

# Making Distributed Computation Trustworthy by Construction

Lantian Zheng   Andrew C. Myers  
Computer Science Department  
Cornell University  
{zlt,andru}@cs.cornell.edu

## Abstract

Trustworthy computing systems must provide data confidentiality and data integrity, and must be available. This paper shows that these security properties can be provided by construction, by compiling high-level, security-typed source code into explicitly distributed, security-typed target code. This code transformation provably preserves the confidentiality, integrity, and availability properties of the source. A key technical contribution is the new target language, which describes distributed computation. In this language, any well-typed program satisfies noninterference properties that ensure confidentiality and integrity. Further, the language supports the distribution and replication of code and data using quorum replication, which enables simultaneous enforcement of integrity and availability. A novel timestamp scheme handles out-of-order accesses by concurrent distributed threads without creating covert channels.

## 1 Introduction

Distributed computing systems are ubiquitous, yet we lack techniques for building them to be trustworthy. A system is *trustworthy* when we have assurance that it will protect the confidentiality and integrity of data it manipulates, and that the system will remain available. We use the term “trustworthy” to emphasize that the notion of security used here includes all three core security properties: confidentiality, integrity, and availability. Many different mechanisms are used to build trustworthy systems: for example, access controls, replica coordination protocols, and cryptography. However, standard practice for designing distributed systems does not give any direct assurance that security properties are enforced.

Our goal is systems that are *trustworthy by construction*; we aim for a construction process that uses formal, high-level security requirements (including availability requirements) as inputs to guide the automatic synthesis of trustworthy distributed systems. These systems should be trustworthy in the presence of an attacker who has some power to observe, to modify, and to stop computation taking place on some host machines in the system. Attacks on the system are assumed to take place at host machines. Network-level attacks are not considered, although some network attacks can be viewed as attacks on the affected hosts; other mechanisms (e.g., encryption) can help too.

In this approach, programmers use a high-level language to write programs that are not explicitly distributed, but that instead contain annotations (labels) specifying security requirements. It is the job of the compiler to translate these high-level programs into trustworthy distributed systems satisfying the security requirements. Earlier work on the Jif/split system [27, 28] explored security by construction, using confidentiality and integrity policies to guide automatic partitioning and replication of code and data onto a distributed system. But Jif/split could not express or enforce availability properties, did not formalize translation, and consequently had only limited formal validation.

This paper shows how to enforce not only confidentiality and integrity, but also simple availability properties, within a common framework of program analysis and rewriting. This is the first unified enforcement

scheme for all three major security properties in a distributed setting. Jif/split does not enforce availability—in fact, its simple replication mechanisms can only reduce availability.

The paper makes the following specific contributions:

- A security-typed language, DSR (for Distributed Secure Reactors), is introduced to represent distributed, concurrent computation on replicated hosts. The type system of this language enforces data confidentiality and integrity (specifically, it enforces noninterference [9]) even when low-integrity components of the system are controlled by the attacker.
- A translation is given from a simple, high-level, sequential source language to DSR. The translation formalizes the kind of translation done by Jif/split, while adding new mechanisms for enforcing availability. Perhaps most importantly, the translation is shown to result in a trustworthy system, because the translation preserves typing for confidentiality and integrity, and preserves availability properties as well.
- To make this translation possible, quorum replication [8, 11] 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.

The remainder of this paper is organized as follows. Section 2 describes how the decentralized label model [16] is extended to specify availability and explains how to interpret these trustworthiness specifications as properties of the distributed system. Section 3 describes the semantics of Aimp. Section 4 presents a distributed system model and various mechanisms needed to obtain trustworthy computation. Section 5 introduces a security-typed language (DSR) for developing distributed programs. Section 6 proves the non-interference result of DSR. Section 7 presents a translation from a sequential program to a distributed DSR program, and prove the adequacy of the translation. Section 8 covers related work, and Section 9 concludes.

## 2 Overview

### 2.1 Labels and security assurance

Security requirements are explicitly defined using *labels* that annotate data items, memory locations, computations, and host machines with policies. The label on a host specifies the strongest policies that the host is trusted to enforce. Static dependency analysis is used to determine whether an annotated program enforces its security policy annotations.

To take into account all the three core security properties, a security label  $\ell = \{C = l_1, I = l_2, A = l_3\}$  contains three base labels  $l_1, l_2$  and  $l_3$ , specifying confidentiality, integrity and availability policies, respectively. Furthermore, all the base labels form a lattice  $\mathcal{L}$ , where the ordering  $l \leq l'$  means that  $l'$  imposes requirements as least as strong as  $l$ , and the usual join ( $\sqcup$ ) and meet ( $\sqcap$ ) operations exist. The precise structure of  $\mathcal{L}$  is unimportant to the results in this paper. As usual in type systems for information flow,  $\ell \sqsubseteq \ell'$  denotes that data labeled with  $\ell$  is allowed to affect data labeled with  $\ell'$ , defined as  $C(\ell) \leq C(\ell') \wedge I(\ell') \leq I(\ell)$ : low-confidentiality information can flow to higher-confidentiality levels, and high-integrity information can flow to lower. Availability does not appear in the definition of  $\sqsubseteq$  because low-availability data can affect high-availability data without reducing its availability.

Labels express security requirements and, if enforced, constrain system behavior. Information remains confidential if attackers cannot learn it by observing the system. Information has integrity if it is computed as intended in the program. And information is available if it is received eventually, assuming that network

messages from uncompromised hosts arrive eventually. These informal statements can be understood as non-interference properties [9]. Intuitively, high-confidentiality inputs cannot interfere with low-confidentiality outputs, and attackers cannot affect high-integrity outputs or make high-availability outputs unavailable [29].

The notions of low and high security are relative to the attacker’s power, which is specified by a base label  $l_A$ . Any base label  $l$  is a *low-security label* if  $l \leq l_A$ , and a *high-security label* otherwise. It is important not to assume that there are only two security levels (e.g., “high” and “low”), because distributed systems need to be trustworthy from the viewpoint of all their users, who may not trust each other fully. We assume the attacker can directly observe all information at confidentiality levels  $l \leq l_A$ , can arbitrarily modify information at integrity levels  $l \leq l_A$ , and can make resources at availability levels  $l \leq l_A$  unavailable. Further, attackers can compromise hosts whose host label components are lower-security than  $l_A$  (more precisely, they can compromise the corresponding aspects of security on those hosts). The system is trustworthy if no attacker, regardless of its power  $l_A$ , can violate policies expressed by labels that are high-security relative to  $l_A$ . For example, the attacker should not be able to leverage its ability to directly modify low-security locations to indirectly cause changes to high-integrity locations or to make high-availability locations unavailable.

## 2.2 The Aimp language

The source language for the distributed system construction process is the simple sequential language Aimp (for Availability + IMP) [29]. Aimp supports specifying high-level policies for confidentiality, integrity, and availability, and its type system ensures that a well-typed program enforces these policies. The syntax of Aimp is as follows:

Values	$v ::= n \mid \text{none}$
Expressions	$e ::= v \mid !m \mid e_1 + e_2$
Statements	$S ::= \text{skip} \mid m := e \mid S_1; S_2$ $\quad \mid \text{if } e \text{ then } S_1 \text{ else } S_2 \mid \text{while } e \text{ do } S$
Types	$\tau ::= \text{int}_\ell \mid \text{int}_\ell \text{ ref} \mid \text{stmt}_\mathcal{R}$

A memory location  $m$  is essentially a mutable variable. It can take the special value none to represent an unavailable value. In Aimp, inputs and outputs are modeled by memory locations, so a location representing an output is considered unavailable if its value is none.

Type  $\text{int}_\ell$  is an integer annotated with security label  $\ell$ . A memory location  $m$  has type  $\text{int}_\ell \text{ ref}$ , indicating the value stored at  $m$  has type  $\text{int}_\ell$ . A statement  $S$  in Aimp has type  $\text{stmt}_\mathcal{R}$  where  $\mathcal{R}$  contains the set of output locations left unassigned and hence unavailable if  $S$  terminates.

The operational semantics and type system of Aimp are straightforward [29] and are not repeated here. However, it is worth noting that to control implicit flows [5], the typing environment of an Aimp statement includes a *program counter label*  $pc$ , which is the security label on information about the current program point. It captures both the integrity and the sensitivity of control flow information.

## 2.3 Example

Figure 1 shows a simple bidding program involving a seller S, a buyer B, and a mediator T. The buyer has three chances to bid for an item. If a bid price is above the reserve price of the seller, the buyer is required to confirm the transaction. If the buyer confirms, its credit card is charged the bid price; otherwise, the reputation of the buyer is negatively affected for not honoring a bid. For the purpose of presentation, this example uses an array `bid`, where `bid[t]` can be viewed as syntactic sugar for three locations `bid1`, `bid2` and `bid3` that are accessed based on the value of `t`. At the end of the transaction, an exit code is assigned to `exit`, which initially has the value none.

Security types of memory locations are provided by a typing assignment  $\Gamma$ , which maps locations to types such that location  $m$  has type  $\Gamma(m) \text{ ref}$ . For this example, security labels are specified using the

```

1 t := 0;
2 while (!t < 3) do
3   price := bid[t];
4   if (!price > !reserve) then t := 5
5   else t := !t + 1;
6 if (!t = 5) then
7   if (!cfm = 1) then cc := !cc + !price
8   else rep := !rep - 1;
9   exit := 1
10 else exit := 0

```

Typing assignment  $\Gamma$ :

$$\begin{aligned}
& \text{price, reserve, t, cfm, exit} : \text{int}_{\ell_0} \quad \text{cc} : \text{int}_{\ell_1} \quad \text{rep} : \text{int}_{\ell_2} \\
& \ell_0 = \{C = * : S \vee B, I = * : (S \wedge B) \vee (S \wedge T) \vee (B \wedge T), A = l\} \\
& \ell_1 = \{C = * : B, I = * : B \vee (S \wedge T), A = l\} \\
& \ell_2 = \{C = * : S, I = * : S \vee (B \wedge T), A = l\}
\end{aligned}$$

Figure 1: Bidding example

extended decentralized label model (DLM) [29]. In the DLM, a base label is a set of *owned policies*, which have the form  $u : p$ , where  $u$  is the policy owner, and  $p$  is a principal that  $u$  trusts with enforcing the security property that the owned policy is applied to. It is possible to construct more complex principals using conjunction and disjunction operators  $\wedge$  and  $\vee$ . The composite principal  $p_1 \wedge p_2$  represents a principal as trustworthy as both  $p_1$  and  $p_2$ ; principal  $p_1 \vee p_2$  represents a group of  $p_1$  and  $p_2$ .

For example, consider the label  $\ell_0$  of `price`. In  $\ell_0$ , principal  $*$  represents the conjunction  $S \wedge B \wedge T$ , and  $T$  represents a third party helping to mediate the transaction. The confidentiality label means that both  $S$  and  $B$  can learn the contents of these locations. The integrity label indicates that affecting the values in these locations requires the cooperation of at least two parties. For example,  $S \wedge B$  can cooperatively affect `price`, since they are the two directly involved parties. If  $S$  and  $B$  disagree on the value of `price`, the mediator  $T$  can keep the transaction going by agreeing with the value claimed by either  $S$  or  $B$ . As a result, both  $B \wedge T$  and  $S \wedge T$  can affect the value of `price`. The availability component  $A(\ell_0)$  is left unspecified but given the name  $l$ . Suppose `exit` is the only output location. Because all the variables share the same availability policy  $l$ , the availability policy of `exit` cannot be violated by making other variables unavailable.

Programs create interactions between availability and integrity. In this example, the value of  $t$  affects the termination of the `while` statement, and thus the availability of `exit`. Therefore, the program is insecure unless the constraint  $A(\ell_0) \leq I(\ell_0)$  holds, intuitively meaning that any principal trusted with the integrity of  $t$  is also trusted with the availability of `exit`. More generally, the constraint  $A(\ell) \leq I(\ell)$  means that in any situation where the availability of  $l$  is compromised, the integrity of  $l$  can be too.

## 2.4 Trustworthiness by construction

The type system of Aimp can guarantee that a well-typed program is trustworthy, assuming the program is executed on a single trusted host. But we are interested in performing the computation of an Aimp program  $S$  in a distributed system with untrusted hosts, while enforcing the security policies of  $S$ .

In a distributed setting, a location can be replicated onto a set of hosts to improve its integrity and availability. Suppose the integrity labels of hosts  $h_1$ ,  $h_2$  and  $h_3$  are  $* : S$ ,  $* : B$  and  $* : T$ , respectively. Then replicating `price` on hosts  $h_1$  and  $h_2$  is sufficient to enforce its integrity: a program reading the value of `price` can request it from  $h_1$  and  $h_2$ , and accept the value only if the responses from the two hosts

are equal. To convince the reader to accept a corrupted value, attackers need to compromise both  $h_1$  and  $h_2$ , but that means attackers have the power of principal  $S \wedge B$  and are allowed to update `price` by  $I(\ell_0)$ . However, if attackers compromise the integrity of either  $h_1$  or  $h_2$ , they can make `price` unavailable by sending the reader an inconsistent value. Higher availability can be achieved by replicating `price` on all the three hosts. A reader would accept its value if any *two* hosts agree. Interestingly, the reader can continue its computation if it receives three inconsistent responses. In that case, at least two hosts are compromised. For this to be true, the attackers must have the power of principal  $S \wedge B$ , of  $B \wedge T$ , or of  $S \wedge T$ . Thus, the label  $I(\ell_0)$  permits attackers to directly update `price`, and therefore the inconsistent values do not violate the security guarantee. Therefore, the reader can choose an arbitrary value as the value of `price` and continue its computation.

Our approach is to translate a well-typed Aimp program into a distributed target program that faithfully and securely performs the computation of the source program. To define this translation formally, we first present a target language DSR whose type system enforces noninterference for confidentiality and integrity in distributed settings. Then we present an adequate translation from Aimp to DSR, which preserves the typing and semantics of the source program. Suppose a well-typed source program  $S$  is translated into a target program  $P$ . By the preservation of typing,  $P$  is also well-typed, which means that  $P$  also satisfies confidentiality and integrity noninterference. The adequacy of the translation means  $P$  makes as much progress as the source program, and thus provides at least the same availability guarantees as  $S$ .

### 3 Semantics of Aimp

This section describes the operational semantics and type system of the source language Aimp.

#### 3.1 Operational semantics

The small-step operational semantics of Aimp is given in Figure 2. Let  $M$  represent a memory that is a finite map from locations to values (including `none`), 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  $\langle S, M \rangle \mapsto \langle S', M' \rangle$ .

The evaluation rules (S1)–(S7) are standard for an imperative language. Rules (E1)–(E3) are used to evaluate expressions. Because an expression has no side-effect, 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 dereference expression  $!m$ . In rule (E2),  $v_1 + v_2$  is computed using the following formula:

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

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  $\langle e, M \rangle \Downarrow \text{none}$ , the evaluation of  $\langle S, M \rangle$  gets stuck.

#### 3.2 Examples

By its simplicity, the Aimp language helps focus on the essentials of an imperative language. Figure 3 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  $m_1$  is unavailable, the execution gets stuck at the first assignment. Therefore, the availability of  $m_o$  depends on the availability of  $m_1$ .

In code segment (B), the `while` statement gets stuck if  $m_1$  is unavailable. Moreover, it diverges if the value of  $m_1$  is positive. Thus, the availability of  $m_o$  depends on both the availability and the value of  $m_1$ .

$$\begin{array}{l}
[E1] \quad \frac{m \in \text{dom}(M)}{\langle !m, M \rangle \Downarrow M(m)} \\
[E2] \quad \frac{\langle e_1, M \rangle \Downarrow v_1 \quad \langle e_2, M \rangle \Downarrow v_2 \quad v = v_1 + v_2}{\langle e_1 + e_2, M \rangle \Downarrow v} \\
[E3] \quad \langle v, M \rangle \Downarrow v \\
[S1] \quad \frac{\langle e, M \rangle \Downarrow n}{\langle m := e, M \rangle \mapsto \langle \text{skip}, M[m \mapsto n] \rangle} \\
[S2] \quad \frac{\langle S_1, M \rangle \mapsto \langle S'_1, M' \rangle}{\langle S_1; S_2, M \rangle \mapsto \langle S'_1; S_2, M' \rangle} \\
[S3] \quad \langle \text{skip}; s, M \rangle \mapsto \langle S, M \rangle \\
[S4] \quad \frac{\langle e, M \rangle \Downarrow n \quad n > 0}{\langle \text{if } e \text{ then } S_1 \text{ else } S_2, M \rangle \mapsto \langle S_1, M \rangle} \\
[S5] \quad \frac{\langle e, M \rangle \Downarrow n \quad n \leq 0}{\langle \text{if } e \text{ then } S_1 \text{ else } S_2, M \rangle \mapsto \langle S_2, M \rangle} \\
[S6] \quad \frac{\langle e, M \rangle \Downarrow n \quad n > 0}{\langle \text{while } e \text{ do } S, M \rangle \mapsto \langle s; \text{while } e \text{ do } S, M \rangle} \\
[S7] \quad \frac{\langle e, M \rangle \Downarrow n \quad n \leq 0}{\langle \text{while } e \text{ do } S, M \rangle \mapsto \langle \text{skip}, M \rangle}
\end{array}$$

Figure 2: Operational semantics for Aimp

- (A)  $m_2 := !m_1; \quad m_o := 1;$
- (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: Examples

In code segment (C), the if statement does not terminate if  $m_1$  is positive, so the availability of  $m_o$  depends on the value 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 value 2. Therefore, for the whole program, the availability of  $m_o$  depends on the value of  $m_1$ .

### 3.3 Noninterference properties

This section formalizes the noninterference properties (in particular, availability noninterference). 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 the computation using 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.

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  $M(m) = \text{none}$ , 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  $\langle S, M \rangle \mapsto \langle S_1, M_1 \rangle \mapsto \dots \mapsto \langle S_n, M_n \rangle$  such that  $T = [M, M_1, \dots, M_n]$ . 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  $\Gamma$ . Then  $m$  belongs to the low-confidentiality part of  $M$  if  $C(\Gamma(m)) \leq l_A$ , where  $C(\Gamma(m))$  denotes  $C(\ell)$  if  $\Gamma(m)$  is  $\text{int}_\ell$ . Similarly,  $m$  is a high-integrity reference if  $I(\Gamma(m)) \not\leq l_A$ , and a high-availability reference if  $A(\Gamma(m)) \not\leq l_A$ . 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  $v_1 = v_2$ ,  $v_1 = \text{none}$  or  $v_2 = \text{none}$ . With these settings, given two memories  $M_1$  and  $M_2$  with respect to  $\Gamma$ , we define three kinds of indistinguishability relations between  $M_1$  and  $M_2$  as follows:

**Definition 3.1** ( $\Gamma \vdash M_1 \approx_{C \leq l_A} M_2$ ). The low-confidentiality parts of  $M_1$  and  $M_2$  are indistinguishable, written  $\Gamma \vdash M_1 \approx_{C \leq l_A} M_2$ , if for any  $m \in \text{dom}(\Gamma)$ ,  $C(\Gamma(m)) \leq l_A$  implies  $M_1(m) \approx M_2(m)$ .

**Definition 3.2** ( $\Gamma \vdash M_1 \approx_{I \not\leq l_A} M_2$ ). The high-integrity parts of  $M_1$  and  $M_2$  are indistinguishable, written  $\Gamma \vdash M_1 \approx_{I \not\leq l_A} M_2$ , if for any  $m \in \text{dom}(\Gamma)$ ,  $I(\Gamma(m)) \not\leq l_A$  implies  $M_1(m) \approx M_2(m)$ .

**Definition 3.3** ( $\Gamma \vdash M_1 \approx_{A \not\leq l_A} M_2$ ). The high-availability parts of  $M_1$  and  $M_2$  are equally available, written  $\Gamma \vdash M_1 \approx_{A \not\leq l_A} M_2$ , if for any  $m \in \text{dom}(\Gamma)$ ,  $A(\Gamma(m)) \not\leq l_A$  implies that  $M_1(m) = \text{none}$  if and only if  $M_2(m) = \text{none}$ .

Based on the definitions of memory indistinguishability, we can define trace indistinguishability, which formalizes the notion of equivalent outputs. 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. Given two traces  $T_1$  and  $T_2$  of memories with respect to  $\Gamma$ , let  $\Gamma \vdash T_1 \approx_{C \leq l_A} T_2$  denote that the low-confidentiality parts of  $T_1$  and  $T_2$  are indistinguishable, and  $\Gamma \vdash T_1 \approx_{I \leq l_A} T_2$  denote that the high-integrity parts of  $T_1$  and  $T_2$  are indistinguishable. These two notions are defined as follows:

**Definition 3.4** ( $\Gamma \vdash T_1 \approx_{C \leq l_A} T_2$ ). Given two traces  $T_1$  and  $T_2$ ,  $\Gamma \vdash T_1 \approx_{C \leq l_A} T_2$  if there exist  $T'_1 = [M_1, \dots, M_n]$  and  $T'_2 = [M'_1, \dots, M'_m]$  such that  $T_1 \approx T'_1$ , and  $T_2 \approx T'_2$ , and  $\Gamma \vdash M_i \approx_{C \leq l_A} M'_i$  for any  $i$  in  $\{1, \dots, \min(m, n)\}$ .

**Definition 3.5** ( $\Gamma \vdash T_1 \approx_{I \leq l_A} T_2$ ). Given two traces  $T_1$  and  $T_2$ ,  $\Gamma \vdash T_1 \approx_{I \leq l_A} T_2$  if there exist  $T'_1 = [M_1, \dots, M_n]$  and  $T'_2 = [M'_1, \dots, M'_m]$  such that  $T_1 \approx T'_1$ , and  $T_2 \approx T'_2$ , and  $\Gamma \vdash M_i \approx_{I \leq l_A} M'_i$  for any  $i$  in  $\{1, \dots, \min(m, n)\}$ .

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  $\approx_{I \leq l_A}$  and  $\approx_{C \leq l_A}$  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 3.6** (Confidentiality noninterference). A program  $S$  has the *confidentiality noninterference* property w.r.t. a typing assignment  $\Gamma$ , written  $\Gamma \vdash \text{NI}_C(S)$ , 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  $\Gamma \vdash M_1 \approx_{C \leq l_A} M_2$  implies  $\Gamma \vdash T_1 \approx_{C \leq l_A} T_2$ .

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 [21, 1, 19].

**Definition 3.7** (Integrity noninterference). A program  $S$  has the *integrity noninterference* property w.r.t. a typing assignment  $\Gamma$ , written  $\Gamma \vdash \text{NI}_I(S)$ , 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  $\Gamma \vdash M_1 \approx_{I \leq l_A} M_2$  implies  $\Gamma \vdash T_1 \approx_{I \leq l_A} T_2$ .

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 represented by  $\forall m. (A(\Gamma(m)) \not\leq l_A \wedge m \notin \mathcal{R}) \Rightarrow M(m) \neq \text{none}$ , as in the following definition of availability noninterference:

**Definition 3.8** (Availability noninterference). A program  $S$  has the *availability noninterference* property w.r.t. a typing assignment  $\Gamma$  and a set of unassigned output references  $\mathcal{R}$ , written  $\Gamma; \mathcal{R} \vdash \text{NI}_A(S)$ , if for any two memories  $M_1, M_2$ , the following statements

- $\Gamma \vdash M_1 \approx_{I \leq l_A} M_2$
- For  $i \in \{1, 2\}$ ,  $\forall m \in \text{dom}(\Gamma). A(\Gamma(m)) \not\leq l_A \wedge m \notin \mathcal{R} \Rightarrow M_i(m) \neq \text{none}$
- $\langle S, M_i \rangle \mapsto^* \langle S'_i, M'_i \rangle$  for  $i \in \{1, 2\}$

imply that there exist  $\langle S''_i, M''_i \rangle$  for  $i \in \{1, 2\}$  such that  $\langle S'_i, M'_i \rangle \mapsto^* \langle S''_i, M''_i \rangle$  and  $\Gamma \vdash M''_1 \approx_{A \leq l_A} M''_2$ .

[INT]	$\Gamma; \mathcal{R} \vdash n : \text{int}_\ell$
[NONE]	$\Gamma; \mathcal{R} \vdash \text{none} : \text{int}_\ell$
[REF]	$\frac{\Gamma(m) = \text{int}_\ell}{\Gamma; \mathcal{R} \vdash m : \text{int}_\ell \text{ ref}}$
[DEREF]	$\frac{m \notin \mathcal{R} \quad \Gamma(m) = \text{int}_\ell}{\Gamma; \mathcal{R} \vdash !m : \text{int}_\ell}$
[ADD]	$\frac{\Gamma; \mathcal{R} \vdash e_1 : \text{int}_{\ell_1} \quad \Gamma; \mathcal{R} \vdash e_2 : \text{int}_{\ell_2}}{\Gamma; \mathcal{R} \vdash e_1 + e_2 : \text{int}_{\ell_1 \sqcup \ell_2}}$
[SKIP]	$\Gamma; \mathcal{R}; pc \vdash \text{skip} : \text{stmt}_{\mathcal{R}}$
[SEQ]	$\frac{\Gamma; \mathcal{R}; pc \vdash S_1 : \text{stmt}_{\mathcal{R}_1} \quad \Gamma; \mathcal{R}_1; pc \vdash S_2 : \text{stmt}_{\mathcal{R}_2}}{\Gamma; \mathcal{R}; pc \vdash S_1; S_2 : \text{stmt}_{\mathcal{R}_2}}$
[ASSIGN]	$\frac{\Gamma; \mathcal{R} \vdash m : \text{int}_\ell \text{ ref} \quad \Gamma; \mathcal{R} \vdash e : \text{int}_{\ell'} \quad C(pc) \sqcup C(\ell) \leq C(\ell) \quad I(\ell) \leq I(pc) \sqcap I(\ell') \quad A_\Gamma(\mathcal{R}) \leq A(\ell')}{\Gamma; \mathcal{R}; pc \vdash m := e : \text{stmt}_{\mathcal{R} - \{m\}}}$
[IF]	$\frac{\Gamma; \mathcal{R} \vdash e : \text{int}_\ell \quad A_\Gamma(\mathcal{R}) \leq A(\ell) \quad \Gamma; \mathcal{R}; pc \sqcup \ell \vdash S_i : \tau \quad i \in \{1, 2\}}{\Gamma; \mathcal{R}; pc \vdash \text{if } e \text{ then } S_1 \text{ else } S_2 : \tau}$
[WHILE]	$\frac{\Gamma \vdash e : \text{int}_\ell \quad \Gamma; \mathcal{R}; pc \sqcup \ell \vdash S : \text{stmt}_{\mathcal{R}} \quad A_\Gamma(\mathcal{R}) \leq I(\ell) \sqcap I(pc) \sqcap A(\ell)}{\Gamma; \mathcal{R}; pc \vdash \text{while } e \text{ do } S : \text{stmt}_{\mathcal{R}}}$
[SUB]	$\frac{\Gamma; \mathcal{R}; pc \vdash S : \tau \quad \Gamma; \mathcal{R}; pc \vdash \tau \leq \tau'}{\Gamma; \mathcal{R}; pc \vdash S : \tau'}$

Figure 4: Typing rules for Aimp

### 3.4 Type system

The type system of Aimp is designed to ensure that any well-typed Aimp program satisfies the noninterference properties defined in Section 3.3. For confidentiality and integrity, the type system performs a standard static information flow analysis [5, 23]. For availability, the type system tracks the set of unassigned output references and uses them to ensure that availability requirements are not violated.

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:  $\Gamma; \mathcal{R}; pc \vdash S : \text{stmt}_{\mathcal{R}'}$ , where  $\Gamma$  is the typing assignment, and  $pc$  is the *program counter* label [4] used to track security levels of the program counter. The typing judgment for expressions has the form  $\Gamma; \mathcal{R} \vdash e : \tau$

The typing rules are shown in Figure 3.4. Rules (INT) and (NONE) check constants. An integer  $n$  has type  $\text{int}_\ell$  where  $\ell$  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 `intℓ`.

Rule (REF) says that the type of a reference  $m$  is  $\tau \text{ ref}$  if  $\Gamma(m) = \tau$ . In `Aimp`, a memory maps references to values, and values always have integer types.

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

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

Rule (SEQ) checks sequential statements. The premise  $\Gamma; \mathcal{R}; pc \vdash S_1 : \text{stmt}_{\mathcal{R}_1}$  means that  $\mathcal{R}_1$  is 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  $I(\Gamma(m)) \leq I(pc) \sqcap C(\ell')$  to protect the integrity of  $m$ .

If the value of  $e$  is unavailable, the assignment  $m := e$  will get stuck. Therefore, rule (ASSIGN) has the premise  $A_\Gamma(\mathcal{R}) \leq A(\ell')$ , where  $A_\Gamma(\mathcal{R}) = \bigsqcup_{m \in \mathcal{R}} A(\Gamma(m))$ , to ensure the availability of  $e$  is as high as the availability of any unassigned output reference. For example, in the code segment (A) of Figure 3, the type system ensures that  $A(\Gamma(m_o)) \leq A(\Gamma(m_1))$ .

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

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  $\tau$  if both  $S_1$  and  $S_2$  have type  $\tau$ . As in rule (ASSIGN), the premise  $A_\Gamma(\mathcal{R}) \leq A(\ell)$  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, in which  $\mathcal{R} = \{m_o\}$ . Suppose  $A(\Gamma(m_o)) \not\leq l_A$ . In (B), the constraint  $A_\Gamma(\mathcal{R}) \leq I(\ell)$  of rule (WHILE) ensures  $I(\Gamma(m_1)) \not\leq l_A$ , so attackers cannot affect the value of  $m_1$ , and whether the `while` statement diverges. In (C), the constraint  $A_\Gamma(\mathcal{R}) \leq I(pc)$  guarantees  $I(pc) \not\leq l_A$ , and thus  $I(\Gamma(m_1)) \not\leq l_A$  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  $\tau$  is a subtype of  $\tau'$  with respect to the typing environment  $\Gamma; \mathcal{R}; pc$ . The type system of `Aimp` has one subtyping rule:

$$[\text{ST}] \quad \frac{\mathcal{R}' \subseteq \mathcal{R}'' \subseteq \mathcal{R} \quad \forall m, m \in \mathcal{R}'' - \mathcal{R}' \Rightarrow A(\Gamma(m)) \leq I(pc)}{\Gamma; \mathcal{R}; pc \vdash \text{stmt}_{\mathcal{R}'} \leq \text{stmt}_{\mathcal{R}''}}$$

Suppose  $\Gamma; \mathcal{R}; pc \vdash \text{stmt}_{\mathcal{R}'} \leq \text{stmt}_{\mathcal{R}''}$  and  $\Gamma; \mathcal{R}; pc \vdash S : \text{stmt}_{\mathcal{R}'}$ . Then  $\Gamma; \mathcal{R}; pc \vdash S : \text{stmt}_{\mathcal{R}''}$  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, that the statement  $S$  can be treated as having type  $\text{stmt}_{\mathcal{R}''}$  is because there exists 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. By rule (ASSIGN),  $\Gamma; \{m_o\}; pc \vdash m_o := 0 : \text{stmt}_\emptyset$ . For the else branch of the if statement, we have  $\Gamma; \{m_o\}; pc \vdash \text{skip} : \text{stmt}_{\{m_o\}}$ . By rule (IF),  $\Gamma; \{m_o\}; pc \vdash m_o := 0 : \text{stmt}_{\{m_o\}}$  needs to hold, which requires  $\Gamma; \{m_o\}; pc \vdash \text{stmt}_\emptyset \leq \text{stmt}_{\{m_o\}}$ . 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 subject reduction property. Moreover, we can prove that any well-typed program has confidentiality, integrity and availability noninterference properties. The proofs of the following two theorems can be found in our previous technical report [30].

**Theorem 3.1** (Subject reduction). Suppose  $\Gamma; \mathcal{R}; pc \vdash S : \tau$ , and  $\text{dom}(\Gamma) = \text{dom}(M)$ . If  $\langle S, M \rangle \mapsto \langle S', M' \rangle$ , then there exists  $\mathcal{R}'$  such that  $\Gamma; \mathcal{R}'; pc \vdash S' : \tau$ , and  $\mathcal{R}' \subseteq \mathcal{R}$ , and for any  $m \in \mathcal{R} - \mathcal{R}'$ ,  $M'(m) \neq \text{none}$ .

**Theorem 3.2** (Noninterference). If  $\Gamma; \mathcal{R}; pc \vdash S : \tau$ , then  $\Gamma \vdash \text{NI}_C(S)$ ,  $\Gamma \vdash \text{NI}_I(S)$  and  $\Gamma; \mathcal{R} \vdash \text{NI}_A(S)$ .

## 4 Secure distributed computation

The programming language DSR is a calculus for describing secure distributed programs. This section 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 Section 5.

### 4.1 Reactors

A distributed system is a set of networked host machines. Each host is a state machine that acts upon incoming network messages, changing its local state and/or sending out messages to other hosts. The local state of a host includes a memory mapping location names (references) to values. In the DSR language, the reactions of a host to incoming messages are specified by *reactors* whose (simplified) syntax is:

$$c\{pc, loc, \bar{z}:\bar{\tau}_z, \lambda\bar{y}:\bar{\tau}.s\}$$

where  $c$  is a unique name;  $pc$  is both the initial program counter label in the reactor (and an upper bound to the labels of any side effects generated by the reactor);  $loc$  indicates where the reactor is located (which may be a host, a set of hosts, or a more complicated replication scheme);  $s$  is the reactor body, a statement to be executed when the reactor is invoked; both  $\bar{y}:\bar{\tau}$  and  $\bar{z}:\bar{\tau}_z$  are a list of variable declarations:  $y_1:\tau_1, \dots, y_n:\tau_n$  and  $z_1:\tau_{z1}, \dots, z_k:\tau_{zk}$ , where each  $y_i$  or  $z_i$  is a free variable of  $s$ . An empty list is denoted by  $\epsilon$  and may be omitted from a reactor declaration. When reactor  $c$  is invoked on host  $h$  by a network message  $\mu$ , a thread is created on  $h$  to execute the statement  $s$  with variables  $\bar{y}$  bound to the arguments embedded in  $\mu$ , and variables  $\bar{z}$  bound to the values in an invocation context.

Each invocation of reactor  $c$  on host  $h$  is associated with an unique invocation context, which is called a *reactor closure* and has the form  $\langle c, \eta, \ell, \mathcal{A}, \bar{a} \rangle$ , where  $\eta$  is an integer *context identifier*,  $\ell$  is an *access control label*, a runtime-enforced lower bound to the program counter label of the invoker,  $\mathcal{A}$  is a record that maps variables  $\bar{z}$  to values, and  $\bar{a}$  is a list of additional attributes discussed later. Because  $\eta$  is unique for a given  $c$ , the pair  $\langle c, \eta \rangle$  can be used to uniquely identify a closure, and is called a *context identifier*. For simplicity, we use the term “closure  $\langle c, \eta \rangle$ ” to denote the closure identified by  $\langle c, \eta \rangle$ . A message invoking reactor  $c$  has the simplified form  $[\text{exec } \langle c, \eta \rangle :: pc, \bar{v}]$ , where  $pc$  is the program counter label at the point of invocation, and  $\bar{v}$  is a list of arguments to which the variables  $\bar{y}$  of reactor  $c$  are bound. All control transfers are done by explicit `exec` statements; as in other process calculi (e.g., [15, 7]), reactors do not implicitly return to their invokers.

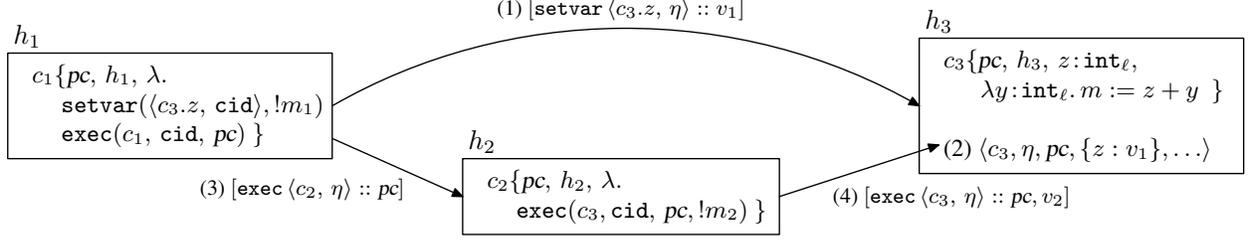


Figure 5: A distributed program

In general, a network message  $\mu$  has the form  $[\alpha :: \beta]$ , where  $\alpha$  is the *message head* identifying the message and indicating its purpose, and  $\beta$  is the *message body* containing specific parameters. Both  $\alpha$  and  $\beta$  are lists of components. Two messages with the same message head if they are logically the same message, sent by different replicas. We say that a message  $[\alpha :: \beta]$  is sent to reactor  $c$  if  $\alpha$  indicates that the purpose of the message is to invoke  $c$  or change the state of some closure  $\langle c, \eta \rangle$ .

A distributed program in DSR is simply a set of reactor declarations. Figure 5 shows a DSR program that simulates the Aimp statement  $m := !m_1 + !m_2$ . 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. Accordingly, reactors  $c_1$  on host  $h_1$  reads the value of  $m_1$  and sends it 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 5, reactor  $c_1$  is invoked with a context identifier  $\eta$ . It executes the statement  $\text{setvar}(\langle c_3.z_3, \text{cid} \rangle, !m_1, \_)$ , where  $\text{cid}$  is a variable bound to the current context identifier  $\eta$ . The statement sends the message  $[\text{setvar} \langle c_3.z_3, \eta \rangle :: v_1]$  to  $h_3$  where  $v_1$  is the value of  $m_1$ . Upon receiving the message,  $h_3$  updates the variable record of the closure  $\langle c_3, \eta \rangle$ , mapping  $z$  to  $v$ . In parallel,  $c_1$  invokes  $c_2$  by running the statement  $\text{exec}(c_2, \text{cid}, pc)$ . Then reactor  $c_2$  executes the statement  $\text{exec}(c_3, \text{cid}, pc, !m_2)$  to send the invocation message  $[\text{exec} \langle c_3, \eta \rangle :: pc, v_2]$  to  $h_3$ , where  $v_2$  is the value of  $m_2$ . Once invoked, reactor  $c_3$  executes the statement  $m := z + y$  with  $y$  bound to the argument  $v_2$  of the invocation request, and  $z$  bound to  $v_1$  according to the closure  $\langle c_3, \eta \rangle$ .

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 then make  $c_2$  compute  $v_1 + !m_2$  and send the result to  $c_3$ . This way, there is no need for the variable binding mechanism based on closures. However, the value of  $m_1$  needs to be sent to  $h_2$ , imposing an additional security requirement that  $h_2$  respects the confidentiality of  $m_1$ . As we can see,  $\text{setvar}$  operations and reactor closures make it possible to separate data flow and control flow, providing more flexibility for constructing secure distributed computation.

## 4.2 Control transfer

Suppose a message  $\mu = [\text{exec} \langle c, \eta \rangle :: pc_\mu, \bar{v}]$  is sent to invoke a reactor  $c\{pc, \dots\}$  on host  $h$ . The condition  $pc_\mu \sqsubseteq pc$  is important because any side effect of  $c$  is caused by  $\mu$ . This constraint prevents low-integrity reactors, possibly controlled by attackers, from invoking high-integrity reactors and causing effects they could not have on their own. It also prevents covert information channels via implicit flow [5]. However, taken naively, this constraint leads to the infamous “label creep” problem: the integrity of control flow can only be weakened and may eventually be unable to invoke any reactor.

In fact, a low-integrity message *can* be used to invoke a high-integrity closure  $\langle c, \eta \rangle$ , if the closure is a *linear entry*, which means that it is the only closure that can be invoked by a low-integrity message, and furthermore that there is no reactor running at a high-integrity level. Low-integrity reactors cannot harm the integrity of computation by invoking a high-integrity linear entry, because that entry is the only way to continue high-integrity computation.

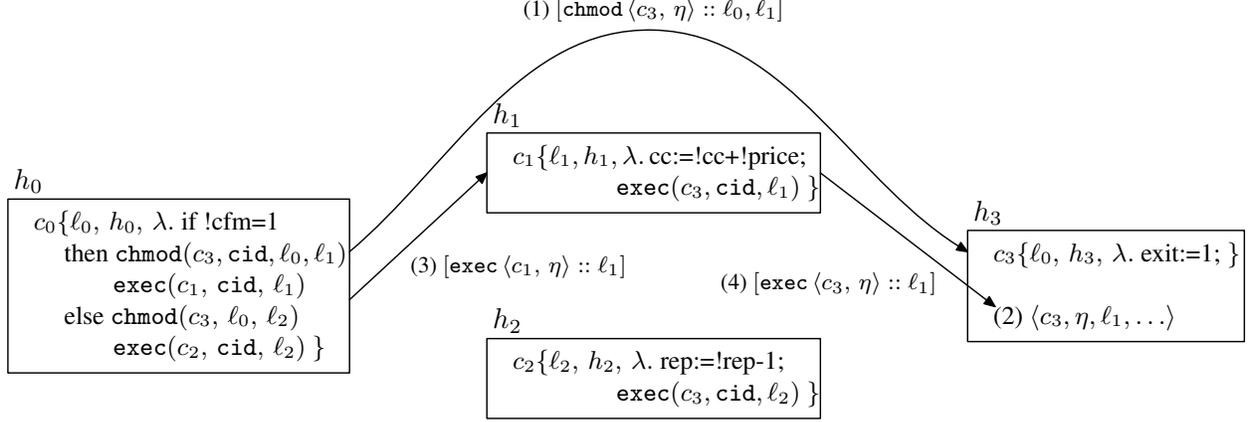


Figure 6: Control transfer example

A simple protocol allows a low-integrity message to invoke a high-integrity linear entry: when a high-integrity reactor  $c_0$  is ready to invoke a low-integrity reactor, it may choose to instruct a single high-integrity closure  $\langle c, \eta \rangle$  to accept low-integrity invocation requests. More concretely,  $c_0$  sends a message  $[\text{chmod} \langle c, \eta \rangle :: pc, \ell']$  to host  $h'$  where  $c$  is located. The  $\text{chmod}$  message means that  $\langle c, \eta \rangle$  is a linear entry that may be invoked by a message  $\mu$  satisfying  $pc_\mu \sqsubseteq \ell$ . Suppose the closure  $\langle c, \eta \rangle$  on  $h$  has the form  $\langle c, \eta, \ell, \mathcal{A}, \bar{a} \rangle$ . Host  $h$  first checks the constraint  $pc \sqsubseteq \ell$  to ensure the  $\text{chmod}$  message has sufficient integrity. If the constraint is satisfied,  $h$  sets  $\ell'$  as the access control label of the closure  $\langle c, \eta \rangle$ . When the closure  $\langle c, \eta \rangle$  is first created, its access control label is set as  $pc$  (the program counter label of  $c$ ) to provide normal protection.

Figure 6 illustrates a run of the control transfer protocol. The distributed program in Figure 6 performs the same computation as lines 7–9 in Figure 1. Assume locations  $\text{cfm}$ ,  $\text{cc}$ ,  $\text{rep}$  and  $\text{exit}$  are located at hosts  $h_0$ ,  $h_1$ ,  $h_2$  and  $h_3$ , respectively. The  $\text{if}$  statement on line 7 needs to access  $\text{cfm}$ , and thus it is executed by reactor  $c_0$  on host  $h_0$ . Suppose  $c_0$  is invoked with a context identifier  $\eta$ , and the value of  $\text{cfm}$  is 1. Then reactor  $c_0$  invokes reactor  $c_1$  that updates  $\text{cc}$  and invokes  $\ell_0$ . Note that  $[\text{exec} \langle c_3, \eta \rangle :: \ell_1]$  is a low-integrity message with respect to  $c_3$ . And  $\langle c_3, \eta \rangle$  is a linear entry when  $c_0$  invokes  $c_1$ . To notify  $h_3$  that  $\langle c_3, \eta \rangle$  is a linear entry,  $c_0$  runs the statement  $\text{chmod}(c_3, \text{cid}, \ell_0, \ell_1)$  to send  $h_3$  a  $\text{chmod}$  message, which requests  $h_3$  to change the access control label of  $\langle c_3, \eta \rangle$  to  $\ell_1$ . The request is accepted because  $c_0$ 's program counter label  $\ell_0$  satisfies the access check. After executing the  $\text{chmod}$  statement, execution enters a low-integrity phase, in which  $c_0$  is not allowed to produce any effect with an integrity label higher than  $I(\ell_1)$ . Then  $c_0$  invokes  $c_1$ , which is able to invoke  $\langle c_3, \eta \rangle$  because  $\langle c_3, \eta \rangle$  has a low-integrity access control label.

### 4.3 Replication and message synthesis

Replicating data is an effective way to achieve fault tolerance and ensure integrity and availability. In general, computation has to be replicated too, because unreplicated computation is vulnerable to failure and attack. In DSR, a reactor  $c$  can be replicated on a set of hosts  $H$ . The messages for  $c$  are sent to all the hosts in  $H$ . The memory references accessed by  $c$  are also replicated on  $H$ , so that each replicas of  $c$  only accesses its local memory and is executed independently of other replicas. The replicas of  $c$  are supposed to produce the same messages, while replicas on bad hosts may send corrupted messages or simply not send anything. The receiver host  $h$  of a message sent by  $c$  may receive the replicas of the message from different hosts, including corrupted messages from bad hosts. Host  $h$  needs to identify the correct message from those message replicas. This process is called *message synthesis*, using a message synthesizer  $\pi$  that takes in a set of messages sent by reactor replicas running on a set of hosts  $H$ , and returns one message with sufficient

integrity.

The integrity of a message depends on the integrity of its senders. Suppose host  $h$  receives a message  $\mu$  with integrity label  $I(\mu)$  from the host set  $H$ . Then the constraint  $I(\mu) \leq I_{\sqcup}(H) = \bigsqcup_{h \in H} I(h)$  guarantees the integrity of  $\mu$ . If  $I(\mu) \not\leq l_A$ , then an attacker needs to compromise all the hosts in  $H$  in order to fabricate  $\mu$ , which means that the attacker has the power to affect integrity level  $I_{\sqcup}(H)$ , that is,  $I_{\sqcup}(H) \leq l_A$ , contradicting  $I(\mu) \leq I_{\sqcup}(H)$ . Therefore,  $\mu$  has sufficient integrity if  $I(\mu) \leq I_{\sqcup}(H)$ . By enforcing this constraint, a message synthesizer is guaranteed to deliver messages with sufficient integrity. However, it is not sufficient to guarantee availability, because the attacker can deliberately send inconsistent messages to prevent a synthesizer from producing a message.

Intuitively, on receiving enough inconsistent messages, a synthesizer may be able to determine that the attacker already controls enough hosts to fabricate a message with sufficient integrity. In this case, the synthesizer can stop waiting and return a default message since the contents of the message are not trustworthy, and to continue to wait would turn an integrity failure into an availability failure. Abstractly, each synthesizer  $\pi$  is associated with a condition  $\text{qualified}_{\pi}$ , and if  $\text{qualified}_{\pi}(H)$  is true, then receiving inconsistent messages from  $H$  implies that the integrity level of the attacker is higher than or equal to the integrity of the expected message. Therefore,  $\pi$  is able to produce a message if any qualified set  $H'$  is available, that is, all the hosts in  $H'$  are available. The availability label of  $H'$  can be computed by the formula  $A(H') = A_{\cap}(H') = \prod_{h \in H'} A(h)$ , because  $A(H')$  is at most as strong as the availability label of any host in  $H'$ . Suppose a reactor replicated on  $H$  sends a message to be synthesized by  $\pi$ . Then the availability label of the message is computed using the following formula:

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

For example, consider a *label threshold* synthesizer  $\text{LT}[I]$ , which produces message  $\mu$  if  $\mu$  is sent by a host set  $H$  such that  $I \leq I_{\sqcup}(H)$ . Suppose  $H'$  is a qualified host set for  $\text{LT}[I]$ . Since a host is either good (high-integrity) or bad (low-integrity),  $H'$  can be partitioned into two disjoint sets: good hosts  $H_1$  and bad hosts  $H_2$ . The messages sent by good hosts are always the same. If  $I \leq I_{\sqcup}(H_1)$ , then  $\text{LT}[I]$  would deliver the message from  $H_1$ . Otherwise, by the definition of a qualified set,  $I \leq l_A$ , which follows  $I \leq I_{\sqcup}(H_2)$  since  $I_{\sqcup}(H_2) \leq l_A$ . In other words,  $H'$  is qualified if  $I \not\leq I_{\sqcup}(H_1)$  implies  $I \leq I_{\sqcup}(H_2)$ . Based on this observation, we have the following  $\text{qualified}_{\text{LT}[I]}$  condition:  $H'$  is *qualified* if it cannot be partitioned into two disjoint sets  $H_1$  and  $H_2$  such that  $I \not\leq I_{\sqcup}(H_1)$  and  $I \not\leq I_{\sqcup}(H_2)$ . And the algorithm of  $\text{LT}[I]$  can be described by the following pseudo-code, which synthesizes a set of messages  $\mu_1, \dots, \mu_n$  from  $H$ :

```

LT[I](H,  $\mu_1, \dots, \mu_n$ ) {
  if  $\exists H' \subseteq H. I \leq I_{\sqcup}(H') \wedge \forall h_j \in H'. \mu_j = \mu$ 
  then return  $\mu$ 
  if  $\text{qualified}_{\text{LT}[I]}(H)$  then return  $\mu_{\text{default}}$ 
}

```

#### 4.4 Using quorum systems

The label threshold synthesizer is based on the assumption that all the replicas on good hosts generate the same outputs. The assumption requires that good hosts have the same local state. In particular, if a message  $\mu$  has its contents depending on the value of some memory reference  $m$ , then the replicas of  $m$  on good hosts must have the same value, or the replicas of  $\mu$  cannot be synthesized using  $\text{LT}[I]$ . The consistency (equality) between the replicas of  $m$  on good hosts essentially requires to synchronize the replicas of any reactor that updates  $m$ . However, this strong synchronization requirement makes it difficult to guarantee availability. Therefore, to achieve high availability, we need other replication schemes and message synthesis algorithms.

Quorum systems are a well-know approach to implementing highly available shared memory [11, 14]. A quorum system  $\mathcal{Q}$  is a collection of sets (quorums) of hosts, having the form  $\langle H, W_1, \dots, W_n \rangle$ , where  $H$  is all the hosts in  $\mathcal{Q}$ , and quorums  $W_1, \dots, W_n$  are subsets of  $H$ . Suppose a memory reference  $m$  is replicated on a quorum system. Then an update to  $m$  is considered *stable* if it is completed on a quorum of hosts. In DSR as in some other quorum systems [14], timestamps are used to distinguish different version of the same replicated memory location. A read operation can get the most recent update by consulting with a set of hosts intersecting each 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.

Quorum system protocols can be incorporated into the reactor model. First, the execution of reactors keeps tracks of a timestamp that is incremented with every execution step. If  $m$  is assigned value  $v$  at timestamp  $t$ , the versioned value  $v@t$  is recorded in the local memory as a version of  $m$ . The asynchronous execution model for reactors makes it possible that a reactor on host  $h$  updates  $m$  at time  $t$  while another reactor on  $h$  still needs to access  $m$  at an earlier time. To deal with this write-read conflict, the local memory  $M$  on a host remembers the old versions of references, mapping a memory reference  $m$  to a set of versioned values  $v@t$ . A dereference  $!m$  evaluated at time  $t$  results in the most recent version of  $m$  by the time  $t$ . The type system of DSR prevents a versioned value from being used in any computation since a versioned value may be outdated.

If reactor  $c$  on host  $h$  needs to use the value of a reference  $m$  replicated on a quorum system  $\mathcal{Q}$ , some reactor  $c'$  replicated on  $\mathcal{Q}$  is invoked to send to  $h$  the message  $[\text{setvar } \langle c.z, \eta \rangle :: v_i@t_i]$  from each host  $h_i$  in  $\mathcal{Q}$ , where  $v_i@t_i$  is the most recent version of  $m$  on host  $h_i$  by the timestamp  $t'$  of  $c'$ . Host  $h$  uses a *quorum read synthesizer* (written as  $\text{QR}[\mathcal{Q}, I]$ , where  $I$  is the integrity label of  $m$ ) to find out the most recent value of  $m$  before  $t'$  and bind  $z$  to that value. It is possible that some high-integrity hosts replicating  $m$  missed the update to it needed by  $c$ . However, all the hosts in at least one quorum in  $\mathcal{Q}$  hold the needed version of  $m$ . Therefore, if  $\text{QR}[\mathcal{Q}, I]$  receives sufficient *setvar* messages from every quorum of  $\mathcal{Q}$ , it can identify the needed value with integrity  $I$ . Based on this insight, we have the following qualified condition for  $\text{QR}[\mathcal{Q}, I]$ : a host set  $H'$  is qualified with respect to  $\text{QR}[\mathcal{Q}, I]$  if  $\text{qualified}_{\text{LT}[I]}(W \cap H')$  holds for any quorum  $W$  in  $\mathcal{Q}$ . Intuitively, it means that the intersection between  $H'$  and  $W$  is a qualified set for  $\text{LT}[I]$ , and thus the messages from  $H' \cap W$  are sufficient to determine the value held by  $W$ , if  $W$  is the quorum holding the most recent version of  $m$ . In fact, if the quorum  $W$  holds the most recent value  $v@t$ , then any good host in  $W \cap H'$  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.

Suppose  $\mu_i = [\text{setvar } \langle c.z, \eta \rangle :: v_i@t_i]$  ( $1 \leq i \leq n$ ) are sent from  $H'$ . Let  $H' \vdash v@t : I$  denote that there exists a subset  $H''$  of  $H'$  such that  $I \leq I_{\sqcup}(H'')$  and for any host  $h_j$  in  $H''$ ,  $v_j@t_j = v@t$ . Intuitively, the notation means that  $v@t$  is a version of  $m$  with sufficient integrity. Then the following  $\text{QR}[\mathcal{Q}, I]$  algorithm is able to return the appropriate version of  $m$  with sufficient integrity:

```

QR[ $\mathcal{Q}, I$ ]( $H', \mu_1, \dots, \mu_n$ ) {
  if qualifiedQR[ $\mathcal{Q}, I$ ]( $H'$ ) then
    if  $H' \vdash v@t : I$  and  $\forall t_i. t < t_i \Rightarrow H' \not\vdash v_i@t_i : I$ 
      then return  $\{ \langle c.\eta, z \rangle, \text{setvar} :: v \}$ 
      else return  $\{ \langle c.\eta, z \rangle, \text{setvar} :: v_{\text{default}} \}$ 
}

```

The quorum read synthesizer assumes that an update to  $m$  is stable by the time  $m$  is read by another reactor. Suppose  $m$  is replicated on  $\mathcal{Q}$  and updated by reactor  $c$ . Then the reactor invoked by  $c$  is required to wait for the invocation requests from a quorum of  $\mathcal{Q}$  to ensure that the execution of  $c$ , including the update to  $m$ , is completed on a quorum. This way, the update is guaranteed to be stable by the time  $m$  is accessed by another reactor.

An available quorum ensures that a write operation to terminate, while an available qualified set for the quorum read synthesizer ensures a read operation to terminate. Therefore, the availability guarantees provided by a quorum system for the read and write operations are as follows:

$$A_{\text{write}}(\mathcal{Q}) = \bigsqcup_{W \in \mathcal{Q}} A_{\cap}(W)$$

$$A_{\text{read}, I}(\mathcal{Q}) = \bigsqcup_{H | \text{qualified}_{\text{QR}[\mathcal{Q}, I]}(H)} A_{\cap}(H)$$

## 4.5 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  $\ell$  such that  $\ell \sqsubseteq pc$  (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 reactor. Formally, a multi-level timestamp is a tuple  $\langle \overline{pc} : \overline{n}, \delta \rangle$ : the global part  $\overline{pc} : \overline{n}$  is a list of pairs  $\langle pc_1 : n_1, \dots, pc_k : n_k \rangle$ , where  $pc_1, \dots, pc_k$  are program counter labels satisfying the constraint  $pc_1 \sqsubseteq \dots \sqsubseteq pc_k$ , and  $n_1, \dots, n_k$  are integers. Intuitively, the component  $pc_i : n_i$  means that the number of reactors invoked at the level  $pc_i$  is  $n_i$ . The local part  $\delta$  is less significant than the global part in timestamp comparison, and its concrete form will be discussed later.

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  $\delta_0$ . Suppose  $t = \langle pc_1 : n_1, \dots, pc_k : n_k; \delta \rangle$ , and  $pc_i \sqsubseteq pc$  and  $pc_{i+1} \not\sqsubseteq pc$ . Then  $pc_{i+1} : n_{i+1}, \dots, pc_k : n_k$  are low-integrity components to be discarded, and incrementing  $t$  at level  $pc$  is carried out by the following formula:

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

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

## 4.6 Example

Like Figure 6, the distributed program in Figure 7 performs the same computation as lines 7–9 in Figure 1, but with the assumption that reference `price` 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. A new reactor `readprice` is used to read the value of `price` and send the value to  $c_1$  so that  $c_1$  can compute `!cc+!price`. The reactor `readprice` is replicated on the same quorum system as `price` so that each replica of `readprice` 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}, I]` (specified in the declaration of  $c_1$ , and  $I = I(\ell_1)$ ) to synthesize the `setvar` messages sent by replicas of `readprice`. If `QR[\mathcal{Q}, I]` produces a message `[setvar \langle c_1.amt, \eta \rangle :: v]`, then the value  $v$  is recorded in the closure  $\langle c_1, \eta \rangle$  as the value of `amt`.

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 timestamp  $t_1$  of the `chmod` message sent to  $h_3$  has the form  $\langle \ell_0 : 1; \delta_1 \rangle$ . On receiving the `chmod` message, host  $h_3$  increments  $t_1$  and stores the timestamp  $t_2 = \text{inc}(t_1, \ell_0) = \langle \ell_0 : 2; \delta_0 \rangle$  in the closure  $\langle c_3, \eta \rangle$ . When  $\langle c_3, \eta \rangle$  is invoked by a low-integrity message,  $t_2$  would be used

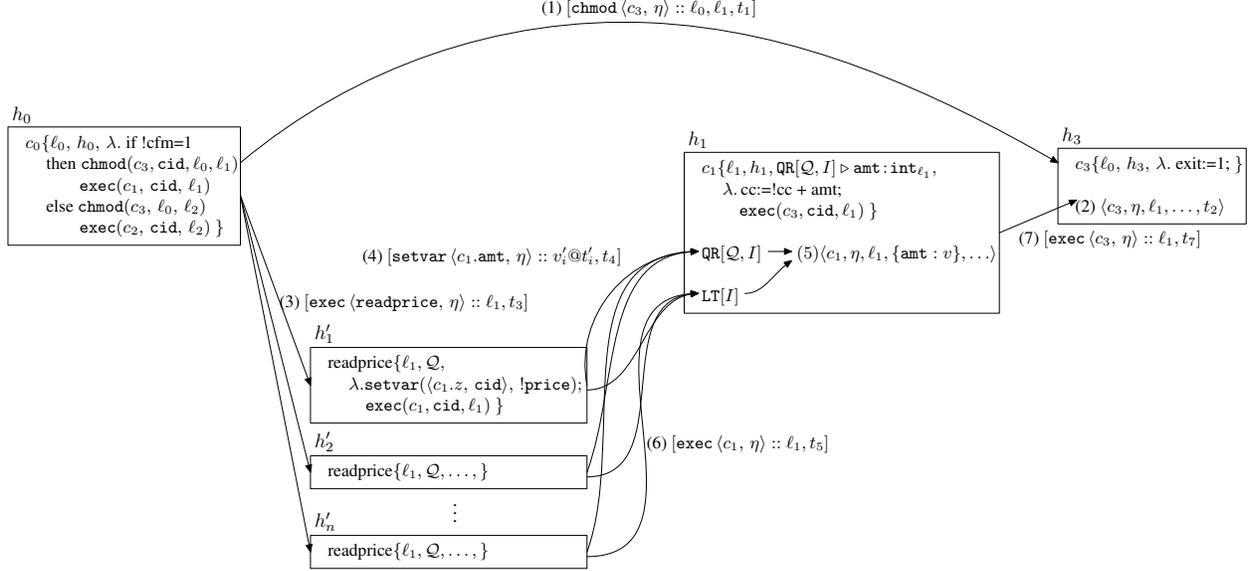


Figure 7: A running example

as the initial timestamp of the thread of  $\langle c_3, \eta \rangle$  instead of the timestamp obtained by incrementing  $t_7$ , the timestamp of the invocation message, because  $t_7$  is not sufficiently trustworthy. In Figure 7, replicas of  $\text{readprice}$  are invoked by an  $\text{exec}$  message carrying timestamp  $t_3 = \langle \ell_0 : 1; \delta_3 \rangle$ . Since the program counter label of  $\text{readprice}$  is  $\ell_1$ , the initial timestamp for  $\langle \text{readprice}, \eta \rangle$  is  $\text{inc}(t_3, \ell_1) = \langle \ell_0 : 1, \ell_1 : 1; \delta_0 \rangle$ . Similarly, the initial timestamp of  $\langle c_1, \eta \rangle$  is  $\text{inc}(t_5, \ell_1) = \langle \ell_0 : 1, \ell_1 : 2; \delta_0 \rangle$ . More interestingly, when the linear entry  $\langle c_3, \eta \rangle$  is invoked, the timestamp of the invocation request is ignored. And the initial timestamp of  $\langle c_3, \eta \rangle$  is  $t_2$ , which equals to  $\text{inc}(t_7, \ell_0)$  if  $t_7$  is correct.

## 5 The DSR language

This section formally describes the DSR language, making the security mechanisms already introduced more precise. The key difference between DSR and prior distributed calculi [15, 7] is that DSR provides explicit language constructs for replication and run-time security labels so that these mechanisms can be statically analyzed by a type system.

### 5.1 Syntax

The syntax of the DSR language is shown in Figure 5.1. A value  $v$  may be a variable  $x$ , an integer  $n$ , a closure identifier  $\eta$ , a memory reference  $m$ , the unavailable value  $\text{none}$ , a reactor  $c[\bar{v}]$ , a remote variable  $\langle c[\bar{v}].z, v \rangle$ , a versioned value  $v@t$ , or a label  $\ell$ . Expressions and statements are standard except for the three reactor operations  $\text{exec}$ ,  $\text{chmod}$  and  $\text{setvar}$ :

- $\text{exec}(c, \eta, pc, \mathcal{Q}, \bar{e})$  invokes a reactor  $\langle c, \eta \rangle$ . Arguments  $pc$  and  $\mathcal{Q}$  are the program counter label and the quorum system of the caller; the values of  $\bar{e}$  are arguments for the reactor.
- $\text{chmod}(c, \eta, pc, \mathcal{Q}, \ell)$  changes the access-control label of reactor instance  $\langle c, \eta \rangle$  to  $\ell$ .
- $\text{setvar}(v, e)$  initializes a remote variable  $v$  with the value of  $e$ . The value is stored in the closure of  $v$ .

Host	$h$	$\in$	$\mathcal{H}$
Base labels	$l$	$\in$	$\mathcal{L}$
Labels	$\ell, pc$	$::=$	$\{C = l_1, I = l_2, A = l_3\} \mid x$
Timestamps	$t$	$::=$	$\langle \overline{pc} : \overline{n}; \overline{n} \rangle$
Values	$v$	$::=$	$x \mid n \mid \eta \mid m \mid \mathbf{none} \mid c[\overline{v}] \mid \langle c[\overline{v}].z, v \rangle \mid v@t \mid \ell$
Expressions	$e$	$::=$	$v \mid !e \mid e_1 + e_2$
Statements	$s$	$::=$	$\mathbf{skip} \mid v := e \mid s_1; s_2 \mid \mathbf{if} \ e \ \mathbf{then} \ s_1 \ \mathbf{else} \ s_2$ $\mid \mathbf{exec}(v_1, v_2, pc, \mathcal{Q}, \overline{e}) \mid \mathbf{chmod}(v_1, v_2, pc, \mathcal{Q}, \ell) \mid \mathbf{setvar}(v, e)$
Reactor decls	$r$	$::=$	$c[\overline{x}:\overline{\sigma}]\{pc, \mathcal{Q}, \overline{\pi} \triangleright \overline{z}:\overline{\tau}, \lambda \overline{y}:\overline{\tau}.s\}$
Synthesizers	$\pi$	$::=$	$\mathbf{QR}[\mathcal{Q}, I] \mid \mathbf{LT}[I]$
Base types	$\beta$	$::=$	$\mathbf{int} \mid \mathbf{label} \mid \tau \ \mathbf{ref} \mid \tau \ \mathbf{var}$ $\mid \mathbf{reactor}[\overline{x}:\overline{\sigma}]\{pc, \overline{\pi} \triangleright \overline{z}:\overline{\tau}_1, \overline{\tau}_2\}$ $\mid \mathbf{reactor}[\overline{x}:\overline{\sigma}]\{pc, \overline{\tau}_2\}$
Security types	$\sigma$	$::=$	$\beta_\ell$
Types	$\tau$	$::=$	$\sigma \mid \sigma@Q \mid \mathbf{stmt}_{pc}$
Host sets	$H, W$	$::=$	$\{h_1, \dots, h_n\}$
Quorum systems	$\mathcal{Q}$	$::=$	$\langle H, \overline{W} \rangle \mid \&v \mid \#v$
Programs	$P$	$::=$	$\{r_1, \dots, r_n\}$

Figure 8: Syntax of the DSR language

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

$$c[\overline{x}:\overline{\sigma}]\{pc, \mathcal{Q}, \overline{\pi} \triangleright \overline{z}:\overline{\tau}_1, \lambda \overline{y}:\overline{\tau}_2. s\}$$

where  $\overline{x}:\overline{\sigma}$  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  $s$ . Variables  $\overline{z}$  are initialized by messages passing through synthesizers  $\overline{\pi}$ , which can be either  $\mathbf{QR}[\mathcal{Q}, I]$  or  $\mathbf{LT}[I]$ .

A base type  $\beta$  can be  $\mathbf{int}$  (integer),  $\mathbf{label}$  (security label),  $\tau \ \mathbf{ref}$  (reference of type  $\tau$ ),  $\tau \ \mathbf{var}$  (remote variable of type  $\tau$ ) and reactor type  $\mathbf{reactor}[\overline{x}:\overline{\sigma}]\{pc, \overline{\pi} \triangleright \overline{z}:\overline{\tau}_1, \overline{\tau}_2\}$  whose components are interpreted the same way as in a reactor declaration. A reactor type may also have a simplified form  $\mathbf{reactor}[\overline{x}:\overline{\sigma}]\{pc, \overline{\tau}_2\}$ , which contains sufficient typing information for checking the invocation, while providing polymorphism over the arguments  $\overline{z}$ .

A security type  $\sigma$  has the form  $\beta_\ell$ , a base type annotated with security label  $\ell$ . Like security policies, replication schemes are also specified as type annotations. A located type  $\sigma@Q$  indicates that data with this type is replicated on the quorum system  $Q$ . Moreover, if  $v$  is a reference replicated on  $Q$ ,  $\&v$  represents  $Q$ , and  $\#v$  represents  $\langle H, \emptyset \rangle$ , where  $H = |Q|$  is the set of hosts in  $Q$ . The type of a statement  $s$  has the form  $\mathbf{stmt}_{pc}$ , which means that after  $s$  terminates, the program counter label is  $pc$ .

## 5.2 Operational semantics

A system configuration is a tuple  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle$  where  $\Theta$  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. A thread is represented by 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;  $\langle c[\overline{v}], \eta \rangle$  is the reactor instance of this thread. 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 sent 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 identified by  $\langle c[\overline{v}], \eta \rangle$  on  $h$ .

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

- $\mathcal{M}[h, m, t]$ : the value of  $m$  on host  $h$  at time  $t$ . If  $\mathcal{M}[h, m, t] = v$ , then  $v@t \in \mathcal{M}[h][m]$ .

$$\begin{array}{c}
[E1] \frac{M(m) = v}{\langle !m, M \rangle \Downarrow v} \quad [E2] \frac{\langle e_i, M \rangle \Downarrow n_i \quad i \in \{1, 2\} \quad n = n_1 + n_2}{\langle e_1 + e_2, M \rangle \Downarrow n} \quad [E3] \quad \langle v, M \rangle \Downarrow v \\
[S1] \frac{\langle e, M \rangle \Downarrow n}{\langle m := e, M, \Omega, t \rangle \mapsto \langle \text{skip}, M[m \mapsto n @ t], \Omega, t + 1 \rangle} \quad [S2] \frac{\langle s_1, M, \Omega, t \rangle \mapsto \langle s'_1, M', \Omega', t' \rangle}{\langle s_1; s_2, M, \Omega, t \rangle \mapsto \langle s'_1; s_2, M', \Omega', t' \rangle} \\
[S3] \quad \langle \text{skip}; s, M, \Omega, t \rangle \mapsto \langle s, M, \Omega, t + 1 \rangle \quad [S4] \quad \langle \text{fi}; s, M, \Omega, t \rangle \mapsto \langle s, M, \Omega, t \triangleright 1 \rangle \\
[S5] \frac{\langle e, M \rangle \Downarrow n \quad n > 0}{\langle \text{if } e \text{ then } s_1 \text{ else } s_2, M, \Omega, t \rangle \mapsto \langle s_1; \text{fi}, M, \Omega, t < 1 \rangle} \quad [S6] \frac{\langle e, M \rangle \Downarrow n \quad n \leq 0}{\langle \text{if } e \text{ then } s_1 \text{ else } s_2, M, \Omega, t \rangle \mapsto \langle s_2; \text{fi}, M, \Omega, t < 1 \rangle} \\
[S7] \frac{\langle \bar{e}, M \rangle \Downarrow \bar{v}_1 \quad \bar{v}_1 \neq \text{none}}{\langle \text{exec}(c[\bar{v}], \eta, pc, \mathcal{Q}, \bar{e}), M, \Omega, t \rangle \mapsto \langle \text{halt}, M, \Omega \cup [\text{exec}(c[\bar{v}], \eta) :: pc, \bar{v}_1, \mathcal{Q}, t], t + 1 \rangle} \\
[S8] \quad \langle \text{chmod}(c[\bar{v}], \eta, pc, \mathcal{Q}, \ell), M, \Omega, t \rangle \mapsto \langle \text{skip}, M, \Omega \cup [\text{chmod}(c[\bar{v}], \eta) :: pc, \ell, \mathcal{Q}, t], t + 1 \rangle \\
[S9] \frac{\langle e, M \rangle \Downarrow v \quad v \neq \text{none}}{\langle \text{setvar}(\langle c[\bar{v}].z, \eta \rangle, e), M, \Omega, t \rangle \mapsto \langle \text{skip}, M, \Omega \cup [\text{setvar}(c[\bar{v}].z, \eta) :: v, t], t + 1 \rangle} \\
[G1] \frac{\langle s, M, \Omega, t \rangle \mapsto \langle s', M', \Omega', t' \rangle \quad \mathcal{M}(h, t) = M \quad \mathcal{E}' = (\text{if } \Omega' = \Omega \cup \{\mu\} \text{ then } \mathcal{E}[\text{messages}(h) \mapsto_+ \mu] \text{ else } \mathcal{E})}{\langle \langle s, t, h, c[\bar{v}], \eta \rangle \parallel \Theta, \mathcal{M}, \mathcal{E} \rangle \mapsto \langle \langle s', t', h, c[\bar{v}], \eta \rangle \parallel \Theta, \mathcal{M}[h \mapsto_t M'], \mathcal{E}' \rangle} \\
[M1] \frac{\begin{array}{l} \mathcal{E}.\text{closure}(h, c[\bar{v}], \eta) = \langle c[\bar{v}], \eta, \ell, \mathcal{A}, t', \text{on} \rangle \quad P(c[\bar{v}]) = c[\bar{v}]\{pc', \mathcal{Q}', \bar{\pi} \triangleright z : \bar{\tau}_2, \lambda \bar{y} : \bar{\tau}_1.s\} \quad \forall z_i. \mathcal{A}(z_i) \neq \text{none} \\ \mathcal{E}.\text{messages}(*, h, [c[\bar{v}], \eta, \text{exec} :: *]) = \langle H, h, \bar{\mu} \rangle \quad \text{LT}[\ell](H, \bar{\mu}) = [c[\bar{v}], \eta, \text{exec} :: pc, \bar{v}_1, \mathcal{Q}, t] \\ \exists W \in \mathcal{Q}. W \subseteq H \quad pc \sqsubseteq \ell \quad \Gamma \vdash \bar{v}_1 : \bar{\tau}_1 \quad t'' = (\text{if } pc \sqsubseteq pc' \text{ then } \text{inc}(t, pc') \text{ else } t') \quad t' \neq \text{none} \Rightarrow t \leq t' \\ \mathcal{E}' = \mathcal{E}[\text{closure}(h, c[\bar{v}], \eta) \mapsto \langle c[\bar{v}], \eta, \ell, \mathcal{A}, t'', \text{off} \rangle] \quad \mathcal{A}' = \mathcal{A}[\bar{y} \mapsto \bar{v}_1][\text{cid} \mapsto \eta][\text{nid} \mapsto \text{hash}(t'')] \end{array}}{\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \mapsto \langle \Theta \parallel \langle s[\mathcal{A}'], t'', h, c[\bar{v}], \eta \rangle, \mathcal{M}, \mathcal{E}' \rangle} \\
[M2] \frac{\begin{array}{l} \mathcal{E}.\text{closure}(h, c[\bar{v}], \eta) = \langle c[\bar{v}], \eta, \ell, \mathcal{A}, t', \text{on} \rangle \quad \mathcal{E}.\text{messages}(*, h, [c[\bar{v}], \eta, \text{chmod} :: x, y, *], x \sqsubseteq \ell \sqsubseteq y) = \langle H, h, \bar{\mu} \rangle \\ \text{LT}[\ell](H, \bar{\mu}) = [c[\bar{v}], \eta, \text{chmod} :: pc, \ell', \mathcal{Q}, t] \quad \exists W \in \mathcal{Q}. W \subseteq H \quad \ell \neq \ell' \quad t'' = (\text{if } pc \sqsubseteq pc' \text{ then } \text{inc}(t, \ell) \text{ else } t') \end{array}}{\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \mapsto \langle \Theta, \mathcal{M}, \mathcal{E}[\text{closure}(h, c[\bar{v}], \eta) \mapsto \langle c[\bar{v}], \eta, \ell', \mathcal{A}, t'', \text{on} \rangle] \rangle} \\
[M3] \frac{\begin{array}{l} \mathcal{E}.\text{closure}(h, c[\bar{v}], \eta) = \langle c[\bar{v}], \eta, \ell, \mathcal{A}, t', \text{on} \rangle \quad \mathcal{A}(z_i) = \text{none} \quad P(c[\bar{v}]) = c[\bar{v}]\{pc', H', \bar{\pi} \triangleright z : \bar{\tau}, \lambda \bar{y} : \bar{\tau}_1.s\} \\ \mathcal{E}.\text{messages}(*, h, [c[\bar{v}].z_i, \eta], \text{setvar} :: *) = \langle H, h, \bar{\mu} \rangle \quad \pi_i(H, \bar{\mu}) = [c[\bar{v}].\eta, z_i, \text{setvar} :: v, t] \quad \Gamma \vdash v : \tau_i[\bar{v}/\bar{x}] \end{array}}{\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \mapsto \langle \Theta, \mathcal{M}, \mathcal{E}[\text{closure}(h, c[\bar{v}], \eta) \mapsto \langle c[\bar{v}], \eta, \ell, \mathcal{A}[z_i \mapsto v], t', \text{on} \rangle] \rangle} \\
[A1] \frac{I(h) \leq l_A \quad \mathcal{M}(h, t) = M \quad \Gamma(m) = \sigma \text{ or } \sigma @ \mathcal{Q} \quad M' = M[m \mapsto v @ t] \quad \Gamma \vdash v : \sigma}{\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \mapsto \langle \Theta, \mathcal{M}[h \mapsto_t M'], \mathcal{E} \rangle} \quad [A2] \frac{I(h) \leq l_A \quad \Gamma \vdash \mu}{\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \mapsto \langle \Theta, \mathcal{M}, \mathcal{E}[\text{messages}(h, h') \mapsto_+ \mu] \rangle} \\
[A3] \frac{A(h) \leq l_A}{\langle \langle s, t, h, c[\bar{v}], \eta \rangle \parallel \Theta, \mathcal{M}, \mathcal{E} \rangle \mapsto \langle \langle \text{abort}, t, h, c[\bar{v}], \eta \rangle \parallel \Theta, \mathcal{M}, \mathcal{E} \rangle}
\end{array}$$

Figure 9: Operational semantics of DSR with respect to  $\Gamma$  and  $P$

- $\mathcal{M}(h, t)$ : 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  $t$ .
- $\mathcal{E}[\text{messages}(h) \mapsto_+ \mu]$ : the environment obtained by adding the message  $\mu$  sent by host  $h$  to  $\mathcal{E}$ . Suppose  $\mathcal{E}[\text{messages}(h) \mapsto_+ \mu] = \mathcal{E}'$ . Then  $\mathcal{E}'.MT[h, h'] = \mathcal{E}.MT[h, h'] \cup \{\mu\}$  for any  $h' \in \text{receivers}(\mu)$ , and for any other host pair  $h_1, h_2$ ,  $\mathcal{E}'.MT[h_1, h_2] = \mathcal{E}.MT[h_1, h_2]$ .
- $\mathcal{E}[\text{messages}(h_1, h_2) \mapsto_+ \mu]$ : the environment obtained by adding  $\mu$  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  $\mathcal{M}'[h', m', t'] = \mathcal{M}[h', m', t']$ .
- $\mathcal{E}[\text{closure}(h, c[\bar{v}], \eta) \mapsto k]$ : the environment obtained by mapping  $\langle h, c[\bar{v}], \eta \rangle$  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  $\Gamma$  of memory, which maps references to types. For succinctness,  $\Gamma$  and  $P$  are implicitly used by the evaluation rules in Figure 5.2. 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`, ending the execution of a conditional statement.

Rules (E1)–(E3) are used to evaluate expressions. These rules are standard. Because expressions have no side effects, the notation  $\langle e, M \rangle \Downarrow v$  means that evaluating  $e$  in  $M$  results in the value  $v$ . The notation  $M(m)$  represents the value of  $m$  in  $M$ . If  $M[m] = v@t$ , then  $M(m)$  is  $v@t$  if  $m$  is replicated on multiple hosts, and  $v$  otherwise.

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

Rules (S1)–(S6) are largely standard. The interesting part is the manipulation of timestamps. Each evaluation step increments the local part of the timestamp  $t$ . 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 component is appended to the local part of  $t$ ; when exiting a branch in (S4), the last component is discarded. Given  $t = \langle \overline{pc} : \bar{n}; n'_1, \dots, n'_k \rangle$ , the following auxiliary functions manipulate timestamps:

$$\begin{aligned} t + 1 &= \langle \overline{pc} : \bar{n}; n'_1, \dots, n'_k + 1 \rangle \\ t \triangleleft 1 &= \langle \overline{pc} : \bar{n}; n'_1, \dots, n'_k, 1 \rangle \\ t \triangleright 1 &= \langle \overline{pc} : \bar{n}; n'_1, \dots, n'_{k-1} + 1 \rangle \end{aligned}$$

Rules (S7)–(S9) evaluate the three reactor primitives. They all send out a network message encoding the corresponding command. In rule (S7), the `exec` statement produces the message  $[c[\bar{v}], \eta, \text{exec} :: pc, \bar{v}_1, \mathcal{Q}, t]$ , where  $\mathcal{Q}$  is the quorum system of the sender that potentially contains an unstable memory update. The destination hosts of this message are determined by  $c[\bar{v}]$ . After the execution of an `exec` statement, the current thread is terminated, evaluating to `halt`.

A global evaluation step is a transition  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \mapsto \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$ . 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 a reactor instance  $\langle c[\bar{v}], \eta \rangle$ . The following auxiliary function retrieves the set of messages with some property from environment  $\mathcal{E}$ :

$$\mathcal{E}.\text{messages}(\tilde{h}_s, \tilde{h}_r, \tilde{\mu}, \mathcal{C}) = \langle \bar{h}, \bar{h}', \bar{\mu} \rangle$$

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  $\langle \bar{h}, \bar{h}', \bar{\mu} \rangle$  is a list of  $\langle h_i, h'_i, \mu_i \rangle$  tuples where  $h_i$  and  $h'_i$  are the sender and  $\mu_i$  matches the pattern and satisfies  $\mathcal{C}$ . To abuse notation a bit,  $\bar{h}$  can be represented by  $H = \{h_1, \dots, h_n\}$ , or  $h_s$  if  $\bar{h} = h_s, \dots, h_s$ . For example, in rule (M1), the function  $\mathcal{E}.messages(*, h, [c[\bar{v}], \eta, \text{exec} :: *])$  returns all the messages in  $\mathcal{E}$  that are sent to  $h$  and have the message head  $\langle c[\bar{v}], \eta, \text{exec} \rangle$ . The result of the function is  $\langle H, h, \bar{\mu} \rangle$ , where  $H = \{h_1, \dots, h_n\}$ , and  $h_i$  sends  $\mu_i$  to  $h$ . Therefore,  $H$  and  $\bar{\mu}$  can be fed to the message synthesizer  $\text{LT}[\ell]$  (abbreviation for  $\text{LT}[I(\ell)]$ ), where  $\ell$  is the access control label of the closure of  $\langle c[\bar{v}], \eta \rangle$ . This guarantees that the set of senders have sufficient integrity to invoke the reactor instance. The closure of  $\langle c[\bar{v}], \eta \rangle$  on host  $h$  is retrieved from  $\mathcal{E}$  by the auxiliary function  $\mathcal{E}.closure(h, c[\bar{v}], \eta)$ . A reactor closure has six attributes:  $\ell$ , the access control label,  $\mathcal{A}$ , an argument map that maps variables  $\bar{z}$  to values,  $t$ , 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  $P(c)$  gives the declaration of reactor  $c$ . Then  $P(c[\bar{v}]) = P(c)[\bar{v}/\bar{x}]$ , where  $\bar{x}$  are parameters of  $c$ . Once  $\text{LT}[\ell]$  returns an invocation request  $[c[\bar{v}], \eta, \text{exec} :: pc, \bar{v}_1, \mathcal{Q}, t]$ , host  $h$  verifies a few constraints to ensure the validity of the request:

- $\forall z_i. \mathcal{A}(z_i) \neq \text{none}$ . This constraint guarantees that variables  $\bar{z}$  are all initialized.
- $\exists W \in \mathcal{Q}. W \subseteq H$ . This constraint ensures that all memory updates of the caller are stable.
- $pc \sqsubseteq \ell$ . This label constraint controls the implicit flows by ensuring the program point of the caller has sufficient integrity and does not reveal confidential information.
- $\Gamma \vdash \bar{v}_1 : \bar{\tau}_1$ . Run-time type checking ensures that the arguments of the request are well-typed.

After the request is checked, host  $h$  creates a new thread whose code is  $s[\mathcal{A}']$ , meaning the result of applying substitution  $\mathcal{A}'$  to  $s$ . In particular, the current closure identifier **cid** is  $\eta$ , and the new closure identifier **nid** is the hash of the current timestamp  $t''$ , which is either  $t'$ , or  $\text{inc}(t, pc')$  if  $pc \sqsubseteq pc'$ . 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 the reactor instance  $\langle c[\bar{v}], \eta \rangle$ . Like in (M1), the closure  $\langle c[\bar{v}], \eta, \ell, \mathcal{A}, t', \text{on} \rangle$  of the reactor instance is retrieved from  $\mathcal{E}$ ;  $\text{LT}[\ell]$  is used to synthesize the **chmod** messages. Then rule (M2) verifies three constraints. The quorum constraint and the label constraint  $pc \sqsubseteq \ell$  are similar to those in (M1). The label constraint  $\ell \neq \ell'$  ensures that the **chmod** request has not been processed. Once the request is accepted, the closure's timestamp is initialized if necessary, and its access control label is set to  $\ell'$ .

Rule (M3) handles **setvar** messages. Suppose the corresponding request is to initialize variable  $z_i$  of the reactor instance  $\langle c[\bar{v}], \eta \rangle$ . Then  $\pi_i$  is the message synthesizer to use, according to the declaration of  $c[\bar{v}]$ . If  $\pi_i$  returns a **setvar** request with a well-typed initial value  $v$ , and  $z_i$  has not yet been initialized, then  $z_i$  is bound to  $v$  in 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  $I(h) \leq l_A$  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  $\mu$  is sent from host  $h$ . Again, we assume that  $\mu$  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.

$$\begin{array}{c}
\text{[INT]} \quad \vdash n : \text{int}_\ell \quad \text{[CID]} \quad \vdash \eta : \text{int}_\ell \quad \text{[LABEL]} \quad \vdash \{C = l_1, I = l_2, A = l_3\} : \text{label}_\ell \quad \text{[VAR]} \quad \Gamma \vdash x : \Gamma(x) \\
\\
\text{[REACTOR]} \quad \frac{
\begin{array}{c}
P(c) = c[\bar{x}:\bar{\sigma}]\{pc, Q, \bar{\pi} \triangleright z:\bar{\tau}_1, \lambda \bar{y}:\bar{\tau}_2. s\} \\
\Gamma \vdash \bar{v} : \bar{\sigma}[\bar{v}/\bar{x}] \quad \ell \sqsubseteq \bar{\sigma}[\bar{v}/\bar{x}] \quad \ell \sqsubseteq pc[\bar{v}/\bar{x}] \\
C_{\sqcup}(\bar{\tau}_1[\bar{v}/\bar{x}]) \sqcup C_{\sqcup}(\bar{\tau}_2[\bar{v}/\bar{x}]) \sqcup C(\text{pc}[\bar{v}/\bar{x}]) \leq C_{\cap}(Q[\bar{v}/\bar{x}])
\end{array}
}{\Gamma; P \vdash c[\bar{v}] : \text{reactor}[\bar{v}/\bar{x}]\{pc, \bar{\pi} \triangleright z:\bar{\tau}_1, \bar{\tau}_2\}_\ell}
\quad
\text{[ARG]} \quad \frac{
\begin{array}{c}
\Gamma; P \vdash c[\bar{v}] : \text{reactor}\{pc, \bar{\pi} \triangleright z:\bar{\tau}, \bar{\tau}_2\}_\ell \\
\vdash v : \text{int}_\ell \quad FV(\bar{v}) = \emptyset \quad \ell \sqsubseteq \tau_i
\end{array}
}{\Gamma; P \vdash \langle c[\bar{v}].z_i, v \rangle : (\pi_i \otimes \tau_i \text{ var})_\ell}
\\
\\
\text{[LOC]} \quad \frac{\Gamma(m) = \tau}{\Gamma \vdash m : (\tau \text{ ref})_\ell} \quad \text{[TV]} \quad \frac{\Gamma \vdash v : \sigma}{\Gamma; Q \vdash v@t : \sigma@Q} \quad \text{[ADD]} \quad \frac{\Gamma \vdash e_i : \text{int}_{\ell_i} \quad i \in \{1, 2\}}{\Gamma \vdash e_1 + e_2 : \text{int}_{\ell_1 \sqcup \ell_2}} \quad \text{[DEREF]} \quad \frac{\Gamma \vdash e : (\tau \text{ ref})_\ell \quad \text{readable}(Q, \tau)}{\Gamma; Q \vdash !e : \tau \sqcup \ell}
\\
\\
\text{[ASSI]} \quad \frac{\Gamma \vdash v : (\tau \text{ ref})_\ell \quad \Gamma \vdash e : \sigma \quad \text{base}(\tau) = \sigma \quad \text{writable}(Q, \tau) \quad pc \sqcup \ell \sqsubseteq \sigma}{\Gamma; Q; pc \vdash v := e : \text{stmt}_{pc}} \quad \text{[SEQ]} \quad \frac{\Gamma; P; Q; pc \vdash s_1 : \text{stmt}_{pc_1} \quad \Gamma; P; Q; pc_1 \vdash s_2 : \text{stmt}_{pc_2}}{\Gamma; P; Q; pc \vdash s_1; s_2 : \text{stmt}_{pc_2}} \quad \text{[IF]} \quad \frac{\Gamma; Q \vdash e : \text{int}_\ell \quad \Gamma; P; Q; pc \sqcup \ell \vdash s_i : \tau \quad i \in \{1, 2\}}{\Gamma; P; Q; pc \vdash \text{if } e \text{ then } s_1 \text{ else } s_2 : \tau}
\\
\\
\text{[EXEC]} \quad \frac{\Gamma; P \vdash v_1 : \text{reactor}\{pc', \bar{\pi} \triangleright z:\bar{\tau}, \bar{\tau}_2\}_\ell \quad \Gamma; Q \vdash v_2 : \text{int}_\ell \quad \Gamma \vdash \ell : \text{label}_\ell \quad \Gamma; P; Q \vdash \bar{e} : \bar{\tau}_2 \quad pc \sqsubseteq \bar{\tau}_2 \quad pc \sqsubseteq \ell}{\Gamma; P; Q; pc \vdash \text{exec}(v_1, v_2, \ell, Q, \bar{e}) : \text{stmt}_{pc}} \quad \text{[CHMD]} \quad \frac{\Gamma; P \vdash v_1 : \text{reactor}\{pc', \bar{\pi} \triangleright z:\bar{\tau}, \bar{\tau}_2\}_\ell \quad \Gamma; Q \vdash v_2 : \text{int}_\ell \quad \Gamma \vdash \ell : \text{label}_\ell \quad \Gamma \vdash \ell' : \text{label}_\ell \quad pc \sqsubseteq \ell \quad \ell \sqsubseteq \ell'}{\Gamma; P; Q; pc \vdash \text{chmod}(v_1, v_2, \ell, Q, \ell') : \text{stmt}_{\ell'}}
\\
\\
\text{[SETV]} \quad \frac{\Gamma; Q \vdash v : (\tau \text{ var})_\ell \quad \Gamma; Q \vdash e : \tau \quad pc \sqcup \ell \sqsubseteq \tau}{\Gamma; P; Q; pc \vdash \text{setvar}(v, e) : \text{stmt}_{pc}} \quad \text{[RD]} \quad \frac{\Gamma, \bar{x}:\bar{\sigma}, \bar{y}:\bar{\tau}_1, \bar{z}:\bar{\tau}_2, \text{cid}:\text{int}_{pc}, \text{nid}:\text{int}_{pc}; P; Q; pc \vdash s : \text{stmt}_{pc'}}{\Gamma; P \vdash c[\bar{\sigma} \bar{x}]\{pc, Q, \bar{\pi} \triangleright \bar{\tau} \bar{z}, \lambda \bar{y}. s\}}
\\
\\
\text{[Auxiliary notations]} \\
\pi \otimes \tau : \text{QR}[Q] \otimes \sigma = \sigma@Q \quad \text{LT}[I] \otimes \tau = \tau \quad \text{writable}(Q, \tau) : \tau = \sigma@Q \vee (\tau = \sigma \wedge |Q| = \{h\}) \\
\text{readable}(Q, \tau) : (\tau = \sigma@Q' \wedge |Q| = |Q'|) \vee (\tau = \sigma \wedge |Q| = \{h\})
\end{array}$$

Figure 10: Typing rules

### 5.3 Type system

The typing rules of DSR are shown in Figure 10. A program  $P$  is well-typed in  $\Gamma$ , written as  $\Gamma \vdash P$ , if every reactor declaration  $r$  in  $P$  is well-typed with respect to  $\Gamma$  and  $P$ , written  $\Gamma; P \vdash r$ , where  $P$  provides the typing information of reactors.

A reactor declaration is well-typed if its body statement is well-typed. The typing judgment for a statement  $s$  has the form  $\Gamma; P; Q; pc \vdash s : \tau$ , meaning that  $s$  has type  $\tau$  under the typing environment  $\Gamma; P; Q; pc$ , where  $s$  is replicated on  $Q$ , and  $pc$  is the program counter label. The typing judgment for an expression  $e$  has the form  $\Gamma; P; Q \vdash e : \tau$ , meaning that  $e$  has type  $\tau$  under the typing environment  $\Gamma; P; Q$ . For simplicity, a component in the typing environment of a typing judgment may be omitted if the component is irrelevant. The notations  $\ell \sqsubseteq \sigma$  and  $\ell \sqsubseteq \sigma@Q$  represent  $\ell \sqsubseteq \ell'$  if  $\sigma = \beta\ell'$ . Rules (INT), (CID), (LABEL), (VAR), (LOC), (ADD) and (IF) are standard.

Rule (REACTOR) checks reactor  $c[\bar{v}]$ . Suppose the declaration of  $c$  is  $c[\bar{x}:\bar{\sigma}]\{pc, Q, \bar{\pi} \triangleright z:\bar{\tau}, \lambda \bar{y}:\bar{\tau}. s\}$ . Then the list of parameters  $\bar{v}$  must have types  $\bar{\sigma}[\bar{v}/\bar{x}]$ , where the substitution is necessary because  $\bar{x}$  may appear in  $\bar{\sigma}$ . The values of the reactor parameters and the effects of this reactor depend on the reactor value itself. Thus,  $\ell \sqsubseteq \bar{\sigma}[\bar{v}/\bar{x}]$  and  $\ell \sqsubseteq pc[\bar{v}/\bar{x}]$  are enforced. Since this reactor is replicated on  $Q' = Q[\bar{v}/\bar{x}]$ , any data processed by the reactor is observable to the hosts in  $Q'$ . The last constraint ensures that the hosts in  $Q'$  would not leak the data of  $c[\bar{v}]$ .

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

Rule (TV) checks versioned values. If  $v$  has type  $\sigma$ , then  $v@t$  has type  $\sigma@Q$  where  $Q$  is the quorum system producing this value.

Rules (DEREF) and (ASSI) are largely standard. These two rules need to ensure that the involved memory reference is accessible on  $Q$ . In rule (DEREF),  $\text{readable}(Q, \tau)$  means that data with type  $\tau$  can be

generated from the host set of  $\mathcal{Q}$ . In rule (ASSI),  $writable(\mathcal{Q}, \tau)$  ensures that the reference to be assigned is replicated on  $\mathcal{Q}$ . The function  $base(\tau)$  strips the location part of  $\tau$ .

Rule (SEQ) checks sequential statement  $s_1; s_2$ . If  $s_1$  has type  $stmt_{pc_1}$ , then  $s_2$  is checked with  $pc_1$ , since  $pc_1$  is the program counter label when  $s_1$  terminates.

Rule (EXEC) checks `exec` statements. It resembles checking a function call. The constraints  $pc \sqsubseteq \bar{\tau}_2$  and  $pc \sqsubseteq \ell$  ensure that the reactor to be invoked would not leak the information about the current program counter.

Rule (CHMD) checks `chmd` statements. The label  $\ell'$  is meant to be the new access control label of  $\langle v_1, v_2 \rangle$ . After executing this statement, the program counter label becomes  $\ell'$ , effectively preventing the following code from running another `chmd` statement with label  $\ell$  before  $\langle v_1, v_2 \rangle$  is invoked. The constraint  $\ell \sqsubseteq \ell'$  implies  $pc \sqsubseteq \ell'$ , guaranteeing 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  $(\tau \text{ var})_\ell$ , representing a remote variable. The value of expression  $e$  is used to initialize the remote variable, and thus  $e$  has type  $\tau$ . 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) is used to check reactor declarations. The declaration  $c[x:\sigma]\{pc, \mathcal{Q}, \bar{\pi} \triangleright z:\bar{\tau}, \lambda \bar{y}:\bar{\tau}. s\}$  is well-typed with respect to  $\Gamma$  and  $P$  as long as the reactor body  $s$  is well-typed in the typing environment  $\Gamma, \bar{x}:\bar{\tau}, \bar{y}:\bar{\tau}_1, \bar{z}:\bar{\tau}_2; P; \mathcal{Q}; pc$ .

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

**Definition 5.1** (Well-typed memory).  $\mathcal{M}$  is well-typed in  $\Gamma$ , 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 5.2** (Well-typed environment).  $\mathcal{E}$  is well-typed in  $\Gamma$  and  $P$ , written  $\Gamma; P \vdash \mathcal{E}$ , if for any closure  $\langle c[\bar{v}], \eta, \ell, t, \mathcal{A}, * \rangle$  in  $\mathcal{E}$  and any  $x \in dom(\mathcal{A})$ ,  $\vdash \mathcal{A}(x) : \tau$  where  $\tau$  is the type of  $x$  based on  $\Gamma$  and  $c[\bar{v}]$ , and for any  $\mu$  in  $\mathcal{E}$ , we have  $\Gamma; P \vdash \mu$ , which means the contents of  $\mu$  are well-typed. The inference rules for  $\Gamma; P \vdash \mu$  are standard:

$$\begin{array}{c}
[M-EXEC] \quad \frac{\Gamma; P \vdash c[\bar{v}] : \text{reactor}\{pc', \bar{\pi} \triangleright z:\bar{\tau}_1, \bar{\tau}_2\} \\ \quad \vdash \bar{v}_1 : \bar{\tau}_1}{\Gamma; P \vdash [\text{exec } \langle c[\bar{v}], \eta \rangle :: pc, \bar{v}_1, \mathcal{Q}, t]} \quad [M-CHMD] \quad \frac{\Gamma; P \vdash c[\bar{v}] : \text{reactor}\{pc', \bar{\pi} \triangleright z:\bar{\tau}_1, \bar{\tau}_2\}}{\Gamma; P \vdash [\text{chmd } \langle c[\bar{v}], \eta \rangle :: pc, \ell, \mathcal{Q}, t]} \\
[M-SETV] \quad \frac{\Gamma; P \vdash \langle c[\bar{v}].z, \eta \rangle : (\tau \text{ var})_\ell \quad \vdash v_1 : \tau}{\Gamma; P \vdash [\text{setvar } \langle c[\bar{v}].z, \eta \rangle :: v_1, t]}
\end{array}$$

**Definition 5.3** (Well-typed configuration).  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle$  is well-typed in  $\Gamma$  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  $\langle s, t, h, c[\bar{v}], \eta \rangle$  in  $\Theta$ ,  $\Gamma; P; \mathcal{Q}; pc \vdash s : \tau$ .

**Lemma 5.1** (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  $\tau$  is  $\Gamma(m)$ , and  $v$  is  $M(m)$ . If  $\Gamma(m) = \text{int}_\ell$ , then  $M(m) = n$  while  $M[m] = n @ t$ , and  $\Gamma; \mathcal{Q} \vdash n : \text{int}_\ell$ . Otherwise,  $\Gamma(m) = \text{int}_\ell @ \mathcal{Q}$ , and  $M(m) = M[m] = n @ t$ . We have  $\Gamma; \mathcal{Q} \vdash n @ t : \text{int}_\ell @ \mathcal{Q}$ .
- Case (E2). By induction,  $\Gamma; \mathcal{Q} \vdash v_i : \text{int}_{\ell_i}$  for  $i \in \{1, 2\}$ . Thus,  $\Gamma; \mathcal{Q} \vdash v_1 + v_2 : \text{int}_{\ell_1 \sqcup \ell_2}$ .

□

**Lemma 5.2** (Substitution). Suppose  $\Gamma, x : \tau \vdash s : \tau'$ , and  $\Gamma \vdash v : \tau$ . Then  $\Gamma \vdash s[v/x] : \tau'[v/x]$ .

*Proof.* By induction on the structure of  $s$ . □

**Lemma 5.3** (Subject reduction). Suppose  $\Gamma \vdash \langle s, M, \Omega, t \rangle \mapsto \langle s', M', \Omega', t' \rangle$ , and  $\Gamma; P; Q; pc \vdash s : \tau$  and  $\Gamma \vdash M, \Omega$ . Then  $\Gamma; P; Q; pc \vdash s' : \tau$  and  $\Gamma \vdash M', \Omega'$ .

*Proof.* By induction on the derivation of  $\langle s, M, \Omega, t \rangle \mapsto \langle s', M', \Omega', t' \rangle$ . Here we only show the cases (S1) and (S7). Other cases are similar.

- Case (S1). By rule (ASSI),  $\Gamma; Q \vdash m : (\tau \text{ ref})_\ell$  and  $\Gamma; Q \vdash e : \tau$ . By Lemma 5.1,  $\Gamma; Q \vdash v : \tau$ . Therefore,  $\Gamma \vdash M[m \mapsto v @ t]$ .
- Case (S7). Suppose the type of  $c[\bar{v}]$  is  $\text{reactor}\{pc', Q', \overline{\pi \triangleright \tau z}, \overline{\tau_1}\}$ . By Lemma 5.1,  $\Gamma; Q \vdash \bar{v}_1 : \overline{\tau_1}$ . Therefore, the new `exec` message is well-typed, and  $\Gamma \vdash \Omega'$ .

□

**Theorem 5.1** (Subject reduction). Suppose  $\Gamma; P \vdash \langle \Theta, \mathcal{M}, \mathcal{E} \rangle$ , and  $\Gamma; P \vdash \langle \Theta, \mathcal{M}, \mathcal{E} \rangle \mapsto \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$ . Then  $\Gamma; P \vdash \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$ .

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

- Case (G1). The evaluation step is derived from  $\langle s, M, \Omega, t \rangle \mapsto \langle s', M', \Omega', t' \rangle$  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,  $\Omega' = \Omega \cup \{\mu\}$  where  $\mu$  is well-typed, and  $\mathcal{E}' = \mathcal{E}[\text{messages}(h) \mapsto_+ \mu]$ . Therefore,  $\mathcal{E}'$  is well-typed.
- Case (M1). In this case, we only need to prove that the newly created thread is well-typed. Since  $\Gamma \vdash \bar{v}_1 : \overline{\tau_1}$ . By  $\Gamma \vdash \bar{v}_1 : \overline{\tau_1}[\bar{v}/\bar{x}]$ , we have  $\Gamma' \vdash \mathcal{A}'$ . By Lemma 5.2,  $\Gamma' \vdash s[\mathcal{A}'] : \tau'$ .
- 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  $\Gamma \vdash v : \tau_i[\bar{v}/\bar{x}]$ . 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.

□

## 6 Noninterference

The noninterference results of DSR are concerned with confidentiality and integrity. Intuitively, confidentiality noninterference means that high-confidentiality inputs do not interfere with low-confidentiality outputs; integrity noninterference means that low-integrity inputs do not interfere with high-integrity outputs.

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). 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, intuitively 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

$$\begin{array}{c}
\text{[VE1]} \quad v \approx v \qquad \text{[VE2]} \quad \text{none} \approx v \qquad \text{[VE3]} \quad \frac{t_1 = t_2 \Rightarrow v_1 \approx v_2}{v_1 @ t_1 \approx v_2 @ t_2} \\
\text{[MSE1]} \quad \frac{P(c[\bar{v}]) = c\{\text{pc}', \mathcal{Q}, \overline{\pi \triangleright z : \tau}, \lambda \bar{x} : \overline{\tau_1}.s\} \quad \forall i. \zeta(\tau_{1i}) \Rightarrow v_{1i} \approx v_{2i}}{[\text{exec} \langle c[\bar{v}], \eta \rangle :: \text{pc}, \bar{v}_1, \mathcal{Q}, t] \approx_{\zeta} [\text{exec} \langle c[\bar{v}], \eta \rangle :: \text{pc}, \bar{v}_2, \mathcal{Q}, t]} \\
\text{[MSE2]} \quad \frac{\zeta(\text{pc}) \Rightarrow \ell_1 = \ell_2}{[\text{chmod} \langle c[\bar{v}], \eta \rangle :: \text{pc}, \ell_1, \mathcal{Q}, t] \approx_{\zeta} [\text{chmod} \langle c[\bar{v}], \eta \rangle :: \text{pc}, \ell_2, \mathcal{Q}, t]} \\
\text{[MSE3]} \quad \frac{\zeta(c[\bar{v}].z) \Rightarrow v_1 \approx v_2}{[\text{setvar} \langle c[\bar{v}].z, \eta \rangle :: v_1, t] \approx_{\zeta} [\text{setvar} \langle c[\bar{v}].z, \eta \rangle :: v_2, t]} \\
\text{[ME]} \quad \frac{\forall h_1, h_2, m, t. \zeta(m, h_1) \wedge \zeta(m, h_2) \wedge t \leq \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]}{\Gamma \vdash \langle \mathcal{M}_1, \mathcal{T}_1 \rangle \approx_{\zeta} \langle \mathcal{M}_2, \mathcal{T}_2 \rangle} \\
\text{[CE]} \quad \frac{\text{varmap}(P, c[\bar{v}]) \vdash \mathcal{A}_1 \approx_{\zeta} \mathcal{A}_2 \quad \zeta(c[\bar{v}]) \Rightarrow t_1 = t_2}{P \vdash \langle c[\bar{v}], \eta, \ell_1, \mathcal{A}_1, t_1, * \rangle \approx_{\zeta} \langle c[\bar{v}], \eta, \ell_2, \mathcal{A}_2, t_2, * \rangle} \\
\text{[EE]} \quad \frac{\forall h_1, h_2. \forall t \leq \min(\mathcal{T}_1(h_1, t), \mathcal{T}_2(h_2, t)). \\ ((\exists j \in \{1, 2\}. \langle h_j, h'_j, \mu_j \rangle \in \mathcal{E}_j.\text{messages}(h_j, *, [* :: *, t]) \wedge \forall i \in \{1, 2\}. \zeta(\mu_j, h_i)) \Rightarrow \\ (\forall i \in \{1, 2\}. \mathcal{E}_i.\text{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[\bar{v}], \eta \rangle. \zeta(c[\bar{v}], h_1) \wedge \zeta(c[\bar{v}], h_2) \Rightarrow P \vdash \mathcal{E}_1.\text{closure}(h_1, c[\bar{v}], \eta) \approx_{\zeta} \mathcal{E}_2.\text{closure}(h_2, c[\bar{v}], \eta)}{P \vdash \langle \mathcal{E}_1, \mathcal{T}_1 \rangle \approx_{\zeta} \langle \mathcal{E}_2, \mathcal{T}_2 \rangle} \\
\text{[TE]} \quad \frac{t_1 \approx t_2}{\langle s_1, t_1, h_1, c[\bar{v}], \eta \rangle \approx_{\zeta} \langle s_2, t_2, h_2, c[\bar{v}], \eta \rangle} \\
\text{[TPE]} \quad \frac{\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\}. \text{stable}_{\zeta}(\Theta_i, \mathcal{Q}, t')}{t \vdash \Theta_1 \approx_{\zeta} \Theta_2} \\
\text{[SE]} \quad \frac{\forall i \in \{1, 2\}. \mathcal{T}_i = \text{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 \\ \min(\max(\mathcal{T}_1, \zeta), \max(\mathcal{T}_2, \zeta)) \vdash \Theta_1 \approx_{\zeta} \Theta_2}{\Gamma \vdash \langle \Theta_1, \mathcal{M}_1, \mathcal{E}_1 \rangle \approx_{\zeta} \langle \Theta_2, \mathcal{M}_2, \mathcal{E}_2 \rangle} \\
\text{[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[\bar{v}], \eta \rangle) \wedge \Gamma; \mathcal{Q}; \text{pc}_i \vdash s_i : \tau \wedge \neg \zeta(\text{pc}_i) \wedge \exists W \in \mathcal{Q}. W \subseteq H}{\text{stable}_{\zeta}(\Theta, \mathcal{Q}, t)}
\end{array}$$

Figure 11:  $\zeta$ -Equivalence relation

different low-confidentiality outputs even with the same low-confidentiality inputs. However, these are not effective confidentiality attacks because differences in low-confidentiality outputs are due to those attacks rather than the differences in high-confidentiality inputs. Therefore, we assume that attackers would not affect low-confidentiality data when the confidentiality noninterference is under consideration.

This section formalizes the notions of confidentiality and integrity noninterference, and proves that a well-typed DSR program has the noninterference properties.

## 6.1 $\zeta$ -Equivalence

Another way to state confidentiality noninterference is that equivalence low-confidentiality inputs always generate equivalent low-confidentiality outputs. Similarly, integrity noninterference means that equivalent high-integrity inputs generate equivalent high-integrity outputs. Abstractly, a noninterference property can be expressed as the preservation of equivalence between the program states that satisfies a  $\zeta$  condition. As for confidentiality and integrity noninterferences, the  $\zeta$  condition intuitively means “low-confidentiality” and “high-integrity”, respectively. Formally, for confidentiality,  $\zeta(x)$  denotes  $C(x) \leq l_A$  if  $x$  is a label, and  $C(\text{label}(x)) \leq l_A$ , if  $x$  is a program term such as type  $\tau$ , reference  $m$ , host  $h$ , timestamp  $t$  and message  $\mu$ . The label of term  $x$  is defined as follows:

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

For integrity,  $\zeta(x)$  denotes  $C(x) \leq l_A$  or  $C(\text{label}(x)) \leq l_A$ . Whether a term  $x$  satisfies the  $\zeta$  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  $\zeta$  can be determined by  $\zeta(\text{label}(x) \sqcap \text{label}(h))$ , which is written as  $\zeta(x, h)$ .

The key issue in formalizing the noninterference property is to define the  $\zeta$ -equivalence relation between program states, including thread pools, memories, and environments. These equivalence relations are formally defined by inference rules in Figure 11.

Rules (VE1)–(VE3) define an equivalence 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 equivalent to 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 equivalent if  $t_1 = t_2$  implies  $v_1 \approx v_2$ . Two versioned values with different timestamps are considered equivalent, because they may be used in the same way and in the same execution.

Rules (MSE1)–(MSE3) define the  $\zeta$ -equivalence between messages. Rule (MSE1) says that two `exec` messages  $[\text{exec } \langle c[\bar{v}], \eta \rangle :: pc, \bar{v}_1, Q, t]$  and  $[\text{exec } \langle c[\bar{v}], \eta \rangle :: pc, \bar{v}_2, Q, t]$  are  $\zeta$ -equivalent if any two corresponding arguments  $\bar{v}_{1i}$  and  $\bar{v}_{2i}$  are equivalent on condition that  $\zeta(\tau_{1i})$  holds. Intuitively,  $\neg \zeta(\tau_{1i})$  means that values with type  $\tau_{1i}$  can be distinguishable. Rules (MSE2) and (MSE3) are interpreted similarly.

Rule (ME) defines memory  $\zeta$ -equivalence. Intuitively, two global memories  $\mathcal{M}_1$  and  $\mathcal{M}_2$  are considered  $\zeta$ -equivalent with respect to the typing assignment  $\Gamma$ , 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  $\mathcal{M}_1[h_1, m, t] \approx \mathcal{M}_2[h_2, m, t]$ . However, with the knowledge of the thread timestamps,  $\mathcal{M}_1$  and  $\mathcal{M}_2$  may be distinguishable if  $\mathcal{M}_1[h_1, m, t] = n$  and  $\mathcal{M}_2[h_2, m, t] = \text{none}$ , because  $\mathcal{M}_2[h_2, m, t] = \text{none}$  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  $\zeta$ -equivalence 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  $t \leq \min(\mathcal{T}_1(h_1, t), \mathcal{T}_2(h_2, t))$ , 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  $\zeta$ -equivalent 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 reference  $\langle c[\bar{v}], \eta \rangle$  and have  $\zeta$ -equivalent variable records. In this rule, the notation  $\text{varmap}(P, c[\bar{v}])$  represents the local typing assignment  $\Gamma'$  of  $c[\bar{v}]$  with respect to  $P$ , mapping local variables of  $c[\bar{v}]$  to types. The notation  $\Gamma' \vdash \mathcal{A}_1 \approx_\zeta \mathcal{A}_2$  means that for any  $z$  in  $\text{dom}(\Gamma')$ ,  $\zeta(\Gamma'(z))$  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  $\zeta$ -equivalent, and the corresponding (with the same reference) closures are  $\zeta$ -equivalent. 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, the  $\zeta$ -equivalence between two environments  $\mathcal{E}_1$  and  $\mathcal{E}_2$  are 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  $\mu_j$  in  $\mathcal{E}_j$  such that  $\mu_j$  has the timestamp  $t$  and the program counter label  $pc_{\mu_j}$  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 ( $\mu_1$  and  $\mu_2$ , 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  $\langle c[\bar{v}], \eta \rangle$ , if  $\zeta(c[\bar{v}], h_1)$  and  $\zeta(c[\bar{v}], h_2)$ , then the closures identified by  $\langle c[\bar{v}], \eta \rangle$  on hosts  $h_1$  and  $h_2$  are  $\zeta$ -equivalent.

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  $\zeta$ -equivalence between thread pools. Two thread pools  $\Theta_1$  and  $\Theta_2$  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  $\theta_i$  on  $h_i$  and with timestamp  $t_i$  such that  $t_i \approx t'$  in  $\Theta_i$ , then  $\theta_1 \approx_\zeta \theta_2$ . Second, for any timestamp  $t'$  less than  $t$ , if there exists a thread at  $t'$  in either  $\Theta_1$  or  $\Theta_2$ , then the threads at time  $t'$  are *stable* with respect to the quorum system  $\mathcal{Q}$  and the condition  $\zeta$  in both  $\Theta_1$  and  $\Theta_2$ . Intuitively, these two conditions ensure that both  $\Theta_1$  and  $\Theta_2$  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 the function  $\text{timestamps}(\Theta, \mathcal{E}, \zeta)$ . Suppose  $\mathcal{T} = \text{timestamps}(\Theta, \mathcal{E}, \zeta)$ . 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  $\ell$  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  $\text{current-time}(\mathcal{T}, \zeta)$  is the most recent timestamp  $t$  such that  $\mathcal{T}[h, t] = t$  and  $\zeta(t, h)$ . Intuitively,  $\min(\max(\mathcal{T}_1, \zeta), \max(\mathcal{T}_2, \zeta))$  is the current timestamp of the lagging one of the two configuration.

## 6.2 Preventing races

In DSR, a race is used to refer to the scenario that two threads with different closure references are running at the same timestamp or sending messages with the same message head. A race makes it possible for attackers to choose to endorse the operations of one of the two racing threads, and affect the execution that they are not allowed to affect. Furthermore, message races increases the complexity of maintaining consistency between reactor replicas. Therefore, it is desired to prevent races in DSR programs.

The evaluation rule (S7) of DSR halts the current thread 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. The 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 the races between other messages, our approach is to enforce the following linearity constraints:

- A closure reference 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 communication terms (including used by each reactor and ensures those terms to be used only once. The communication terms include reactor names, closure references, context identifiers and remote variables. Given a statement  $s$  and the typing assignment  $\Gamma$  for that statement, let  $RV(s, \Gamma)$  represent the multi-set of communication terms appearing in the `exec` and `setvar` statements in  $s$ . Note that  $RV(s, \Gamma)$  is a multi-set so that multiple occurrences of the same value can be counted. Given a reactor declaration  $r = c[\overline{x:\sigma}]\{pc, \mathcal{Q}, \overline{\pi \triangleright z:\tau}, \lambda \overline{y:\tau}.s\}$ , let  $RV(r, \Gamma)$  denote the multi-set of communication port terms appearing in  $r$  with respect to  $\Gamma$ . Then we have

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

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  $P$ ,  $RV(r, \Gamma)$  is a set.
- **RV2.** If  $\langle c[\overline{v}].z, v \rangle \in RV(r, \Gamma)$ , then  $v$  is either `cid` or `nid`, and for any other  $r'$  in  $P$ ,  $\langle c[\overline{v}].z, \text{cid} \rangle \notin RV(r', \Gamma)$ . Furthermore, if  $v$  is `cid`, then  $c$  has no reactor parameters, and  $\overline{v}$  contains no variables.
- **RV3.** If  $\langle c[\overline{v}], v \rangle \in RV(r, \Gamma)$ , and  $r$  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 instance is allowed to refer to  $\langle c[\overline{v}].z, \text{cid} \rangle$  in its body. According to (RV2), if  $\langle c[\overline{v}].z, \text{cid} \rangle$  appears in reactor  $c'$ , then  $c'$  has no parameters. Therefore, only one reactor instance  $\langle c', \eta \rangle$  can use  $\langle c[\overline{v}].z, \eta \rangle$  without receiving the variable from its caller. By (RV1),  $\langle c', \eta \rangle$  can either initialize the variable or pass it on to another unique reactor instance, ensuring that only one reactor may initialize  $\langle c[\overline{v}].z, \eta \rangle$ . The third condition (RV3) ensures that no two threads with different closure references can invoke the same reactor with the same context identifier. We use the notation  $\Gamma \Vdash P$  to denote that a program  $P$  is well-typed in  $\Gamma$  and satisfies (RV1)–(RV3).

### 6.3 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 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 invariant. Then the subject reduction result of

DSR\* implies that the invariant is preserved during evaluation. In particular, the invariant may represent the  $\zeta$ -equivalence 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 that the preservation of the  $\zeta$ -equivalence between two DSR local configurations. This proof technique was first used by Pottier and Simonet to prove the noninterference result of a security-typed ML-like language [18].

### 6.3.1 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.

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

The bracket constructs cannot be nested, so the subterms of a bracket construct must be DSR terms. Given a DSR\* statement  $s$ , let  $[s]_i$  represent the DSR statements that  $s$  encodes. The projection functions satisfy  $[(s_1, \dots, s_n)]_i = s_i$  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  $\langle s, M, \Omega, t \rangle_1$  means that  $s$  belongs to the first branch of a bracket, and  $s$  can only access the first projection of  $M$ . We write “ $\langle s, M, \Omega, t \rangle$ ” for “ $\langle s, M, \Omega, t \rangle_\bullet$ ”.

The operational semantics of DSR\* is shown in Figure 12. Since DSR\* is used to analyze the local evaluation steps of DSR, only evaluation rules for statements are presented. An evaluation step of a DSR\* statement is denoted by  $\langle s, M, \Omega, t \rangle_i \mapsto \langle s', M', \Omega', t' \rangle_i$ . Most evaluation rules are 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  $M[m \mapsto_i v @ [t]_i]$ , where  $[t]_i$  is the  $i$ th projection of  $t$ . Suppose  $M[m] = v'$ . Then  $M'[m] = ([v']_1, \dots, v @ [t]_i, \dots, [v']_n)$ . In DSR\*, the local part of a timestamp  $t$  may have the form  $\bar{n}$ , or  $\bar{n}, (\bar{n}_1, \dots, \bar{n}_k)$ , which indicates that the execution deviates after local time  $\bar{n}$ . Suppose  $t = \bar{n}, (\bar{n}_1, \dots, \bar{n}_k)$ . Then we have

$$\begin{array}{l} [t]_i = \bar{n}, \bar{n}_i \\ t \triangleleft_i 1 = \bar{n}, (\bar{n}_1, \dots, \bar{n}_i \triangleleft 1, \dots, \bar{n}_k) \\ t \triangleright_i 1 = \bar{n}, (\bar{n}_1, \dots, \bar{n}_i \triangleright 1, \dots, \bar{n}_k) \\ t \triangleright 1 = \bar{n} + 1 \end{array}$$

where  $\bar{n} \triangleleft 1 = \bar{n}, 1$ , and  $\bar{n} \triangleright 1 = \bar{n}_1, \dots, \bar{n}_{k-1} + 1$ , and  $\bar{n} + 1 = \bar{n}_1, \dots, \bar{n}_k + 1$ . If  $t = \bar{n}$ , then  $t \triangleleft_i 1 = \bar{n}, (\epsilon, \dots, 1, \dots, \epsilon)$ .

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 6.1** (Projection i). Suppose  $\langle e, M \rangle \Downarrow v$ . Then  $\langle [e]_i, [M]_i \rangle \Downarrow [v]_i$  holds for  $i \in \{1, \dots, n\}$ .

*Proof.* By induction on the structure of  $e$ . □

$$\begin{array}{l}
[E1] \frac{[M(m)]_i = v}{\langle !m, M, t \rangle_i \Downarrow v} \quad [E2] \frac{\langle e_1, M \rangle_i \Downarrow v_1 \quad \langle e_2, M \rangle_i \Downarrow v_2 \quad v = v_1 \oplus v_2}{\langle e_1 + e_2, M \rangle_i \Downarrow v} \\
[S1] \frac{\langle e, M \rangle_i \Downarrow v}{\langle m := e, M, \Omega, t \rangle_i \mapsto \langle \text{skip}, M[m \mapsto_i v @ [t]_i], \Omega, t + 1 \rangle_i} \quad [S2] \frac{\langle s_1, M, \Omega, t \rangle_i \mapsto \langle s'_1, M', \Omega', t' \rangle_i}{\langle s_1; s_2, M, \Omega, t \rangle_i \mapsto \langle s'_1; s_2, M', \Omega', t' \rangle_i} \\
[S3] \langle \text{skip}; s, M, \Omega, t \rangle_i \mapsto \langle s, M, \Omega, t \rangle_i \quad [S4] \langle \text{fi}; s, M, t \rangle_i \mapsto \langle s, M, t \triangleright_i 1 \rangle_i \\
[S5] \frac{\langle e, M \rangle_i \Downarrow n \quad n > 0}{\langle \text{if } e \text{ then } s_1 \text{ else } s_2, M, \Omega, t \rangle_i \mapsto \langle s_1; \text{fi}, M, \Omega, t <_i 1 \rangle_i} \quad [S6] \frac{\langle e, M \rangle_i \Downarrow n \quad n \leq 0}{\langle \text{if } e \text{ then } s_1 \text{ else } s_2, M, \Omega, t \rangle_i \mapsto \langle s_2; \text{fi}, M, \Omega, t <_i 1 \rangle_i} \\
[S7] \frac{\langle \bar{e}, M \rangle_i \Downarrow \bar{v}_1}{\langle \text{exec}(v, v', pc, \mathcal{Q}, \bar{e}), M, \Omega, t \rangle_i \mapsto \langle \text{halt}, M, \Omega \cup [v, v', \text{exec} :: t, pc, \mathcal{Q}, \bar{v}_1]_i, t + 1 \rangle_i} \\
[S8] \langle \text{chmod}(c[\bar{v}], n, pc, \mathcal{Q}, \ell), M, \Omega, t \rangle_i \mapsto \langle \text{skip}, M, \Omega \cup [c[\bar{v}], \eta, \text{chmod} :: t, pc, \mathcal{Q}, \ell]_i, t + 1 \rangle_i \\
[S9] \langle \text{setvar}(c[\bar{v}].n_1, n, v), M, \Omega, t \rangle_i \mapsto \langle \text{skip}, M, \Omega \cup [c[\bar{v}].z, \eta, \text{setvar} :: t, v]_i, t + 1 \rangle_i \\
[S10] \langle (\text{skip}, \dots, \text{skip}), M, t \rangle \mapsto \langle \text{skip}, M, t \rangle \quad [S11] \langle (\text{fi}, \dots, \text{fi}), M, t \rangle \mapsto \langle \text{skip}, M, t \triangleright 1 \rangle \\
[S12] \frac{\langle e, M \rangle \Downarrow (v_1, \dots, v_n)}{\langle \text{if } e \text{ then } s_1 \text{ else } s_2, M, \Omega, t \rangle \mapsto \langle (\text{if } v_i \text{ then } [s_1]_i \text{ else } [s_2]_i \mid 1 \leq i \leq n), M, \Omega, t \rangle} \\
[S13] \frac{\langle s_i, M, \Omega, t \rangle_i \mapsto \langle s'_i, M', \Omega', t' \rangle_i}{\langle (s_1, \dots, s_i, \dots, s_n), M, \Omega, t \rangle \mapsto \langle (s_1, \dots, s'_i, \dots, s_n), M', \Omega', t' \rangle} \\
[S14] \langle (m_1, \dots, m_n) := e, M, \Omega, t \rangle \mapsto \langle (m_1 := [e]_1, \dots, m_n := [e]_n), M, \Omega, t \rangle
\end{array}$$

Figure 12: The operational semantics of DSR\*

**Lemma 6.2** (Projection ii). Suppose  $[M]_i = M_i$  and  $[\Omega]_i = \Omega_i$  and  $[t]_i = t_i$ . Then  $\langle s, M, \Omega, t \rangle_i \mapsto \langle s', M', \Omega', t' \rangle_i$  if and only if  $\langle s, M_i, \Omega_i, t_i \rangle \mapsto \langle s', M'_i, \Omega'_i, t'_i \rangle$  and  $[M']_i = M'_i$  and  $[\Omega']_i = \Omega'_i$  and  $[t']_i = t'_i$

*Proof.* By induction on the structure of  $s$ . □

**Lemma 6.3** (Expression adequacy). Suppose  $\langle e_i, M_i \rangle \Downarrow v_i$  for  $i \in \{1, \dots, n\}$ , and there exists a DSR\* configuration  $\langle e, M \rangle$  such that  $[e]_i = e_i$  and  $[M]_i = M_i$ . Then  $\langle e, M \rangle \Downarrow v$  such that  $[v]_i = v_i$ .

*Proof.* By induction on the structure of  $e$ . □

**Definition 6.1** (Local run). A local run  $\langle s, M, \Omega, t \rangle \mapsto^* \langle s', M', \Omega', t' \rangle$  represents consecutive local evaluation steps:  $\langle s, M, \Omega, t \rangle \mapsto \langle s_1, M_1, \Omega_1, t_1 \rangle, \langle s_1, M'_1, \Omega_1, t_1 \rangle \mapsto \langle s_2, M_2, \Omega_2, t_2 \rangle, \dots, \langle s_n, M_n, \Omega_n, t_n \rangle \mapsto \langle s', M', \Omega', t' \rangle$ , where  $M'_i$  and  $M_i$  may differ because the execution of other threads or active attacks may change the local memory snapshot.

**Lemma 6.4** (One-step adequacy). Suppose  $E_i = \langle s_i, M_i, \Omega_i, t_i \rangle \mapsto \langle s'_i, M'_i, \Omega'_i, t'_i \rangle$  for  $i \in \{1, \dots, n\}$ , and there exists a DSR\* configuration  $\langle s, M, \Omega, t \rangle$  such that  $[\langle s, M, \Omega, t \rangle]_i = \langle s_i, M_i, \Omega_i, t_i \rangle$ . Then there exists  $E = \langle s, M, \Omega, t \rangle \mapsto^* \langle s', M', \Omega', t' \rangle$  such that for any  $i$ ,  $[E]_i \preceq E_i$ , and for some  $j$ ,  $[E]_j \approx E_j$ .

*Proof.* By induction on the structure of  $s$ .

- $s$  is skip. Then  $s_i$  is also skip and cannot be further evaluated. Therefore, the lemma is correct 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  $\langle \lfloor v \rfloor_i := \lfloor e \rfloor_i, M_i, \Omega_i, t_i \rangle \mapsto \langle \text{skip}, M_i[m_i \mapsto_{t_i} v_i], \Omega_i, t_i + 1 \rangle$  where  $m_i = \lfloor v \rfloor_i$  and  $\langle \lfloor e \rfloor_i, M_i \rangle \Downarrow v_i$ . By Lemma 6.3,  $\langle e, M \rangle \Downarrow v'$  and  $\lfloor v' \rfloor_i = v_i$ . If  $v = m$ , then  $\langle v := e, M, \Omega, t \rangle \mapsto \langle \text{skip}, M[m \mapsto_t v'], \Omega, t + 1 \rangle$ . Since  $\lfloor M \rfloor_i = M_i$ , we have  $\lfloor M[m \mapsto_t v] \rfloor_i = M_i[m \mapsto \lfloor v \rfloor_i]$ . In addition, we have  $\lfloor s' \rfloor_i = s'_i = \text{skip}$ . If  $v = (m_1, \dots, m_n)$ , then  $v = (m_1, \dots, m_n)$ , and  $\langle v := e, M, \Omega, t \rangle \mapsto \langle (m_1 := v_1, \dots, m_n := v_n), M, \Omega, t \rangle \mapsto \langle (\text{skip}, \dots, m_n := v_n), M[m_1 \mapsto_1 v_1 @ [t]_1], \Omega, t + 1 \rangle$ .
- $s$  is  $\text{if } e \text{ then } s'_1 \text{ else } s'_2$ . By Lemma 6.3,  $\langle e, M \rangle \Downarrow v$ . If  $v = n$ ,  $\langle s, M \rangle \mapsto \langle s'_j, M \rangle$  for some  $j$  in  $\{1, 2\}$ , and  $s_i$  is  $\text{if } \lfloor e \rfloor_i \text{ then } \lfloor s'_1 \rfloor_i \text{ else } \lfloor s'_2 \rfloor_i$ . Therefore,  $\langle s_i, M_i \rangle \mapsto \langle \lfloor s'_j \rfloor_i, M_i \rangle$  holds because  $\langle \lfloor e \rfloor_i, \lfloor M \rfloor_i \rangle \Downarrow n$ . If  $v = (n_1, \dots, n_k)$ , then  $\langle s, M, \Omega, t \rangle \mapsto^* \langle (\lfloor s'_{j_1} \rfloor_1, \dots, \lfloor s'_{j_k} \rfloor_k), M, \Omega, t \rangle$  where  $j_1, \dots, j_k \in \{1, 2\}$ . In addition, we have  $\langle e, M_i \rangle \Downarrow n_i$ . Therefore,  $\langle s_i, M_i, \Omega_i, t_i \rangle \mapsto \langle s''_{j_i}, M_i, \Omega_i, t_i \triangleleft 1 \rangle$ .
- $s$  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  $\text{skip}$  or  $\text{up}$  for any  $i$ , then the lemma holds by induction.
  - $s''_1$  is  $\text{skip}$  or  $(\text{skip}, \dots, \text{skip})$ , then it is clear  $\langle s, M, \Omega, t \rangle \mapsto^* \langle s''_2, M, \Omega, t \rangle$ . Correspondingly,  $\langle s_i, M_i, \Omega_i, t_i \rangle \mapsto \langle \lfloor s''_2 \rfloor_i, M_i, \Omega_i, t_i \rangle$ .
  - $s''_1$  is  $\text{up}$  or  $(\text{up}, \dots, \text{up})$ , then  $\langle s, M, \Omega, t \rangle \mapsto^* \langle s''_2, M, \Omega, t + 1 \rangle$ , while  $\langle s_i, M_i, \Omega_i, t_i \rangle \mapsto \langle \lfloor s''_2 \rfloor_i, M_i, \Omega_i, t_i + 1 \rangle$ . In addition,  $\lfloor t + 1 \rfloor_i = \lfloor t \rfloor_i + 1 = t_i + 1$ .
  - $s''_1$  is  $(s_{11}, \dots, s_{1n})$ , and some  $s_{1j}$  is not  $\text{skip}$  or  $\text{up}$ . Then  $\langle s_{1j}, M, \Omega, t \rangle_j \mapsto \langle s'_{1j}, M', \Omega', t' \rangle_j$  and  $\langle s, M, \Omega, t \rangle \mapsto \langle (s_{11}, \dots, s'_{1j}, \dots, s_{1n}), s''_2, M', \Omega', t' \rangle$ . By Lemma 6.2,  $\langle s_{1j}, M_j, \Omega_j, t_j \rangle \mapsto \langle s'_{1j}, M'_j, \Omega'_j, t'_j \rangle$ .
- $s$  is  $\text{exec}(v, v', pc, \mathcal{Q}, \bar{e})$ . Then  $\langle s, M, \Omega, t \rangle \mapsto \langle \text{skip}, M, \Omega \cup \{\mu\}, t + 1 \rangle$  while  $\mu = [v, v', \text{exec} :: t, pc, \mathcal{Q}, \bar{v}_1]$  and  $\langle \bar{e}, M \rangle \Downarrow \bar{v}_1$ . Since  $\lfloor s \rfloor_i = s_i$ , it is easy to show that  $\lfloor \mu \rfloor_i = \mu_i$  and  $\langle s_i, M_i, \Omega_i, t_i \rangle \mapsto \langle \text{skip}, M_i, \Omega_i \cup \{\mu_i\}, t_i + 1 \rangle$ , since by Lemma 6.3,  $\langle \bar{e}, M \rangle \Downarrow \bar{v}_1$  implies  $\langle \lfloor \bar{e} \rfloor_i, M_i \rangle \Downarrow \lfloor \bar{v}_1 \rfloor_i$ .
- $s$  is  $\text{chmod}(v_1, v_2, pc, \mathcal{Q}, \ell)$  or  $\text{setvar}(v_1, v_2, v)$ . By the same argument of the above case.
- $s$  is  $(s_1, \dots, s_n)$ . By rule (S13) and Lemma 6.2.

□

**Lemma 6.5 (Adequacy).** Suppose  $E_i = \langle s_i, M_i, \Omega_i, t_i \rangle \mapsto^* \langle s'_i, M'_i, \Omega'_i, t'_i \rangle$  for all  $i$  in  $\{1, \dots, n\}$ , and there exists a DSR\* configuration  $\langle s, M, \Omega, t \rangle$  such that  $\lfloor \langle s, M, \Omega, t \rangle \rfloor_i = \langle s_i, M_i, \Omega_i, t_i \rangle$ . Then there exists  $E = \langle s, M, \Omega, t \rangle \mapsto^* \langle s', M', \Omega', t' \rangle$ , such that for any  $i$ ,  $\lfloor E \rfloor_i \preceq E_i$ , and there exists  $j$  such that  $\lfloor E \rfloor_j \approx E_j$ .

*Proof.* By induction on the number of steps of  $E_1$  through  $E_n$ . If  $\langle s_j, M_j, \Omega_j, t_j \rangle = \langle s'_j, M'_j, \Omega'_j, t'_j \rangle$  for some  $j$ , then the lemma holds immediately. Otherwise, for all  $i$ ,  $\langle s_i, M_i, \Omega_i, t_i \rangle \mapsto \langle s''_i, M''_i, \Omega''_i, t''_i \rangle \mapsto^* \langle s'_i, M', \Omega'_i, t'_i \rangle$ . By Lemma 6.4, there exists  $E' = \langle s, M, \Omega, t \rangle \mapsto^* \langle s'', M'', \Omega'', t'' \rangle$  such that  $\lfloor E \rfloor_i \preceq \langle s_i, M_i, \Omega_i, t_i \rangle \mapsto \langle s''_i, M''_i, \Omega''_i, t''_i \rangle$ , and for some  $j$ ,  $\lfloor E \rfloor_j \approx \langle s_j, M_j, \Omega_j, t_j \rangle \mapsto \langle s'_j, M'_j, \Omega'_j, t'_j \rangle$ . Let  $E''_i = E_i - \lfloor E' \rfloor_i$ . By induction, there exists  $E'' = \langle s'', M'', \Omega'', t'' \rangle \mapsto^* \langle s', M', \Omega', t' \rangle$  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. □

### 6.3.2 Typing rules

The type system of DSR\* is shown in Figure 13. The typing judgment has the forms  $\Gamma; P \vdash e : \tau$  and  $\Gamma; P; \mathcal{Q}; pc \vdash s : \tau$ , where  $\mathcal{Q}$  is the quorum system on which  $e$  or  $s$  is replicated. Except for additional rules for bracket terms, this type system is similar to the type system of DSR.

$$\begin{array}{c}
\text{[BV1]} \quad \frac{\Gamma \vdash v_i : \tau \quad \neg\zeta(\tau) \text{ or } \forall i. v_i = v \vee v_i = \text{none}}{\Gamma \vdash (v_1, \dots, v_n) : \tau} \quad \text{[BV2]} \quad \frac{\Gamma \vdash v_i : \tau \quad \tau = \sigma @ \mathcal{Q} \quad K(v_1, \dots, v_n)}{\Gamma \vdash (v_1, \dots, v_n) : \tau} \quad \text{[BS]} \quad \frac{[\Gamma]_i ; P ; \mathcal{Q} ; [pc']_i \vdash s_i : [\tau]_i \quad \neg\zeta(pc')}{\Gamma ; P ; \mathcal{Q} ; pc \vdash (s_1, \dots, s_n) : \tau} \\
\text{[M-EXEC]} \quad \frac{\Gamma ; P \vdash c[\bar{v}] : \text{reactor}\{pc', \bar{\pi} \triangleright z : \bar{\tau}_1, \bar{\tau}_2\} \quad \vdash \bar{v}_i : \bar{\tau}_i \quad i \in \{1, \dots, n\} \Rightarrow \neg\zeta(pc)}{\Gamma ; P \vdash [\text{exec} \langle c[\bar{v}], \eta \rangle :: pc, \bar{v}_1, \mathcal{Q}, t]_i} \quad \text{[M-CHMD]} \quad \frac{\Gamma ; P \vdash c[\bar{v}] : \text{reactor}\{pc', \bar{\pi} \triangleright z : \bar{\tau}_1, \bar{\tau}_2\} \quad \vdash \ell : \text{label}_{\ell'} \quad \neg\zeta(\ell') \quad i \in \{1, \dots, n\} \Rightarrow \neg\zeta(pc)}{\Gamma ; P \vdash [\text{chmod} \langle c[\bar{v}], \eta \rangle :: pc, \ell, \mathcal{Q}, t]_i} \\
\text{[M-SETV]} \quad \frac{\vdash c[\bar{v}].\eta_1 : \tau \text{ var} \quad \vdash v_1 : \tau \quad i \in \{1, \dots, n\} \Rightarrow \neg\zeta(pc)}{\Gamma ; P \vdash [\text{setvar} \langle c[\bar{v}].z, \eta \rangle :: v_1, t]_i}
\end{array}$$

Figure 13: Typing rules of DSR\*

The bracket constructs captures the differences between DSR terms. As a result, any effect of a bracket construct does not satisfy  $\zeta$ . Let  $\neg\zeta(x)$  denote that  $x$  does not satisfy  $\zeta$ . 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 equivalent as none is equivalent to any value. An additional rule (BV2) may be used to check a bracket value with a located type  $\sigma @ \mathcal{Q}$ . In this case, the components of the bracket value are versioned values, which are treated differently because values with different timestamps may be equivalent. Rule (BV2) relies on an abstract function  $K(v_1, \dots, v_n)$  to determine whether a bracket of versioned values can have a type satisfying  $\zeta$ .

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

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

An important constraint that  $\zeta$  needs to satisfy is that  $\neg\zeta(\ell)$  implies  $\neg\zeta(\ell \sqcup \ell')$  for any  $\ell'$ . The purpose of this constraint is best illustrated by an example. In DSR\*, if expression  $e$  is evaluated to a bracket value  $(v_1, \dots, v_n)$ , statement **if**  $e$  **then**  $s_1$  **else**  $s_2$  would be reduced to a bracket statement  $(s'_1, \dots, s'_n)$ , where  $s'_i$  is either  $[s_1]_i$  or  $[s_2]_i$ . To show  $(s'_1, \dots, s'_n)$  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  $\zeta$ . Suppose  $e$  has type  $\text{int}_\ell$ , 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  $\neg\zeta(pc \sqcup \ell)$ .

### 6.3.3 Subjection reduction

In this section, we prove the subject reduction theorem of DSR\*.

**Lemma 6.6.** Suppose  $\Gamma ; P \vdash e : \tau$ , and  $\Gamma \vdash M$ , and  $\langle e, M \rangle_i \Downarrow v$ . Then  $\Gamma ; P \vdash v : \tau$ .

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

- **Cases E1.** Since  $\Gamma \vdash M$ , we have  $\Gamma \vdash M(m) : \tau$ . By rules (BV1) and (BV2),  $\vdash [M(m)]_i : \tau$ .
- **Case E2.** By induction,  $\Gamma ; \mathcal{Q} \vdash v_i : \tau$  for  $i \in \{1, 2\}$ . Therefore,  $\Gamma ; \mathcal{Q} \vdash v : \tau$ .

□

**Theorem 6.1** (Subject reduction). Suppose  $\Gamma; P; Q; pc \vdash s : \tau$ , and  $\Gamma \vdash M$ , and  $\Gamma; P \vdash \Omega$ , and  $\langle s, M, \Omega, t \rangle_i \mapsto \langle s', M', \Omega', t' \rangle_i$ , and  $i \in \{1, \dots, n\}$  implies that  $\neg\zeta(pc)$ . Then  $\Gamma; P; Q; pc \vdash s' : \tau$ , and  $\Gamma \vdash M'$ , and  $\Gamma; P \vdash \Omega'$ .

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

- **Case S1.** In this case,  $s$  is  $m := e$ ;  $\tau$  is  $\text{stmt}_{pc}$ ;  $s'$  is  $\text{skip}$ . We have  $\Gamma; P; Q; pc \vdash \text{skip} : \text{stmt}_{pc}$ . By (S1),  $M'$  is  $M[m \mapsto_i v@t]$ . By Lemma 6.6, we have  $\Gamma \vdash v : \Gamma(m)$ . If  $i$  is  $\bullet$ , then  $M'(m)$  is  $v$  or  $v@t$  according to  $\Gamma(m)$ , and in either case, the type of  $M'(m)$  is  $\Gamma(m)$ . Otherwise,  $\neg\zeta(\Gamma(m))$  holds, and thus  $M'(m)$  has type  $\Gamma(m)$  according to rule (BV1).
- **Case S2.** By typing rule (SEQ),  $\Gamma; P; Q; pc \vdash s_1 : \text{stmt}_{pc'}$  and  $\Gamma; P; Q; pc' \vdash s_2 : \text{stmt}_{pc''}$ . By induction,  $\Gamma; P; Q; pc \vdash s'_1 : \text{stmt}_{pc'}$ . Therefore,  $\Gamma; P; Q; pc \vdash s'_1; s_2 : \text{stmt}_{pc''}$ . By induction,  $\Gamma \vdash M'$  and  $\Gamma; P \vdash \Omega'$ .
- **Case S3.**  $s$  is  $\text{skip}$ ;  $s'$ . By rule (SEQ),  $\Gamma; P; Q; pc \vdash s' : \tau$ .
- **Case S5.**  $s$  is  $\text{if } e \text{ then } s_1 \text{ else } s_2$ . By typing rule (IF),  $\Gamma; P; Q; pc \sqcup \ell_e \vdash s_1 : \tau$ , which implies  $\Gamma; P; Q; pc \vdash s_1 : \tau$ .
- **Case S6.** By the same argument as case (S5).
- **Case S7.** In this case,  $s$  is  $\text{exec}(c[\bar{v}], \eta, pc, Q, \bar{e})$ . By Lemma 6.6,  $\Gamma; Q \vdash \bar{v}_1 : \bar{\tau}_1$ , where  $\bar{\tau}_1$  are the types of the corresponding arguments of  $c[\bar{v}]$ . Thus  $\Gamma \vdash [\text{exec}(c[\bar{v}], \eta) :: pc, \bar{v}_1, Q, t]$ .
- **Case S8.** By the same argument as case (S7).
- **Case S9.** By Lemma 6.6.
- **Case S10.** By (BS),  $\tau$  is  $\text{stmt}_{pc}$ . Therefore,  $\Gamma; P; Q; pc \vdash \text{skip} : \tau$ .
- **Case S12.** In this case,  $s$  is  $\text{if } e \text{ then } s_1 \text{ else } s_2$  and  $\langle e, M \rangle \Downarrow (v_1, \dots, v_n)$ . By the typing rule (IF),  $\Gamma; Q \vdash e : \text{int}_\ell$ . By Lemma 6.6,  $\Gamma; Q \vdash (v_1, \dots, v_n) : \text{int}_\ell$ . By the typing rule (BV1), we have  $\neg\zeta(\ell)$ , which implies  $\neg\zeta(pc \sqcup \ell)$ . Moreover, by rule (IF),  $\Gamma; Q; pc \sqcup \ell \vdash [s_j]_i : \tau$  for  $i \in \{1, \dots, n\}$  and  $j \in \{1, 2\}$ . Therefore, by rule (BS),  $\Gamma; Q; pc \vdash s' : \tau$ .
- **Case S13.** By induction,  $\Gamma \vdash M'$  and  $\Gamma; P \vdash \Omega'$ , and  $\Gamma; P; Q; pc' \vdash s'_i : \tau$ . Therefore,  $\Gamma; P; Q; pc \vdash s' : \tau$ .
- **Case S14.**  $s'$  is  $(m_1 := [e]_1, \dots, m_n := [e]_n)$ . Suppose  $\Gamma; P \vdash (m_1, \dots, m_n) : (\text{int}_\ell \text{ ref})_{\ell'}$ . By (BV1),  $\neg\zeta(\ell')$ , which implies  $\neg\zeta(\ell)$ . As a result,  $\Gamma; P; Q; \ell \vdash s' : \tau$ .

□

## 6.4 Noninterference proof

Let  $\Theta_0$  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 6.7** (Noninterference). Suppose  $\Gamma \Vdash P$ , and  $E_i = \langle \Theta_0, \mathcal{M}_i, \mathcal{E}_0 \rangle \mapsto^* \langle \Theta'_i, \mathcal{M}'_i, \mathcal{E}'_i \rangle$  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  $|E_1| + |E_2|$ . The base case is trivial. Without loss of generality, suppose  $|E_1| \leq |E_2|$  and  $\langle \Theta, \mathcal{M}_i, \mathcal{E} \rangle \mapsto^* \langle \Theta''_i, \mathcal{M}''_i, \mathcal{E}''_i \rangle \mapsto \langle \Theta'_i, \mathcal{M}'_i, \mathcal{E}'_i \rangle$  for  $i \in \{1, 2\}$ . Let  $T''_i = \text{timestamps}(\Theta''_i)$  and  $T'_i = \text{timestamps}(\Theta'_i)$ . By induction,  $\Gamma; P \vdash \langle \Theta''_i, \mathcal{M}''_i, \mathcal{E}''_i \rangle \approx_\zeta \langle \Theta'_i, \mathcal{M}'_i, \mathcal{E}'_i \rangle$ . 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  $\langle \Theta''_i, \mathcal{M}''_i, \mathcal{E}''_i \rangle \mapsto \langle \Theta'_i, \mathcal{M}'_i, \mathcal{E}'_i \rangle$ :

- **Case (G1).** In this case, the evaluation step is derived from  $\langle s, M_1'', \Omega_1'', t_1'' \rangle \mapsto \langle s', M_1', \Omega_1', t_1' \rangle$  on some host  $h_1$ . We need to prove that the local state of  $h_1$  in  $\langle \Theta_1', \mathcal{M}_1', \mathcal{E}_1' \rangle$  is still  $\zeta$ -equivalent to 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. Consider the local run of the thread at  $t_1'$  on host  $h_i$ :  $E_i' = \langle s_i, M_i, \emptyset, t \rangle \mapsto^* \langle s_i', M_i', \Omega_i', t_i' \rangle$  for  $i \in \{1, 2\}$ . By rule (TPE), the two local runs correspond to the same closure reference  $\langle c[\bar{v}], \eta \rangle$ . Then we can show that  $s_i = s[\mathcal{A}_i']$  and  $\Gamma' \vdash \mathcal{A}_1' \approx_\zeta \mathcal{A}_2'$ , where  $\Gamma'$  is the local typing assignment for reactor  $c[\bar{v}]$ . By rule (M1), we have  $\mathcal{A}_i' = \mathcal{A}_i[\bar{y} \mapsto \bar{v}_i][cid \mapsto \eta][nid \mapsto hash(t)]$ , where  $\mathcal{A}_i$  is the variable record in the corresponding closure,  $\bar{v}_i$  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  $\zeta$  condition, then the program counter labels of the corresponding invocation messages also satisfies  $\zeta$ . Since  $P$  satisfies (RV3), the invocation messages are sent by threads of 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  $\zeta$ -equivalent, which implies that the arguments are  $\zeta$ -equivalent 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  $dom(\Gamma)$ ,  $\zeta(\Gamma(m))$  implies  $M_1(m) \approx_\zeta M_2(m)$ . 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  $M_j[m] = \text{none}$  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  $\langle s, M, \emptyset, t \rangle$  such that  $\lfloor s \rfloor_i = s_i$  and  $\lfloor M \rfloor_i = M_i$ , and  $s$  and  $M$  are well-typed with respect to the following  $K$  condition:  $K(v_1 @ t_1, \dots, v_n @ t_n)$  is true if for any  $i, j$ ,  $t_i = t_j$  implies  $v_i \approx v_j$ . By Lemma 6.5, there exists  $E' = \langle s, M, \emptyset, t \rangle \mapsto^* \langle s', M', \Omega', t' \rangle$  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  $\lfloor M' \rfloor_2 = M_2''$  and  $\lfloor \Omega' \rfloor_2 = \Omega_2''$  and  $\lfloor t' \rfloor_2 = t_2''$ . By Theorem 6.1,  $M'$  and  $\Omega'$  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 equivalent after this execution step.

- **Case (M1).** In this case, the goal is to prove  $t \vdash \Theta_1' \approx_\zeta \Theta_2'$  where  $t = \min(\max(\mathcal{T}_1', \zeta), \max(\mathcal{T}_2', \zeta))$ . Suppose the newly created thread is  $\theta = \langle s, h, t_1, c[\bar{v}], \eta \rangle$ , and the program counter label of  $c[\bar{v}]$  is  $pc$ , and  $t_1' = \max(\mathcal{T}_1', \zeta)$ . If  $\neg\zeta(pc, h)$ , 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 that  $\zeta(pc, h)$  holds.

If  $t_1 < inc(t_1', pc)$ , then we need to prove that  $\theta$  is not the only thread at time  $t_1$ . Suppose otherwise. By  $t_1 < inc(t_1', pc)$ ,  $\theta$  is not invoked by the threads at  $t_1'$ . Let  $n$  be the number of  $\zeta$ -threads with timestamps having different global parts in  $\Theta_1'$ . Then  $n - 1$  different  $\zeta$ -threads need to invoke  $n$  different  $\zeta$ -threads. Therefore, threads at some time  $t_d$  needs to invoke two threads with different timestamps, which means that different invocation messages satisfying the  $\zeta$  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,  $\theta$  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,  $\theta$  is  $\zeta$ -equivalent 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  $t_1 = inc(t_1', pc)$ , by rule (M1), at least one quorum finishes executing the thread at  $t_1'$ . Suppose  $\langle \Theta_2'', \mathcal{M}_2'', \mathcal{E}_2'' \rangle \mapsto \langle \Theta_2', \mathcal{M}_2', \mathcal{E}_2' \rangle$ . Let  $t_2' = \text{timestamp}(\Theta_2'', \mathcal{E}_2'')$  and  $t_2 = \text{timestamp}(\Theta_2', \mathcal{E}_2')$ . If  $t_2 \leq t_1'$ , then we have  $t \vdash \Theta_1' \approx_\zeta \Theta_2'$  by  $t \vdash \Theta_1'' \approx_\zeta \Theta_2'$ . Similarly, if  $t_1 \leq t_2'$ , 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  $\Theta_2''$  has

its counterpart in  $\Theta_1''$  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  $\mathcal{E}_1' = \mathcal{E}_1''[\text{closure}(h_1, c[\bar{v}], \eta) \mapsto \langle c[\bar{v}], \eta, \ell, \mathcal{A}_1'[z \mapsto v], t', \text{on} \rangle]$ . Then we need to show that for any host  $h_2$  in  $\text{loc}(c[\bar{v}])$  such that  $\zeta(c[\bar{v}], h_2), P \vdash \mathcal{E}_1'.\text{closure}(h_1, c[\bar{v}], \eta) \approx_\zeta \mathcal{E}_2'.\text{closure}(h_2, c[\bar{v}], \eta)$ . 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  $\tau$  is the type of  $z$ .

First, we prove that the  $\zeta$ -messages used to initialize  $z$  have the same timestamp. Since  $P$  satisfies (RV1) and (RV2), the threads that first operate on  $\langle c[\bar{v}].z, \eta \rangle$  correspond to either  $\langle c', \eta' \rangle$ , or  $\langle c_1[\bar{v}_1], \eta_1 \rangle$  with  $\langle c[\bar{v}].z, \text{nid} \rangle$  appearing in its code. In either 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  $\zeta$ -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  $\langle c[\bar{v}].z, \eta \rangle$ . 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  $C(x) \leq l_A$ , then all the `setvar` message satisfy the  $\zeta$  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  $\langle c[\bar{v}].z, \eta \rangle$  are equal in both runs.

Suppose  $\zeta(x)$  is  $I(x) \not\leq l_A$ . Consider the message synthesizer  $\pi$  for  $z$ . There are two cases:

- $\pi$  is  $\text{LT}[I(\ell)]$ . The `setvar` messages have the form  $[\text{setvar } \langle c[\bar{v}].z, \eta \rangle :: v, t]$ , and  $z$  has type  $\text{int}_\ell$ . 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,  $\mathcal{A}_1(z) \approx \mathcal{A}_2(z)$ .
- $\pi$  is  $\text{QR}[\mathcal{Q}, I]$ . Suppose the set of high-integrity senders are  $h_1, \dots, h_n$  in  $E_1$  and  $h'_1, \dots, h'_k$  in  $E_2$ , and the local memory snapshots for these hosts when executing the thread at  $t$  are  $M_1, \dots, M_n$  and  $M'_1, \dots, M'_k$ , respectively. Let  $\mathbf{M}$  incorporate those local memories. By rule (TPE), we can show that  $\mathbf{M}$  is well-typed with respect to the following  $K$  constraint:

$$\frac{\forall i. v_i = v \vee v_i = \text{none}}{(v_1, \dots, v_n) \Downarrow v} \quad \frac{\exists v_j @ t_j. v_j @ t_j = v @ t \quad \forall i. t_i \leq t}{(v_1 @ t_1, \dots, v_n @ t_n) \Downarrow v} \quad \frac{(v_1, \dots, v_n) \Downarrow v \quad (v'_1, \dots, v'_k) \Downarrow v}{K(v_1, \dots, v_n, v'_1, \dots, v'_k)}$$

In addition, we can construct a  $\text{DSR}^*$  statement  $s$  such that  $[s]_i = s_i$  where  $1 \leq i \leq n + k$ . Then we have a well-typed  $\text{DSR}^*$  configuration  $\langle s, \mathbf{M}, \emptyset, t \rangle$ . By Lemma 6.5,  $\langle s, \mathbf{M}, \emptyset, t \rangle \mapsto^* \langle s', \mathbf{M}', \Omega', t' \rangle$  and  $[t']_i \leq t'_i$  and for some  $j$ ,  $[t']_j = t'_j$ . By Theorem 6.1,  $\Omega'$  is well-typed, and the message  $[\text{setvar } \langle c[\bar{v}].z, \eta \rangle :: v, t]$  in  $\Omega'$  is also well-typed, which means that  $v = (v_1, \dots, v_n, v'_1, \dots, v'_k)$  is well-typed. Furthermore,  $K(v_1, \dots, v_n, v'_1, \dots, v'_k)$  implies that the `setvar` messages produced by  $\text{QR}[\mathcal{Q}, I]$  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  $\zeta$ -equivalence between program states is not affected.

□

**Theorem 6.2** (Integrity Noninterference). Suppose  $\Gamma \Vdash P$ , and  $\langle \Theta_0, \mathcal{M}_i, \mathcal{E}_0 \rangle \mapsto^* \langle \Theta'_i, \mathcal{M}'_i, \mathcal{E}'_i \rangle$  for  $i \in \{1, 2\}$ . If  $\Gamma; P \vdash \langle \Theta_0, \mathcal{M}_1, \mathcal{E}_0 \rangle \approx_{I \not\leq l_A} \langle \Theta_0, \mathcal{M}_2, \mathcal{E}_0 \rangle$ , then  $\Gamma; P \vdash \langle \Theta'_1, \mathcal{M}'_1, \mathcal{E}'_1 \rangle \approx_{I \not\leq l_A} \langle \Theta'_2, \mathcal{M}'_2, \mathcal{E}'_2 \rangle$ .

*Proof.* Let  $\zeta(\ell)$  be  $I(\ell) \not\leq L$  and apply Lemma 6.7. □

**Theorem 6.3** (Confidentiality Noninterference). Suppose  $\Gamma \Vdash P$ , and  $\langle \Theta_0, \mathcal{M}_i, \mathcal{E}_0 \rangle \mapsto^* \langle \Theta'_i, \mathcal{M}'_i, \mathcal{E}'_i \rangle$  for  $i \in \{1, 2\}$  and  $\Gamma; P \vdash \langle \Theta_0, \mathcal{M}_1, \mathcal{E}_0 \rangle \approx_{C \leq l_A} \langle \Theta_0, \mathcal{M}_2, \mathcal{E}_0 \rangle$ . Then  $\Gamma; P \vdash \langle \Theta'_1, \mathcal{M}'_1, \mathcal{E}'_1 \rangle \approx_{C \leq l_A} \langle \Theta'_2, \mathcal{M}'_2, \mathcal{E}'_2 \rangle$ .

*Proof.* Let  $\zeta(\ell)$  be  $C(\ell) \leq L$  and apply Lemma 6.7. □

## 7 Security by construction

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

### 7.1 Secure distribution schemes

The Aimp–DSR translator takes as input a *distribution scheme*  $D$ , which specifies where the target code of source statements is replicated, and where memory references are replicated. To distinguish the same statement appearing in different places of the source, each non-sequence statement  $S$  in the source program is instrumented with a unique name  $c$ , and the instrumented statement is written as  $\{c\} S$ . The instrumentation does not affect the semantics of Aimp. A distribution scheme maps those statement names to host sets.

Formally, a distribution scheme maps statement names to host sets and maps memory references to quorum systems. If  $D(m) = Q$ , then  $m$  is replicated on  $Q$  in the target program. If  $D(c) = H$ , then the target code of  $\{c\} S$  (not including the target code of the substatements of  $S$ ) is replicated on  $H$ .

In general, many distribution schemes are possible. Because security is the concern here, we do not give an algorithm for generating distribution schemes, but instead focus on identifying security constraints sufficient to guarantee that a given distribution scheme is able to enforce the source program’s security policies.

First, we need to determine the security policies of the source. The security policies of memory references are given by the typing assignment  $\Gamma$  of the source program. The security policies of a statement  $S$  are represented by a label derived from the program counter label and the labels of data processed by  $S$ . Let  $C(S)$  and  $I(S)$  represent the confidentiality and integrity labels of  $S$ . Intuitively,  $C(S)$  is the join of the confidentiality labels of inputs of  $S$ ;  $I(S)$  is the join of the integrity labels of outputs of  $S$ . Furthermore, the termination of a `while` statement depends on the integrity of the statement. Therefore,  $A(\mathcal{R}) \leq I(S)$  needs to hold if  $S$  is a `while` statement with type `stmtR`. A statement label has two additional confidentiality components:  $C_{\text{end}}$ , the confidentiality label of the information that can be inferred by knowing the termination program point of  $S$ , and  $C_{\text{pc}}$ , the confidentiality component of the program counter label of  $S$ .

The rules for determining the label of a statement are shown in Figure 14. It is safe to assign a stronger security label to a statement. In practice, assigning a stronger integrity label to a statement makes it easier to generate control transfer code for that statement because of the extra integrity allows the hosts to perform more freely. A valid label assignment  $\Delta$  satisfies  $\Gamma; \mathcal{R}_c \vdash S' : \ell'$  and  $\ell' \leq \Delta(c)$ . for any statement  $\{c\} S'$  appearing in the source program  $S$ .

$$\begin{array}{l}
[L1] \frac{\Gamma; \mathcal{R} \vdash e : \text{int}_{\ell'} \quad \Gamma; \mathcal{R} \vdash m : \text{int}_{\ell} \text{ ref}}{\Gamma; \mathcal{R}; pc \vdash m := e : \{C = C(\ell'), I = I(\ell), A = A(\mathcal{R}), C_{pc} = C(pc), C_{\text{end}} = C(pc)\}} \\
[L2] \Gamma; \mathcal{R}; pc \vdash \text{skip} : \{C = \perp, I = \perp, A = A(\mathcal{R}), C_{pc} = C(pc), C_{\text{end}} = C(pc)\} \\
[L3] \frac{\Gamma; \mathcal{R} \vdash e : \text{int}_{\ell} \quad \Gamma; \mathcal{R}; pc \sqcup \ell \vdash S_i : \ell_i \quad i \in \{1, 2\}}{\ell' = \{C = C(\ell), I = I(\ell_1) \sqcup I(\ell_2), A = A(\mathcal{R}), C_{pc} = C(pc), C_{\text{end}} = C_{pc}(\ell_1) \sqcup C_{pc}(\ell_2)\}} \\
\Gamma; \mathcal{R}; pc \vdash \text{if } e \text{ then } S_1 \text{ else } S_2 : \ell' \\
[L4] \frac{\Gamma; \mathcal{R}; pc \vdash S_1 : \ell_1 \quad \Gamma; \mathcal{R}; pc \vdash S_2 : \ell_2}{\Gamma; \mathcal{R}; pc \vdash S_1; S_2 : \{C = C(\ell_1) \sqcup C(\ell_2), I = I(\ell_1) \sqcup I(\ell_2), A = A(\mathcal{R}), C_{pc} = C(pc), C_{\text{end}} = C_{pc}(\ell_2)\}} \\
[L5] \frac{\Gamma; \mathcal{R}; pc \vdash e : \text{int}_{\ell} \quad \Gamma; \mathcal{R}; pc \sqcup \ell \vdash S : \ell'}{\Gamma; \mathcal{R}; pc \vdash \text{while } e \text{ do } S : \{C = C(\ell), I = I(\ell') \sqcup A(\mathcal{R}), A = A(\mathcal{R}), C_{pc} = C(pc), C_{\text{end}} = C(pc)\}} \\
[L6] \frac{\Gamma; \mathcal{R}; pc \vdash S : \ell \quad \ell \leq \ell'}{\Gamma; \mathcal{R}; pc \vdash S : \ell}
\end{array}$$

Figure 14: Rules for inferring statement labels

We also impose an additional constraint on  $\Delta$  to help generate control transfer protocols. Suppose  $S_1$  and  $S_2$  are two non-sequence statements in the source program  $S$ , and  $S_2$  is a post dominator of  $S_1$  in the control flow graph of  $S$ . Let  $I_1 = I(\Delta(s_1))$  and  $I_2 = I(\Delta(s_2))$ . Moreover, for any post dominator  $S'$  of  $S_1$ , that  $S'$  dominates  $S_2$  implies  $I_1 \not\leq I(\text{label}(s'))$ . Then  $I_1 \leq I_2$  or  $I_1 \leq I_2$  is required. Otherwise, it is difficult to construct the protocol for transferring control from  $S_1$  to  $S_2$ . Intuitively, because  $I_1 \not\leq I_2$ , the target code of  $S_1$  needs to run a `chmod` statement to notify some following reactor at integrity level  $I_1$  to expect invocation requests of integrity level  $I_1 \sqcap I_2$ . However, after running the `chmod` statement, the integrity level of control flow is lowered to  $I_1 \sqcap I_2$ , which makes it difficult to invoke the target code of  $S_2$  because  $I_2 \not\leq I_1 \sqcap I_2$ .

A straightforward way to enforce this constraint is to insert a no-op statement  $S_0 = \text{skip}$  between  $S_1$  and  $S_2$  and instrument  $S_0$  with label  $\{C = C(pc), I = I_1 \sqcup I_2, A = A(\text{label}(s_2))\}$ , where  $pc$  is the program-counter label of this program point, if  $I_1$  and  $I_2$  are incomparable.

Let  $\mathcal{Q} \models \text{int}_{\ell} \text{ ref}$  denote that it is secure to store memory references with type  $\text{int}_{\ell} \text{ ref}$  on  $\mathcal{Q}$ , and  $D; \Delta; S \models \{c\} S'$  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 source program  $S$ , and the label assignment  $\Delta$ . The following rules can be used to infer these two kinds of judgments:

$$\begin{array}{l}
[DM] \frac{C(\ell) \leq C_{\sqcap}(\mathcal{Q}) \quad A(\ell) \leq A_{\text{write}}(\mathcal{Q}) \sqcap A_{\text{read}, I(\ell)}(\mathcal{Q})}{\mathcal{Q} \models \text{int}_{\ell} \text{ ref}} \\
[DS] \frac{\begin{array}{l} \{c_1\} S_1; \{c\} S' \in S \Rightarrow C_{\text{end}}(\Delta(c_1)) \leq C_{\sqcap}(H) \\ \Delta(c) = \ell \quad D(c) = H \quad C(\ell) \leq C_{\sqcap}(H) \end{array}}{A(\ell) \leq A(H, \text{LT}[\ell]) \quad \forall m \in \text{UM}(S'). C_{pc}(\ell) \leq C_{\sqcap}(D(m))} \\
D; \Delta; S \models \{c\} S'
\end{array}$$

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  $A(\ell)$ .

In rule (DS), the first premise ensures that  $H$  is allowed to know the program counter of the caller of  $c$ . In particular, if  $S'$  follows a conditional statement,  $H$  is allowed to know which branch is taken. The

premise  $C(\ell) \leq C_{\sqcap}(H)$  says that  $H$  is allowed to see the data needed for executing  $S'$ . The premise  $A(\ell) \leq A(H, \text{LT}[\ell])$  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 read operations. Let  $UM(S')$  be the set of references accessed by  $S'$ . Then for any  $m$  in  $UM(S')$ , on receiving a read request for  $m$ , hosts in  $D(m)$  may be able to infer that control reaches that program point of  $\{c\} S'$ . Thus, the constraint  $C_{pc}(\ell) \leq C_{\sqcap}(D(m))$  is imposed.

## 7.2 Remote Memory Accesses

Rule (DS) does not require a statement  $S$  to be distributed to the hosts where the references accessed by  $S$  are distributed. Therefore, the target code of  $S$  may need to access memory references on remote hosts. Accordingly, hosts storing a memory reference need to provide reactors to handle remote accesses. Using DSR, we can implement generic read and write reactors to handle remote memory reads and writes:

```

read[x:labelx, x1:labelx, x2:(intx1@&x2 ref)x1,
      x3:reactor{x}x, x4:intx, x5:(intx1⊔x@&x2 var)x]
{ x, #x2, λ(). setvar(x5, !x2); exec(x3, x4, x, #x2, ()) }

write[x:labelx, x1:(intx@&x1 ref)x, x2:reactor{x}x,
      x3:intx]
{ x, &x1, λy:intx. x1 := y; exec(x2, x3, x, &x1, ()) }

```

To achieve genericity, both read and write reactors carry several reactor parameters. The read reactor has six parameters:  $x$ , the program counter label of this reactor,  $x_2$  and  $x_1$ , the reference to be read and its label,  $x_3$  and  $x_4$ , specifying the continuation reactor instance  $\langle x_3, x_4 \rangle$  for returning control to, and  $x_5$ , the remote variable to receive the value of reference  $x_2$ . The read reactor should be invoked on the hosts holding replicas of reference  $x_1$ , and the reactor does not update any reference. Thus,  $\#x_1$  specifies where this reactor is located. The code of the reactor initializes the remote variable  $x_5$  with the value of  $!x_2$ , and then invokes  $\langle x_3, x_4 \rangle$ .

The write reactor has four parameters:  $x$ , the program counter label of this reactor,  $x_1$ , the reference to write to,  $x_2$  and  $x_3$ , specifying the continuation reactor instance  $\langle x_2, x_3 \rangle$ . This reactor has one argument  $y$ , which is the value to be assigned to  $x_1$ . The code of the reactor is self-explanatory. Since the write reactor updates  $x_1$ , it needs to be located on the quorum system  $\&x_1$ .

## 7.3 Translation rules

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. An entry reactor  $c$  at security level  $\ell$  is the reactor whose program counter label is  $\ell$  and there is no other reactor in  $P'$  preceding  $c$  with a program counter label  $pc$  satisfying  $pc \sqsubseteq \ell$ . Therefore, the translation context for  $S$  includes  $\Psi' = \&P' = \psi_1, \dots, \psi_n$ , the list of entries (entry reactors) of  $P'$ , where  $\psi_i$  has the form  $\langle c_i, w_i \rangle$ , where  $w_i$  is a variable whose value is a context identifier with which  $c_i$  invokes its following computation. In most cases,  $w_i$  is `cid`, and thus  $c_i$  is used as an abbreviation for  $\langle c_i, \text{cid} \rangle$ . A well-formed entry list  $\Psi'$  satisfies the following condition: if  $\text{label}(\psi_{i+1}) \sqsubseteq \text{label}(\psi_i)$  holds for any  $i \in \{1, \dots, n\}$ , where  $\text{label}(\langle c, w \rangle) = \text{label}(c)$ . In  $P'$ ,  $\psi_1$  through  $\psi_n$  are to be invoked in order, and for any reactor  $c''$  to be invoked between  $\psi_i$  and  $\psi_{i+1}$ , the constraint  $\text{label}(\psi_i) \sqsubseteq \text{label}(c'')$  is satisfied so that  $\psi_i$  has sufficient integrity to handle the invocation of  $c''$  on its own. Formally, the translation of  $S$  is denoted by  $\llbracket S \rrbracket \Psi' = \langle P, \Psi \rangle$ , where  $\Psi$  is the entries of  $P \cup P'$ .

$$\begin{array}{c}
\text{[TE1]} \quad \llbracket \eta \rrbracket \langle c, c', c_u, \ell, \mathcal{Q} \rangle = \eta \quad \text{[TE2]} \quad \llbracket m \rrbracket \langle c, c', c_u, \ell, \mathcal{Q} \rangle = m \quad \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) = \text{int}_{\ell_1} @ \mathcal{Q}_m \quad r = c\{\ell, \mathcal{Q}, \lambda.\text{exec}(\text{read}[\ell, \ell_1, m, c', \text{cid}, \langle c_u.z, \text{cid} \rangle], \text{nid}, \ell, \mathcal{Q}, ())\}}{\llbracket !m \rrbracket \langle c, c', c_u, \ell, \mathcal{Q} \rangle = \langle \{r\}, \lambda(\text{QR}[\mathcal{Q}_m, I(\ell_1)] \triangleright z : \text{int}_{\ell_1}). z \rangle} \\
\text{[TE5]} \quad \frac{\llbracket e_1 \rrbracket \langle c, c_1, c_u, \ell, \mathcal{Q} \rangle = \langle P_1, \lambda \overline{\pi_1} \triangleright \overline{z_1} : \overline{\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 \overline{z_2} : \overline{\tau_2}. e'_2 \rangle}{c_1 = (\text{if } P_2 \neq \emptyset \text{ then } \text{new-reactor}(P_1, c) \text{ else } c')} \\
\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 \overline{z_1} : \overline{\tau_1}, \overline{\pi_2} \triangleright \overline{z_2} : \overline{\tau_2}. e'_1 + e'_2 \rangle \\
\text{[TC1]} \quad \frac{\Psi = \{\psi_1, \dots, \psi_n\} \quad \ell_i = \text{label}(c_i) \quad i \in \{1, \dots, n\} \quad \ell_0 = \top \quad \ell_{n+1} = \perp \quad \ell_{j+1} \sqsubseteq \text{label}(c) \sqsubseteq \ell_j}{w_{j+1} = w \quad \llbracket \langle c, w_{i+1} \rangle \rrbracket \langle \ell_i, \psi_{i+1} \rangle = \langle s_i, w_i \rangle \quad i \in \{0, \dots, j\}} \\
\llbracket \langle c, w \rangle \rrbracket \Psi = \langle s_j; \dots; s_0, \{c, w\}, \psi_{j+1}, \dots, \psi_n \rangle \\
\text{[TC2]} \quad \frac{\Delta; D \vdash c : \langle \ell, \mathcal{Q} \rangle \quad s = (w' = c''.z) ? \text{setvar}(\langle c''.z, \text{nid} \rangle, w) : \text{skip} \quad w'' = (w' = c''.z) ? w : \text{nid}}{\ell' = \text{label}(c) \sqcup \text{label}(c') \quad s' = (\ell = \top) ? \text{exec}(c', w'', \ell', \mathcal{Q}, ()) : \text{chmod}(c', w'', \ell', \mathcal{Q}, \ell)} \\
\llbracket \langle c, w \rangle \rrbracket \langle \ell, \langle c', w' \rangle \rangle = \langle s; s', w'' \rangle \\
\text{[TS1]} \quad \frac{\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 \overline{z} : \overline{\tau}. e' \rangle \quad c_1 = \text{new-reactor}(P_e, c)}{r_1 = c_1\{\ell, \mathcal{Q}, \overline{\pi} \triangleright \overline{z} : \overline{\tau}, \lambda.\text{exec}(\text{write}[\ell, m, c_2, \text{cid}], \text{nid}, \ell, \mathcal{Q}, e')\} \quad \llbracket c \rrbracket \Psi = \langle s', \Psi' \rangle \quad r_2 = c_2\{\ell, \mathcal{Q}, \lambda.s'\}} \\
\llbracket \{c\} m := e \rrbracket \Psi = \langle P_e \cup \{r_1, r_2\}, \Psi' \rangle \\
\text{[TS2]} \quad \frac{\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 \overline{z} : \overline{\tau}. e' \rangle \quad c_1 = \text{new-reactor}(P_e, c)}{\llbracket c \rrbracket \Psi = \langle s', \Psi' \rangle \quad r_1 = c_1\{\ell, \mathcal{Q}, \overline{\pi} \triangleright \overline{z} : \overline{\tau}, \lambda. m := e'; s'\}} \\
\llbracket \{c\} m := e \rrbracket \Psi = \langle P_e \cup \{r_1\}, \Psi' \rangle \\
\text{[TS3]} \quad \frac{\Delta; D \vdash c : \langle \ell, \mathcal{Q} \rangle \quad \llbracket c \rrbracket \Psi = \langle s, \Psi' \rangle \quad r = c\{\ell, \mathcal{Q}, \lambda.s\}}{\llbracket \{c\} \text{skip} \rrbracket \Psi = \langle \{r\}, \Psi' \rangle} \quad \text{[TS4]} \quad \frac{\llbracket S_2 \rrbracket \Psi = \langle P_2, \Psi_2 \rangle \quad \llbracket S_1 \rrbracket \Psi_2 = \langle P_1, \Psi_1 \rangle}{\llbracket S_1; S_2 \rrbracket \Psi = \langle P_1 \cup P_2, \Psi_1 \rangle} \\
\text{[TS5]} \quad \frac{\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 \overline{z} : \overline{\tau}. e' \rangle \quad c_1 = \text{new-reactor}(P_e, c)}{\llbracket S_i \rrbracket \Psi = \langle P_i, \Psi_i \rangle \quad \llbracket c \rrbracket \Psi_i = \langle s'_i, \Psi'_i \rangle \quad i \in \{1, 2\} \quad r_1 = c_1\{\ell, \mathcal{Q}, \overline{\pi} \triangleright \overline{z} : \overline{\tau}, \lambda. \text{if } e' \text{ then } s'_1 \text{ else } s'_2\}} \\
\llbracket \{c\} \text{if } e \text{ then } S_1 \text{ else } S_2 \rrbracket \Psi = \langle P_e \cup P_1 \cup P_2 \cup \{r_1\}, \Psi' \rangle \\
\text{[TS6]} \quad \frac{\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 \overline{z} : \overline{\tau}. e' \rangle \quad \llbracket S \rrbracket c = \langle P, \Psi_1 \rangle}{c_1 = \text{new-reactor}(P_e, c) \quad \llbracket \langle c_1, \text{nid} \rangle \rrbracket \Psi_1 = \langle s_1, \langle c_1, \text{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 \overline{z} : \overline{\tau}, \text{LT}[\ell] \triangleright z' : \text{int}_{\ell}, \lambda. \text{if } e' \text{ then } \text{setvar}(\langle c_1.z', \text{nid} \rangle, z'); s_1 \text{ else } s_2\} \\
\llbracket \{c\} \text{while } e \text{ do } S \rrbracket \Psi = \langle P_e \cup P \cup \{r_1\}, \Psi' \rangle
\end{array}$$

Figure 15: Aimp–DSR Translation rules

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  $\lambda \overline{\pi} \triangleright \overline{z} : \overline{\tau}. e'$ , 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  $e'$ ,  $\ell$  is the program counter label for  $e$ , and  $\mathcal{Q}$  is the quorum system where  $P$  is replicated.

The syntax-directed translation rules are shown in Figure 15. 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 are with respect to a translation environment  $\langle \Gamma, \Delta, D \rangle$ ,

where  $\Gamma$  is the typing assignment for the target program,  $\Delta$  is the label assignment, and  $D$  is the distribution scheme. The typing assignment  $\Gamma = \llbracket \Gamma' \rrbracket D$  is derived from the typing assignment  $\Gamma'$  of the source program. For any  $m$  in  $\text{dom}(\Gamma')$ , suppose  $D(m) = \mathcal{Q}_m$ . Then  $\Gamma(m)$  is  $\Gamma'(m)$  if  $\mathcal{Q}_m$  contains only one host, and  $\Gamma'(m) @ \mathcal{Q}_m$  otherwise. Notation  $\Delta ; D \vdash c : \langle \ell, \mathcal{Q} \rangle$  means that  $\ell$  and  $\mathcal{Q}$  are the program counter label and the location of reactor  $c$ . Formally,  $D(c) = \mathcal{Q}$ , and  $\ell = \{C = C_{\text{pc}}(\ell'), I = I(\ell'), A = A(\ell')\}$ , where  $\ell' = \Delta(c)$ . The rules use a function  $\text{new-reactor}(P, c)$ , which is a fresh reactor name unless  $P$  is empty, in which case it is  $c$ .

Rules (TE1)–(TE3) translate constants and dereferences of non-replicated references, which remain the same after translation. Rule (TE4) is used to translate  $!m$  when  $m$  is replicated on multiple hosts. The target code invokes  $\text{read}[\ell, \ell_1, m, c', \text{cid}, \langle c_u.z, \text{cid} \rangle]$ , which initializes  $\langle c_u.z, \text{cid} \rangle$  with the value of  $m$  and invokes  $\langle c', \text{cid} \rangle$ . Note that the read reactor is invoked with  $\text{nid}$  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  $\langle P_i, \lambda \bar{\pi}_i \triangleright \bar{z}_i : \bar{\tau}_i . e'_i \rangle$  for  $i \in \{1, 2\}$ . Then  $e_1 + e_2$  is translated into  $\langle P_1 \cup P_2, \lambda \bar{\pi}_1 \triangleright \bar{z}_1 : \bar{\tau}_1, \bar{\pi}_2 \triangleright \bar{z}_2 : \bar{\tau}_2 . e'_1 + e'_2 \rangle$ . 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 (if  $P_2 \neq \emptyset$  then  $\text{new-reactor}(P_1, c)$  else  $c'$ ).

Rules (TC1) and (TC2) generate the code for  $c$  to invoke  $\Psi$  with the context identifier  $w$ . It can be viewed as translating  $\langle c, w \rangle$  in the context  $\Psi$ . The translation result is a tuple  $\langle s, \Psi' \rangle$  where  $s$  is the control transfer code, and  $\Psi'$  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  $\Psi$ .

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

Let  $\llbracket \langle c, w_{i+1} \rangle \rrbracket \langle \ell_i, \psi_{i+1} \rangle = \langle s_i, w_i \rangle$  denote that  $s$  is the code to invoke  $\psi_{i+1}$  with  $w$  to the label  $\ell_i$ , and  $w''$  is the context identifier to be used after executing  $s_i$ . Therefore, the code to invoke  $\Psi$  is  $s_j; \dots; s_0$ .

Rule (TC2) is used to compute  $\llbracket \langle c, w \rangle \rrbracket \langle \ell, \langle c', w' \rangle \rangle$ . The translation depends on whether  $w'$  is some remote variable  $c''.z$  and whether  $\ell$  is  $\top$ . If  $w' = c''.z$ , then the translation includes a  $\text{setvar}$  statement to initialize  $\langle c''.z, \text{nid} \rangle$  with  $w$  so that  $c''$  can invoke the following computation with the context identifier  $w$ . Moreover, after executing the  $\text{setvar}$  statement,  $c$  needs to invoke the remaining entries with  $\text{nid}$ , and  $w''$  is set to  $\text{nid}$ . If  $\ell$  is  $\top$ , it means that  $\langle c', w' \rangle$  is to be invoked directly, and thus the translation includes an  $\text{exec}$  statement to invoke  $c'$ . Otherwise, the translation uses a  $\text{chmod}$  statement to invoke  $c'$  to label  $\ell$ .

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  $\text{write}$  reactors on  $\mathcal{Q}$ . The reactor  $\text{write}[\ell, m, c_2, \text{cid}]$  updates  $m$  and then invokes  $\langle c_2, \text{cid} \rangle$ . The reactor  $c_2$  contains the code to invoke  $\Psi$  with  $\text{cid}$ . The value of  $e$  is computed by  $P_e$  and  $\lambda \bar{\pi} \triangleright \bar{z} : \bar{\tau} . e'$ . Reactor  $c$  is the entry of  $P_e$ . Reactor  $c_1$  computes  $e'$  and issues the  $\text{write}$  requests. Thus,  $c_1$  contains  $\bar{\pi} \triangleright \bar{z} : \bar{\tau}$  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  $\Psi$ .

Rule (TS3) translates the skip statement. Since  $\text{skip}$  does nothing, the translation only needs to generate code to invoke  $\Psi$ .

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  $\Psi$ . Then,  $S_1$  is translated in the context  $\Psi_2$ . 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  $\Psi$ , 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  $\Psi_1$  if  $e'$  is evaluated to a positive value, and  $\Psi_2$  if otherwise. The control transfer code is generated by  $\llbracket c \rrbracket \Psi_i$ . 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,  $\llbracket c \rrbracket \Psi_1$  and  $\llbracket c \rrbracket \Psi_2$  generate the same initial entries  $\Psi'$ .

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,  $\Psi$  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  $\Psi$  is the variable  $z'$  of reactor  $c_1$ , which computes  $e'$  and determines whether to enter the loop body or to invoke  $\Psi$  with  $z'$ . The code for entering the loop body starts with `setvar`( $\langle c_1.z', \text{nid} \rangle, z'$ ) 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  $\llbracket S \rrbracket c = \langle P, \Psi_1 \rangle$  says that the entries of the target code of  $S$  is  $\Psi_1$ . Therefore,  $c_1$  needs to invoke  $\Psi_1$  with `nid` if the value of  $e'$  is positive. And the control transfer code is generated by  $\llbracket \langle c_1, \text{nid} \rangle \rrbracket \Psi_1$ .

## 7.4 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 7.1), which roughly says that the target code of a well-typed source program is a well-typed program in DSR.

**Definition 7.1** (Well-formed entry list). An entry list  $\Psi$  is well-formed with respect to  $P$ , written  $P \models \Psi$ , if the following two conditions hold. First, for any entry  $\langle c, w \rangle$  in  $\Psi$ ,  $P(c) = c[x:\bar{\sigma}]\{pc, \mathcal{Q}, \bar{\pi} \triangleright z:\bar{\tau}, \lambda.s\}$ , and if  $w = c'.z$ , then  $P \vdash \langle c'.z, \text{cid} \rangle : (\text{int}_\ell \text{ var})_{\ell'}$ . Second, if  $\Psi = \langle c_1, w_1 \rangle, \dots, \langle c_n, w_n \rangle$ , then  $label(\psi_{i+1}) \sqsubseteq label(\psi_i)$  holds for any  $i \in \{1, \dots, n\}$ , where  $label(\langle c_i, w_i \rangle) = label(c_i)$ .

**Lemma 7.1** (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  $P \models \Psi$ , and  $\llbracket \langle c, w \rangle \rrbracket \Psi = \langle s, \Psi' \rangle$ . Then  $\Gamma, w : \text{int}_{pc}, \text{nid} : \text{int}_{pc}; P; \mathcal{Q}; pc \vdash s : \tau$ , and  $P \models \Psi'$ .

*Proof.* By inspecting the translation rules (TC1) and (TC2).  $\square$

**Lemma 7.2** (Typing preservation). Suppose  $\llbracket \Gamma \rrbracket D = \Gamma'$ , and  $P'$  is the target code of an Aimp program  $S'$ . If  $e$  is an expression in  $S'$ , and  $\Gamma; \mathcal{R}; pc \vdash e : \tau$ , and  $\llbracket e \rrbracket \langle c, c', c_u, \ell, H \rangle = \langle P, \lambda \bar{\pi} \triangleright \bar{z} : \bar{\tau}. e' \rangle$ , and  $P' \models c, c'$ , then  $\Gamma'; P' \vdash P$  and  $\Gamma', \bar{z} : \bar{\tau} \vdash e' : \tau$ . If  $S$  is a statement in  $S'$ , and  $\Gamma; \mathcal{R}; pc \vdash S : \tau$ , and  $\llbracket S \rrbracket \Psi = \langle P, \Psi' \rangle$  and  $P' \models \Psi'$ , then  $\Gamma'; P' \vdash P$ .

*Proof.* By induction on the derivation of  $\Gamma; \mathcal{R}; pc \vdash e : \tau$  or  $\Gamma; \mathcal{R}; pc \vdash S : \tau$ .

- **Cases (INT) and (REF).** Obvious.
- **Case (DEREF).** If  $\Gamma'(m) = \sigma$ , then  $e'$  is  $!m$ , and  $P$  is  $\emptyset$  by rule (TE3). We have  $\Gamma' \vdash !m : \tau$ , since  $\tau = \sigma$ . If  $\Gamma'(m) = \text{int}_{\ell_1} @ \mathcal{Q}_m$ , by rule (TE4),  $P = \{r\}$  where

$$r = c\{\ell, \mathcal{Q}, \lambda.\text{exec}(\text{read}[\ell, \ell_1, m, c', \text{cid}, \langle c_u.z, \text{cid} \rangle], \text{nid}, \ell, \mathcal{Q}, ())\}.$$

By rules (EXEC) and (RD), we have:

$$\frac{\frac{\Gamma' \vdash \text{read}[\ell, \ell_1, m, c', \text{cid}, \langle c_u.z, \text{cid} \rangle] : \text{reactor}\{\ell, \mathcal{Q}_m\} \quad \ell \sqsubseteq \ell}{\Gamma'; P'; \mathcal{Q}; \ell \vdash \text{exec}(\text{read}[\ell, \ell_1, m, c', \text{cid}, \langle c_u.z, \text{cid} \rangle], \text{nid}, \ell, \mathcal{Q}, ()) : \text{stmt}_\ell}}{\Gamma'; P' \vdash r}$$

- **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,  $\Gamma', \bar{z}_i : \bar{\tau}_i \vdash e'_i : \tau$  for  $i \in \{1, 2\}$ . Thus,  $\Gamma', \bar{z}_1 : \bar{\tau}_1, \bar{z}_2 : \bar{\tau}_2 \vdash e'_1 + e'_2 : \tau$ .
- **Case (SKIP).** By Lemma 7.1.
- **Case (SEQ).**  $S$  is  $S_1; S_2$ , and we have  $\Gamma; \mathcal{R}; pc \vdash S_1 : \text{stmt}_{\mathcal{R}_1}$  and  $\Gamma; \mathcal{R}_1; pc \vdash S_2 : \tau$ . By rule (TS4),  $\llbracket S_2 \rrbracket \Psi = \langle P_2, \Psi_1 \rangle$  and  $\llbracket S_1 \rrbracket \Psi_1 = \langle P_1, \Psi' \rangle$ . 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  $\Gamma; \mathcal{R} \vdash e : \text{int}_{\ell'}$ . By rules (TS1) and (TS2),  $\llbracket e \rrbracket \langle c, c_1, \ell, \mathcal{Q} \rangle = \langle P_e, \lambda \bar{\pi} \triangleright z : \bar{\tau}. e' \rangle$ . By induction,  $\Gamma'; P' \vdash P_e$  and  $\Gamma', \bar{z} : \bar{\tau} \vdash e' : \text{int}_{\ell'}$ .

If  $\Gamma(m) = \sigma @ \mathcal{Q}_m$ , then (TS1) is used. By Lemma 7.1,  $\Gamma'; P' \vdash r_2$ . Let  $\Gamma'' = \Gamma', \bar{z} : \bar{\tau}, \text{cid} : \text{int}_\ell, \text{nid} : \text{int}_\ell$ . Then the following derivation shows that  $r_1$  is also well-typed:

$$\frac{\frac{\Gamma'; P' \vdash \text{write}[\ell, m, c_2, \text{cid}] : \text{reactor}\{\ell, \mathcal{Q}_m, \text{int}_\ell\}}{\Gamma'' \vdash \text{nid} : \text{int}_\ell \quad \Gamma'' \vdash \ell : \text{label}_\perp \quad \ell \sqsubseteq \ell \quad \Gamma, \bar{z} : \bar{\tau} \vdash e' : \text{int}_\ell}}{\Gamma''; P'; \mathcal{Q}; \ell \vdash \text{exec}(\text{write}[\ell, m, c_2, \text{cid}], \text{nid}, \ell, \mathcal{Q}_m, e') : \text{stmt}_\ell}}{\Gamma'; P' \vdash r_1}$$

If  $\Gamma(m) = \sigma$ , then (TS2) is used. By Lemma 7.1,  $\Gamma''; P'; \mathcal{Q}; \ell \vdash s' : \tau$ . Therefore, we have the following derivation:

$$\frac{\frac{\Gamma' \vdash m : (\text{int}_\ell \text{ ref})_\ell \quad \Gamma', \bar{z} : \bar{\tau} \vdash e' : \text{int}_\ell \quad \ell \sqsubseteq \text{int}_\ell}{\Gamma', \bar{z} : \bar{\tau}; P'; \mathcal{Q}; \ell \vdash m := e' : \text{stmt}_\ell} \quad \Gamma''; P'; \ell \vdash s : \tau}{\Gamma'; P' \vdash r_1}$$

- **Case (IF).**  $S$  is **if**  $e$  **then**  $S_1$  **else**  $S_2$ . By induction,  $P_e, P_1$  and  $P_2$  are well-typed, and  $e'$  is well-typed with respect to  $\Gamma', \bar{z} : \bar{\tau}$ . By Lemma 7.1,  $s'_1$  and  $s'_2$  are well-typed. Therefore, the statement **if**  $e'$  **then**  $s'_1$  **else**  $s'_2$  is well-typed, and so is  $r_1$ .
- **Case (WHILE).**  $S$  is **while**  $e$  **do**  $S'$ . By induction,  $P_e$  and  $P$  are well-typed. The following derivation proves that  $r_1$  is well-typed:

$$\frac{\frac{\Gamma', \bar{z} : \bar{\tau}, z' : \text{int}_\ell \vdash z' : \text{int}_\ell \quad \vdash \langle c_1.z', \text{nid} \rangle : (\text{int}_\ell \text{ var})_\ell}{\Gamma', \bar{z} : \bar{\tau}, z' : \text{int}_\ell; \ell \vdash \text{setvar}(\langle c_1.z', \text{nid} \rangle, z') : \text{stmt}_{\ell, \{c_1.z', \text{nid}\}}} \quad \Gamma', \bar{z} : \bar{\tau}, z' : \text{int}_\ell; \ell \vdash s'_1 : \tau}}{\Gamma', \bar{z} : \bar{\tau}, z' : \text{int}_\ell; P'; \mathcal{Q}; \ell \vdash \text{setvar}(\langle c_1.z', \text{nid} \rangle, z'); s'_1 : \tau}$$

$$\frac{\frac{\Gamma', \bar{z} : \bar{\tau}, z' : \text{int}_\ell; \ell \vdash e' : \text{int}_{\ell'}}{\Gamma', \bar{z} : \bar{\tau}, z' : \text{int}_\ell; \ell \vdash \text{setvar}(\langle c_1.z', \text{nid} \rangle, z'); s'_1 : \tau \quad \Gamma', \bar{z} : \bar{\tau}, z' : \text{int}_\ell; \ell \vdash s'_2 : \tau}}{\Gamma', \bar{z} : \bar{\tau}, z' : \text{int}_\ell; P'; \mathcal{Q}; \ell \vdash \text{if } e' \text{ then setvar}(\langle c_1.z', \text{nid} \rangle, z'); s'_1 \text{ else } s'_2 : \tau}}{\Gamma'; P' \vdash r_1}$$

- **Case (SUB).** Obvious by induction.

□

**Theorem 7.1** (Typing preservation). Suppose  $\Gamma; \mathcal{R}; pc \vdash S : \tau$ , and  $\llbracket S \rrbracket \emptyset = \langle P, c \rangle$  with respect to a distribution scheme  $D$ , and  $S = \{c\} S_1; S_2$ . Then  $\Gamma' \Vdash P$ , where  $\Gamma' = \llbracket \Gamma \rrbracket D$ .

*Proof.* By Lemma 7.2,  $\Gamma' \Vdash P$ . By examining the translation rules,  $P$  satisfies (RV1)–(RV3). □

## 7.5 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 6.2, 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  $\Theta$  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 7.2** ( $\Gamma; D \vdash M \approx \mathcal{M}$ ). For any  $m$  in  $\text{dom}(\Gamma)$ , then  $M(m) = \mathcal{M}(h, m)$  for any  $h \in D(m)$ , where  $\mathcal{M}(m, h) = v$  if  $v@t$  is the most recent version of  $m$  on host  $h$ .

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 7.3** ( $\mathcal{E}; \eta \models \Psi$ ). That  $\Psi$  is activated with context identifier  $\eta$  in the environment  $\mathcal{E}$ , written  $\mathcal{E}; \eta \models \Psi$ , if it can be inferred using the following rules, where  $\mathcal{E}(w, \eta)$  returns  $\eta$  if  $w$  is *cid*, and the value of  $\langle c.z, \eta \rangle$  in  $\mathcal{E}$  if  $w$  is *c.z*.

$$\frac{\mathcal{E}; \eta \models \langle c, w \rangle \quad \mathcal{E}; \mathcal{E}(w, \eta); \text{label}(c) \models \Psi}{\mathcal{E}; \eta \models \langle c, w \rangle, \Psi} \quad \frac{\mathcal{E}; \eta; \ell \models \langle c, w \rangle \quad \mathcal{E}; \mathcal{E}(w, \eta); \text{label}(c) \models \Psi}{\mathcal{E}; \eta; \ell \models \langle c, w \rangle, \Psi}$$

$$\frac{\forall h \in \text{hosts}(c). \langle c, \eta, \ell, \mathcal{A}, t, \text{off} \rangle \in \mathcal{E}(h)}{\mathcal{E}; \eta \models \langle c, w \rangle} \quad \frac{\forall h \in \text{hosts}(c). \langle c, \eta, \ell', \mathcal{A}, t, * \rangle \in \mathcal{E}(h) \quad \ell \sqsubseteq \ell'}{\mathcal{E}; \eta; \ell \models \langle c, w \rangle}$$

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  $\Psi$  would result in the list of activated entries  $\Psi'$ . Formally, it is defined using the following inference rules:

$$\begin{array}{ll} \text{[EL1]} & P; \Psi \vdash \text{skip} : \Psi \\ \text{[EL2]} & \frac{\llbracket S \rrbracket \Psi' = \langle P', \Psi \rangle \quad P' \subseteq P}{P; \Psi \vdash S : \Psi'} \\ \text{[EL3]} & \frac{P; \Psi \vdash S : \Psi' \quad \Psi_1 = \langle c, c_1.z \rangle, \Psi_2}{P; \Psi, \Psi_1 \vdash S : \Psi' \otimes \Psi_1} \\ \text{[EL4]} & \frac{P; \Psi \vdash S_1 : \Psi_1 \quad P; \Psi_1 \vdash S_2 : \Psi_2}{P; \Psi \vdash S_1; S_2 : \Psi_2} \end{array}$$

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  $\llbracket S \rrbracket \Psi' = \langle P', \Psi \rangle$ . Rule (EL3) is applied to the case that  $S$  belongs to the body of a `while` statement, and  $\Psi_1$  is the entry list for the computation following  $S$ . Based on the translation rule (TS6),  $\Psi_1 = \langle c, c_1.z \rangle, \Psi_2$ , where  $\langle c, c_1.z \rangle$  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  $\langle 1 \rangle$ . Otherwise, the entry list at the point that  $S$  terminates is  $\Psi', \Psi_1$ . Suppose  $\Psi_1 = \langle c, c_1.z \rangle, \Psi_2$ . Then the notation  $\Psi' \otimes \Psi_1$  denotes  $\Psi_1$  if  $\Psi' = c$ , and  $\Psi', \Psi_1$  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,  $\Psi$  is the entry list for the program  $P$  following the current thread, and  $\Pi$  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  $\Psi$  are activated, and there are no messages for the communication ports in  $\Pi$  yet. Formally, we have the following definition:

**Definition 7.4** ( $\Theta; \mathcal{E}; \eta \models \langle s; \Psi; \Pi \rangle$ ). 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  $\eta$ , 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  $\Theta$  has the form  $\langle s, t, h, c, \eta \rangle$ , and the timestamp of any thread in  $\Theta$  is less than or equal to  $t$ . Second,  $\mathcal{E}; \eta \models \Psi$ . Third, for any  $\pi$  in  $\Pi$ , 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  $\Psi$ , 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 7.5** (DSR-Aimp 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  $\Gamma, P, \eta$  and  $\Psi'$ , 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  $\Pi_S$  are the set of communication ports of the target code of  $S$ . Third,  $\Gamma \vdash M \approx \mathcal{M}$ .

**Lemma 7.3** (Expression translation soundness). Suppose  $\llbracket e \rrbracket \langle c, c', c_u, \ell, H \rangle = \langle P, \lambda \bar{\pi} \triangleright \tau z.e' \rangle$ , and  $\langle e, M \rangle \Downarrow v$ , and  $\Gamma \vdash M \approx \mathcal{M}$ , and  $\Theta; \mathcal{E}; \eta \models \langle c, \Psi; \Pi_P \cup \{c', c_u.\bar{z}\} \cup \Pi \rangle$ . Then there exists a run  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \mapsto^* \langle \Theta', \mathcal{M}, \mathcal{E}' \rangle$  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  $\Gamma(m) = \text{int}_{\ell_1} @ \mathcal{Q}$ . By (TE4),  $P$  is  $\{r\}$ , and

$$r = c\{\ell, \mathcal{Q}', \lambda.\text{exec}(\text{read}[\ell, \ell_1, m, c', \text{cid}, \langle c_u.z, \text{cid} \rangle], \text{nid}, \ell, \mathcal{Q}', ()).$$

Then by running the `exec` statement, we have  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \mapsto^* \langle \Theta_1, \mathcal{M}, \mathcal{E}_1 \rangle$ , and

$$\Theta_1; \mathcal{E}_1; \eta' \models \langle s'; \text{read}[\ell, \ell_1, m, c', \text{cid}, \langle c_u.z, \text{cid} \rangle], \Psi; \{c', c_u.z\} \cup \Pi \rangle,$$

where  $s'$  is  $\text{setvar}(\langle c_u.z, \eta \rangle, !m)$ ;  $\text{exec}(c', \eta, \ell, \emptyset, ())$ . In other words, the execution reaches the point that all the replicas of the read reactor are invoked with the newly-created context identifier  $\eta'$ . Further, by executing  $s'$  on all the hosts of  $m$  and processing all the messages sent by  $s'$ , the execution produces  $\langle \Theta_1, \mathcal{M}, \mathcal{E}_1 \rangle \mapsto^* \langle \Theta', \mathcal{M}, \mathcal{E}' \rangle$  such that  $\Theta'; \mathcal{E}'; \eta \models \langle c'; \Psi; \Pi \rangle$ . By  $\Gamma \vdash M \approx \mathcal{M}$ , the synthesizer  $\text{QR}[\mathcal{Q}, I]$  associated with  $c_u.z$  receives the  $\text{setvar}$  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,  $\langle z[\mathcal{A}], \mathcal{M}(h, t) \rangle \Downarrow v$ .

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).

$$\begin{array}{l} \langle \Theta, \mathcal{M}, \mathcal{E} \rangle \\ \mapsto^* \langle \Theta_1, \mathcal{M}, \mathcal{E}_1 \rangle \quad \langle s'; \Psi'; \{c', c_u.z\} \cup \Pi \rangle \\ \mapsto^* \langle \Theta', \mathcal{M}, \mathcal{E}' \rangle \quad \langle c', \Psi; \Pi \rangle \quad \langle z[\mathcal{A}], \mathcal{M}(h, t) \rangle \Downarrow M(m) \quad \text{By } \Gamma \vdash M \approx \mathcal{M} \end{array}$$

- $e$  is  $e_1 + e_2$ . By (TE5), we have  $\llbracket e_1 \rrbracket \langle c, c_1, c_u, \ell, \mathcal{Q} \rangle = \langle P_1, \lambda \overline{\pi_1} \triangleright \tau_1 \overline{z_1}. e'_1 \rangle$  and  $\llbracket e_2 \rrbracket \langle c_1, c', c_u, \ell, \mathcal{Q} \rangle = \langle P_2, \lambda \overline{\pi_2} \triangleright \tau_2 \overline{z_2}. e'_2 \rangle$ . Then we have the following execution:

$$\begin{array}{l} \langle \Theta, \mathcal{M}, \mathcal{E} \rangle \\ \mapsto^* \langle \Theta_1, \mathcal{M}, \mathcal{E}_1 \rangle \quad \langle c_1, \Psi; \Pi_{P_2} \cup \{c', c_u.\overline{z_2}\} \cup \Pi \rangle \quad \langle e'_1[\mathcal{A}], \mathcal{M}(h, t) \rangle \Downarrow v_1 \quad \text{By induction} \\ \mapsto^* \langle \Theta', \mathcal{M}, \mathcal{E}' \rangle \quad \langle c', \Psi; \Pi \rangle \quad \langle e'_2[\mathcal{A}], \mathcal{M}(h, t) \rangle \Downarrow v_2 \quad \text{By induction} \end{array}$$

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  $h$ . □

**Lemma 7.4** (Control transfer soundness). Suppose  $\llbracket \langle c, w \rangle \rrbracket \Psi' = \langle s, \Psi \rangle$ , and  $\Psi = \langle c, w \rangle, \Psi''$ , and  $\Theta; \mathcal{E}; \eta \models \langle s; c_1, \Psi''; \Pi \rangle$ . Then  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \mapsto^* \langle \Theta', \mathcal{M}, \mathcal{E}' \rangle$  such that  $\Theta'; \mathcal{E}'; \eta' \models \langle \Psi'; \Pi \rangle$ , where  $\eta' = \mathcal{E}(w, \eta)$ .

*Proof.* By inspecting translation rules (TC1) and (TC2). □

**Theorem 7.2** (Translation soundness). Suppose  $\Gamma; \mathcal{R}; pc \vdash S : \tau$ , and  $\langle S, M \rangle \mapsto \langle S', M' \rangle$ , and  $\Gamma; P; \eta \vdash \langle \Theta, \mathcal{M}, \mathcal{E} \rangle \approx \langle S, M, \Psi' \rangle$ . Then there exists a run  $E = \langle \Theta, \mathcal{M}, \mathcal{E} \rangle \mapsto^* \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$  such that  $\Gamma; P; \eta' \vdash \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle \approx \langle S', M', \Psi' \rangle$ . In addition, for any message  $\mu$  sent in  $E$ , the port of  $\mu$  is either in  $\Psi$  or in  $\Pi_S$ .

*Proof.* By induction on the evaluation step  $\langle S, M \rangle \mapsto \langle S', M' \rangle$ . 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  $\Theta; \mathcal{E}; \eta \models \langle \Psi; \Pi_S \rangle$ , 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

$$\begin{array}{l} \langle \Theta, \mathcal{M}, \mathcal{E} \rangle \\ \mapsto^* \langle \Theta_1, \mathcal{M}, \mathcal{E}_1 \rangle \quad \langle c_1, \Psi_1; \Pi_S - \Pi_{P_e} \rangle \quad \langle e'[\mathcal{A}], \mathcal{M}(h, t) \rangle \Downarrow v \quad \text{By Lemma 7.3} \end{array}$$

If  $\Gamma(m) = \sigma @ \mathcal{Q}$ , then rule (TS1) is used, and the code of  $c_1$  is  $\text{exec}(\text{write}[\ell, m, c_2, \text{cid}], \text{nid}, \ell, \emptyset, e')$ . Thus, we have

$$\begin{array}{l} \langle \Theta_1, \mathcal{M}, \mathcal{E}_1 \rangle \\ \mapsto^* \langle \Theta_2, \mathcal{M}, \mathcal{E}_2 \rangle \quad \langle m := v; \text{exec}(c_2, \eta, \ell, \&m, ()); \text{write}[\ell, m, c_2, \eta], \Psi_1; \{c_2\} \rangle \\ \mapsto^* \langle \Theta_3, \mathcal{M}', \mathcal{E}_3 \rangle \quad \langle c_2, \Psi_1; \emptyset \rangle \quad \mathcal{M}' = \mathcal{M}[m \mapsto v] \\ \mapsto^* \langle \Theta_4, \mathcal{M}', \mathcal{E}_4 \rangle \quad \langle \Psi'; \emptyset \rangle \quad \Psi' \vdash \text{skip} : \Psi' \quad \text{By Lemma 7.4} \end{array}$$

If  $\Gamma(m) = \sigma$ , the rule (TS2) is used, and the code of  $c_1$  is  $m := e'; s'$ , where  $s'$  comes from  $\llbracket c \rrbracket \Psi' = \langle s', \Psi \rangle$ . Thus, we have

$$\begin{array}{l} \langle \Theta_1, \mathcal{M}, \mathcal{E}_1 \rangle \\ \longmapsto^* \langle \Theta_2, \mathcal{M}', \mathcal{E}_2 \rangle \quad \langle s'; c_1, \Psi_1; \emptyset \rangle \quad \mathcal{M}'(h, m) = v \\ \longmapsto^* \langle \Theta_3, \mathcal{M}', \mathcal{E}_3 \rangle \quad \langle \Psi'; \emptyset \rangle \end{array} \quad \text{By Lemma 7.4}$$

- 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  $\pi$  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  $\{c\}$  skip;  $S'$ . By  $\Psi \vdash \{c\}$  skip;  $S' : \Psi'$ , we have  $\Psi \vdash \{c\}$  skip :  $\Psi'_1$  and  $\Psi'_1 \vdash S' : \Psi'$ . Then we have

$$\begin{array}{l} \langle \Theta, \mathcal{M}, \mathcal{E} \rangle \quad \langle c, \Psi_1; \Pi_S \rangle \\ \longmapsto^* \langle \Theta', \mathcal{M}, \mathcal{E}' \rangle \quad \langle \Psi'_1; \Pi_{S'} \rangle \end{array} \quad \text{By rule (TS3) and Lemma 7.4}$$

- Case (S4). Since  $P; \Psi \vdash S : \Psi'$ , we have that  $\llbracket S \rrbracket \Psi'_1 = \langle P', \Psi_1 \rangle$ , and  $\Psi = \Psi_1, \Psi_2$  and  $\Psi' = \Psi'_1 \otimes \Psi_2$ . By rule (TS5),  $\Psi_1 = c, \Psi''$ . Then we have

$$\begin{array}{l} \langle \Theta, \mathcal{M}, \mathcal{E} \rangle \\ \longmapsto^* \langle \Theta_1, \mathcal{M}, \mathcal{E}_1 \rangle \quad \langle c_1, \Psi''; \Pi_S \rangle \quad \langle e'[\mathcal{A}_{c_1, \eta}], \mathcal{M}(h, t) \rangle \downarrow n \quad \text{By Lemma 7.3} \\ \longmapsto^* \langle \Theta_2, \mathcal{M}, \mathcal{E}_2 \rangle \quad \langle s_1; c_1, \Psi''; \Pi_{S_1} \rangle \quad \text{By (S5)} \\ \longmapsto^* \langle \Theta_3, \mathcal{M}, \mathcal{E}_3 \rangle \quad \langle \Psi''_1; \Pi_{S_1} \rangle \quad \Psi''_1 \vdash S_1 : \Psi'_1 \quad \text{By Lemma 7.4} \end{array}$$

Also the above run is limited to the code of  $S$  and does not affect  $\Psi_2$ . 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 while  $e$  do  $S_1$ , and  $S'$  is  $S_1; \text{while } e \text{ do } S_1$ , and  $\langle e, M \rangle \downarrow n$  ( $n > 0$ ). Then we have:

$$\begin{array}{l} \langle \Theta, \mathcal{M}, \mathcal{E} \rangle \\ \longmapsto^* \langle \Theta_1, \mathcal{M}, \mathcal{E}_1 \rangle \quad \langle c_1, \Psi''; \Pi_S \rangle \quad \langle e'[\mathcal{A}_{c_1, \eta}], \mathcal{M} \rangle \downarrow n \quad \text{By Lemma 7.3} \\ \longmapsto^* \langle \Theta_2, \mathcal{M}, \mathcal{E}_2 \rangle \quad \langle \text{setvar}(\langle c_1.z', \text{nid} \rangle, z'); s_1; c_1, \Psi''; \Pi_S \rangle \quad \text{By (S5)} \\ \longmapsto^* \langle \Theta_3, \mathcal{M}, \mathcal{E}_3 \rangle \quad \langle s_1; c_1; \Pi_{S_1} \rangle \quad \mathcal{E}_3; \mathcal{A}_{\Theta_3}(\text{nid}); \ell_c \models \langle c, c_1.z' \rangle, \Psi'' \\ \longmapsto^* \langle \Theta', \mathcal{M}, \mathcal{E}' \rangle \quad \langle \Psi_1; \Pi_{S_1} \rangle \quad \mathcal{A}_{\Theta'}(\text{cid}) = \mathcal{A}_{\Theta_3}(\text{nid}) \quad \text{By Lemma 7.4} \end{array}$$

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  $\langle \Theta', \mathcal{M}, \mathcal{E}' \rangle \approx \langle S_1; \text{while } e \text{ do } S_1, M, \Psi' \rangle$ .

- Case (S7).  $S$  is while  $e$  do  $S_1$ , and  $\langle e, M \rangle \downarrow n$ , and  $n \leq 0$ . Then we have:

$$\begin{array}{l} \langle \Theta, \mathcal{M}, \mathcal{E} \rangle \\ \longmapsto^* \langle \Theta_1, \mathcal{M}, \mathcal{E}_1 \rangle \quad \langle c_1, \Psi''; \Pi_S \rangle \quad c_1 \vdash \langle e', \mathcal{M} \rangle \downarrow n \quad \text{By Lemma 7.3} \\ \longmapsto^* \langle \Theta_2, \mathcal{M}, \mathcal{E}_2 \rangle \quad \langle s_2; c_1, \Psi''; \emptyset \rangle \\ \longmapsto^* \langle \Theta_3, \mathcal{M}, \mathcal{E}_3 \rangle \quad \langle \Psi''; \emptyset \rangle \quad \mathcal{E}_3; \text{nid}; \ell_c \models \langle c, c_1, w \rangle, \Psi'' \quad \text{By Lemma 7.4} \end{array}$$

□

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 7.6** ( $\Gamma \vdash \mathcal{M} \approx_{A \not\leq l_A} M$ ). For any  $m$  such that  $A(\Gamma(m)) \not\leq l_A$ , if  $M(m) \neq \text{none}$ , then for any  $h$  in  $\mathcal{Q}_m$ ,  $A(h) \not\leq l_A$  implies  $\mathcal{M}(h, m) \neq \text{none}$ .

Again, we prove the availability preservation result by induction. First, we prove two lemmas about the availability of expression target code and control transfer code. The availability results need to be applicable to all executions. Thus, we use “ $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \rightsquigarrow^* \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$  such that a condition holds” to mean that for any run  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \mapsto^* \langle \Theta_1, \mathcal{M}_1, \mathcal{E}_1 \rangle$ , there exists  $\langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$  satisfying the condition and  $\langle \Theta_1, \mathcal{M}_1, \mathcal{E}_1 \rangle \mapsto^* \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$ . Let  $\mathcal{E} \models \langle c.z, \eta \rangle$  denote that the value of  $\langle c.z, \eta \rangle$  is already set 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 7.5** (Expression availability). Suppose  $\Gamma; \mathcal{R}; pc \vdash e : \text{int}_\ell$ , and  $\langle e, M \rangle \Downarrow n$ , and  $A(\mathcal{R}) \not\leq l_A$ , and  $\llbracket e \rrbracket \langle c, c', c_u, \ell, \mathcal{Q} \rangle = \langle P_e, \lambda \bar{\pi} \triangleright \tau \bar{z}. e' \rangle$ , and there exists  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle$  such that  $\mathcal{E} \models \langle c, \eta \rangle$ , and  $\Gamma \vdash \mathcal{M} \approx_{A \not\leq l_A} M$ . Then  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \rightsquigarrow^* \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$  such that  $\mathcal{E}' \models \langle c', \eta \rangle$  and  $\mathcal{E}' \models \langle c_u.\bar{z}, \eta \rangle$ .

*Proof.* By induction on the structure of  $e$ .

- $e$  is  $n$ ,  $m$ , or  $!m$  with  $\Gamma(m) = \sigma$ . In this case,  $\llbracket e \rrbracket \langle c, c', c_u, \ell, H \rangle = e$  and  $c = c'$ . Thus,  $\mathcal{E} \models \langle c', \eta \rangle$  and  $\mathcal{E}' \models \langle c_u.\bar{z}, \eta \rangle$  immediately hold.
- $e$  is  $!m$ , with  $\Gamma(m) = \sigma @ \mathcal{Q}$ . By rule (TE3),  $P_e = \{r\}$  and

$$r = c\{\ell, \mathcal{Q}, \lambda \text{.exec}(\text{read}[\ell, \ell_1, m, c', \text{cid}(c_u.z, \text{cid})], \text{nid}, \ell, \emptyset, ( ))\}.$$

Since  $\mathcal{E} \models \langle c, \eta \rangle$ ,  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \rightsquigarrow^* \langle \Theta_1, \mathcal{M}_1, \mathcal{E}_1 \rangle$  such that  $\mathcal{E}_1 \models \langle \text{read}[\ell, \ell_1, m, c, \eta, \langle c_u.z, \eta \rangle], \eta' \rangle$  where  $\eta' = \mathcal{E}(c.\text{nid}, \eta)$ . Since  $A(\mathcal{R}) \not\leq l_A$ , by rule (DM),  $A(\mathcal{Q}) \not\leq l_A$ , which means that at least a QR[ $\mathcal{Q}, I(\ell)$ ]-qualified set of hosts in  $\mathcal{Q}$  are available to finish executing the read reactor. Therefore,  $\langle \Theta_1, \mathcal{M}_1, \mathcal{E}_1 \rangle \rightsquigarrow^* \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$  such that  $\mathcal{E}' \models \langle c, \eta \rangle$  and  $\mathcal{E}' \models \langle c_u.z, \eta \rangle$ .

- $e$  is  $e_1 + e_2$ . By induction,  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \rightsquigarrow^* \langle \Theta_1, \mathcal{M}_1, \mathcal{E}_1 \rangle$  such that  $\mathcal{E}_1 \models \langle c_1, \eta \rangle$  and  $\mathcal{E}_1 \models \langle c_u.\bar{z}_1, \eta \rangle$ . Again, by induction,  $\langle \Theta_1, \mathcal{M}_1, \mathcal{E}_1 \rangle \rightsquigarrow^* \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$  such that  $\mathcal{E}' \models \langle c', \eta \rangle$  and  $\mathcal{E}' \models \langle c_u.\bar{z}_2, \eta \rangle$ .

□

**Lemma 7.6** (Control transfer availability). Suppose  $\llbracket \langle c, w \rangle \rrbracket \Psi' = \langle s, \Psi \rangle$ , and  $\langle \Theta_0, \mathcal{M}_0, \mathcal{E}_0 \rangle \mapsto^* \langle \Theta, \mathcal{M}, \mathcal{E} \rangle$  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  $A(c_1) \not\leq l_A$ . Then  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \rightsquigarrow^* \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$  such that  $\mathcal{E}'; \eta' \models \Psi'$ .

*Proof.* By inspecting rules (TC1) and (TC2). □

**Lemma 7.7** (Availability preservation I). Suppose  $\Gamma; \mathcal{R}; pc \vdash S : \text{stmt}_{\mathcal{R}'}$ , and  $I(pc) \leq l_A$  and  $A(\mathcal{R}) \not\leq l_A$ , and  $P; \Psi \vdash S : \Psi'$ , and  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle$  satisfies  $\mathcal{E}; \eta \models \Psi$  and  $\text{available}(\mathcal{M}, \mathcal{R}, l_A)$ , which means that for any  $m$  in  $\text{dom}(\Gamma)$ ,  $A(\Gamma(m)) \not\leq l_A$  and  $m \notin \mathcal{R}$  imply that  $m$  is available in  $\mathcal{M}$ . Then  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \rightsquigarrow^* \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$  such that  $\mathcal{E}'; \eta \models \Psi'$ , and  $\text{available}(\mathcal{M}', \mathcal{R}', l_A)$ .

*Proof.* By induction on the structure of  $S$ .

- $S$  is skip. Since  $\Psi' = \Psi$ ,  $\langle \mathcal{E}, \mathcal{M}, \Theta \rangle$  already satisfies the conditions.

- $S$  is  $\{c\}$  skip. By Lemma 7.6.
- $S$  is  $\{c\} m := e$ . Then we have  $\llbracket S \rrbracket \Psi' = \langle P_1, \Psi_1 \rangle$ , and  $P_1 \subseteq P$ . First, suppose  $\Gamma(m) = \sigma$ . By (TS2),  $\llbracket e \rrbracket \langle c_1, c'_1, \ell, H \rangle = \langle P_e, \lambda \bar{\pi} \bar{\tau} \bar{z}. e' \rangle$ . Since  $A(\mathcal{R}) \not\leq l_A$ , we have  $\langle e, M \rangle \Downarrow n$ . By Lemma 7.5 and  $\mathcal{E}; \eta \vDash \Psi_1$ , we have  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \rightsquigarrow^* \langle \Theta_1, \mathcal{M}_1, \mathcal{E}_1 \rangle$  such that  $\mathcal{E}_1 \vDash \langle c'_1, \eta \rangle$ . Suppose  $h'$  is the host where  $c'_1$  resides. By rule (DS),  $A(m) \leq A(h')$ . If  $A(\mathcal{R}) \not\leq l_A$ , then  $A(m) \not\leq l_A$  and  $A(h') \not\leq l_A$ , which means that  $h'$  is available. Since  $\mathcal{R}'$  is  $\mathcal{R} - \{m\}$ , we have  $\mathcal{R}' \vdash \mathcal{M}' \approx_{A \not\leq l_A} M'$ . By rule (TS2) and Lemma 7.6,  $\langle \Theta_1, \mathcal{M}_1, \mathcal{E}_1 \rangle \rightsquigarrow^* \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$  such that  $\mathcal{E}' \vDash \Psi'$ .
- $S$  is  $S_1; S_2$ . By induction.
- $S$  is  $\{c\}$  if  $e$  then  $S_1$  else  $S_2$ . Since  $A(\mathcal{R}) \not\leq l_A$ ,  $\langle e, M \rangle \Downarrow n$ . Suppose  $\Gamma; \mathcal{R} \vdash S : \ell$ , and  $\mathcal{Q}_c = \langle H, \emptyset \rangle$ . Then  $A(\mathcal{R}) \leq A(H, \text{LT}[\ell])$ . Since  $A(\mathcal{R}) \not\leq l_A$ , there exists a  $\text{LT}[\ell]$ -qualified subset  $H'$  of  $H$  such that  $A_{\cap}(H') \not\leq l_A$ . 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 7.6,  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \rightsquigarrow^* \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$  such that  $\mathcal{E}' \vDash \Psi''$  and  $\Psi'' \vdash S_1 : \Psi'$ . Then the induction hypothesis is applicable.
- $S$  is while  $e$  do  $S'$ . By the typing rule (WHILE) of Aimp,  $I(pc) \leq l_A$  implies  $A(\mathcal{R}) \leq l_A$ . Thus, this case cannot occur.

□

According to the translation soundness theorem, for a run of the source program  $\langle S, M \rangle \mapsto^* \langle S', M' \rangle$ , 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  $\langle S_1, M_1 \rangle_{\eta_1} \mapsto \langle S_2, M_2 \rangle_{\eta_2}$  to denote that  $\eta_1$  and  $\eta_2$  are the corresponding context identifier of configurations  $\langle S_1, M_1 \rangle$  and  $\langle S_2, M_2 \rangle$ .

**Lemma 7.8** (Availability preservation II). Suppose  $\Gamma; \mathcal{R}; pc \vdash S : \text{stmt}_{\mathcal{R}'}$  and  $I(pc) \not\leq l_A$  and  $A(\mathcal{R}) \not\leq l_A$  and  $\langle S, M \rangle_{\eta} \mapsto \langle S_1, M_1 \rangle_{\eta'}$ , and  $P; \Psi \vdash S : \Psi'$ , and  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle$  satisfies  $\mathcal{E}; \eta \vDash \Psi$  and  $\Gamma \vdash \mathcal{M} \approx_{A \not\leq l_A} M$ . Then  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \rightsquigarrow^* \langle \Theta_2, \mathcal{M}_2, \mathcal{E}_2 \rangle$  such that  $\mathcal{E}_2; \eta' \vDash \Psi_2$ , and  $\Psi_2 \vdash S_2 : \Psi'$ , and  $\Gamma \vdash \mathcal{M}_2 \approx_{A \not\leq l_A} M_1$ , 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  $\Gamma; \mathcal{R}; pc \vdash S'_i : \text{stmt}_{\mathcal{R}'}$  and  $I(pc) \leq L$ .

*Proof.* By induction on  $\langle S, M \rangle \mapsto \langle S', M' \rangle$ . Without loss of generality, suppose  $\llbracket S \rrbracket \Psi' = \langle P, \Psi \rangle$ . In general,  $\llbracket S \rrbracket \Psi'' = \langle P, \Psi_1 \rangle$  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),  $\llbracket e \rrbracket \langle c, c_1, \ell, H \rangle = \langle P_e, \lambda \bar{\pi} \bar{\tau} \bar{z}. e' \rangle$ , and the first element of  $\Psi$  is  $c$ . By Lemma 7.5 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  $\llbracket c \rrbracket \Psi = \langle s', \Psi' \rangle$ . Suppose  $h_1$  is the host where  $c_1$  resides. By rule (DM),  $A(m) \leq A(h_1)$ . Since  $A(\mathcal{R}) \not\leq l_A$ , we have  $A(h_1) \not\leq l_A$ , 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  $\Gamma' \vdash \mathcal{M}_2 \approx_{A \not\leq l_A} M_1$ . By rule (TS2),  $\llbracket c \rrbracket \Psi' = \langle s', \Psi \rangle$ . By Lemma 7.6,  $\langle \Theta_1, \mathcal{M}_1, \mathcal{E}_1 \rangle \rightsquigarrow^* \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$  in finite steps such that  $\mathcal{E}' \vDash \Psi'$ . In addition,  $S_2$  is skip, and  $\Psi' \vdash \text{skip} : \Psi'$ .

Second,  $\Gamma(m) = \sigma @ \mathcal{Q}_m$ . By rule (DS),  $A(\mathcal{R}) \leq A(H, \text{LT}[I(m)])$ . As a result, at least a  $\text{LT}[I(m)]$ -qualified subset  $H'$  of  $H$  are available to invoke  $\text{write}[\ell, m, c_2, \eta]$ . Since  $A(\ell) \not\leq l_A$ , 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  $A(\ell) \not\leq l_A$ , the available hosts in  $\mathcal{Q}$  have sufficient integrity so that the remote requests sent by  $s'$  would be accepted. By Lemma 7.6,  $\langle \Theta_1, \mathcal{M}_1, \mathcal{E}_1 \rangle \rightsquigarrow^* \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$  such that  $\mathcal{E}' ; \eta' \models \Psi'$ .

- Case (S2).  $S$  is  $S_1; S_2$ , and  $\langle S_1, M \rangle \mapsto \langle S''_1, M' \rangle$ . 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  $\mathcal{E}_2 ; \eta \models \Psi_2$ , 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 7.6. Otherwise,  $S$  is skip;  $S_2$ . Thus,  $S_1 = S_2$ , and  $P ; \Psi \vdash S_2 : \Psi'$  since  $P ; \Psi \vdash \text{skip} : \Psi$ .
- Case (S4).  $S$  is if  $e$  then  $S_1$  else  $S_2$ , and  $\langle e, M \rangle \Downarrow n$  and  $n > 0$ . By Lemma 7.5,  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \rightsquigarrow^* \langle \Theta_1, \mathcal{M}, \mathcal{E}_1 \rangle$  such that  $\mathcal{E}_1 ; \eta \models c_1$ . By Theorem 7.2, there exists a benchmark execution  $\langle \Theta_0, \mathcal{M}_0, \mathcal{E}_0 \rangle \mapsto^* \langle \Theta_2, \mathcal{M}_2, \mathcal{E}_2 \rangle$  such that  $\langle e'[\mathcal{A}_{c_1, \eta}], \mathcal{M}_2 \rangle \Downarrow n$ . If  $I(e) \not\leq L$ , then by Theorem 6.2, 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 7.6,  $\langle \Theta_1, \mathcal{M}, \mathcal{E}_1 \rangle \rightsquigarrow^* \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$  such that  $\mathcal{E}' ; \eta \vdash \Psi_2$  where  $\llbracket S_1 \rrbracket \Psi' = \langle P_1, \Psi_2 \rangle$ .  
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  $S_1; \text{while } e \text{ do } S_1$ . By Lemma 7.5,  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \rightsquigarrow^* \langle \Theta', \mathcal{M}', \mathcal{E}' \rangle$  such that  $\mathcal{E}' ; \eta \models c_1$ . Moreover,  $A(\mathcal{R}) \not\leq l_A$  implies  $I(e) \not\leq l_A$ . By Theorem 7.2, for any  $h$  in  $\mathcal{Q}(c_1)$  such that  $I(h) \not\leq l_A$ ,  $\langle e'[\mathcal{A}_{c_1, \eta}], \mathcal{M}'(h, t) \rangle \Downarrow n$ . Since  $n > 0$ , “setvar( $\langle c_1.z', \text{id} \rangle, z'$ );  $s_1$ ” is executed on host  $h$ . By executing setvar( $\langle c_1.z', \text{id} \rangle, z'$ ) 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 7.5,  $\langle \Theta, \mathcal{M}, \mathcal{E} \rangle \rightsquigarrow^* \langle \Theta_1, \mathcal{M}, \mathcal{E}_1 \rangle$  such that  $\mathcal{E}'' ; \eta \models c_1$ . Since  $I(e) \not\leq l_A$ , for any  $h$  in  $\mathcal{Q}_{c_1}$  such that  $I(h) \not\leq l_A$ ,  $\langle e'[\mathcal{A}_{c_1, \eta}], \mathcal{M}(h, t) \rangle \Downarrow n$ , and  $s_2$  is executed on  $h$ . Therefore, by Lemma 7.6,  $\langle \Theta_1, \mathcal{M}, \mathcal{E}_1 \rangle \rightsquigarrow^* \langle \Theta', \mathcal{M}, \mathcal{E}' \rangle$  such that  $\mathcal{E}' ; \eta' \models \Psi'$ .

□

**Theorem 7.3** (Availability preservation). Suppose  $\Gamma ; \mathcal{R} ; pc \vdash S : \tau$ , and  $\langle S, M \rangle \mapsto^* \langle S', M' \rangle$ , and  $\llbracket S \rrbracket \emptyset = \langle P, c \rangle$ , 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  $\Gamma \vdash \mathcal{M}' \approx_{A \leq l_A} M'$

*Proof.* By induction on the number of steps of  $\langle S, M \rangle \mapsto^* \langle S', M' \rangle$ , and Lemmas 7.7 and 7.8. □

## 8 Related Work

The most closely related work is the Jif/split system [27, 28] that introduced secure program partitioning and automatic replication of code and data. However, Jif/split cannot specify or enforce availability, and there is no correctness proof for the replication mechanisms in Jif/split.

Language-based information flow control techniques [5, 23, 10, 17, 18, 3, 20] can enforce noninterference, including in concurrent and distributed systems [21, 19, 26]. But this work does not address availability and assumes a trusted computing platform.

In previous work [29], we extend the DLM to specify availability policies and present a type-based approach for enforcing availability policies in a sequential program. This paper examines the distributed setting to permit formal analysis of the availability guarantees of various replication schemes.

Walker et al. [24] designed  $\lambda_{\text{zap}}$ , a lambda calculus that exhibits intermittent data faults, and use it to formalize the idea of achieving fault tolerance through replication and majority voting. However,  $\lambda_{\text{zap}}$  describes a single machine with at most one integrity fault.

Quorum systems [11, 14] 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.

The Replica Management System (RMS) [12] 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.

Program slicing techniques [25, 22] 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 [6, 13], the focus of work on program slicing has been debugging and understanding existing software.

The design of DSR was inspired by concurrent process calculi such as the join calculus [7] and the actor model [2].

## 9 Conclusions

This paper presents a framework for running a security-typed sequential program and enforcing its availability, integrity and confidentiality policies in a realistic distributed platform that contains mutually distrusted hosts. To achieve a strong availability guarantee along with strong integrity and confidentiality guarantees, this paper has solved several technical challenges. More complicated replication schemes such as quorum systems have been used to enforce both integrity and availability and guarantee replica consistency. Multi-level timestamps are used to achieve synchronization while preserving confidentiality of control flow.

This paper also presents a security-typed distributed intermediate language DSR, and a formal translation from a sequential source program to a target distributed program in DSR. The noninterference results of DSR and the adequacy of the translation together provably guarantee that the target code of a well-typed source program enforces the security policies of the source.

## Acknowledgements

The authors would like to thank Lorenzo Alvisi, Michael Clarkson, Andrei Sabelfeld, Stephen Chong and Heiko Mantel for their insightful suggestions and comments on this work. Michael Clarkson also helped improve the presentation of this work.

This research was supported in part by National Science Foundation grants 0208642 and 0430161. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright annotation thereon. The views and conclusions here are those of the authors and do not necessarily reflect the views of these sponsors.

## References

- [1] Johan Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, January 2000.
- [2] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

- [3] 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.
- [4] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.
- [5] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.
- [10] 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.
- [11] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992.
- [16] 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.
- [17] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proc. 5nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 46–57, 2000.
- [18] 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.
- [19] 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.
- [20] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.
- [21] 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.
- [22] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.
- [23] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

- [24] 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.
- [25] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.
- [26] 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.
- [27] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.
- [28] 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.
- [29] Lantian Zheng and Andrew C. Myers. End-to-end availability policies and noninterference. In *Proc. 18th IEEE Computer Security Foundations Workshop*, pages 272–286, June 2005.
- [30] Lantian Zheng and Andrew C. Myers. End-to-end availability policies and noninterference. Technical Report 2005–1987, Cornell University Computing and Information Science, 2005.