R}\\ \forall m,\ \ m \in \mathcal{R}'' - \mathcal{R}' \Rightarrow A(\Gamma(m)) \leq I(pc)\end{array}}{\Gamma\,;\mathcal{R}\,;pc \vdash \mathtt{stmt}_{\mathcal{R}'} \leq \mathtt{stmt}_{\mathcal{R}''}}$$

Suppose $\Gamma\,;\mathcal{R}\,;pc \vdash \mathtt{stmt}_{\mathcal{R}'} \leq \mathtt{stmt}_{\mathcal{R}''}$ and $\Gamma\,;\mathcal{R}\,;pc \vdash S : \mathtt{stmt}_{\mathcal{R}'}$. Then $\Gamma\,;\mathcal{R}\,;pc \vdash S : \mathtt{stmt}_{\mathcal{R}''}$ by rule (SUB). In other words, if $\mathcal{R}'$ contains all the unassigned output references after S terminates, so does $\mathcal{R}''$. This is guaranteed by the premise $\mathcal{R}' \subseteq \mathcal{R}''$ of rule (ST). The reference set $\mathcal{R}$ contains all the unassigned output references before S is executed, so rule (ST) requires $\mathcal{R}'' \subseteq \mathcal{R}$. Intuitively, the statement S can be treated as having type $\mathtt{stmt}_{\mathcal{R}''}$ because there might exist another control flow path that bypasses S and does not assign to references in $\mathcal{R}'' - \mathcal{R}'$. Consequently, for any m in $\mathcal{R}'' - \mathcal{R}'$, the availability of m may depend on whether S is executed. Therefore, rule (ST) enforces the constraint $\forall m,\ \ m \in \mathcal{R}'' - \mathcal{R}' \Rightarrow A(\Gamma(m)) \leq I(pc)$.

Consider the assignment $m_o := 1$ in the code segment (D) of Figure 3.2. By rule (ASSIGN), $\Gamma\,;\{m_o\}\,;pc \vdash m_o := 0 : \mathtt{stmt}_\emptyset$. For the `else` branch of the `if` statement, we have $\Gamma\,;\{m_o\}\,;pc \vdash \mathtt{skip} : \mathtt{stmt}_{\{m_o\}}$. By rule (IF), $\Gamma\,;\{m_o\}\,;pc \vdash m_o := 0 : \mathtt{stmt}_{\{m_o\}}$ needs to hold, which requires $\Gamma\,;\{m_o\}\,;pc \vdash \mathtt{stmt}_\emptyset \leq \mathtt{stmt}_{\{m_o\}}$. In this example, the availability of m_o depends on which branch is taken, and we need to ensure $A(\Gamma(m_o)) \leq I(\Gamma(m_1))$. Indeed, if (D) is well typed, by rules (ST) and (IF), we have $A(\Gamma(m_o)) \leq I(pc) \leq I(\Gamma(m_1))$.

This type system satisfies the property of subject reduction, or type preservation, as stated in the following theorem, which is proved in the next section.

Theorem 3.4.1 (Subject reduction). Suppose $\Gamma\,;\mathcal{R}\,;pc \vdash S : \tau$, and $dom(\Gamma) = dom(M)$. If $\langle S,\, M\rangle \longmapsto \langle S',\, M'\rangle$, then there exists $\mathcal{R}'$ such that $\Gamma\,;\mathcal{R}'\,;pc \vdash S' : \tau$, and $\mathcal{R}' \subseteq \mathcal{R}$, and for any $m \in \mathcal{R} - \mathcal{R}'$, $M'(m) \neq \texttt{none}$.

3.5 Security by type checking

As discussed in Section 2.2, security policies can be enforced by noninterference. This section shows that the type system of Aimp can enforce the security policies specified by type annotations (labels), by proving that every well-typed program satisfies noninterference.

3.5.1 Noninterference properties

In general, a program can affect three security properties: the confidentiality of an input, the integrity of an output and the availability of an output. Thus, the notion of noninterference can be formalized as three more specific noninterference properties, corresponding to the three security properties. Although this formalization is done in the context of Aimp, it can be easily generalized to other state transition systems.

For both confidentiality and integrity, noninterference has a simple, intuitive description: equivalent low-confidentiality (high-integrity) inputs always result in equivalent low-confidentiality (high-integrity) outputs. The notion of availability noninterference is more subtle, because an attacker has two ways to compromise the availability of an output. First, the attacker can make an input unavailable and block computation that depends on the input. Second, the attacker can try to affect the integrity of control flow and make the program diverge (fail to terminate). In other words, the availability of an output may depend on both the integrity and availability of an input. The observation is captured by this intuitive description of availability noninterference:

> With all high-availability inputs available, equivalent high-integrity inputs will eventually result in equally available high-availability outputs.

This formulation of noninterference provides a separation of concerns (and policies) for availability and integrity, yet prevents the two attacks discussed above.

The intuitive concepts of high and low security are based on the power of the potential attacker, which is represented by a base label $l_{\mathtt{A}}$. In the DLM, $l_{\mathtt{A}} = \{* : p_1 \wedge \ldots \wedge p_n\}$, if the attacker can act for $p_1, \ldots, p_n$. Given a base label l, if $l \leq l_{\mathtt{A}}$ then the label represents a low-security level and is not protected from the attacker. Otherwise, l is a high-security label.

For an imperative language, the inputs of a program are just the initial memory and the outputs are the observable aspects of a program execution, which is defined by the *observation model* of the language. In Aimp, we have the following observation model:

- Memories are observable.
- The value `none` is not observable. In other words, if $M(m) = \mathtt{none}$, an observer cannot determine the value of m in M.

Suppose S is a program, and M is the initial memory. Based on the observation model, the outputs of S are a set $\mathcal{T}$ of finite traces of memories, and for any trace T in $\mathcal{T}$, there exists an evaluation $\langle S,\, M\rangle \longmapsto \langle S_1,\, M_1\rangle \longmapsto \ldots \longmapsto \langle S_n,\, M_n\rangle$ such that $T = [M, M_1, \ldots, M_n]$. Intuitively, every trace in $\mathcal{T}$ is the outputs observable to users at some point during the evaluation of $\langle S,\, M\rangle$, and $\mathcal{T}$ represents all the outputs of $\langle S,\, M\rangle$

observable to users. Since the Aimp language is deterministic, for any two traces in $\mathcal{T}$, it must be the case that one is a prefix of the other.

In the intuitive description of noninterference, equivalent low-confidentiality inputs can be represented by two memories whose low-confidentiality parts are indistinguishable. Suppose the typing information of a memory M is given by a typing assignment Γ. Then m belongs to the low-confidentiality part of M if $C(\Gamma(m)) \leq l_{\mathtt{A}}$, where $C(\Gamma(m)) = C(\ell)$ if $\Gamma(m) = \mathtt{int}_\ell$. Similarly, m is a low-integrity reference if $I(\Gamma(m)) \leq l_{\mathtt{A}}$, a high-integrity reference if $I(\Gamma(m)) \not\leq l_{\mathtt{A}}$, and a high-availability reference if $A(\Gamma(m)) \not\leq l_{\mathtt{A}}$. Let $v_1 \approx v_2$ denote that v_1 and v_2 are indistinguishable. By the observation model of Aimp, a user cannot distinguish `none` from any other value. Consequently, $v_1 \approx v_2$ if and only if $v_1 = v_2$, $v_1 = \mathtt{none}$ or $v_2 = \mathtt{none}$. With these settings, given two memories M_1 and M_2 with respect to Γ, we define three kinds of indistinguishability relations between M_1 and M_2 as follows:

Definition 3.5.1 ($\Gamma \vdash M_1 \approx_{C\leq l_{\mathtt{A}}} M_2$). The low-confidentiality parts of M_1 and M_2 are indistinguishable, written $\Gamma \vdash M_1 \approx_{C\leq l_{\mathtt{A}}} M_2$, if for any $m \in \mathit{dom}(\Gamma)$, $C(\Gamma(m)) \leq l_{\mathtt{A}}$ implies $M_1(m) \approx M_2(m)$.

Definition 3.5.2 ($\Gamma \vdash M_1 \approx_{I\not\leq l_{\mathtt{A}}} M_2$). The high-integrity parts of M_1 and M_2 are indistinguishable, written $\Gamma \vdash M_1 \approx_{I\not\leq l_{\mathtt{A}}} M_2$, if for any $m \in \mathit{dom}(\Gamma)$, $I(\Gamma(m)) \not\leq l_{\mathtt{A}}$ implies $M_1(m) \approx M_2(m)$.

Definition 3.5.3 ($\Gamma \vdash M_1 \approx_{A\not\leq l_{\mathtt{A}}} M_2$). The high-availability parts of M_1 and M_2 are equally available, written $\Gamma \vdash M_1 \approx_{A\not\leq l_{\mathtt{A}}} M_2$, if for any $m \in \mathit{dom}(\Gamma)$, $A(\Gamma(m)) \not\leq l_{\mathtt{A}}$ implies that $M_1(m) = \mathtt{none}$ if and only if $M_2(m) = \mathtt{none}$.

Based on the definitions of memory indistinguishability, we can define trace indistinguishability, which formalizes the notion of equivalent outputs. Intuitively, two traces are indistinguishable if they may be produced by the same execution. First, we assume that users cannot observe timing. As a result, traces $[M, M]$ and $[M]$ look the same to a user. In general, two traces T_1 and T_2 are equivalent, written $T_1 \approx T_2$, if they are equal up to stuttering, which means the two traces obtained by eliminating repeated elements in T_1 and T_2 are equal. For example, $[M_1, M_2, M_2] \approx [M_1, M_1, M_2]$. Second, T_1 and T_2 are indistinguishable, if T_1 appears to be a prefix of T_2, because in that case, T_1 and T_2 may be generated by the same execution. This implies that trace indistinguishability is not an equivalence relation because two distinguishable traces may share the same prefix.

Given two traces T_1 and T_2 of memories with respect to Γ, let $\Gamma \vdash T_1 \approx_{C \leq l_{\mathtt{A}}} T_2$ denote that the low-confidentiality parts of T_1 and T_2 are indistinguishable, and $\Gamma \vdash T_1 \approx_{I \not\leq l_{\mathtt{A}}} T_2$ denote that the high-integrity parts of T_1 and T_2 are indistinguishable. These two notions are defined as follows:

Definition 3.5.4 ($\Gamma \vdash T_1 \approx_{C \leq l_{\mathtt{A}}} T_2$). Given two traces T_1 and T_2, $\Gamma \vdash T_1 \approx_{C \leq l_{\mathtt{A}}} T_2$ if there exist $T_1' = [M_1, \ldots, M_n]$ and $T_2' = [M_1', \ldots, M_m']$ such that $T_1 \approx T_1'$, and $T_2 \approx T_2'$, and $\Gamma \vdash M_i \approx_{C \leq l_{\mathtt{A}}} M_i'$ for any i in $\{1, \ldots, \mathtt{min}(m, n)\}$.

Definition 3.5.5 ($\Gamma \vdash T_1 \approx_{I \not\leq l_{\mathtt{A}}} T_2$). Given two traces T_1 and T_2, $\Gamma \vdash T_1 \approx_{I \not\leq l_{\mathtt{A}}} T_2$ if there exist $T_1' = [M_1, \ldots, M_n]$ and $T_2' = [M_1', \ldots, M_m']$ such that $T_1 \approx T_1'$, and $T_2 \approx T_2'$, and $\Gamma \vdash M_i \approx_{I \not\leq l_{\mathtt{A}}} M_i'$ for any i in $\{1, \ldots, \mathtt{min}(m, n)\}$.

Note that two executions are indistinguishable if any two finite traces generated by those two executions are indistinguishable. Thus, we can still reason about the indistinguishability of two nonterminating executions, even though $\approx_{I \not\leq l_{\mathtt{A}}}$ and $\approx_{C \leq l_{\mathtt{A}}}$ are defined on finite traces.

With the formal definitions of memory indistinguishability and trace indistinguishability, it is straightforward to formalize confidentiality noninterference and integrity noninterference:

Definition 3.5.6 (Confidentiality noninterference). A program S has the *confidentiality noninterference* property w.r.t. a typing assignment Γ, written $\Gamma \vdash \mathtt{NI}_C(S)$, if for any two traces T_1 and T_2 generated by evaluating $\langle S,\ M_1\rangle$ and $\langle S,\ M_2\rangle$, we have that $\Gamma \vdash M_1 \approx_{C\leq l_\mathtt{A}} M_2$ implies $\Gamma \vdash T_1 \approx_{C\leq l_\mathtt{A}} T_2$.

Note that this confidentiality noninterference property does not treat covert channels based on termination and timing. Static control of timing channels is largely orthogonal to this work, and has been partially addressed elsewhere [79, 3, 72].

Definition 3.5.7 (Integrity noninterference). A program S has the *integrity noninterference* property w.r.t. a typing assignment Γ, written $\Gamma \vdash \mathtt{NI}_I(S)$, if for any two traces T_1 and T_2 generated by evaluating $\langle S,\ M_1\rangle$ and $\langle S,\ M_2\rangle$, we have that $\Gamma \vdash M_1 \approx_{I\not\leq l_\mathtt{A}} M_2$ implies $\Gamma \vdash T_1 \approx_{I\not\leq l_\mathtt{A}} T_2$.

Consider the intuitive description of availability noninterference. To formalize the notion that all the high-availability inputs are available, we need to distinguish input references from unassigned output references. Given a program S, let $\mathcal{R}$ denote the set of unassigned output references. In general, references in $\mathcal{R}$ are mapped to `none` in the initial memory. If $m \notin \mathcal{R}$, then reference m represents either an input, or an output that is already been generated. Thus, given an initial memory M, the notion that all the high-availability inputs are available can be formalized as $\forall m.\ (A(\Gamma(m)) \not\leq l_\mathtt{A} \wedge m \notin \mathcal{R}) \Rightarrow M(m) \neq \mathtt{none}$, as in the following definition of availability noninterference:

Definition 3.5.8 (Availability noninterference). A program S has the *availability noninterference* property w.r.t. a typing assignment Γ and a set of unassigned output references $\mathcal{R}$, written $\Gamma\,;\mathcal{R} \vdash \mathtt{NI}_A(S)$, if for any two memories M_1, M_2, the following statements

- $\Gamma \vdash M_1 \approx_{I\not\leq l_\mathtt{A}} M_2$
- For $i \in \{1, 2\}$, $\forall m \in \mathit{dom}(\Gamma).\ A(\Gamma(m)) \not\leq l_\mathtt{A} \wedge m \notin \mathcal{R} \Rightarrow M_i(m) \neq \mathtt{none}$

- $\langle S,\, M_i \rangle \longmapsto^* \langle S'_i,\, M'_i \rangle$ for $i \in \{1, 2\}$

imply that there exist $\langle S''_i,\, M''_i \rangle$ for $i \in \{1, 2\}$ such that $\langle S'_i,\, M'_i \rangle \longmapsto^* \langle S''_i,\, M''_i \rangle$ and $\Gamma \vdash M''_1 \approx_{A \not\leq l_{\mathtt{A}}} M''_2$.

3.5.2 The Aimp* language

The noninterference result for Aimp is proved by extending the language to a new language Aimp*. Each configuration $\langle S,\, M \rangle$ in Aimp* encodes two Aimp configurations $\langle S_1,\, M_1 \rangle$ and $\langle S_2,\, M_2 \rangle$. Moreover, the operational semantics of Aimp* is consistent with that of Aimp in the sense that the result of evaluating $\langle S,\, M \rangle$ is an encoding of the results of evaluating $\langle S_1,\, M_1 \rangle$ and $\langle S_2,\, M_2 \rangle$ in Aimp. The type system of Aimp* ensures that if $\langle S,\, M \rangle$ is well-typed, then the low-confidentiality or high-integrity parts of $\langle S_1,\, M_1 \rangle$ and $\langle S_2,\, M_2 \rangle$ are equivalent. Intuitively, if the result of $\langle S,\, M \rangle$ is well-typed, then the results of evaluating $\langle S_1,\, M_1 \rangle$ and $\langle S_2,\, M_2 \rangle$ should also have equivalent low-confidentiality or high-integrity parts. Therefore, the preservation of type soundness in an Aimp* evaluation implies the preservation of low-confidentiality or high-integrity equivalence between two Aimp evaluations. Thus, to prove the confidentiality and integrity noninterference theorems of Aimp, we only need to prove the subject reduction theorem of Aimp*. This proof technique was first used by Pottier and Simonet to prove the noninterference result of a security-typed ML-like language [70].

What is new here is that the availability noninterference theorem of Aimp can by proved by a *progress* property of the type system of Aimp*.

This section details the syntax and semantic extensions of Aimp* and proves the key subject reduction and progress theorems of Aimp*.

Syntax extensions

The syntax extensions of Aimp* include the bracket construct, which is composed of two Aimp terms and captures the difference between two Aimp configurations.

$$\begin{array}{rrcl} \text{Values} & v & ::= & \ldots \mid (v_1 \mid v_2) \\ \text{Statements} & S & ::= & \ldots \mid (S_1 \mid S_2) \end{array}$$

Bracket constructs cannot be nested, so the subterms of a bracket construct must be Aimp terms. Given an Aimp* statement S, let $\lfloor S \rfloor_1$ and $\lfloor S \rfloor_2$ represent the two Aimp statements that S encodes. The projection functions satisfy $\lfloor (S_1 \mid S_2) \rfloor_i = S_i$ and are homomorphisms on other statement and value forms. An Aimp* memory M maps references to Aimp* values that encode two Aimp values. Thus, the projection function can be defined on memories too. For $i \in \{1, 2\}$, $\mathit{dom}(\lfloor M \rfloor_i) = \mathit{dom}(M)$, and for any $m \in \mathit{dom}(M)$, $\lfloor M \rfloor_i(m) = \lfloor M(m) \rfloor_i$.

Since an Aimp* term effectively encodes two Aimp terms, evaluation of an Aimp* term can be projected into two Aimp evaluations. An evaluation step of a bracket statement $(S_1 \mid S_2)$ is an evaluation step of either S_1 or S_2, and S_1 or S_2 can only access the corresponding projection of the memory. Thus, the configuration of Aimp* has an index $i \in \{\bullet, 1, 2\}$ that indicates whether the term to be evaluated is a subterm of a bracket expression, and if so, which branch of a bracket the term belongs to. For example, the configuration $\langle S,\, M \rangle_1$ means that S belongs to the first branch of a bracket, and S can only access the first projection of M. We write "$\langle S,\, M \rangle$" for "$\langle S,\, M \rangle_\bullet$", which means S does not belong to any bracket. To abuse notation a bit, let $\lfloor S \rfloor_\bullet = S$ and $\lfloor v \rfloor_\bullet = v$.

The operational semantics of Aimp* is shown in Figure 3.5. It is based on the semantics of Aimp and contains some new evaluation rules (S8)–(S9) for manipulating bracket constructs. Rules (E1) and (S1) are modified to access the memory projection corresponding to index i. The function $v[v'/\pi_i]$ returns the value obtained by replacing the ith component of v with v'. The rest of the rules in Figure 5.2 are adapted to Aimp*

(E1) $\dfrac{\lfloor M(m) \rfloor_i = v}{\langle !m,\, M \rangle_i \Downarrow v}$

(S1) $\dfrac{\langle e,\, M \rangle_i \Downarrow v \qquad \lfloor v \rfloor_1 \neq \texttt{none} \quad \lfloor v \rfloor_2 \neq \texttt{none} \qquad \lfloor M[m] \rfloor_i \neq \texttt{void}}{\langle m := e,\, M \rangle_i \longmapsto \langle \texttt{skip},\, M[m \mapsto M[m][v/\pi_i]] \rangle_i}$

(S8) $\dfrac{\langle e,\, M \rangle \Downarrow (n_1 \mid n_2)}{\begin{array}{l}\langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\, M \rangle \longmapsto \\ \quad \langle (\texttt{if}\ n_1\ \texttt{then}\ \lfloor S_1 \rfloor_1\ \texttt{else}\ \lfloor S_2 \rfloor_1 \mid \\ \qquad \texttt{if}\ n_2\ \texttt{then}\ \lfloor S_1 \rfloor_2\ \texttt{else}\ \lfloor S_2 \rfloor_2), M \rangle\end{array}}$

(S9) $\dfrac{\langle S_i,\, M \rangle_i \longmapsto \langle S_i',\, M' \rangle_i \qquad S_j = S_j' \qquad \{i, j\} = \{1, 2\}}{\langle (S_1 \mid S_2),\, M \rangle \longmapsto \langle (S_1' \mid S_2'),\, M' \rangle}$

(S10) $\langle (\texttt{skip} \mid \texttt{skip}),\, M \rangle \longmapsto \langle \texttt{skip},\, M \rangle$

[Auxiliary functions]

$$v[v'/\pi_\bullet] = v' \qquad v[v'/\pi_1] = (v' \mid \lfloor v \rfloor_2) \qquad v[v'/\pi_2] = (\lfloor v \rfloor_1 \mid v')$$

Figure 3.5: The operational semantics of Aimp*

by indexing each configuration with i. The following adequacy and soundness lemmas state that the operational semantics of Aimp* is adequate to encode the execution of two Aimp terms.

Let the notation $\langle S,\, M \rangle \longmapsto^T \langle S',\, M' \rangle$ denote that $\langle S,\, M \rangle \longmapsto \langle S_1,\, M_1 \rangle \longmapsto \ldots \longmapsto \langle s_n,\, M_n \rangle \longmapsto \langle S',\, M' \rangle$ and $T = [M, M_1, \ldots, M_n, M']$, or $S = S'$ and $M = M'$ and $T = [M]$. In addition, let $|T|$ denote the length of T, and $T_1 \oplus T_2$ denote the trace obtained by concatenating T_1 and T_2. Suppose $T_1 = [M_1, \ldots, M_n]$ and $T_2 = [M_1', \ldots, M_m']$. If $M_n = M_1'$, then $T_1 \oplus T_2 = [M_1, \ldots, M_n, M_2', \ldots, M_m']$. Otherwise, $T_1 \oplus T_2 = [M_1, \ldots, M_n, M_1', \ldots, M_m']$.

Lemma 3.5.1 (Projection i). Suppose $\langle e,\, M \rangle \Downarrow v$. Then $\langle \lfloor e \rfloor_i,\, \lfloor M \rfloor_i \rangle \Downarrow \lfloor v \rfloor_i$ holds for $i \in \{1, 2\}$.

Proof. By induction on the derivation of $\langle e,\, M \rangle \Downarrow v$.

- Case (E1). v is $M(m)$. Thus, $\lfloor v \rfloor_i = \lfloor M(m) \rfloor_i = \lfloor M \rfloor_i(m)$.

- Case (E2). By induction, $\langle \lfloor e_1 \rfloor_i,\ \lfloor M \rfloor_i \rangle \Downarrow \lfloor v_1 \rfloor_i$ and $\langle \lfloor e_2 \rfloor_i,\ \lfloor M \rfloor_i \rangle \Downarrow \lfloor v_2 \rfloor_i$. Thus, $\langle \lfloor e_1 + e_2 \rfloor_i,\ \Downarrow \rangle \lfloor v_1 \oplus v_2 \rfloor_i$.
- Case (E3). e is v. Thus, $\langle \lfloor v \rfloor_i,\ \lfloor M \rfloor_i \rangle \Downarrow \lfloor v \rfloor_i$.

□

Lemma 3.5.2 (Projection ii). Suppose M is an Aimp* memory, and $\lfloor M \rfloor_i = M_i$ for $i \in \{1, 2\}$, and $\langle S,\ M_i \rangle$ is an Aimp configuration. Then $\langle S,\ M_i \rangle \longmapsto \langle S',\ M_i' \rangle$ if and only if $\langle S,\ M \rangle_i \longmapsto \langle S',\ M' \rangle_i$ and $\lfloor M' \rfloor_i = M_i'$.

Proof. By induction on the structure of S. □

Lemma 3.5.3 (Expression adequacy). Suppose $\langle e_i,\ M_i \rangle \Downarrow v_i$ for $i \in \{1, 2\}$, and there exists an Aimp* configuration $\langle e,\ M \rangle$ such that $\lfloor e \rfloor_i = e_i$ and $\lfloor \mathrm{M} \rfloor_i = M_i$ for $i \in \{1, 2\}$. Then $\langle e,\ M \rangle \Downarrow v$ such that $\lfloor v \rfloor_i = v_i$.

Proof. By induction on the structure of e. □

Lemma 3.5.4 (One-step adequacy). If for $i \in \{1, 2\}$, $\langle S_i,\ M_i \rangle \longmapsto \langle S_i',\ M_i' \rangle$ is an evaluation in Aimp, and there exists $\langle S,\ M \rangle$ in Aimp* such that $\lfloor S \rfloor_i = S_i$ and $\lfloor M \rfloor_i = M_i$, then there exists $\langle S',\ M' \rangle$ such that $\langle S,\ M \rangle \longmapsto^T \langle S',\ M' \rangle$, and one of the following conditions holds:

i. For $i \in \{1, 2\}$, $\lfloor T \rfloor_i \approx [M_i, M_i']$ and $\lfloor S' \rfloor_i = S_i'$.

ii. For $\{j, k\} = \{1, 2\}$, $\lfloor T \rfloor_j \approx [M_j]$ and $\lfloor S' \rfloor_j = S_j$, and $\lfloor T \rfloor_k \approx [M_k, M_k']$ and $\lfloor S' \rfloor_k = S_k'$.

Proof. By induction on the structure of S.

- S is `skip`. Then S_1 and S_2 are also `skip` and cannot be further evaluated. Therefore, the lemma is correct because its premise does not hold.

- S is $m := e$. In this case, S_i is $m := \lfloor e \rfloor_i$, and we have $\langle m := \lfloor e \rfloor_i,\, M_i \rangle \longmapsto \langle \texttt{skip},\, M_i[m \mapsto v_i] \rangle$ where $\langle \lfloor e \rfloor_i,\, M_i \rangle \Downarrow v_i$. By Lemma 3.5.3, we have $\langle e,\, M \rangle \Downarrow v$ and $\lfloor v \rfloor_i = v_i$. Therefore, $\langle m := e,\, M \rangle \longmapsto \langle \texttt{skip},\, M[m \mapsto v] \rangle$. Since $\lfloor M \rfloor_i = M_i$, we have $\lfloor M[m \mapsto v] \rfloor_i = M_i[m \mapsto \lfloor v \rfloor_i]$.

- S is $\texttt{if}\ e\ \texttt{then}\ S_1''\ \texttt{else}\ S_2''$. Suppose $\langle e_i,\, M_i \rangle \Downarrow n_i$. By Lemma 3.5.3, $\langle e,\, M \rangle \Downarrow v$ such that $\lfloor v \rfloor_i = n_i$ for $i \in \{1, 2\}$. Since S_i is $\texttt{if}\ \lfloor e \rfloor_i\ \texttt{then}\ \lfloor S_1'' \rfloor_i\ \texttt{else}\ \lfloor S_2'' \rfloor_i$ for $i \in \{1, 2\}$, S_i' is $\lfloor S_{j_i}'' \rfloor_i$ where $j_i \in \{1, 2\}$. If $v = n$, then $\langle S,\, M \rangle \longmapsto^T \langle S_j'',\, M \rangle$ for some j in $\{1, 2\}$, and $j_i = j$ for $i \in \{1, 2\}$. If $v = (n_1 \,|\, n_2)$, then $\langle S,\, M \rangle \longmapsto^T \langle (\lfloor S_{j_1}'' \rfloor_1 \mid \lfloor S_{j_2}'' \rfloor_2),\, M \rangle$, where $j_1, j_2 \in \{1, 2\}$. In both cases, $\lfloor S_i' \rfloor_i = S_i'$ for $i \in \{1, 2\}$ and $T \approx [M, M]$.

- S is $\texttt{while}\ e\ \texttt{do}\ S''$. By the same argument as the above case.

- S is $S_3; S_4$. There are three cases:

  - S_3 is $\texttt{skip}$ or $(\texttt{skip} \mid \texttt{skip})$. Then $\langle S,\, M \rangle \longmapsto^T \langle S_4,\, M \rangle$, and $T \approx [M]$. For $i \in \{1, 2\}$, since $S_i = \texttt{skip}; \lfloor S_4 \rfloor_i$, $\langle S_i,\, \lfloor M \rfloor_i \rangle \longmapsto^* \langle \lfloor S_4 \rfloor_i,\, \lfloor M \rfloor_i \rangle$. Therefore, the lemma holds for this case.

  - S_3 is $(S_5 \,|\, \texttt{skip})$ or $(\texttt{skip} \,|\, S_5)$ where S_5 is not $\texttt{skip}$. Without loss of generality, suppose S_3 is $(S_5 \,|\, \texttt{skip})$. Then S_1 is $S_5; \lfloor S_4 \rfloor_1$, and S_2 is $\texttt{skip}; \lfloor S_4 \rfloor_1$. Since $\langle S_5; \lfloor S_4 \rfloor_1,\, \lfloor M \rfloor_1 \rangle \longmapsto \langle S_1',\, M_1' \rangle$, we have $\langle S_5,\, \lfloor M \rfloor_1 \rangle \longmapsto \langle S_5',\, M_1' \rangle$ and S_1' is $S_5'; \lfloor S_4 \rfloor_1$. By (S9) and Lemma 3.5.2, we have $\langle S,\, M \rangle \longmapsto \langle (S_5' \mid \texttt{skip}); S_4,\, M' \rangle$, and $\lfloor M' \rfloor_1 = M_1'$, and $\lfloor M' \rfloor_2 = \lfloor M \rfloor_2 = M_2$. It is clear that condition (ii) holds.

  - For $i \in \{1, 2\}$, $\lfloor S_3 \rfloor_i$ is not $\texttt{skip}$. For $i \in \{1, 2\}$, because $\langle S_i,\, M_i \rangle \longmapsto \langle S_i',\, M_i' \rangle$ and $S_i = \lfloor S_3 \rfloor_i; \lfloor S_4 \rfloor_i$, we have $\langle \lfloor S_3 \rfloor_i,\, M_i \rangle \longmapsto \langle S_{3i},\, M_i' \rangle$. By induction, $\langle S_3,\, M \rangle \longmapsto^T \langle S_3',\, M' \rangle$, and condition (i) or (ii) holds for T and S_3'. Suppose condition (i) holds. Then for $i \in \{1, 2\}$, $\lfloor T \rfloor_i \approx [M_i, M_i']$ and

$\lfloor S'_3 \rfloor_i = S_{3i}$. By evaluation rule (S2), $\langle S,\, M\rangle \longmapsto^{T} \langle S'_3; S_4,\, M'\rangle$. Moreover, both $\lfloor S'_3; S_4 \rfloor_i$ and S'_i are $S_{3i}; \lfloor S_4 \rfloor_i$ for $i \in \{1,2\}$. Therefore, the lemma holds. For the case that condition (ii) holds for T and S_3, the same argument applies.

- S is $(S_1 \,|\, S_2)$. Since $\langle S_i,\, M\rangle \longmapsto \langle S'_i,\, M'\rangle$ for $i \in \{1,2\}$, we have $\langle S_1,\, M\rangle_1 \longmapsto \langle S'_1,\, M''\rangle_1$ and $\langle S_2,\, M''\rangle_2 \longmapsto \langle S'_2,\, M'\rangle_2$. Therefore, $\langle S,\, M\rangle \longmapsto^{T} \langle (S'_1 | S'_2),\, M'\rangle$ where $T = [M, M'', M']$. By Lemma 3.5.2, $\lfloor T \rfloor_i \approx [M_i, M'_i]$ for $i \in \{1,2\}$.

□

Lemma 3.5.5 (Adequacy). Suppose $\langle S_i,\, M_i\rangle \longmapsto^{T_i} \langle S'_i,\, M'_i\rangle$ for $i \in \{1,2\}$ are two evaluations in Aimp. Then for an Aimp* configuration $\langle S,\, M\rangle$ such that $\lfloor S \rfloor_i = S_i$ and $\lfloor M \rfloor_i = M_i$ for $i \in \{1,2\}$, we have $\langle S,\, M\rangle \longmapsto^{T} \langle S',\, M'\rangle$ such that $\lfloor T \rfloor_j \approx T_j$ and $\lfloor T \rfloor_k \approx T'_k$, where T'_k is a prefix of T_k and $\{k,j\} = \{1,2\}$.

Proof. By induction on the sum of the lengths of T_1 and T_2: $|T_1| + |T_2|$.

- $|T_1| + |T_2| \leq 3$. Without loss of generality, suppose $|T_1| = 1$. Then $T_1 = [M_1]$. Let $T = [M]$. We have $\langle S,\, M\rangle \longmapsto^{T} \langle S,\, M\rangle$. It is clear that $\lfloor T \rfloor_1 = T_1$, and $\lfloor T \rfloor_2 = [M_2]$ is a prefix of T_2.
- $|T_1| + |T_2| > 3$. If $|T_1| = 1$ or $|T_2| = 1$, then the same argument in the above case applies. Otherwise, we have $\langle S_i,\, M_i\rangle \longmapsto \langle S''_i,\, M''_i\rangle \longmapsto^{T'_i} \langle S'_i,\, M'_i\rangle$ and $T_i = [M_i] \oplus T'_i$ for $i \in \{1,2\}$. By Lemma 3.5.4, $\langle S,\, M\rangle \longmapsto^{T'} \langle S'',\, M''\rangle$ such that

  i. For $i \in \{1,2\}$, $\lfloor T' \rfloor_i \approx [M_i, M''_i]$ and $\lfloor S'' \rfloor_i = S''_i$. Since $|T'_1| + |T'_2| < |T_1| + |T_2|$, by induction we have $\langle S'',\, M''\rangle \longmapsto^{T''} \langle S',\, M'\rangle$ such that for $\{k,j\} = \{1,2\}$, $\lfloor T'' \rfloor_j \approx T'_j$ and $\lfloor T'' \rfloor_k \approx T''_k$, and T''_k is a prefix of T'_k. Let $T = T' \oplus T''$. Then $\langle S,\, M\rangle \longmapsto^{T} \langle S',\, M'\rangle$, and $\lfloor T \rfloor_j \approx T_j$, and $\lfloor T \rfloor_k \approx T'_k$ where $T'_k = [M_k, M''_k] \oplus T''_k$ is a prefix of T_k.

ii. For $\{j,k\} = \{1,2\}$, $\lfloor T' \rfloor_j \approx [M_j]$ and $\lfloor S \rfloor_j = S_j$, and $\lfloor T' \rfloor_k \approx [M_k, M''_k]$ and $\lfloor S \rfloor_k = s''_k$. Without loss of generality, suppose $j = 1$ and $k = 2$. Since $\langle S_1,\, M_1 \rangle \longmapsto^{T_1} \langle S'_1,\, M'_1 \rangle$ and $\langle S''_2,\, M'' \rangle \longmapsto^{T'_2} \langle S'_2,\, M'_2 \rangle$, and $\lfloor S' \rfloor_1 = S_1$ and $\lfloor S' \rfloor_2 = S''_2$, and $|T'_2| < |T_2|$, we can apply the induction hypothesis to $\langle S'',\, M'' \rangle$. By the similar argument in the above case, this lemma holds for this case.

$\square$

Typing rules

The type system of Aimp* includes all the typing rules in Figure 3.4 and has two additional rules for typing bracket constructs. Both confidentiality and integrity noninterference properties are instantiations of an abstract noninterference property: inputs with security labels that does not satisfy a condition ζ cannot affect outputs with security labels that satisfies ζ. Intuitively, ζ represents "low-confidentiality" or "high-integrity". Two Aimp configurations are called ζ-consistent if the terms and memory locations with security labels that satisfy ζ are indistinguishable. Another way to put the abstract noninterference property is that the ζ-consistency relation between two configurations is preserved during evaluation.

The bracket constructs captures the differences between two Aimp configurations. As a result, any effect and result of a bracket construct should have a security label that does not satisfy ζ. Let $\zeta(\ell)$ and $\zeta(\texttt{int}_\ell)$ denote that ℓ satisfies ζ. If v_1 and v_2 are not `none`, rule (V-PAIR) ensures that the value $(v_1 \mid v_2)$ has a label that does not satisfy ζ; otherwise, there is no constraint on the label of $(v_1 \mid v_2)$, because the unavailable value `none` is indistinguishable from other values. In rule (S-PAIR), the premise $\neg\zeta(pc')$ ensures that the statement $(S_1 \mid S_2)$ may have only effects with security labels that do not satisfy ζ.

$$\text{(V-PAIR)}\quad \frac{\begin{array}{c}\Gamma \vdash v_1 : \tau \qquad \Gamma \vdash v_2 : \tau \\ \neg\zeta(\tau) \text{ or } v_1 = \texttt{none} \text{ or } v_2 = \texttt{none}\end{array}}{\Gamma \vdash (v_1 \mid v_2) : \tau}$$

$$\text{(S-PAIR)}\quad \frac{\begin{array}{c}\Gamma\,; \lfloor \mathcal{R} \rfloor_1\,; pc' \vdash S_1 : \tau \\ \Gamma\,; \lfloor \mathcal{R} \rfloor_2\,; pc' \vdash S_2 : \tau \qquad \neg\zeta(pc')\end{array}}{\Gamma\,; \mathcal{R}\,; pc \vdash (S_1 \mid S_2) : \tau}$$

The key observation is that the inputs with labels not satisfying ζ do not interfere with the outputs with labels satisfying ζ, as long as all the bracket constructs are well-typed.

An important constraint that condition ζ needs to satisfy is that $\neg\zeta(\ell)$ implies $\neg\zeta(\ell \sqcup \ell')$ for any ℓ'. In Aimp*, if expression e is evaluated to a bracket value $(n_1|n_2)$, statement $\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2$ would be reduced to a bracket statement $(S_1' \mid S_2')$ where S_i' is either S_1 or S_2. To show $(S_1' \mid S_2')$ is well-typed, we need to show that S_1 and S_2 are well-typed under a program-counter label that satisfying $\neg\zeta$, and we can show it by using the constraint on $\neg\zeta$. Suppose e has type $\texttt{int}_\ell$, then we know that S_1 and S_2 are well-typed under the program counter label $pc \sqcup \ell$. Furthermore, ℓ satisfies $\neg\zeta$ because the result of e is a bracket value. Thus, by the constraint that $\neg\zeta(\ell)$ implies $\neg\zeta(\ell \sqcup \ell')$, we have $\neg\zeta(pc \sqcup \ell)$.

Suppose $\Gamma\,; \mathcal{R}\,; pc \vdash (S_1 \mid S_2) : \tau$, and $m \in \mathcal{R}$. By the evaluation rule (S9), it is possible that $\langle (S_1 \mid S_2),\, M\rangle \longmapsto^* \langle (S_1' \mid S_2),\, M'\rangle$ and $M'(m) = (n \mid \texttt{none})$, which means that m still needs to be assigned in S_2, but not in S_1'. Assume there exists $\mathcal{R}'$ such that $\Gamma\,; \mathcal{R}'\,; pc \vdash (S_1' \mid S_2) : \tau$. Then by rule (S-PAIR), we have $\Gamma\,; \lfloor \mathcal{R}' \rfloor_1\,; pc \vdash S_1' : \tau$ and $\Gamma\,; \lfloor \mathcal{R}' \rfloor_2\,; pc \vdash S_2 : \tau$. Intuitively, we want to have $m \notin \lfloor \mathcal{R}' \rfloor_1$ and $m \in \lfloor \mathcal{R}' \rfloor_2$, which are consistent with M'. To indicate such a situation, a reference m in $\mathcal{R}$ may have an index: m^1 or m^2 means that m needs to be assigned only in the first or second component of a bracket statement, and $m^\bullet$ is the same as m. The projection of $\mathcal{R}$ is computed in the following way:

$$\lfloor \mathcal{R} \rfloor_i = \{m \mid m^i \in \mathcal{R} \vee m \in \mathcal{R}\}$$

Note that indexed references are not allowed to appear in a statement type $\texttt{stmt}_{\mathcal{R}}$. To make this explicit, we require $\texttt{stmt}_{\mathcal{R}}$ is well-formed only if $\mathcal{R}$ does not contain any indexed reference m^i. For convenience, we introduce two notations dealing with indexed reference sets. Let the notation $\mathcal{R} \leq \mathcal{R}'$ denote $\lfloor \mathcal{R} \rfloor_1 \subseteq \lfloor \mathcal{R}' \rfloor_1$ and $\lfloor \mathcal{R} \rfloor_2 \subseteq \lfloor \mathcal{R}' \rfloor_2$, and let $\mathcal{R} - m^i$ denote the reference set obtained by eliminating m^i from $\mathcal{R}$, and it is computed as follows:

$$\mathcal{R} - m^i = \begin{cases} \mathcal{R}' & \text{if } \mathcal{R} = \mathcal{R}' \cup \{m^j\} \wedge i \in \{j, \bullet\} \\ \mathcal{R}' \cup \{m^j\} & \text{if } \mathcal{R} = \mathcal{R}' \cup \{m\} \wedge \{i, j\} = \{1, 2\} \\ \mathcal{R} & \text{if otherwise} \end{cases}$$

Subject reduction

Lemma 3.5.6 (Update). If $\Gamma\,; \mathcal{R} \vdash v : \tau$ and $\Gamma\,; \mathcal{R} \vdash v' : \tau$, then $\Gamma\,; \mathcal{R} \vdash v[v'/\pi_i] : \tau$.

Proof. If i is $\bullet$, then $v[v'/\pi_i] = v'$, and we have $\Gamma \vdash v' : \tau$. If i is 1, then $v[v'/\pi_i] = (v' \mid \lfloor v \rfloor_2)$. Since $\Gamma \vdash v : \tau$, we have $\Gamma \vdash \lfloor v \rfloor_2 : \tau$. By rule (V-PAIR), $\Gamma \vdash (v' \mid \lfloor v \rfloor_2) : \tau$. Similarly, if i is 2, we also have $\Gamma \vdash v[v'/\pi_i] : \tau$. □

Lemma 3.5.7 (Relax). If $\Gamma\,; \mathcal{R}\,; pc \sqcup \ell \vdash S : \tau$, then $\Gamma\,; \mathcal{R}\,; pc \vdash S : \tau$.

Proof. By induction on the derivation of $\Gamma\,; \mathcal{R}\,; pc \sqcup \ell \vdash S : \tau$. □

Lemma 3.5.8. Suppose $\Gamma\,; \mathcal{R} \vdash e : \tau$, and $\Gamma \vdash M$, and $\langle e,\, M \rangle \Downarrow v$. Then $\Gamma\,; \mathcal{R} \vdash v : \tau$.

Proof. By induction on the structure of e. □

Lemma 3.5.9. Suppose $\Gamma\,; \mathcal{R}\,; pc \vdash S : \texttt{stmt}_{\mathcal{R}'}$. If $m^i \in \mathcal{R}$ where $i \in \{1, 2\}$, then $m \notin \mathcal{R}'$.

Proof. By induction on the derivation of $\Gamma\,; \mathcal{R}\,; pc \vdash S : \texttt{stmt}_{\mathcal{R}'}$. □

Definition 3.5.9 (Well-typed memory). Memory M is well-typed in Γ, written $\Gamma \vdash M$, if $\mathit{dom}(\Gamma) = \mathit{dom}(M)$, and for any $m \in \mathit{dom}(\Gamma)$, $\Gamma\,;\mathcal{R} \vdash M(m) : \Gamma(m)$.

Definition 3.5.10 ($\Gamma\,;\mathcal{R} \vdash M$). A memory M is consistent with Γ, $\mathcal{R}$, written $\Gamma\,;\mathcal{R} \vdash M$, if $\Gamma \vdash M$, and for any m in $\mathit{dom}(M)$ such that $A_\Gamma(m) \not\leq L$, $M(m) = \texttt{none}$ implies $m \in \mathcal{R}$, and $M(m) = (\texttt{none} \mid n)$ implies $m^1 \in \mathcal{R}$, and $M(m) = (n \mid \texttt{none})$ implies $m^2 \in \mathcal{R}$.

Theorem 3.5.1 (Subject reduction). Suppose $\Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash S : \tau$, and $\Gamma \vdash M$, and $\langle S,\, M\rangle_i \longmapsto \langle S',\, M'\rangle_i$, and $i \in \{1, 2\}$ implies $V(\mathit{pc})$. Then there exists $\mathcal{R}'$ such that the following conditions hold:

i. $\Gamma\,;\mathcal{R}'\,;\mathit{pc} \vdash S' : \tau$, and $\mathcal{R}' \leq \mathcal{R}$, and $\Gamma \vdash M'$.

ii. For any $m^j \in \mathcal{R} - \mathcal{R}'$, $\lfloor M' \rfloor_i(m^j) \neq \texttt{none}$.

iii. Suppose $V(\ell)$ is $I(\ell) \leq L$. Then $\Gamma\,;\mathcal{R} \vdash \lfloor M \rfloor_i$ implies $\Gamma\,;\mathcal{R}' \vdash \lfloor M' \rfloor_i$.

iv. If $\lfloor M \rfloor_i(m) = \texttt{none}$, and $\lfloor M' \rfloor_i(m) = n$, and $A(\Gamma(m)) \not\leq I(\mathit{pc})$, then $m \notin \mathcal{R}'$.

Proof. By induction on the evaluation step $\langle S,\, M\rangle_i \longmapsto \langle S',\, M'\rangle_i$. Without loss of generality, we assume that the derivation of $\Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash S : \tau$ does not end with using the (SUB) rule. In fact, if $\Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash S : \texttt{stmt}_{\mathcal{R}_2}$ is derived by $\Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash S : \texttt{stmt}_{\mathcal{R}_1}$ and $\Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash \texttt{stmt}_{\mathcal{R}_1} \leq \texttt{stmt}_{\mathcal{R}_2}$, and there exists $\mathcal{R}''$ such that conditions (i)–(iv) hold for $\Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash S : \texttt{stmt}_{\mathcal{R}_1}$, then by Lemma 3.5.9, we can show that $\mathcal{R}' = \mathcal{R}'' \cup (\mathcal{R}_2 - \mathcal{R}_1)$ satisfies conditions (i)–(iv) for $\Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash S : \texttt{stmt}_{\mathcal{R}_2}$.

- **Case (S1)**. In this case, S is $m := e$, S' is $\texttt{skip}$, and τ is $\texttt{stmt}_{\mathcal{R}-\{m\}}$. By (S1), M' is $M[m \mapsto M(m)[v/\pi_i]]$. By Lemma 3.5.8, we have $\Gamma \vdash v : \Gamma(m)$, which implies that $M(m)[v/\pi_i]$ has type $\Gamma(m)$. Therefore, $\Gamma \vdash M'$. The well-formedness of τ implies that $\mathcal{R}$ does not contain any indexed references. Let $\mathcal{R}'$ be $\mathcal{R} - \{m\}$. It is clear that $\mathcal{R}' \leq \mathcal{R}$. By rule (SKIP), $\Gamma\,;\mathcal{R}'\,;\mathit{pc} \vdash \texttt{skip} : \texttt{stmt}_{\mathcal{R}'}$.

Because $\lfloor M' \rfloor_i(m) = v \neq \texttt{none}$, and $\mathcal{R} - \mathcal{R}' = \{m\}$, condition (ii) holds. Since $\lfloor M' \rfloor_i(m) = n$ and $\mathcal{R} - \mathcal{R}' = \{m\}$, we have that $\Gamma\,;\mathcal{R} \vdash \lfloor M \rfloor_i$ implies $\Gamma\,;\mathcal{R}'\,;L \vdash \lfloor M' \rfloor_i$.

- **Case (S2)**. Obvious by induction.
- **Case (S3)**. Trivial.
- **Case (S4)**. In this case, S is $\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2$. By the typing rule (IF), we have $\Gamma\,;\mathcal{R}\,;pc \sqcup \ell_e \vdash S_1 : \tau$. By Lemma 3.5.7, $\Gamma\,;\mathcal{R}\,;pc \vdash S_1 : \tau$. In this case, $M' = M$ and $\mathcal{R}' = \mathcal{R}$, so conditions (ii) and (iii) immediately hold.
- **Case (S5)**. By the similar argument of case (S4).
- **Case (S6)**. In this case, S is $\texttt{while}\ e\ \texttt{do}\ S_1$, and τ is $\texttt{stmt}_{\mathcal{R}}$. By rule (WHILE), $\Gamma\,;\mathcal{R}\,;pc \sqcup \ell \vdash S_1 : \texttt{stmt}_{\mathcal{R}}$, where ℓ is the label of e. By the typing rule (SEQ), $\Gamma\,;\mathcal{R}\,;pc \sqcup \ell \vdash S_1; \texttt{while} e \texttt{do} S_1 : \texttt{stmt}_{\mathcal{R}}$. Since $M' = M$ and $\mathcal{R}' = \mathcal{R}$, conditions (ii)–(iv) hold.
- **Case (S7)**. In this case, S' is $\texttt{skip}$, and τ is $\texttt{stmt}_{\mathcal{R}}$. We have $\Gamma\,;\mathcal{R}\,;pc \vdash \texttt{skip} : \texttt{stmt}_{\mathcal{R}}$. Furthermore, $M' = M$ and $\mathcal{R}' = \mathcal{R}$. Thus, conditions (ii)–(iv) hold.
- **Case (S8)**. In this case, S is $\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2$, and i must be $\bullet$. Suppose $\Gamma \vdash e : \texttt{int}_{\ell}$. By Lemma 3.5.8, $\Gamma \vdash (n_1 \mid n_2) : \texttt{int}_{\ell}$. By rule (V-PAIR), $V(\ell)$ holds, which implies $V(pc \sqcup \ell)$. By rule (IF), $\Gamma\,;\mathcal{R}\,;pc \sqcup \ell \vdash S_i : \tau$, which implies $\Gamma\,;\mathcal{R}\,;pc \sqcup \ell \vdash \texttt{if}\ n_i\ \texttt{then}\ \lfloor S_1 \rfloor_i\ \texttt{else}\ \lfloor S_2 \rfloor_i : \tau$. By rule (S-PAIR), $\Gamma\,;\mathcal{R}\,;pc \vdash S' : \tau$. Again, since $M' = M$ and $\mathcal{R}' = \mathcal{R}$, conditions (ii) and (iii) hold.
- **Case (S9)**. In this case, S is $(S_1 \mid S_2)$. Without loss of generality, suppose $\langle S_1,\, M\rangle_1 \longmapsto \langle S_1',\, M'\rangle_1$, and $\langle S,\, M\rangle \longmapsto \langle (S_1' \mid S_2),\, M'\rangle$. By rule (S-PAIR), $\Gamma\,;\lfloor \mathcal{R} \rfloor_1\,;pc \vdash S_1 : \tau$. By induction, there exists $\mathcal{R}_1'$ such that $\Gamma\,;\mathcal{R}_1'\,;pc \vdash S_1' : \tau$, and $\mathcal{R}_1' \subseteq \lfloor \mathcal{R} \rfloor_1$, and $\Gamma \vdash M'$. Let $\mathcal{R}'$ be $\mathcal{R}_1' \bullet \lfloor \mathcal{R} \rfloor_2$, which is computed by the

formula:

$$\begin{aligned}\mathcal{R}_1 \bullet \mathcal{R}_2 &= \{m \mid m \in \mathcal{R}_1 \cap \mathcal{R}_2\} \cup \\ &\quad \{m^1 \mid m \in \mathcal{R}_1 - \mathcal{R}_2\} \cup \\ &\quad \{m^2 \mid m \in \mathcal{R}_2 - \mathcal{R}_1\}\end{aligned}$$

Since $\lfloor \mathcal{R}' \rfloor_1 = \mathcal{R}_1'$ and $\lfloor \mathcal{R}' \rfloor_2 = \lfloor \mathcal{R} \rfloor_2$, we have $\Gamma\,; \lfloor \mathcal{R}' \rfloor_1\,; \mathit{pc} \vdash S_1' : \tau$. By rule (S-PAIR), $\Gamma\,; \mathcal{R}'\,; \mathit{pc} \vdash S' : \tau$ holds. Since $\lfloor \mathcal{R}' \rfloor_2 = \lfloor \mathcal{R} \rfloor_2$, for any $m^j \in \mathcal{R} - \mathcal{R}'$, it must be the case that $j = 1$, and $m \in \lfloor \mathcal{R} \rfloor_1 - \mathcal{R}_1'$. By induction, $\lfloor M' \rfloor_1(m) \neq$ `none`. Therefore, condition (ii) holds.

If $\Gamma\,; \mathcal{R} \vdash M$, then $\Gamma\,; \lfloor \mathcal{R} \rfloor_1 \vdash \lfloor M \rfloor_1$. By induction, $\Gamma\,; \mathcal{R}_1' \vdash \lfloor M' \rfloor_1$. Therefore, $\Gamma\,; \mathcal{R}' \vdash M'$ holds.

- **Case (S10)**. In this case, S is $(\texttt{skip} \mid \texttt{skip})$. We have $\Gamma\,; \lfloor \mathcal{R} \rfloor_i\,; \mathit{pc} \vdash \texttt{skip} : \texttt{stmt}_{\lfloor \mathcal{R} \rfloor_i}$ for $i \in \{1, 2\}$. By rule (S-PAIR), $\Gamma\,; \lfloor \mathcal{R} \rfloor_i\,; \mathit{pc}' \vdash \texttt{skip} : \tau$. Therefore, $\Gamma\,; \lfloor \mathcal{R} \rfloor_i\,; \mathit{pc}' \vdash \texttt{stmt}_{\lfloor \mathcal{R} \rfloor_i} \leq \tau$. By the subtyping rule, $\tau = \texttt{stmt}_{\lfloor \mathcal{R} \rfloor_i}$. So $\lfloor \mathcal{R} \rfloor_1 = \lfloor \mathcal{R} \rfloor_2 = \mathcal{R}$ and $\tau = \texttt{stmt}_{\mathcal{R}}$. By rule (SKIP), $\Gamma\,; \mathcal{R}\,; \mathit{pc} \vdash \texttt{skip} : \tau$.

□

Progress

Lemma 3.5.10 (Expression availability). Suppose $\Gamma\,; \mathcal{R} \vdash e : \tau$ and $\Gamma\,; \mathcal{R} \vdash M$. Then $\langle e,\, M \rangle_i \Downarrow v$ such that v is available.

Proof. By induction on the structure of e.

- e is n. Obvious.
- e is $!m$. Since $\Gamma\,; \mathcal{R} \vdash M$, $\lfloor M \rfloor_i(m)$ is either n or $(n_1 \mid n_2)$.
- e is $e_1 + e_2$. Then $\langle e,\, M \rangle_i \Downarrow v_i$. By induction, both v_1 and v_2 are available. Thus, $v_1 + v_2$ is an available value.

□

Theorem 3.5.2 (Progress). Let $\zeta(\ell)$ be $I(\ell) \not\leq l_{\mathtt{A}}$, and let $|S|$ represent the size of the statement S, i.e. the number of syntactical tokens in S. Suppose $\Gamma\,;\mathcal{R}\,;pc \vdash S : \mathtt{stmt}_{\mathcal{R}'}$, and $\Gamma\,;\mathcal{R} \vdash \lfloor M \rfloor_i$, and S is not skip, and $A_{\Gamma}(\mathcal{R}) \not\leq l_{\mathtt{A}}$, and $i \in \{1,2\}$ implies $I(pc) \leq l_{\mathtt{A}}$. Then $\langle S,\, M\rangle_i \longmapsto \langle S',\, M'\rangle_i$. Furthermore, if S is $(S_1 \mid S_2); S_3$ or $(S_1 \mid S_2)$, then $|S'| < |S|$.

Proof. By induction on the structure of S.

- S is $m := e$. Suppose $A(\mathcal{R}) \not\leq l_{\mathtt{A}}$. By Lemma 3.5.10, $\langle e,\, M\rangle_i \Downarrow v$ and v is available. By rule (S1), $\langle m := e,\, M\rangle_i \longmapsto \langle \mathtt{skip},\, M[m \mapsto M(x)[v/\pi_i]]\rangle_i$.
- S is $S_1; S_2$. Suppose S_1 is not skip. By induction, $\langle S_1,\, M\rangle_i \longmapsto \langle S_1',\, M\rangle_i$. By (S2), $\langle S_1; S_2,\, M\rangle_i \longmapsto \langle S_1'; S_2,\, M\rangle_i$. Moreover, by induction, $|S_1'; S_2| < |S_1; S_2|$. If S_1 is skip, then $\langle S_1; S_2,\, M\rangle_i \longmapsto \langle S_2,\, M\rangle_i$.
- S is $\mathtt{if}\ e\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2$. By Lemma 3.5.10, $\langle e,\, M\rangle_i \Downarrow v$, and v is available. If $v = n$, then $\langle \mathtt{if}\ e\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2,\, M\rangle_i \longmapsto \langle S_j,\, M\rangle_i$ where $j \in \{1,2\}$. If $v = (n_1 \mid n_2)$, then $\lfloor\mathcal{R}\rfloor_1 = \lfloor\mathcal{R}\rfloor_2$ because S is not a pair statement.
- S is $\mathtt{while}\ e\ \mathtt{do}\ S_1$. By Lemma 3.5.10, $\langle e,\, M\rangle_i \Downarrow v$, and v is available. If $v = n$, then $\langle \mathtt{while}\ e\ \mathtt{do}\ S_1,\, M\rangle_i \longmapsto \langle \mathtt{skip},\, M\rangle_i$ or $\langle \mathtt{while}\ e\ \mathtt{do}\ S_1,\, M\rangle_i \longmapsto \langle S_1; \mathtt{while}\ e\ \mathtt{do}\ S_1,\, M\rangle_i$. If $i \in \{1,2\}$, then $I(pc) \leq l_{\mathtt{A}}$. By the typing rule (WHILE), $A(\mathcal{R}) \leq l_{\mathtt{A}}$, contradicting $A(\mathcal{R}) \not\leq l_{\mathtt{A}}$. Therefore, i is $\bullet$, which implies $\lfloor\mathcal{R}\rfloor_1 = \lfloor\mathcal{R}\rfloor_2$.
- S is $(S_1 \mid S_2)$. If S_1 and S_2 are both skip, then $\langle S,\, M\rangle \longmapsto \langle \mathtt{skip},\, M\rangle$. Otherwise, without loss of generality, suppose S_1 is not skip. By (S-PAIR), $\Gamma\,;\lfloor\mathcal{R}\rfloor_1\,;pc' \vdash S_1 : \tau$, where $I(pc') \leq l_{\mathtt{A}}$ holds. By induction, $\langle S_1,\, M\rangle_1 \longmapsto \langle S_1',\, M'\rangle_2$. By $I(pc') \leq l_{\mathtt{A}}$ and $A(\mathcal{R}) \not\leq l_{\mathtt{A}}$, S_1 is not a while statement. Thus, $|S_1'| < |S_1|$. By (S9), $\langle S,\, M\rangle \longmapsto \langle (S_1' \mid S_2),\, M'\rangle$. In addition, $|(S_1' \mid S_2)| < |S|$.

$\square$

3.5.3 Noninterference proof

Theorem 3.5.3 (Confidentiality noninterference). If $\Gamma\,;\mathcal{R}\,;pc \vdash S : \tau$, then $\Gamma \vdash \mathtt{NI}_C(S)$.

Proof. Given two memories M_1 and M_2 in Aimp, let $M = M_1 \uplus M_2$ be an Aimp* memory computed by merging M_1 and M_2 as follows:

$$M_1 \uplus M_2(m) = \begin{cases} M_1(m) & \text{if } M_1(m) = M_2(m) \\ (M_1(m) \mid M_2(m)) & \text{if } M_1(m) \neq M_2(m) \end{cases}$$

Let $\zeta(\ell)$ be $C(\ell) \leq l_{\mathtt{A}}$. Then $\Gamma \vdash M_1 \approx_{C \leq l_{\mathtt{A}}} M_2$ implies that $\Gamma \vdash M$. Suppose $\langle S_i,\, M_i\rangle \longmapsto^{T_i} \langle S_i',\, M'\rangle$ for $i \in \{1, 2\}$. Then by Lemma 3.5.5, there exists $\langle S',\, M'\rangle$ such that $\langle S,\, M\rangle \longmapsto^{T} \langle S',\, M'\rangle$, and $\lfloor T\rfloor_j \approx T_j$ and $\lfloor T\rfloor_k \approx T_k'$ where $\{j, k\} = \{1, 2\}$ and T_k' is a prefix of T_k. By Theorem 3.5.1, for each M' in T, $\Gamma \vdash M'$, which implies that $\lfloor M'\rfloor_1 \approx_{C \leq l_{\mathtt{A}}} \lfloor M'\rfloor_2$. Therefore, we have $\Gamma \vdash T_j \approx_{C \leq l_{\mathtt{A}}} T_k$. Thus, $\Gamma \vdash \mathtt{NI}_C(S)$. □

Theorem 3.5.4 (Integrity noninterference). If $\Gamma\,;\mathcal{R}\,;pc \vdash S : \tau$, then $\Gamma \vdash \mathtt{NI}_I(S)$.

Proof. Let $\zeta(\ell)$ be $I(\ell) \not\leq l_{\mathtt{A}}$. By the same argument as in the proof of the confidentiality noninterference theorem. □

Lemma 3.5.11 (Balance). Let $\zeta(\ell)$ be $I(\ell) \not\leq l_{\mathtt{A}}$. Suppose $\Gamma\,;\mathcal{R}\,;pc \vdash S : \tau$, and $\Gamma\,;\mathcal{R} \vdash M$. Then $\langle S,\, M\rangle \longmapsto^{*} \langle S',\, M'\rangle$ such that $\Gamma \vdash \lfloor M'\rfloor_1 \approx_{A \not\leq l_{\mathtt{A}}} \lfloor M'\rfloor_2$.

Proof. By induction on the size of S.

- $|S| = 1$. In this case, S must be `skip`. However, $\Gamma\,;\mathcal{R}\,;pc \vdash \mathtt{skip} : \mathtt{stmt}_{\mathcal{R}}$ implies $\lfloor\mathcal{R}\rfloor_1 = \lfloor\mathcal{R}\rfloor_2$, which is followed by $\Gamma \vdash \lfloor M\rfloor_1 \approx_{A \not\leq l_{\mathtt{A}}} \lfloor M\rfloor_2$ because $\Gamma\,;\mathcal{R} \vdash M$.
- $|S| > 1$. By the definition of $\Gamma\,;\mathcal{R} \vdash M$, $\Gamma \vdash \lfloor M\rfloor_1 \not\approx_{A \not\leq L} \lfloor M\rfloor_2$ implies $\lfloor\mathcal{R}\rfloor_1 \neq \lfloor\mathcal{R}\rfloor_2$. By Theorem 3.5.2, $\langle S,\, M\rangle \longmapsto \langle S',\, M'\rangle$ and $|S'| < |S|$. By Theorem 3.5.1, there exists $\mathcal{R}'$ such that $\Gamma\,;\mathcal{R}'\,;pc \vdash S' : \tau$ and $\Gamma\,;\mathcal{R}' \vdash M'$. By induction, $\langle S',\, M'\rangle \longmapsto^{*} \langle S'',\, M''\rangle$ and $\Gamma \vdash \lfloor M''\rfloor_1 \approx_{A \not\leq l_{\mathtt{A}}} \lfloor M''\rfloor_2$.

$\square$

Theorem 3.5.5 (Availability noninterference). If $\Gamma\,;\mathcal{R}\,;pc \vdash S : \tau$, then $\Gamma\,;\mathcal{R} \vdash \mathtt{NI}_A(S)$.

Proof. Let $\zeta(\ell)$ be $I(\ell) \not\leq l_\mathtt{A}$. Given two memories M_1 and M_2 in Aimp such that $\Gamma \vdash M_1 \approx_{I\not\leq l_\mathtt{A}} M_2$ and for any m in *dom*(Γ), $m \notin \mathcal{R}$ and $A(\Gamma(m)) \not\leq l_\mathtt{A}$ imply $M_i(m) \neq \mathtt{none}$. To prove $\Gamma \vdash \mathtt{NI}_A(S)$, we only need to show that there exists $\langle S_i',\, M_i'\rangle$ such that $\langle S,\, M_i\rangle \longmapsto^* \langle S_i',\, M_i'\rangle$, and for any $\langle S_i'',\, M_i''\rangle$ such that $\langle S_i',\, M_i'\rangle \longmapsto^* \langle S_i'',\, M_i''\rangle$, $\Gamma \vdash M_1'' \approx_{A\not\leq l_\mathtt{A}} M_2''$ holds.

Let $M = M_1 \uplus M_2$. Intuitively, by Lemma 3.5.11, evaluating $\langle S,\, M\rangle$ will eventually result in a memory M' such that $\Gamma \vdash \lfloor M'\rfloor_1 \approx_{A\not\leq l_\mathtt{A}} \lfloor M'\rfloor_2$, and if any high-availability reference m is unavailable in M', m will remain unavailable. This conclusion can be projected to $\langle S,\, M_i\rangle$ for $i \in \{1,2\}$ by Lemma 3.5.2.

Suppose $\langle S,\, M\rangle \longmapsto^* \langle S',\, M'\rangle$ such that for any m with $A_\Gamma(m) \not\leq l_\mathtt{A}$, $\lfloor M'\rfloor_i(m) \neq \mathtt{none}$ for $i \in \{1,2\}$. By Lemma 3.5.2, $\langle S,\, M_i\rangle \longmapsto^* \langle \lfloor S'\rfloor_i,\, \lfloor M'\rfloor_i\rangle$. Moreover, for any $\langle S_i',\, M_i'\rangle$ such that $\langle \lfloor S'\rfloor_i,\, \lfloor M'\rfloor_i\rangle \longmapsto^* \langle S_i',\, M_i'\rangle$, and any m with $A_\Gamma(m) \not\leq L$, it must be the case that $M_i'(m) \neq \mathtt{none}$. Therefore, $\Gamma \vdash M_1' \approx_{A\not\leq l_\mathtt{A}} M_2'$.

Otherwise, $\langle S,\, M\rangle \longmapsto^* \langle S',\, M'\rangle$ such that there exists m with $A(\Gamma(m)) \not\leq l_\mathtt{A}$ and $\lfloor M'\rfloor_i(m) = \mathtt{none}$ for some $i \in \{1,2\}$, and for any $\langle S'',\, M''\rangle$ such that $\langle S',\, M'\rangle \longmapsto^* \langle S'',\, M''\rangle$, $\Gamma \vdash \lfloor M'\rfloor_i \approx_{A\not\leq l_\mathtt{A}} \lfloor M''\rfloor_i$. By Lemma 3.5.11, $\Gamma \vdash \lfloor M'\rfloor_1 \approx_{A\not\leq l_\mathtt{A}} \lfloor M'\rfloor_2$ must hold. Assume $\Gamma \vdash \lfloor M'\rfloor_1 \approx_{A\not\leq l_\mathtt{A}} \lfloor M'\rfloor_2$ does not hold. Then there exists $\langle S'',\, M''\rangle$ such that $\langle S',\, M'\rangle \longmapsto^* \langle S'',\, M''\rangle$ and $\Gamma \vdash \lfloor M''\rfloor_1 \approx_{A\not\leq l_\mathtt{A}} \lfloor M''\rfloor_2$. Because for $i \in \{1,2\}$, $\Gamma \vdash \lfloor M'\rfloor_i \approx_{A\not\leq l_\mathtt{A}} \lfloor M''\rfloor_i$, we have $\Gamma \vdash \lfloor M'\rfloor_1 \approx_{A\not\leq l_\mathtt{A}} \lfloor M'\rfloor_2$, which contradicts the original assumption.

In addition, we can show that $\langle S',\, M'\rangle$ would generate an evaluation of infinite steps, and both projections of the evaluation also have infinite steps so that they always cover the evaluations of $\langle S,\, M_1\rangle$ and $\langle S,\, M_2\rangle$. By Theorem 3.5.1, there exists $\mathcal{R}'$ such that $\Gamma\,;\mathcal{R}'\,;pc \vdash S' : \tau$, and $\Gamma\,;\mathcal{R}' \vdash M'$. It is clear that $A(\mathcal{R}') \not\leq l_{\mathtt{A}}$ holds, because there exists m such that $A(\Gamma(m)) \not\leq l_{\mathtt{A}}$ and $\lfloor M' \rfloor_i(m) = \mathtt{none}$ for some $i \in \{1, 2\}$. By Theorem 3.5.2, $\langle S',\, M' \rangle \longmapsto \langle S'',\, M'' \rangle$. Since $\Gamma \vdash \lfloor M' \rfloor_i \approx_{A \not\leq l_{\mathtt{A}}} \lfloor M'' \rfloor_i$ for $i \in \{1, 2\}$, $\langle S'',\, M'' \rangle$ can make progress by the same argument. Therefore, $\langle S',\, M' \rangle$ will generate an evaluation of infinite steps. Suppose the first projection of the evaluation is finite. Then $\langle S',\, M' \rangle \longmapsto^* \langle S_1,\, M_1 \rangle \longmapsto \langle S_2,\, M_2 \rangle \longmapsto \ldots \longmapsto \langle S_n,\, M_n \rangle \ldots$, and $\lfloor \langle S_j,\, M_j \rangle \rfloor_1 = \lfloor \langle S_1,\, M_1 \rangle \rfloor_1$ for any j. It must be the case that S_1 is $(S_1' \mid S_2'); S_3'$ or $(S_1' \mid S_2')$. This contradicts Theorem 3.5.2, which implies $|S_{j+1}| < |S_j|$ for any j.

By Lemma 3.5.2, the projections of the evaluation are Aimp evaluations. Therefore, for $i \in \{1, 2\}$, there exists $\langle S_i'',\, M_i'' \rangle$ such that $\langle S',\, M' \rangle \longmapsto^* \langle S'',\, M'' \rangle$ and $\lfloor M'' \rfloor_i = M_i''$. Since $\Gamma \vdash M_i'' \approx_{A \not\leq l_{\mathtt{A}}} \lfloor M' \rfloor_i$ for $i \in \{1, 2\}$, $\Gamma \vdash M_1'' \approx_{A \not\leq l_{\mathtt{A}}} M_2''$ holds.

$\square$

3.6 Related work

Using static program analysis to check information flow was first proposed by Denning and Denning [18], and is one of the four classes of information flow control mechanisms as discussed in Section 2.5. Later work phrased the static information flow analysis as type checking (e.g., [66]). Noninterference was later developed as a more semantic characterization of security [31], followed by many extensions. Volpano, Smith and Irvine [88] first showed that type systems can be used to enforce noninterference and proved a version of noninterference theorem for a simple imperative language, starting a line of research pursuing the noninterference result for more expressive security-typed languages. Heintze and Riecke [34] proved the noninterference theorem for the SLam calculus, a purely functional language. Zdancewic and Myers [102] investigated a secure calculus with first-class continuations and references. Pottier and Simonet [70] considered an ML-like functional language and introduced the proof technique that is extended in this paper. A more complete survey of language-based information-flow techniques can be found in [73, 107]. Compared with those previous work, the main contribution of Aimp is to apply the security-typed language approach to enforcing availability policies.

Volpano and Smith [87] introduce the notion of *termination agreement*, which requires two executions indistinguishable to low-confidentiality users to both terminate or both diverge. The integrity dual of termination agreement can be viewed as a special case of the availability noninterference in which termination is treated as the only output of a program.

Lamport first introduced the concepts of *safety* and *liveness* properties [46]. Being available is often characterized as a liveness property, which informally means "something good will eventually happen". In general, verifying whether a program will eventually produce an output is equivalent to solving the halting problem, and thus incomputable for a Turing-complete language. This work proposes a security model in which an availability policy can be enforced by a noninterference property [31]. It is well known that a noninterference property is not a property on traces [58], and unlike safety or liveness properties, cannot be specified by a trace set. However, a noninterference property can be treated as a property on pairs of traces. For example, consider a trace pair (T_1, T_2). It has the confidentiality noninterference property if the first elements of T_1 and T_2 are distinguishable, or T_1 and T_2 are indistinguishable to low-confidentiality users. Therefore, a noninterference property can be represented by a set of trace pairs $\mathcal{P}$, and a program satisfies the property if all the pairs of traces produced by the program belong to $\mathcal{P}$. Interestingly, with respect to a trace pair, the confidentiality and integrity noninterference properties have the informal meaning of safety properties ("something bad will not happen"), and availability noninterference takes on the informal meaning of liveness.

Li et al. [47] formalize the notion that highly available data does not depend on low-availability data. However, their definition is *termination-insensitive* [73], which makes it inappropriate to model availability noninterference.

Lafrance and Mullins [45] define a semantic security property *impassivity* for preventing DoS attacks. Intuitively, impassivity means that low-cost actions cannot interfere with high-cost actions. In some sense, impassivity is an integrity noninterference property, if we treat low-cost as low-integrity and high-cost as high-integrity. With the implicit assumption that high-cost actions may exhaust system resources and render a system unavailable, impassivity corresponds to one part of our notion of availability noninterference: low-integrity inputs cannot affect the availabilities of highly available outputs.

Chapter 4
Secure distributed computation

A static analysis like the type system of Aimp described in the previous chapter can check for potential security violations in a program that is executed on a trusted computing platform. This chapter considers how to perform secure computation in a distributed system with untrusted hosts. The main result is a programming language DSR, which is designed for writing secure distributed programs and has some novel features compared to other security-typed process calculi [37, 103]:

- *Dynamic label checking*, which combines static analysis and dynamic mechanisms needed to deal with untrusted hosts,
- *Quorum replication*, which is built into the syntax and type system of DSR, making it possible to reason about the security assurances provided by this replication technique,
- *Multilevel timestamp*, a novel timestamp scheme used to coordinate concurrent computations running on different replicas, without introducing covert channels.

This chapter gives an overview of the key mechanisms of DSR and how they can be used to build secure programs. A formal description of DSR is found in Chapter 5.

4.1 System model

A distributed system is a set of networked host machines. Each host can be viewed as a state machine that acts upon incoming network messages, changing its local state and/or sending out messages to other hosts. Figure 4.1 shows a distributed system composed of five hosts, which communicate with each other through messages such as $\mu_1, \ldots, \mu_7$. As the close-up of host h_3 shows, a host is composed of a memory M, a thread pool Θ,

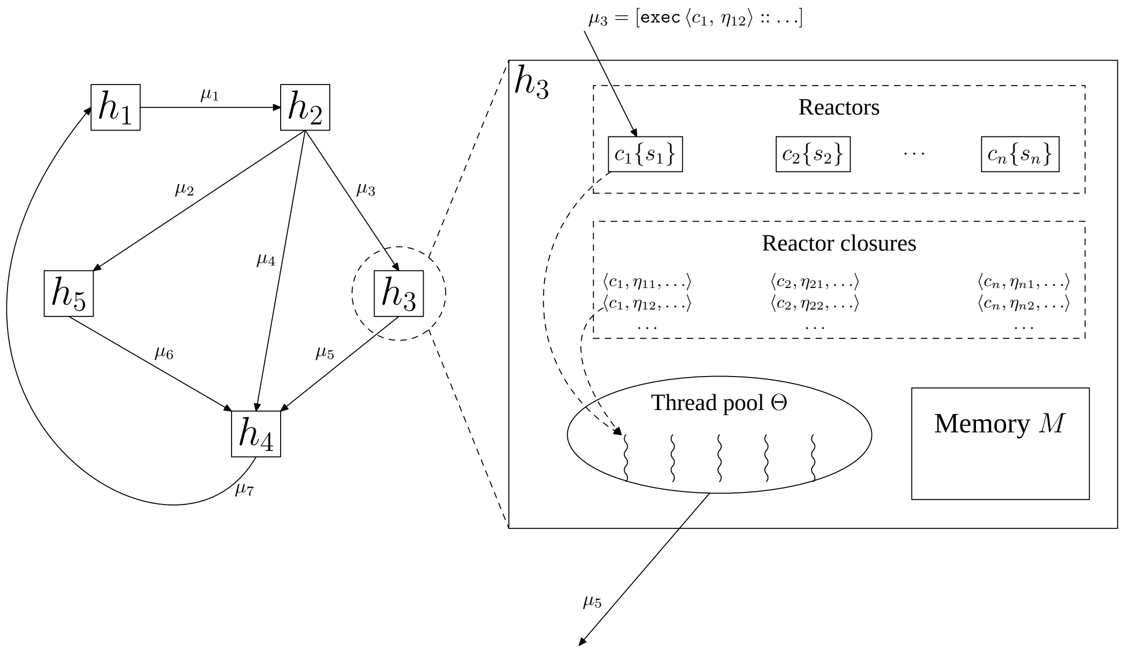

$\mu_3 = [\texttt{exec}\,\langle c_1,\, \eta_{12}\rangle :: \ldots]$
h_1
h_2
h_3
h_4
h_5
μ_1
μ_2
μ_3
μ_4
μ_5
μ_6
μ_7
h_3
Reactors
$c_1\{s_1\}$
$c_2\{s_2\}$
$\cdots$
$c_n\{s_n\}$
Reactor closures
$\langle c_1, \eta_{11}, \ldots\rangle$
$\langle c_1, \eta_{12}, \ldots\rangle$
$\langle c_2, \eta_{21}, \ldots\rangle$
$\langle c_2, \eta_{22}, \ldots\rangle$
$\langle c_n, \eta_{n1}, \ldots\rangle$
$\langle c_n, \eta_{n2}, \ldots\rangle$
Thread pool Θ
Memory M
μ_5

Figure 4.1: System model

a set of *reactors* that specifies the code to be executed in reaction to incoming messages, and a set of *reactor closures* that contains data (parameters) needed to execute code in corresponding reactors.

Each reactor has a unique name c and a program statement s. When invoked by a network message, reactor c spawns a new thread to execute its statement. Each invocation message carries an integer identifier η, corresponding to an *invocation context*, which is embedded in a reactor closure on the receiving host and contains parameters needed to handle the invocation. A context identifier η is unique for a given reactor, so the pair $\langle c,\, \eta\rangle$ uniquely identifies a closure, and is called a *closure identifier*. For simplicity, we use the term "closure $\langle c,\, \eta\rangle$" to denote the closure identified by $\langle c,\, \eta\rangle$. Every invocation message for reactor c carries a context identifier η and can be viewed as invoking the closure $\langle c,\, \eta\rangle$. For example, as shown in Figure 4.1, message $\mu_3 = [\texttt{exec}\,\langle c_1,\, \eta_{12}\rangle :: \ldots]$ requests executing (invoking) the closure $\langle c_1,\, \eta_{12}\rangle$, and a new thread is spawned to execute the code of c_1. Since an invocation context is associated with a particular invocation, a reactor closure can be invoked only once.

Intuitively, a reactor is like a function that can be invoked remotely, and a reactor closure is like a function closure. Closure [71, 77, 61] is a well-known mechanism for handling first-class functions with lexical scoping. Introducing closures explicitly makes DSR different from other process calculi [59, 60, 29] with an evaluation model based on substitution. The substitution needs in general to happen on code located on a different host than the current one, and it is an implicit distributed operation. In comparison, representing closures explicitly is more faithful to the way that computation occurs in a real distributed system, and this treatment makes the security of DSR clear.

We assume that each host h has a label $\textit{label}(h) = \ell$ that specifies the security level of the host. Let $C(h)$, $I(h)$ and $A(h)$ respectively represent the confidentiality, integrity and availability components of $\textit{label}(h)$. Intuitively, these base labels place upper bounds on the base labels of data processed by h. For example, host h is trusted to protect the confidentiality of data with a confidentiality label less than or equal to $C(h)$. Let H be a set of hosts. We often need to compute the join and meet of base labels of hosts in H. Thus, we introduce the following notations: $C_{\sqcup}(H) = \bigsqcup_{h \in H} C(h)$, $C_{\sqcap}(H) = \bigsqcap_{h \in H} C(h)$, $I_{\sqcup}(H) = \bigsqcup_{h \in H} I(h)$, $I_{\sqcap}(H) = \bigsqcap_{h \in H} I(h)$, $A_{\sqcup}(H) = \bigsqcup_{h \in H} A(h)$, and $A_{\sqcap}(H) = \bigsqcap_{h \in H} A(h)$.

4.2 Reactors

A distributed program is simply a set of reactor declarations written in the DSR language with the following (simplified) syntax:

$$\begin{array}{rrcl} \text{Reactor declarations} & r & ::= & c\{\textit{pc},\, \textit{loc},\, \overline{z\,{:}\,\tau_z},\, \lambda\overline{y\,{:}\,\tau}.s\} \\ \text{Statements} & s & ::= & \texttt{skip} \mid m := e \mid s_1; s_2 \mid \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \\ & & \mid & \texttt{exec}(c,\, \eta,\, \textit{pc},\, \textit{loc},\, \overline{e}) \mid \texttt{chmod}(c,\, \eta,\, \textit{pc},\, \textit{loc},\, \ell) \\ & & \mid & \texttt{setvar}(\langle c,\, \eta\rangle.z,\, e) \end{array}$$

A reactor declaration r contains the following components:

- c, the reactor name.
- *pc*, a lower bound (with respect to the label ordering $\sqsubseteq$) to the labels of any side effects generated by the reactor.
- *loc*, the location of c. In DSR, a reactor may be replicated on multiple hosts to achieve high integrity and availability. Thus, *loc* may be a single host, a set of hosts, or a more complicated replication scheme.
- $\overline{z:\tau_z}$, a list of variable declarations $z_1 : \tau_{z1}, \ldots, z_k : \tau_{zk}$. Variables $\overline{z}$ are free variables of the code to be executed when the reactor is invoked, and are bound to the values provided by the invocation context. For simplicity, an empty variable list, denoted by ϵ, may be omitted from a reactor declaration.
- $\lambda\overline{y:\tau}.s$, the *reactor body*, in which statement s is the code to be executed when the reactor is invoked, and $\overline{y:\tau}$ is a list of variable declarations: $y_1 : \tau_1, \ldots, y_n : \tau_n$. Variables $\overline{y}$ are free variables of s, and are bound to the value arguments carried by an invocation message. The reactor body resembles a lambda term, since invoking a reactor is like invoking a function.

A message invoking reactor c has the form $[\texttt{exec}\,\langle c,\, \eta\rangle :: \textit{pc}, \overline{v}, \textit{loc}, t]$, where $\overline{v}$ is a list of values to which variables $\overline{y}$ of reactor c are bound, and *pc*, *loc* and t are the program counter label, location and timestamp of the sender, respectively. In general, a network message μ has the form $[\alpha :: \beta]$, where α is the *message head* specifying the purpose and destination of the message, and β is the *message body* containing specific parameters. Both α and β are lists of components.

When reactor c receives the invocation message $\mu = [\texttt{exec}\,\langle c,\, \eta\rangle :: \textit{pc}, \overline{v}, \textit{loc}, t]$, it needs to check the validity of the message, because the message may be sent by a host controlled by attackers. Therefore, the closure $\langle c,\, \eta\rangle$ contains an *access control label*: $\textit{acl}(c, \eta) = \ell$, and the constraint $\textit{pc} \sqsubseteq \ell$ is checked to ensure that implicit flows from μ to the thread of $\langle c,\, \eta\rangle$ are secure. Other validity checks are discussed later in

Section 5.2. If request μ is deemed valid, then c creates a new thread to execute s with all the variables in s replaced by certain values: variables $\overline{y}$ are replaced by values $\overline{v}$, and variables $\overline{z}$ are replaced by values from the closure $\langle c,\, \eta\rangle$. The closure bound to $\langle c,\, \eta\rangle$ has the form $\langle c, \eta, \ell, \mathcal{A}, \overline{a}\rangle$, where ℓ is the access control label, $\mathcal{A}$ is a record that maps variables $\overline{z}$ to values, and $\overline{a}$ is a list of additional attributes discussed later in Section 5.2.

In DSR, a statement s may be empty statement `skip`, the assignment statement $m := e$, a sequential composition $s_1; s_2$, an `if` statement, or one of three primitives for invoking a reactor closure or updating the state of a closure:

- `exec`$(c,\, \eta,\, \mathit{pc},\, \mathit{loc},\, \overline{e})$. The `exec` statement sends an `exec` message $[$`exec` $\langle c,\, \eta\rangle :: \mathit{pc}, \overline{v}, \mathit{loc}, t]$ to the hosts where c is located. The list of values $\overline{v}$ are the results of $\overline{e}$, and t is the timestamp of the current thread. After running the `exec` statement, the current thread is terminated. Thus, the `exec` statement explicitly transfers control between reactors, which may be located on different hosts. As in other process calculi (e.g., [60, 29]), reactors do not implicitly return to their invokers. A return from an invoked closure requires an `exec` statement, and the return address (closure) must be sent to the closure explicitly.

- `chmod`$(c,\, \eta,\, \mathit{pc},\, \mathit{loc},\, \ell)$. The `chmod` statement sends a message $[$`chmod` $\langle c,\, \eta\rangle :: \mathit{pc}, \ell, \mathit{loc}, t]$ to the hosts of c. The purpose is to set ℓ as the access control label of closure $\langle c,\, \eta\rangle$. This statement essentially provides a remote security management mechanism, which is useful because a remote reactor may have more precise information for making access control decisions than a local one.

- `setvar`$(\langle c,\, \eta\rangle.z,\, e)$. Suppose value v is the result of e. Then the `setvar` statement sends a message $[$`setvar` $\langle c,\, \eta\rangle.z :: v, t]$ to the hosts to c to set v as the value of variable z in closure $\langle c,\, \eta\rangle$.

4.2.1 Example

Figure 4.2 shows a DSR program that computes $m :=!m_1+!m_2$. In this figure, messages and closures are labeled with sequence numbers indicating their order of occurrence. Assume memory references m_1, m_2 and m are located at hosts h_1, h_2 and h_3, respectively. Reactor c_1 on host h_1 delivers the value of m_1 to h_3; reactor c_2 on host h_2 delivers the value of m_2 to h_3; reactor c_3 on host h_3 computes the sum and updates m. In Figure 4.2, reactor c_1 is invoked with a context identifier η. Then the program is executed as follows:

(1) The thread of $\langle c_1,\ \eta\rangle$ executes the statement $\texttt{setvar}(\langle c_3,\ \texttt{cid}\rangle.z_3,\ !m_1)$. In this statement, variable `cid` represents the context identifier of the current thread and is bound to η. Thus, the `setvar` statement sends the message $[\texttt{setvar}\ \langle c_3,\ \eta\rangle.z_3 :: v_1, t_1]$ to h_3 where v_1 is the value of m_1.

(2) Upon receiving the `setvar` message, h_3 updates the closure $\langle c_3,\ \eta\rangle$ to map z to v_1.

(3) Concurrently with (2), the thread of $\langle c_1,\ \eta\rangle$ invokes $\langle c_2,\ \eta\rangle$ by executing the statement $\texttt{exec}(c_2, \texttt{cid}, pc, h_1)$.

(4) The thread of $\langle c_2,\ \eta\rangle$ invokes closure $\langle c_3,\ \eta\rangle$ by executing the `exec` statement $\texttt{exec}(c_3, \texttt{cid}, pc, h_2, !m_2)$, which sends the message $[\texttt{exec}\ \langle c_3,\ \eta\rangle :: pc, v_2, h_2, t_4]$ to h_3, where v_2 is the value of m_2. Once invoked, $\langle c_3,\ \eta\rangle$ spawns a thread to execute the statement $m := z + y$ with z and y bound to v_1 and v_2, respectively.

An alternative way to implement $m :=!m_1+!m_2$ would be to make reactor c_1 send the value v_1 to c_2 as an argument in the invocation request, and let c_2 compute $v_1+!m_2$ and send the result to c_3. This implementation does not need the closure-based variable binding mechanism. However, the value of m_1 is sent to h_2, imposing an additional security requirement: h_2 must be able to protect the confidentiality of m_1. In essence,

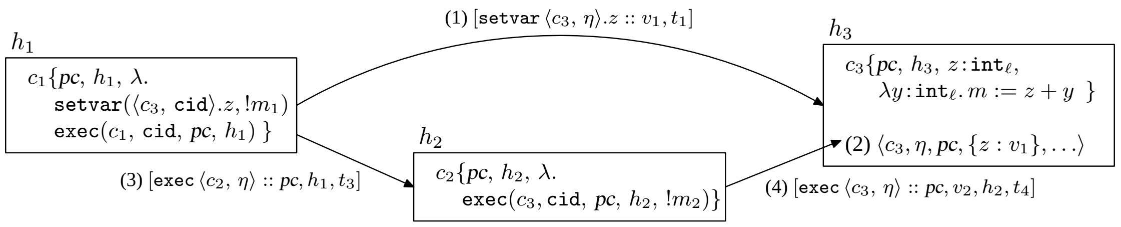

(1) [setvar $\langle c_3, \eta\rangle.z :: v_1, t_1$]
h_1
$c_1\{pc, h_1, \lambda.$ setvar($\langle c_3$, cid$\rangle.z, !m_1$) exec(c_1, cid, pc, h_1) }
h_2
$c_2\{pc, h_2, \lambda.$ exec(c_3, cid, pc, h_2, $!m_2$)}
h_3
$c_3\{pc, h_3, z$:int$_\ell$, λy:int$_\ell$. $m := z + y$ }
(2) $\langle c_3, \eta, pc, \{z : v_1\}, \ldots\rangle$
(3) [exec $\langle c_2, \eta\rangle :: pc, h_1, t_3$]
(4) [exec $\langle c_3, \eta\rangle :: pc, v_2, h_2, t_4$]

Figure 4.2: A distributed program

the closure-based binding mechanism enables the separation of data flow and control flow, providing more flexibility for constructing secure distributed computation.

4.3 Dynamic label checking

As shown in the previous section, the DSR language provides a dynamic label checking mechanism, which is composed of three elements:

- *Dynamic labels,* labels with run-time representations, including the access control label (mutable) in a reactor closure, and the program counter labels in `exec` or `chmod` statements and messages.
- *Dynamic label checks,* such as checking the constraint $pc_\mu \sqsubseteq acl(c, \eta)$ when receiving an `exec` message μ for $\langle c, \eta\rangle$. The label pc_μ is the program counter label of μ.
- *Dynamic label updates,* such as the `chmod` statements that can be used to change the access control label of a reactor closure.

The dynamic label mechanism is necessary because static program analysis alone cannot guarantee that untrusted hosts behave correctly. This section discusses how the dynamic label mechanism is used to enforce information security.

Suppose a message $\mu = [\texttt{exec}\ \langle c, \eta\rangle :: pc_\mu, \overline{v}, loc, t]$ is sent to invoke a closure $\langle c, \eta\rangle$ on host h. The security implication of this invocation is to cause information flows and dependencies between the sender thread and the thread of $\langle c, \eta\rangle$. Let pc_c represent the program counter label of c. Then the following constraints are sufficient to ensure the security of this invocation:

$$I(pc_c) \leq I(loc) \sqcap I(pc_\mu) \qquad\qquad C(pc_\mu) \leq C(pc_c)$$

The first constraint ensures that the sender thread and the sender hosts have sufficient integrity to cause the effects produced by the thread of $\langle c,\, \eta\rangle$. The second constraint prevents information about the program counter of the sender thread from being leaked through the effects of the thread of $\langle c,\, \eta\rangle$. Dually, the confidentiality constraint should be $C(pc_\mu) \leq C(pc_c) \sqcap C(loc_c)$ where loc_c is the location of c. This constraint prevents the information about the program point where μ is sent from being leaked by hosts in loc_c and the effects of the thread of $\langle c,\, \eta\rangle$. In general, the host of a reactor c should always have sufficient confidentiality level to read the information processed by c, and the corresponding constraint $C(pc_c) \leq C(loc_c)$ is enforced by static program analysis. Therefore, the constraint $C(pc_\mu) \leq C(pc_c) \sqcap C(loc_c)$ is equivalent to $C(pc_\mu) \leq C(pc_c)$.

Although the two constraints are sufficient to enforce confidentiality and integrity, they may be overly conservative and lead to the infamous "label creep" problem [17]: the integrity of control flow can only be weakened and may eventually be unable to invoke any reactor.

A static information flow analysis such as the Aimp type system solves the label creep problem by lowering the program counter label at merge points of conditional branches. For example, consider the following code:

$$\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2;\quad S_3$$

Suppose $\Gamma\,;\mathcal{R}\,;pc \vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : \tau$. Then S_1 and S_2 are checked with respect to the program counter label $pc \sqcup \ell_e$. However, S_3 can still be checked with pc because both branches would transfer control to S_3, and the fact that control reaches S_3 does not reveal which branch is taken.

Similarly, in a distributed setting, it may be secure to allow a message μ to invoke $\langle c, \eta \rangle$ even with $pc_{\mu} \not\sqsubseteq pc_{c}$, if $\langle c, \eta \rangle$ is a merge point for high-confidentiality and low-integrity (with respect to pc_{c}) branches, or the *only* invokable closure at or above the security level pc_{c}, which is also called a *linear entry*.

Formally, a closure $\langle c, \eta \rangle$ is a linear entry if there are no threads running at or above the level *pc*, and $\langle c, \eta \rangle$ is the only closure such that $acl(c, \eta) \not\sqsubseteq pc$ and $pc_{c} \sqsubseteq pc$, which mean that $\langle c, \eta \rangle$ may be invoked by a message μ such that $pc_{\mu} \not\sqsubseteq pc_{c}$. In terms of integrity, the existence of a high-integrity linear entry implies that high-integrity computation is suspended, and attackers cannot harm the integrity of computation by invoking a high-integrity linear entry, because that is the only way to continue high-integrity computation.

Using the `chmod` statement, a distributed program can set up linear entries and allow low-integrity (or high-confidentiality) messages to invoke high-integrity (or low-confidentiality) reactors. In general, the creation of a linear entry always happens when a high-integrity reactor c_0 invokes a low-integrity reactor c_1, but eventually control returns to a high-integrity reactor c_2. In this case, the program counter labels of c_0, c_1 and c_2 satisfy $pc_{c_2} \sqcup pc_{c_1} \sqsubseteq pc_{c_0}$ and $pc_{c_2} \not\sqsubseteq pc_{c_1}$. To set up the linear entry $\langle c_2, \eta \rangle$, the thread of $\langle c_0, \eta \rangle$ sends a message $[\texttt{chmod}\, \langle c_2, \eta \rangle :: pc_{c_0}, pc_{c_1}, \ldots]$ to the host of c_2. The `chmod` message changes the access control label of $\langle c_2, \eta \rangle$ to pc_{c_1} such that c_1 is able to invoke $\langle c_2, \eta \rangle$. Moreover, after running the `chmod` message, the thread of $\langle c_0, \eta \rangle$ must be running at the program counter level pc_{c_1} so that there are no high-integrity threads running. Thus, $\langle c_2, \eta \rangle$ becomes a linear entry. When receiving a `chmod` message μ for closure $\langle c, \eta \rangle$, a host performs the same label check $pc_{\mu} \sqsubseteq acl(c, \eta)$ as that for an `exec` message, ensuring that the sender has sufficient integrity. When a closure $\langle c, \eta \rangle$ is first created, its access control label is set as pc_{c} so that the access check enforces the constraint $pc_{\mu} \sqsubseteq pc_{c}$.

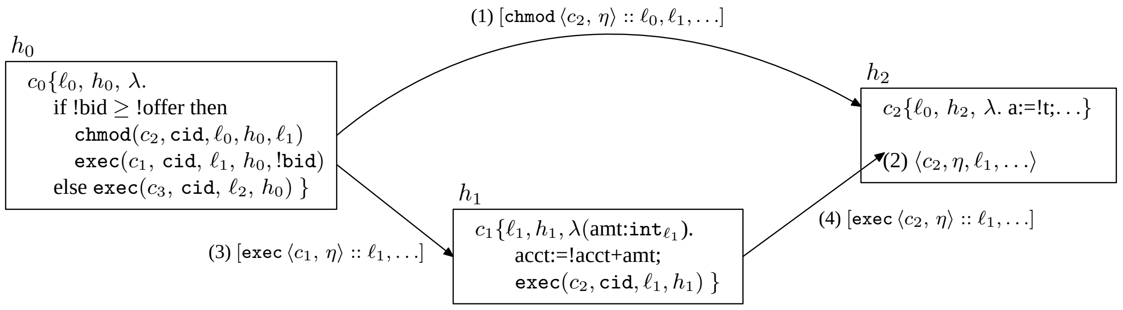

(1) [chmod $\langle c_2,\ \eta\rangle :: \ell_0, \ell_1, \ldots$]
h_0
$c_0\{\ell_0,\ h_0,\ \lambda.$
if !bid $\geq$!offer then
chmod$(c_2, \texttt{cid}, \ell_0, h_0, \ell_1)$
exec$(c_1,\ \texttt{cid},\ \ell_1,\ h_0,$!bid$)$
else exec$(c_3,\ \texttt{cid},\ \ell_2,\ h_0)$ }
h_2
$c_2\{\ell_0,\ h_2,\ \lambda.$ a:=!t;...}
(2) $\langle c_2, \eta, \ell_1, \ldots\rangle$
h_1
$c_1\{\ell_1, h_1, \lambda($amt:int$_{\ell_1})$.
acct:=!acct+amt;
exec$(c_2, \texttt{cid}, \ell_1, h_1)$ }
(3) [exec $\langle c_1,\ \eta\rangle :: \ell_1, \ldots$]
(4) [exec $\langle c_2,\ \eta\rangle :: \ell_1, \ldots$]

Figure 4.3: Linear entry creation

Figure 4.3 illustrates the creation and invocation of a linear entry. The distributed program in Figure 4.3 performs the same computation as lines 3–4 in Figure 3.3. Assume memory locations `bid` and `offer` are located at host h_0, `acct` is located at host h_1, and `a` and `t` are located at host h_2. Reactor c_0 on host h_0 invokes reactor c_1 or c_3 based on whether the value of `bid` is greater than or equal to the value of `offer`. Reactor c_1 updates `acct` and invokes c_2, which assigns the value of `t` to `a`. Suppose c_0 is invoked with a context identifier η, and the value of `bid` is greater than or equal to the value of `offer`. Then the program is executed as follows:

(1) The thread of $\langle c_0,\ \eta\rangle$ executes the statement `chmod`$(c_2, \ldots)$ to send the message [`chmod` $\langle c_2,\ \eta\rangle :: \ell_0, \ell_1, \ldots]$ to h_2.

(2) On receiving the `chmod` message, h_2 changes the access control label of $\langle c_2,\ \eta\rangle$ to ℓ_1. Note that the program counter label of the message is ℓ_0, which passes the access control check since *acl*(c_2, η) is ℓ_0 initially.

(3) Concurrently with (2), the thread of $\langle c_0,\ \eta\rangle$ invokes $\langle c_1,\ \eta\rangle$ by running the statement `exec`$(c_1, \texttt{cid}, \ell_1, h_0,$!`bid`$)$. The program counter label of the `exec` statement is ℓ_1 instead of ℓ_0, since the program counter label is bounded by ℓ_1 after the `chmod` statement.

(4) The thread of $\langle c_1,\ \eta\rangle$ invokes $\langle c_2,\ \eta\rangle$ after updating `acct`. The invocation message [`exec` $\langle c_2,\ \eta\rangle :: \ell_1, \ldots]$ is accepted because *acl*(c_2, η) is ℓ_1.

4.4 Replication and message synthesis

Replicating code and data is an effective way to achieve fault tolerance and ensure integrity and availability. In DSR, both reactors and memory references may be replicated on multiple hosts. Suppose reactor c is replicated on a set of hosts H. Then other reactors interact with c as follows:

- Any message for c is sent to all the hosts in H.
- The replicas of c process incoming messages independently of each other. To make this possible, all the program states of c have a local copy on every host in H. In particular, every memory reference (location) accessed by c is also replicated on H.
- If invoked with the same context identifier, the replicas of c are supposed to produce the same messages. Thus, the receiver host h of such a message μ may receive the replicas of μ from different hosts in H. The redundancy is crucial for achieving fault tolerance. Some hosts in H may be compromised, and these bad hosts may send corrupted messages or simply not send anything. In general, the replicas of μ received by h contain some correct ones, which are the same, and some bad ones, which can be arbitrarily inconsistent. It is up to h to identify the correct μ from those message replicas. This process is called *message synthesis*, and the algorithm for identifying the correct message is called a *message synthesizer*.

For example, consider the program in Figure 4.4, which computes $m := !m_1 + !m_2$ like the program in Figure 4.2, except that m_2 is replicated on three hosts h_{21}, h_{22} and h_{23}. Accordingly, the reactor c_2 that reads the value of m_2 is also replicated on hosts h_{21}, h_{22} and h_{23}. To invoke c_2, the statement $\texttt{exec}(c_1, \texttt{cid}, pc, h_1)$ of c_1 sends an `exec` message to all three hosts where c_2 is replicated. Then each replica of c_2 sends an `exec`

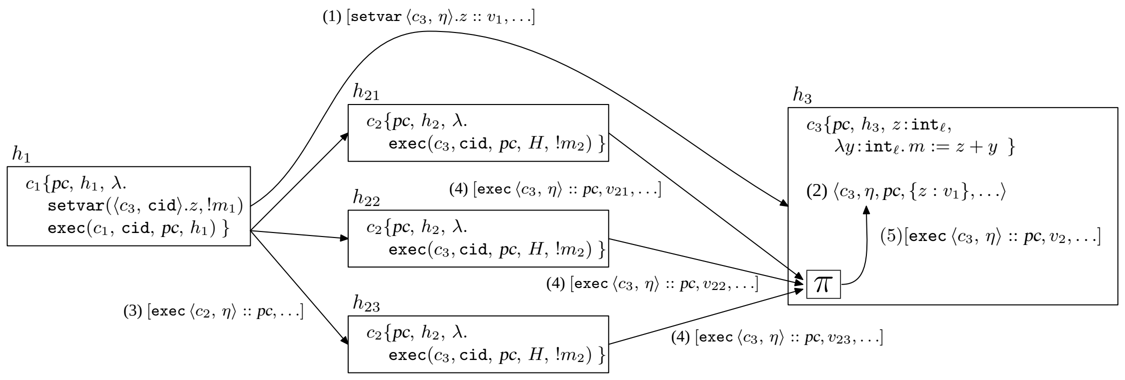

(1) [setvar $\langle c_3,\ \eta\rangle.z :: v_1, \ldots$]
h_{21}
$c_2\{pc,\ h_2,\ \lambda.$ exec$(c_3, \texttt{cid}, pc, H, !m_2)$ }
h_3
$c_3\{pc,\ h_3,\ z{:}\texttt{int}_\ell,$ $\lambda y{:}\texttt{int}_\ell.\ m := z + y$ }
h_1
$c_1\{pc,\ h_1,\ \lambda.$ setvar$(\langle c_3,\ \texttt{cid}\rangle.z, !m_1)$ exec$(c_1,\ \texttt{cid},\ pc,\ h_1)$ }
(4) [exec $\langle c_3,\ \eta\rangle :: pc, v_{21}, \ldots$]
h_{22}
(2) $\langle c_3, \eta, pc, \{z : v_1\}, \ldots\rangle$
$c_2\{pc,\ h_2,\ \lambda.$ exec$(c_3, \texttt{cid}, pc, H, !m_2)$ }
(5)[exec $\langle c_3,\ \eta\rangle :: pc, v_2, \ldots$]
(4) [exec $\langle c_3,\ \eta\rangle :: pc, v_{22}, \ldots$]
π
(3) [exec $\langle c_2,\ \eta\rangle :: pc, \ldots$]
h_{23}
$c_2\{pc,\ h_2,\ \lambda.$ exec$(c_3, \texttt{cid}, pc, H, !m_2)$ }
(4) [exec $\langle c_3,\ \eta\rangle :: pc, v_{23}, \ldots$]

Figure 4.4: Replication example

message containing the local value of m_2 to h_3. The `exec` messages from h_{12}, h_{22} and h_{23} are synthesized into a single message $[\texttt{exec}\ \langle c_3,\ \eta\rangle :: pc, v_2, \ldots]$ by the synthesizer π on h_3. By doing a majority voting, the synthesizer π can produce the correct message if only one host of h_{21}, h_{22} and h_{23} is compromised.

4.4.1 Analyzing security assurances of message synthesizers

Intuitively, the main goal of a message synthesizer is to identify the correct message. By "correct", we mean "with sufficient integrity". Let $I(\mu)$ be the integrity label of μ, specifying the integrity level of the contents of μ. When receiving the replicas of μ, message synthesizer π is intended to produce the correct μ, if $I(\mu) \not\leq l_{\texttt{A}}$, which says that μ is a high-integrity message and should not be compromised by attackers. If $I(\mu) \leq l_{\texttt{A}}$, then μ is a low-integrity message, and the system is considered secure even if μ is compromised, as discussed in Chapter 2. In other words, if $I(\mu) \leq l_{\texttt{A}}$, the synthesizer π is under no obligation to ensure the correctness of μ. Therefore, if π determines that $I(\mu) \leq l_{\texttt{A}}$ holds, it has two options: (1) reporting an error, and (2) producing μ while knowing that μ might be incorrect. Both options do not reduce the integrity assurance. But the first option effectively makes the message unavailable, reducing availability assurance. Therefore, the second option is generally preferred.

Now we demonstrate how to analyze the security assurances of a message synthesis algorithm. Consider the synthesizer π in Figure 4.4. Suppose π produces μ using the following algorithm, which returns $[\texttt{exec}\,\langle c_3,\,\eta\rangle\ ::\ pc, v_{21}, \ldots]$ if v_{21}, v_{22} and v_{23} are equal, and otherwise returns none, which means that the message is unavailable.

```
if ($v_{21}$ = $v_{22}$ && $v_{21}$ = $v_{23}$) then return $[\texttt{exec}\,\langle c_3,\,\eta\rangle :: pc, v_{21}, \ldots]$
else return none
```

With this algorithm, attackers can convince π to produce a message μ containing a fabricated value by compromising the integrity of all three hosts h_{21}, h_{22} and h_{23}. Therefore, this algorithm imposes the following label constraint:

$$I(\mu) \le I(h_{21}) \sqcup I(h_{22}) \sqcup I(h_{23}).$$

At the same time, if attackers compromise the integrity of one host and send a message replica inconsistent with the other two, μ is unavailable according to the algorithm. Thus, the algorithm places an upper bound on the available label $A(\mu)$ of μ:

$$A(\mu) \le I(h_{21}) \sqcap I(h_{22}) \sqcap I(h_{23}).$$

To increase the availability assurance at the expense of integrity assurance, π can use the following algorithm:

```
if ($v_{21}$ = $v_{22}$ || $v_{21}$ = $v_{23}$) return $[\texttt{exec}\,\langle c_3,\,\eta\rangle :: pc, v_{21}, \ldots]$
if ($v_{22}$ = $v_{23}$) return $[\texttt{exec}\,\langle c_3,\,\eta\rangle :: pc, v_{22}, \ldots]$
else return none
```

in which, π produces message μ if two of v_{21}, v_{22} and v_{23} are equal. Attackers can compromise the integrity of μ by compromising the integrity of two hosts. At the same time, to make μ unavailable, attackers only need to make v_{21}, v_{22} and v_{23} all different from each other by compromising the integrity of two hosts. Thus, the algorithm imposes the follow constraints:

$$I(\mu) \le (I(h_{21}) \sqcup I(h_{22})) \sqcap (I(h_{22}) \sqcup I(h_{23})) \sqcap (I(h_{21} \sqcup I(h_{23}))$$

$$A(\mu) \le (I(h_{21}) \sqcup I(h_{22})) \sqcap (I(h_{22}) \sqcup I(h_{23})) \sqcap (I(h_{21} \sqcup I(h_{23}))$$

Suppose the integrity constraint holds, and v_{21}, v_{22} and v_{23} differ from each other. Then $I(\mu) \le l_{\mathtt{A}}$ can be concluded. There are only two cases in which attackers can make v_{21}, v_{22} and v_{23} differ from each other. First, the values are computed using low-integrity data. In this case, the type system of DSR ensures $I(\mu) \le l_{\mathtt{A}}$, as discussed in the next chapter. Second, attackers are able to compromise two hosts of h_{21}, h_{22} and h_{23}. Without loss of generality, suppose attackers compromise the integrity of h_{21} and h_{22}. Then $I(h_{21}) \sqcup I(h_{22}) \le l_{\mathtt{A}}$, which implies $I(\mu) \le l_{\mathtt{A}}$ by the above integrity constraint. Based on the discussion at the beginning of this section, π can choose to return a message with an arbitrary v_2 without reducing integrity assurance. This idea is applied to the following algorithm:

```
if ($v_{21}$ = $v_{22}$ || $v_{21}$ = $v_{23}$) return $[\mathtt{exec}\,\langle c_3,\,\eta\rangle :: pc, v_{21}, \ldots]$
if ($v_{22}$ = $v_{23}$) return $[\mathtt{exec}\,\langle c_3,\,\eta\rangle :: pc, v_{22}, \ldots]$
else return $[\mathtt{exec}\,\langle c_3,\,\eta\rangle :: pc, 0, \ldots]$
```

which returns a message with a default value 0 if the three incoming message replicas differ from each other. This algorithm provides higher availability assurance than the last one without sacrificing integrity assurance. In fact, the algorithm is bound to return some message μ. Thus, μ is available as long as hosts h_{21}, h_{22} and h_{23} are available. Therefore,

$$A(\mu) = A(h_{21}) \sqcap A(h_{22}) \sqcap A(h_{23}).$$

Qualified host sets

In general, given a message synthesizer π, a host set H is *qualified* with respect to π, written as $\mathit{qualified}_\pi(H)$, if receiving messages from all the hosts in H guarantees π will produce a message. For example, $\{h_{21}, h_{22}, h_{23}\}$ is a qualified set for the third synthesizer algorithm discussed above. Suppose a reactor replicated on H sends a message to be synthesized by π. Then the availability of μ is guaranteed if a π-qualified subset of H is available. Thus, $A(\mu)$ is enforced if $A(\mu) \le A(H, \pi)$, where H is the sender set of μ, and $A(H,\pi)$ is computed as follows:

$$A(H,\pi) = \bigsqcup_{H' \subseteq H \ \wedge \ \textit{qualified}_{\pi}(H')} A_{\sqcap}(H')$$

4.4.2 Label threshold synthesizer

The label threshold synthesizer $\texttt{LT}[l]$, parameterized with an integrity label l, produces a message μ if it receives μ from a set of hosts H satisfying $l \leq I_{\sqcup}(H)$, or if it can determine from the incoming messages that $l \leq l_{\texttt{A}}$ holds. Using the label threshold synthesizer to handle exec messages allows a reactor to be invoked by any host set with sufficient integrity. This flexibility is important because a reactor, like a remote function, may be invoked by different callers.

The components of a message may have different integrity labels. Thus, $\texttt{LT}[l]$ synthesizes the message components separately. For example, suppose $\texttt{LT}[l]$ receives message $\mu_i = [\texttt{exec}\,\langle c,\, \eta\rangle :: \textit{pc}_i, \overline{v_i}, \textit{loc}_i, t_i]$ from host h_i for $i \in \{1,\ldots,n\}$, and produces a message $\mu = [\texttt{exec}\,\langle c,\, \eta\rangle :: \textit{pc}, \overline{v}, \textit{loc}, t]$. Then $\texttt{LT}[l]$ uses $\textit{pc}_1,\ldots,\textit{pc}_n$ to produce *pc*, and $v_{11},\ldots,v_{n1}$ to produce v_1, and so on. Intuitively, $\texttt{LT}[l]$ produces messages with an integrity level upper bounded by l. Therefore, the integrity label of every message component is bounded by l.

First, let us consider how $\texttt{LT}[l]$ synthesizes the corresponding components $v_1,\ldots,v_n$ into v, where v_i belongs to message μ_i sent by host h_i. This synthesis algorithm can be described by the following pseudo-code.

```
LT[$l$]($H$, $v_1$, ..., $v_n$) {
    if $\exists H' \subseteq H.\ (l \leq I_{\sqcup}(H') \ \wedge\ \forall h_j \in H'.\ v_j = v)$
        return $v$
    if ($\textit{qualified}_{\texttt{LT}[l]}(H)$) return $v_d$
    else return none
}
```

First, this algorithm returns value v if v is sent by a host set H' satisfying the label constraint: $l \leq I_{\sqcup}(H')$. Second, if no subset of H with sufficient integrity endorses the same value, and H is LT$[l]$-qualified, then the algorithm returns a default value v_d. In this case, returning a default value is justified because the following qualified condition for LT$[l]$ implies $l' \leq l_{\mathtt{A}}$, where l' is the integrity label of the component.

> H is LT$[l]$-qualified if and only if H cannot be partitioned into two disjoint sets H_1 and H_2 such that $l \not\leq I_{\sqcup}(H_1)$ and $l \not\leq I_{\sqcup}(H_2)$.

Now we show that the qualified condition implies $l' \leq l_{\mathtt{A}}$ if there does not exist a subset H' of H such that for any $h_i \in H'$, $v_i = v$, and $l \leq I_{\sqcup}(H')$. Suppose, by way of contradiction, $l' \not\leq l_{\mathtt{A}}$. Then attackers can only compromise v_i by compromising h_i. Then the set H_g of good (high-integrity) hosts send the same value. Therefore, $l \not\leq I_{\sqcup}(H_g)$ holds, which implies $l \leq I_{\sqcup}(H_b)$ where $H_b = H - H_g$ is the set of bad hosts, since H satisfies the above qualified condition for LT$[l]$. Therefore, $l \leq l_{\mathtt{A}}$, which contradicts $l' \not\leq l_{\mathtt{A}} \wedge l' \leq l$.

In some sense, the LT$[l]$-qualified condition is like the byzantine fault tolerance condition, which requires $2f + 1$ hosts to tolerate f failures. A set of $2f + 1$ hosts cannot be partitioned into two disjoint sets such that either set may be composed of only failed hosts, if there are at most f failures.

It is straightforward to construct LT$[l]$ for messages using the LT$[l]$ algorithm for message components. Suppose LT$[l]$ receives messages $\mu_1, \ldots, \mu_n$ from hosts H, and $\mu_i = [\alpha :: v_{i1}, \ldots, v_{ik}]$. Here notation is abused a bit: v_{ij} may be a value, a location or a timestamp. The following pseudo-code describes how LT$[l]$ synthesizes these messages.

```
LT[$l$]($H$, $\mu_1$, ..., $\mu_n$) {
    if ($\forall j.$ LT[$l$]($H$, $v_{j1}, \ldots, v_{jn}$) $= v_j \wedge v_j \neq$ none)
        return $[\alpha :: v_1, \ldots, v_k]$
    else return none
}
```

4.5 Using quorum systems

The label threshold synthesizer $\texttt{LT}[l]$ is based on the assumption that all replicas on good hosts generate the same high-integrity outputs. The assumption requires that good hosts agree on their local states. In particular, if the contents of a message μ depend on the value of some memory reference m, replicas of m on good hosts must have the same value. Otherwise, the replicas of μ cannot be synthesized using $\texttt{LT}[l]$. To maintain the consistency (equality) between the replicas of m on good hosts essentially requires that the updates to m are synchronized on all the hosts of m. However, this strong synchronization requirement makes it difficult to guarantee the availability of a memory write operation because all the hosts of m need to be available to synchronize a write operation on m. To achieve high availability for both memory read and write operations, we need more complex replication schemes and message synthesis algorithms.

Quorum systems are a well-known replication scheme for ensuring the consistency and availability of replicated data [35, 51]. A quorum system $\mathcal{Q}$ is a collection of sets (quorums) of hosts, having the form $\langle H,\ W_1, \ldots, W_n\rangle$, where H is all the hosts in $\mathcal{Q}$, and quorums $W_1, \ldots, W_n$ are subsets of H. Suppose a memory location m is replicated on a quorum system. Then an update to m is considered *stable* (finished) if it is completed on a quorum of hosts. In DSR as in some other quorum systems [51], timestamps are used to distinguish different versions of the same replicated memory location. A read operation can get the most recent update by consulting with a set R of hosts intersecting every quorum. In some literature [52], each W_i is called a *write quorum*, and R is called a *read quorum*. Using quorum protocols, only a subset of hosts is needed to finish either a read or write operation. That is why replicating a memory location in a quorum system can potentially achieve high availability for both reads and writes.

This section describes how the DSR language incorporates quorum system protocols.

4.5.1 Tracking timestamps

In quorum protocols, timestamps play an important role: to distinguish different versions of the contents of a memory reference.

To track timestamps, DSR provides the following mechanisms:

- **Thread timestamps**

  Each thread has a timestamp that is incremented with every execution step of the thread.

- **Message timestamps**

  When a thread θ sends a message μ, the current timestamp of θ is embedded in μ.

- **Versioned memory**

  The contents of a memory reference are associated with timestamps that indicate whether the contents are up to date. If a memory reference m is assigned value v at timestamp t, the *versioned value* $v@t$ is stored in the local memory as a version of m.

  A new version of m does not overwrite old versions of m. This is necessary because execution is asynchronous. It is possible that a thread on host h updates m at time t while another thread on h at a logically earlier time still needs to read m. Old versions of m resolve this write-read conflict.

  In general, a local memory on a host maps a memory reference m to a set of versioned values $v@t$. A derereference $!m$ evaluated at time t results in the most recent version of m by the time t.

  If m is replicated on multiple hosts H, it is possible that some hosts in H may be running behind, and they do not have an up-to-date version of m. Thus, the type system of DSR prevents a versioned value from being used in any computation, since the value may be outdated. To compute using the value of m, a host needs to obtain the replicas of m from sufficient number of hosts and figure out the most recent version.

4.5.2 Quorum read

In general, to read the value of m replicated on hosts $H = \{h_1, \dots, h_n\}$, a program invokes a reactor c replicated on H, and each c on host h_i will send back its local version $v_i@t_i$ of m in a `setvar` message $[\texttt{setvar}\,\langle c',\, \eta\rangle.z \,::\, v_i@t_i, t'_i]$. After a host h' of c' receives those `setvar` messages, h' uses a message synthesizer to produce a message $[\texttt{setvar}\,\langle c',\, \eta\rangle.z \,::\, v]$ such that $v@t$ is the most recent version of m by the time those `setvar` messages are sent. If H forms a quorum system $\mathcal{Q}$, then h' uses a *quorum read* synthesizer, written as $\texttt{QR}[\mathcal{Q}, l]$, where l is the integrity label of m.

Suppose the most recent update to m by the time c is invoked is stable. Then at least all the hosts in one quorum of $\mathcal{Q}$ complete the update. Therefore, if $\texttt{QR}[\mathcal{Q}, l]$ receives sufficient `setvar` messages from every quorum of $\mathcal{Q}$, it can identify the needed value with sufficient integrity. Based on this insight, a host set R is $\texttt{QR}[\mathcal{Q}, l]$-qualified if the following condition holds:

$$\forall\, W \in \mathcal{Q}.\ \textit{qualified}_{\texttt{LT}[l]}(W \cap R)$$

The condition requires that the intersection between R and each quorum W is a qualified set for $\texttt{LT}[l]$. Intuitively, the messages from $W \cap R$ are sufficient to determine the value held by W, if W is the quorum holding the most recent version of m. Suppose the quorum W holds the most recent value $v@t$. Then any good host in $W \cap R$ must provide the value $v@t$. Furthermore, any good host in $\mathcal{Q}$ would not provide a value $v'@t'$ such that $t < t'$, since $v@t$ is the most recent version.

The $\texttt{QR}[\mathcal{Q}, l]$-qualified condition can be viewed as a generalization of the requirement that the intersection of any read quorum and any write quorum has a size at least $2f + 1$ in order to tolerate f byzantine failures.

Suppose $\mu_i = [\texttt{setvar}\,\langle c,\, \eta\rangle.z \,::\, v_i@t_i, t'_i]$ $(1 \le i \le n)$ from R are received. Then the following $\texttt{QR}[\mathcal{Q}, l]$ algorithm is able to return the appropriate version of m with sufficient integrity, if R is $\texttt{QR}[\mathcal{Q}, l]$-qualified.

```
QR[$\mathcal{Q}, l$]($R$, $\mu_1$, ..., $\mu_n$) {
  if qualified$_{\texttt{QR}[\mathcal{Q},l]}(R)$
    if ($R \vdash v@t : l$ and $\forall t_i . t < t_i \Rightarrow\ R \not\vdash v_i@t_i : l$)
        return [setvar $\langle c,\, \eta\rangle.z :: v$]
    else return [setvar $\langle c,\, \eta\rangle.z :: v_d$]
  else return none
}
```

In this algorithm, the notation $R \vdash v@t : l$ means that there exists a subset R' of R such that $l \le I_{\sqcup}(R')$ and for any host h_j in R', $v_j@t_j = v@t$. Intuitively, the notation means that $v@t$ is a version of m with sufficient integrity. Essentially, this algorithm returns the versioned value with sufficient integrity and the highest timestamp. If there does not exist $v@t$ such that $R \vdash v@t : l$, then $l \le l_{\texttt{A}}$ must hold, which justifies returning a message with a default value.

Applying the general formula for $A(H, \pi)$, the availability guarantee of a read operation on the quorum system $\mathcal{Q} = \langle H, \overline{W}\rangle$ is as follows:

$$A(H, \texttt{QR}[\mathcal{Q}, l]) = \bigsqcup_{R \subseteq H \wedge \mathit{qualified}_{\texttt{QR}[\mathcal{Q},l]}(R)} A_{\sqcap}(R)$$

4.5.3 Quorum write

Similar to a quorum read operation, a write operation for reference m replicated on $\mathcal{Q}$ is performed by invoking some reactor c that is also replicated on $\mathcal{Q}$ and contains an assignment to m.

The quorum read synthesizer assumes that an update to m is stable by the time m is read again. Suppose m is replicated on $\mathcal{Q}$ and updated by reactor c. To maintain the stability of m, the reactor c' invoked by c is required to wait for the invocation requests from a quorum of $\mathcal{Q}$. This ensures that the execution of c, including the update to m, is completed on a quorum, and the update is therefore guaranteed to be stable by the time m is read by another reactor. Recall that an `exec` message has the form $[\texttt{exec}\,\langle c,\,\eta\rangle \;::\; \mathit{pc}, \overline{v}, \mathit{loc}, t]$, where *loc* may be used to describe the quorum system of the memory locations being written to by the sender reactor. So the receiver has all the information to do the quorum-write check.

Essentially, an available quorum ensures that a write operation terminates. Therefore, the availability guarantee of a quorum write is as follows:

$$A_{\texttt{write}}(\mathcal{Q}) = \bigsqcup_{W \in \mathcal{Q}} A_{\sqcap}(W)$$

4.6 Multi-level timestamps

Timestamps introduce new, potentially covert, information channels. First, timestamps are incremented at execution steps, and thus contain information about the execution path. Second, in quorum protocols, timestamps can affect the result of a memory read.

We want to increment the timestamp so that (1) it stays consistent across different good replicas, and (2) its value only depends on the part of the execution path with label ℓ such that $\ell \sqsubseteq \mathit{pc}$ (where *pc* is the current program counter label). To achieve this, DSR uses *multi-level timestamps* that track execution history at different security levels. To simplify computation local to a reactor, a timestamp has two parts: the *global part* tracks the invocations of reactors at different security levels; the *local part* tracks the execution steps of a local thread. Formally, a multi-level timestamp is a tuple $\langle \overline{\mathit{pc}\!:\!n}, \delta\rangle$: the global part $\overline{\mathit{pc}\!:\!n}$ is a list of pairs $\langle \mathit{pc}_1 : n_1, \ldots, \mathit{pc}_k : n_k\rangle$, where $\mathit{pc}_1, \ldots, \mathit{pc}_k$ are program counter labels satisfying the constraint $\mathit{pc}_1 \sqsubseteq \ldots \sqsubseteq \mathit{pc}_k$, and $n_1, \ldots, n_k$ are integers. Intuitively, the component $\mathit{pc}_i : n_i$ means that the number of reactors invoked at the level pc_i is n_i. The local part δ is less significant than the global part in timestamp comparison, and its concrete form will be discussed later in Section 5.2.

When a multi-level timestamp t is incremented at a program point with label *pc*, the high-confidentiality and low-integrity (with respect to *pc*) components of t are discarded, because those components are not needed to track the time at the level *pc*, and discarding those components prevents insecure information flows. Furthermore, the local part of a timestamp after the increment is reset to an initial state δ_0. Suppose $t = \langle pc_1 : n_1, \ldots, pc_k : n_k;\ \delta\rangle$, and $pc_i \sqsubseteq pc$ and $pc_{i+1} \not\sqsubseteq pc$. Then $pc_{i+1} : n_{i+1}, \ldots, pc_k : n_k$ are low-integrity components to be discarded, and incrementing t at level *pc* is carried out by the following formula:

$$inc(t,\ pc) = \begin{cases} \langle pc_1 : n_1, \ldots, pc_i : n_i + 1;\ \delta_0\rangle & \text{if} \quad pc_i = pc \\ \langle pc_1 : n_1, \ldots, pc_i : n_i, pc : 1;\ \delta_0\rangle & \text{if} \quad pc_i \neq pc \end{cases}$$

When comparing two timestamps, low global components are more significant than high ones. Therefore, for any *pc*, we always have $t < inc(t,\ pc)$.

4.7 Example

Like Figure 4.3, the distributed program in Figure 4.5 performs the same computation as lines 3–4 in Figure 3.3, except that reference `bid` is replicated on a quorum system $\mathcal{Q}$. This example illustrates how to read a memory reference replicated on a quorum system and how timestamps are tracked in a system. Reactor `readbid` is used to read the value of `bid` and send the value to c_1 so that c_1 can compute !`acct`+!`bid`. Reactor `readbid` is replicated on the same quorum system as `price` so that each replica of `readbid` can read the local replica of `price` and send it to host h_1 using a `setvar` message. Host h_1 uses `QR`$[\mathcal{Q}, l]$ (specified in the declaration of c_1, and $l = I(\ell_1)$) to synthesize the `setvar` messages sent by replicas of `readbid`. If `QR`$[\mathcal{Q}, l]$ produces a message [`setvar` $\langle c_1,\ \eta\rangle$.`amt` :: v], then the value v is recorded in the closure $\langle c_1,\ \eta\rangle$ as the value of `amt`.

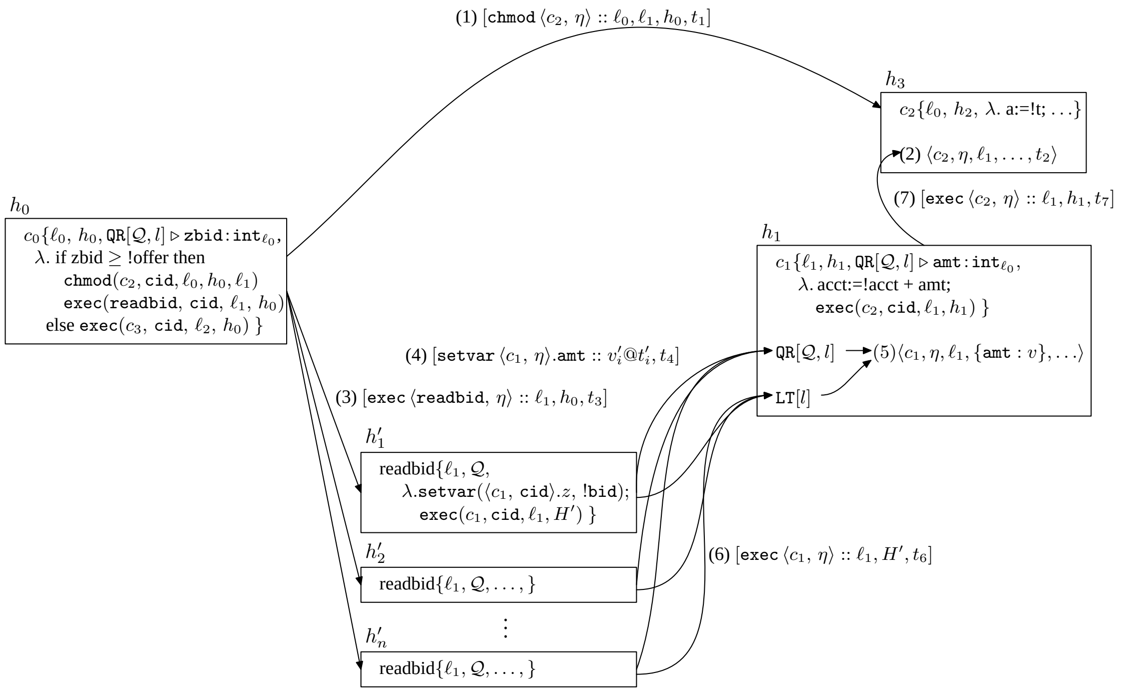

(1) [chmod $\langle c_2,\,\eta\rangle :: \ell_0, \ell_1, h_0, t_1$]
h_3
$c_2\{\ell_0,\, h_2,\, \lambda$. a:=!t; $\ldots\}$
(2) $\langle c_2, \eta, \ell_1, \ldots, t_2\rangle$
(7) [exec $\langle c_2,\,\eta\rangle :: \ell_1, h_1, t_7$]
h_0
$c_0\{\ell_0,\, h_0,$ QR$[\mathcal{Q}, l] \triangleright$ zbid : int$_{\ell_0}$,
λ. if zbid $\geq$!offer then
chmod(c_2, cid, ℓ_0, h_0, ℓ_1)
exec(readbid, cid, ℓ_1, h_0)
else exec(c_3, cid, ℓ_2, h_0) }
h_1
$c_1\{\ell_1, h_1,$ QR$[\mathcal{Q}, l] \triangleright$ amt : int$_{\ell_0}$,
λ. acct:=!acct + amt;
exec(c_2, cid, ℓ_1, h_1) }
QR$[\mathcal{Q}, l]$
(5)$\langle c_1, \eta, \ell_1, \{$amt : $v\}, \ldots\rangle$
LT$[l]$
(4) [setvar $\langle c_1,\,\eta\rangle$.amt :: $v_i'@t_i', t_4$]
(3) [exec $\langle$readbid$,\,\eta\rangle :: \ell_1, h_0, t_3$]
h_1'
readbid$\{\ell_1, \mathcal{Q}$,
λ.setvar($\langle c_1$, cid$\rangle.z$, !bid);
exec(c_1, cid, ℓ_1, H') }
h_2'
readbid$\{\ell_1, \mathcal{Q}, \ldots, \}$
(6) [exec $\langle c_1,\,\eta\rangle :: \ell_1, H', t_6$]
h_n'
readbid$\{\ell_1, \mathcal{Q}, \ldots, \}$

Figure 4.5: Quorum replication and timestamps

To track the time globally, every message carries the timestamp of its sender. Suppose the timestamp of $\langle c_0,\, \eta\rangle$ is $t_0 = \langle \ell_0 : 1; \delta_0\rangle$. Then the timestamps are incremented as follows:

(1) $t_1 = \langle \ell_0 : 1; \delta_{11}\rangle$. The local part δ_{11} is obtained by incrementing δ_0.

(2) $t_2 = \mathit{inc}(t_1,\, \ell_0) = \langle \ell_0 : 2; \delta_0\rangle$. On receiving the `chmod` message, host h_3 increments t_1 to obtain t_2 and stores t_2 in closure $\langle c_2,\, \eta\rangle$. When $\langle c_2,\, \eta\rangle$ is invoked by a low-integrity message μ, the initial timestamp of the thread of $\langle c_2,\, \eta\rangle$ will be t_2 instead of the timestamp obtained by incrementing the timestamp of the low-integrity μ.

(3) $t_3 = \langle \ell_0 : 1; \delta_{12}\rangle$. The local part δ_{12} is obtained by incrementing δ_{11}.

(4) $t_4 = \mathit{inc}(t_3,\, \ell_1) = \langle \ell_0 : 1, \ell_1 : 1; \delta_0\rangle$. Since the program counter label of `readbid` is ℓ_1, the initial timestamp for $\langle$`readbid`$,\, \eta\rangle$ is obtained by incrementing t_3 at the level ℓ_1.

(6) $t_6 = \langle \ell_0 : 1, \ell_1 : 1; \delta_{61} \rangle$. The local part δ_{61} is obtained by incrementing δ_0.

(7) $t_7 = \langle \ell_0 : 1, \ell_1 : 2; \delta_{71} \rangle$. The initial timestamp of $\langle c_1,\ \eta \rangle$ is $\mathit{inc}(t_6,\ \ell_1) = \langle \ell_0 : 1, \ell_1 : 2; \delta_0 \rangle$. Thus, t_7 is obtained by incrementing the local part of the initial timestamp. Note that $t_2 = \mathit{inc}(t_7,\ \ell_0)$, which means that if t_7 is correct, the initial timestamp of $\langle c_2,\ \eta \rangle$ is the same as that obtained by incrementing t_7.

4.8 Related work

The design of the reactor model and DSR is inspired by concurrent process calculi and by object-oriented programming [2]. Well-known examples of process calculi include CSP [36], CCS [59], the pi calculus [60], and the join calculus [29]. In these calculi, process communication is modeled by message passing. The key difference between DSR and prior process calculi is that DSR provides explicit language constructs for replication and run-time security labels, allowing these mechanisms to be statically analyzed by a type system.

There has been some work on type-based information flow analyses for process calculi. Honda and Yoshida [37] develop a typed π-calculus for secure information flow based on linear/affine type disciplines. Zdancewic and Myers [103] present a security-typed language $\lambda^{\text{PAR}}_{\text{SEC}}$, which extends the join calculus with linear channels, and demonstrate that internal timing attacks can be prevented by eliminating races. In these languages, linear channels provide additional structure to facilitate more accurate information flow analyses. Linearity also plays an important role in security types for low-level languages in continuation passing style [102]. In DSR, a closure can be viewed as a linear continuation, since it can be invoked only once.

Quorum systems [84, 15, 35, 51, 8, 5] are a well studied technique for improving fault tolerance in distributed systems. Quorum systems achieve high data availability by providing multiple quorums capable of carrying out read and write operations. If some hosts in one quorum fail to respond, another quorum may still be available. Martin, Alvisi and Dahlin [52] proposed Small Byzantine Quorum protocols, which place different constraints on read and write quorums, and require that every read quorum intersects with every write quorum rather than every two quorums intersect with each other.

The Replica Management System (RMS) [48] computes a placement and replication level for an object based on programmer-specified availability and performance parameters. RMS does not consider Byzantine failures or other security properties.

Keeping multiple versions of the program state is a well-known approach to efficient concurrency control, especially for providing consistency for read-only transactions [13, 92, 4]. This work uses the approach to provide consistency for reactor replicas running behind.

Multipart timestamps have been used to provide a *vector clock* scheme [90, 68, 11, 43, 44, 42], in which a timestamp contains multiple components tracking incomparable times, such as the times of different processes. This work uses multiple components in a timestamp to count events at different security levels.

Walker et al. [91] designed λ_{zap}, a lambda calculus that models intermittent data faults, and they use it to formalize the idea of achieving fault tolerance through replication and majority voting. However, λ_{zap} is designed for a single-machine platform with at most one integrity failure.

The DSR language supports run-time labels, and the evaluation model of DSR relies on run-time label checking to control information flows between hosts. Dynamic information flow control mechanisms [94, 95] track security labels dynamically and use run-time security checks to constrain information propagation. These mechanisms are transparent to programs, but they cannot prevent illegal implicit flows arising from the control flow paths not taken at run time. The Jif language [62, 65] is the first security-typed language with explicit language features for run-time labels and run-time security checks. Some recent work focuses on presenting sound static analyses of run-time labels. Tse and Zdancewic proved a noninterference result for a security-typed lambda calculus ($\lambda_{\texttt{RP}}$) with run-time principals [86]. Zheng and Myers proved a noninterference result for a security-typed lambda calculus ($\lambda_{\texttt{DSEC}}$) with run-time labels [106].

Various general security models [56, 83, 28] have been proposed to incorporate dynamic labeling. Unlike noninterference, these models define what it means for a system to be secure according to a certain relabeling policy, which may allow downgrading labels.

Chapter 5
The DSR language

This chapter formally describes the syntax and semantics of DSR and proves that the type system of DSR can enforce the confidentiality and integrity noninterference properties.

5.1 Syntax

The syntax of the DSR language is shown in Figure 5.1. We use the name l to range over a lattice of base labels $\mathcal{L}$, x, y and z to range over variable names, m to range over a space of memory locations, h to range over host names, and c to range over reactor names.

To facilitate writing generic code, reactors may be polymorphic. The full form of a reactor declaration is:

$$c[\overline{x:\sigma}]\{pc,\ \mathcal{Q},\ \overline{\pi \triangleright z:\tau_1},\ \lambda\overline{y:\tau_2}.\ s\}$$

where $\overline{x:\sigma}$ is a list of parameter declarations. If values $\overline{v}$ have types $\overline{\sigma}$, then $c[\overline{v}]$ can be used as the name of a reactor. Variables $\overline{y}$ and $\overline{z}$ may be used in statement s. Variables $\overline{z}$ are initialized by `setvar` messages synthesized by $\overline{\pi}$. In DSR, a message synthesizer π is either a quorum read synthesizer $\texttt{QR}[\mathcal{Q}, l]$ or a label threshold synthesizer $\texttt{LT}[l]$. For simplicity, empty-list components may be omitted from a reactor declaration.

A value v may be a variable x, an integer n, a context identifier η, a memory reference m, a reactor $c[\overline{v}]$, a remote variable $\langle c[\overline{v}],\ v\rangle.z$, a versioned value $v@t$, or a label ℓ. Expressions and statements are standard except for the three reactor operations `exec`, `chmod` and `setvar`. In statement $\texttt{exec}(v_1,\ v_2,\ pc,\ \mathcal{Q},\ \overline{e})$ or $\texttt{chmod}(v_1,\ v_2,\ pc,\ \mathcal{Q},\ \ell)$, value v_1 is either a reactor value $c[\overline{v}]$ or a variable, and v_2 is either η or a variable. In statement $\texttt{setvar}(v,\ e)$, value v is either $\langle c[\overline{v}],\ \eta\rangle.z$ or a variable.

Base labels	l	$\in$	$\mathcal{L}$
Labels	ℓ, pc	$::=$	$\{C = l_1,\ I = l_2,\ A = l_3\} \mid x$
Timestamps	t	$::=$	$\langle \overline{pc\,{:}\,n}\,;\,\overline{n_\delta}\rangle$
Values	v	$::=$	$x \mid n \mid \eta \mid m \mid c[\overline{v}] \mid \langle c[\overline{v}],\ v\rangle.z \mid v@t \mid \ell$
Expressions	e	$::=$	$v \mid !e \mid e_1 + e_2$
Statements	s	$::=$	$\texttt{skip} \mid v := e \mid s_1 ; s_2 \mid \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2$
		$\mid$	$\texttt{exec}(v_1,\ v_2,\ pc,\ \mathcal{Q},\ \overline{e}) \mid \texttt{chmod}(v_1,\ v_2,\ pc,\ \mathcal{Q},\ \ell)$
		$\mid$	$\texttt{setvar}(v,\ e)$
Reactor decls	r	$::=$	$c[\overline{x\,{:}\,\sigma}]\{pc,\ \mathcal{Q},\ \overline{\pi \triangleright z\,{:}\,\tau},\ \lambda\overline{y\,{:}\,\tau}.s\}$
Synthesizers	π	$::=$	$\texttt{QR}[\mathcal{Q}, l] \mid \texttt{LT}[l]$
Base types	β	$::=$	$\texttt{int} \mid \texttt{label} \mid \tau\ \texttt{ref} \mid \tau\ \texttt{var}$
		$\mid$	$\texttt{reactor}[\overline{x\,{:}\,\sigma}]\{pc,\ \overline{\pi \triangleright z\,{:}\,\tau_1},\ \overline{\tau_2}\}$
		$\mid$	$\texttt{reactor}[\overline{x\,{:}\,\sigma}]\{pc,\ \overline{\tau_2}\}$
Security types	σ	$::=$	β_ℓ
Types	τ	$::=$	$\sigma \mid \sigma@\mathcal{Q} \mid \texttt{stmt}_\ell$
Host sets	H, W	$::=$	$\{h_1, \ldots, h_n\}$
Quorum systems	$\mathcal{Q}$	$::=$	$\langle H, \overline{W}\rangle \mid h \mid H \mid \&v \mid \#v$
Programs	P	$::=$	$\{r_1, \ldots, r_n\}$

Figure 5.1: Syntax of the DSR language

A base type β can be `int` (integer), `label` (security label), τ `ref` (reference of type τ), τ `var` (remote variable of type τ) and reactor type $\texttt{reactor}[\overline{x\,{:}\,\sigma}]\{pc,\ \overline{\pi \triangleright z\,{:}\,\tau_1},\ \overline{\tau_2}\}$ whose components are interpreted the same way as in a reactor declaration. A reactor type may also have a simplified form $\texttt{reactor}[\overline{x\,{:}\,\sigma}]\{pc,\ \overline{\tau_2}\}$, which contains sufficient typing information for checking the invocation, while providing polymorphism over the arguments $\overline{z}$.

A security type σ is a base type β annotated with security label ℓ. Like security labels, replication schemes are also specified as type annotations. A located type $\sigma@\mathcal{Q}$ indicates that data with this type is replicated on the quorum system $\mathcal{Q}$. In general, a quorum system $\mathcal{Q}$ has the form $\langle H, \overline{W}\rangle$. If $\mathcal{Q}$ is $\langle H, \epsilon\rangle$ or $\langle \{h\}, \epsilon\rangle$, then $\mathcal{Q}$ also has a simplified form: H or h. In addition, if v is a memory reference replicated on $\mathcal{Q}$, then $\&v$ represents $\mathcal{Q}$, and $\#v$ represents $H = |\mathcal{Q}|$, which is the set of hosts in $\mathcal{Q}$. The type of a statement s has the form $\texttt{stmt}_\ell$, which means that after s terminates, the program counter label is lower bounded by ℓ with respect to the ordering $\sqsubseteq$.

A timestamp t has the form $\langle \overline{pc\,{:}\,n}\,;\,\overline{n_\delta}\rangle$, where the list $\overline{n}$ of integers is the local part of t. In DSR, a program P is simply a set of reactor declarations.

5.2 Operational semantics

In DSR, a system configuration incorporates the program states of all the hosts in the system. A system configuration is a tuple $\langle\Theta,\, \mathcal{M},\, \mathcal{E}\rangle$ where Θ is a thread pool, $\mathcal{M}$ is a global memory, and $\mathcal{E}$ is a system environment that captures system state other than memory, including messages and closures.

- The thread pool Θ is a set of threads. A thread θ is a tuple $\langle s,\, t,\, h,\, c[\overline{v}],\, \eta\rangle$ where s, t and h are the code, timestamp, and location of the thread, respectively; $\langle c[\overline{v}],\, \eta\rangle$ identifies the closure of this thread.
- The global memory $\mathcal{M}$ maps hosts to their local memories, and a local memory maps references to lists of versioned values. Thus, $\mathcal{M}[h][m] = v_1@t_1, \ldots, v_n@t_n$ means that $v_1@t_1, \ldots, v_n@t_n$ are the versions of m on host h. If $\mathcal{M}[h][m] = \texttt{none}@t$, then the value of m is unavailable on host h. Unlike in Aimp, memory failures are modeled by host failures instead of mapping references to `void`.
- The environment $\mathcal{E}$ is a tuple $\langle MT,\, CT\rangle$ where MT is a *message table* mapping a host pair $\langle h_s, h_r\rangle$ to the set of messages from h_s to h_r, and CT is a *closure table* mapping a tuple $\langle h, c[\overline{v}], \eta\rangle$ to the closure $\langle c[\overline{v}],\, \eta\rangle$ on h.

To read and update various program states in a system configuration, the evaluation rules of DSR uses the following notations:

- $\mathcal{M}[h, m, t]$: the value of m on host h at time t. If $v@t \in \mathcal{M}[h][m]$, then $\mathcal{M}[h, m, t] = v$. Otherwise, $\mathcal{M}[h, m, t]$ is not defined.
- $\mathcal{M}(h, t)$: a snapshot of $\mathcal{M}$ on host h at time t. Suppose $\mathcal{M}(h, t) = M$. Then M maps references to versioned values, and $M[m]$ is the most recent version of m on host h by the time t.

(E1) $$\frac{M(m) = v}{\langle !m,\ M\rangle \Downarrow v}$$ (E2) $$\frac{\langle e_i,\ M\rangle \Downarrow v_i \quad i \in \{1,2\} \quad v = v_1 \oplus v_2}{\langle e_1 + e_2,\ M\rangle \Downarrow v}$$ (E3) $$\langle v,\ M\rangle \Downarrow v$$

(S1) $$\frac{\langle e,\ M\rangle \Downarrow n \qquad M' = M[m \mapsto n@t]}{\langle m := e,\ M,\ \Omega,\ t\rangle \longmapsto \langle \texttt{skip},\ M',\ \Omega,\ t+1\rangle}$$ (S2) $$\frac{\langle s_1,\ M,\ \Omega,\ t\rangle \longmapsto \langle s_1',\ M',\ \Omega',\ t'\rangle}{\langle s_1; s_2,\ M,\ \Omega,\ t\rangle \longmapsto \langle s_1'; s_2,\ M',\ \Omega',\ t'\rangle}$$

(S3) $$\langle \texttt{skip}; s,\ M,\ \Omega,\ t\rangle \longmapsto \langle s,\ M,\ \Omega,\ t+1\rangle$$ (S4) $$\langle \texttt{fi}; s,\ M,\ \Omega,\ t\rangle \longmapsto \langle s,\ M,\ \Omega,\ t \triangleright 1\rangle$$

(S5) $$\frac{\langle e,\ M\rangle \Downarrow n \qquad n > 0}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2,\ M,\ \Omega,\ t\rangle \longmapsto \langle s_1; \texttt{fi},\ M,\ \Omega,\ t \triangleleft 1\rangle}$$

(S6) $$\frac{\langle e,\ M\rangle \Downarrow n \qquad n \le 0}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2,\ M,\ \Omega,\ t\rangle \longmapsto \langle s_2; \texttt{fi},\ M,\ \Omega,\ t \triangleleft 1\rangle}$$

(S7) $$\frac{\langle \overline{e},\ M\rangle \Downarrow \overline{v_1} \qquad \texttt{none} \notin \overline{v_1}}{\langle \texttt{exec}(c[\overline{v}],\ \eta,\ pc,\ \mathcal{Q},\ \overline{e}),\ M,\ \Omega,\ t\rangle \longmapsto \langle \texttt{halt},\ M,\ \Omega \cup [\texttt{exec}\ \langle c[\overline{v}],\ \eta\rangle :: pc, \overline{v_1}, \mathcal{Q}, t],\ t+1\rangle}$$

(S8) $$\langle \texttt{chmod}(c[\overline{v}],\ \eta,\ pc,\ \mathcal{Q},\ \ell),\ M,\ \Omega,\ t\rangle \longmapsto \langle \texttt{skip},\ M,\ \Omega \cup [\texttt{chmod}\ \langle c[\overline{v}],\ \eta\rangle :: pc, \ell, \mathcal{Q}, t],\ t+1\rangle$$

(S9) $$\frac{\langle e,\ M\rangle \Downarrow v \qquad v \ne \texttt{none}}{\langle \texttt{setvar}(\langle c[\overline{v}],\ \eta\rangle.z,\ e),\ M,\ \Omega,\ t\rangle \longmapsto \langle \texttt{skip},\ M,\ \Omega \cup [\texttt{setvar}\ \langle c[\overline{v}],\ \eta\rangle.z :: v, t],\ t+1\rangle}$$

(G1) $$\frac{\begin{gathered}\langle s,\ M,\ \Omega,\ t\rangle \longmapsto \langle s',\ M',\ \Omega',\ t'\rangle \qquad \mathcal{M}(h,t) = M \\ \mathcal{E}' = (\mathit{if}\ \Omega' = \Omega \cup \{\mu\}\ \mathit{then}\ \mathcal{E}[\mathit{messages}(h) \mapsto_+ \mu]\ \mathit{else}\ \mathcal{E})\end{gathered}}{\langle \{\langle s,\ t,\ h,\ c[\overline{v}],\ \eta\rangle\} \cup \Theta,\ \mathcal{M},\ \mathcal{E}\rangle \longmapsto \langle \{\langle s',\ t',\ h,\ c[\overline{v}],\ \eta\rangle\} \cup \Theta,\ \mathcal{M}[h \mapsto_t M'],\ \mathcal{E}'\rangle}$$

(M1) $$\frac{\begin{gathered}\mathcal{E}.\mathit{closure}(h, c[\overline{v}], \eta) = \langle c[\overline{v}], \eta, \ell, \mathcal{A}, t', \texttt{on}\rangle \quad P(c[\overline{v}]) = c[\overline{v}]\{pc',\ \mathcal{Q}',\ \overline{\pi \triangleright z : \tau_2},\ \lambda\overline{y : \tau_1}.s\} \\ \forall z_i.\mathcal{A}(z_i) \ne \texttt{none} \quad \mathcal{E}.\mathit{messages}(*, h, [\texttt{exec}\ \langle c[\overline{v}],\ \eta\rangle :: *]) = \langle H, h, \overline{\mu}\rangle \\ \texttt{LT}[\ell](H, \overline{\mu}) = [\texttt{exec}\ \langle c[\overline{v}],\ \eta\rangle :: pc, \overline{v_1}, \mathcal{Q}, t] \qquad \exists W \in \mathcal{Q}.\ W \subseteq H \qquad pc \sqsubseteq \ell \qquad \Gamma \vdash \overline{v_1} : \overline{\tau_1} \\ t'' = (\mathit{if}\ pc \sqsubseteq pc'\ \mathit{then}\ \mathit{inc}(t,\ pc')\ \mathit{else}\ t') \qquad t' \ne \texttt{none} \Rightarrow t \le t' \\ \mathcal{E}' = \mathcal{E}[\mathit{closure}(h, c[\overline{v}], \eta) \mapsto \langle c[\overline{v}], \eta, \ell, \mathcal{A}, t'', \texttt{off}\rangle] \qquad \mathcal{A}' = \mathcal{A}[\overline{y} \mapsto \overline{v_1}][\texttt{cid} \mapsto \eta][\texttt{nid} \mapsto \mathit{hash}(t'')]\end{gathered}}{\langle \Theta,\ \mathcal{M},\ \mathcal{E}\rangle \longmapsto \langle \Theta \cup \{\langle s[\mathcal{A}'],\ t'',\ h,\ c[\overline{v}],\ \eta\rangle\},\ \mathcal{M},\ \mathcal{E}'\rangle}$$

(M2) $$\frac{\begin{gathered}\mathcal{E}.\mathit{closure}(h, c[\overline{v}], \eta) = \langle c[\overline{v}], \eta, \ell, \mathcal{A}, t', \texttt{on}\rangle \\ \mathcal{E}.\mathit{messages}(*, h, [\texttt{chmod}\ \langle c[\overline{v}],\ \eta\rangle :: x, y, *], x \sqsubseteq \ell \sqsubseteq y, \ell \ne y) = \langle H, h, \overline{\mu}\rangle \qquad \exists W \in \mathcal{Q}.\ W \subseteq H \\ \texttt{LT}[\ell](H, \overline{\mu}) = [\texttt{chmod}\ \langle c[\overline{v}],\ \eta\rangle :: pc, \ell', \mathcal{Q}, t] \qquad t'' = (\mathit{if}\ pc \sqsubseteq pc'\ \mathit{then}\ \mathit{inc}(t,\ \ell)\ \mathit{else}\ t')\end{gathered}}{\langle \Theta,\ \mathcal{M},\ \mathcal{E}\rangle \longmapsto \langle \Theta,\ \mathcal{M},\ \mathcal{E}[\mathit{closure}(h, c[\overline{v}], \eta) \mapsto \langle c[\overline{v}], \eta, \ell', \mathcal{A}, t'', \texttt{on}\rangle]\rangle}$$

(M3) $$\frac{\begin{gathered}\mathcal{E}.\mathit{closure}(h, c[\overline{v}], \eta) = \langle c[\overline{v}], \eta, \ell, \mathcal{A}, t', \texttt{on}\rangle \qquad P(c[\overline{v}]) = c[\overline{v}]\{pc',\ H',\ \overline{\pi \triangleright z : \tau},\ \lambda\overline{y : \tau_1}.s\} \\ \mathcal{A}(z_i) = \texttt{none} \qquad \mathcal{E}.\mathit{messages}(*, h, [\texttt{setvar}\ \langle c[\overline{v}],\ \eta\rangle.z_i :: *]) = \langle H, h, \overline{\mu}\rangle \\ \pi_i(H, \overline{\mu}) = [\texttt{setvar}\ \langle c[\overline{v}],\ z_i\rangle.\eta :: v, t] \quad \Gamma \vdash v : \tau_i[\overline{v}/\overline{x}]\end{gathered}}{\langle \Theta,\ \mathcal{M},\ \mathcal{E}\rangle \longmapsto \langle \Theta,\ \mathcal{M},\ \mathcal{E}[\mathit{closure}(h, c[\overline{v}], \eta) \mapsto \langle c[\overline{v}], \eta, \ell, \mathcal{A}[z_i \mapsto v], t', \texttt{on}\rangle]\rangle}$$

(A1) $$\frac{\begin{gathered}I(h) \le l_{\texttt{A}} \quad \mathcal{M}(h,t) = M \quad \Gamma(m) = \sigma\ \texttt{or}\ \sigma@\mathcal{Q} \\ M' = M[m \mapsto v@t] \quad \Gamma \vdash v : \sigma\end{gathered}}{\langle \Theta,\ \mathcal{M},\ \mathcal{E}\rangle \longmapsto \langle \Theta,\ \mathcal{M}[h \mapsto_t M'],\ \mathcal{E}\rangle}$$ (A2) $$\frac{\begin{gathered}I(h) \le l_{\texttt{A}} \quad \Gamma \vdash \mu \\ \mathcal{E}' = \mathcal{E}[\mathit{messages}(h, h') \mapsto_+ \mu]\end{gathered}}{\langle \Theta,\ \mathcal{M},\ \mathcal{E}\rangle \longmapsto \langle \Theta,\ \mathcal{M},\ \mathcal{E}'\rangle}$$

(A3) $$\frac{A(h) \le l_{\texttt{A}}}{\langle \{\langle s,\ t,\ h,\ c[\overline{v}],\ \eta\rangle\} \cup \Theta,\ \mathcal{M},\ \mathcal{E}\rangle \longmapsto \langle \{\langle \texttt{abort},\ t,\ h,\ c[\overline{v}],\ \eta\rangle\} \cup \Theta,\ \mathcal{M},\ \mathcal{E}\rangle}$$

Figure 5.2: Operational semantics of DSR with respect to Γ and P

- $\mathcal{M}(h,m)$: the most recent value of m on host h. If $\mathcal{M}(h,m) = v$, then $v@t \in \mathcal{M}[h][m]$, and for any $v'@t' \in \mathcal{M}[h][m]$, $t' \leq t$.
- $\mathcal{E}[\mathit{messages}(h) \mapsto_+ \mu]$: the environment obtained by adding to $\mathcal{E}$ the message μ sent by h. Suppose $\mathcal{E}[\mathit{messages}(h) \mapsto_+ \mu] = \mathcal{E}'$. Then $\mathcal{E}'.MT[h,h'] = \mathcal{E}.MT[h,h'] \cup \{\mu\}$ for any $h' \in \mathit{receivers}(\mu)$, and for any other host pair h_1, h_2, $\mathcal{E}'.MT[h_1,h_2] = \mathcal{E}.MT[h_1,h_2]$. Suppose μ is for reactor c replicated on $\mathcal{Q}$. Then $\mathit{receivers}(\mu) = |\mathcal{Q}|$.
- $\mathcal{E}[\mathit{messages}(h_1,h_2) \mapsto_+ \mu]$: the environment obtained by adding μ to $\mathcal{E}$ as a message sent from h_1 to h_2.
- $\mathcal{M}[h \mapsto_t M]$: the memory obtained by incorporating into $\mathcal{M}$ the memory snapshot M on host h at time t. Suppose $\mathcal{M}[h \mapsto_t M] = \mathcal{M}'$. Then $M[m] = v@t$ implies that $\mathcal{M}'[h,m,t] = v$, and for any host h', time t' and reference m', $h' \neq h$ or $t' \neq t$ or $M[m'] \neq v@t$ implies $\mathcal{M}'[h',m',t'] = \mathcal{M}[h',m',t']$.
- $\mathcal{E}[\mathit{closure}(h, c[\overline{v}], \eta) \mapsto k]$: the environment obtained by mapping $\langle h, c[\overline{v}], \eta\rangle$ to closure k in the closure table of $\mathcal{E}$.

The operational semantics of DSR is given in Figure 5.2. The evaluation of a term may need to use the reactor declarations (the program text P) and the typing assignment Γ of memory, which maps references to types. For brevity, Γ and P are implicitly used by the evaluation rules in Figure 5.2, though they are technically an (unchanging) part of the evaluation relation defined by the operational semantics. In addition, three auxiliary statements may appear during execution, although they cannot appear in programs. They are `halt`, indicating the normal termination of a thread, `abort`, indicating an availability failure, and `fi`, indicating the end of the execution of a conditional statement.

Rules (E1)–(E3) are used to evaluate expressions on a single host. The notation $\langle e,\, M\rangle \Downarrow v$ means that evaluating e in a local memory snapshot M results in the value

v. These rules are standard. In (E1), the notation $M(m)$ represents the value of m in M. If $M[m] = v@t$, then $M(m)$ is computed as follows:

$$M(m) = \begin{cases} v@t & \text{if } \Gamma(m) = \sigma@\mathcal{Q} \\ v & \text{if } \Gamma(m) = \sigma \end{cases}$$

In rule (E2), $v_1 \oplus v_2$ is computed as follows:

$$v_1 \oplus v_2 = \begin{cases} n_1 + n_2 & \text{if } v_1 = n_1 \text{ and } v_2 = n_2 \\ \texttt{none} & \text{if } v_1 = \texttt{none} \text{ or } v_2 = \texttt{none} \end{cases}$$

Rules (S1) through (S9) are used to execute statements on a single host, defining a local evaluation relation $\langle s,\, M,\, \Omega,\, t\rangle \longmapsto \langle s',\, M',\, \Omega',\, t'\rangle$, where the output Ω keeps track of outgoing messages from the thread of s.

Rules (S1)–(S6) are largely standard. The interesting part is the manipulation of timestamps. Each evaluation step increments the local part of the timestamp t, which is a list of integer components. To avoid covert implicit flows, executing a conditional statement should eventually cause the timestamp to be incremented exactly once no matter which branch is taken. When entering a branch, in (S5) and (S6), a new integer component is appended to the local part of t; when exiting a branch in (S4), the last component is discarded. Given $t = \langle \overline{pc : n}\,;\, n'_1, \ldots, n'_k\rangle$, the following auxiliary functions manipulate local parts of timestamps:

$$\begin{aligned} t + 1 &= \langle \overline{pc : n}\,;\, n'_1, \ldots, n'_k + 1\rangle \\ t \lhd 1 &= \langle \overline{pc : n}\,;\, n'_1, \ldots, n'_k, 1\rangle \\ t \rhd 1 &= \langle \overline{pc : n}\,;\, n'_1, \ldots, n'_{k-1} + 1\rangle \end{aligned}$$

Rules (S7)–(S9) evaluate the three reactor operations. They all send out a network message encoding the corresponding command. In rule (S7), the `exec` statement produces the message $[c[\overline{v}], \eta, \texttt{exec} :: pc, \overline{v_1}, \mathcal{Q}, t]$, where $\mathcal{Q}$ is a quorum system of the current thread that potentially contains an unstable memory update. The destination hosts of this message are determined by $c[\overline{v}]$. After executing an `exec` statement, the current thread is terminated, evaluating to `halt`.

A global evaluation step is a transition $\langle\Theta,\, \mathcal{M},\, \mathcal{E}\rangle \longmapsto \langle\Theta',\, \mathcal{M}',\, \mathcal{E}'\rangle$. Rule (G1) defines global transitions by lifting local evaluation steps, using changes to the local memory and outgoing messages to update the system configuration.

Rule (M1) handles `exec` messages. This rule is applicable when host h receives `exec` messages that can be synthesized into a valid invocation request for closure $\langle c[\overline{v}],\, \eta\rangle$. The following auxiliary function retrieves the set of messages with some property from environment $\mathcal{E}$:

$$\mathcal{E}.\mathit{messages}(\tilde{h_s}, \tilde{h_r}, \tilde{\mu}, \mathcal{C}) = \langle\overline{h}, \overline{h'}, \overline{\mu}\rangle$$

where $\tilde{h_s}$ are $\tilde{h_r}$ are *host patterns* that may be some host h, or a wild card $*$ representing any host, or some variable `x`; $\tilde{\mu}$ is a message pattern, a message with some components replaced by $*$ or `x`; $\mathcal{C}$ is a set of constraints on the variables appearing in these patterns. The result $\langle\overline{h}, \overline{h'}, \overline{\mu}\rangle$ represents a list of $\langle h_i, h'_i, \mu_i\rangle$ tuples where h_i and h'_i are the sender and receiver of μ_i, and μ_i matches the pattern and satisfies $\mathcal{C}$. To abuse notation a bit, $\overline{h}$ can be represented by $H = \{h_1, \ldots, h_n\}$, or h_s if all the hosts in $\overline{h}$ are h_s. For example, in rule (M1), the function $\mathcal{E}.\mathit{messages}(*, h, [\texttt{exec}\ \langle c[\overline{v}],\, \eta\rangle\ ::\ *])$ returns all the messages in $\mathcal{E}$ that are sent to h and have the message head "$\texttt{exec}, \langle c[\overline{v}],\, \eta\rangle$". The result of the function is $\langle H, h, \overline{\mu}\rangle$, where $H = \{h_1, \ldots, h_n\}$, and each h_i sends μ_i to h. Then H and $\overline{\mu}$ can be fed to the message synthesizer $\texttt{LT}[\ell]$ (abbreviation for $\texttt{LT}[I(\ell)]$), where $\ell = \mathit{acl}(c[\overline{v}], \eta)$. This enforces the constraint $I(\ell) \leq I_{\sqcup}(H)$, ensuring that the set of sender hosts have sufficient integrity to invoke the closure. As discussed in Section 4.2, a reactor closure has the form $\langle c, \eta, \ell, \mathcal{A}, \overline{a}\rangle$. The extra attributes $\overline{a}$ include t, the initial timestamp of the thread generated by invoking the closure, and `state`, a flag for the invocation state, which could be either `on` (yet to be invoked on this host) or `off` (already invoked). Suppose $P(c)$ is the declaration of reactor c. Then $P(c[\overline{v}])$

represents $P(c)[\overline{v}/\overline{x}]$, where $\overline{x}$ are parameters of c. Once $\texttt{LT}[\ell]$ returns an invocation request $[\texttt{exec}\,\langle c[\overline{v}],\,\eta\rangle :: \mathit{pc}, \overline{v_1}, \mathcal{Q}, t]$, host h verifies the following constraints to ensure the validity of the request:

- $\forall z_i.\mathcal{A}(z_i) \neq \texttt{none}$. This constraint guarantees that variables $\overline{z}$ are all initialized.
- $\exists W \in \mathcal{Q}.W \subseteq H$. This constraint ensures that all memory updates of the sender thread are stable.
- $\mathit{pc} \sqsubseteq \ell$. This label constraint controls the implicit flows by ensuring the program point of the sender thread has sufficient integrity and does not reveal confidential information.
- $\Gamma \vdash \overline{v_1} : \overline{\tau_1}$. Run-time type checking ensures that the arguments of the request are well-typed. This check is necessary because bad hosts may send ill-typed messages.
- $t' \neq \texttt{none} \Rightarrow t \leq t'$. This constraint ensures that the invocation request is not out of order.

After the request is validated, host h creates a new thread whose code is $s[\mathcal{A}']$, the statement obtained by applying substitution $\mathcal{A}'$ to s. In particular, the current context identifier `cid` is replaced by η, and the new closure identifier `nid` is replaced by the hash of the current timestamp t'', which is either t', or $\mathit{inc}(t,\,\mathit{pc}')$ if $\mathit{pc} \sqsubseteq \mathit{pc}'$. The state of the closure is set to `off` to prevent more invocations.

Rule (M2) handles `chmod` messages. Suppose the `chmod` messages to be processed are for closure $\langle c[\overline{v}],\,\eta\rangle$. Like in (M1), the closure $\langle c[\overline{v}], \eta, \ell, \mathcal{A}, t', \texttt{on}\rangle$ is retrieved from $\mathcal{E}$; $\texttt{LT}[\ell]$ is used to synthesize the `chmod` messages that attempt to change the access control label of $\langle c[\overline{v}],\,\eta\rangle$ from ℓ to ℓ' such that $\ell \sqsubseteq \ell'$ and $\ell \neq \ell'$. The `chmod` messages are extracted from $\mathcal{E}$ by $\mathcal{E}.\mathit{messages}(*, h, [\texttt{chmod}\,\langle c[\overline{v}],\,\eta\rangle :: x, y, *], x \sqsubseteq \ell \sqsubseteq y, \ell \neq y)$, which produces only `chmod` messages that are meant to change $\mathit{acl}(c, \eta)$ to a label higher than ℓ. Once a message $[\texttt{chmod}\,\langle c[\overline{v}],\,\eta\rangle :: \mathit{pc}, \ell', \mathcal{Q}, t]$ is produced by $\texttt{LT}[\ell]$, rule (M2) verifies the quorum constraint and the label constraint $\mathit{pc} \sqsubseteq \ell$, just like rule (M1). Once the constraints are verified, the closure's timestamp is initialized if necessary, and the access control label of the closure is set to ℓ'.

Rule (M3) handles `setvar` messages. Suppose the corresponding request is to initialize variable z_i of the closure identified by $\langle c[\overline{v}],\,\eta\rangle$. Then π_i is the message synthesizer to use, according to the declaration of $c[\overline{v}]$. If π_i returns a `setvar` request with a well-typed initial value v, and z_i has not yet been initialized, then z_i is mapped to v in the variable record of the closure.

In a distributed system, attackers can launch active attacks using the hosts they control. Rules (A1) through (A3) simulate the effects of those attacks. In general, integrity attacks fall into two categories: modifying the memory of a bad host and sending messages from a bad host. Rules (A1) and (A2) correspond to these two kinds of attacks. The constraint $I(h) \leq l_{\texttt{A}}$ indicates that the attacker is able to compromise the integrity of host h. In rule (A1), an arbitrary memory reference m on host h is modified. Note that we assume the attack does not violate the well-typedness of the memory. This assumption does not limit the power of an attacker because the effects of an ill-typed memory would either cause the execution of a thread to get stuck—essentially an availability attack—or produce an ill-typed message, which a correct receiver would ignore. In rule (A2), an arbitrary message μ is sent from host h. Again, we assume that μ is well-typed without loss of generality. Rule (A3) simulates an availability attack by aborting a thread of a host h whose availability may be compromised by the attacker.

5.3 Type system

This section describes the type system of DSR, which is designed to control information flow in distributed programs.

(ST1) $\dfrac{\tau_1 \le \tau_2 \quad \tau_2 \le \tau_1}{\tau_1\ \texttt{ref} \le \tau_2\ \texttt{ref}}$ (ST2) $\dfrac{\tau_2 \le \tau_1}{\tau_1\ \texttt{var} \le \tau_2\ \texttt{var}}$

(ST3) $\dfrac{\overline{\tau_4} \le \overline{\tau_2} \qquad \overline{\tau_3} \le \overline{\tau_1} \qquad pc' \sqsubseteq pc}{\texttt{reactor}[\overline{x\,{:}\,\sigma}]\{pc,\ \overline{\pi \rhd z\,{:}\,\tau_2},\ \overline{\tau_1}\} \le \texttt{reactor}[\overline{x\,{:}\,\sigma}]\{pc',\ \overline{\pi \rhd z\,{:}\,\tau_4},\ \overline{\tau_3}\}}$

(ST4) $\texttt{reactor}[\overline{x\,{:}\,\sigma}]\{pc,\ \overline{\pi \rhd z\,{:}\,\tau_2},\ \overline{\tau_1}\} \le \texttt{reactor}[\overline{x\,{:}\,\sigma}]\{pc,\ \overline{\tau_1}\}$

(ST5) $\dfrac{\beta_1 \le \beta_2 \quad \ell_1 \sqsubseteq \ell_2}{\mathcal{C} \vdash (\beta_1)_{\ell_1} \le (\beta_2)_{\ell_2}}$ (ST6) $\dfrac{\sigma_1 \le \sigma_2}{\sigma_1 @ \mathcal{Q} \le \sigma_2 @ \mathcal{Q}}$ (ST7) $\dfrac{pc_1 \sqsubseteq pc_2}{\texttt{stmt}_{pc_1} \le \texttt{stmt}_{pc_2}}$

Figure 5.3: Subtyping rules

5.3.1 Subtyping

The subtyping relationship between security types plays an important role in enforcing information flow security. Given two security types $\tau_1 = \beta_{1\ell_1}$ and $\tau_2 = \beta_{2\ell_2}$, suppose τ_1 is a subtype of τ_2, written as $\tau_1 \le \tau_2$. Then any data of type τ_1 can be securely treated as data of type τ_2, and any data with label ℓ_1 may be treated as data with label ℓ_2, which requires $\ell_1 \sqsubseteq \ell_2$, that is, $C(\ell_1) \le C(\ell_2)$ and $I(\ell_2) \le I(\ell_1)$. In DSR, a label may be a variable, and a label variable x is incomparable with other labels except for x itself.

The subtyping rules are shown in Figure 5.3. Rules (ST1)–(ST4) are about subtyping on base types. These rules demonstrate the expected covariance or contravariance, as reactors are like functions, and remote variables are like final fields in Java [81]. As shown in rule (ST3), the argument types are contravariant, and the premise $pc_2 \sqsubseteq pc_1$ is needed because the *pc* of a reactor type is an upper bound on the *pc* of the caller. Rule (ST4) says that any reactor of type $\texttt{reactor}[\overline{x\,{:}\,\sigma}]\{pc,\ \overline{\pi \rhd z\,{:}\,\tau_2},\ \overline{\tau_1}\}$ can be treated as a reactor of type $\texttt{reactor}[\overline{x\,{:}\,\sigma}]\{pc, \overline{\tau_2}\}$. Intuitively, it is safe to associate a more restrictive program counter label with a program point, since it permits fewer implicit flows. Therefore, a statement of type $\texttt{stmt}_{pc_1}$ also has type $\texttt{stmt}_{pc_2}$ if $pc_1 \sqsubseteq pc_2$, as shown in (ST7).

5.3.2 Typing

The typing rules of DSR are shown in Figure 5.4. A program P is well-typed in Γ, written as $\Gamma \vdash P$, if every reactor declaration r in P is well-typed with respect to Γ and P, written $\Gamma\,;P \vdash r$, where Γ and P provides the typing information for memory and reactors, respectively.

A reactor declaration is well-typed if its body statement is well-typed. The typing judgment for a statement s has the form $\Gamma\,;P\,;\mathcal{Q}\,;\mathit{pc} \vdash s : \tau$, meaning that s has type τ under the typing environment $\Gamma\,;P\,;\mathcal{Q}\,;\mathit{pc}$, where $\mathcal{Q}$ is the quorum system where s is replicated, and *pc* is the program counter label. The typing judgment for an expression e has the form $\Gamma\,;P\,;\mathcal{Q} \vdash e : \tau$, meaning that e has type τ under the typing environment $\Gamma\,;P\,;\mathcal{Q}$. For simplicity, a component in the typing environment of a typing judgment may be omitted if the component is irrelevant. For example, in rule (INT), the type of n has nothing to do with the typing environment, and thus the typing judgment is simplified as $\vdash n : \texttt{int}_\ell$.

Rules (INT), (CID), (LABEL), (VAR), (LOC), (ADD), (ESUB), (IF) and (SUB) are standard for a security type system aimed to analyze information flows.

In DSR, only the `chmod` statement imposes a lower bound on the program counter label after termination. Thus, the types of `skip`, $v := e$, and the `exec` and `setvar` statements are the same: $\texttt{stmt}_\bot$, which effectively places no lower bound on the program counter label after termination, as $\bot \sqsubseteq \mathit{pc}$ holds for any *pc*.

Rule (REACTOR) is used to check reactor value $c[\overline{v}]$. The notations $\ell \sqsubseteq \sigma$ and $\ell \sqsubseteq \sigma@\mathcal{Q}$ represent $\ell \sqsubseteq \ell'$ if $\sigma = \beta_{\ell'}$. Suppose $c[\overline{x\!:\!\sigma}]\{\mathit{pc},\ \mathcal{Q},\ \overline{\pi \triangleright z\!:\!\tau_1},\ \lambda\,\overline{y\!:\!\tau_2}.\ s\}$ is the declaration of c in P. Then the list of parameters $\overline{v}$ must have types $\overline{\sigma}[\overline{v}/\overline{x}]$, where the substitution is necessary because $\overline{x}$ may appear in $\overline{\sigma}$. The values of the reactor parameters and the effects of this reactor depend on the reactor value itself. Thus, $\ell \sqsubseteq \overline{\sigma}[\overline{v}/\overline{x}]$ and $\ell \sqsubseteq \mathit{pc}[\overline{v}/\overline{x}]$ are enforced. Since this reactor is replicated on $\mathcal{Q}' = \mathcal{Q}[\overline{v}/\overline{x}]$,

$$\text{(INT)}\quad \vdash n : \mathtt{int}_\ell \qquad \text{(CID)}\quad \vdash \eta : \mathtt{int}_\ell \qquad \text{(LABEL)}\quad \vdash \{C = l_1,\ I = l_2,\ A = l_3\} : \mathtt{label}_\ell$$

$$\text{(VAR)}\quad \Gamma \vdash x : \Gamma(x) \qquad \text{(LOC)}\quad \frac{\Gamma(m) = \tau}{\Gamma \vdash m : (\tau\ \mathtt{ref})_\ell} \qquad \text{(ADD)}\quad \frac{\Gamma \vdash e_i : \mathtt{int}_{\ell_i} \quad i \in \{1,2\}}{\Gamma \vdash e_1 + e_2 : \mathtt{int}_{\ell_1 \sqcup \ell_2}}$$

$$\text{(REACTOR)}\quad \frac{\begin{array}{c} P(c) = c[\overline{x{:}\sigma}]\{\mathit{pc},\ \mathcal{Q},\ \overline{\pi \triangleright z{:}\tau_1},\ \lambda\overline{y{:}\tau_2}.\ s\} \\ \Gamma \vdash \overline{v} : \overline{\sigma}[\overline{v}/\overline{x}] \quad \ell \sqsubseteq \overline{\sigma}[\overline{v}/\overline{x}] \quad \ell \sqsubseteq \mathit{pc}[\overline{v}/\overline{x}] \\ C_\sqcup(\overline{\tau_1}[\overline{v}/\overline{x}]) \sqcup C_\sqcup(\overline{\tau_2}[\overline{v}/\overline{x}]) \sqcup C(\mathit{pc}[\overline{v}/\overline{x}]) \leq C_\sqcap(\mathcal{Q}[\overline{v}/\overline{x}]) \end{array}}{\Gamma\,;P \vdash c[\overline{v}] : \mathtt{reactor}[\overline{v}/\overline{x}]\{\mathit{pc},\ \overline{\pi \triangleright z{:}\tau_1},\ \overline{\tau_2}\}_\ell}$$

$$\text{(ARG)}\quad \frac{\begin{array}{c} \Gamma\,;P \vdash c[\overline{v}] : \mathtt{reactor}\{\mathit{pc},\ \overline{\pi \triangleright z{:}\tau},\ \overline{\tau_2}\}_\ell \\ \vdash v : \mathtt{int}_\ell \quad \mathit{FV}(\overline{v}) = \emptyset \quad \ell \sqsubseteq \tau_i \end{array}}{\Gamma\,;P \vdash \langle c[\overline{v}],\ v\rangle.z_i : (\pi_i \otimes \tau_i\ \mathtt{var})_\ell} \qquad \text{(TV)}\quad \frac{\Gamma \vdash v : \sigma}{\Gamma\,;\mathcal{Q} \vdash v@t : \sigma@\mathcal{Q}}$$

$$\text{(DEREF)}\quad \frac{\Gamma \vdash e : (\tau\ \mathtt{ref})_\ell \quad \mathit{readable}(\mathcal{Q},\tau)}{\Gamma\,;\mathcal{Q} \vdash !e : \tau \sqcup \ell} \qquad \text{(ESUB)}\quad \frac{\Gamma\,;P\,;\mathcal{Q} \vdash e : \tau_1 \quad \tau_1 \leq \tau_2}{\Gamma\,;P\,;\mathcal{Q} \vdash e : \tau_2}$$

$$\text{(SKIP)}\quad \Gamma\,;P\,;\mathcal{Q}\,;\mathit{pc} \vdash \mathtt{skip} : \mathtt{stmt}_\perp \qquad \text{(ASSI)}\quad \frac{\Gamma \vdash v : (\tau\ \mathtt{ref})_\ell \quad \mathit{writable}(\mathcal{Q},\tau)\Gamma \vdash e : \sigma \quad \mathit{base}(\tau) = \sigma \quad \mathit{pc} \sqcup \ell \sqsubseteq \sigma}{\Gamma\,;\mathcal{Q}\,;\mathit{pc} \vdash v := e : \mathtt{stmt}_\perp}$$

$$\text{(SEQ)}\quad \frac{\begin{array}{c} \Gamma\,;P\,;\mathcal{Q}\,;\mathit{pc} \vdash s_1 : \mathtt{stmt}_{\ell_1} \\ \Gamma\,;P\,;\mathcal{Q}\,;\mathit{pc} \sqcup \ell_1 \vdash s_2 : \mathtt{stmt}_{\ell_2} \end{array}}{\Gamma\,;P\,;\mathcal{Q}\,;\mathit{pc} \vdash s_1; s_2 : \mathtt{stmt}_{\ell_1 \sqcup \ell_2}} \qquad \text{(IF)}\quad \frac{\begin{array}{c} \Gamma\,;\mathcal{Q} \vdash e : \mathtt{int}_\ell \\ \Gamma\,;P\,;\mathcal{Q}\,;\mathit{pc} \sqcup \ell \vdash s_i : \tau \quad i \in \{1,2\} \end{array}}{\Gamma\,;P\,;\mathcal{Q}\,;\mathit{pc} \vdash \mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 : \tau}$$

$$\text{(EXEC)}\quad \frac{\begin{array}{c} \Gamma\,;P \vdash v_1 : \mathtt{reactor}\{\mathit{pc}',\ \overline{\pi \triangleright z{:}\tau},\ \overline{\tau_2}\}_\ell \\ \Gamma\,;\mathcal{Q} \vdash v_2 : \mathtt{int}_\ell \quad \Gamma \vdash \ell : \mathtt{label}_\ell \quad \Gamma\,;P\,;\mathcal{Q} \vdash \overline{e} : \overline{\tau_2} \\ \mathit{pc} \sqsubseteq \overline{\tau_2} \quad \mathit{pc} \sqsubseteq \ell \end{array}}{\Gamma\,;P\,;\mathcal{Q}\,;\mathit{pc} \vdash \mathtt{exec}(v_1,\ v_2,\ \ell,\ \mathcal{Q},\ \overline{e}) : \mathtt{stmt}_\perp}$$

$$\text{(CHMD)}\quad \frac{\begin{array}{c} \Gamma\,;P \vdash v_1 : \mathtt{reactor}\{\mathit{pc}',\ \overline{\pi \triangleright z{:}\tau},\ \overline{\tau_2}\}_\ell \\ \Gamma\,;\mathcal{Q} \vdash v_2 : \mathtt{int}_\ell \quad \Gamma \vdash \ell : \mathtt{label}_\ell \\ \Gamma \vdash \ell' : \mathtt{label}_\ell \quad \mathit{pc} \sqsubseteq \ell \quad \ell \sqsubseteq \ell' \end{array}}{\Gamma\,;P\,;\mathcal{Q}\,;\mathit{pc} \vdash \mathtt{chmod}(v_1,\ v_2,\ \ell,\ \mathcal{Q},\ \ell') : \mathtt{stmt}_{\ell'}} \qquad \text{(SETV)}\quad \frac{\begin{array}{c} \Gamma\,;\mathcal{Q} \vdash v : (\tau\ \mathtt{var})_\ell \quad \Gamma\,;\mathcal{Q} \vdash e : \tau \\ \mathit{pc} \sqcup \ell \sqsubseteq \tau \end{array}}{\Gamma\,;P\,;\mathcal{Q}\,;\mathit{pc} \vdash \mathtt{setvar}(v,\ e) : \mathtt{stmt}_\perp}$$

$$\text{(RD)}\quad \frac{\Gamma, \overline{x{:}\sigma}, \overline{y{:}\tau_1}, \overline{z{:}\tau_2}, \mathtt{cid}{:}\mathtt{int}_{\mathit{pc}}, \mathtt{nid}{:}\mathtt{int}_{\mathit{pc}}\,; P\,;\mathcal{Q}\,;\mathit{pc} \vdash s : \mathtt{stmt}_{\mathit{pc}'}}{\Gamma\,;P \vdash c[\overline{\sigma\ x}]\{\mathit{pc},\ \mathcal{Q},\ \overline{\pi \triangleright \tau\ z},\ \lambda\overline{\tau\ y}.\ s\}}$$

$$\text{(SUB)}\quad \frac{\Gamma\,;P\,;\mathcal{Q}\,;\mathit{pc} \vdash s : \tau_1 \quad \tau_1 \leq \tau_2}{\Gamma\,;P\,;\mathcal{Q}\,;\mathit{pc} \vdash s : \tau_2}$$

[Auxiliary notations]

$\pi \otimes \tau :\quad \mathtt{QR}[\mathcal{Q}] \otimes \sigma = \sigma@\mathcal{Q} \qquad \mathtt{LT}[I] \otimes \tau = \tau$

$\mathit{writable}(\mathcal{Q},\tau) :\quad \tau = \sigma@\mathcal{Q}\ \vee\ (\tau = \sigma \wedge |\mathcal{Q}| = \{h\})$

$\mathit{readable}(\mathcal{Q},\tau) :\quad (\tau = \sigma@\mathcal{Q}' \wedge |\mathcal{Q}| = |\mathcal{Q}'|)\ \vee\ (\tau = \sigma \wedge |\mathcal{Q}| = \{h\})$

Figure 5.4: Typing rules of DSR

any data processed by the reactor is observable to the hosts in $\mathcal{Q}'$. The last constraint ensures that the hosts in $\mathcal{Q}'$ would not leak information about $c[\overline{v}]$.

Rule (ARG) checks remote variable $\langle c[\overline{v}],\, v\rangle.z_i$. If the type of $c[\overline{v}]$ shows that z_i has type τ_i and synthesizer π_i, then the values used to initialize z_i have type $\pi_i \otimes \tau_i$ such that they can be synthesized by π_i into a value with type τ_i. Therefore, the type of $\langle c[\overline{v}],\, v\rangle.z_i$ is $(\pi_i \otimes \tau_i\ \texttt{var})_\ell$ where ℓ is the label of $c[\overline{v}]$.

Rule (TV) checks versioned values. If v has type σ, then $v@t$ has type $\sigma@\mathcal{Q}$ in the typing environment $\Gamma\,;\mathcal{Q}$, which indicates that the versioned value is evaluated at $\mathcal{Q}$.

Rules (DEREF) and (ASSI) checks memory dereferences and assignments. These two rules need to ensure that the involved memory reference is accessible on $\mathcal{Q}$. Intuitively, if a memory reference m is replicated on $\mathcal{Q}$, then a read or write operation needs to be performed on all the hosts in $\mathcal{Q}$. In rule (DEREF), the premise *readable*$(\mathcal{Q}, \tau)$ ensures that $\mathcal{Q}$ contains the same set of hosts as $\mathcal{Q}'$ where the reference of type $\tau\ \texttt{ref}$ is replicated, or the reference is not replicated at all. In rule (ASSI), the premise *writable*$(\mathcal{Q}, \tau)$ ensures that $\mathcal{Q}$ is the quorum system where the reference to be assigned is replicated.

In rule (DEREF), if e has type $(\tau\ \texttt{ref})_\ell$, then $!e$ has type $\tau \sqcup \ell$. We use the notation $\beta_\ell \sqcup \ell'$ to represent $\beta_{\ell\sqcup\ell'}$, and $\sigma@\mathcal{Q} \sqcup \ell$ to represent $\sigma \sqcup \ell@\mathcal{Q}$. The label ℓ is folded into the type of $!e$ because the result of $!e$ depends on the value of e.

Rule (ASSI) says that $v := e$ is well-typed if v has type $(\tau\ \texttt{ref})_\ell$, and e has type $\sigma = \mathit{base}(\tau)$, which strips the location part of τ. The constraint $\mathit{pc} \sqcup \ell \sqsubseteq \sigma$ ensures the safety of both the explicit information flow from e to reference v and the implicit flow from the program counter to v.

Rule (SEQ) checks sequential statement $s_1; s_2$. If s_1 has type $\texttt{stmt}_{\ell_1}$, then s_2 is checked with $\mathit{pc} \sqcup \ell_1$, since ℓ_1 is a lower bound to the program counter label after s_1 terminates. If the type of s_2 is $\texttt{stmt}_{\ell_2}$, then the type of $s_1; s_2$ is $\texttt{stmt}_{\ell_1\sqcup\ell_2}$, as both ℓ_1 and ℓ_2 are a lower bound to the program counter label after $s_1; s_2$ terminates.

Rule (EXEC) checks `exec` statements. It resembles checking a function call. The constraints $pc \sqsubseteq \overline{\tau_2}$ and $pc \sqsubseteq \ell$ ensure that the reactor to be invoked would not leak the information about the current program counter.

Rule (CHMD) checks `chmod` statements. The label ℓ' is meant to be the new access control label of closure $\langle v_1,\, v_2\rangle$. After executing this statement, the program counter label is lower bounded by ℓ', effectively preventing the following code from running another `chmod` statement with label ℓ before $\langle v_1,\, v_2\rangle$ is invoked. The constraint $\ell \sqsubseteq \ell'$ implies $pc \sqsubseteq \ell'$, ensuring the new program counter label is as restrictive as the current one.

Rule (SETV) is used to check the `setvar` statement. Value v has type $(\tau\ \texttt{var})_\ell$, representing a remote variable. The value of expression e is used to initialize the remote variable, and thus e has type τ. The constraint $pc \sqcup \ell \sqsubseteq \tau$ is imposed because v and the program counter may affect the value of the remote variable.

Rule (RD) checks reactor declarations: $c[\overline{x\!:\!\sigma}]\{pc,\, \mathcal{Q},\, \overline{\pi \triangleright z\!:\!\tau_2},\, \lambda\overline{y\!:\!\tau_1}.s\}$ is well-typed with respect to Γ and P as long as the reactor body s is well-typed in the typing environment $\Gamma, \overline{x\!:\!\sigma}, \overline{y\!:\!\tau_1}, \overline{z\!:\!\tau_2}\,;P\,;\mathcal{Q}\,;pc$.

5.3.3 Subject reduction

The type system of DSR satisfies the subject reduction property, which is stated in the subject reduction theorem, following the definitions of well-typed memories and configurations.

Definition 5.3.1 (Well-typed memory). $\mathcal{M}$ is well-typed in Γ, written $\Gamma \vdash \mathcal{M}$, if for any m in $dom(\Gamma)$ and any host h and any timestamp t, $\mathcal{M}[h, m, t] = v$ and $\Gamma(m) = \sigma$ `or` $\sigma@\mathcal{Q}$ imply $\Gamma \vdash v : \sigma$.

Definition 5.3.2 (Well-typed memory snapshot). M is well-typed in Γ, written $\Gamma \vdash M$, if for any m in $dom(\Gamma)$, $\vdash M(m) : \Gamma(m)$.

Definition 5.3.3 (Well-typed environment). $\mathcal{E}$ is well-typed in Γ and P, written $\Gamma\,;P \vdash \mathcal{E}$, if for any closure $\langle c[\overline{v}], \eta, \ell, t, \mathcal{A}, *\rangle$ in $\mathcal{E}$ and any $x \in$ *dom*$(\mathcal{A})$, $\vdash \mathcal{A}(x) : \tau$ where τ is the type of x based on Γ and $c[\overline{v}]$, and for any μ in $\mathcal{E}$, we have $\Gamma\,;P \vdash \mu$, which means the contents of μ are well-typed. The inference rules for $\Gamma\,;P \vdash \mu$ are standard:

$$\text{(M-EXEC)}\quad \frac{\Gamma\,;P \vdash c[\overline{v}] : \texttt{reactor}\{pc', \overline{\pi \rhd z\!:\!\tau_1}, \overline{\tau_2}\} \qquad \vdash \overline{v_1} : \overline{\tau_1}}{\Gamma\,;P \vdash [\texttt{exec}\,\langle c[\overline{v}],\, \eta\rangle :: pc, \overline{v_1}, \mathcal{Q}, t]}$$

$$\text{(M-CHMD)}\quad \frac{\Gamma\,;P \vdash c[\overline{v}] : \texttt{reactor}\{pc', \overline{\pi \rhd z\!:\!\tau_1}, \overline{\tau_2}\}}{\Gamma\,;P \vdash [\texttt{chmod}\,\langle c[\overline{v}],\, \eta\rangle :: pc, \ell, \mathcal{Q}, t]}$$

$$\text{(M-SETV)}\quad \frac{\Gamma\,;P \vdash \langle c[\overline{v}],\, \eta\rangle.z : (\tau\ \texttt{var})_\ell \qquad \vdash v_1 : \tau}{\Gamma\,;P \vdash [\texttt{setvar}\,\langle c[\overline{v}],\, \eta\rangle.z :: v_1, t]}$$

Definition 5.3.4 (Well-typed configuration). $\langle \Theta,\, \mathcal{M},\, \mathcal{E}\rangle$ is well-typed in Γ and P, written $\Gamma\,;P \vdash \langle \Theta,\, \mathcal{M},\, \mathcal{E}\rangle$, if $\Gamma \vdash \mathcal{M}$, and $\Gamma\,;P \vdash \mathcal{E}$, and for any $\langle s,\, t,\, h,\, c[\overline{v}],\, \eta\rangle$ in Θ, $\Gamma\,;P\,;\mathcal{Q}\,;pc \vdash s : \tau$.

Lemma 5.3.1 (Expression subject reduction). Suppose $\Gamma \vdash \langle e,\, M\rangle \Downarrow v$, and $\Gamma\,;\mathcal{Q} \vdash e : \tau$, and $\Gamma \vdash M$. Then $\Gamma\,;\mathcal{Q} \vdash v : \tau$.

Proof. By induction on the derivation of $\langle e,\, M\rangle \Downarrow v$.

- Case (E1). In this case, e is $!m$, and τ is $\Gamma(m)$, and v is $M(m)$. If $\Gamma(m) = \texttt{int}_\ell$, then $M(m) = n$ while $M[m] = n@t$, and $\Gamma\,;\mathcal{Q} \vdash n : \texttt{int}_\ell$. Otherwise, $\Gamma(m) = \texttt{int}_\ell@\mathcal{Q}$, and $M(m) = M[m] = n@t$. We have $\Gamma\,;\mathcal{Q} \vdash n@t : \texttt{int}_\ell@\mathcal{Q}$.
- Case (E2). By induction, $\Gamma\,;\mathcal{Q} \vdash v_i : \texttt{int}_{\ell_i}$ for $i \in \{1,2\}$. Thus, $\Gamma\,;\mathcal{Q} \vdash v_1 + v_2 : \texttt{int}_{\ell_1 \sqcup \ell_2}$.

□

Lemma 5.3.2 (Substitution). Suppose $\Gamma \vdash v : \tau$. Then $x : \tau, \Gamma\,;P\,;\mathcal{Q} \vdash e : \tau'$ implies $\Gamma[v/x]\,;P\,;\mathcal{Q}[v/x] \vdash e[v/x] : \tau'[v/x]$, and $x : \tau, \Gamma\,;P\,;\mathcal{Q}\,;pc \vdash s : \tau'$ implies $\Gamma[v/x]\,;P\,;\mathcal{Q}[v/x]\,;pc[v/x] \vdash s[v/x] : \tau'[v/x]$.

Proof. By induction on the structure of e and s. Without loss of generality, assume that the typing derivations of e and s end with applying rule (ESUB) or (SUB).

- e is y. If $y = x$, then $e[v/x] = v$, and x does not appear in τ. Therefore, $\Gamma \vdash e[v/x] : \tau'[v/x]$ immediately follows $\Gamma \vdash v : \tau$. If $y \neq x$, then $e[v/x] = y$, and $y : \tau'[v/x]$ belongs to $\Gamma[v/x]$. Thus, $\Gamma[v/x] \vdash y : \tau'[v/x]$.
- e is n, η, ℓ, or m. This case is obvious.
- e is $c[\overline{v}]$, $\langle c[\overline{v},\, v\rangle.z$, $v@t$, $!e$ or $e_1 + e_2$. By induction.
- s is $v' := e$. By induction, $\Gamma[v/x] \vdash v'[v/x] : \tau''\ \texttt{ref}_\ell[v/x]$, and $\Gamma[v/x] \vdash e[v/x] : \sigma[v/x]$. Since *base*$(\tau'') = \sigma$, we have *base*$(\tau''[v/x]) = \sigma[v/x]$. In addition, *writable*$(\mathcal{Q}, \tau'')$ implies *writable*$(\mathcal{Q}[v/x], \tau''[v/x])$, and $pc \sqcup \ell \sqsubseteq \sigma$ implies $pc[v/x] \sqcup \ell[v/x] \sqsubseteq \sigma[v/x]$. Therefore, $\Gamma[v/x] \,;\, P \,;\, \mathcal{Q}[v/x] \vdash s[v/x] : \tau'[v/x]$.
- s is $s_1; s_2$ or $\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2$. By induction.

□

Lemma 5.3.3 (Subject reduction). Suppose $\langle s,\, M,\, \Omega,\, t\rangle \longmapsto \langle s',\, M',\, \Omega',\, t'\rangle$, and $\Gamma \vdash M$, and $\Gamma \,;\, P \vdash \Omega$, and $\Gamma \,;\, P \,;\, \mathcal{Q} \,;\, pc \vdash s : \tau$. Then $\Gamma \,;\, P \,;\, \mathcal{Q} \,;\, pc \vdash s' : \tau$ and $\Gamma \vdash M'$ and $\Gamma \,;\, P \vdash \Omega'$.

Proof. By induction on the derivation of $\langle s,\, M,\, \Omega,\, t\rangle \longmapsto \langle s',\, M',\, \Omega',\, t'\rangle$.

- Case (S1). By rule (ASSI), $\Gamma \,;\, \mathcal{Q} \vdash m : (\tau\ \texttt{ref})_\ell$ and $\Gamma \,;\, \mathcal{Q} \vdash e : \tau$. By Lemma 5.3.1, $\Gamma \,;\, \mathcal{Q} \vdash v : \tau$. Therefore, $\Gamma \vdash M[m \mapsto v@t]$.
- Case (S2). By induction.
- Case (S3). s is $\texttt{skip}; s'$. Since $\Gamma \,;\, P \,;\, \mathcal{Q} \,;\, pc \vdash \texttt{skip}; s' : \tau$, we have $\Gamma \,;\, P \,;\, \mathcal{Q} \,;\, pc \vdash \texttt{skip} : \texttt{stmt}_\ell$ and $\Gamma \,;\, P \,;\, \mathcal{Q} \,;\, pc \sqcup \ell \vdash s' : \tau$, which implies that $\Gamma \,;\, P \,;\, \mathcal{Q} \,;\, pc \vdash s' : \tau$.

- Case (S4). By the same argument as case (S3).
- Case (S5). s is $\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2$. By the typing rule (IF), $\Gamma\,;P\,;\mathcal{Q}\,;pc \sqcup \ell_e \vdash s_1 : \tau$, which implies $\Gamma\,;P\,;\mathcal{Q}\,;pc \vdash s_1 : \tau$.
- Case (S6). By the same argument as case (S5).
- Case (S7). Suppose $c[\overline{v}]$ has type $\texttt{reactor}\{pc', \mathcal{Q}', \overline{\pi \triangleright \tau z}, \overline{\tau_1}\}$. By Lemma 5.3.1, $\Gamma\,;\mathcal{Q} \vdash \overline{v_1} : \overline{\tau_1}$. By (M-EXEC), $\Gamma\,;P \vdash [\texttt{exec}\,\langle c[\overline{v}],\, \eta\rangle :: pc, \overline{v_1}, \mathcal{Q}, t]$, which implies $\Gamma\,;P \vdash \Omega'$.
- Case (S8). Since $c[\overline{v}]$ is well-typed, the $\texttt{chmod}$ message sent in this step is also well-typed.
- Case (S9). By Lemma 5.3.1, v_1 is of the correct type, and the $\texttt{setvar}$ message is well-typed.

□

Theorem 5.3.1 (Subject reduction). Suppose $\Gamma\,;P \vdash \langle\Theta,\, \mathcal{M},\, \mathcal{E}\rangle \longmapsto \langle\Theta',\, \mathcal{M}',\, \mathcal{E}'\rangle$ and $\Gamma\,;P \vdash \langle\Theta,\, \mathcal{M},\, \mathcal{E}\rangle$. Then $\Gamma\,;P \vdash \langle\Theta',\, \mathcal{M}',\, \mathcal{E}'\rangle$.

Proof. By induction on the derivation of $\langle\Theta,\, \mathcal{M},\, \mathcal{E}\rangle \longmapsto \langle\Theta',\, \mathcal{M}',\, \mathcal{E}'\rangle$.

- Case (G1). The evaluation step is derived from $\langle s,\, M,\, \Omega,\, t\rangle \longmapsto \langle s',\, M',\, \Omega',\, t'\rangle$ on host h, and $\mathcal{M}' = \mathcal{M}[h \mapsto_t M']$. Since M' and $\mathcal{M}$ are well-typed, $\mathcal{M}'$ is also well-typed. If $\Omega' = \Omega$, then $\mathcal{E}' = \mathcal{E}$ is well-typed. Otherwise, $\Omega' = \Omega \cup \{\mu\}$, and $\mathcal{E}' = \mathcal{E}[\mathit{messages}(h) \mapsto_+ \mu]$. Since μ is well-typed, $\mathcal{E}'$ is well-typed.
- Case (M1). In this case, we only need to prove that the newly created thread is well-typed. Since $\Gamma \vdash \overline{v_1} : \overline{\tau_1}$. By $\Gamma \vdash \overline{v_1} : \overline{\tau_1}[\overline{v}/\overline{x}]$, we have $\Gamma' \vdash \mathcal{A}'$. By Lemma 5.3.2, $\Gamma' \vdash s[\mathcal{A}'] : \tau'$.
- Case (M2). In this case, only the access control label of a closure is changed, which does not affect the well-typedness of the closure.

- Case (M3). In this case, we need to prove that $\mathcal{A}[z_i \mapsto v]$ is well-typed. By the run-time type checking in rule (M3), we have $\Gamma \vdash v : \tau_i[\overline{v}/\overline{x}]$. Furthermore, $\mathcal{A}$ is well-typed. Thus, $\mathcal{A}[z_i \mapsto v]$ is well-typed.
- Case (A1). By the premise $\Gamma \vdash v : \Gamma(m)$ in rule (A1).
- Case (A2). By the premise $\Gamma \vdash \mu$.
- Case (A3). The statement `abort` is considered well-typed.

$\square$

5.3.4 Preventing races

In DSR, a race is used to refer to the scenario that two threads with different closure identifiers are running at the same timestamp or sending messages with the same message head. A race makes it possible for attackers to choose to side with one of the two racing threads, and affect execution that the security policies do not allow them to affect. Furthermore, message races increase the complexity of maintaining consistency between reactor replicas. Therefore, it is desirable to prevent races in DSR programs.

According to the evaluation rule (S7) of DSR, a thread is terminated after sending out an `exec` message. As a result, if the execution of a distributed program starts from a single program point, then threads generated from normal execution can be serialized, and so can the memory accesses by those threads.

We now discuss how to prevent the races between messages. Races between `chmod` messages are harmless because `chmod` messages with different labels are processed separately, and the type system of DSR ensures that no two different `chmod` requests would be issued by the same thread at the same program counter label. As for preventing races between other messages, our approach is to enforce the following linearity constraints:

- A closure can be invoked by at most one reactor instance.

- A remote variable can be initialized by at most one reactor instance.

These constraints can be enforced by a static program analysis, which tracks the uses of communication terms, including reactor names, closure identifiers, context identifiers and remote variables. Given a statement s and the typing assignment Γ for that statement, let $\textit{\textbf{RV}}(s,\Gamma)$ represent the multi-set of communication terms appearing in the `exec` and `setvar` statements in s. Note that $\textit{\textbf{RV}}(s,\Gamma)$ is a multi-set so that multiple occurrences of the same value can be counted. Given a reactor declaration $r = c[\overline{x\!:\!\sigma}]\{\textit{\textbf{pc}},\ \mathcal{Q},\ \overline{\pi \triangleright z\!:\!\tau},\ \lambda\overline{y\!:\!\tau}.s\}$, let $\textit{\textbf{RV}}(r,\Gamma)$ denote the multi-set of communication terms appearing in r with respect to Γ. Then we have

$$\textit{\textbf{RV}}(r,\Gamma) = \textit{\textbf{RV}}(s,\ \Gamma, \overline{x\!:\!\sigma}, \overline{y\!:\!\tau_1}, \overline{z\!:\!\tau_2})$$

Given a program P such that $\Gamma \vdash P$, we can ensure that there are no races between messages by enforcing the following three conditions:

- **RV1**. For any r in P, $\textit{\textbf{RV}}(r,\Gamma)$ is a set.
- **RV2**. If $\langle c[\overline{v}],\ v\rangle.z \in \textit{\textbf{RV}}(r,\Gamma)$, then v is either `cid` or `nid`, and for any other r' in P, $\langle c[\overline{v}],\ \texttt{cid}\rangle.z \notin \textit{\textbf{RV}}(r',\Gamma)$. Furthermore, if v is `cid`, then c has no reactor parameters, and $\overline{v}$ contains no variables.
- **RV3**. If $\langle c[\overline{v}],\ v\rangle \in \textit{\textbf{RV}}(r,\Gamma)$, then v is a variable. Furthermore, if r may be invoked by c directly or indirectly, then v is `nid`.

The first condition ensures that a reactor can perform at most one operation on a communication term. The second condition ensures that only one reactor is allowed to have $\langle c[\overline{v}],\ \texttt{cid}\rangle.z$ in its body. According to (RV2), if $\langle c[\overline{v}],\ \texttt{cid}\rangle.z$ appears in reactor c', then c' has no parameters. Therefore, only closure $\langle c',\ \eta\rangle$ can use $\langle c[\overline{v}],\ \eta\rangle.z$ without receiving the variable from its invoker. By (RV1), $\langle c',\ \eta\rangle$ can either initialize the variable or pass it on to another closure, ensuring that only one reactor may initialize $\langle c[\overline{v}],\ \eta\rangle.z$.

The third condition (RV3) ensures that no two threads created by normal execution with different closure identifiers can invoke the same closure. Suppose two threads with different closure identifiers $\langle c_1,\, \eta_1\rangle$ and $\langle c_2,\, \eta_2\rangle$ invoke the same closure $\langle c,\, \eta\rangle$. If the two threads are created by normal execution, then $\langle c_1,\, \eta_1\rangle$ and $\langle c_2,\, \eta_2\rangle$ can be serialized. Without loss of generality, suppose $\langle c_1,\, \eta_1\rangle$ is invoked first. Since the thread of $\langle c_1,\, \eta_1\rangle$ invokes $\langle c,\, \eta\rangle$, the thread of $\langle c,\, \eta\rangle$ invokes $\langle c_2,\, \eta_2\rangle$ directly or indirectly. By (RV3), η is the value of `nid` for the thread of $\langle c_2,\, \eta_2\rangle$. This contradicts the assumption that the thread of $\langle c_1,\, \eta_1\rangle$ invokes $\langle c,\, n\rangle$, since η is unique for the thread of $\langle c_2,\, \eta_2\rangle$.

We say that a program P is *race-free* if P satisfies (RV1)–(RV3), and use the notation $\Gamma \Vdash P$ to denote that P is well-typed and race-free.

5.4 Noninterference

This section formalizes the properties of confidentiality and integrity noninterference for the execution model of DSR, and proves that a well-typed and race-free DSR program satisfies the noninterference properties.

Unlike a trusted single-machine platform, a distributed system may be under active attacks launched from bad hosts. Possible active attacks are formalized by the evaluation rules (A1)–(A3), as discussed in Section 5.2. Since we ignore timing channels, the availability attack in rule (A3) does not produce any observable effects, and is irrelevant to confidentiality or integrity noninterference. The attacks in rules (A1) and (A2) only produce low-integrity effects. Thus, those attacks do not affect the integrity noninterference property. For confidentiality, the attacks may be relevant because they may affect low-integrity low-confidentiality data, and generate different low-confidentiality outputs. However, such attacks can be viewed as providing different low-confidentiality inputs. Therefore, we assume that attackers would not affect low-confidentiality data when considering the confidentiality noninterference, which ensures that a program pro-

(VE1) $v \approx v$ (VE2) $\texttt{none} \approx v$ (VE3) $\dfrac{t_1 = t_2 \Rightarrow v_1 \approx v_2}{v_1@t_1 \approx v_2@t_2}$

(MSE1) $$\frac{P(c[\overline{v}]) = c\{pc',\ \mathcal{Q},\ \overline{\pi \triangleright z\!:\!\tau},\ \lambda\overline{x\!:\!\tau_1}.s\} \quad \forall i.\ \zeta(\tau_{1i}) \Rightarrow v_{1i} \approx v_{2i}}{[\texttt{exec}\ \langle c[\overline{v}],\ \eta\rangle :: pc, \overline{v_1}, \mathcal{Q}, t] \approx_\zeta [\texttt{exec}\ \langle c[\overline{v}],\ \eta\rangle :: pc, \overline{v_2}, \mathcal{Q}, t]}$$

(MSE2) $$\frac{\zeta(pc) \Rightarrow \ell_1 = \ell_2}{[\texttt{chmod}\ \langle c[\overline{v}],\ \eta\rangle :: pc, \ell_1, \mathcal{Q}, t] \approx_\zeta [\texttt{chmod}\ \langle c[\overline{v}],\ \eta\rangle :: pc, \ell_2, \mathcal{Q}, t]}$$

(MSE3) $$\frac{\zeta(c[\overline{v}].z) \Rightarrow v_1 \approx v_2}{[\texttt{setvar}\ \langle c[\overline{v}],\ \eta\rangle.z :: v_1, t] \approx_\zeta [\texttt{setvar}\ \langle c[\overline{v}],\ \eta\rangle.z :: v_2, t]}$$

(ME) $$\frac{\begin{array}{c}\forall h_1, h_2, m, t.\ \zeta(m, h_1) \wedge \zeta(m, h_2) \wedge t \leq \mathit{min}(\mathcal{T}_1(h_1, t), \mathcal{T}_2(h_2, t)) \Rightarrow \mathcal{M}_1[h_1, m, t] = \mathcal{M}_2[h_2, m, t] \\ \forall h_1, h_2, m.\ \zeta(m, h_1) \wedge \zeta(m, h_2) \Rightarrow \mathcal{M}_1[h_1, m, t_0] \approx \mathcal{M}_2[h_2, m, t_0]\end{array}}{\Gamma \vdash \langle \mathcal{M}_1, \mathcal{T}_1\rangle \approx_\zeta \langle \mathcal{M}_2, \mathcal{T}_2\rangle}$$

(CE) $$\frac{\mathit{varmap}\,(P, c[\overline{v}]) \vdash \mathcal{A}_1 \approx_\zeta \mathcal{A}_2 \quad \zeta(c[\overline{v}]) \Rightarrow t_1 = t_2}{P \vdash \langle c[\overline{v}], \eta, \ell_1, \mathcal{A}_1, t_1, *\rangle \approx_\zeta \langle c[\overline{v}], \eta, \ell_2, \mathcal{A}_2, t_2, *\rangle}$$

(EE) $$\frac{\begin{array}{l}\forall h_1, h_2.\ \forall t \leq \mathit{min}(\mathcal{T}_1(h_1, t), \mathcal{T}_2(h_2, t)). \\ \qquad ((\exists j \in \{1, 2\}.\ \langle h_j, h'_j, \mu_j\rangle \in \mathcal{E}_j.\mathit{messages}(h_j, *, [* :: *, t]) \wedge \forall i \in \{1, 2\}.\ \zeta(\mu_j, h_i)) \Rightarrow \\ \qquad\qquad (\forall i \in \{1, 2\}.\ \mathcal{E}_i.\mathit{messages}(h_i, *, [* :: *, t]) = \langle h_i, h'_i, \mu_i\rangle)\ \wedge\ \mu_1 \approx_\zeta \mu_2 \\ \forall h_1, h_2.\ \forall\langle c[\overline{v}],\ \eta\rangle.\ \zeta(c[\overline{v}], h_1) \wedge \zeta(c[\overline{v}], h_2) \ \Rightarrow\ P \vdash \mathcal{E}_1.\mathit{closure}(h_1, c[\overline{v}], \eta) \approx_\zeta \mathcal{E}_2.\mathit{closure}(h_2, c[\overline{v}], \eta)\end{array}}{P \vdash \langle \mathcal{E}_1, \mathcal{T}_1\rangle \approx_\zeta \langle \mathcal{E}_2, \mathcal{T}_2\rangle}$$

(TE) $$\frac{t_1 \approx t_2}{\langle s_1, t_1, h_1, c[\overline{v}], \eta\rangle \approx_\zeta \langle s_2, t_2, h_2, c[\overline{v}], \eta\rangle}$$

(TPE) $$\frac{\begin{array}{c}\forall t' \leq t.\ \forall h_1, h_2.\ (\forall i \in \{1, 2\}.\ \zeta(t', h_i) \wedge \Theta_i(h_i, t') = \theta_i) \Rightarrow \theta_1 \approx_\zeta \theta_2 \\ (\forall t' < t.\ (\exists h.\ \exists j \in \{1, 2\}.\ \Theta_j(h, t') = \theta \wedge \zeta(t', h)) \ \Rightarrow\ \forall i \in \{1, 2\}.\ \mathit{stable}_\zeta(\Theta_i,\ \mathcal{Q},\ t')\end{array}}{t \vdash \Theta_1 \approx_\zeta \Theta_2}$$

(SE) $$\frac{\begin{array}{c}\forall i \in \{1, 2\}.\ \mathcal{T}_i = \mathit{timestamps}(\Theta_i, \mathcal{E}_i, \zeta) \quad \Gamma \vdash \langle \mathcal{M}_1, \mathcal{T}_1\rangle \approx_\zeta \langle \mathcal{M}_2, \mathcal{T}_2\rangle \quad \Gamma \vdash \langle \mathcal{E}_1, \mathcal{T}_1\rangle \approx_\zeta \langle \mathcal{E}_2, \mathcal{T}_2\rangle \\ \mathit{min}(\mathit{max}(\mathcal{T}_1, \zeta),\ \mathit{max}(\mathcal{T}_2, \zeta)) \vdash \Theta_1 \approx_\zeta \Theta_2\end{array}}{\Gamma \vdash \langle \Theta_1,\ \mathcal{M}_1,\ \mathcal{E}_1\rangle \approx_\zeta \langle \Theta_2,\ \mathcal{M}_2,\ \mathcal{E}_2\rangle}$$

[Auxiliary definitions]

$$\frac{\exists H.\ (\forall h_i \in H.\ \zeta(t, h) \ \Rightarrow\ \Theta(h_i, t) = \langle s_i, t_i, h_i, c[\overline{v}], \eta\rangle \wedge \Gamma\,;\mathcal{Q}\,; pc_i \vdash s_i : \tau \wedge \neg\zeta(pc_i)\ \wedge\ \exists W \in \mathcal{Q}.\ W \subseteq H)}{\mathit{stable}_\zeta(\Theta,\ \mathcal{Q},\ t)}$$

Figure 5.5: ζ-Equivalence relation

duces the same low-confidentiality outputs only if the program receives the same inputs.

5.4.1 ζ-Consistency

As discussed in Section 3.5.2, both confidentiality and integrity noninterference properties can be viewed as the preservation of a *consistency* relation between the program states that satisfies a ζ condition, which intuitively represents low-confidentiality or high-integrity. In DSR, two system configurations are ζ*-consistent* if their ζ parts are consistent, meaning that it cannot be determined that the two configurations belong to different executions by examining their ζ parts.

For confidentiality, the ζ condition is defined as follows:

$$\zeta(\mathtt{x}) = \begin{cases} C(\mathtt{x}) \leq l_{\mathtt{A}} & \text{if } \mathtt{x} \text{ is a label} \\ C(\mathit{label}(\mathtt{x})) \leq l_{\mathtt{A}} & \text{otherwise} \end{cases}$$

where $\mathit{label}(\mathtt{x})$ denotes the label of $\mathtt{x}$, which is a program term such as type τ, reference m, host h, timestamp t and message μ. The definition of $\mathit{label}(\mathtt{x})$ is shown below:

- $\mathit{label}(h)$ is the label specified on host h.
- $\mathit{label}(\tau)$ is ℓ, if $\tau = \beta_\ell$ or $\tau = \beta_\ell@\mathcal{Q}$.
- $\mathit{label}(\mu)$ is *pc* if μ is an `exec` or `chmod` message and *pc* is the program counter label of μ, and $\mathit{label}(\mu)$ is ℓ if μ is a `setvar` message and ℓ is the label of the remote variable targeted by μ.
- $\mathit{label}(t)$ is the last *pc* component of the global part of t.
- $\mathit{label}(c[\overline{v}])$ is the program counter label of $c[\overline{v}]$.

Whether a term x satisfies the ζ condition may depend on the host where x resides. For instance, any term on a low-integrity host is also low-integrity. In general, whether term x on host h satisfies ζ can be determined by $\zeta(\mathit{label}(x) \sqcap \mathit{label}(h))$, which is written as $\zeta(x, h)$.

For integrity, the ζ condition represents the notion of high-integrity and is defined as below:

$$\zeta(\mathtt{x}) = \begin{cases} I(\mathtt{x}) \not\leq l_{\mathtt{A}} & \text{if } \mathtt{x} \text{ is a label} \\ I(\mathit{label}(\mathtt{x})) \not\leq l_{\mathtt{A}} & \text{otherwise} \end{cases}$$

The key issue in formalizing the noninterference properties is to define ζ-consistency between system configurations, which depends on the ζ-consistency relations between thread pools, memories, and environments. Figure 5.5 shows the definitions of those ζ-consistency relations in the form of inference rules.

Rules (VE1)–(VE3) define a consistency relation ($\approx$) between values. Intuitively, $v_1 \approx v_2$ means they may be used in the same way and in the same execution. More concretely, v_1 and v_2 may be assigned to the replicas of a memory reference, and they may appear as the same component in the replicas of a message. Rule (VE1) is standard. Rule (VE2) says that `none` is consistent with any value v because `none` represents an unavailable value that cannot be used in any computation to generate observable effects. Rule (VE3) says that two versioned $v_1@t_1$ and $v_2@t_2$ are consistent if $t_1 = t_2$ implies $v_1 \approx v_2$. Two versioned values with different timestamps are considered consistent, because they may be used in the same way and in the same execution.

Rules (MSE1)–(MSE3) define the ζ-consistency between messages. Rule (MSE1) says that two messages $[\mathtt{exec}\ \langle c[\overline{v}],\, \eta\rangle :: pc, \overline{v_1}, \mathcal{Q}, t]$ and $[\mathtt{exec}\ \langle c[\overline{v}],\, \eta\rangle :: pc, \overline{v_2}, \mathcal{Q}, t]$ are ζ-consistent if any two corresponding arguments $\overline{v_{1i}}$ and $\overline{v_{2i}}$ are consistent on condition that $\zeta(\tau_{1i})$ holds. Intuitively, $\neg\zeta(\tau_{1i})$ means that values with type τ_{1i} can be distinguishable. Rules (MSE2) and (MSE2) are interpreted similarly.

Rule (ME) defines memory ζ-consistency. Intuitively, two global memories $\mathcal{M}_1$ and $\mathcal{M}_2$ are considered ζ-consistent with respect to the typing assignment Γ, if for any hosts h_1 and h_2, any reference m, and any time t, $\zeta(m, h_1)$ and $\zeta(m, h_2)$ imply $\mathcal{M}_1[h_1, m, t] \approx \mathcal{M}[h_2, m, t]$. However, with knowledge of thread timestamps, $\mathcal{M}_1$ and $\mathcal{M}_2$ may be distinguishable if $\mathcal{M}_1[h_1, m, t] = n$ and $\mathcal{M}_2[h_2, m, t] = \mathtt{none}$, because

$\mathcal{M}_2[h_2, m, t] = \texttt{none}$ can be determined by reading the most recent version of m by t on host h_2. If there exists a thread on h_2 with a timestamp t' such that $t' \approx t$ (the global parts of t and t' are equal) and $t \leq t'$, then $\mathcal{M}_1$ and $\mathcal{M}_2$ must belong to different executions. Therefore, the ζ-consistency of $\mathcal{M}_1$ and $\mathcal{M}_2$ should be considered with respect to the timing information, which is captured by a timing map $\mathcal{T}$ that maps a host h to the set of timestamps of the threads on h. Let $\mathcal{T}(h, t)$ be the timestamp t' in $\mathcal{T}[h]$ such that $t \approx t'$. Then $\mathcal{M}_1[h_1, m, t]$ and $\mathcal{M}_2[h_2, m, t]$ need to be *equal* if $t \leq \mathit{min}(\mathcal{T}_1(h_1, t), \mathcal{T}_2(h_2, t))$, which means the two threads on hosts h_1 and h_2 have reached time t. Therefore, if m is updated at time t in one thread, then m should also be updated at t in another thread. Otherwise, the two threads, along with $\mathcal{M}_1$ and $\mathcal{M}_2$ belong to different executions. Rule (ME) also requires that $\mathcal{M}_1$ and $\mathcal{M}_2$ have ζ-consistent states at the initial time $t_0 = \langle\rangle$. The second premise of rule (ME) says $\mathcal{M}_1[h_1, m, t_0]$ and $\mathcal{M}_2[h_2, m, t_0]$ are equivalent if $\zeta(m, h_i)$ holds for $i \in \{1, 2\}$.

Rule (CE) defines the equivalence relationship between reactor closures. Two closures are equivalent if they have the same closure identifier $\langle c[\overline{v}],\, \eta\rangle$ and ζ-consistent variable records. In this rule, the notation $\mathit{varmap}(P, c[\overline{v}])$ represents the local typing assignment Γ' of $c[\overline{v}]$ with respect to P, mapping local variables of $c[\overline{v}]$ to types. The notation $\Gamma' \vdash \mathcal{A}_1 \approx_\zeta \mathcal{A}_2$ means that for any z in $\mathit{dom}(\Gamma')$, $\zeta(\Gamma'(z))$ implies $\mathcal{A}_1(z) \approx \mathcal{A}_2(z)$.

Rule (EE) defines the equivalence relationship between environments. Intuitively, two environments are equivalent if the corresponding (with the same timestamp) messages in the two environments are ζ-consistent, and the corresponding (with the same reference) closures are ζ-consistent. Like in rule (ME), we need to take into account the case that there exists a message at time t in one environment, but there does not exist such a message in the other environment. Similarly, ζ-consistency between two environments $\mathcal{E}_1$ and $\mathcal{E}_2$ is considered with respect to the corresponding timing maps $\mathcal{T}_1$ and $\mathcal{T}_2$. Formally, given two hosts h_1 and h_2, and some timestamp t that is less than or equal to $\mathcal{T}_i(h_1, t)$, if there exists a message μ_j in $\mathcal{E}_j$ such that μ_j has the timestamp t and the program counter label pc_{μ_j} such that $\zeta(pc_{\mu_j,h_i})$ holds for $i \in \{1, 2\}$, then in both $\mathcal{E}_1$ and $\mathcal{E}_2$, exactly one message (μ_1 and μ_2, respectively) is sent at time t, and $\mu_1 \approx_\zeta \mu_2$. Furthermore, for any hosts h_1 and h_2 and any closure reference $\langle c[\overline{v}],\, \eta\rangle$, if $\zeta(c[\overline{v}], h_1)$ and $\zeta(c[\overline{v}], h_2)$, then the closures identified by $\langle c[\overline{v}],\, \eta\rangle$ on hosts h_1 and h_2 are ζ-consistent.

Rule (TE) defines the equivalence between threads. Two threads are equivalent if they correspond to the same reactor instance, and their base timestamps are the same.

Rule (TPE) defines ζ-consistency between thread pools. Two thread pools Θ_1 and Θ_2 are equivalent with respect to their corresponding timing states $\mathcal{T}_1$ and $\mathcal{T}_2$, written $\langle\Theta_1, \mathcal{T}_1\rangle \approx_\zeta \langle\Theta_2, \mathcal{T}_2\rangle$, if two conditions hold. First, any two hosts h_1 and h_2, and any timestamp t' satisfying $t' \leq t$ where t is the smaller of the largest timestamps satisfying $\zeta(t)$ in $\mathcal{T}_1$ and $\mathcal{T}_2$, if $\zeta(t', h_i)$ and there exists a thread θ_i on h_i and with timestamp t_i such that $t_i \approx t'$ in Θ_i, then $\theta_1 \approx_\zeta \theta_2$. Second, for any timestamp t' less than t, if there exists a thread at t' in either Θ_1 or Θ_2, then the threads at time t' are *stable* with respect to the quorum system $\mathcal{Q}$ and the condition ζ in both Θ_1 and Θ_2. Intuitively, these two conditions ensure that both Θ_1 and Θ_2 have reached t, and the corresponding threads before t are equivalent.

Rule (SE) defines the equivalence relationship between system configurations. Two configurations are considered equivalent if their corresponding components are equivalent with respect to their timing states, which are computed by *timestamps*$(\Theta, \mathcal{E}, \zeta)$. Suppose $\mathcal{T} =$ *timestamps*$(\Theta, \mathcal{E}, \zeta)$. Then $\mathcal{T}[h, t] = t'$ means that one of the following cases occurs. First, there exists a thread on h with timestamp t' such that $t' \approx t$, and for any thread on h with timestamp t'', $t'' \approx t$ implies $t'' \leq t'$. Second, there exists a closure on h with timestamp t' and access control label ℓ such that $\zeta(\ell)$ and $t' \approx t$, and there is no thread on h with timestamp t'' such that $t'' \approx t$. The notation *current-time*$(\mathcal{T}, \zeta)$ is the most recent timestamp t such that $\mathcal{T}[h, t] = t$ and $\zeta(t, h)$. Intuitively, $\mathit{min}(\mathit{max}(\mathcal{T}_1, \zeta),\ \mathit{max}(\mathcal{T}_2, \zeta))$ is the current timestamp of the lagging one of the two configuration.

5.4.2 The DSR* language

To facilitate proving the noninterference results of DSR, we introduce a bracket construct that syntactically captures the differences between executions of the same program on different inputs. The extended language is called DSR*. Except for proving noninterference, the DSR* language also helps reasoning about concurrent execution of threads on different hosts.

Intuitively, each machine configuration in DSR* encodes multiple DSR local configurations that capture the states of concurrent threads on different hosts. The operational semantics of DSR* is consistent with that of DSR in the sense that the evaluation of a DSR* configuration is equivalent to the evaluation of DSR configurations encoded by the DSR* configuration. The type system of DSR* can be instantiated to ensure that a well-typed DSR* configuration satisfies certain invariants. Then the subject reduction result of DSR* implies that the invariant is preserved during evaluation. In particular, the invariant may represent the ζ-consistency relation corresponding to a noninterference result. For example, a DSR* configuration may encode two DSR configurations, and the invariant may be that the low-confidentiality parts of the two configurations are equivalent. Then the subject reduction result of DSR* implies the preservation of the ζ-consistency between two DSR local configurations. The proof technique is similar to the one used to prove the noninterference result of Aimp in Section 3.5.2.

Syntax extensions

The syntax extensions of DSR* are bracket constructs, which are composed of a set of DSR terms and used to capture the differences between DSR configurations.

$$\begin{array}{rrcl} \text{Values} & v & ::= & \dots \mid (v_1, \dots, v_n) \\ \text{Statements} & s & ::= & \dots \mid (s_1, \dots, s_n) \end{array}$$

Bracket constructs cannot be nested, so the subterms of a bracket construct must be DSR terms. Given a DSR* statement s, let $\lfloor s \rfloor_i$ represent the DSR statements that s encodes. The projection functions satisfy $\lfloor (s_1, \dots, s_n) \rfloor_i = s_i$ and are homomorphisms on other statement and expression forms. A DSR* memory M incorporates multiple DSR local memory snapshots.

Since a DSR* term effectively encodes multiple DSR terms, the evaluation of a DSR* term can be projected into multiple DSR evaluations. An evaluation step of a bracket statement $(s_1, \dots, s_n)$ is an evaluation step of any s_i, and s_i can only access the corresponding projection of the memory. Thus, the configuration of DSR* has an index $i \in \{\bullet, 1, \dots, n\}$ that indicates whether the term to be evaluated is a subterm of a bracket term, and if so, which branch of a bracket the term belongs to. For example, the configuration $\langle s,\ \mathrm{M},\ \Omega,\ t \rangle_1$ means that s belongs to the first branch of a bracket, and s can only access the first projection of M. We write "$\langle s,\ \mathrm{M},\ \Omega,\ t \rangle$" for "$\langle s,\ \mathrm{M},\ \Omega,\ t \rangle_\bullet$".

The operational semantics of DSR* is shown in Figure 5.6. Since DSR* is used to analyze the local evaluation steps of DSR, only the evaluation rules for statements are presented. An evaluation step of a DSR* statement is denoted by $\langle s,\ \mathrm{M},\ \Omega,\ t \rangle_i \longmapsto \langle s',\ \mathrm{M}',\ \Omega',\ t' \rangle_i$. Most evaluation rules are straightforwardly adapted from the semantics of DSR by indexing each configuration with i. The main change is that memory accesses and timestamp increments are to be performed on the memory and timestamp projection corresponding to index i. In rule (S1), the updated memory M$'$ is M$[m \mapsto_i v@\lfloor t \rfloor_i]$, where $\lfloor t \rfloor_i$ is the ith projection of t. Suppose M$[m] = v'$. Then

$\mathrm{M}'[m] = (\lfloor v' \rfloor_1, \ldots, v@\lfloor t \rfloor_i, \ldots, \lfloor v' \rfloor_n)$. In DSR*, the local part of a timestamp t may have the form $\overline{n}$, or $\overline{n}, (\overline{n_1}, \ldots, \overline{n_k})$, which indicates that the execution deviates after local time $\overline{n}$. Suppose $t = \overline{n}, (\overline{n_1}, \ldots, \overline{n_k})$. Then we have

$$\begin{array}{rcl} \lfloor t \rfloor_i & = & \overline{n}, \overline{n_i} \\ t \triangleleft_i 1 & = & \overline{n}, (\overline{n_1}, \ldots, \overline{n_i} \triangleleft 1, \ldots, \overline{n_k}) \\ t \triangleright_i 1 & = & \overline{n}, (\overline{n_1}, \ldots, \overline{n_i} \triangleright 1, \ldots, \overline{n_k}) \\ t \triangleright 1 & = & \overline{n} + 1 \end{array}$$

where $\overline{n} \triangleleft 1 = \overline{n}, 1$, and $\overline{n} \triangleright 1 = n_1, \ldots, n_{k-1} + 1$, and $\overline{n} + 1 = n_1, \ldots, n_k + 1$. If $t = \overline{n}$, then $t \triangleleft_i 1 = \overline{n}, (\epsilon, \ldots, 1, \ldots, \epsilon)$.

There are also new evaluation rules (S11–S14) for manipulating bracket constructs. The following adequacy and soundness lemmas state that the operational semantics of DSR* is adequate to encode the execution of multiple DSR terms.

Lemma 5.4.1 (Projection i). Suppose $\langle e,\, \mathrm{M} \rangle \Downarrow v$. Then $\langle \lfloor e \rfloor_i,\, \lfloor \mathrm{M} \rfloor_i \rangle \Downarrow \lfloor v \rfloor_i$ holds for $i \in \{1, \ldots, n\}$.

Proof. By induction on the structure of e.

- e is v. Then $\lfloor e \rfloor_i = \lfloor v \rfloor_i$.
- e is $!m$. Then $\lfloor e \rfloor_i = !m$, and $\langle !m,\, \lfloor \mathrm{M} \rfloor_i \rangle \Downarrow \lfloor \mathrm{M} \rfloor_i(m)$, and $v = \lfloor \mathrm{M}(m) \rfloor_i = \lfloor \mathrm{M} \rfloor_i(m)$.
- e is $!(m_1, \ldots, m_n)$. By (E4), $v = (v_1, \ldots, v_n)$, where $v_i = \lfloor \mathrm{M}(m_i) \rfloor_i$. Moreover, $\langle !m_i,\, \lfloor \mathrm{M} \rfloor_i \rangle \Downarrow v_i$.
- e is $e_1 + e_2$. By induction, $\langle \lfloor e_j \rfloor_i,\, \lfloor \mathrm{M} \rfloor_i \rangle \Downarrow \lfloor v_j \rfloor_i$ for $j \in \{1, 2\}$. Thus, $\langle \lfloor e_1 + e_2 \rfloor_i,\, \lfloor \mathrm{M} \rfloor_i \rangle \Downarrow \lfloor v_1 \uplus v_2 \rfloor_i$.

□

Lemma 5.4.2 (Projection ii). Suppose s is a DSR statement, and $\lfloor \mathrm{M} \rfloor_i = M_i$ and $\lfloor \Omega \rfloor_i = \Omega_i$ and $\lfloor t \rfloor_i = t_i$. Then for $i \in \{1, \ldots, n\}$, $\langle s,\, \mathrm{M},\, \Omega,\, t \rangle_i \longmapsto \langle s',\, \mathrm{M}',\, \Omega',\, t' \rangle_i$

(E1) $\dfrac{\lfloor \mathrm{M}(m) \rfloor_i = v}{\langle !m,\, \mathrm{M} \rangle_i \Downarrow v}$ (E2) $\dfrac{\langle e_1,\, \mathrm{M} \rangle_i \Downarrow v_1 \quad \langle e_2,\, \mathrm{M} \rangle_i \Downarrow v_2 \quad v = v_1 \oplus v_2}{\langle e_1 + e_2,\, \mathrm{M} \rangle_i \Downarrow v}$ (E3) $\langle v,\, \mathrm{M} \rangle_i \Downarrow \lfloor v \rfloor_i$

(E4) $\dfrac{\lfloor \mathrm{M}(m_i) \rfloor_i = v_i}{\langle !(m_1, \ldots, m_n),\, \mathrm{M} \rangle \Downarrow (v_1, \ldots, v_n)}$

(S1) $\dfrac{\langle e,\, \mathrm{M} \rangle_i \Downarrow v}{\langle m := e,\, \mathrm{M},\, \Omega,\, t \rangle_i \longmapsto \langle \texttt{skip},\, \mathrm{M}[m \mapsto_i v@\lfloor t \rfloor_i],\, \Omega,\, t +_i 1 \rangle_i}$ (S2) $\dfrac{\langle s_1,\, \mathrm{M},\, \Omega,\, t \rangle_i \longmapsto \langle s_1',\, \mathrm{M}',\, \Omega',\, t' \rangle_i}{\langle s_1; s_2,\, \mathrm{M},\, \Omega,\, t \rangle_i \longmapsto \langle s_1'; s_2,\, \mathrm{M}',\, \Omega',\, t' \rangle_i}$

(S3) $\langle \texttt{skip}; s,\, \mathrm{M},\, \Omega,\, t \rangle_i \longmapsto \langle s,\, \mathrm{M},\, \Omega,\, t \rangle_i$ (S4) $\langle \texttt{fi}; s,\, \mathrm{M},\, \Omega,\, t \rangle_i \longmapsto \langle s,\, \mathrm{M},\, \Omega,\, t \rhd_i 1 \rangle_i$

(S5) $\dfrac{\langle e,\, \mathrm{M} \rangle_i \Downarrow n \qquad n > 0}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2,\, \mathrm{M},\, \Omega,\, t \rangle_i \longmapsto \langle s_1; \texttt{fi},\, \mathrm{M},\, \Omega,\, t \lhd_i 1 \rangle_i}$

(S6) $\dfrac{\langle e,\, \mathrm{M} \rangle_i \Downarrow n \qquad n \le 0}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2,\, \mathrm{M},\, \Omega,\, t \rangle_i \longmapsto \langle s_2; \texttt{fi},\, \mathrm{M},\, \Omega,\, t \lhd_i 1 \rangle_i}$

(S7) $\dfrac{\langle \overline{e},\, \mathrm{M} \rangle_i \Downarrow \overline{v_1}}{\langle \texttt{exec}(c[\overline{v}],\, \eta,\, pc,\, \mathcal{Q},\, \overline{e}),\, \mathrm{M},\, \Omega,\, t \rangle_i \longmapsto \langle \texttt{halt},\, \mathrm{M},\, \Omega \cup [\texttt{exec}\, \langle c[\overline{v}],\, \eta \rangle :: t, pc, \mathcal{Q}, \overline{v_1}]_i,\, t +_i 1 \rangle_i}$

(S8) $\langle \texttt{chmod}(c[\overline{v}],\, n,\, pc,\, \mathcal{Q},\, \ell),\, \mathrm{M},\, \Omega,\, t \rangle_i \longmapsto \langle \texttt{skip},\, \mathrm{M},\, \Omega \cup [\texttt{chmod}\, \langle c[\overline{v}],\, \eta \rangle :: t, pc, \mathcal{Q}, \ell]_i,\, t +_i 1 \rangle_i$

(S9) $\langle \texttt{setvar}(\langle c[\overline{v}],\, \eta \rangle.z,\, v),\, \mathrm{M},\, \Omega,\, t \rangle_i \longmapsto \langle \texttt{skip},\, \mathrm{M},\, \Omega \cup [\texttt{setvar}\, \langle c[\overline{v}],\, \eta \rangle.z :: t, v]_i,\, t +_i 1 \rangle_i$

(S10) $\langle (\texttt{skip}, \ldots, \texttt{skip}),\, \mathrm{M},\, t \rangle \longmapsto \langle \texttt{skip},\, \mathrm{M},\, t \rangle$ (S11) $\langle (\texttt{fi}, \ldots, \texttt{fi}),\, \mathrm{M},\, t \rangle \longmapsto \langle \texttt{skip},\, \mathrm{M},\, t \rhd 1 \rangle$

(S12) $\dfrac{\langle e,\, \mathrm{M} \rangle \Downarrow (v_1, \ldots, v_n)}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2,\, \mathrm{M},\, \Omega,\, t \rangle \longmapsto \langle (\texttt{if}\ v_i\ \texttt{then}\ \lfloor s_1 \rfloor_i\ \texttt{else}\ \lfloor s_2 \rfloor_i \mid 1 \le i \le n),\, \mathrm{M},\, \Omega,\, t \rangle}$

(S13) $\dfrac{\langle s_i,\, \mathrm{M},\, \Omega,\, t \rangle_i \longmapsto \langle s_i',\, \mathrm{M}',\, \Omega',\, t' \rangle_i}{\langle (s_1, \ldots, s_i, \ldots, s_n),\, \mathrm{M},\, \Omega,\, t \rangle \longmapsto \langle (s_1, \ldots, s_i', \ldots, s_n),\, \mathrm{M}',\, \Omega',\, t' \rangle}$

(S14) $\langle (m_1, \ldots, m_n) := e,\, \mathrm{M},\, \Omega,\, t \rangle \longmapsto \langle (m_1 := \lfloor e \rfloor_1, \ldots, m_n = \lfloor e \rfloor_n),\, \mathrm{M},\, \Omega,\, t \rangle$

Figure 5.6: The operational semantics of DSR*

if and only if $\langle s,\, M_i,\, \Omega_i,\, t_i \rangle \longmapsto \langle s',\, M_i',\, \Omega_i',\, t_i' \rangle$ and $\lfloor \mathrm{M}' \rfloor_i = M_i'$ and $\lfloor \Omega' \rfloor_i = \Omega_i'$ and $\lfloor t' \rfloor_i = t_i'$

Proof. By induction on the derivation of $\langle s,\, \mathrm{M},\, \Omega,\, t \rangle_i \longmapsto \langle s',\, \mathrm{M}',\, \Omega',\, t' \rangle_i$.

- Case (S1). In this case, s is $m := e$. Then $\mathrm{M}' = \mathrm{M}[m \mapsto_i v@\lfloor t \rfloor_i]$, where $\langle e,\, \mathrm{M} \rangle_i \Downarrow v$. By Lemma 5.4.1, $\langle \lfloor e \rfloor_i,\, \lfloor \mathrm{M} \rfloor_i \rangle \Downarrow \lfloor v \rfloor_i$. Therefore, $M_i' = M[m \mapsto \lfloor v \rfloor_i@t_i] = \lfloor \mathrm{M}' \rfloor_i$. By (S1), $t' = t +_i 1$, which implies that $\lfloor t \rfloor_i' = \lfloor t \rfloor_i + 1 = t_i'$.

- Case (S2). By induction.

- Case (S3). $\Omega' = \Omega$, $\mathrm{M}' = \mathrm{M}$ and $t' = t$. In addition, $\langle \texttt{skip}; s,\, M_i,\, \Omega_i,\, t_i\rangle \longmapsto \langle s,\, M_i,\, \Omega_i,\, t_i\rangle$.
- Case (S4). We have $\langle \texttt{fi}; s,\, M_i,\, \Omega_i,\, t_i\rangle \longmapsto \langle s,\, M_i,\, \Omega_i,\, t_i \triangleright 1\rangle$.
- Case (S5). In this case, s is $\texttt{if}\ e \texttt{then} s_1 \texttt{else} s_2$, and $\langle e,\, M\rangle \Downarrow n$. By Lemma 5.4.1, $\langle e_i,\, M_i\rangle \Downarrow n$. By rule (S5), we have $\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2,\, M_i,\, \Omega_i,\, t_i\rangle \longmapsto \langle s_1,\, M_1,\, \Omega_i,\, t_i \triangleleft 1\rangle$.
- Case (S6). By the same argument as that of case (S5).
- Case (S7). By Lemma 5.4.1, $\langle \overline{e},\, M_i\rangle \Downarrow \lfloor \overline{v_1} \rfloor_i$. Therefore, $\langle s,\, M_i,\, \Omega_i,\, t_i\rangle \longmapsto \langle s,\, M_i,\, \Omega_i \cup \mu_i,\, t_i + 1\rangle$, and $\mu_i = \lfloor [\texttt{exec}\ \langle c[\overline{v}],\, \eta\rangle :: t, pc, \mathcal{Q}, \overline{v_1}] \rfloor_i$.
- Cases (S8) and (S9). By the same argument as that of case (S7).

□

Lemma 5.4.3 (Expression adequacy). Suppose $\langle e_i,\, M_i\rangle \Downarrow v_i$ for $i \in \{1, \ldots, n\}$, and there exists a DSR* configuration $\langle e,\, \mathrm{M}\rangle$ such that $\lfloor e \rfloor_i = e_i$ and $\lfloor \mathrm{M} \rfloor_i = M_i$. Then $\langle e,\, \mathrm{M}\rangle \Downarrow v$ such that $\lfloor v \rfloor_i = v_i$.

Proof. By induction on the structure of e. □

Definition 5.4.1 (Local run). A local run $\langle s,\, \mathrm{M},\, \Omega,\, t\rangle \longmapsto^* \langle s',\, \mathrm{M}',\, \Omega',\, t'\rangle$ represents a list of consecutive local evaluation steps: $\langle s,\, \mathrm{M},\, \Omega,\, t\rangle \longmapsto \langle s_1,\, \mathrm{M}_1,\, \Omega_1,\, t_1\rangle$, $\langle s_1,\, \mathrm{M}_1',\, \Omega_1,\, t_1\rangle \longmapsto \langle s_2,\, \mathrm{M}_2,\, \Omega_2,\, t_2\rangle, \ldots, \langle s_n,\, \mathrm{M}_n,\, \Omega_n,\, t_n\rangle \longmapsto \langle s',\, \mathrm{M}',\, \Omega',\, t'\rangle$, where M_i' and M_i may differ because the execution of other threads or active attacks may change the local memory snapshot.

Lemma 5.4.4 (One-step adequacy). Suppose $E_i = \langle s_i,\, M_i,\, \Omega_i,\, t_i\rangle \longmapsto \langle s_i',\, M_i',\, \Omega_i',\, t_i'\rangle$ for $i \in \{1, \ldots, n\}$, and there exists a DSR* configuration $\langle s,\, \mathrm{M},\, \Omega,\, t\rangle$ such that for all i, $\lfloor \langle s,\, \mathrm{M},\, \Omega,\, t\rangle \rfloor_i = \langle s_i,\, M_i,\, \Omega_i,\, t_i\rangle$. Then there exists $E = \langle s,\, \mathrm{M},\, \Omega,\, t\rangle \longmapsto^* \langle s',\, \mathrm{M}',\, \Omega',\, t'\rangle$ such that for any i, $\lfloor E \rfloor_i \preceq E_i$, and for some j, $\lfloor E \rfloor_j \approx E_j$.

Proof. By induction on the structure of s.

- s is $\texttt{skip}$. Then s_i is also $\texttt{skip}$ and cannot be further evaluated. Therefore, the lemma is correct in this case because its premise does not hold.

- s is $v := e$. In this case, s_i is $\lfloor v \rfloor_i := \lfloor e \rfloor_i$, and $\langle \lfloor v \rfloor_i := \lfloor e \rfloor_i,\, M_i,\, \Omega_i,\, t_i \rangle \longmapsto \langle \texttt{skip},\, M_i[m_i \mapsto_{t_i} v_i],\, \Omega_i,\, t_i + 1 \rangle$ where $m_i = \lfloor v \rfloor_i$ and $\langle \lfloor e \rfloor_i,\, M_1 \rangle \Downarrow v_i$. By Lemma 5.4.3, $\langle e,\, M \rangle \Downarrow v'$ and $\lfloor v' \rfloor_i = v_i$. If v is m, then $\langle v := e,\, \mathrm{M},\, \Omega,\, t \rangle \longmapsto \langle \texttt{skip},\, \mathrm{M}[m \mapsto v'@t],\, \Omega,\, t + 1 \rangle$. Since $\lfloor \mathrm{M} \rfloor_i = M_i$, we have $\lfloor \mathrm{M}[m \mapsto v'@t] \rfloor_i = M_i[m \mapsto \lfloor v'@t \rfloor_i]$. In addition, we have $\lfloor s' \rfloor_i = s'_i = \texttt{skip}$. If v is $(m_1, \ldots, m_n)$, then we have

$$E = \langle v := e,\, \mathrm{M},\, \Omega,\, t \rangle$$
$$\longmapsto \langle (m_1 := \lfloor e \rfloor_1, \ldots, m_n := \lfloor e \rfloor_n),\, \mathrm{M},\, \Omega,\, t \rangle$$
$$\longmapsto \langle (\texttt{skip}, \ldots, m_n := \lfloor e \rfloor_n),\, \mathrm{M}[m_1 \mapsto_1 v_1@t_1],\, \Omega,\, t +_1 1 \rangle$$

  It is clear that $\lfloor E \rfloor_1 \approx E_1$ and $\lfloor E \rfloor_i \preceq E_i$ for any i.

- s is $\texttt{if}\ e\ \texttt{then}\ s''_1\ \texttt{else}\ s''_2$. Therefore, s_i is $\texttt{if}\ \lfloor e \rfloor_i\ \texttt{then}\ \lfloor s''_1 \rfloor_i\ \texttt{else}\ \lfloor s''_2 \rfloor_i$. By Lemma 5.4.3, $\langle e,\, \mathrm{M} \rangle \Downarrow v$. If $v = n$, then for some j in $\{1, 2\}$, we have $E = \langle s,\, \mathrm{M},\, \Omega,\, t \rangle \longmapsto \langle s''_j; \texttt{fi},\, \mathrm{M},\, \Omega,\, t \triangleleft 1 \rangle$. By Lemma 5.4.1, $\langle \lfloor e \rfloor_i,\, \lfloor \mathrm{M} \rfloor_i \rangle \Downarrow n$, which implies $\langle s_i,\, M_i,\, \Omega_i,\, t_i \rangle \longmapsto \langle \lfloor s''_j \rfloor_i; \texttt{fi},\, M_i,\, \Omega_i,\, t_i \triangleleft 1 \rangle$. If $v = (n_1, \ldots, n_k)$, then we have

$$E = \langle s,\, \mathrm{M},\, \Omega,\, t \rangle \longmapsto \langle (\texttt{if}\ n_i\ \texttt{then}\ \lfloor s_1 \rfloor_i\ \texttt{else}\ \lfloor s_2 \rfloor_i \mid 1 \leq i \leq n),\, \mathrm{M},\, \Omega,\, t \rangle$$
$$\longmapsto \langle (s''_j; \texttt{fi}, \ldots, \texttt{if}\ n_k\ \texttt{then}\ \lfloor s_1 \rfloor_k\ \texttt{else}\ \lfloor s_2 \rfloor_k),\, \mathrm{M},\, \Omega,\, t \triangleleft_i 1 \rangle.$$

  By Lemma 5.4.1, $\langle \lfloor e \rfloor_i,\, M_i \rangle \Downarrow n_i$. Therefore, $\lfloor E \rfloor_1 \approx E_1$, and $\lfloor E \rfloor_i \preceq E_i$.

- s is $s''_1; s''_2$. In this case, $s_i = \lfloor s''_1 \rfloor_i; \lfloor s''_2 \rfloor_i$. There are four cases:

  – $\lfloor s''_1 \rfloor_i$ is not $\texttt{skip}$ or $\texttt{fi}$ for any i. Then the lemma holds by induction.

  – s''_1 is $\texttt{skip}$ or $(\texttt{skip}, \ldots, \texttt{skip})$. Then $\langle s,\, \mathrm{M},\, \Omega,\, t \rangle \longmapsto^* \langle s''_2,\, \mathrm{M},\, \Omega,\, t \rangle$. Correspondingly, $\langle s_i,\, M_i,\, \Omega_i,\, t_i \rangle \longmapsto \langle \lfloor s''_2 \rfloor_i,\, M_i,\, \Omega_i,\, t_i \rangle$.

  - s''_1 is `fi` or (`fi`, …, `fi`). Then $\langle s,\, \mathrm{M},\, \Omega,\, t\rangle \longmapsto \langle s''_2,\, \mathrm{M},\, \Omega,\, t \triangleright 1\rangle$. In addition, $\langle s_i,\, M_i,\, \Omega_i,\, t_i\rangle \longmapsto \langle \lfloor s''_2 \rfloor_i,\, M_i,\, \Omega_i,\, t_i \triangleright 1\rangle$, and $\lfloor t \triangleright 1\rfloor_i = \lfloor t \rfloor_i \triangleright 1 = t_i \triangleright 1$.

  - s''_1 is $(s_{11}, \ldots, s_{1n})$, and there exists some s_{1j} that is neither `skip` nor `fi`. Then we have $\langle s_{1j},\, \mathrm{M},\, \Omega,\, t\rangle_j \longmapsto \langle s'_{1j},\, \mathrm{M}',\, \Omega',\, t'\rangle_j$, and $\langle s,\, \mathrm{M},\, \Omega,\, t\rangle \longmapsto \langle (s_{11}, \ldots, s'_{1j}, \ldots, s_{1n}); s''_2,\, \mathrm{M}',\, \Omega',\, t'\rangle$. By Lemma 5.4.2, $\langle s_{1j},\, M_j,\, \Omega_j,\, t_j\rangle \longmapsto \langle s'_{1j},\, M'_j,\, \Omega'_j,\, t'_j\rangle$, and $\lfloor \mathrm{M}'\rfloor_j = M'_j$ and $\lfloor \Omega'\rfloor_j = \Omega'_j$ and $\lfloor t'\rfloor_j = t'_j$. It is clear that for any i such that $i \neq j$, $\lfloor \mathrm{M}'\rfloor_i = M_i$ and $\lfloor \Omega'\rfloor_i = \Omega_i$ and $\lfloor t'\rfloor_i = t_i$.

- s is $\mathtt{exec}(c[\overline{v}],\, \eta,\, pc,\, \mathcal{Q},\, \overline{e})$. Then $\langle s,\, \mathrm{M},\, \Omega,\, t\rangle \longmapsto \langle \mathtt{skip},\, \mathrm{M},\, \Omega \cup \{\mu\},\, t+1\rangle$ while $\mu = [\mathtt{exec}\,\langle c[\overline{v}],\, \eta\rangle :: t, pc, \mathcal{Q}, \overline{v_1}]$ and $\langle \overline{e},\, \mathrm{M}\rangle \Downarrow \overline{v_1}$. By Lemma 5.4.3, $\langle \lfloor \overline{e}\rfloor_i,\, M_i\rangle \Downarrow \lfloor \overline{v_1}\rfloor_i$. Therefore, $\langle s_i,\, M_i,\, \Omega_i,\, t_i\rangle \longmapsto \langle \mathtt{skip},\, M_i,\, \Omega_i \cup \{\mu_i\},\, t_i+1\rangle$, and $\mu_i = \lfloor \mu \rfloor_i$.

- s is $\mathtt{chmod}(c[\overline{v}],\, \eta,\, pc,\, \mathcal{Q},\, \ell)$ or $\mathtt{setvar}(\langle c[\overline{v}],\, \eta\rangle.z,\, v)$. By the same argument as in the previous case.

- s is $(s_1, \ldots, s_n)$. By Lemma 5.4.2, we have $\langle s,\, \mathrm{M},\, \Omega,\, t\rangle \longmapsto \langle s',\, \mathrm{M}',\, \Omega',\, t'\rangle$ such that $\lfloor \langle s',\, \mathrm{M}',\, \Omega',\, t'\rangle \rfloor_1 = \langle s'_1,\, M'_1,\, \Omega'_1,\, t'_1\rangle$. By (S13), $\lfloor \langle s',\, \mathrm{M}',\, \Omega',\, t'\rangle \rfloor_i = \langle s_i,\, M_i,\, \Omega_i,\, t_i\rangle$ for any i such that $i \neq 1$.

□

Lemma 5.4.5 (Adequacy). Suppose $E_i = \langle s_i,\, M_i,\, \Omega_i,\, t_i\rangle \longmapsto^* \langle s'_i,\, M'_i,\, \Omega'_i,\, t'_i\rangle$ for all i in $\{1, \ldots, n\}$, and there exists a DSR* configuration $\langle s,\, \mathrm{M},\, \Omega,\, t\rangle$ such that for all i, $\lfloor \langle s,\, \mathrm{M},\, \Omega,\, t\rangle \rfloor_i = \langle s_i,\, M_i,\, \Omega_i,\, t_i\rangle$. Then there exists $E = \langle s,\, \mathrm{M},\, \Omega,\, t\rangle \longmapsto^* \langle s',\, \mathrm{M}',\, \Omega',\, t'\rangle$ such that for any i, $\lfloor E\rfloor_i \preceq E_i$, and for some j, $\lfloor E \rfloor_j \approx E_j$.

Proof. By induction on the total length of E_1 through E_n. The base case is trivial. The lemma holds immediately if $\langle s_j,\, M_j,\, \Omega_j,\, t_j\rangle = \langle s'_j,\, M'_j,\, \Omega'_j,\, t'_j\rangle$ holds for some

j. Suppose for all i, $\langle s_i,\, M_i,\, \Omega_i,\, t_i\rangle \longmapsto \langle s''_i,\, M''_i,\, \Omega''_i,\, t''_i\rangle \longmapsto^* \langle s'_i,\, M',\, \Omega'_i,\, t'_i\rangle$. By Lemma 5.4.4, there exists $E' = \langle s,\, \mathrm{M},\, \Omega,\, t\rangle \longmapsto^* \langle s'',\, \mathrm{M}'',\, \Omega'',\, t''\rangle$ such that $\lfloor E\rfloor_i \preceq \langle s_i,\, M_i,\, \Omega_i,\, t_i\rangle \longmapsto \langle s''_i,\, M''_i,\, \Omega''_i,\, t''_i\rangle$, and for some j, $\lfloor E\rfloor_j \approx \langle s_j,\, M_j,\, \Omega_j,\, t_j\rangle \longmapsto \langle s''_j,\, M''_j,\, \Omega''_j,\, t''_j\rangle$. Let $E''_i = E_i - \lfloor E'\rfloor_i$. By induction, there exists a run $E'' = \langle s'',\, \mathrm{M}'',\, \Omega'',\, t''\rangle \longmapsto^* \langle s',\, \mathrm{M}',\, \Omega',\, t'\rangle$ such that $\lfloor E''\rfloor_i \preceq E''_i$ and for some j', $\lfloor E''\rfloor_{j'} \approx E''_{j'}$. Then $E = E', E''$ is a run satisfying the lemma. $\square$

Typing rules

$$\text{(BV1)}\qquad \frac{\Gamma \vdash v_i : \tau \qquad \neg\zeta(\tau)\ \texttt{or}\ \forall i.\, v_i = v \vee v_i = \texttt{none}}{\Gamma \vdash (v_1, \ldots, v_n) : \tau}$$

$$\text{(BV2)}\qquad \frac{\Gamma \vdash v_i : \tau \qquad \tau = \sigma@\mathcal{Q} \qquad K(v_1, \ldots, v_n)}{\Gamma \vdash (v_1, \ldots, v_n) : \tau}$$

$$\text{(BS)}\qquad \frac{\lfloor\Gamma\rfloor_i \,;P\,;\mathcal{Q}\,;\lfloor pc'\rfloor_i \vdash s_i : \lfloor\tau\rfloor_i \qquad \neg\zeta(pc')}{\Gamma\,;P\,;\mathcal{Q}\,;pc \vdash (s_1, \ldots, s_n) : \tau}$$

$$\text{(M-EXEC)}\qquad \frac{\begin{array}{c}\Gamma\,;P \vdash c[\overline{v}] : \texttt{reactor}\{pc',\, \overline{\pi \rhd z : \tau_1},\, \overline{\tau_2}\} \\ \vdash \overline{v_1} : \overline{\tau_1} \quad i \in \{1, \ldots, n\} \Rightarrow \neg\zeta(pc)\end{array}}{\Gamma\,;P \vdash [\texttt{exec}\,\langle c[\overline{v}],\, \eta\rangle :: pc, \overline{v_1}, \mathcal{Q}, t]_i}$$

$$\text{(M-CHMD)}\qquad \frac{\begin{array}{c}\Gamma\,;P \vdash c[\overline{v}] : \texttt{reactor}\{pc',\, \overline{\pi \rhd z : \tau_1},\, \overline{\tau_2}\} \\ \vdash \ell : \texttt{label}_{\ell'} \quad \neg\zeta(\ell') \\ i \in \{1, \ldots, n\} \Rightarrow \neg\zeta(pc)\end{array}}{\Gamma\,;P \vdash [\texttt{chmod}\,\langle c[\overline{v}],\, \eta\rangle :: pc, \ell, \mathcal{Q}, t]_i}$$

$$\text{(M-SETV)}\qquad \frac{\vdash c[\overline{v}].\eta_1 : \tau\ \texttt{var} \quad \vdash v_1 : \tau \quad i \in \{1, \ldots, n\} \Rightarrow \neg\zeta(pc)}{\Gamma\,;P \vdash [\texttt{setvar}\,\langle c[\overline{v}],\, \eta\rangle.z :: v_1, t]_i}$$

Figure 5.7: Typing rules of DSR*

The type system of DSR* is similar to that of Aimp* except for additional rules for bracket terms, which are shown in Figure 5.7.

Intuitively, bracket constructs capture the differences between DSR terms, and any effect of a bracket construct should not satisfy ζ. Let $\neg\zeta(x)$ denote that x does not satisfy ζ. Rule (BV1) says that a bracket value v is well-typed if its type satisfies $\neg\zeta$, or all the non-`none` components in v are equal, which implies that the components of v are consistent as `none` is consistent with any value. Rule (BV2) is used to check bracket values with located types that may satisfy ζ. The key insight is that versioned values with different timestamps may be consistent. Rule (BV2) relies on an abstract function $K(v_1, \ldots, v_n)$ to determine whether a bracket of versioned values can have a type satisfying ζ. In other words, the type system of DSR* is parameterized with K.

Rule (BS) says that a bracket statement $(s_1, \ldots, s_n)$ is well-typed if every s_i is well-typed with respect to a program counter label not satisfying ζ.

Rules (M-EXEC), (M-CHMD) and (M-SETV) introduce an additional premise: $i \in \{1, \ldots, n\} \Rightarrow \neg\zeta(pc)$, which says that if a message carries an index $i \in \{1, \ldots, n\}$, then $\zeta(pc)$ is not satisfied because the message must have been sent by a statement in a bracket.

In DSR*, a memory M is well-typed with respect to the typing assignment Γ, written $\Gamma \vdash \mathrm{M}$, if $\Gamma \vdash \mathrm{M}(m) : \Gamma(m)$ holds for any m in *dom*(M). If $\mathrm{M}[m] = (v_1@t_1, \ldots, v_n@t_n)$ and $\Gamma(m) = \sigma$, then $\mathrm{M}(m) = (v_1, \ldots, v_n)$. The message set Ω is well-typed with respect to Γ and P, written $\Gamma \,; P \vdash \Omega$, if any message μ in Ω is well-typed with respect to Γ and P.

An important constraint that ζ needs to satisfy is that $\neg\zeta(\ell)$ implies $\neg\zeta(\ell \sqcup \ell')$ for any ℓ'. The purpose of this constraint is best illustrated by an example. In DSR*, if expression e is evaluated to a bracket value $(v_1, \ldots, v_n)$, statement `if` e `then` s_1 `else` s_2 would be reduced to a bracket statement $(s'_1, \ldots, s'_n)$, where s'_i is either $\lfloor s_1 \rfloor_i$ or $\lfloor s_2 \rfloor_i$. To show $(s'_1, \ldots, s'_n)$ is well-typed, we need to show that each s'_i is well-typed under a program-counter label that satisfying $\neg\zeta$, and we can show it by using the constraint on ζ. Suppose e has type $\mathtt{int}_\ell$, then we know that s'_i is well-typed under the program counter label $pc \sqcup \ell$. Furthermore, $\neg\zeta(\ell)$ holds because the result of e is a bracket value. Thus, by the constraint that $\neg\zeta(\ell)$ implies $\neg\zeta(\ell \sqcup \ell')$, we have $\neg\zeta(pc \sqcup \ell)$.

Subject reduction

This section proves the subject reduction theorem of DSR*.

Lemma 5.4.6 (Expression subject reduction). Suppose $\Gamma\,;P\,;\mathcal{Q} \vdash e : \tau$, and $\Gamma \vdash \mathrm{M}$, and $\langle e,\, \mathrm{M}\rangle_i \Downarrow v$. Then $\Gamma\,;P\,;\mathcal{Q} \vdash v : \tau$.

Proof. By induction on the derivation of $\langle e,\, \mathrm{M}\rangle_i \Downarrow v$.

- Cases (E1). Since $\Gamma \vdash \mathrm{M}$, we have $\Gamma \vdash \mathrm{M}(m) : \tau$. According to rules (BV1) and (BV2), $\Gamma \vdash \lfloor \mathrm{M}(m) \rfloor_i : \tau$.
- Case (E2). By induction, $\Gamma\,;P \vdash v_i : \tau$ for $i \in \{1,2\}$, and τ is not a located type. If v_1 or v_2 is a bracket value, then τ satisfies *RV*() by rule (BV1), and thus we have $\Gamma\,;P \vdash v : \tau$ even though v is a bracket value. If neither v_1 nor v_2 is a bracket value, then v is not a bracket value either, which implies $\Gamma\,;P \vdash v : \tau$.
- Case (E3). Since e is v, we have $\Gamma\,;P\,;\mathcal{Q} \vdash v : \tau$, which implies $\Gamma\,;P\,;\mathcal{Q} \vdash \lfloor v \rfloor_i : \tau$.
- Case (E4). By the typing rule (DEREF), τ does not satisfy ζ. Therefore, we have $\Gamma\,;P\,;\mathcal{Q} \vdash (v_1, \ldots, v_n) : \tau$ by (BV1).

□

Theorem 5.4.1 (Subject reduction). Suppose $\Gamma\,;P\,;\mathcal{Q}\,;pc \vdash s : \tau$, and $\Gamma \vdash \mathrm{M}$, and $\Gamma\,;P \vdash \Omega$, and $\langle s,\, \mathrm{M},\, \Omega,\, t\rangle_i \longmapsto \langle s',\, \mathrm{M}',\, \Omega',\, t'\rangle_i$, and $i \in \{1, \ldots, n\}$ implies that $\neg\zeta(pc)$. Then $\Gamma\,;P\,;\mathcal{Q}\,;pc \vdash s' : \tau$, and $\Gamma \vdash \mathrm{M}'$, and $\Gamma\,;P \vdash \Omega'$.

Proof. By induction on the derivation step $\langle s,\, M,\, \Omega,\, t\rangle_i \longmapsto \langle s',\, M',\, \Omega',\, t'\rangle_i$.

- Case (S1). In this case, s is $m := e$; τ is $\texttt{stmt}_{pc}$; s' is $\texttt{skip}$. We have $\Gamma\,;P\,;\mathcal{Q}\,;pc \vdash \texttt{skip} : \texttt{stmt}_{pc}$. By (S1), M' is $\mathrm{M}[m \mapsto_i v@t]$. By Lemma 5.4.6, we have $\Gamma \vdash v : \Gamma(m)$. If i is $\bullet$, then $\mathrm{M}'(m)$ is v or $v@t$ according to $\Gamma(m)$, and in either case, the type of $\mathrm{M}'(m)$ is $\Gamma(m)$. Otherwise, $\neg\zeta(\Gamma(m))$ holds, and thus $\mathrm{M}'(m)$ has type $\Gamma(m)$ according to rule (BV1).

- Case (S2). By typing rule (SEQ), $\Gamma\,;P\,;\mathcal{Q}\,;pc \vdash s_1 : \mathtt{stmt}_{pc'}$ and $\Gamma\,;P\,;\mathcal{Q}\,;pc' \vdash s_2 : \mathtt{stmt}_{pc''}$. By induction, $\Gamma\,;P\,;\mathcal{Q}\,;pc \vdash s_1' : \mathtt{stmt}_{pc'}$. Therefore, $\Gamma\,;P\,;\mathcal{Q}\,;pc \vdash s_1';s_2 : \mathtt{stmt}_{pc''}$. By induction, $\Gamma \vdash \mathrm{M}'$ and $\Gamma\,;P \vdash \Omega'$.
- Case (S3). s is $\mathtt{skip};s'$. By rule (SEQ), $\Gamma\,;P\,;\mathcal{Q}\,;pc \vdash s' : \tau$.
- Case (S5). s is $\mathtt{if}\ e\,\mathtt{then}\,s_1\,\mathtt{else}\,s_2$. By typing rule (IF), $\Gamma\,;P\,;\mathcal{Q}\,;pc\sqcup\ell_e \vdash s_1 : \tau$, which implies $\Gamma\,;P\,;\mathcal{Q}\,;pc \vdash s_1 : \tau$.
- Case (S6). By the same argument as case (S5).
- Case (S7). In this case, s is $\mathtt{exec}(c[\overline{v}],\,\eta,\,pc,\,\mathcal{Q},\,\overline{e})$. By Lemma 5.4.6, $\Gamma\,;\mathcal{Q} \vdash \overline{v_1} : \overline{\tau_1}$, where $\overline{\tau_1}$ are the types of the corresponding arguments of $c[\overline{v}]$. Thus $\Gamma \vdash [\mathtt{exec}\,\langle c[\overline{v}],\,\eta\rangle :: pc, \overline{v_1}, \mathcal{Q}, t]$.
- Case (S8). By the same argument as case (S7).
- Case (S9). By Lemma 5.4.6.
- Case (S10). We have $\Gamma\,;P\,;\mathcal{Q}\,;pc \vdash \mathtt{skip} : \tau$.
- Case (S12). In this case, s is $\mathtt{if}\ e\,\mathtt{then}\,s_1\,\mathtt{else}\,s_2$ and $\langle e,\,\mathrm{M}\rangle \Downarrow (v_1,\ldots,v_n)$. By the typing rule (IF), $\Gamma\,;\mathcal{Q} \vdash e : \mathtt{int}_\ell$. By Lemma 5.4.6, $\Gamma\,;\mathcal{Q} \vdash (v_1,\ldots,v_n) : \mathtt{int}_\ell$. By the typing rule (BV1), we have $\neg\zeta(\ell)$, which implies $\neg\zeta(pc\sqcup\ell)$. Moreover, by rule (IF), $\Gamma\,;\mathcal{Q}\,;pc\sqcup\ell \vdash \lfloor s_j\rfloor_i : \tau$ for $i \in \{1,\ldots,n\}$ and $j \in \{1,2\}$. Therefore, by rule (BS), $\Gamma\,;\mathcal{Q}\,;pc \vdash s' : \tau$.
- Case (S13). By induction, $\Gamma \vdash \mathrm{M}'$ and $\Gamma\,;P \vdash \Omega'$, and $\Gamma\,;P\,;\mathcal{Q}\,;pc' \vdash s_i' : \tau$. Therefore, $\Gamma\,;P\,;\mathcal{Q}\,;pc \vdash s' : \tau$.
- Case (S14). s' is $(m_1 := \lfloor e\rfloor_1,\ldots,m_n := \lfloor e\rfloor_n)$. Suppose $\Gamma\,;P \vdash (m_1,\ldots,m_n) : (\mathtt{int}_\ell\ \mathtt{ref})_{\ell'}$. By (BV1), $\neg\zeta(\ell')$, which implies $\neg\zeta(\ell)$. As a result, $\Gamma\,;P\,;\mathcal{Q}\,;\ell \vdash s' : \tau$.

$\square$

5.4.3 Noninterference proof

Let Θ_0 represent the initial thread pool that is empty, and $\mathcal{E}_0$ represent the initial environment that contains only invocation messages for the starting reactor c (having no arguments) at time $t_0 = \langle\rangle$.

Lemma 5.4.7 (Noninterference). Suppose $\Gamma \Vdash P$, and $E_i = \langle \Theta_0,\, \mathcal{M}_i,\, \mathcal{E}_0 \rangle \longmapsto^* \langle \Theta'_i,\, \mathcal{M}'_i,\, \mathcal{E}'_i \rangle$ for $i \in \{1,2\}$ If $\Gamma\,;P \vdash \langle \Theta_0,\, \mathcal{M}_1,\, \mathcal{E}_0 \rangle \approx_\zeta \langle \Theta_0,\, \mathcal{M}_2,\, \mathcal{E}_0 \rangle$, then $\Gamma\,;P \vdash \langle \Theta'_1,\, \mathcal{M}'_1,\, \mathcal{E}'_1 \rangle \approx_\zeta \langle \Theta'_2,\, \mathcal{M}'_2,\, \mathcal{E}'_2 \rangle$.

Proof. By induction on the total length of E_1 and E_2: $|E_1| + |E_2|$. The base cases are trivial. Without loss of generality, suppose $|E_1| \le |E_2|$ and $\langle \Theta,\, \mathcal{M}_i,\, \mathcal{E} \rangle \longmapsto^* \langle \Theta''_i,\, \mathcal{M}''_i,\, \mathcal{E}''_i \rangle \longmapsto \langle \Theta'_i,\, \mathcal{M}'_i,\, \mathcal{E}'_i \rangle$ for $i \in \{1,2\}$. Let $\mathcal{T}'_i = \mathit{timestamps}(\Theta'_i)$ and $\mathcal{T}''_i = \mathit{timestamps}(\Theta''_i)$. By induction, $\Gamma\,;P \vdash \langle \Theta''_1,\, \mathcal{M}''_1,\, \mathcal{E}''_1 \rangle \approx_\zeta \langle \Theta'_2,\, \mathcal{M}'_2,\, \mathcal{E}'_2 \rangle$. Then we need to show that $\Gamma\,;P \vdash \langle \Theta'_1,\, \mathcal{M}'_1,\, \mathcal{E}'_1 \rangle \approx_\zeta \langle \Theta'_2,\, \mathcal{M}'_2,\, \mathcal{E}'_2 \rangle$ holds for all cases of $\langle \Theta''_1,\, \mathcal{M}''_1,\, \mathcal{E}''_1 \rangle \longmapsto \langle \Theta'_1,\, \mathcal{M}'_1,\, \mathcal{E}'_1 \rangle$:

- Case (G1). In this case, the evaluation step is derived from $\langle s,\, M''_1,\, \Omega''_1,\, t''_1 \rangle \longmapsto \langle s',\, M'_1,\, \Omega'_1,\, t'_1 \rangle$ on some host h_1. We need to show that the local state of h_1 in configuration $\langle \Theta'_1,\, \mathcal{M}'_1,\, \mathcal{E}'_1 \rangle$ is still ζ-consistent with the local state of any host h_2 in $\langle \Theta'_2,\, \mathcal{M}'_2,\, \mathcal{E}'_2 \rangle$.

  By examining rules (S1)–(S9), we only need to consider two cases: (1) $M''_1 = M'_1[m \mapsto_{t''_1} v]$, and $\zeta(m, h_i)$ holds for $i \in \{1,2\}$; (2) $\Omega''_1 = \Omega'_1 \cup \{\mu\}$, and $\zeta(\mu, h_i)$ holds for $i \in \{1,2\}$. Suppose one of the two cases occurs. If there exists no thread on h_2 at t'_1 in Θ'_2, then the evaluation step does not affect the ζ-consistency between the local states of h_1 and h_2. Otherwise, consider the local run of the thread at t'_1 on host h_i: $E'_i = \langle s_i,\, M_i,\, \emptyset,\, t \rangle \longmapsto^* \langle s'_i,\, M'_i,\, \Omega'_i,\, t'_i \rangle$

for $i \in \{1, 2\}$. By rule (TPE), the two local runs correspond to the same closure reference $\langle c[\overline{v}],\, \eta\rangle$. Then we can show that $s_i = s[\mathcal{A}'_i]$ and $\Gamma' \vdash \mathcal{A}'_1 \approx_\zeta \mathcal{A}'_2$, where Γ' is the local typing assignment for reactor $c[\overline{v}]$. By (M1), we have $\mathcal{A}'_i = \mathcal{A}_i[\overline{y} \mapsto \overline{v_i}][\texttt{cid} \mapsto \eta][\texttt{nid} \mapsto \mathit{hash}(t)]$, where $\mathcal{A}_i$ is the variable record in the corresponding closure, and $\overline{v_i}$ is the list of arguments in the invocation requests. By induction, $\Gamma' \vdash \mathcal{A}_1 \approx_\zeta \mathcal{A}_2$. If the type of any y_j satisfies the ζ condition, then the program counter labels of the corresponding invocation also satisfy ζ. Since P satisfies (RV3), the invocation messages are sent by threads with the same closure reference. By $\Gamma\,; P \vdash \langle\Theta''_1,\, \mathcal{M}''_1,\, \mathcal{E}''_1\rangle \approx_\zeta \langle\Theta'_2,\, \mathcal{M}'_2,\, \mathcal{E}'_2\rangle$, those messages are ζ-consistent, which implies that the arguments are ζ-consistent with respect to their types. Therefore, $\Gamma' \vdash \mathcal{A}'_1 \approx_\zeta \mathcal{A}'_2$.

In addition, we can show $\Gamma \vdash M_1 \approx_\zeta M_2$, which means that for any m in $\mathit{dom}(\Gamma)$, $\zeta(\Gamma(m))$ implies $M_1(m) \approx M_2(m)$. In fact, if $\Gamma(m) = \sigma@\mathcal{Q}$, by induction and (ME), we have $M_1(m) \approx M_2(m)$. If $\Gamma(m) = \sigma$, then it must be the case that $M_1[m] = M_2[m]$ or $M_j[m] = \texttt{none}$ for some $j \in \{1, 2\}$. Otherwise, there exists some thread updating m before time t such that this thread is completed in one execution but not in the other. This contradicts (TPE).

Then we can construct a DSR* configuration $\langle s,\, \mathrm{M},\, \emptyset,\, t\rangle$ such that $\lfloor s\rfloor_i = s_i$ and s and M are well-typed with the following K condition: $K(v_1@t_1, \ldots, v_n@t_n)$ is true if for any i, j, $v_i@t_i \approx v_j@t_j$. By Lemma 5.4.5, there exists $E' = \langle s,\, \mathrm{M},\, \emptyset,\, t\rangle \longmapsto^* \langle s',\, \mathrm{M}',\, \Omega',\, t'\rangle$ such that $\lfloor E'\rfloor_i = E'_i$ and $\lfloor E'\rfloor_j \preceq E'_j$ where $\{i, j\} = \{1, 2\}$. Without loss of generality, suppose $\lfloor E'\rfloor_1 = E'_1$ and $\lfloor E'\rfloor_2 \preceq E'_2$. Then there exists a configuration $\langle s''_2,\, M''_2,\, \Omega''_2,\, t''_2\rangle$ such that $\lfloor \mathrm{M}'\rfloor_2 = M''_2$ and $\lfloor\Omega'\rfloor_2 = \Omega''_2$ and $\lfloor t'\rfloor_2 = t''_2$. By Theorem 5.4.1, M' and Ω' are well-typed. Therefore, $\Gamma \vdash M'_1 \approx_\zeta M''_2$, and $\Omega'_1 \approx_\zeta \Omega''_2$. Moreover, the rest of E'_2 modifies the configuration at timestamps greater than t'_1. Thus, $\Gamma \vdash M'_1 \approx_\zeta M'_2$ and $\Gamma \vdash \Omega'_1 \approx_\zeta \Omega'_2$, which means that the local states of h_1 and h_2 are still consistent after this execution step.

- Case (M1). In this case, it is obvious that $\Gamma \vdash \langle \mathcal{M}_1', \mathcal{T}_1' \rangle \approx_\zeta \langle \mathcal{M}_2', \mathcal{T}_2' \rangle$ and $P \vdash \langle \mathcal{E}_1', \mathcal{T}_1' \rangle \approx_\zeta \langle \mathcal{E}_2', \mathcal{T}_2' \rangle$ by induction. Thus, the goal is to prove $t \vdash \Theta_1' \approx_\zeta \Theta_2'$, where t is $\mathit{min}(\mathit{max}(\mathcal{T}_1', \zeta),\ \mathit{max}(\mathcal{T}_2', \zeta))$. Suppose the newly created thread is $\theta = \langle s, h, t_1, c[\overline{v}], \eta \rangle$, and the program counter label of $c[\overline{v}]$ is pc, and $t_1' = \mathit{max}(\mathcal{T}_1'', \zeta)$. If $\neg\zeta(\mathit{pc}, h)$, then $\Gamma \vdash \langle \Theta_1',\ \mathcal{M}_1',\ \mathcal{E}_1' \rangle \approx_\zeta \langle \Theta_2',\ \mathcal{M}_2',\ \mathcal{E}_2' \rangle$ holds immediately by induction. So we focus on the case where $\zeta(\mathit{pc}, h)$ holds.

  If $t_1 < \mathit{inc}(t_1',\ \mathit{pc})$, then we need to prove that θ is not the only thread at time t_1. Suppose otherwise. By $t_1 < \mathit{inc}(t_1',\ \mathit{pc})$, θ is not invoked by the threads at t_1'. Let n be the number of ζ-threads with timestamps having different global parts in Θ_1''. Then $n-1$ different ζ-threads need to invoke n different ζ-threads. Therefore, threads at some time t_d need to invoke two threads with different timestamps, which means that different invocation messages satisfying the ζ condition are sent by the thread replicas at t_d. That contradicts $\Gamma\,;P \vdash \langle \Theta_1'',\ \mathcal{M}_1'',\ \mathcal{E}_1'' \rangle \approx_\zeta \langle \Theta_1'',\ \mathcal{M}_1'',\ \mathcal{E}_1'' \rangle$. Therefore, θ is not the only thread at t_1, and $t \vdash \Theta_1' \approx_\zeta \Theta_2'$ follows $t \vdash \Theta_1'' \approx_\zeta \Theta_2'$. In addition, θ is ζ-consistent with other threads at time t_1 because $\Gamma\,;P \vdash \langle \Theta_1'',\ \mathcal{M}_1'',\ \mathcal{E}_1'' \rangle \approx_\zeta \langle \Theta_1',\ \mathcal{M}_1',\ \mathcal{E}_1' \rangle$ holds by induction.

  If $t_1 = \mathit{inc}(t_1',\ \mathit{pc})$, by rule (M1), at least one quorum finishes executing the thread at t_1'. Suppose $\langle \Theta_2'',\ \mathcal{M}_2'',\ \mathcal{E}_2'' \rangle \longmapsto \langle \Theta_2',\ \mathcal{M}_2',\ \mathcal{E}_2' \rangle$. Let $t_2' = \mathit{timestamp}(\Theta_2'', \mathcal{E}_2'')$ and $t_2 = \mathit{timestamp}(\Theta_2', \mathcal{E}_2')$. If $t_2 \le t_1'$, then we have $t \vdash \Theta_1' \approx_\zeta \Theta_2'$ by $t \vdash \Theta_1'' \approx_\zeta \Theta_2'$. Similarly, if $t_1 \le t_2'$, we have $t \vdash \Theta_1' \approx_\zeta \Theta_2'$ by $t \vdash \Theta_1' \approx_\zeta \Theta_2''$. Now consider the case that $t_2' < t_1$ and $t_1' < t_2$. We can prove that $t_1' = t_2'$ and $t_1 = t_2$. Suppose $t_2' < t_1'$. By $t_1' \vdash \Theta_1'' \approx_\zeta \Theta_2'$, we have that any invariant thread in Θ_2'' has its counterpart in Θ_1'' and has a timestamp less than t_1'. But that contradicts $t_1' < t_2$. By the same argument, we can rule out the case of $t_1' < t_2'$. Therefore,

$t'_1 = t'_2$, which implies $t_1 = t_2$, and it is clear that $t_1 \vdash \Theta'_1 \approx_\zeta \Theta'_2$.

- Case (M2). By the same argument as case (M1).

- Case (M3). In this case, some variable in a closure is initialized. So our goal is to prove that the closure is still equivalent to its counterparts in E_2. Suppose $\mathcal{E}'_1 = \mathcal{E}''_1[\mathit{closure}(h_1, c[\overline{v}], \eta) \mapsto \langle c[\overline{v}], \eta, \ell, \mathcal{A}'_1[z \mapsto v], t', \mathtt{on}\rangle]$. Then we need to show that for any host h_2 in $\mathit{loc}(c[\overline{v}])$ such that $\zeta(c[\overline{v}], h_2)$, $P \vdash \mathcal{E}'_1.\mathit{closure}(h_1, c[\overline{v}], \eta) \approx_\zeta \mathcal{E}'_2.\mathit{closure}(h_2, c[\overline{v}], \eta)$. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be the argument maps in the two closures. Since $\mathcal{E}''_1$ and $\mathcal{E}'_2$ are equivalent, we only need to prove that $\zeta(\tau)$ implies $\mathcal{A}_1(z) \approx \mathcal{A}_2(z)$, where τ is the type of z.

  First, we prove that the ζ-messages used to initialize z have the same timestamp. Since P satisfies (RV1) and (RV2), the threads that first operate on $\langle c[\overline{v}],\, \eta\rangle.z$ correspond to either $\langle c',\, \eta'\rangle$, or $\langle c_1[\overline{v_1}],\, \eta_1\rangle$ with $\langle c[\overline{v}],\, \mathtt{nid}\rangle.z$ appearing in its code. In both cases, the timestamps of those threads are equal because $\langle \Theta''_1,\, \mathcal{M}''_1,\, \mathcal{E}''_1\rangle \approx_\zeta \langle \Theta'_2,\, \mathcal{M}'_2,\, \mathcal{E}'_2\rangle$, and the program counter labels of those threads are ζ-labels. Suppose two `setvar` messages for z have different timestamps. Then it must be the case that in the two runs, two reactor instances with the same timestamp send different messages containing $\langle c[\overline{v}],\, \eta\rangle.z$. By $\mathcal{E}''_1 \approx_\zeta \mathcal{E}'_2$, at least one of the reactor instances sends two different messages containing the remote variable. This contradicts with the fact that P satisfies (RV1). Therefore, the `setvar` messages for z have the same timestamp.

  If $\zeta(x)$ is $C(x) \leq l_{\mathtt{A}}$, then all the `setvar` message satisfy the ζ condition, and they are equivalent by $\Gamma \vdash \langle \Theta''_1,\, \mathcal{M}''_1,\, \mathcal{E}''_1\rangle \approx_\zeta \langle \Theta'_2,\, \mathcal{M}'_2,\, \mathcal{E}'_2\rangle$. Thus, the initial values of $\langle c[\overline{v}],\, \eta\rangle.z$ are equal in both runs.

  Suppose $\zeta(x)$ is $I(x) \not\leq l_{\mathtt{A}}$. Consider the message synthesizer π for z. There are two cases:

- π is $\texttt{LT}[I(\ell)]$. The `setvar` messages have the form $[\texttt{setvar}\,\langle c[\overline{v}],\,\eta\rangle.z :: v,t]$, and z has type $\texttt{int}_\ell$. Since $\Gamma \vdash \langle \Theta_1'',\, \mathcal{M}_1'',\, \mathcal{E}_1''\rangle \approx_\zeta \langle \Theta_2',\, \mathcal{M}_2',\, \mathcal{E}_2'\rangle$, those high-integrity messages are equivalent. Therefore, the values resulted from synthesizing the `setvar` messages are the same in both runs. Thus, $\mathcal{A}_1(z) \approx \mathcal{A}_2(z)$.

- π is $\texttt{QR}[\mathcal{Q}, I]$. Suppose the set of high-integrity senders are $h_1, \ldots, h_n$ in E_1 and $h_1', \ldots, h_k'$ in E_2, and the local memory snapshots for these hosts when executing the thread at t are $M_1, \ldots, M_n$ and $M_1', \ldots, M_k'$, respectively. Let M incorporate those local memories. By rule (TPE), we can show that M is well-typed with respect to the following K constraint:

$$\frac{\forall i.\ v_i = v \vee v_i = \texttt{none}}{(v_1, \ldots, v_n) \Downarrow v} \qquad \frac{\exists v_j@t_j.\ v_j@t_j = v@t \quad \forall i.\ t_i \le t}{(v_1@t_1, \ldots, v_n@t_n) \Downarrow v}$$

$$\frac{(v_1, \ldots, v_n) \Downarrow v \quad (v_1', \ldots, v_k') \Downarrow v}{K(v_1, \ldots, v_n, v_1', \ldots, v_k')}$$

  In addition, we can construct a DSR* statement s such that $\lfloor s \rfloor_i = s_i$ where $1 \le i \le n + k$. Then we have a well-typed DSR* configuration $\langle s,\, \mathrm{M},\, \emptyset,\, t\rangle$. By Lemma 5.4.5, $\langle s,\, \mathrm{M},\, \emptyset,\, t\rangle \longmapsto^* \langle s',\, \mathrm{M}',\, \Omega',\, t'\rangle$ and $\lfloor t' \rfloor_i \le t_i'$ and for some j, $\lfloor t' \rfloor_j = t_j'$. By Theorem 5.4.1, Ω' is well-typed, and the message $[\texttt{setvar}\,\langle c[\overline{v}],\,\eta\rangle.z :: v,t]$ in Ω' is also well-typed, which means that $v = (v_1, \ldots, v_n, v_1', \ldots, v_k')$ is well-typed. Furthermore, $K(v_1, \ldots, v_n, v_1', \ldots, v_k')$ implies that the `setvar` messages produced by $\texttt{QR}[\mathcal{Q}, I]$ contain the same initial value v. Therefore, $\mathcal{A}_1(z) = \mathcal{A}_2(z)$.

- Case (A1). For integrity, $\zeta(m,h)$ does not hold. Therefore, $\Gamma \vdash \langle \mathcal{M}_1', \mathcal{T}_1'\rangle \approx_\zeta \langle \mathcal{M}_2', \mathcal{T}_2'\rangle$ immediately follows $\Gamma \vdash \langle \mathcal{M}_1'', \mathcal{T}_1''\rangle \approx_\zeta \langle \mathcal{M}_2', \mathcal{T}_2'\rangle$. For confidentiality, we assume attackers would refrain from changing low-confidentiality data in this case.

- Case (A2). By the same argument as case (A1).

- Case (A3). In this case, some thread aborts. However, the timestamp of the thread remains unchanged, and the ζ-consistency between program states is not affected.

$\square$

Theorem 5.4.2 (Integrity Noninterference). Suppose $\Gamma \Vdash P$, and $\langle \Theta_0,\, \mathcal{M}_i,\, \mathcal{E}_0 \rangle \longmapsto^* \langle \Theta_i',\, \mathcal{M}_i',\, \mathcal{E}_i' \rangle$ for $i \in \{1, 2\}$. If $\Gamma\,;P \vdash \langle \Theta_0,\, \mathcal{M}_1,\, \mathcal{E}_0 \rangle \approx_{I \not\leq l_{\mathtt{A}}} \langle \Theta_0,\, \mathcal{M}_2,\, \mathcal{E}_0 \rangle$, then $\Gamma\,;P \vdash \langle \Theta_1',\, \mathcal{M}_1',\, \mathcal{E}_1' \rangle \approx_{I \not\leq l_{\mathtt{A}}} \langle \Theta_2',\, \mathcal{M}_2',\, \mathcal{E}_2' \rangle$.

Proof. Let $\zeta(\ell)$ be $I(\ell) \not\leq L$ and apply Lemma 5.4.7. $\square$

Theorem 5.4.3 (Confidentiality Noninterference). Suppose $\Gamma \Vdash P$, and for $i \in \{1, 2\}$, $\langle \Theta_0,\, \mathcal{M}_i,\, \mathcal{E}_0 \rangle \longmapsto^* \langle \Theta_i',\, \mathcal{M}_i',\, \mathcal{E}_i' \rangle$, and $\Gamma\,;P \vdash \langle \Theta_0,\, \mathcal{M}_1,\, \mathcal{E}_0 \rangle \approx_{C \leq l_{\mathtt{A}}} \langle \Theta_0,\, \mathcal{M}_2,\, \mathcal{E}_0 \rangle$. Then $\Gamma\,;P \vdash \langle \Theta_1',\, \mathcal{M}_1',\, \mathcal{E}_1' \rangle \approx_{C \leq l_{\mathtt{A}}} \langle \Theta_2',\, \mathcal{M}_2',\, \mathcal{E}_2' \rangle$.

Proof. Let $\zeta(\ell)$ be $C(\ell) \leq L$ and apply Lemma 5.4.7. $\square$

5.5 Related work

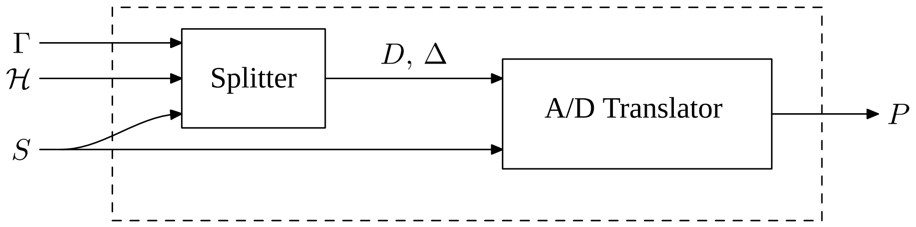
The related work on the language features of DSR is covered in Section 4.8, and the related work on proving a security type system enforces noninterference is covered in Section 3.6.

Following the approach of the $\lambda_{\mathtt{DSEC}}$ [106], the DSR type system uses dependent types to model dynamic labels. Other work [97, 96] has used dependent type systems to specify complex program invariants and to statically catch program errors considered run-time errors by traditional type systems.

Chapter 6
Security by construction

This chapter presents a program transformation that translates an Aimp program into a DSR program to be executed in a distributed system with untrusted hosts.

As shown in the following figure, the transformation generates a DSR program P from a typing assignment Γ, a trust configuration $\mathcal{H}$ that maps hosts to their labels, and an Aimp program S.



The transformation is a two-step process. First, Γ, $\mathcal{H}$ and S are fed to the *splitter*, which generates a *distribution scheme* D and a *label assignment* Δ. The distribution scheme specifies where the target code of source statements is replicated, and where memory references are replicated. The label assignment associates labels with source statements. The label of a statement specifies the security requirements for executing the statement and may be used to generate dynamic labels in the target program.

Second, the Aimp/DSR translator takes in the outputs (D and Δ) of the splitter and the source program S, and generates the target DSR program P.

6.1 Splitter

Given S, Γ and $\mathcal{H}$, the splitter partitions S into small program segments and determines where the target code of each program segment is replicated. Intuitively, it is easier to find a set of hosts that are trusted to run a small program segment than it is to find a set that can run the whole program. Based on this idea, the least restrictive way to partition

S is to treat every *non-sequence substatement* (substatement that is not a sequential composition) of S as a segment. For simplicity, the Aimp/DSR translation uses this partitioning approach. We assume that each non-sequence statement S in the source program is instrumented with a unique name c so that the corresponding segment can be easily identified. The instrumented statement is written as $\{c\}\,S$. The instrumentation does not affect the semantics of the source program.

A distribution scheme is formalized as a map from statement names to host sets and from memory references to quorum systems. Given a distribution scheme D, $D(m) = \mathcal{Q}$ requires that m is replicated on $\mathcal{Q}$; $D(c) = H$ requires that the target code of $\{c\}\,S$ is replicated on set H of hosts.

In general, a distribution scheme D needs to satisfy certain security constraints. For example, suppose $D(m) = \mathcal{Q}$, and m has type $\texttt{int}_{\ell}\ \texttt{ref}$. Then every host h in $\mathcal{Q}$ must satisfy $C(\ell) \leq C(h)$ in order to protect the confidentiality of m. Given a source program S, there may exist many valid distribution schemes that satisfy those security constraints. And it is up to the splitter to select one that helps generate efficient target code. Because the main concern of this work is security rather than performance, we do not consider the problem of selecting a scheme to increase performance. Instead, we focus on identifying security constraints sufficient to ensure that a given distribution scheme is able to enforce the security policies of the source.

6.1.1 Statement labels

How to replicate a statement depends on the label of the statement, which is derived from the program counter label and the labels of data processed by S. The label ℓ of a statement S has five components:

- C: an upper bound to the confidentiality label of any data used by S,
- I: an upper bound to the integrity label of any effect caused by S,

(L1) $$\frac{\Gamma\,;\mathcal{R}\vdash e:\texttt{int}_{\ell'}\qquad \Gamma\,;\mathcal{R}\vdash m:\texttt{int}_{\ell}\;\texttt{ref}}{\Gamma\,;\mathcal{R}\,;\mathcal{R}'\,;pc\vdash m:=e:\{C=C(\ell'),\,I=I(\ell),\,A=A(\mathcal{R}'),\,C_{\texttt{pc}}=C(pc),\,C_{\texttt{end}}=C(pc)\}}$$

(L2) $$\Gamma\,;\mathcal{R}\,;\mathcal{R}'\,;pc\vdash \texttt{skip}:\{C=\bot,\,I=\bot,\,A=A(\mathcal{R}'),\,C_{\texttt{pc}}=C(pc),\,C_{\texttt{end}}=C(pc)\}$$

(L3) $$\frac{\begin{array}{c}\Gamma\,;\mathcal{R}\vdash e:\texttt{int}_{\ell}\qquad \Gamma\,;\mathcal{R}\,;\mathcal{R}'\,;pc\sqcup\ell\vdash S_i:\ell_i\quad i\in\{1,2\}\\ \ell'=\{C=C(\ell),\,I=I(\ell_1)\sqcup I(\ell_2),\,A=A(\mathcal{R}'),\,C_{\texttt{pc}}=C(pc),\,C_{\texttt{end}}=C_{pc}(\ell_1)\sqcup C_{pc}(\ell_2)\}\end{array}}{\Gamma\,;\mathcal{R}\,;\mathcal{R}'\,;pc\vdash \texttt{if}\;e\;\texttt{then}\;S_1\;\texttt{else}\;S_2:\ell'}$$

(L4) $$\frac{\Gamma\,;\mathcal{R}\,;\mathcal{R}'\,;pc\vdash S_1:\ell_1\quad \Gamma\,;\mathcal{R}\,;\mathcal{R}'\,;pc\vdash S_2:\ell_2}{\Gamma\,;\mathcal{R}\,;\mathcal{R}'\,;pc\vdash S_1;S_2:\{C=C(\ell_1)\sqcup C(\ell_2),\,I=I(\ell_1)\sqcup I(\ell_2),\,A=A(\mathcal{R}'),\,C_{\texttt{pc}}=C(pc),\,C_{\texttt{end}}=C_{pc}(\ell_2)\}}$$

(L5) $$\frac{\Gamma\,;\mathcal{R}\,;\mathcal{R}'\,;pc\vdash e:\texttt{int}_{\ell}\qquad \Gamma\,;\mathcal{R}\,;\mathcal{R}'\,;pc\sqcup\ell\vdash S:\ell'}{\Gamma\,;\mathcal{R}\,;\mathcal{R}'\,;pc\vdash \texttt{while}\;e\;\texttt{do}\;S:\{C=C(\ell),\,I=I(\ell')\sqcup A(\mathcal{R}'),\,A=A(\mathcal{R}'),\,C_{\texttt{pc}}=C(pc),\,C_{\texttt{end}}=C(pc)\}}$$

(L6) $$\frac{\Gamma\,;\mathcal{R}\,;\mathcal{R}'\,;pc\vdash S:\ell\quad \ell\le\ell'}{\Gamma\,;\mathcal{R}\,;\mathcal{R}'\,;pc\vdash S:\ell}$$

Figure 6.1: Rules for inferring statement labels

- A: an upper bound to the availability label of any output reference that may still be unassigned after S terminates,
- $C_{\texttt{end}}$, the confidentiality label of the information that can be inferred by knowing the termination point of S, and
- $C_{\texttt{pc}}$, the confidentiality component of the program counter label of S.

The rules for inferring the label of a statement are shown in Figure 6.1. The judgment $\Gamma\,;\mathcal{R}\,;\mathcal{R}'\,;pc\vdash S:\ell$ means that S has a label ℓ while $\Gamma\,;\mathcal{R}\,;pc\vdash S:\texttt{stmt}_{\mathcal{R}'}$. In general, with respect to Γ, $\mathcal{R}$, $\mathcal{R}'$ and pc, the confidentiality label of a statement S is the join of the labels of data used by S; the integrity label of S is the join of the labels of effects caused by S; the availability label of S is $A_\Gamma(\mathcal{R}')$, simply written as $A(\mathcal{R}')$; the $C_{\texttt{pc}}$ label of S is $C(pc)$; the $C_{\texttt{end}}$ label is the join of the confidentiality components of the program counter labels at the program points where S may terminate.

In rule (L5), because the termination of the `while` statement depends on the integrity of e, the integrity label of S is $I(\ell')\sqcup A(\mathcal{R}')$.

Rule (L6) means that it is secure to assign a stronger than necessary security label to a statement. In practice, assigning a stronger integrity label to a statement helps generate more efficient control transfer code for that statement because of the extra integrity allows the hosts to perform more freely. A valid label assignment Δ satisfies $\Gamma\,;\mathcal{R}_c \vdash S' : \ell'$ and $\ell' \leq \Delta(c)$. for any statement $\{c\}\,S'$ appearing in the source program S.

We impose an additional constraint on Δ to help generate control transfer code. Suppose $\{c_1\}\,S_1$ and $\{c_2\}\,S_2$ are two statements in the source program S, and S_2 is a post-dominator of S_1 in the control flow graph of S, which means that every control path starting from S_1 leads to S_2. Let $l_1 = I(\Delta(c_1))$ and $l_2 = I(\Delta(c_2))$. In addition, suppose for any post-dominator $\{c'\}\,S'$ of S_1, if S' dominates S_2, then $l_1 \not\leq I(\Delta(c'))$. Then $l_1 \leq l_2$ or $l_2 \leq l_1$ is required to hold. Otherwise, it is difficult to construct the protocol for transferring control from S_1 to S_2. Suppose $l_1 \not\leq l_2$ and $l_2 \not\leq l_1$. Intuitively, by $l_1 \not\leq l_2$, the target code of S_1 needs to run a `chmod` statement to notify some following reactor at integrity level l_1 to expect invocation requests of integrity level $l_1 \sqcap l_2$. However, after running the `chmod` statement, the integrity level of control flow is lowered to $l_1 \sqcap l_2$, which makes it difficult to invoke the target code of S_2 because $l_2 \not\leq l_1 \sqcap l_2$.

6.1.2 Secure distribution schemes

Let $\mathcal{Q} \vDash \texttt{int}_\ell\ \texttt{ref}$ denote that it is secure to store memory references with type $\texttt{int}_\ell\ \texttt{ref}$ on $\mathcal{Q}$, and $D\,;\Delta\,;S \vDash \{c\}\,S'$ denote that it is safe to replicate the target code of $\{c\}\,S'$ on the host set $D(c)$ with respect to the distribution scheme D, the whole source program S, and the label assignment Δ. The following rules can be used to infer these two kinds of judgments:

$$\text{(DM)}\quad \frac{C(\ell) \leq C_{\sqcap}(\mathcal{Q}) \qquad A(\ell) \leq A_{\texttt{write}}(\mathcal{Q}) \sqcap A(|\mathcal{Q}|, \texttt{QR}[\mathcal{Q}, I(\ell)])}{\mathcal{Q} \vDash \texttt{int}_{\ell}\ \texttt{ref}}$$

$$\text{(DS)}\quad \frac{\begin{array}{c} \Delta(c) = \ell \quad D(c) = H \quad C(\ell) \leq C_{\sqcap}(H) \\ \{c_1\}\, S_1; \{c\}\, S' \in S \Rightarrow C_{\texttt{end}}(\Delta(c_1)) \leq C_{\sqcap}(H) \\ A(\ell) \leq A(H, \texttt{LT}[\ell]) \quad \forall m \in \mathit{UM}(S').\ C_{\texttt{pc}}(\ell) \leq C_{\sqcap}(D(m)) \\ \forall m \in \mathit{UM}(S').\ D(m) = h \Rightarrow D(c) = \{h\} \end{array}}{D\,;\Delta\,;S \vDash \{c\}\, S'}$$

In rule (DM), the first premise $C(\ell) \leq C_{\sqcap}(\mathcal{Q})$ guarantees that every host in $\mathcal{Q}$ is allowed to read the value of m. The second premise ensures that the availability of both the read and write operations on $\mathcal{Q}$ is as high as $A(\ell)$, while enforcing the integrity label $I(\ell)$.

In rule (DS), the premise $C(\ell) \leq C_{\sqcap}(H)$ says that H is allowed to see the data needed for executing S'. The second premise ensures that H is allowed to learn about the termination point of its predecessor $\{c_1\}\, S_1$, since hosts in H can infer the information from the invocation requests for c. In particular, if S' follows a conditional statement, H is allowed to learn which branch is taken. The premise $A(\ell) \leq A(H, \texttt{LT}[\ell])$ ensures that H can produce the outputs of S with sufficient integrity and availability. In addition, a distribution scheme also needs to prevent illegal implicit flows arising from memory accesses, including memory reads. Let $\mathit{UM}(S')$ be the set of references accessed by S'. Then for any m in $\mathit{UM}(S')$, on receiving an access request for m, hosts in $D(m)$ may be able to infer that control reaches that program point of $\{c\}\, S'$. Thus, the constraint $C_{\texttt{pc}}(\ell) \leq C_{\sqcap}(D(m))$ is imposed. The last premise says that if m appears in S, and m is not replicated, then D assigns m and S to the same host h so that the target code of S can simply access m in the local memory.

A distribution scheme D is secure, if for any m in $\mathit{dom}(\Gamma)$, $D(m) \vDash \texttt{int}_{\ell}\ \texttt{ref}$, and for any $\{c\}\, S'$ in S, $D\,;\Delta\,;S \vDash \{c\}\, S'$.

6.2 Aimp/DSR translator

The Aimp/DSR translator is formalized as a set of translation rules, which rely on a generic way of accessing remote replicated memory.

6.2.1 Remote memory accesses

If a memory reference m is replicated on multiple hosts, rule (DS) does not require a statement S that accesses m to be assigned to the hosts where m is replicated. Consequently, the target code of S may need to access memory references on remote hosts.

To support remote memory access, hosts storing a memory reference need to provide reactors to handle memory access requests. Using DSR, we can implement generic `read` and `write` reactors to handle remote memory reads and writes:

$$\begin{array}{l}
\texttt{read}[\texttt{lb}\!:\!\texttt{label}_{\texttt{lb}},\ \texttt{lm}\!:\!\texttt{label}_{\texttt{lb}},\ \texttt{m}\!:\!(\texttt{int}_{\texttt{lm}}@\&\texttt{m}\ \texttt{ref})_{\texttt{lb}},\\
\qquad \texttt{ret}\!:\!\texttt{reactor}\{\texttt{lb}\}_{\texttt{lb}},\ \texttt{rid}\!:\!\texttt{int}_{\texttt{lb}},\ \texttt{rv}\!:\!(\texttt{int}_{\texttt{lm}\sqcup\texttt{lb}}@\&\texttt{m}\ \texttt{var})_{\texttt{lb}}]\\
\{\ \texttt{lb},\ \#\texttt{m},\ \lambda.\ \texttt{setvar}(\texttt{rv},\ !\texttt{m});\ \texttt{exec}(\texttt{ret},\ \texttt{rid},\ \texttt{lb},\ \#\texttt{m},\ \epsilon)\ \ \}\\
\\
\texttt{write}[\texttt{lb}\!:\!\texttt{label}_{\texttt{lb}},\ \texttt{m}\!:\!(\texttt{int}_{\texttt{lb}}@\&\texttt{m}\ \texttt{ref})_{\texttt{lb}},\ \texttt{ret}\!:\!\texttt{reactor}\{\texttt{lb}\}_{\texttt{lb}},\ \texttt{rid}\!:\!\texttt{int}_{\texttt{lb}}]\\
\{\ \texttt{lb},\ \&\texttt{m},\ \ \lambda\,\texttt{val}\!:\!\texttt{int}_{\texttt{lb}}.\ \texttt{m} := \texttt{val};\ \texttt{exec}(\texttt{ret},\ \texttt{rid},\ \texttt{lb},\ \&\texttt{m},\ \epsilon)\ \ \}
\end{array}$$

To achieve genericity, both `read` and `write` reactors carry several reactor parameters. The `read` reactor has six parameters:

- `lb`, the program counter label of this reactor,
- `lm`, the label of the memory reference to be read,
- `m`, the memory reference to be read,
- `ret` and `rid`, specifying the closure $\langle\texttt{ret},\ \texttt{rid}\rangle$ for returning control to, and
- `rv`, the remote variable to receive the value of `m`.

The `read` reactor should be invoked on the hosts holding replicas of reference `m`, and the reactor does not update any reference. The code of the `read` reactor first sets the remote variable `rv` with the value of `m`, and then invokes $\langle\texttt{ret},\ \texttt{rid}\rangle$.

The `write` reactor has four parameters: `lb`, the program counter label of this reactor, `m`, the reference to write to, `ret` and `rid`, specifying the return closure $\langle \texttt{ret}, \texttt{rid} \rangle$. This reactor has one argument `val`, which is the value to be assigned to `m`. The code of the reactor is self-explanatory. Since the `write` reactor updates `m`, the `exec` statement to invoke $\langle \texttt{ret}, \texttt{rid} \rangle$ contains the term $\&\texttt{m}$, indicating that some reference on the quorum system $\&\texttt{m}$ is updated, and the update may still be unstable.

6.2.2 Translation rules

The syntax-directed translation rules are shown in Figure 6.2. Rules (TE1)–(TE5) are used to translate expressions; rules (TS1)–(TS6) are used to translate statements; rules (TC1) and (TC2) are used to generate control transfer code. All these translation rules use a translation environment $\langle D, \Delta, \Gamma' \rangle$ composed of a distribution scheme D, a label assignment Δ, and a typing assignment Γ', which is derived from D and the typing assignment Γ of the source: for any m in *dom*(Γ), if $D(m) = h$, then $\Gamma'(m) = \Gamma(m)$, otherwise $G'(m) = \Gamma'(m)@D(m)$.

The translation judgment for statements has the form $[\![S]\!]\Psi' = \langle P, \Psi \rangle$, meaning that an Aimp statement S is translated into a DSR program P in the translation context Ψ', which is a list of *program entries* of the target code of the rest part of the source program that follows S.

Program entries

In general, the target code P of an Aimp statement S needs to perform the computation of S and invoke the target code P' of the statement following S. On the surface, invoking P' means invoking the starting reactor c' of P'. However, c' may not have sufficient integrity to trigger all the computation of P'. Thus, P may be responsible for notifying (using `chmod` messages) the *entry reactors* of P' at different security levels.

$$\text{(TE1)}\quad \llbracket \eta \rrbracket\langle c, c', c_u, \ell, \mathcal{Q}\rangle = \eta \qquad \text{(TE2)}\quad \llbracket m \rrbracket\langle c, c', c_u, \ell, \mathcal{Q}\rangle = m \qquad \text{(TE3)}\quad \frac{\Gamma'(m) = \sigma}{\llbracket !m \rrbracket\langle c, c', c_u, \ell, \mathcal{Q}\rangle = !m}$$

$$\text{(TE4)}\quad \frac{\Gamma'(m) = \texttt{int}_{\ell_1}@\mathcal{Q}_m \quad r = c\{\ell,\ \mathcal{Q},\ \lambda.\texttt{exec}(\texttt{read}[\ell, \ell_1, m, c', \texttt{cid}, \langle c_u,\ \texttt{cid}\rangle.z],\ \texttt{nid},\ \ell,\ \mathcal{Q},\ \epsilon)\}}{\llbracket !m \rrbracket\langle c, c', c_u, \ell, \mathcal{Q}\rangle = \langle \{r\}, \lambda(\texttt{QR}[\mathcal{Q}_m, I(\ell_1)] \triangleright z : \texttt{int}_{\ell_1}).\ z\rangle}$$

$$\text{(TE5)}\quad \frac{\begin{array}{c}\llbracket e_1 \rrbracket\langle c, c_1, c_u, \ell, \mathcal{Q}\rangle = \langle P_1, \lambda\overline{\pi_1 \triangleright z_1 : \tau_1}.\ e_1'\rangle \quad \llbracket e_2 \rrbracket\langle c_1, c', c_u, \ell, \mathcal{Q}\rangle = \langle P_2, \lambda\overline{\pi_2 \triangleright z_2 : \tau_2}.\ e_2'\rangle \\ c_1 = (\mathit{if}\ P_2 \neq \emptyset\ \mathit{then}\ \mathit{new\text{-}reactor}(P_1,\ c)\ \mathit{else}\ c')\end{array}}{\llbracket e_1 + e_2 \rrbracket\langle c, c', c_u, \ell, \mathcal{Q}\rangle = \langle P_1 \cup P_2,\ \lambda\overline{\pi_1 \triangleright z_1 : \tau_1},\ \overline{\pi_2 \triangleright z_2 : \tau_2}.\ e_1' + e_2'\rangle}$$

$$\text{(TC1)}\quad \frac{\begin{array}{c}\Psi = \{\psi_1, \ldots, \psi_n\} \quad \ell_i = \mathit{label}(c_i)\ \ i \in \{1, ..., n\} \quad \ell_0 = \top \quad \ell_{n+1} = \bot \quad \ell_{j+1} \sqsubseteq \mathit{label}(c) \sqsubseteq \ell_j \\ w_{j+1} = w \quad \llbracket (c, w_{i+1}) \rrbracket\langle \ell_i, \psi_{i+1}\rangle = \langle s_i, w_i\rangle \quad i \in \{0, \ldots, j\}\end{array}}{\llbracket (c, w) \rrbracket \Psi = \langle s_j; \ldots; s_0, \{(c, w), \psi_{j+1}, \ldots, \psi_n\}\rangle}$$

$$\text{(TC2)}\quad \frac{\begin{array}{c}s = (\mathit{if}\ w' = c''.z\ \mathit{then}\ \texttt{setvar}(\langle c'',\ \texttt{nid}\rangle.z,\ w)\ \mathit{else}\ \texttt{skip}) \\ w'' = (\mathit{if}\ w' = c''.z\ \mathit{then}\ w\ \mathit{else}\ \texttt{nid}) \\ \ell' = \mathit{label}(c) \sqcup \mathit{label}(c') \quad s' = (\mathit{if}\ \ell = \top\ \mathit{then}\ \texttt{exec}(c',\ w'',\ \ell',\ \mathcal{Q},\ \epsilon)\ \mathit{else}\ \texttt{chmod}(c',\ w'',\ \ell',\ \mathcal{Q},\ \ell))\end{array}}{\llbracket (c, w) \rrbracket\langle \ell, (c', w')\rangle = \langle s; s',\ w''\rangle}$$

$$\text{(TS1)}\quad \frac{\begin{array}{c}\Delta\,; D \vdash c : \langle \ell, \mathcal{Q}\rangle \quad \Gamma'(m) = \sigma@\mathcal{Q}_m \quad \llbracket e \rrbracket\langle c, c_1, \ell, \mathcal{Q}\rangle = \langle P_e, \lambda\overline{\pi \triangleright z : \tau}.e'\rangle \quad c_1 = \mathit{new\text{-}reactor}(P_e,\ c) \\ r_1 = c_1\{\ell,\ \mathcal{Q},\ \overline{\pi \triangleright z : \tau},\ \lambda.\texttt{exec}(\texttt{write}[\ell, m, c_2, \texttt{cid}],\ \texttt{nid},\ \ell,\ \mathcal{Q},\ e')\} \quad \llbracket c \rrbracket\Psi = \langle s', \Psi'\rangle \quad r_2 = c_2\{\ell,\ \mathcal{Q},\ \lambda.s'\}\end{array}}{\llbracket \{c\}\, m := e \rrbracket\Psi = \langle P_e \cup \{r_1, r_2\}, \Psi'\rangle}$$

$$\text{(TS2)}\quad \frac{\begin{array}{c}\Delta\,; D \vdash c : \langle \ell, \mathcal{Q}\rangle \quad \Gamma'(m) = \sigma \quad \llbracket e \rrbracket\langle c, c_1, \ell, \mathcal{Q}\rangle = \langle P_e, \lambda\overline{\pi \triangleright z : \tau}.\ e'\rangle \\ c_1 = \mathit{new\text{-}reactor}(P_e,\ c) \quad \llbracket c \rrbracket\Psi = \langle s', \Psi'\rangle \quad r_1 = c_1\{\ell,\ \mathcal{Q},\ \overline{\pi \triangleright z : \tau},\ \lambda.\ m := e'; s'\}\end{array}}{\llbracket \{c\}\, m := e \rrbracket\Psi = \langle P_e \cup \{r_1\}, \Psi'\rangle}$$

$$\text{(TS3)}\quad \frac{\begin{array}{c}\Delta\,; D \vdash c : \langle \ell, \mathcal{Q}\rangle \quad \llbracket c \rrbracket\Psi = \langle s, \Psi'\rangle \\ r = c\{\ell,\ \mathcal{Q},\ \lambda.s\}\end{array}}{\llbracket \{c\}\,\texttt{skip} \rrbracket\Psi = \langle \{r\}, \Psi'\rangle} \qquad \text{(TS4)}\quad \frac{\begin{array}{c}\llbracket S_2 \rrbracket\Psi = \langle P_2, \Psi_2\rangle \\ \llbracket S_1 \rrbracket\Psi_2 = \langle P_1, \Psi_1\rangle\end{array}}{\llbracket S_1; S_2 \rrbracket\Psi = \langle P_1 \cup P_2, \Psi_1\rangle}$$

$$\text{(TS5)}\quad \frac{\begin{array}{c}\Delta\,; D \vdash c : \langle \ell, \mathcal{Q}\rangle \quad c_1 = \mathit{new\text{-}reactor}(P_e,\ c) \quad \llbracket S_i \rrbracket\Psi = \langle P_i, \Psi_i\rangle \ \ \llbracket c \rrbracket\Psi_i = \langle s_i', \Psi'\rangle \ \ i \in \{1, 2\} \\ \llbracket e \rrbracket\langle c, c_1, \ell, \mathcal{Q}\rangle = \langle P_e, \lambda\overline{\pi \triangleright z : \tau}.\ e'\rangle \quad r_1 = c_1\{\ell,\ \mathcal{Q},\ \overline{\pi \triangleright z : \tau},\ \lambda.\ \texttt{if}\ e'\ \texttt{then}\ s_1'\ \texttt{else}\ s_2'\}\end{array}}{\llbracket \{c\}\,\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \rrbracket\Psi = \langle P_e \cup P_1 \cup P_2 \cup \{r_1\}, \Psi'\rangle}$$

$$\text{(TS6)}\quad \frac{\begin{array}{c}\Delta\,; D \vdash c : \langle \ell, \mathcal{Q}\rangle \quad \llbracket e \rrbracket\langle c, c_1, \ell, \mathcal{Q}\rangle = \langle P_e, \lambda\overline{\pi \triangleright z : \tau}.e'\rangle \quad \llbracket S \rrbracket c = \langle P, \Psi_1\rangle \\ c_1 = \mathit{new\text{-}reactor}(P_e,\ c) \quad \llbracket \langle c_1,\ \texttt{nid}\rangle \rrbracket\Psi_1 = \langle s_1, \langle c_1, \texttt{nid}\rangle\rangle \quad \llbracket \langle c,\ c_1.z'\rangle \rrbracket\Psi = \langle s_2, \Psi'\rangle \\ r_1 = c_1\{\ell,\ \mathcal{Q},\ \overline{\pi \triangleright z : \tau},\ \texttt{LT}[\ell] \triangleright z' : \texttt{int}_\ell,\ \lambda.\ \texttt{if}\ e'\ \texttt{then}\ \texttt{setvar}(\langle c_1,\ \texttt{nid}\rangle.z',\ z');\ s_1\ \texttt{else}\ s_2\}\end{array}}{\llbracket \{c\}\,\texttt{while}\ e\ \texttt{do}\ S \rrbracket\Psi = \langle P_e \cup P \cup \{r_1\}, \Psi'\rangle}$$

Figure 6.2: Aimp/DSR Translation rules

An entry reactor c at security level ℓ is the reactor whose program counter label is ℓ and there is no other reactor in P' preceding c with a program counter label *pc* satisfying *pc* $\sqsubseteq \ell$. Formally, a program entry ψ has the form (c, w), where c is the reactor name of the entry, and w is a variable whose value is the context identifier used by c to invoke its next reactor. In most cases, w_i is `cid`, and thus c_i is used as an abbreviation for $(c_i, \texttt{cid})$. Let $\&P'$ represent the list of entries of P', which has the form $\Psi' = \psi_1, \ldots, \psi_n$ such that $\textit{label}(\psi_{i+1}) \sqsubseteq \textit{label}(\psi_i)$ holds for any $i \in \{1, \ldots, n\}$, where $\textit{label}((c, w)) = \textit{label}(c)$. Intuitively, ψ_1 through ψ_n are to be invoked in order, and for any reactor c'' to be invoked between ψ_i and ψ_{i+1}, the constraint $\textit{label}(\psi_i) \sqsubseteq \textit{label}(c'')$ is satisfied so that ψ_i has sufficient integrity to handle the invocation of c'' on its own. The translation of S depends on P', and thus is denoted by $[\![S]\!]\Psi' = \langle P, \Psi\rangle$, where Ψ should be the entries of $P \cup P'$.

Translating expressions

The translation of a source expression e generates a DSR expression e' that results in the same value as e does in the source program. In addition, the memory accesses in e might require invoking `read` reactors on remote hosts. Therefore, the translation result of e is composed of two parts: P, a distributed program that fetches the values of replicated memory references, and $\lambda\overline{\pi \triangleright z : \tau}.e'$, where e' computes the final value of e, and $\overline{z}$ are free variables of e', initialized by messages going through $\overline{\pi}$. The translation context of e is a five-element tuple $\langle c, c', c_u, \ell, \mathcal{Q}\rangle$, where c is the starting reactor of P, c' is the continuation reactor of P, c_u is the reactor that computes e', ℓ is the program counter label of reactor c, and $\mathcal{Q}$ is the quorum system where P is replicated.

Rules (TE1)–(TE3) translate constants and dereferences of non-replicated references, which remain the same after translation. In rules (TE1)–(TE3), there is no need to access remote references, and the translation result is just an expression. Rule (TE4) is used to translate $!m$ when m is replicated on multiple hosts. The target code invokes $\texttt{read}[\ell, \ell_1, m, c', \texttt{cid}, \langle c_u,\ \texttt{cid}\rangle.z]$, which initializes $\langle c_u,\ \texttt{cid}\rangle.z$ with the value of m and invokes $\langle c',\ \texttt{cid}\rangle$. Note that the `read` reactor is invoked with `nid` so that `read` requests issued by different reactors are distinguishable.

Rule (TE5) translates the addition expression $e_1 + e_2$. It combines the translations of e_1 and e_2 in a natural way. Suppose e_i is translated into $\langle P_i, \lambda\overline{\pi_i \triangleright z_i : \tau_i}.e'_i\rangle$ for $i \in \{1, 2\}$. Then $e_1 + e_2$ is translated into $\langle P_1 \cup P_2, \lambda\overline{\pi_1 \triangleright z_1 : \tau_1},\ \overline{\pi_2 \triangleright z_2 : \tau_2}.\ e'_1 + e'_2\rangle$. The tricky part is to figure out the translation contexts of e_1 and e_2. Expression e_1 is computed first, so P_1 is executed before P_2. Therefore, c is the entry of P_1, c' is the successor of P_2, and both the entry of P_2 and the successor of P_1 are some reactor c_1. In general, c_1 is a fresh reactor name. However, there are two exceptions. First, P_2 is empty. Second, P_2 is not empty, but P_1 is empty. In the first exception, c' is the successor of P_1, and thus $c_1 = c'$. In the second exception, c is the entry of P_2, and $c_1 = c$. Putting it all together, c_1 is computed by the formula $(\textit{if}\ P_2 \neq \emptyset\ \textit{then new-reactor}(P_1,\ c)\ \textit{else}\ c')$.

Translating entries

Rules (TC1) and (TC2) generate the code for c to invoke Ψ with the context identifier w. It can be viewed as translating (c, w) in the context Ψ. The translation result is a tuple $\langle s, \Psi'\rangle$ where s is the control transfer code, and Ψ' is the entries of the computation starting with c. In practice, c can also invoke a reactor c' that has the same security level as c, and let c' run s to invoke Ψ.

Suppose $\Psi = \psi_1, \ldots, \psi_n$, and $\ell_i = \textit{label}(\psi_i)$ for $i \in \{1, \ldots, n\}$, $\ell_0 = \top$, and $\ell_{n+1} = \bot$. If $\ell_{j+1} \sqsubseteq \textit{label}(c) \sqsubseteq \ell_j$, then c is able to invoke $\psi_1, \ldots, \psi_j$, and Ψ' is $\{(c, w), \psi_{j+1}, \ldots, \psi_n\}$. Now the only remaining task is to generate the code for invoking $\psi_j, \ldots, \psi_1$ in order.

Let $[\![\langle c,\ w_{i+1}\rangle]\!]\langle \ell_i, \psi_{i+1}\rangle = \langle s_i, w_i\rangle$ denote that s_i is the code to invoke ψ_{i+1} with context identifier w_{i+1} and program counter label ℓ_i, and w_i is the context identifier to be used after executing s_i. Then $s_j; \ldots; s_0$ is the code to invoke Ψ.

Rule (TC2) is used to compute $[\![(c, w)]\!]\langle \ell, (c', w')\rangle$. The translation depends on whether w' is some remote variable $c''.z$ and whether ℓ is $\top$. If $w' = c''.z$, then the translation includes a `setvar` statement to initialize $\langle c'', \texttt{nid}\rangle.z$ with w so that c'' can invoke the following computation with the context identifier w. Moreover, after executing the `setvar` statement, c needs to invoke or notify other entries with `nid`, and thus w'' is set to `nid`. If ℓ is $\top$, it means that $\langle c', w'\rangle$ is to be invoked directly, and thus the translation includes an `exec` statement to invoke c'. Otherwise, the target code includes a `chmod` statement, which changes the access control label of $\langle c', w''\rangle$ to ℓ.

Translating statements

Rules (TS1)–(TS6) are used to translate statements. Notation $\Delta\,; D \vdash c : \langle \ell, \mathcal{Q}\rangle$ means that ℓ and $\mathcal{Q}$ are the program counter label and the location of reactor c. Formally, $D(c) = \mathcal{Q}$, and $\ell = \{C = C_{\texttt{pc}}(\ell'),\ I = I(\ell'),\ A = A(\ell')\}$, where $\ell' = \Delta(c)$. The rules use a function *new-reactor*(P, c), which is a fresh reactor name unless P is empty, in which case it is c.

Rule (TS1) is used to translate $\{c\}\, m := e$ when $\Gamma(m) = \sigma@\mathcal{Q}_m$. Since m is replicated on $\mathcal{Q}$, the assignment is done by invoking the `write` reactors on $\mathcal{Q}$. The reactor $\texttt{write}[\ell, m, c_2, \texttt{cid}]$ updates m and then invokes $\langle c_2, \texttt{cid}\rangle$. The reactor c_2 contains the code to invoke Ψ with `cid`. The value of e is computed by P_e and $\lambda\overline{\pi \triangleright z : \tau}.e'$. Reactor c is the entry of P_e. Reactor c_1 computes e' and issues the `write` requests. Thus, c_1 contains $\overline{\pi \triangleright z : \tau}$ as its variables. Therefore, the translation context of e is $\langle c, c_1, \ell, H\rangle$, which is an abbreviation for $\langle c, c_1, c_1, \ell, H\rangle$. Note that if P_e is empty, then c_1 is the entry of the translation, and $c_1 = c$.

Rule (TS2) translates $\{c\}\, m := e$ when $\Gamma(m) = \sigma$. Expression e is translated in the same way as in rule (TS1). Since m is not replicated, $m := e$ is simply translated into $m := e'$, followed by the code for invoking Ψ.

Rule (TS3) translates the skip statement. Since `skip` does nothing, the translation only needs to generate code to invoke Ψ.

Rule (TS4) translates the sequential statement $S_1; S_2$. First, S_2 is translated into $\langle P_2, \Psi_2\rangle$ with respect to Ψ. Then, S_1 is translated in the context Ψ_2. The target code of $S_1; S_2$ is the union of the target code of S_1 and S_2.

Rule (TS5) is used to translate conditional statements. Expression e is translated in the same way as in rule (TS1). Reactor c_1 computes e' and executes the conditional statement to determine which branch to take and invoke the target code of that branch. The two branches S_1 and S_2 have the same continuation. Therefore, S_1 and S_2 are translated in the same context Ψ, and the translation results are $\langle P_1, \Psi_1\rangle$ and $\langle P_2, \Psi_2\rangle$. Then reactor c_1 needs to invoke Ψ_1 if e' is evaluated to a positive value, and Ψ_2 if otherwise. The control transfer code is generated by $[\![c]\!]\Psi_i$. Note that *label*(c) is a lower bound to the security label of any reactor in P_1 and P_2 because it affects whether these reactors are invoked. As a result, $[\![c]\!]\Psi_1$ and $[\![c]\!]\Psi_2$ generate the same initial entries Ψ'.

Rule (TS6) translates `while` statements. Expression e is translated in the same way as in rule (TS1). Implementing a loop, the target code of a `while` statement may be invoked multiple times, and each invocation needs to have a different context identifier so that it would not be confused with other invocations. When the loop terminates, Ψ needs to be invoked with the same context identifier w regardless of the number of iterations. Thus, w cannot be `cid` or `nid`, which changes in each iteration. Therefore, the context identifier used to invoke Ψ is the variable z' of reactor c_1, which computes e' and determines whether to enter the loop body or to invoke Ψ with z'. The code for entering the loop body starts with $\texttt{setvar}(\langle c_1,\ \texttt{nid}\rangle.z',\ z')$ so that z' is initialized with the same value in every iteration. The loop body S is translated with respect to c, because control is returned to c after the loop body terminates. The premise $[\![S]\!]c = \langle P, \Psi_1\rangle$ says that the entries of the target code of S is Ψ_1. Therefore, c_1 needs to invoke Ψ_1 with `nid` if the value of e' is positive. And the control transfer code is generated by $[\![\langle c_1,\ \texttt{nid}\rangle]\!]\Psi_1$.

6.3 Example

Consider the Aimp program in Figure 3.3 with the following typing assignment as discussed in Section 2.4:

$$\texttt{bid}, \texttt{offer}, \texttt{t}, \texttt{a}, \texttt{result} : \texttt{int}_{\ell_0} \qquad \texttt{acct} : \texttt{int}_{\ell_1}$$

where

$\ell_0 = \{C = \texttt{A}\wedge\texttt{B}:\texttt{A}\vee\texttt{B},\ I = \texttt{A}\wedge\texttt{B}:(\texttt{A}\wedge\texttt{B})\vee(\texttt{B}\wedge\texttt{T})\vee(\texttt{A}\wedge\texttt{T}),\ A = l\}$

$\ell_1 = \{C = \texttt{A}:\texttt{A},\ I = \texttt{A}:\texttt{A}\vee(\texttt{B}\wedge\texttt{T}),\ A = l\}$

$l = \texttt{A}\wedge\texttt{B}:(\texttt{A}\wedge\texttt{B})\vee(\texttt{B}\wedge\texttt{T})\vee(\texttt{A}\wedge\texttt{T})\vee(\texttt{C1}\wedge\texttt{C2})\vee(\texttt{C1}\wedge\texttt{C3})\vee(\texttt{C2}\wedge\texttt{C3})$

Suppose the bidding application is to be executed in a distributed system composed of a host h_P and three clusters of hosts: C1, C2 and C3. For all $i \in \{1, 2, 3\}$, cluster Ci contains hosts h_{iA}, h_{iB} and h_{iT} with integrity labels $\texttt{A}\wedge\texttt{B} : \texttt{A}$, $\texttt{A}\wedge\texttt{B} : \texttt{B}$ and $\texttt{A}\wedge\texttt{B} : \texttt{T}$, respectively. Hosts in cluster $\texttt{C}_i$ has an availability label $\texttt{A}\wedge\texttt{B} : \texttt{C}i$. All the hosts in the three clusters have a confidentiality label $\texttt{A}\wedge\texttt{B}:\texttt{A}\vee\texttt{B}$. The label of h_P is ℓ_1. Based on this trust configuration, acct can be located on h_P, and bid, offer, t, a and result can be replicated on the following quorum system

$$\mathcal{Q} = \langle\{h_{1A}, h_{1B}, h_{1T}, h_{2A}, h_{2B}, h_{2T}, h_{3A}, h_{3B}, h_{3T}\},\ W_{12},\ W_{13},\ W_{23}\rangle$$

where $W_{ij} = \{h_{iA}, h_{iB}, h_{iT}, h_{jA}, h_{jB}, h_{jT}\}$. It is easy to verify that $\mathcal{Q} \models \texttt{int}_{\ell_0}\ \texttt{ref}$. The non-sequence statements of the bidding program has an integrity label $I(\ell_0)$, except for $\texttt{acct} := !\texttt{acct}+!\texttt{bid}$ whose integrity label is $I(\ell_1)$. As a result, the target code of $\texttt{acct} := !\texttt{acct}+!\texttt{bid}$ can be distributed to h_P, while the target code of other statements can be replicated on the host set $H = \{h_{1A}, h_{1B}, h_{1T}, h_{2A}, h_{2B}, h_{2T}\}$, which satisfies $A(\ell_0) \leq A(H, \texttt{LT}[\ell_0])$.

With the distribution scheme just described, the source program in Figure 3.3 can be translated into a DSR program shown in Figure 6.3. For convenience, the reactor names are based on the line numbers of the corresponding source statements. For example, the while statement in line 2 of the source program is translated into reactors

$\texttt{line1}\{\ell_0,\, H,\, \lambda.\texttt{exec}(\texttt{write}[\ell_0, \texttt{t}, \texttt{line1a}, \texttt{cid}],\, \texttt{nid},\, \ell_0,\, H,\, 0)\}$

$\texttt{line1a}\{\ell_0,\, H,\, \lambda.\texttt{exec}(\texttt{write}[\ell_0, \texttt{a}, \texttt{line1b}, \texttt{cid}],\, \texttt{nid},\, \ell_0,\, H,\, -1)\}$

$\texttt{line1b}\{\ell_0,\, H,\, \lambda.\texttt{setvar}(\langle\texttt{line2a, nid}\rangle.z',\, \texttt{cid});\ \texttt{exec}(\texttt{line2},\, \texttt{nid},\, \ell_0,\, H,\, \epsilon)\ \}$

$\texttt{line2}\{\ell_0,\, H,\, \lambda.\texttt{exec}(\texttt{read}[\ell_0, \ell_0, \texttt{t}, \texttt{line2a}, \texttt{cid}, \langle\texttt{line2a, cid}\rangle.z],\, \texttt{nid},\, \ell_0,\, H,\, \epsilon)\}$

$\texttt{line2a}\{\ell_0,\, H,\, \texttt{QR}[\mathcal{Q}, I(\ell)] \triangleright z : \texttt{int}_{\ell_0},\, \texttt{LT}[\ell] \triangleright z' : \texttt{int}_{\ell_0},$
$\quad\quad \lambda.\texttt{if}\ z < 3\ \texttt{then}\ \texttt{setvar}(\langle\texttt{line2a, nid}\rangle.z',\, z');\ \texttt{exec}(\texttt{line3},\, \texttt{nid},\, \ell_0,\, H,\, \epsilon)$
$\quad\quad\quad \texttt{else}\ \texttt{exec}(\texttt{line7},\, z',\, \ell_0,\, H,\, \epsilon)\ \}$

$\texttt{line3}\{\ell_0,\, H,\, \lambda.\texttt{exec}(\texttt{read}[\ell_0, \ell_0, \texttt{bid}, \texttt{line3a}, \texttt{cid}, \langle\texttt{line3b, cid}\rangle.z_1],\, \texttt{nid},\, \ell_0,\, H,\, \epsilon)\}$

$\texttt{line3a}\{\ell_0,\, H,\, \lambda.\texttt{exec}(\texttt{read}[\ell_0, \ell_0, \texttt{offer}_\texttt{t}, \texttt{line3b}, \texttt{cid}, \langle\texttt{line3b, cid}\rangle.z_2],\, \texttt{nid},\, \ell_0,\, H,\, \epsilon)\}$

$\texttt{line3b}\{\ell_0,\, H,\, \texttt{QR}[\mathcal{Q}, I(\ell)] \triangleright z_1 : \texttt{int}_{\ell_0},\, \texttt{QR}[\mathcal{Q}, I(\ell)] \triangleright z_2 : \texttt{int}_{\ell_0},$
$\quad\quad \lambda.\texttt{if}\ z_1 \geq z_2\ \texttt{then}\ \texttt{chmod}(\texttt{line4b},\, \texttt{cid},\, \ell_0,\, H,\, \ell_1);\ \texttt{exec}(\texttt{line4},\, \texttt{cid},\, \ell_1,\, H,\, \epsilon)$
$\quad\quad\quad \texttt{else}\ \texttt{exec}(\texttt{line6},\, \texttt{cid},\, \ell_1,\, H,\, \epsilon)\}$

$\texttt{line4}\{\ell_1,\, h_P,\, \lambda.\texttt{exec}(\texttt{read}[\ell_1, \ell_0, \texttt{bid}, \texttt{line4a}, \texttt{cid}, \langle\texttt{line4b, cid}\rangle.z],\, \texttt{nid},\, \ell_1,\, H,\, \epsilon)\}$

$\texttt{line4a}\{\ell_1,\, h_P,\, \texttt{QR}[\mathcal{Q}, I(\ell)] \triangleright z : \texttt{int}_{\ell_0},\, \lambda.\texttt{acct} := !\texttt{acct} + z;\ \texttt{exec}(\texttt{line4b},\, \texttt{cid},\, \ell_1,\, h_P,\, \epsilon)\ \}$

$\texttt{line4b}\{\ell_0,\, H,\, \lambda.\texttt{exec}(\texttt{read}[\ell_0, \ell_0, \texttt{t}, \texttt{line4c}, \texttt{cid}, \langle\texttt{line4c, cid}\rangle.z],\, \texttt{nid},\, \ell_0,\, H,\, \epsilon)\}$

$\texttt{line4c}\{\ell_0,\, H,\, \texttt{QR}[\mathcal{Q}, I(\ell)] \triangleright z : \texttt{int}_{\ell_0},\, \lambda.\texttt{exec}(\texttt{write}[\ell_0, \texttt{a}, \texttt{line5}, \texttt{cid}],\, \texttt{nid},\, \ell_0,\, H,\, z)\}$

$\texttt{line5}\{\ell_0,\, H,\, \lambda.\texttt{exec}(\texttt{write}[\ell_0, \texttt{t}, \texttt{line3}, \texttt{cid}],\, \texttt{nid},\, \ell_0,\, H,\, 5)\}$

$\texttt{line6}\{\ell_0,\, H,\, \lambda.\texttt{exec}(\texttt{read}[\ell_0, \ell_0, \texttt{t}, \texttt{line6a}, \texttt{cid}, \langle\texttt{line6a, cid}\rangle.z],\, \texttt{nid},\, \ell_0,\, H,\, \epsilon)\}$

$\texttt{line6a}\{\ell_0,\, H,\, \texttt{QR}[\mathcal{Q}, I(\ell)] \triangleright z : \texttt{int}_{\ell_0},\, \lambda.\texttt{exec}(\texttt{write}[\ell_0, \texttt{a}, \texttt{line2}, \texttt{cid}],\, \texttt{nid},\, \ell_0,\, H,\, z + 1)\}$

$\texttt{line7}\{\ell_0,\, H,\, \lambda.\texttt{exec}(\texttt{read}[\ell_0, \ell_0, \texttt{a}, \texttt{line7a}, \texttt{cid}, \langle\texttt{line6a, cid}\rangle.z],\, \texttt{nid},\, \ell_0,\, H,\, \epsilon)\}$

$\texttt{line7a}\{\ell_0,\, H,\, \texttt{QR}[\mathcal{Q}, I(\ell)] \triangleright z : \texttt{int}_{\ell_0},\, \lambda.\texttt{exec}(\texttt{write}[\ell_0, \texttt{result}, \texttt{exit}, \texttt{cid}],\, \texttt{nid},\, \ell_0,\, H,\, z)\}$

$\texttt{exit}\{\ell_0,\, H,\, \lambda.\texttt{skip}\}$

Figure 6.3: The target DSR code of the bidding example

`line2` and `line2a` using rule (TS6) of Figure 6.2. Reactor `line2` invokes a `read` reactor on $\mathcal{Q}$, which initializes $\langle\texttt{line2a, cid}\rangle.z$ with the value of t and invokes `line2a`. Once invoked, reactor `line2a` executes a conditional statement with the guard expression $z < 3$, where z has the value of t. If z is not less than 3, then reactor `line2a` invokes $\langle \texttt{line7},\ z' \rangle$, where z' is the context identifier of reactor $\texttt{line1b}$, which invokes $\langle \texttt{line2},\ \texttt{nid} \rangle$ after initializing $\langle \texttt{line2a},\ \texttt{nid} \rangle.z'$ with $\texttt{cid}$. If z is less than 3, then reactor $\texttt{line2a}$ invokes $\langle \texttt{line3},\ \texttt{nid} \rangle$ after recursively initializing $\langle \texttt{line2a},\ \texttt{nid} \rangle.z'$ with z'. The target code is not very efficient, and there is much potential for optimization, which is left for future work.

6.4 Typing preservation

The DSR language relies on static typing to enforce security. Therefore, the Aimp/DSR translation needs to produce well-typed target programs. This is guaranteed by the typing preservation theorem (Theorem 6.4.1), which roughly says that the target code of a well-typed source program is a well-typed program in DSR.

Definition 6.4.1 (Well-formed entry list). An entry list Ψ is well-formed with respect to P, written $P \vDash \Psi$, if the following two conditions hold. First, for any entry (c, w) in Ψ, $P(c) = c[\overline{x:\sigma}]\{pc,\ \mathcal{Q},\ \overline{\pi \triangleright z:\tau},\ \lambda.s\}$, and if $w = c'.z$, then $P \vdash \langle c',\ \texttt{cid} \rangle.z : (\texttt{int}_\ell\ \texttt{var})_{\ell'}$. Second, if $\Psi = (c_1, w_1), \ldots, (c_n, w_n)$, then $\mathit{label}(\psi_{i+1}) \sqsubseteq \mathit{label}(\psi_i)$ holds for any $i \in \{1, \ldots, n\}$, where $\mathit{label}((c_i, w_i)) = \mathit{label}(c_i)$.

Lemma 6.4.1 (Control transfer typing soundness). Suppose P is the target code of an Aimp program under the translation environment $\langle \Gamma, \Delta, D \rangle$, and $\Delta\,; D \vdash c : \langle pc, \mathcal{Q} \rangle$, and $P \vDash \Psi$, and $[\![(c, w)]\!] \Psi = \langle s_c, \Psi' \rangle$. Then $\Gamma, w : \texttt{int}_{pc}, \texttt{nid} : \texttt{int}_{pc}\,; P\,; \mathcal{Q}\,; pc \vdash s_c : \tau$, and $P \vDash \Psi'$.

Proof. Let $\Gamma' = \Gamma, w : \texttt{int}_{pc}, \texttt{nid} : \texttt{int}_{pc}$ Suppose $\Psi = \psi_1, \ldots, \psi_n$. By (TC1), s_c is $s_j, \ldots, s_0$, where $[\![(c, w_{i+1})]\!] \langle \ell_i, \psi_{i+1} \rangle = \langle s_i, w_i \rangle$. By (TC2), $s_i = s; s'$. Statement s is $\texttt{setvar}(\langle c'',\ \texttt{nid} \rangle.z, w_{i+1})$ or $\texttt{skip}$. In either case, $\Gamma'\,; P\,; \mathcal{Q}\,; pc \vdash s : \texttt{stmt}_\perp$. Statement s' is $\texttt{exec}(c_1,\ w'',\ \ell',\ \mathcal{Q},\ \epsilon)$ if $i = 0$. Otherwise s' is $\texttt{chmod}(c_{i+1},\ w'',\ \ell',\ \mathcal{Q},\ \ell_i)$, where $\ell' = \textit{label}(c) \sqcup \textit{label}(c_{i+1})$. In the first case, $\Gamma' ; P ; \mathcal{Q} ; \ell' \vdash s' : \texttt{stmt}_{\bot}$. In the second case, $\Gamma' ; P ; \mathcal{Q} ; \ell' \vdash s' : \texttt{stmt}_{\ell_i}$. Therefore, we have $\Gamma' ; P ; \mathcal{Q} ; \ell' \vdash s; s' : \texttt{stmt}_{\ell_i}$, and ℓ' is ℓ_i if $0 \leq i \leq j-1$, and ℓ' is $\textit{pc}$ if $i = j$. By the typing rule (SEQ), $\Gamma' ; P ; \mathcal{Q} ; \textit{pc} \vdash s_j ; \ldots, s_0 : \tau$. □

Lemma 6.4.2 (Typing preservation). Suppose $[\![\Gamma]\!]D = \Gamma'$, and P' is the target code of an Aimp program S'. If e is an expression in S', and $\Gamma ; \mathcal{R} ; \textit{pc} \vdash e : \tau$, and $[\![e]\!]\langle c, c', c_u, \ell, \mathcal{Q}\rangle = \langle P, \lambda\overline{\pi \triangleright z : \tau}.e'\rangle$, and $P' \vDash c, c'$, then $\Gamma' ; P' \vdash P$ and $\Gamma', \overline{z : \tau} ; \mathcal{Q} \vdash e' : \tau$. If S is a statement in S', and $\Gamma ; \mathcal{R} ; \textit{pc} \vdash S : \tau$, and $[\![S]\!]\Psi = \langle P, \Psi'\rangle$ and $P' \vDash \Psi'$, then $\Gamma' ; P' \vdash P$.

Proof. By induction on the derivation of $\Gamma ; \mathcal{R} ; \textit{pc} \vdash e : \tau$ or $\Gamma ; \mathcal{R} ; \textit{pc} \vdash s : \tau$.

- **Cases (INT) and (REF)**. Obvious.
- **Case (DEREF)**. If $\Gamma'(m) = \sigma$, then e' is $!m$, and P is $\emptyset$ by rule (TE3). We have $\Gamma' ; \mathcal{Q} \vdash !m : \tau$, since $\tau = \sigma$, and $\mathcal{Q}$ contains only one host. If $\Gamma'(m) = \texttt{int}_{\ell_1}@\mathcal{Q}_m$, by rule (TE4), $P = \{r\}$ where

  $$r = c\{\ell,\ \mathcal{Q},\ \lambda.\texttt{exec}(\texttt{read}[\ell, \ell_1, m, c', \texttt{cid}, \langle c_u,\ \texttt{cid}\rangle.z],\ \texttt{nid},\ \ell,\ \mathcal{Q},\ \epsilon)\}.$$

  By rules (EXEC) and (RD), we have:

  $$\dfrac{\dfrac{\Gamma' \vdash \texttt{read}[\ell, \ell_1, m, c', \texttt{cid}, \langle c_u,\ \texttt{cid}\rangle.z] : \texttt{reactor}\{\ell,\ \mathcal{Q}_m\} \quad \ell \sqsubseteq \ell}{\Gamma'' ; P' ; \mathcal{Q} ; \ell \vdash \texttt{exec}(\texttt{read}[\ell, \ell_1, m, c', \texttt{cid}, \langle c_u,\ \texttt{cid}\rangle.z],\ \texttt{nid},\ \ell,\ \mathcal{Q},\ \epsilon) : \texttt{stmt}_{\ell}}}{\Gamma' ; P' \vdash r}$$

  where $\Gamma'' = \Gamma', \texttt{cid} : \texttt{int}_{\ell}, \texttt{nid} : \texttt{int}_{\ell}$.
- **Case (ADD)**. By induction, $\Gamma' ; P' \vdash P_1$ and $\Gamma' ; P' \vdash P_2$. Thus, $\Gamma' ; P' \vdash P_1 \cup P_2$. By induction, $\Gamma', \overline{z_i : \tau_i} ; \mathcal{Q} \vdash e'_i : \tau$ for $i \in \{1, 2\}$. Thus, $\Gamma', \overline{z_1 : \tau_1}, \overline{z_2 : \tau_2} ; \mathcal{Q} \vdash e'_1 + e'_2 : \tau$.

- **Case (SKIP)**. By (TS3), $P = \{r\}$ and $r = c\{\ell, \mathcal{Q}, \lambda.s\}$, where s is obtained from $[\![c]\!]\Psi = \langle s, \Psi'\rangle$. By Lemma 6.4.1, $\Gamma', \texttt{cid} : \texttt{int}_\ell, \texttt{nid} : \texttt{int}_\ell \,;P' \,; \mathcal{Q} \,; \ell \vdash s : \tau$. Therefore, $\Gamma' \,; P' \vdash P$.

- **Case (SEQ)**. S is $S_1; S_2$, and we have $\Gamma \,; \mathcal{R} \,; pc \vdash S_1 : \texttt{stmt}_{\mathcal{R}_1}$ and $\Gamma \,; \mathcal{R}_1 \,; pc \vdash S_2 : \tau$. By rule (TS4), $[\![S_2]\!]\Psi = \langle P_2, \Psi_1\rangle$ and $[\![S_1]\!]\Psi_1 = \langle P_1, \Psi'\rangle$. By induction, $\Gamma' \,; P' \vdash P_2$ and $\Gamma' \,; P' \vdash P_1$. Therefore $\Gamma' \,; P' \vdash P_1 \cup P_2$.

- **Case (ASSIGN)**. S is $m := e$, and $\Gamma \,; \mathcal{R} \vdash e : \texttt{int}_{\ell'}$. By rules (TS1) and (TS2), $[\![e]\!]\langle c, c_1, \ell, \mathcal{Q}\rangle = \langle P_e, \lambda\overline{\pi \triangleright z\!:\!\tau}.e'\rangle$. By induction, $\Gamma' \,; P' \vdash P_e$ and $\Gamma', \overline{z\!:\!\tau} \,; \mathcal{Q} \vdash e' : \texttt{int}_{\ell'}$. If $\Gamma(m) = \sigma@\mathcal{Q}_m$, then (TS1) is used. By Lemma 6.4.1, $\Gamma' \,; P' \vdash r_2$. Let $\Gamma'' = \Gamma', \overline{z\!:\!\tau}, \texttt{cid}\!:\!\texttt{int}_\ell, \texttt{nid}\!:\!\texttt{int}_\ell$. Then the following derivation shows that r_1 is also well-typed:

$$\cfrac{\cfrac{\begin{array}{c}\Gamma' \,; P' \vdash \texttt{write}[\ell, m, c_2, \texttt{cid}] : \texttt{reactor}\{\ell,\ \mathcal{Q}_m,\ \texttt{int}_\ell\}\\ \Gamma'' \vdash \texttt{nid} : \texttt{int}_\ell \quad \Gamma'' \vdash \ell : \texttt{label}_\bot \quad \ell \sqsubseteq \ell \quad \Gamma, \overline{z\!:\!\tau} \,; \mathcal{Q} \vdash e' : \texttt{int}_\ell\end{array}}{\Gamma'' \,; P' \,; \mathcal{Q} \,; \ell \vdash \texttt{exec}(\texttt{write}[\ell, m, c_2, \texttt{cid}],\ \texttt{nid},\ \ell,\ \mathcal{Q}_m,\ e') : \texttt{stmt}_\ell}}{\Gamma' \,; P' \vdash r_1}$$

If $\Gamma(m) = \sigma$, then (TS2) is used. By Lemma 6.4.1, $\Gamma'' \,; P' \,; \mathcal{Q} \,; \ell \vdash s' : \tau$. Therefore, we have the following derivation:

$$\cfrac{\cfrac{\Gamma' \vdash m : (\texttt{int}_\ell\, \texttt{ref})_\ell \quad \Gamma', \overline{z\!:\!\tau} \vdash e' : \texttt{int}_\ell \quad \ell \sqsubseteq \texttt{int}_\ell}{\Gamma', \overline{z\!:\!\tau} \,; P' \,; \mathcal{Q} \,; \ell \vdash m := e' : \texttt{stmt}_\bot} \qquad \Gamma'' \,; P' \,; \mathcal{Q} \,; \ell \vdash s' : \tau}{\Gamma' \,; P' \vdash r_1}$$

- **Case (IF)**. S is $\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2$. By induction, P_e, P_1 and P_2 are well-typed, and e' is well-typed with respect to $\Gamma', \overline{z\!:\!\tau}$ and $\mathcal{Q}$. By Lemma 6.4.1, s_1' and s_2' are well-typed. Therefore, the statement $\texttt{if}\ e'\ \texttt{then}\ s_1'\ \texttt{else}\ s_2'$ is well-typed, and so is r_1.

- **Case (WHILE)**. S is $\texttt{while}\ e\ \texttt{do}\ S'$. By induction, P_e and P are well-typed. The following derivation shows that r_1 is well-typed:

$$\dfrac{\dfrac{\Gamma', \overline{z\!:\!\tau}, z'\!:\!\texttt{int}_\ell \vdash z' : \texttt{int}_\ell \qquad \vdash \langle c_1, \texttt{nid}\rangle.z' : (\texttt{int}_\ell\ \texttt{var})_\ell}{\Gamma', \overline{z\!:\!\tau}, z'\!:\!\texttt{int}_\ell\,;\mathcal{Q}\,;\ell \vdash \texttt{setvar}(\langle c_1, \texttt{nid}\rangle.z',\ z') : \texttt{stmt}_\bot} \quad \Gamma', \overline{z\!:\!\tau}, z'\!:\!\texttt{int}_\ell\,;\mathcal{Q}\,;\ell \vdash s_1' : \tau}{\Gamma', \overline{z\!:\!\tau}, z'\!:\!\texttt{int}_\ell\,;P'\,;\mathcal{Q}\,;\ell \vdash \texttt{setvar}(\langle c_1, \texttt{nid}\rangle.z',\ z'); s_1' : \tau}$$

$$\dfrac{\dfrac{\begin{array}{c}\Gamma', \overline{z\!:\!\tau}, z'\!:\!\texttt{int}_\ell\,;\mathcal{Q} \vdash e' : \texttt{int}_{\ell'}\\ \Gamma', \overline{z\!:\!\tau}, z'\!:\!\texttt{int}_\ell\,;P'\,;\mathcal{Q}\,;\ell \vdash \texttt{setvar}(\langle c_1, \texttt{nid}\rangle.z',\ z'); s_1' : \tau\\ \Gamma', \overline{z\!:\!\tau}, z'\!:\!\texttt{int}_\ell\,;P'\,;\mathcal{Q}\,;\ell \vdash s_2' : \tau\end{array}}{\Gamma', \overline{z\!:\!\tau}, z'\!:\!\texttt{int}_\ell\,;P'\,;\mathcal{Q}\,;\ell \vdash \texttt{if}\ e'\ \texttt{then}\ \texttt{setvar}(\langle c_1, \texttt{nid}\rangle.z',\ z'); s_1'\ \texttt{else}\ s_2' : \tau}}{\Gamma'\,;P' \vdash r_1}$$

- **Case (SUB)**. By induction.

□

Theorem 6.4.1 (Typing preservation). Suppose $\Gamma\,;\mathcal{R}\,;pc \vdash S : \tau$, and $[\![S]\!]\emptyset = \langle P, c\rangle$ with respect to a distribution scheme D, and $S = \{c\}\, S_1; S_2$. Then $\Gamma' \Vdash P$, where $\Gamma' = [\![\Gamma]\!]D$.

Proof. By Lemma 6.4.2, $\Gamma' \vdash P$. By examining the translation rules, P satisfies (RV1)–(RV3). □

6.5 Semantics preservation

In general, an adequate translation needs to preserve semantics of the source program. In a distributed setting, attackers may launch active attacks from bad hosts, making the low-integrity part of the target execution deviate from the source execution. However, the trustworthiness of the target code does not depend on the low-integrity program state. Therefore, we consider a translation adequate if it preserves high-integrity semantics.

This notion of semantics preservation is formalized as two theorems. First, the translation soundness theorem says that there exists a benchmark execution of the target program generating the same outputs as the source program execution. Based on Theorem 5.4.2, any execution of the target program would result in equivalent high-integrity outputs as the benchmark execution and the source program. Therefore, we only need another theorem stating that any target execution achieves the same availability as the source.

To prove the translation soundness theorem, we construct an equivalence relation between an Aimp configuration and a DSR configuration, and show that there exists a DSR evaluation to preserve the equivalence relation. Informally, a target configuration $\langle \Theta,\, \mathcal{M},\, \mathcal{E} \rangle$ and a source configuration $\langle S,\, M \rangle$ are equivalent, if $\mathcal{M}$ and M are equivalent, and Θ and $\mathcal{E}$ indicate that the code to be executed by $\langle \Theta,\, \mathcal{M},\, \mathcal{E} \rangle$ is exactly the target code of S. Suppose D is the distribution scheme used in the translation. The equivalence between M and $\mathcal{M}$ is defined as follows:

Definition 6.5.1 ($\Gamma\,; D \vdash \mathcal{M} \approx M$). For any m in *dom*(Γ), then $\mathcal{M}(h,m) = M(m)$ for any $h \in D(m)$.

The configuration $\langle \Theta,\, \mathcal{M},\, \mathcal{E} \rangle$ must be able to execute the target code of S. As a result, the entries of the target code of S must be *activated* in $\langle \Theta,\, \mathcal{M},\, \mathcal{E} \rangle$ with respect to the current context identifier, as defined below:

Definition 6.5.2 ($\mathcal{E}\,; \eta \models \Psi$). That Ψ is activated with context identifier η in the environment $\mathcal{E}$, written $\mathcal{E}\,; \eta \models \Psi$, if it can be inferred using the following rules, where auxiliary function $\mathcal{E}(w,\eta)$ returns η if w is `cid`, and the value of $\langle c,\, \eta \rangle.z$ in $\mathcal{E}$ if w is $c.z$.

$$\frac{\mathcal{E}\,; \eta \models (c,w) \quad \mathcal{E}\,; \mathcal{E}(w,\eta)\,; \mathit{label}(c) \models \Psi}{\mathcal{E}\,; \eta \models (c,w), \Psi} \qquad \frac{\mathcal{E}\,; \eta\,; \ell \models (c,w) \quad \mathcal{E}\,; \mathcal{E}(w,\eta)\,; \mathit{label}(c) \models \Psi}{\mathcal{E}\,; \eta\,; \ell \models (c,w), \Psi}$$

$$\frac{\forall h \in \mathit{hosts}(c).\ \langle c, \eta, \ell, \mathcal{A}, t, \texttt{off} \rangle \in \mathcal{E}(h)}{\mathcal{E}\,; \eta \models (c,w)} \qquad \frac{\forall h \in \mathit{hosts}(c).\ \langle c, \eta, \ell', \mathcal{A}, t, * \rangle \in \mathcal{E}(h) \quad \ell \sqsubseteq \ell'}{\mathcal{E}\,; \eta\,; \ell \models (c,w)}$$

To track the activated entries during program execution, we introduce the notation $P\,; \Psi \vdash S : \Psi'$, which intuitively means that executing the target code of S with the list of activated entries Ψ would result in the list of activated entries Ψ'. Formally, it is defined using the following inference rules:

(EL1) $P\,;\Psi \vdash \texttt{skip} : \Psi$ (EL2) $\dfrac{[\![S]\!]\Psi' = \langle P', \Psi\rangle \quad P' \subseteq P}{P\,;\Psi \vdash S : \Psi'}$

(EL3) $\dfrac{P\,;\Psi \vdash S : \Psi' \quad \Psi_1 = \langle c,\, c_1.z\rangle, \Psi_2}{P\,;\Psi, \Psi_1 \vdash S : \Psi' \otimes \Psi_1}$ (EL4) $\dfrac{P\,;\Psi \vdash S_1 : \Psi_1 \quad P\,;\Psi_1 \vdash S_2 : \Psi_2}{P\,;\Psi \vdash S_1; S_2 : \Psi_2}$

The unnamed statement `skip` has no effects or target code. Thus, rule (EL1) says that executing the target code of `skip` does not activate any new entry. Rule (EL2) is straightforward based on the meaning of $[\![S]\!]\Psi' = \langle P', \Psi\rangle$. Rule (EL3) is applied to the case that S belongs to the body of a `while` statement, and Ψ_1 is the entry list for the computation following S. Based on the translation rule (TS6), $\Psi_1 = (c, c_1.z), \Psi_2$, where $(c, c_1.z)$ is the entry for the next iteration of the `while` statement. Suppose $P\,;\Psi \vdash S : \Psi'$. If $\Psi' = c$, then after S terminates, the next iteration of the loop would start, and the activated entry list would be $(,1)$. Otherwise, the entry list at the point that S terminates is Ψ', Ψ_1. Suppose $\Psi_1 = \langle c,\, c_1.z\rangle, \Psi_2$. Then the notation $\Psi' \otimes \Psi_1$ denotes Ψ_1 if $\Psi' = c$, and Ψ', Ψ_1 if otherwise. Rule (EL4) is standard for composing $P\,;\Psi \vdash S_2 : \Psi_1$ and $P\,;\Psi_1 \vdash S_2 : \Psi_2$, as the termination point of S_1 is the starting point of S_2.

To construct the benchmark execution, it is convenient to assume that all the reactor replicas are running synchronously, and to formalize the program point that a target configuration corresponds to. A program point is represented by $\langle s; \Psi; \Pi\rangle$, where s is the code of the current running threads, Ψ is the entry list for the program P following the current thread, and Π is a set of *communication ports* used by P. A communication port is either a reactor name c or a remote variable name $c.z$. Intuitively, at the program point represented by $\langle s; \Psi; \Pi\rangle$, the entry list Ψ are activated, and there are no messages for the communication ports in Π yet. Formally, we have the following definition:

Definition 6.5.3 ($\Theta\,;\mathcal{E}\,;\eta \models \langle s;\ \Psi;\ \Pi\rangle$). A configuration $\langle \Theta,\ \mathcal{M},\ \mathcal{E}\rangle$ corresponds to the program point $\langle s;\ \Psi;\ \Pi\rangle$ with respect to the context identifier η, written $\Theta\,;\mathcal{E}\,;\eta \models \langle s;\ \Psi;\ \Pi\rangle$, if the following conditions hold with $\Psi = c\,;\Psi'$. First, any unfinished thread in Θ has the form $\langle s, t, h, c, \eta\rangle$, and the timestamp of any thread in Θ is less than or equal to t. Second, $\mathcal{E}\,;\eta \models \Psi$. Third, for any π in Π, if $\pi = c'$ and $c' \neq c$, then $\mathcal{E}$ contains no `exec` messages for $\langle \pi, \eta\rangle$; if $\pi = c.z$ does not appear in Ψ, then $\mathcal{E}$ contains no `setvar` messages for $\langle \pi, \eta\rangle$. If s is the code of c, then $\langle \Psi; \Pi\rangle$ is an abbreviation of $\langle s;\ \Psi;\ \Pi\rangle$.

Now we define the DSR-Aimp configuration equivalence and prove the translation soundness theorem after proving two lemmas.

Definition 6.5.4 (D-A configuration equivalence). A DSR configuration $\langle\Theta,\ \mathcal{M},\ \mathcal{E}\rangle$ and an Aimp configuration $\langle S,\ M\rangle$ are equivalent with respect to Γ, P, η and Ψ', written as $\Gamma\,; P\,;\eta \vdash \langle\Theta,\ \mathcal{M},\ \mathcal{E}\rangle \approx \langle S,\ M,\ \Psi'\rangle$, if the following conditions hold. First, $P\,;\Psi \vdash S : \Psi'$. Second, $\Theta\,;\mathcal{E}\,;\eta \models \langle\Psi;\ \Pi_S\rangle$, where Π_S are the set of communication ports of the target code of S. Third, $\Gamma \vdash M \approx \mathcal{M}$.

Lemma 6.5.1 (Expression translation soundness). Suppose e is an Aimp expression, and $[\![e]\!]\langle c, c', c_u, \ell, H\rangle = \langle P, \lambda\overline{\pi \triangleright \tau\ z}.e'\rangle$, and $\langle e,\ M\rangle \Downarrow v$, and $\Gamma \vdash M \approx \mathcal{M}$, and $\Theta\,;\mathcal{E}\,;\eta \models \langle c, \Psi;\ \Pi_P \cup \{c',\ c_u.\overline{z}\} \cup \Pi\rangle$. Then there exists a run $\langle\Theta,\ \mathcal{M},\ \mathcal{E}\rangle \longmapsto^* \langle\Theta',\ \mathcal{M},\ \mathcal{E}'\rangle$ such that $\Theta'\,;\mathcal{E}'\,;\eta \models \langle c', \Psi;\ \Pi\rangle$, and $\langle e'[A],\ \mathcal{M}[h,t]\rangle \Downarrow v$, where A is the variable record in the closure $\langle c_u,\ \eta\rangle$ on host h.

Proof. By induction on the structure of e.

- e is n. Trivial.
- e is $!m$ and $\Gamma(m) = \sigma$. Then P is empty, and e' is $!m$. Since $\Gamma \vdash M \approx \mathcal{M}$, we have that $\langle !m,\ \mathcal{M}(h,t)\rangle \Downarrow M(m)$.
- e is $!m$ and $\Gamma(m) = \mathtt{int}_{\ell_1}@\mathcal{Q}$. By (TE4), P is $\{r\}$, and

  $$r = c\{\ell,\ \mathcal{Q}',\ \lambda.\mathtt{exec}(\mathtt{read}[\ell, \ell_1, m, c', \mathtt{cid}, \langle c_u,\ \mathtt{cid}\rangle.z],\ \mathtt{nid},\ \ell,\ \mathcal{Q}',\ \epsilon).$$

  Then by running the `exec` statement, we have $\langle\Theta,\ \mathcal{M},\ \mathcal{E}\rangle \longmapsto^* \langle\Theta_1,\ \mathcal{M},\ \mathcal{E}_1\rangle$, and

  $$\Theta_1\,;\mathcal{E}_1\,;\eta' \models \langle s';\ \mathtt{read}[\ell, \ell_1, m, c', \mathtt{cid}, \langle c_u,\ \mathtt{cid}\rangle.z], \Psi;\ \{c', c_u.z\} \cup \Pi\rangle,$$

where s' is $\texttt{setvar}(\langle c_u,\, \eta\rangle.z,\, !m);\ \texttt{exec}(c',\, \eta,\, \ell,\, \oslash,\, \epsilon)$. In other words, the execution reaches the point that all the replicas of the `read` reactor are invoked with the newly-created context identifier η'. Further, by executing s' on all the hosts of m and processing all the messages sent by s', the execution produces $\langle \Theta_1,\, \mathcal{M},\, \mathcal{E}_1\rangle \longmapsto^* \langle \Theta',\, \mathcal{M},\, \mathcal{E}'\rangle$ such that $\Theta'\,;\mathcal{E}'\,;\eta \vDash \langle c';\ \Psi;\ \Pi\rangle$. By $\Gamma \vdash M \approx \mathcal{M}$, the synthesizer $\texttt{QR}[\mathcal{Q}, I]$ associated with $c_u.z$ receives the `setvar` messages containing the same versioned value $v@t'$ where $v = M(m)$. Therefore, z is mapped to v in the closure $\langle c_u,\, \eta\rangle$ by the evaluation rule (M3). Thus, we have $\langle z[\mathcal{A}],\, \mathcal{M}(h,t)\rangle \Downarrow v$.

For simplicity, we write such an execution run in the form of the following table, where each line denotes that the execution produces a system configuration (the first column), which corresponds to a program point (the second column) and satisfies certain constraints (the third column), based on some reasoning (the fourth column).

$\langle \Theta,\, \mathcal{M},\, \mathcal{E}\rangle$			
$\longmapsto^* \langle \Theta_1,\, \mathcal{M},\, \mathcal{E}_1\rangle$	$\langle s';\ \Psi';\ \{c', c_u.z\} \cup \Pi\rangle$		
$\longmapsto^* \langle \Theta',\, \mathcal{M},\, \mathcal{E}'\rangle$	$\langle c', \Psi;\ \Pi\rangle$	$\langle z[\mathcal{A}],\, \mathcal{M}(h,t)\rangle \Downarrow M(m)$	By $\Gamma \vdash M \approx \mathcal{M}$

- e is $e_1 + e_2$. By rule (TE5), we have $\llbracket e_1\rrbracket\langle c, c_1, c_u, \ell, \mathcal{Q}\rangle = \langle P_1, \lambda\overline{\pi_1 \triangleright \tau_1\ z_1}.\ e_1'\rangle$ and $\llbracket e_2\rrbracket\langle c_1, c', c_u, \ell, \mathcal{Q}\rangle = \langle P_2, \lambda\overline{\pi_2 \triangleright \tau_2\ z_2}.\ e_2'\rangle$. Then we have the following execution:

$\langle \Theta,\, \mathcal{M},\, \mathcal{E}\rangle$			
$\longmapsto^* \langle \Theta_1,\, \mathcal{M},\, \mathcal{E}_1\rangle$	$\langle c_1, \Psi;\ \Pi_{P_2} \cup \{c', c_u.\overline{z_2}\} \cup \Pi\rangle$	$\langle e_1'[\mathcal{A}],\, \mathcal{M}(h,t)\rangle \Downarrow v_1$	By induction
$\longmapsto^* \langle \Theta',\, \mathcal{M},\, \mathcal{E}'\rangle$	$\langle c', \Psi;\ \Pi\rangle$	$\langle e_2'[\mathcal{A}],\, \mathcal{M}(h,t)\rangle \Downarrow v_2$	By induction

Therefore, $\langle e_1' + e_2'[\mathcal{A}],\, \mathcal{M}(h,t)\rangle \Downarrow v$, where $v = v_1 + v_2$ and $\mathcal{A}$ is the variable record of the closure $\langle c_u,\, \eta\rangle$ on h.

$\square$

Lemma 6.5.2 (Control transfer soundness). Suppose $[\![(c,w)]\!]\Psi' = \langle s, \Psi\rangle$, and $\Psi = (c,w), \Psi''$, and $\Theta\,;\mathcal{E}\,;\eta \vDash \langle s;\ c_1, \Psi'';\ \Pi\rangle$. Then $\langle\Theta,\ \mathcal{M},\ \mathcal{E}\rangle \longmapsto^* \langle\Theta',\ \mathcal{M},\ \mathcal{E}'\rangle$ such that $\Theta'\,;\mathcal{E}'\,;\eta' \vDash \langle\Psi';\ \Pi\rangle$, where $\eta' = \mathcal{E}(w,\eta)$.

Proof. By (TC1), s is $s_j; \ldots; s_0$, and $\Psi' = \psi_1, \ldots, \psi_n$, and $\Psi = (c,w), \psi_{j+1}, \ldots, \psi_n$. By (TC2), each s_i activates ψ_i, and s_0 invokes c_1. Let $\langle\Theta,\ \mathcal{M},\ \mathcal{E}\rangle \longmapsto^* \langle\Theta',\ \mathcal{M},\ \mathcal{E}'\rangle$ be the run that finishes executing s on the quorum system of c and processing the messages sent by s. Then $\Theta'\,;\mathcal{E}'\,;\eta' \vDash \langle\Psi';\ \Pi\rangle$. □

Theorem 6.5.1 (Translation soundness). Suppose $\Gamma\,;\mathcal{R}\,;pc \vdash S : \tau$, and $\langle S,\ M\rangle \longmapsto \langle S',\ M'\rangle$, and $\Gamma\,;P\,;\eta \vdash \langle\Theta,\ \mathcal{M},\ \mathcal{E}\rangle \approx \langle S,\ M,\ \Psi'\rangle$. Then there exists a run $E = \langle\Theta,\ \mathcal{M},\ \mathcal{E}\rangle \longmapsto^* \langle\Theta',\ \mathcal{M}',\ \mathcal{E}'\rangle$ such that $\Gamma\,;P\,;\eta' \vdash \langle\Theta',\ \mathcal{M}',\ \mathcal{E}'\rangle \approx \langle S',\ M',\ \Psi'\rangle$. In addition, for any message μ sent in E, the port of μ is in either Ψ or Π_S.

Proof. By induction on the evaluation step $\langle S,\ M\rangle \longmapsto \langle S',\ M'\rangle$. Because $\Gamma\,;P\,;\eta \vdash \langle\Theta,\ \mathcal{M},\ \mathcal{E}\rangle \approx \langle S,\ M,\ \Psi'\rangle$, we have $P\,;\Psi \vdash S : \Psi'$, and $\Theta\,;\mathcal{E}\,;\eta \vDash \langle\Psi;\ \Pi_S\rangle$, and $\Gamma \vdash \mathcal{M} \approx M$.

- Case (S1). In this case, S is $\{c\}\,m := e$, and $M' = M[m \mapsto v]$, and $\langle e,\ M\rangle \Downarrow v$. Suppose $\Psi = c, \Psi_1$. Then we have

  $$\begin{array}{llll} \langle\Theta,\ \mathcal{M},\ \mathcal{E}\rangle & & & \\ \longmapsto^* \langle\Theta_1,\ \mathcal{M},\ \mathcal{E}_1\rangle & \langle c_1, \Psi_1;\ \Pi_S - \Pi_{P_e}\rangle & \langle e'[\mathcal{A}],\ \mathcal{M}(h,t)\rangle \Downarrow v & \text{By Lemma 6.5.1} \end{array}$$

  If $\Gamma(m) = \sigma@\mathcal{Q}$, then rule (TS1) is used, and the code of c_1 is

  $$\texttt{exec}(\texttt{write}[\ell, m, c_2, \texttt{cid}],\ \texttt{nid},\ \ell,\ \oslash,\ e').$$

  Thus, we have

  $$\begin{array}{llll} \langle\Theta_1,\ \mathcal{M},\ \mathcal{E}_1\rangle & & & \\ \longmapsto^* \langle\Theta_2,\ \mathcal{M},\ \mathcal{E}_2\rangle & \langle m := v; \texttt{exec}(c_2,\ \eta,\ \ell,\ \&m,\ \epsilon);\ \texttt{write}[\ell, m, c_2, \eta], \Psi_1;\ \{c_2\}\rangle & & \\ \longmapsto^* \langle\Theta_3,\ \mathcal{M}',\ \mathcal{E}_3\rangle & \langle c_2, \Psi_1;\ \emptyset\rangle & \mathcal{M}' = \mathcal{M}[m \mapsto v] & \\ \longmapsto^* \langle\Theta_4,\ \mathcal{M}',\ \mathcal{E}_4\rangle & \langle\Psi';\ \emptyset\rangle & \Psi' \vdash \texttt{skip} : \Psi' & \text{By Lemma 6.5.2} \end{array}$$

If $\Gamma(m) = \sigma$, rule (TS2) is used, and the code of c_1 is $m := e'; s'$, where s' comes from $[\![c]\!]\Psi' = \langle s', \Psi\rangle$. Thus, we have

$$\begin{array}{llll} \langle \Theta_1, \mathcal{M}, \mathcal{E}_1\rangle & & & \\ \longmapsto^* \langle \Theta_2, \mathcal{M}', \mathcal{E}_2\rangle & \langle s'; c_1, \Psi_1; \emptyset\rangle & \mathcal{M}'(h,m) = v & \\ \longmapsto^* \langle \Theta_3, \mathcal{M}', \mathcal{E}_3\rangle & \langle \Psi'; \emptyset\rangle & & \text{By Lemma 6.5.2} \end{array}$$

- Case (S2). S is $S_1; S_2$, and $P\,; \Psi \vdash S_1; S_2 : \Psi'$, which implies that $P\,; \Psi \vdash S_1 : \Psi_1$ and $P\,; \Psi_1 \vdash S_2 : \Psi'$. By induction, there exists a run $E = \langle \Theta, \mathcal{M}, \mathcal{E}\rangle \longmapsto^* \langle \Theta', \mathcal{M}', \mathcal{E}'\rangle$ such that $\Gamma\,; P\,; \eta \vdash \langle \Theta', \mathcal{M}', \mathcal{E}'\rangle \approx \langle S_1', M', \Psi_1\rangle$. Therefore, $\Theta'\,; \mathcal{E}'\,; \eta \models \langle \Psi_1'; \Pi_{S_1'}\rangle$, and for any π that receives a message in E, if $\pi \notin \Pi_{S_1}$, then $\pi \in \Psi_1'$. Thus, we have $\Theta'\,; \mathcal{E}'\,; \eta \models \langle \Psi_1'; \Pi_{S_1';S_2}\rangle$. In addition, $\Psi_1' \vdash S_1' : \Psi_1$ holds. So $P\,; \Psi_1' \vdash S_1'; S_2 : \Psi'$. Thus, we have $\Gamma\,; P\,; \eta \vdash \langle \Theta', \mathcal{M}', \mathcal{E}'\rangle \approx \langle S_1'; S_2, M', \Psi'\rangle$.

- Case (S3). S is $\{c\}\,\texttt{skip}; S'$. By $\Psi \vdash \{c\}\,\texttt{skip}; S' : \Psi'$, we have $\Psi \vdash \{c\}\,\texttt{skip} : \Psi_1'$ and $\Psi_1' \vdash S' : \Psi'$. Then we have

$$\begin{array}{lll} \langle \Theta, \mathcal{M}, \mathcal{E}\rangle & \langle c, \Psi_1; \Pi_S\rangle & \\ \longmapsto^* \langle \Theta', \mathcal{M}, \mathcal{E}'\rangle & \langle \Psi_1'; \Pi_{S'}\rangle & \text{By rule (TS3) and Lemma 6.5.2} \end{array}$$

- Case (S4). Since $P\,; \Psi \vdash S : \Psi'$, we have that $[\![S]\!]\Psi_1' = \langle P', \Psi_1\rangle$, and $\Psi = \Psi_1, \Psi_2$ and $\Psi' = \Psi_1' \otimes \Psi_2$. By rule (TS5), $\Psi_1 = c, \Psi''$. Then we have

$$\begin{array}{llll} \langle \Theta, \mathcal{M}, \mathcal{E}\rangle & & & \\ \longmapsto^* \langle \Theta_1, \mathcal{M}, \mathcal{E}_1\rangle & \langle c_1, \Psi''; \Pi_S\rangle & \langle e'[\mathcal{A}_{c_1,\eta}], \mathcal{M}(h,t)\rangle \Downarrow n & \text{By Lemma 6.5.1} \\ \longmapsto^* \langle \Theta_2, \mathcal{M}, \mathcal{E}_2\rangle & \langle s_1; c_1, \Psi''; \Pi_{S_1}\rangle & & \text{By (S5)} \\ \longmapsto^* \langle \Theta_3, \mathcal{M}, \mathcal{E}_3\rangle & \langle \Psi_1''; \Pi_{S_1}\rangle & \Psi_1'' \vdash S_1 : \Psi_1' & \text{By Lemma 6.5.2} \end{array}$$

  Also the above run is limited to the code of S and does not affect Ψ_2. Therefore, $\Theta_3\,; \mathcal{E}_3\,; \eta \models \langle \Psi_1'', \Psi_2; \Pi_{S_1}\rangle$, and $P\,; \Psi_1'', \Psi_2 \vdash S_1 : \Psi'$. Thus, $\langle \Theta_3, \mathcal{M}, \mathcal{E}_3\rangle \approx \langle S_1, M, \Psi_1'\rangle$.

- Case (S5). By the same argument as in case (S4).

- Case (S6). S is $\texttt{while}\,e\,\texttt{do}\,S_1$, and S' is $S_1; \texttt{while}\,e\,\texttt{do}\,S_1$, and $\langle e,\ M\rangle \Downarrow n$ ($n > 0$). Then we have:

$$\begin{array}{llll}
\langle \Theta,\ \mathcal{M},\ \mathcal{E}\rangle & & & \\
\longmapsto^* \langle \Theta_1,\ \mathcal{M},\ \mathcal{E}_1\rangle & \langle c_1, \Psi'';\ \Pi_S\rangle & \langle e'[\mathcal{A}_{c_1,\eta}],\ \mathcal{M}\rangle \Downarrow n & \text{By Lemma 6.5.1} \\
\longmapsto^* \langle \Theta_2,\ \mathcal{M},\ \mathcal{E}_2\rangle & \langle \texttt{setvar}(\langle c_1,\ \texttt{nid}\rangle.z', z'); s_1;\ c_1, \Psi'';\ \Pi_S\rangle & & \text{By (S5)} \\
\longmapsto^* \langle \Theta_3,\ \mathcal{M},\ \mathcal{E}_3\rangle & \langle s_1;\ c_1;\ \Pi_{S_1}\rangle & \mathcal{E}_3\,; \mathcal{A}_{\Theta_3}(\texttt{nid})\,; \ell_c \vDash \langle c,\ c_1.z'\rangle, \Psi'' & \\
\longmapsto^* \langle \Theta',\ \mathcal{M},\ \mathcal{E}'\rangle & \langle \Psi_1;\ \Pi_{S_1}\rangle & \mathcal{A}_{\Theta'}(\texttt{cid}) = \mathcal{A}_{\Theta_3}(\texttt{nid}) & \text{By Lemma 6.5.2}
\end{array}$$

Therefore, $\langle \Theta',\ \mathcal{E}'\rangle \approx \langle \Psi_1, \langle c,\ c_1.z'\rangle, \Psi'';\ \Pi_{S_1;S}\rangle$. In addition, $\Psi_1, \langle c, c_1.z'\rangle, \Psi'' \vdash S_1; S : \Psi'$. Thus, we have $\langle \Theta',\ \mathcal{M},\ \mathcal{E}'\rangle \approx \langle S_1; \texttt{while}\ e\ \texttt{do}\ S_1,\ M,\ \Psi'\rangle$.

- Case (S7). S is $\texttt{while}\ e\ \texttt{do}\ S_1$, and $\langle e,\ M\rangle \Downarrow n$, and $n \le 0$. Then we have:

$$\begin{array}{llll}
\langle \Theta,\ \mathcal{M},\ \mathcal{E}\rangle & & & \\
\longmapsto^* \langle \Theta_1,\ \mathcal{M},\ \mathcal{E}_1\rangle & \langle c_1, \Psi'';\ \Pi_S\rangle & c_1 \vdash \langle e',\ \mathcal{M}\rangle \Downarrow n & \text{By Lemma 6.5.1} \\
\longmapsto^* \langle \Theta_2,\ \mathcal{M},\ \mathcal{E}_2\rangle & \langle s_2;\ c_1, \Psi'';\ \emptyset\rangle & & \\
\longmapsto^* \langle \Theta_3,\ \mathcal{M},\ \mathcal{E}_3\rangle & \langle \Psi'';\ \emptyset\rangle & \mathcal{E}_3\,; \texttt{nid}\,; \ell_c \vDash \langle c,\ c_1, w\rangle, \Psi'' & \text{By Lemma 6.5.2}
\end{array}$$

□

Now we show that a target program achieves the same availability as the source program. First, we formally define the notion that a target memory M has the same availability as a source memory $\mathcal{M}$:

Definition 6.5.5 ($\Gamma \vdash \mathcal{M} \approx_{A \not\le l_\mathtt{A}} M$). For any m such that $A(\Gamma(m)) \not\le l_\mathtt{A}$, if $M(m) \neq \texttt{none}$, then for any h in $\mathcal{Q}_m$, $A(h) \not\le l_\mathtt{A}$ implies $\mathcal{M}(h, m) \neq \texttt{none}$.

Again, we prove the availability preservation result by induction. First, we prove two lemmas that are concerned with the availability of expression target code and control transfer code, respectively. The availability results need to be applicable to all executions. Accordingly, we say "$\langle \Theta,\ \mathcal{M},\ \mathcal{E}\rangle \leadsto^* \langle \Theta',\ \mathcal{M}',\ \mathcal{E}'\rangle$ such that a condition holds" if for any run $\langle \Theta,\ \mathcal{M},\ \mathcal{E}\rangle \longmapsto^* \langle \Theta_1,\ \mathcal{M}_1,\ \mathcal{E}_1\rangle$, there exists $\langle \Theta',\ \mathcal{M}',\ \mathcal{E}'\rangle$ satisfying the condition and $\langle \Theta_1,\ \mathcal{M}_1,\ \mathcal{E}_1\rangle \longmapsto^* \langle \Theta',\ \mathcal{M}',\ \mathcal{E}'\rangle$. Let $\mathcal{E} \vDash \langle c,\ \eta\rangle.z$ denote that variable

$\langle c,\, \eta\rangle.z$ is already initialized in $\mathcal{E}$. More concretely, For any host h of c, the variable record of the closure $\langle c,\, \eta\rangle$ on host h maps z to a value that is not none. In addition, let $\mathcal{E} \models \langle c,\, \eta\rangle$ denote that the closure $\langle c,\, \eta\rangle$ has been invoked on all the hosts of c in $\mathcal{E}$. Then the expression availability lemma is formalized as follows:

Lemma 6.5.3 (Expression availability). Suppose $\Gamma\,;\mathcal{R}\,;pc \vdash e : \mathtt{int}_\ell$, and $\langle e,\, M\rangle \Downarrow n$, and $A(\mathcal{R}) \not\leq l_{\mathtt{A}}$, and $[\![e]\!]\langle c, c', c_u, \ell, \mathcal{Q}\rangle = \langle P_e,\, \lambda\overline{\pi \triangleright \tau\ z}.\, e'\rangle$, and there exists $\langle \Theta,\, \mathcal{M},\, \mathcal{E}\rangle$ such that $\mathcal{E} \models \langle c,\, \eta\rangle$, and $\Gamma \vdash \mathcal{M} \approx_{A \not\leq l_{\mathtt{A}}} M$. Then $\langle \Theta,\, \mathcal{M},\, \mathcal{E}\rangle \leadsto^* \langle \Theta',\, \mathcal{M}',\, \mathcal{E}'\rangle$ such that $\mathcal{E}' \models \langle c',\, \eta\rangle$ and $\mathcal{E}' \models \langle c_u,\, \eta\rangle.\overline{z}$.

Proof. By induction on the structure of e.

- e is n, m, or $!m$ with $\Gamma(m) = \sigma$. In this case, $[\![e]\!]\langle c, c', c_u, \ell, H\rangle = e$ and $c = c'$. Thus, $\mathcal{E} \models \langle c',\, \eta\rangle$ and $\mathcal{E}' \models \langle c_u,\, \eta\rangle.\overline{z}$ immediately hold.
- e is $!m$, with $\Gamma(m) = \sigma@\mathcal{Q}$. By rule (TE3), $P_e = \{r\}$ and

  $$r = c\{\ell,\ \mathcal{Q},\ \lambda.\mathtt{exec}(\mathtt{read}[\ell, \ell_1, m, c', \mathtt{cid}, \langle c_u,\ \mathtt{cid}\rangle.z],\, \mathtt{nid},\, \ell,\, \oslash,\, \epsilon)\}.$$

  Since $\mathcal{E} \models \langle c,\, \eta\rangle$ holds, we have $\langle \Theta,\, \mathcal{M},\, \mathcal{E}\rangle \leadsto^* \langle \Theta_1,\, \mathcal{M}_1,\, \mathcal{E}_1\rangle$ such that $\mathcal{E}_1 \models \langle \mathtt{read}[\ell, \ell_1, m, c', \eta, \langle c_u,\, \eta\rangle.z],\, \eta'\rangle$ where $\eta' = \mathcal{E}(c.\mathtt{nid}, \eta)$. By $A(\mathcal{R}) \not\leq l_{\mathtt{A}}$ and rule (DM), $A(\mathcal{Q}) \not\leq l_{\mathtt{A}}$, which means that at least a $\mathtt{QR}[\mathcal{Q}, I(\ell)]$-qualified set of hosts in $\mathcal{Q}$ are available to finish executing the read reactor. Therefore, we have $\langle \Theta_1,\, \mathcal{M}_1,\, \mathcal{E}_1\rangle \leadsto^* \langle \Theta',\, \mathcal{M}',\, \mathcal{E}'\rangle$ such that $\mathcal{E}' \models \langle c',\, \eta\rangle$ and $\mathcal{E}' \models \langle c_u,\, \eta\rangle.z$.
- e is $e_1 + e_2$. By induction, $\langle \Theta,\, \mathcal{M},\, \mathcal{E}\rangle \leadsto^* \langle \Theta_1,\, \mathcal{M}_1,\, \mathcal{E}_1\rangle$ such that $\mathcal{E}_1 \models \langle c_1,\, \eta\rangle$ and $\mathcal{E}_1 \models \langle c_u,\, \eta\rangle.\overline{z_1}$. Again, by induction, $\langle \Theta_1,\, \mathcal{M}_1,\, \mathcal{E}_1\rangle \leadsto^* \langle \Theta',\, \mathcal{M}',\, \mathcal{E}'\rangle$ such that $\mathcal{E}' \models \langle c',\, \eta\rangle$ and $\mathcal{E}' \models \langle c_u,\, \eta\rangle.\overline{z_2}$.

□

Lemma 6.5.4 (Control transfer availability). Suppose $[\![(c, w)]\!]\Psi' = \langle s, \Psi\rangle$, and there exists a run $\langle \Theta_0,\, \mathcal{M}_0,\, \mathcal{E}_0\rangle \longmapsto^* \langle \Theta,\, \mathcal{M},\, \mathcal{E}\rangle$ such that $\mathcal{E}\,;\eta \models \Psi$, and $\mathcal{E} \models \langle c_1,\, \eta\rangle$, and the body of c_1 ends with s, and $A(c_1) \not\leq l_{\mathtt{A}}$. Then $\langle\Theta,\, \mathcal{M},\, \mathcal{E}\rangle \leadsto^* \langle\Theta',\, \mathcal{M}',\, \mathcal{E}'\rangle$ such that $\mathcal{E}'\,;\eta' \models \Psi'$.

Proof. By inspecting rules (TC1) and (TC2). $\square$

Lemma 6.5.5 (Availability preservation I). Suppose $\Gamma\,;\mathcal{R}\,;pc \vdash S : \mathtt{stmt}_{\mathcal{R}'}$, and $I(pc) \leq l_{\mathtt{A}}$ and $A(\mathcal{R}) \not\leq l_{\mathtt{A}}$, and $P\,;\Psi \vdash S : \Psi'$, and $\langle\Theta,\, \mathcal{M},\, \mathcal{E}\rangle$ satisfies $\mathcal{E}\,;\eta \models \Psi$ and *available* $(\mathcal{M},\, \mathcal{R},\, l_{\mathtt{A}})$, which means that for any m in *dom*(Γ), $A(\Gamma(m)) \not\leq l_{\mathtt{A}}$ and $m \notin \mathcal{R}$ imply that m is available in $\mathcal{M}$. Then $\langle\Theta,\, \mathcal{M},\, \mathcal{E}\rangle \leadsto^* \langle\Theta',\, \mathcal{M}',\, \mathcal{E}'\rangle$ such that $\mathcal{E}'\,;\eta \models \Psi'$, and *available* $(\mathcal{M}',\, \mathcal{R}',\, l_{\mathtt{A}})$.

Proof. By induction on the structure of S.

- S is $\mathtt{skip}$. Since $\Psi' = \Psi$, $\langle\Theta,\, \mathcal{M},\, \mathcal{E}\rangle$ already satisfies the conditions.
- S is $\{c\}\,\mathtt{skip}$. The target code of S just invokes Ψ'. By Lemma 6.5.4, this lemma holds.
- S is $\{c\}\,m := e$. Then we have $[\![S]\!]\Psi' = \langle P_1, \Psi_1\rangle$, and $P_1 \subseteq P$. First, suppose $\Gamma(m) = \sigma$. By (TS2), $[\![e]\!]\langle c_1, c_1', \ell, H\rangle = \langle P_e, \lambda\overline{\pi \triangleright \tau\; z}.e'\rangle$. Since $A(\mathcal{R}) \not\leq l_{\mathtt{A}}$, we have $\langle e,\, M\rangle \Downarrow n$. By Lemma 6.5.3 and $\mathcal{E}\,;\eta \models \Psi_1$, we have $\langle\Theta,\, \mathcal{M},\, \mathcal{E}\rangle \leadsto^* \langle\Theta_1,\, \mathcal{M}_1,\, \mathcal{E}_1\rangle$ such that $\mathcal{E}_1 \models \langle c_1',\, \eta\rangle$. Suppose h' is the host where c_1' resides. By rule (DS), $A(m) \leq A(h')$. If $A(\mathcal{R}) \not\leq l_{\mathtt{A}}$, then $A(m) \not\leq l_{\mathtt{A}}$ and $A(h') \not\leq l_{\mathtt{A}}$, which means that h' is available. Since $\mathcal{R}'$ is $\mathcal{R} - \{m\}$, we have $\mathcal{R}' \vdash \mathcal{M}' \approx_{A \not\leq l_{\mathtt{A}}} M'$. By rule (TS2) and Lemma 6.5.4, $\langle\Theta_1,\, \mathcal{M}_1,\, \mathcal{E}_1\rangle \leadsto^* \langle\Theta',\, \mathcal{M}',\, \mathcal{E}'\rangle$ such that $\mathcal{E}'\,;\eta \models \Psi'$.
- S is $S_1; S_2$. By induction.
- S is $\{c\}\,\mathtt{if}\; e\; \mathtt{then}\; S_1\; \mathtt{else}\; S_2$. Since $A(\mathcal{R}) \not\leq l_{\mathtt{A}}$, $\langle e,\, M\rangle \Downarrow n$. Suppose $\Gamma\,;\mathcal{R} \vdash S : \ell$, and $\mathcal{Q}_c = \langle H, \emptyset\rangle$. Then $A(\mathcal{R}) \leq A(H, \mathtt{LT}[\ell])$. Since $A(\mathcal{R}) \not\leq l_{\mathtt{A}}$, there exists a $\mathtt{LT}[\ell]$-qualified subset H' of H such that $A_{\sqcap}(H') \not\leq l_{\mathtt{A}}$. Therefore, there

exists a subset H'' of H' such that $I(\ell) \leq I(H'')$ and all the hosts of H'' takes the same branch. Without loss of generality, suppose the first branch is taken. Then by (TS5) and Lemma 6.5.4, $\langle \Theta,\, \mathcal{M},\, \mathcal{E} \rangle \rightsquigarrow^* \langle \Theta'',\, \mathcal{M}'',\, \mathcal{E}'' \rangle$ such that $\mathcal{E}''\,;\eta \models \Psi''$ and $\Psi'' \vdash S_1 : \Psi'$. By induction, $\langle \Theta'',\, \mathcal{M}'',\, \mathcal{E}'' \rangle \rightsquigarrow^* \langle \Theta',\, \mathcal{M}',\, \mathcal{E}' \rangle$ such that $\mathcal{E}'\,;\eta \models \Psi'$.

- S is `while` e `do` S'. By the typing rule (WHILE) of Aimp, $I(pc) \leq l_{\mathtt{A}}$ implies $A(\mathcal{R}) \leq l_{\mathtt{A}}$. Thus, this case cannot occur.

□

According to the translation soundness theorem, for a run of the source program $\langle S,\, M \rangle \longmapsto^* \langle S',\, M' \rangle$, there is a benchmark run of the target program that behaves similar to the source run. Therefore, we can associate each evaluation step of the source program with the context identifier of the corresponding evaluation step in the benchmark target execution, and use the notation $\langle S_1,\, M_1 \rangle_{\eta_1} \longmapsto \langle S_2,\, M_2 \rangle_{\eta_2}$ to denote that η_1 and η_2 are the corresponding context identifier of configurations $\langle S_1,\, M_1 \rangle$ and $\langle S_2,\, M_2 \rangle$.

Lemma 6.5.6 (Availability preservation II). Suppose $\Gamma\,;\mathcal{R}\,;pc \vdash S : \mathtt{stmt}_{\mathcal{R}'}$ and $I(pc) \not\leq l_{\mathtt{A}}$ and $A(\mathcal{R}) \not\leq l_{\mathtt{A}}$ and $\langle S,\, M \rangle_\eta \longmapsto \langle S_1,\, M_1 \rangle_{\eta'}$, and $P\,;\Psi \vdash S : \Psi'$, and $\langle \Theta,\, \mathcal{M},\, \mathcal{E} \rangle$ satisfies $\mathcal{E}\,;\eta \models \Psi$ and $\Gamma \vdash \mathcal{M} \approx_{A \not\leq l_{\mathtt{A}}} M$. Then $\langle \Theta,\, \mathcal{M},\, \mathcal{E} \rangle \rightsquigarrow^* \langle \Theta_2,\, \mathcal{M}_2,\, \mathcal{E}_2 \rangle$ such that $\mathcal{E}_2\,;\eta' \models \Psi_2$, and $\Psi_2 \vdash S_2 : \Psi'$, and $\Gamma \vdash \mathcal{M}_2 \approx_{A \not\leq l_{\mathtt{A}}} M_1$, and $S_1 \approx S_2$, which means either $S_1 = S_2$ or for $i \in \{1,2\}$, $S_i = S_i'; S''$ such that $\Gamma\,;\mathcal{R}\,;pc \vdash S_i' : \mathtt{stmt}_{\mathcal{R}}'$ and $I(pc) \leq L$.

Proof. By induction on $\langle S,\, M \rangle \longmapsto \langle S',\, M' \rangle$. Without loss of generality, suppose $[\![S]\!]\Psi' = \langle P, \Psi \rangle$. In general, $[\![S]\!]\Psi'' = \langle P, \Psi_1 \rangle$ and $\Psi = \Psi_1, \Psi_3$ and $\Psi' = \Psi'' \otimes \Psi_3$. If the theorem holds for $\Psi_1 \vdash S : \Psi''$, then we have $\Psi_2 \vdash S_2 : \Psi''$. Therefore, $\Psi_2, \Psi_3 \vdash S_2 : \Psi'' \otimes \Psi_3$, that is, $\Psi_2' \vdash S_2 : \Psi'$.

- Case (S1). S is $m := e$, and $M_1 = M[m \mapsto v]$ where $\langle e,\ M\rangle \Downarrow v$. There are two cases. First, $\Gamma(m) = \sigma$. By (TS2), $[\![e]\!]\langle c, c_1, \ell, H\rangle = \langle P_e, \lambda\overline{\pi \triangleright \tau\ z}.e'\rangle$, and the first element of Ψ is c. By Lemma 6.5.3 and $\mathcal{E}\,;\eta \vDash \Psi$, we have $\langle\Theta,\ \mathcal{M},\ \mathcal{E}\rangle \rightsquigarrow^* \langle\Theta_1,\ \mathcal{M}_1,\ \mathcal{E}_1\rangle$ such that $\mathcal{E}_1\,;\eta \vDash c_1$. By (TS2), the code of c_1 is $m := e'; s'$ where $[\![c]\!]\Psi = \langle s', \Psi'\rangle$. Suppose h_1 is the host where c_1 resides. By rule (DM), $A(m) \le A(h_1)$. Since $A(\mathcal{R}) \not\le l_{\mathtt{A}}$, we have $A(h_1) \not\le l_{\mathtt{A}}$, which means that h_1 is available to finish executing the thread of $\langle c_1, \eta\rangle$. Since m is the only location updated in this evaluation step, and m is also updated during executing the target program, we have $\Gamma' \vdash \mathcal{M}_2 \approx_{A \not\le l_{\mathtt{A}}} M_1$. By rule (TS2), $[\![c]\!]\Psi' = \langle s', \Psi\rangle$. By Lemma 6.5.4, $\langle\Theta_1,\ \mathcal{M}_1,\ \mathcal{E}_1\rangle \rightsquigarrow^* \langle\Theta',\ \mathcal{M}',\ \mathcal{E}'\rangle$ in finite steps such that $\mathcal{E}'\,;\eta' \vDash \Psi'$. In addition, S_2 is `skip`, and $\Psi' \vdash$ `skip` $: \Psi'$.

  Second, $\Gamma(m) = \sigma@\mathcal{Q}_m$. By rule (DS), $A(\mathcal{R}) \le A(H, \mathtt{LT}[I(m)])$. As a result, at least a $\mathtt{LT}[I(m)]$-qualified subset H' of H are available to invoke $\mathtt{write}[\ell, m, c_2, \eta]$. Since $A(\ell) \not\le l_{\mathtt{A}}$, at least a quorum of $\mathcal{Q}_m$ is available. The available quorum is able to finish executing the `write` reactor and invoke c_2 on $\mathcal{Q}$. By rule (TS1), the code of c_2 is s'. Due to $A(\ell) \not\le l_{\mathtt{A}}$, the available hosts in $\mathcal{Q}$ have sufficient integrity so that the remote requests sent by s' would be accepted. By Lemma 6.5.4, $\langle\Theta_1,\ \mathcal{M}_1,\ \mathcal{E}_1\rangle \rightsquigarrow^* \langle\Theta',\ \mathcal{M}',\ \mathcal{E}'\rangle$ such that $\mathcal{E}'\,;\eta' \vDash \Psi'$.

- Case (S2). S is $S_1; S_2$, and $\langle S_1,\ M\rangle \longmapsto \langle S_1'',\ M'\rangle$. By $\Psi \vdash S : \Psi'$, we have $\Psi \vdash S_1 : \Psi_1$, and $\Psi_1 \vdash S_2 : \Psi'$. By induction, $\langle\Theta,\ \mathcal{M},\ \mathcal{E}\rangle \rightsquigarrow^* \langle\Theta_2,\ \mathcal{M}_2,\ \mathcal{E}_2\rangle$ such that $\mathcal{E}_2\,;\eta \vDash \Psi_2$, and $\Psi_2 \vdash S_1' : \Psi_1$ and $S_1 \approx S_1'$. Therefore, $S_1; S_2 \approx S_1'; S_2$, and $\Psi_2 \vdash S_1'; S_2 : \Psi'$.

- Case (S3). If S is $\{c\}$ `skip`$; S_2$, the conclusions immediately hold by Lemma 6.5.4. Otherwise, S is `skip`$; S_2$. Thus, $S_1 = S_2$, and $P\,;\Psi \vdash S_2 : \Psi'$ since $P\,;\Psi \vdash$ `skip` $: \Psi$.

- Case (S4). S is `if` e `then` S_1 `else` S_2, and $\langle e,\ M\rangle \Downarrow n$ and $n > 0$. By

Lemma 6.5.3, $\langle \Theta,\, \mathcal{M},\, \mathcal{E}\rangle \rightsquigarrow^* \langle \Theta_1,\, \mathcal{M},\, \mathcal{E}_1\rangle$ such that $\mathcal{E}_1\,;\eta \models c_1$. By Theorem 6.5.1, there exists a benchmark execution $\langle \Theta_0,\, \mathcal{M}_0,\, \mathcal{E}_0\rangle \longmapsto^* \langle \Theta_2,\, \mathcal{M}_2,\, \mathcal{E}_2\rangle$ such that $\langle e'[\mathcal{A}_{c_1,\eta}],\, \mathcal{M}_2\rangle \Downarrow n$. If $I(e) \not\leq L$, then by Theorem 5.4.2, for any h in $\mathcal{Q}_{c_1}$, $\langle e'[\mathcal{A}_{c_1,\eta}],\, \mathcal{M}(h,t)\rangle \Downarrow n$, and the execution takes the branch s'_1. By Lemma 6.5.4, $\langle \Theta_1,\, \mathcal{M},\, \mathcal{E}_1\rangle \rightsquigarrow^* \langle \Theta',\, \mathcal{M}',\, \mathcal{E}'\rangle$ such that $\mathcal{E}'\,;\eta \vdash \Psi_2$ where $[\![S_1]\!]\Psi' = \langle P_1, \Psi_2\rangle$.

If $I(e) \leq L$, attackers may be able to compromise the integrity of e and make the execution to take the second branch. In that case, we have $\langle \Theta_1,\, \mathcal{M},\, \mathcal{E}_1\rangle \rightsquigarrow^* \langle \Theta',\, \mathcal{M},\, \mathcal{E}'\rangle$ such that $\mathcal{E}'\,;\eta \models \Psi_2$ and $P\,;\Psi_2 \vdash S_2 : \Psi'$. Furthermore, $S_1 \approx S_2$ since $I(e) \leq L$.

- Case (S5). By the same argument as case (S4).

- Case (S6). S is `while` e `do` S_1, $\langle e,\, M\rangle \Downarrow n$, $n > 0$, and S' is S_1;`while` e `do` S_1. By Lemma 6.5.3, $\langle \Theta,\, \mathcal{M},\, \mathcal{E}\rangle \rightsquigarrow^* \langle \Theta',\, \mathcal{M}',\, \mathcal{E}'\rangle$ such that $\mathcal{E}'\,;\eta \models c_1$. Moreover, $A(\mathcal{R}) \not\leq l_{\mathtt{A}}$ implies $I(e) \not\leq l_{\mathtt{A}}$. By Theorem 6.5.1, for any h in $\mathcal{Q}(c_1)$ such that $I(h) \not\leq l_{\mathtt{A}}$, $\langle e'[\mathcal{A}_{c_1,\eta}],\, \mathcal{M}'(h,t)\rangle \Downarrow n$. Since $n > 0$, "`setvar`$(\langle c_1,\, \mathtt{nid}\rangle.z',\, z')$; s_1" is executed on host h. By executing `setvar`$(\langle c_1,\, \mathtt{nid}\rangle.z',\, z')$ and processing the messages the statement, $\langle \Theta',\, \mathcal{M}',\, \mathcal{E}'\rangle \rightsquigarrow^* \langle \Theta_1,\, \mathcal{M}_1,\, \mathcal{E}_1\rangle$ such that $\mathcal{E}_1\,;\eta' \models \Psi_2$. By executing s_1 and processing the messages sent by s_1, $\langle \Theta_1,\, \mathcal{M}_1,\, \mathcal{E}_1\rangle \rightsquigarrow^* \langle \Theta_2,\, \mathcal{M}_2,\, \mathcal{E}_2\rangle$ such that $\mathcal{E}_2\,;\eta' \models \Psi'$.

- Case (S7). S is `while` e `do` S_1, $\langle e,\, M\rangle \Downarrow n$, $n \leq 0$, and S' is `skip`. By Lemma 6.5.3, $\langle \Theta,\, \mathcal{M},\, \mathcal{E}\rangle \rightsquigarrow^* \langle \Theta_1,\, \mathcal{M},\, \mathcal{E}_1\rangle$ such that $\mathcal{E}''\,;\eta \models c_1$. Since $I(e) \not\leq l_{\mathtt{A}}$, for any h in $\mathcal{Q}_{c_1}$ such that $I(h) \not\leq l_{\mathtt{A}}$, $\langle e'[\mathcal{A}_{c_1,\eta}],\, \mathcal{M}(h,t)\rangle \Downarrow n$, and s_2 is executed on h. Therefore, by Lemma 6.5.4, $\langle \Theta_1,\, \mathcal{M},\, \mathcal{E}_1\rangle \rightsquigarrow^* \langle \Theta',\, \mathcal{M},\, \mathcal{E}'\rangle$ such that $\mathcal{E}'\,;\eta' \models \Psi'$.

□

Theorem 6.5.2 (Availability preservation). Suppose $\Gamma\,;\mathcal{R}\,;pc \vdash S : \tau$, and $\langle S,\, M\rangle \longmapsto^* \langle S',\, M'\rangle$, and $[\![S]\!]\emptyset = \langle P, c\rangle$, and $M \approx \mathcal{M}$. Then $\langle \Theta_0,\, \mathcal{M},\, \mathcal{E}_0\rangle \rightsquigarrow^* \langle \Theta',\, \mathcal{M}',\, \mathcal{E}'\rangle$ such that $\Gamma \vdash \mathcal{M}' \approx_{A\not\leq l_{\mathtt{A}}} M'$

Proof. By induction on the number of steps of $\langle S,\, M\rangle \longmapsto^* \langle S',\, M'\rangle$, we can prove a stronger result: $\langle S',\, M'\rangle \longmapsto^* \langle S'',\, M''\rangle$ such that $\Gamma\,;\mathcal{R}''\,;pc'' \vdash S'' : \tau$ implies $I(pc'') \not\leq l_{\mathtt{A}}$, and and $\langle \Theta_0,\, \mathcal{M},\, \mathcal{E}_0\rangle \rightsquigarrow^* \langle \Theta',\, \mathcal{M}',\, \mathcal{E}'\rangle$ such that $\Gamma \vdash \mathcal{M}' \approx_{A\not\leq l_{\mathtt{A}}} M''$ and $\mathcal{E}'\,;\eta \models \Psi'$ where $\Psi' = \&S''$,

Suppose $\langle S,\, M\rangle \longmapsto^* \langle S_1,\, M_1\rangle \longmapsto \langle s',\, M'\rangle$. By induction, the result stated above holds for $\langle S_1,\, M_1\rangle \longmapsto^* \langle S_1',\, M_1'\rangle$. If $S_1 \neq S_1'$, then the result immediately holds for $\langle S',\, M'\rangle \longmapsto^* \langle S_1',\, M_1'\rangle$. Otherwise, we have that $\Gamma\,;\mathcal{R}_1\,;pc_1 \vdash \S_1 : \tau$ implies $I(pc_1) \not\leq l_{\mathtt{A}}$, and $\langle \Theta_0,\, \mathcal{M},\, \mathcal{E}_0\rangle \rightsquigarrow^* \langle \Theta_1,\, \mathcal{M}_1,\, \mathcal{E}_1\rangle$ such that $\Gamma \vdash \mathcal{M}_1 \approx_{A\not\leq l_{\mathtt{A}}} M_1$ and $\mathcal{E}_1\,;\eta \models \Psi_1$ where $\Psi_1 = \&S_1$. By Lemma 6.5.6, $\langle \Theta_1,\, \mathcal{M}_1,\, \mathcal{E}_1\rangle \longmapsto^* \langle \Theta_2,\, \mathcal{M}_2,\, \mathcal{E}_2\rangle$ such that $\Gamma \vdash \mathcal{M}_2 \approx_{A\not\leq l_{\mathtt{A}}} M'$ and $\mathcal{E}_2\,;\eta_2 \models \Psi_2$ where $\Psi_2 = \&S_2$ and $S' \approx S_2$. Suppose $S_2 = S_2'; S''$ such that $\Gamma\,;\mathcal{R}_2\,;pc_2 \vdash S_2' : \mathtt{stmt}_{\mathcal{R}'}$ and $I(pc_2) \leq l_{\mathtt{A}}$. By Lemma 6.5.5, $\langle \Theta_2,\, \mathcal{M}_2,\, \mathcal{E}_2\rangle \rightsquigarrow^* \langle \Theta',\, \mathcal{M}',\, \mathcal{E}'\rangle$ such that $\mathcal{E}'\,;\eta' \models \Psi''$ where $\Psi'' = \&S''$, and *available*$(\mathcal{M}',\, \mathcal{R}',\, l_{\mathtt{A}})$. Moreover, $\langle S_1,\, M_1\rangle \longmapsto^* \langle S'',\, M''\rangle$. Suppose $S_1 = S_1'; S''$. By $S_1 \approx S_2$, the high-availability memory references initialized by S_1' are also initialized by S_2'. Therefore, $\Gamma \vdash \mathcal{M}' \approx_{A\not\leq l_{\mathtt{A}}} M''$.

□

6.6 Related work

The closest work to the Aimp/DSR translation is the Jif/split system [104, 105] that introduced the secure program partitioning technique and automatic replication of code and data. However, the Jif/split system cannot provide strong availability assurance, and it does not have a formal correctness proof yet, due to its complexity.

Program slicing techniques [93, 85] provide information about the data dependencies in a piece of software. Although the use of backward slices to investigate integrity and related security properties has been proposed [26, 49], the focus of work on program slicing has been debugging and understanding existing software.

Using program transformation to enforce security policies is a widely used approach. The SFI (Software Fault Isolation) technique [89] enforces memory safety by a program transformation, which inserts *checking* or *sandboxing* code before every operation of the original program that may violate memory safety. The sandboxing code inserted before an operation updates the program state such that the operation would never violate memory safety and the update is equivalent to a no-op if the operation is safe in the original program. The checking code inserted before an operation determines whether the operation would violate memory safety and aborts execution if it would. The SFI technique has been applied to enforcing other safety properties [78, 22, 67]. Erlingsson and Schneider proposed the SASI framework [21] that generalizes SFI to any security policy that can be specified as a security automaton and enforced by a reference monitor [76].

The general program transformation approach has also been applied to implementing secure function evaluation and preventing timing channels. Fairplay [50] is a system implementing generic secure function evaluation. Fairplay uses a compiler to translate a two-party secure function specified in a high-level procedural language into low-level Boolean circuits evaluated in a manner suggested by Yao [98].

Agat [3] proposed a padding transformation that eliminates the timing channels in the source program with respect to a target execution model with observable timing information.

Chapter 7
Conclusions

This thesis proposes a unified approach to building distributed programs that enforce end-to-end confidentiality, integrity and availability policies, within a common framework of program analysis and transformation. The key innovative idea is that end-to-end availability policies can also be enforced by a form of noninterference and it is thus possible to apply the techniques for enforcing confidentiality and integrity policies (such as static information flow control and secure program partitioning) to enforcing end-to-end availability policies. Based on the idea, this thesis presents

- a universal decentralized label model for specifying end-to-end security policies,
- a sequential language Aimp with a security type system ensuring that a well-typed program enforces the security policies (including availability policies) specified as type annotations,
- a distributed language DSR that uses quorum replication and a form of multipart timestamps to enforce availability policies without sacrificing confidentiality and integrity, and
- a formal translation from Aimp to DSR, which allows programmers to use Aimp to develop applications running on distributed systems.

This thesis proves that if a well-typed Aimp program S is translated into a DSR program P with respect to a trust configuration, then the distributed target program P enforces the security policies of the source in the given trust configuration. Therefore, this constructive approach for building secure distributed programs is correct in theory. Whether this approach is practical is not addressed in this thesis because both Aimp and DSR are minimal in terms of language features. Nevertheless, previous work on the Jif/split system provides some evidence that the constructive approach is practical by showing more examples, including various auction applications and an implementation of the battleship game. To some extent, the theoretical result of this thesis and the experimental result of the Jif/split system complement each other, as the Jif/split compiler and the Aimp/DSR translator follow the same principles.

BIBLIOGRAPHY

[1] Martín Abadi, Michael Burrows, Butler W. Lampson, and Gordon D. Plotkin. A calculus for access control in distributed systems. *TOPLAS*, 15(4):706–734, 1993.

[2] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, 1996.

[3] Johan Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, January 2000.

[4] Divyakant Agrawal, Arthur J. Bernstein, Pankaj Gupta, and Soumitra Sengupta. Distributed optimistic concurrency control with reduced rollback. 2(1):45–59, March 1987.

[5] Lorenzo Alvisi, Dahlia Malkhi, Evelyn Pierce, Michael Reiter, and Rebecca N. Wright. Dynamic byzantine quorum systems. In *International Conference on Dependable Systems and Networks (DSN00)*, 2000.

[6] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In *The Eleventh International Symposium on Static Analysis Proceedings*, pages 100–115, Verona, Italy, 2004.

[7] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proc. 15th IEEE Computer Security Foundations Workshop*, June 2002.

[8] Rida A. Bazzi. Synchronous byzantine quorum systems. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of distributed computing (PODC 97)*, pages 259–266, Santa Barbara, California, United States, 1997.

[9] D. E. Bell and L. J. LaPadula. Secure computer systems: mathematical foundations and model. Technical Report M74-244, MITRE Corp., Bedford, MA, 1973.

[10] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975. Available as DTIC AD-A023 588.

[11] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group

multicast. In *ACM Transactions on Computer Systems*, volume 9(3), August 1991.

[12] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, 2002.

[13] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE Transactions on Software Engineering*, SE-11(2):205–212, February 1985.

[14] David Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proc. IEEE Symposium on Security and Privacy*, pages 184–194, 1987.

[15] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *Computing Surveys*, 17(3):341–370, September 1985.

[16] Dorothy E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, 1976.

[17] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[18] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[19] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD (The Orange Book) edition, December 1985.

[20] 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 *Proc. 20th ACM Symp. on Operating System Principles (SOSP)*, October 2005.

[21] Ulfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigm Workshop*, Caledon Hills, Ontario, Canada, 1999.

[22] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, May 1999.

[23] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a

system design. *Proc. 6th ACM Symp. on Operating System Principles (SOSP), ACM Operating Systems Review*, 11(5):57–66, November 1977.

[24] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.

[25] David Ferraiolo and Richard Kuhn. Role-based access controls. In *15th National Computer Security Conference*, 1992.

[26] George Fink and Karl Levitt. Property-based testing of privileged programs. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 154–163, Orlando, FL, 1994. IEEE Computer Society Press.

[27] Riccardo Focardi and Roberto Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1995.

[28] Simon Foley, Li Gong, and Xiaolei Qian. A security model of dynamic labeling providing a tiered approach to verification. In *IEEE Symposium on Security and Privacy*, pages 142–154, Oakland, CA, 1996. IEEE Computer Society Press.

[29] C. Fournet and G. Gonthier. The Reflexive CHAM and the Join-Calculus. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 372–385, 1996.

[30] D. K. Gifford. Weighted voting for replicated data. In *Proc. of the Seventh Symposium on Operating Systems Principles*, pages 150–162, Pacific Grove, CA, December 1979. ACM SIGOPS.

[31] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[32] James W. Gray, III. Probabilistic interference. In *Proc. IEEE Symposium on Security and Privacy*, pages 170–179, May 1990.

[33] Joshua D. Guttman et al. Trust management in strand spaces: A rely-guarantee method. In *Proc. European Symposium on Programming*, pages 325–339, April 2004.

[34] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, California, January 1998.

[35] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[36] C. A. R. Hoare. Communicating sequential processes. *Comm. of the ACM*, 21(8):666–677, August 1978.

[37] Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 81–92. ACM Press, June 2002.

[38] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, 1979.

[39] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Proc. 33th ACM Symp. on Principles of Programming Languages (POPL)*, pages 79–90, Charleston, South Carolina, USA, January 2006.

[40] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[41] Ari Juels and John Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of NDSS'99 (Network and Distributed System Security Symposium)*, pages 151–165, 1999.

[42] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of 19th Int'l Symp. on Computer Architecture*, pages 13–21, Queensland, Australia, May 1992.

[43] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services (extended abstract). In *Proceedings of the Fourth ACM European Workshop on Fault Tolerance Support in Distributed Systems*, Bologna, Italy, September 1990.

[44] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[45] Stephane Lafrance and John Mullins. Using admissible interference to detect denial of service vulnerabilities. In *6th International Workshop on Formal Methods*, pages 1–19, July 2003.

[46] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

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

[48] Mark C. Little and Daniel McCue. The Replica Management System: a scheme for flexible and dynamic replication. In *Proceedings of the 2nd International Workshop on Configurable Distributed Systems*, pages 46–57, Pittsburgh, March 1994.

[49] James R. Lyle, Dolores R. Wallace, James R. Graham, Keith. B. Gallagher, Joseph. P. Poole, and David. W. Binkley. Unravel: A CASE tool to assist evaluation of high integrity software. IR 5691, NIST, 1995.

[50] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay—a secure two-party computation system. In *Proceedings of the 13th Usenix Security Symposium*, pages 287–302, San Diego, CA, August 2004.

[51] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *Proc. of the 29th ACM Symposium on Theory of Computing*, pages 569–578, El Paso, Texas, May 1997.

[52] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Small byzantine quorum systems. In *International Conference on Dependable Systems and Networks (DSN02)*, June 2002.

[53] Catherine J. McCollum, Judith R. Messing, and LouAnna Notargiacomo. Beyond the pale of MAC and DAC—defining new forms of access control. In *Proc. IEEE Symposium on Security and Privacy*, pages 190–200, 1990.

[54] Daryl McCullough. Specifications for multi-level security and a hook-up property. In *Proc. IEEE Symposium on Security and Privacy*. IEEE Press, May 1987.

[55] M. Douglas McIlroy and James A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8):673–694, August 1992.

[56] John McLean. The algebra of security. In *IEEE Symposium on Security and Privacy*, pages 2–7, Oakland, California, 1988.

[57] John McLean. Security models and information flow. In *Proc. IEEE Symposium on Security and Privacy*, pages 180–187, 1990.

[58] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Security and Privacy*, pages 79–93, May 1994.

[59] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[60] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992.

[61] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. pages 271–283, 1996.

[62] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[63] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, Saint-Malo, France, 1997.

[64] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[65] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. Software release, at http://www.cs.cornell.edu/jif, July 2001–.

[66] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. In *Proc. 2nd International Symposium on Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 314–329. Springer, September 1995.

[67] Raju Pandey and Brant Hashii. Providing fine-grained access control for java programs. In *ECOOP '99 Conference Proceedings*, Lisbon, Portugal, June 1999.

[68] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, May 1983.

[69] Charles P. Pfleeger and Shari Lawrence Pfleeger. *Security in Computing*. Prentice Hall PTR, Upper Saddle River, New Jersey, third edition, 2003.

[70] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 319–330, 2002.

[71] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, 1972.

[72] Andrei Sabelfeld and Heiko Mantel. Static confidentiality enforcement for distributed programs. In *Proceedings of the 9th International Static Analysis Symposium*, volume 2477 of *LNCS*, Madrid, Spain, September 2002. Springer-Verlag.

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

[74] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[75] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[76] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2001. Also available as TR 99-1759, Computer Science Department, Cornell University, Ithaca, New York.

[77] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *LISP and Functional Programming*, pages 150–161, 1994.

[78] Christopher Small. A tool for constructing safe extensible c++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, pages 175–184, Portland, Oregon, USA, June 1997.

[79] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 355–364, San Diego, California, January 1998.

[80] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The flask security architecture: System support for diverse

security policies. In *The Eighth USENIX Security Symposium Proceedings*, pages 123–139, August 1999.

[81] Sun Microsystems. *Java Language Specification*, version 1.0 beta edition, October 1995. Available at `ftp://ftp.javasoft.com/docs/javaspec.ps.zip`.

[82] David Sutherland. A model of information. In *Proc. 9th National Security Conference*, pages 175–183, Gaithersburg, Md., 1986.

[83] Ian Sutherland, Stanley Perlo, and Rammohan Varadarajan. Deducibility security with dynamic level assignments. In *Proc. 2nd IEEE Computer Security Foundations Workshop*, Franconia, NH, June 1989.

[84] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[85] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[86] Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[87] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proc. 10th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.

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

[89] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symp. on Operating System Principles*, pages 203–216. ACM Press, December 1993.

[90] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the 9th Symposium on Operating Systems Principles*, Bretton Woods, NH, October 1983. ACM.

[91] David Walker, Lester Mackey, Jay Ligatti, George Reis, and David August. Static typing for a faulty lambda calculus. In *ACM SIGPLAN International Conference on Functional Programming*, September 2006. To appear.

[92] William E. Weihl. Distributed Version Management for Read-only Actions. *IEEE Transactions on Software Engineering*, SE-13(1):55–64, January 1987.

[93] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[94] Clark Weissman. Security controls in the ADEPT-50 time-sharing system. In *AFIPS Conference Proceedings*, volume 35, pages 119–133, 1969.

[95] John P. L. Woodward. Exploiting the dual nature of sensitivity labels. In *IEEE Symposium on Security and Privacy*, pages 23–30, Oakland, California, 1987.

[96] Hongwei Xi. Imperative programming with dependent types. In *Proceedings of 15th Symposium on Logic in Computer Science*, Santa Barbara, June 2000.

[97] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 214–227, San Antonio, TX, January 1999.

[98] A. Yao. How to generate and exchange secrets. In *Proceedings of the twenty-seventh annual IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.

[99] Che-Fn Yu and Virgil D. Gligor. A specification and verification method for preventing denial of service. In *Proc. IEEE Symposium on Security and Privacy*, pages 187–202, Oakland, CA, USA, April 1988.

[100] Aris Zakinthinos and E. Stewart Lee. A general theory of security properties and secure composition. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, 1997.

[101] Steve Zdancewic and Andrew C. Myers. Secure information flow and CPS. In *Proc. 10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 46–61, 2001.

[102] Steve Zdancewic and Andrew C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2–3):209–234, September 2002.

[103] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *Proc. 16th IEEE Computer Security Foundations Workshop*, pages 29–43, Pacific Grove, California, June 2003.

[104] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.

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

[106] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. In *Proc. 2nd Workshop on Formal Aspects in Security and Trust, IFIP TC1 WG1.7*. Springer, August 2004.

[107] Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. Technical Report 2004–1924, Cornell University Computing and Information Science, 2004.