End-to-end Availability Policies and Noninterference

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

Abstract

This paper introduces the use of static information flow analysis for the specification and enforcement of end-to-end availability policies in programs. We generalize the decentralized label model, which is about confidentiality and integrity, to also include security policies for availability. These policies characterize acceptable risks by representing them as principals. We show that in this setting, a suitable extension of noninterference corresponds to a strong, end-to-end availability guarantee. This approach provides a natural way to specify availability policies and enables existing static dependency analysis techniques to be adapted for availability. The paper presents a simple language in which fine-grained information security policies can be specified as type annotations. These annotations can include requirements for all three major security properties: confidentiality, integrity, and availability. The type system for the language provably guarantees that any well-typed program has the desired noninterference properties, ensuring confidentiality, integrity, and availability.

1. Introduction

Availability is an important aspect of security, and attacks that harm availability may cause considerable damage. For example, denial-of-service attacks have been an increasing problem for web services. Although availability is often considered one of the three key aspects of information security (along with confidentiality and integrity), assuring availability has been the province of the fault tolerance community, largely divorced from other security concerns.

This paper suggests that the divide between availability and the other security properties can be bridged. It shows that single, common framework can accommodate reasoning about confidentiality, integrity, and availability. The first part of this framework is a policy language for the specification of rich security policies for confidentiality, integrity, and availability. This policy language is an extension to the decentralized label model [13], and similarly, it is able to describe security policies to be enforced on behalf of mutually distrusting principals. This ability is just as important for availability as it is for confidentiality and integrity.

The second part of the framework is a formal meaning for security policies in the policy language. A security policy demands that the system behave in a way that enforces the policy; this paper characterizes precisely what the behavior can be. In the context of confidentiality and integrity, end-to-end security policies have generally been interpreted as information flow policies requiring that the system obey noninterference. As this paper shows, availability policies too can be interpreted as requiring a form of noninterference.

The third part of the framework is a static program analysis that enforces policies for confidentiality, integrity, and availability. Previous work has shown that it is possible to enforce end-to-end confidentiality and integrity properties by static, compile-time analysis of program text (for a survey see [15]). What is new here is a demonstration that the same approach applies to availability: an availability analysis can be expressed in tractable form as a programming language type system that also enforces confidentiality and integrity.

The paper is structured as follows. Section 2 presents the new policy language for expressing requirements for availability, integrity, and confidentiality. Section 3 instantiates this label system as program annotations in a simple programming language. Section 4 uses the operational semantics of the language to express trace-based security properties that correspond to availability, integrity, and confidentiality policies. Section 5 gives a type system for this programming language and states the corresponding security theorem: well-typed programs are semantically secure (see the appendix for proofs). Section 6 extends the simple programming language to express richer notions of availability and also to describe some aspects of distributed systems. Section 7 discusses related work, and Section 8 concludes.

2. Availability policies

We begin by pinning down more precisely what is meant by "availability", then define an expressive policy language for availability, and demonstrate the policy language can be used for confidentiality and integrity too.

2.1. Availability

A system output is considered to be *available* if it will be produced *eventually*. Note that the value of the output does not have to be correct—that is the province of integrity.

The availability of an output is the degree to which the output is available. There are two common ways to specify this degree of availability. The first approach is to quantify system reliability using measurable criteria such as the failure probability or the MTTF/MTTR (*mean time to fail / mean time to recover*) ratio [17]. The second approach, from the fault tolerance community, is to specify what factors may cause the system to fail. For example, it is common to specify the minimum number of host failures (either fail-stop or Byzantine) needed to bring down the system [16]. In this work we explore the second approach: specifying availabilities as failure factors.

The above description of availability glosses over another aspect of availability: timeliness. How soon does an output have to occur after it is expected in order to be considered to be available? For real-time services, there may be hard time bounds beyond which a late output is useless. Reasoning about how long it takes to generate an output adds considerable complexity, however, so for now let us consider an output to be available if it arrives eventually. Section 6 presents an extension to this framework that supports reasoning about timeliness.

2.2. Failures as principals

We assume that the unavailability of a system output can be attributed to a *failure*. There are many kinds of possible failures: for example, hardware failures such as losing power, software failures such as subversion by an attacker, and human failures, such as a user who provides incorrect or even malicious inputs. Our goal is a general policy language that can describe all these kinds of failures and how the availability of the system is affected by them. This description can then aid in designing systems that resist failure.

In general, we regard a failure as the malfunction of a *principal*, an entity that may affect the behavior of a system. Therefore, the failure can be denoted by the responsible principal. For some failures, the corresponding principal is simply an abstract name, which might represent hardware, users, attacks or defense mechanisms, as shown in the following examples:

- `power`: the main power supply of a system, whose failure may bring down the entire system.
- `root`: user `root`, which usually has the ability to shut down a system.
- $\texttt{DDoS}_{1000}$: the distributed denial of service attack launched from 1000 machines. This principal can be used to specify the availability of a system that can tolerate DDoS attacks launched from less than 1000 machines.
- `puzzle`: the puzzle generated by a puzzle-based defense mechanism [8] for DoS attacks. This principal fails if attackers can easily solve the puzzle and launch DoS attacks successfully.

It is also useful to describe more complex failure scenarios using a combination of principals. For example, suppose that there is a principal `ups` representing a back-up power supply, and to make the system unavailable, both `power` and `ups` need to fail. This joint failure is represented by a conjunction, $\texttt{power}\wedge\texttt{ups}$.

More generally, principals p may be constructed using conjunction and disjunction operators $\wedge$ and $\vee$:

$$p \ ::= \ a \ | \ p_1 \wedge p_2 \ | \ p_1 \vee p_2$$

The notation a represents an abstract name that a principal. The principal conjunction $p_1 \wedge p_2$ represents a joint failure factor: $p_1 \wedge p_2$ fails only if both p_1 and p_2 fail. Another constructor $\vee$ is used to construct a group (disjunction): the principal $p_1 \vee p_2$ represents a failure that happens if either p_1 or p_2 fails. For example, if the principal $\texttt{Bob}\vee\texttt{power}$ can make a system fail, then Bob and the power supply each can cause the failure.

To demonstrate the expressiveness of this principal language, we specify the availability of a quorum system [10]. A quorum system is a collection $\{Q_1, \ldots, Q_n\}$ of sets (quorums) of hosts, every two of which intersect. A quorum system is available as long as there is some quorum in which no hosts fail. Therefore, a quorum system cannot tolerate the failure of a set of hosts B such that for every quorum Q_i, $B \cap Q_i$ is not empty. Thus, if the principal h represents a host, the availability of a quorum system can be specified by the principal $\bigvee_{B \mid \forall Q_i . B \cap Q_i \neq \emptyset} (\bigwedge_{h \in B} h)$.

2.3. Principal hierarchy

We write $p_1 \succeq p_2$ if the principal p_1 *acts for* another principal p_2 [13]. Interpreting failures as principals, this means the failure of p_1 is worse than the failure of p_2 (or the same). The acts-for relationship is useful for formally analyzing availability, because $p_1 \succeq p_2$ means that the availability level represented by p_1 is at least as high as the availability level represented by p_2.

The acts-for relationship between principals is called a *principal hierarchy* $\mathcal{H}$, an ordering (actually, a pre-order) on the set of principals. By the definition of the acts-for relationship, a principal hierarchy needs to satisfy the follow-

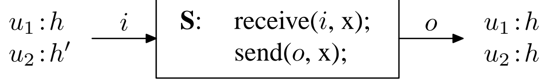

Figure 1. Availability policies vs. assertions

ing deductive rules:

$$p_1 \wedge p_2 \succeq p_1 \qquad \frac{p_1 \succeq p_2 \quad p_2 \succeq p_3}{p_1 \succeq p_3} \qquad \frac{p_1 \succeq p_2}{p_1 \succeq p_2 \vee p_3}$$

$$\frac{p_1 \succeq p_3 \quad p_2 \succeq p_3}{p_1 \vee p_2 \succeq p_3} \qquad \frac{p_1 \succeq p_2 \quad p_1 \succeq p_3}{p_1 \succeq p_2 \wedge p_3}$$

2.4. Owned availability policies

Mutual distrust is intrinsic to security. In order for all stakeholders (such as users) to believe that a computer system enforces their security, it is necessary that they be able to express their distinct security requirements. Thus, each individual user should be able to specify and manage their own policies. Decentralized policy management is especially important for distributed systems.

This observation applies just as much to availability as it does to confidentiality and integrity. Therefore, the availability policies defined here support a notion of *ownership*, as in the decentralized label model (DLM), which applies ownership to confidentiality and integrity policies. An owned availability policy has the form $u : p$, where principal u is the policy owner, and principal p represents the availability level required by u.

The owner of a policy is allowed to affect the meaning of the policy by making relevant *security assertions*. For example, Figure 1 shows a simple system S, which receives an input i and sends the value of the input to an output o. Suppose two availability policies $u_1 : h$ and $u_2 : h$ are specified on the output o. Thus, both u_1 and u_2 requires the output o to be available if host h does not fail.

Our goal is to determine whether a system can enforce the availability policies on its outputs. Since a system cannot affect the availabilities of its inputs, an availability policy specified on its input is considered a security assertion by the policy owner that the policy is already enforced. In this example, two availability policies $u_1 : h$ and $u_2 : h'$ are specified on i. In other words, u_1 asserts that i is available if h does not fail, while u_2 asserts that i is available if h' does not fail. In general, the security assertions of u can be used to enforce the security policies of u. For u_1, if h does not fail, then i is available, then o will be available. Thus, its availability policy on o is enforced. But for u_2, if h does not fail, i may still be unavailable, which will cause o to be unavailable. Thus, the availability policy $u_2 : h$ on o is not enforced by S.

Thus, if an output has the availability policy $u : p$, it means that the availability of the output will be enforced subject to two security assumptions. The first assumption is that p does not fail. The second assumption is that the security assertions of u are valid.

2.5. General security policies

Although availability is the focus of this paper, it cannot be considered in isolation from confidentiality and integrity, for two reasons. First, availability can be in tension with confidentiality and integrity because a mechanism that helps improve availability (such as replication) can harm confidentiality and integrity. Second, availability can depend on integrity. For example, consider the following pseudo-code:

```
while (x > 0) skip;
send(o, y);
```

This program outputs the value of `y` after the `while` statement terminates. If the value of `x` is positive, the `while` statement is an infinite loop, and the output is unavailable. Thus, an attacker can compromise the *availability* of the output `o` by compromising the *integrity* of `x`.

Essentially the same policy language can be used for all three major kinds of policies: confidentiality, integrity, and availability. In the DLM, an integrity policy applied as a label to some data d is written $u : p_1, \ldots, p_n$, meaning that u allows only principals $p_1, \ldots, p_n$ to affect updates to d. (This is of course stronger than the corresponding access control policy because the prohibited effect on updates might be indirect.) And a confidentiality policy $u : p_1, \ldots, p_n$ means that u allows only principals $p_1, \ldots p_n$ to receive information affected by the labeled data.

The key insight is that for every aspect of security, principals on the right-hand side of a DLM policy can be interpreted as failure factors. A confidentiality policy $u : p$ means that u requires the data will remain confidential as long as p does not fail to keep it confidential. For integrity, u requires the data will have integrity unless p fails to provide correct data. As an availability policy, it says that u requires that the data is available as long as p does not fail. In each case the policy $u : p$ can be interpreted as u's requirement that only the failure of p may compromise the corresponding aspect of security.

The DLM's ability to list multiple principals on the right-hand side of a policy is subsumed by disjunctive principals. The confidentiality or integrity policy $u : p_1, \ldots, p_n$ can be written in the form $u : p_1 \vee \ldots \vee p_n$, indicating that $p_1 \vee \ldots \vee p_n$ are expected not to fail. This is equivalent to assuming that none of p_1 through p_n will fail, as desired.

Thus, the policy form $u : p$ is a generic security policy applicable not only to availability but also to confidentiality and integrity. This commonality aids the analysis of the interactions between the three security properties.

2.6. Policy semantics

Whether the policy $u\!:\!p$ is used to talk about confidentiality, integrity, or availability, it corresponds to the two security assumptions above: that p is a trustworthy enforcer of the security property under consideration, and that the security assertions of u are valid.

These assumptions can be formalized as a proposition σ using the following syntax:

$$\sigma ::= \mathit{trust}(p) \mid \mathit{believe}(p) \mid \sigma_1 \wedge \sigma_2 \mid \sigma_1 \vee \sigma_2$$

where $\mathit{trust}(p)$ means that principal p does not fail (that is, does not violate the security property under consideration), $\mathit{believe}(p)$ means that p's assertions are valid, and $\wedge$ and $\vee$ are the ordinary logical connectives.

A security policy can be given a formal semantics in terms of these propositions. Using semantic brackets $[\![\cdot]\!]$ to indicate the semantic function, the meaning of a policy $u\!:\!p$ is:

$$[\![u\!:\!p]\!] = \mathit{believe}(u) \wedge \mathit{trust}(p)$$

To enforce a policy P is to guarantee a security property (such as availability) under the assumption that $[\![P]\!]$ is true. For example, suppose Alice specifies an availability policy `Alice` : $h_1 \wedge h_2$ on one of her files, and Alice assumes $h_3 \succeq h_1$ and $h_3 \succeq h_2$, where h_1, h_2 and h_3 are host machines. To enforce the policy is to guarantee the availability of the file under the assumption that $\mathit{believe}(Alice) \wedge \mathit{trust}(h_1 \wedge h_2)$ is true. Therefore, one way to enforce the policy is to replicate the file on hosts h_1 and h_2 because $\mathit{trust}(h_1 \wedge h_2)$ means that h_1 and h_2 cannot fail at the same time, which ensures that at least one host is available to serve accesses to the file. Moreover, there exists another way to enforce the policy: storing the file on h_3. Since $\mathit{believe}(Alice)$ is true, Alice's assumption that h_3 acts for h_1 and h_2 is valid, which implies that h_3 does not fail because either h_1 or h_2 does not fail.

To enforce system-wide availability (confidentiality, integrity) it is necessary to be able to determine whether one security policy is at least as strong as another. A policy P_2 is as strong as another policy P_1, written $P_1 \leq P_2$, if the enforcement of P_2 implies the enforcement of P_1. This policy ordering falls out naturally from the semantics of policies. Intuitively, a policy representing a weaker security assumption is more difficult to enforce, because a security property is more difficult to satisfy under a weaker assumption. Then $[\![P_1]\!] \Rightarrow [\![P_2]\!]$ implies $P_1 \leq P_2$.

Consider the above file access example. Suppose a password is needed to access the file. Then the password should have an availability policy P such that `Alice`$:h_1 \wedge h_2 \leq P$, because the availability of the file depends on the availability of the password.

By the definition of the acts-for relationship between principals, the following statements hold:

- $\mathit{trust}(p_2) \Rightarrow \mathit{trust}(p_1)$ if $p_1 \succeq p_2$.
- $\mathit{believe}(u_2) \Rightarrow \mathit{believe}(u_1)$ if $u_1 \succeq u_2$.

The first holds because in this case the failure of p_1 implies the failure of p_2; the second, because u_2 is subject to any security assertions made by u_1. From these statements we see that $\mathit{trust}(p_1 \wedge p_2) \equiv \mathit{trust}(p_1) \vee \mathit{trust}(p_2)$ and $\mathit{trust}(p_1 \vee p_2) \equiv \mathit{trust}(p_1) \wedge \mathit{trust}(p_2)$, and similarly with $\mathit{believe}(\cdot)$.

From these two observations and the semantics, the following rule for ordering policies immediately follows:

$$[\mathit{CP}] \qquad \frac{u_2 \succeq u_1 \quad p_2 \succeq p_1}{u_1\!:\!p_1 \leq u_2\!:\!p_2}$$

2.7. Combining owned policies

In general, different principals may have different security requirements. It is convenient to incorporate the security policies of several principals into one entity so that they can be analyzed and manipulated together. This is accomplished by writing a *set* of policies $\beta = \{P_1, \ldots, P_n\}$, where each P_i is an owned policy $u_i\!:\!p_i$ of the same kind (confidentiality, integrity, or availability).

A combined policy β is enforced if and only if all the policies in β are enforced. As a result, the security assumption described by β must be weaker than or equal to the security assumptions described by policies in β. Therefore, the semantics of β is the proposition $[\![\beta]\!] = \bigvee_{P \in \beta} [\![P]\!]$. Just as with simple policies, combined policy β_2 is as strong as combined policy β_1, written $\beta_1 \leq \beta_2$, if $[\![\beta_1]\!] \Rightarrow [\![\beta_2]\!]$. From the semantics, the $\leq$ ordering on policies can be lifted up to an ordering on combined policies by the following rule:

$$\frac{\forall P \in \beta_1.\ \exists P' \in \beta_2.\ P \leq P'}{\beta_1 \leq \beta_2}$$

Importantly, the set of all the combined policies form a lattice with the following *join* ($\sqcup$) and *meet* ($\sqcap$) operations:

$$\beta_1 \sqcup \beta_2 = \beta_1 \cup \beta_2$$
$$\beta_1 \sqcap \beta_2 = \{u_1 \vee u_2\!:\!p_1 \vee p_2 \mid u_1\!:\!p_1 \in \beta_1 \wedge u_2\!:\!p_2 \in \beta_2\}$$

The join and meet operations are sound with respect to the policy semantics, because it is easily shown that $[\![\beta_1 \sqcup \beta_2]\!] = [\![\beta_1]\!] \vee [\![\beta_2]\!]$ and $[\![\beta_1 \sqcap \beta_2]\!] = [\![\beta_1]\!] \wedge [\![\beta_2]\!]$.

Having a lattice of policies supports static program analysis [5]. For example, consider an addition expression $e_1 + e_2$. Let $A(e_1)$ and $A(e_2)$ represent the availability policies of the results of e_1 and e_2. Since the result $e_1 + e_2$ is available if and only if the results of e_1 and e_2 are both available, we have $A(e_1 + e_2) \leq A(e_1)$ and $A(e_1 + e_2) \leq A(e_2)$. Because the policies form a lattice, $A(e_1 + e_2) = A(e_1) \sqcap A(e_2)$ is the least restrictive availability policy we can assign to the result of e_1 and e_2. Dually, if $C(e_1)$ and $C(e_2)$ are the confidentiality policies of e_1 and e_2, then $C(e_1) \leq C(e_1 + e_2)$ and $C(e_2) \leq C(e_1 + e_2)$. The least restrictive confidentiality policy that can be assigned to the result $C(e_1 + e_2)$ is $C(e_1) \sqcup C(e_2)$.

2.8. Security labels

In general, a system will need to simultaneously enforce policies for confidentiality, integrity, and availability of the information it manipulates. These policies can be applied to information as *security labels*. A label ℓ is written as a triple $\langle \beta_C, \beta_I, \beta_A \rangle$, where β_C represents the (possibly combined) policy for confidentiality, β_I represents the integrity policy, and β_A represents availability. The notations $C(\ell)$, $I(\ell)$, and $A(\ell)$ represent the confidentiality, integrity, and availability components of ℓ.

For example, suppose expression e_1 has a security label ℓ_1, and e_2 has label ℓ_2. Then $e_1 + e_2$ has a label $\langle C(\ell_1) \sqcup C(\ell_2), I(\ell_1) \sqcap I(\ell_2), A(\ell_1) \sqcap A(\ell_2) \rangle$.

3. Applying policies to computation

In this paper, a system is modeled by a program with which users (including attackers) can interact only by affecting its inputs and observing its outputs. Security policies, including confidentiality, integrity and availability policies, are specified on the inputs and outputs of a program. This section shows this approach with a simple programming language.

3.1. Security model

Our goal is to ensure that a program does not allow attackers to violate its security policies at run time. A program itself has no influence on how its inputs are computed or how its outputs are used by external users. Therefore, a program is not responsible for the enforcement of the integrity and availability policies of its inputs, or the confidentiality policies of its outputs. For example, as shown in Figure 1, the owners of the availability policies on input i assume those policies are enforced. More generally, we have the following security assumption:

> **SA1** Confidentiality policies specified on inputs, and integrity and availability policies specified on outputs are already enforced.

We are interested in the security violations that may be caused by attackers, and we assume that the power of an attacker is limited to affecting the inputs and observing the outputs of a program. This leads to our second security assumption:

> **SA2** If an output is unavailable, then it is because the availability or value of some input is compromised by attackers.

By (SA1) and (SA2), an availability policy P specified on an output o can be enforced by a *noninterference* property [6]: the availability of o is not interfered by the availability of any input whose availability policy is not as strong as P, or the value of any input whose integrity policy is not as strong as P.

Indeed, suppose o is unavailable. By (SA2), it is because the availability or value of some input i is compromised by attackers. Without loss of generality, suppose the availability of i is compromised. Let P_i be the availability policy of i. By (SA1), P_i is enforced. Therefore, the unavailability of i implies that $[\![P_i]\!]$ is false, as discussed in Section 2.5. By the noninterference property, we have $P \leq P_i$, which is equivalent to $[\![P]\!] \Rightarrow [\![P_i]\!]$. Thus, $[\![P]\!]$ is false. Therefore, the unavailability of o implies that $[\![P]\!]$ is false. In other words, if $[\![P]\!]$ is true, then o must be available, which means that P is enforced on o.

3.2. The Aimp programming language

It is well known that confidentiality and integrity policies can be enforced by static program analyses that verify whether a program satisfies a noninterference property [19, 7, 20]. Since availability policies also correspond to a noninterference property in our security model, a static dependency analysis can be used to determine whether a system satisfies these policies. We now demonstrate this approach by formally representing the system as a program written in a security-typed imperative language called Aimp.

The Aimp language is a basic imperative language with assignments, sequential composition, conditionals and loops. The only non-standard construct in Aimp is a special value `none`, which is used to represent *unavailability*: a value is unavailable if and only if it is `none`. Intuitively, there are three rules on using the value `none`:

- The value `none` cannot appear in a program.
- The result of expression e is `none` if the evaluation of e depends on `none`.
- The execution of a statement gets stuck if the execution depends on `none`.

A program of Aimp is just a statement, and the state of a program is captured by a memory M that maps memory references (memory locations) to values. We assume that a memory is observable to users, so memory references can be used to represent I/O channels. A reference representing an input is called an *input reference*. If the value of an input reference is `none`, then the corresponding input is unavailable. Similarly, a reference representing an output is called an *output reference*. Suppose m is an output reference, then the corresponding output becomes available if m is assigned an integer value. An unassigned output reference represents an output still expected by users.

The syntax of Aimp is shown in Figure 2. Let m range over memory locations. In Aimp, values include integer n, and `none`. Expressions include integer n, dereference expression $!m$, and addition expression $e_1 + e_2$. Note that `none` is not a valid expression so that it cannot appear in a program. Statements include the empty statement `skip`, the as-

$$
\begin{array}{rcll}
\text{Values} & v & ::= & n \mid \texttt{none} \\
\text{Expressions} & e & ::= & n \mid\ !m \mid e_1 + e_2 \\
\text{Statements} & s & ::= & \texttt{skip} \mid m := e \mid s_1; s_2 \\
& & \mid & \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \\
& & \mid & \texttt{while}\ e\ \texttt{do}\ s
\end{array}
$$

Figure 2. Syntax of Aimp

signment statement $m := e$, sequential composition $s_1; s_2$, `if` and `while` statements.

Let β range over a lattice $\mathcal{L}$ of base labels, such as policies as defined in Section 2. The top and bottom elements of $\mathcal{L}$ are represented by $\top$ and $\bot$, respectively. The syntax for types in Aimp is shown as follows:

$$
\begin{array}{rcll}
\text{Base labels} & \beta & \in & \mathcal{L} \\
\text{Labels} & \ell, pc & ::= & \langle \beta_C, \beta_I, \beta_A \rangle \\
\text{Types} & \tau & ::= & \texttt{int}_\ell \mid \texttt{int}_\ell\ \texttt{ref} \mid \texttt{stmt}_\mathcal{R}
\end{array}
$$

In Aimp, the only data type is $\texttt{int}_\ell$, an integer type annotated with security label ℓ, which contains three combined policies as described in Section 2.

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

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

3.3. Operational semantics

The small-step operational semantics of Aimp is given in Figure 3. 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 $\langle s, M \rangle \longmapsto \langle s', M' \rangle$.

The evaluation rules (S1)–(S6) are standard for an imperative language. Rules (E1) and (E2) 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:

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

Rules (S1), (S4) and (S5) show that if the evaluation of configuration $\langle s, M \rangle$ depends on the result of an expression e, it must be the case that $\langle e, M \rangle \Downarrow n$. In other words, if $\langle e, M \rangle \Downarrow \texttt{none}$, the evaluation of $\langle s, M \rangle$ gets stuck.

$$
[E1] \quad \frac{m \in dom(M)}{\langle !m, M \rangle \Downarrow M(m)}
$$

$$
[E2] \quad \frac{\langle e_1, M \rangle \Downarrow v_1 \qquad \langle e_2, M \rangle \Downarrow v_2 \qquad v = v_1 + v_2}{\langle e_1 + e_2, M \rangle \Downarrow v}
$$

$$
[S1] \quad \frac{\langle e, M \rangle \Downarrow n}{\langle m := e, M \rangle \longmapsto \langle \texttt{skip}, M[m \mapsto n] \rangle}
$$

$$
[S2] \quad \frac{\langle s_1, M \rangle \longmapsto \langle s_1', M' \rangle}{\langle s_1; s_2, M \rangle \longmapsto \langle s_1'; s_2, M' \rangle}
$$

$$
[S3] \quad \langle \texttt{skip}; s, M \rangle \longmapsto \langle s, M \rangle
$$

$$
[S4] \quad \frac{\langle e, M \rangle \Downarrow n \qquad n > 0}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2, M \rangle \longmapsto \langle s_1, M \rangle}
$$

$$
[S5] \quad \frac{\langle e, M \rangle \Downarrow n \qquad n \leq 0}{\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2, M \rangle \longmapsto \langle s_2, M \rangle}
$$

$$
[S6] \quad \langle \texttt{while}\ e\ \texttt{do}\ s, M \rangle \longmapsto \langle \texttt{if}\ e\ \texttt{then}\ s; \texttt{while}\ e\ \texttt{do}\ s\ \texttt{else skip}, M \rangle
$$

Figure 3. Small-step operational semantics for Aimp

3.4. Examples

By its simplicity, the Aimp language helps focus on the basic constructs of an imperative language. Figure 4 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.

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 o depends on the

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

Figure 4. Examples

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.

4. Noninterference properties

This section formalizes the noninterference properties, including availability noninterference, that correspond to the security policies of Section 2. 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 an 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 that tries to read the input. Second, the attacker can try to affect the integrity of control flow and make the program diverge (fail to terminate). Intuitively, availability noninterference means that with all high-availability inputs available, equivalent high-integrity inputs will eventually result in equally available high-availability outputs.

The intuitive concepts of high and low security are based on the power of the potential attacker, which is represented by a base label L. Suppose the attacker is able to compromise principals $p_1, \ldots, p_n$, and that there exists a top principal (denoted by $*$) that acts for every principal. In the DLM, we have $L = \{* : p_1 \wedge \ldots \wedge p_n\}$, because $p_1 \wedge \ldots \wedge p_n$ is the most powerful principal that the attacker controls. Given a base label β, if $\beta \leq L$ then the label represents a low-security level that is not protected from the attacker. Otherwise, β is a high-security label.

For an imperative language, the inputs of a program is just the initial memory. However, what are the outputs of a program depends on the *observation model* of the language, which defines what aspects of a program execution are observable to external users. The observation model of Aimp is defined as follows:

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

Suppose s is a program, and M is the initial configuration. Based on the observation model, the outputs of s are a set $\mathcal{T}$ of finite traces of memories, and for any trace T in $\mathcal{T}$, there exists an evaluation $\langle s, M\rangle \longmapsto \langle s_1, M_1\rangle \longmapsto \ldots \longmapsto \langle s_n, M_n\rangle$ such that $T = [M, M_1, \ldots, M_n]$. Intuitively, every trace in $\mathcal{T}$ is the outputs observable to users at some point during the evaluation of $\langle s, M\rangle$, and $\mathcal{T}$ represents all the outputs of $\langle s, M\rangle$ observable to users. Since the Aimp language is deterministic, for any two traces in $\mathcal{T}$, it must be the case that one is a prefix of the other.

In the intuitive description of noninterference, equivalent low-confidentiality inputs can be represented by two memories whose low-confidentiality parts are indistinguishable. Suppose the typing information of a memory M is given by a typing assignment Γ. Then m belongs to the low-confidentiality part of M if $C(\Gamma(m)) \leq L$. Similarly, m is a high-integrity reference if $I(\Gamma(m)) \not\leq L$, and a high-availability reference if $A(\Gamma(m)) \not\leq L$. Given two memories M_1 and M_2, let $\Gamma \vdash M_1 \approx_{C\leq L} M_2$ denote that low-confidentiality parts of M_1 and M_2 are indistinguishable with respect to Γ, $\Gamma \vdash M_1 \approx_{I\not\leq L} M_2$ denote that high-integrity parts of M_1 and M_2 are indistinguishable with respect to Γ, and $\Gamma \vdash M_1 \approx_{A\not\leq L} M_2$ denote that the high-availability parts of M_1 and M_2 have indistinguishable availability with respect to Γ.

By the observation model of Aimp, a user cannot distinguish `none` from any other value. Let $v_1 \approx v_2$ denote that v_1 and v_2 are indistinguishable. Then $v_1 \approx v_2$ if and only if $v_1 = v_2$, $v_1 = \texttt{none}$ or $v_2 = \texttt{none}$. With these definitions, the three kinds of memory indistinguishability are defined as follows:

Definition 4.1 **($\Gamma \vdash M_1 \approx_{C\leq L} M_2$).** Suppose $\mathit{dom}(\Gamma) = \mathit{dom}(M_1) = \mathit{dom}(M_2)$. Then $\Gamma \vdash M_1 \approx_{C\leq L} M_2$ if for any $m \in \mathit{dom}(\Gamma)$, $C(\Gamma(m)) \leq L$ implies $M_1(m) \approx M_2(m)$.

Definition 4.2 **($\Gamma \vdash M_1 \approx_{I\not\leq L} M_2$).** Suppose $\mathit{dom}(\Gamma) = \mathit{dom}(M_1) = \mathit{dom}(M_2)$. Then $\Gamma \vdash M_1 \approx_{I\not\leq L} M_2$ if for any $m \in \mathit{dom}(\Gamma)$, $I(\Gamma(m)) \not\leq L$ implies $M_1(m) \approx M_2(m)$.

Definition 4.3 **($\Gamma \vdash M_1 \approx_{A\not\leq L} M_2$).** Suppose $\mathit{dom}(\Gamma) = \mathit{dom}(M_1) = \mathit{dom}(M_2)$. Then $\Gamma \vdash M_1 \approx_{A\not\leq L} M_2$ if for any $m \in \mathit{dom}(\Gamma)$, $A(\Gamma(m)) \not\leq L$ implies that $M_1(m) = \texttt{none}$ if and only if $M_2(m) = \texttt{none}$.

Based on the definitions of memory indistinguishability, we can define trace indistinguishability, which formalizes the notion of equivalent outputs in the intuitive description of noninterference. 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 stut-

tering, 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 Γ, let $\Gamma \vdash T_1 \approx_{C\le L} T_2$ and denote that the low-confidentiality parts of T_1 and T_2 are indistinguishable, and $\Gamma \vdash T_1 \approx_{I\not\le L} T_2$ denote that the high-integrity parts of T_1 and T_2 are indistinguishable. These two notions are defined as follows:

Definition 4.4 **(**$\Gamma \vdash T_1 \approx_{C\le L} T_2$**).** Given two traces T_1 and T_2, $\Gamma \vdash T_1 \approx_{C\le L} T_2$ if there exists $T'_1 = [M_1, \ldots, M_n]$ and $T'_2 = [M'_1, \ldots, M'_m]$ such that $T_1 \approx T'_1$, and $T_2 \approx T'_2$, and $\Gamma \vdash M_i \approx_{C\le L} M'_i$ for any i in $\{1, \ldots, \mathtt{min}(m, n)\}$.

Definition 4.5 **(**$\Gamma \vdash T_1 \approx_{I\not\le L} T_2$**).** Given two traces T_1 and T_2, $\Gamma \vdash T_1 \approx_{I\not\le L} T_2$ if there exists $T'_1 = [M_1, \ldots, M_n]$ and $T'_2 = [M'_1, \ldots, M'_m]$ such that $T_1 \approx T'_1$, and $T_2 \approx T'_2$, and $\Gamma \vdash M_i \approx_{I\not\le L} M'_i$ for any i in $\{1, \ldots, \mathtt{min}(m, n)\}$.

Note that two executions are indistinguishable if any two finite traces generated by those two executions are indistinguishable. Thus, we can still reason about the indistinguishability of two nonterminating executions, even though $\approx_{I\not\le L}$ and $\approx_{C\le L}$ 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 4.6 (Confidentiality noninterference). A program s has the *confidentiality noninterference* property w.r.t. a typing assignment Γ, written $\Gamma \vdash \mathtt{NI}_C(s)$, if for any two traces T_1 and T_2 generated by evaluating $\langle s,\, M_1\rangle$ and $\langle s,\, M_2\rangle$, where $\Gamma \vdash M_1 \approx_{C\le L} M_2$, we have $\Gamma \vdash T_1 \approx_{C\le L} T_2$.

Definition 4.7 (Integrity noninterference). A program s has the *integrity noninterference* property w.r.t. a typing assignment Γ, written $\Gamma \vdash \mathtt{NI}_I(s)$, if for any two traces T_1 and T_2 generated by evaluating $\langle s,\, M_1\rangle$ and $\langle s,\, M_2\rangle$, where $\Gamma \vdash M_1 \approx_{I\not\le L} M_2$, we have $\Gamma \vdash T_1 \approx_{I\not\le L} T_2$.

The intuitive description of availability noninterference has a premise that all the high-availability inputs are available. To formalize this premise, 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. Given an initial memory M, the premise that all the high-availability inputs are available can be represented by $\forall m, A(\Gamma(m)) \not\le L \wedge m \notin \mathcal{R} \Rightarrow M(m) \neq \mathtt{none}$. The formal definition of availability noninterference is given below:

Definition 4.8 (Availability noninterference). A program s has the *availability noninterference* property w.r.t. a typing assignment Γ and a set of unassigned output references $\mathcal{R}$, written $\Gamma\,;\mathcal{R} \vdash \mathtt{NI}_A(s)$, if for any two memories M_1, M_2, the following statements

- $\Gamma \vdash M_1 \approx_{I\not\le L} M_2$
- For any m in $\mathit{dom}(M_i)$ ($i \in \{1,2\}$), if $m \notin \mathcal{R}$ and $A(\Gamma(m)) \not\le L$, then $M_i(m) \neq \mathtt{none}$.
- $\langle s,\, M_i\rangle \longmapsto^* \langle s'_i,\, M'_i\rangle$ for $i \in \{1,2\}$

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

5. Security typing and soundness

The type system of Aimp is designed to ensure that any well-typed Aimp program satisfies the noninterference properties defined in Section 4. For confidentiality and integrity, the type system performs a standard static information flow analysis. 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: $\Gamma\,; \mathcal{R}\,; pc \vdash s : \mathtt{stmt}_{\mathcal{R}'}$, where Γ 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 $\Gamma\,; \mathcal{R} \vdash e : \tau$. Let the notation $A_\Gamma(\mathcal{R})$ denote $\bigsqcup_{m\in\mathcal{R}} A(\Gamma(m)$. The typing rules are shown in Figure 5.

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

Rule (REF) says that the type of a reference m is $\tau\ \mathtt{ref}$ 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. If the value of m is `none`, then dereferencing m will block the computation, causing the unassigned output references unavailable. Therefore, rule (DEREF) has the premise $A_\Gamma(\mathcal{R}) \le A(\ell)$, which ensures the availability of m is as high as the availability of any unassigned output reference. For example, in code segment (A) of Figure 4, the type system ensures that $A(\Gamma(m_o)) \le A(\Gamma(m_1))$ when checking $!m_1$.

[*INT*]	$\Gamma\,;\mathcal{R}\vdash n : \texttt{int}_\ell$
[*NONE*]	$\Gamma\,;\mathcal{R}\vdash \texttt{none} : \texttt{int}_\ell$
[*REF*]	$\dfrac{\Gamma(m)=\texttt{int}_\ell}{\Gamma\,;\mathcal{R}\vdash m : \texttt{int}_\ell\ \texttt{ref}}$
[*DEREF*]	$\dfrac{m\notin\mathcal{R}\qquad \Gamma(m)=\texttt{int}_\ell\qquad A_\Gamma(\mathcal{R})\leq A(\ell)}{\Gamma\,;\mathcal{R}\vdash !m : \texttt{int}_\ell}$
[*ADD*]	$\dfrac{\Gamma\,;\mathcal{R}\vdash e_1 : \texttt{int}_{\ell_1}\qquad \Gamma\,;\mathcal{R}\vdash e_2 : \texttt{int}_{\ell_2}}{\Gamma\,;\mathcal{R}\vdash e_1+e_2 : \texttt{int}_{\ell_1\sqcup\ell_2}}$
[*SKIP*]	$\Gamma\,;\mathcal{R}\,;pc\vdash \texttt{skip} : \texttt{stmt}_\mathcal{R}$
[*SEQ*]	$\dfrac{\Gamma\,;\mathcal{R}\,;pc\vdash s_1 : \texttt{stmt}_{\mathcal{R}_1}\quad \Gamma\,;\mathcal{R}_1\,;pc\vdash s_2 : \texttt{stmt}_{\mathcal{R}_2}}{\Gamma\,;\mathcal{R}\,;pc\vdash s_1;s_2 : \texttt{stmt}_{\mathcal{R}_2}}$
[*ASSIGN*]	$\dfrac{\Gamma\,;\mathcal{R}\vdash m : \texttt{int}_\ell\ \texttt{ref}\qquad \Gamma\,;\mathcal{R}\vdash e : \texttt{int}_{\ell'}\quad C(pc)\sqcup C(\ell')\leq C(\ell)\quad I(\ell)\leq I(pc)\sqcap I(\ell')}{\Gamma\,;\mathcal{R}\,;pc\vdash m:=e : \texttt{stmt}_{\mathcal{R}-\{m\}}}$
[*IF*]	$\dfrac{\Gamma\,;\mathcal{R}\vdash e : \texttt{int}_\ell\quad \Gamma\,;\mathcal{R}\,;pc\sqcup\ell\vdash s_i : \tau\quad i\in\{1,2\}}{\Gamma\,;\mathcal{R}\,;pc\vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \tau}$
[*WHILE*]	$\dfrac{\Gamma\vdash e : \texttt{int}_\ell\qquad \Gamma\,;\mathcal{R}\,;pc\sqcup\ell\vdash s : \texttt{stmt}_\mathcal{R}\quad A_\Gamma(\mathcal{R})\leq I(\ell)\sqcap I(pc)}{\Gamma\,;\mathcal{R}\,;pc\vdash \texttt{while}\ e\ \texttt{do}\ s : \texttt{stmt}_\mathcal{R}}$
[*SUB*]	$\dfrac{\Gamma\,;\mathcal{R}\,;pc\vdash s : \tau\qquad \Gamma\,;\mathcal{R}\,;pc\vdash \tau\leq\tau'}{\Gamma\,;\mathcal{R}\,;pc\vdash s : \tau'}$

Figure 5. Typing rules for Aimp

Rule (ADD) checks addition expressions. Let $\ell_1 \sqcup \ell_2$ be $\langle C(\ell_1)\sqcup C(\ell_2), I(\ell_1)\sqcap I(\ell_2), A(\ell_1)\sqcap A(\ell_2)\rangle$. As discussed in Section 2.8, the label of $e_1 + e_2$ is exactly $\ell_1 \sqcup \ell_2$ if e_i has the label ℓ_i for $i \in \{1, 2\}$.

Rule (SEQ) checks sequential statements. The premise $\Gamma\,;\mathcal{R}\,;pc \vdash s_1 : \texttt{stmt}_{\mathcal{R}_1}$ 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 time. Thus, $s_1; s_2$ has the same type as s_2.

Rule (ASSIGN) checks assignment statements. The assignment statement $m := e$ assigns the value of e to m, which is an explicit information flow from e to m. Therefore, both $C(\ell) \leq C(\Gamma(m))$ and $I(\Gamma(m)) \leq I(\ell)$ must hold to protect the confidentiality of e and the integrity of m. Whether this assignment to m happens depends on the control flow. Thus, this rule has the premises $C(pc) \leq C(\ell)$, which prevents attackers from inferring sensitive information about the control flow, and $I(\ell) \leq I(pc)$, which prevents attackers from compromising the integrity of m by affecting the control flow. Finally, when the statement terminates, m should be removed from the set of unassigned output references, and thus the statement has type $\texttt{stmt}_{\mathcal{R}-\{m\}}$.

Rule (IF) checks `if` statements. Consider the statement `if` e `then` s_1 `else` s_2. The value of e determines which branch is executed, so the program-counter labels for branches s_1 and s_2 subsume the label of e to protect e from implicit flows. As usual, the `if` statement has type τ if both s_1 and s_2 have type τ.

Rule (WHILE) checks `while` statements. A `while` statement may diverge, which affects the availability of any reference in $\mathcal{R}$. Therefore, if there is any high-availability reference in $\mathcal{R}$ ($A_\Gamma(\mathcal{R}) \not\leq L$), this rule needs to prevent attackers from affecting whether this `while` statement diverges or whether control flow reaches this statement. For example, consider the code segments (B) and (C) in Figure 4, in which $\mathcal{R} = \{m_o\}$. Suppose $A(\Gamma(m_o)) \not\leq L$. In code segment (B), the premise $A_\Gamma(\mathcal{R}) \leq I(\ell)$ of this rule ensures $I(\Gamma(m_1)) \not\leq L$, which means that attackers cannot affect the value of m_1, and whether the `while` statement diverges.

In code segment (C), the premise $A_\Gamma(\mathcal{R}) \leq I(pc)$ of rule (WHILE) guarantees $I(\Gamma(m_1)) \not\leq L$ since $I(pc) \leq I(\Gamma(m_1))$. Thus, attackers cannot affect which branch of the `if` statement would be taken, i.e. whether control flow reaches the `while` statement.

Rule (SUB) is the standard subsumption rule. Let $\Gamma\,;\mathcal{R}\,;pc \vdash \tau \leq \tau'$ denote that τ is a subtype of τ' with respect to the typing environment $\Gamma\,;\mathcal{R}\,;pc$. Suppose $\Gamma\,;\mathcal{R}\,;pc \vdash \texttt{stmt}_{R'} \leq \texttt{stmt}_{R''}$ and $\Gamma\,;\mathcal{R}\,;pc \vdash s : \texttt{stmt}_{R'}$. Based on rule (SUB), we have $\Gamma\,;\mathcal{R}\,;pc \vdash s : \texttt{stmt}_{R''}$. In addition, suppose $\mathcal{R}'$ is exactly the set of unassigned outputs after the execution of s. Then $\Gamma\,;\mathcal{R}\,;pc \vdash s : \texttt{stmt}_{R''}$ implies $\mathcal{R}' \subseteq \mathcal{R}'' \subseteq \mathcal{R}$ by definition, since $\mathcal{R}''$ needs to contain all unassigned output references when s terminates.

Consider the assignment $m_o := 1$ in code segment (D) of Figure 4. For the branch `skip` of the `if` statement, we have $\Gamma\,;\{m_o\}\,;pc \vdash \texttt{skip} : \texttt{stmt}_{\{m_o\}}$. Thus, by rule (IF), $\Gamma\,;\{m_o\}\,;pc \vdash m_o := 0 : \texttt{stmt}_{\{m_o\}}$ needs to hold. Therefore, $\Gamma\,;\{m_o\}\,;pc \vdash \texttt{stmt}_\emptyset \leq \texttt{stmt}_{\{m_o\}}$ is needed. Because the availability of m_o depends on which branch is taken, we need to ensure $A(\Gamma(m_o)) \leq I(\Gamma(m_1))$. By rule (IF), $I(pc) \leq I(\Gamma(m_1))$. Therefore, $A(\Gamma(m_o)) \leq I(\Gamma(m_1))$ can be ensured by imposing the constraint $A(\Gamma(m)) \leq I(pc)$ on $\Gamma\,;\mathcal{R}\,;pc \vdash \texttt{stmt}_{\mathcal{R}'} \leq \texttt{stmt}_{\mathcal{R}''}$ for $m \in \mathcal{R}'' - \mathcal{R}'$.

The type system of Aimp contains only one subtyping

rule (ST), which is based on the above discussion.

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

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 are included in Appendix A.

Theorem 5.1 (Subject reduction). Suppose $\Gamma\,;\mathcal{R}\,;pc \vdash s : \tau$, and $dom(\Gamma) = dom(M)$. If $\langle s,\, M\rangle \longmapsto \langle s',\, M'\rangle$, then there exists $\mathcal{R}'$ such that $\Gamma\,;\mathcal{R}'\,;pc \vdash s' : \tau$, and $\mathcal{R}' \subseteq \mathcal{R}$, and for any $m \in \mathcal{R} - \mathcal{R}'$, $M'(m) \neq \texttt{none}$.

Theorem 5.2 (Noninterference). If $\Gamma\,;\mathcal{R}\,;pc \vdash s : \tau$, then $\Gamma \vdash \texttt{NI}_C(s)$, $\Gamma \vdash \texttt{NI}_I(s)$ and $\Gamma\,;\mathcal{R} \vdash \texttt{NI}_A(s)$.

6. Extensions

This section describes two language extensions that can be used to reduce availability dependencies and allow a program to use low-availability data in a more flexible and practical way.

6.1. Timeout

Timeouts can effectively turn a blocking operation into a non-blocking operation, and thus provide a strong availability guarantee for a computation that uses low-availability inputs. To support timeouts, we introduce two syntax extensions to Aimp: timed integer values and a race expression.

$$\begin{array}{rrcl} \text{Values} & v & ::= & \ldots \mid \langle n,\, t\rangle \\ \text{Expressions} & e & ::= & \ldots \mid e_1\#e_2 \end{array}$$

A timed integer $\langle n,\, t\rangle$ is similar to integer n except that it would take t units of time to use this value. A race expression $e_1\#e_2$ evaluates e_1 and e_2 at the same time and returns the result of the expression that finishes first. If both e_1 and e_2 finish at the same time, the result of e_1 would be the final result. Suppose we want to set a timeout t for expression e such that if the evaluation of e does not finish in t units of time, a default value n is returned as the result of e. This can be implemented by the expression $e\#\langle n,\, t\rangle$.

Using the timeout mechanism, the following program implements an auction for two clients Alice and Bob. Reference m_A represents Alice's bid, and Alice has 30 units of time to make a bid, otherwise time runs out, and 0 is returned as her bid. Similarly, Bob also has 30 units of time to make a bid. Even though the result of this auction depends on the bids of Alice and Bob, the availability of the auction result is not affected by them.

```
m1 := !mA#⟨0, 30⟩;
m2 := !mB#⟨0, 30⟩;
if (!m1 ≥ !m2) mo := !m1
else mo := !m2
```

6.1.1. Operational semantics

Note that value n can be treated as a syntax sugar for $\langle n,\, 0\rangle$. As a result, the evaluation rules in Figure 3 can be adapted to the timeout extension by replacing any occurrence of $\langle e,\, M\rangle \Downarrow n$ with a more general form $\langle e,\, M\rangle \Downarrow \langle n,\, t\rangle$. For example, the adapted rule (S1) is shown below:

$$[S1]\quad \frac{\langle e,\, M\rangle \Downarrow \langle n,\, t\rangle}{\langle m := e,\, M\rangle \longmapsto \langle \texttt{skip},\, M[m \mapsto n]\rangle}$$

In addition, the formula for computing $v_1 + v_2$ in rule (E2) also needs to be adapted to this more general form of values:

$$v_1 + v_2 = \begin{cases} \langle n_1 + n_2,\, t_1 + t_2\rangle & \text{if } \forall i \in \{1,2\}.\, v_i = \langle n_i,\, t_i\rangle \\ \texttt{none} & \text{if } v_1 = \texttt{none} \text{ or } v_2 = \texttt{none} \end{cases}$$

The operational semantics of the race expression is given by the following rules (E3)–(E5). Suppose e_1 and e_2 are evaluated to $\langle n_1,\, t_1\rangle$ and $\langle n_2,\, t_2\rangle$, which means evaluating e_1 and e_2 takes t_1 and t_2 units of time, respectively. Thus, if $t_1 \leq t_2$ (E3), the result of e_1 should be the final result, and if $t_1 > t_2$ (E4), $\langle n_2,\, t_2\rangle$ is the final result. Rule (E5) applies when only the result of one expression e_i is available.

$$[E3]\quad \frac{\langle e_1,\, M\rangle \Downarrow \langle n_1,\, t_1\rangle \quad \langle e_2,\, M\rangle \Downarrow \langle n_2,\, t_2\rangle \quad t_1 \leq t_2}{\langle e_1\#e_2,\, M\rangle \Downarrow \langle n_1,\, t_1\rangle}$$

$$[E4]\quad \frac{\langle e_1,\, M\rangle \Downarrow \langle n_1,\, t_1\rangle \quad \langle e_2,\, M\rangle \Downarrow \langle n_2,\, t_2\rangle \quad t_1 > t_2}{\langle e_1\#e_2,\, M\rangle \Downarrow \langle n_2,\, t_2\rangle}$$

$$[E5]\quad \frac{\langle e_i,\, M\rangle \Downarrow \langle n,\, t\rangle \quad \langle e_j,\, M\rangle \Downarrow \texttt{none} \quad \{i,j\} = \{1,2\}}{\langle e_1\#e_2,\, M\rangle \Downarrow \langle n,\, t\rangle}$$

6.1.2. Typing

The race expression is essential for the timeout mechanism to provide strong availability guarantees. Consider a race expression $e_1\#e_2$. According to rule (E5), the result of expression $e_1\#e_2$ is available as long as the result of e_1 or e_2 is available. Therefore, the availability of e is as high as the availability of e_1 and e_2. Let $A(e)$ represent the availability label of e. Then we have $A(e_1\#e_2) = A(e_1) \sqcup A(e_2)$. On the other hand, the value of $e_1\#e_2$ depends on the availability and timing of both e_1 and e_2. Consequently, an attacker can try to compromise the integrity of $e_1\#e_2$ by compromising the availability or timing of e_1 or e_2. Intuitively, the race expression trades integrity for availability.

To take into account attacks on timing, a security label may contain a new base label component β_{IT} (IT stands for integrity of timing), and $IT(\ell)$ is used to retrieve the component in ℓ. Suppose expression e has a label ℓ, and the result of e is $\langle n,\, t\rangle$. Then an attacker with a security level L can affect the value of t if and only if $IT(\ell) \leq L$.

Suppose e_1 and e_2 have type $\texttt{int}_{\ell_1}$ and $\texttt{int}_{\ell_2}$, respectively. Then $e_1 \# e_2$ has type $\texttt{int}_{\ell_1 \# \ell_2}$, where $\ell_1 \# \ell_2$ is a label computed from ℓ_1 and ℓ_2. Based on the above discussion, we have
$A(\ell_1 \# \ell_2) = A(\ell_1) \sqcup A(\ell_2)$
$I(\ell_1 \# \ell_2) = I(\ell_1) \sqcap I(\ell_2) \sqcap A(\ell_1) \sqcap A(\ell_2) \sqcap IT(\ell_1) \sqcap IT(\ell_2)$

By rule (E5), if the result of $e_1 \# e_2$ is $\langle n, t \rangle$, the value of t may be affected by the availability of e_1 and e_2. Therefore,

$$IT(\ell_1 \# \ell_2) = IT(\ell_1) \sqcap IT(\ell_2) \sqcap A(\ell_1) \sqcap A(\ell_2)$$

As usual, $C(\ell_1 \# \ell_2) = C(\ell_1) \sqcup C(\ell_2)$, since the result of $e_1 \# e_2$ depends on the results of both e_1 and e_2. With these formulas for computing $\ell_1 \# \ell_2$, the typing rule for checking the race expression is straightforward:

$$[\mathit{RACE}] \quad \frac{\Gamma\,;\mathcal{R} \vdash e_1 : \texttt{int}_{\ell_1} \qquad \Gamma\,;\mathcal{R} \vdash e_2 : \texttt{int}_{\ell_2}}{\Gamma\,;\mathcal{R} \vdash e_1 \# e_2 : \texttt{int}_{\ell_1 \# \ell_2}}$$

Because the timeout mechanism trades integrity for availability and allows attackers to compromise the integrity of an output by affecting the availability or timing of an input, the definition of integrity noninterference needs to be adapted to these new risks. Intuitively, the adapted integrity noninterference would require two sets of inputs M_1 and M_2 to generate equivalent high-integrity outputs, if the high-integrity parts, the availability of the high-availability parts and the timing of the high-integrity-of-timing parts of M_1 and M_2 are indistinguishable. The formal definition is given below, following the definition of the memory indistinguishability with respect to the integrity of timing:

Definition 6.1 **($\Gamma \vdash M_1 \approx_{IT \not\leq L} M_2$).** Suppose $\mathit{dom}(\Gamma) = \mathit{dom}(M_1) = \mathit{dom}(M_2)$. Then $\Gamma \vdash M_1 \approx_{IT \not\leq L} M_2$ means for any $m \in \mathit{dom}(\Gamma)$, $IT(\Gamma(m)) \not\leq L$ and $M_1(m) = \langle n_1, t_1 \rangle$ and $M_2(m) = \langle n_1, t_2 \rangle$ imply $t_1 = t_2$.

Definition 6.2 (Integrity noninterference). A program s has the *integrity noninterference* property w.r.t. a typing assignment Γ, written $\Gamma \vdash \texttt{NI}_I(s)$, if for any two traces T_1 and T_2 generated by evaluating $\langle s, M_1 \rangle$ and $\langle s, M_2 \rangle$, we have that $\Gamma \vdash M_1 \approx_{I \not\leq L} M_2$, $\Gamma \vdash M_1 \approx_{A \not\leq L} M_2$ and $\Gamma \vdash M_1 \approx_{IT \not\leq L} M_2$ imply $\Gamma \vdash T_1 \approx_{I \not\leq L} T_2$.

6.2. Run-time reference generation

For a program s in Aimp, the set of outputs that s is expected to generate are statically determined by a set of references $\mathcal{R}$. However, in some realistic applications, an output may be expected only after control reaches certain program points. For example, consider a simple service that responds to the request from a client. The response is expected only after the service receives a client request. To express such kind of availability requirements, we extend Aimp with a `new` statement that creates a new reference in memory. Intuitively, the output represented by this reference is expected by users only after the point where it is created. The syntax of this extension is shown below:

$$\begin{array}{rrcl} \text{References} & r & ::= & m \mid x \\ \text{Expressions} & e & ::= & \ldots \mid\ !r \\ \text{Statements} & s & ::= & \ldots \mid r := e \\ & & \mid & \texttt{new}\ x\,{:}\,\ell_x = \texttt{ref}(\ell)\ \texttt{in}\ s \end{array}$$

The name x is used to range over a set of reference variables. The `new` statement $\texttt{new}\ x : \ell_x = \texttt{ref}(\ell)\ \texttt{in}\ s$ creates a new reference m with type $\texttt{int}_\ell\ \texttt{ref}$, substitutes the occurrences of x in s with m, and then executes s. Now a reference r may be a memory location m or a variable x. Accordingly, the dereference expression and the assignment statement have the form $!r$ and $r := e$, respectively.

Because the memory is observable to users, the creation of a new reference is an observable event and may be used as an information channel. In a `new` statement $\texttt{new}\ x : \ell_x = \texttt{ref}(\ell)\ \texttt{in}\ s$, the label ℓ_x is used to specify the security level of this event and control this new kind of implicit flows. For example, any user with a confidentiality level not as high as $C(\ell_x)$ should not observe the creation of the reference.

Consider the simple service example. In Aimp, a straightforward implementation is shown below:

```
m := !m1;
m2 := 1;
```

where m_1 represents the client request, and m_2 represents the output generated by the server in response to the client request. This implementation is problematic because the availability of m_2 depends on that of m_1. In practice, we can imagine that the availability labels of m_1 and m_2 are `{*:client}` and `{*:server}`, respectively, where `client` represents the client machine, and `server` represents the server machine. However, in general, `client` does not act for `server`, and thus `{*:server}` $\not\leq$ `{*:client}`. Therefore, the above program is not well-typed in practice.

With the `new` statement, the simple service can be implemented by the following program in which the server response is represented by a reference variable x instead of a memory location. Since x is created after m_1 is dereferenced, the availability of x does not depend on that of m_1.

```
m := !m1;
new x:ℓx = ref(⟨βC, βI, {*:server}A⟩) in
  x := 1;
```

6.2.1. Operational semantics

Formally, the following rule is used to evaluate the `new` statement:

$$[S7] \quad \frac{m = \mathit{newloc}(M, \ell_x)}{\langle \texttt{new}\ x\,{:}\,\ell_x = \texttt{ref}(\ell)\ \texttt{in}\ s,\ M \rangle \longmapsto \langle s[m/x],\ M[m \mapsto \texttt{none}] \rangle}$$

The function $newloc(M, \ell_x)$ deterministically returns a fresh reference m such that $m \notin dom(M)$. The observability and integrity of the newly created reference are specified by a label ℓ_x. To associate a memory reference with its label, we assume there exists a map Σ from the memory space $\mathcal{M}$ (an infinite set of memory locations) to labels. Given a label ℓ, let $\mathcal{M}_\ell = \{m \mid m \in \mathcal{M} \wedge \Sigma(m) = \ell\}$. In addition, we assume that for any ℓ, $\mathcal{M}_\ell$ is infinite. The function $newloc(M, \ell_x)$ deterministically picks a reference m from $\mathcal{M}_{\ell_x}$ such that $m \notin dom(M)$.

The definitions of memory indistinguishability need to take into account the reference labels, which determine the observability and integrity of references themselves. Due to the space limit, we only give the new definition for $\Gamma \vdash M_1 \approx_{A \not\leq L} M_2$ below. Compared to Definition 4.3, this definition does not require $dom(M_1) = dom(M_2)$, but $I(\Sigma(m)) \not\leq L$ implies $m \in dom(M_1) \cap dom(M_2)$. The new definitions for $\Gamma \vdash M_1 \approx_{I \not\leq L} M_2$ and $\Gamma \vdash M_1 \approx_{C \leq L} M_2$ have similar adjustments.

Definition 6.3 ($\Gamma \vdash M_1 \approx_{A \not\leq L} M_2$). Suppose $dom(\Gamma) = dom(M_1) \cup dom(M_2)$. Then $\Gamma \vdash M_1 \approx_{A \not\leq L} M_2$ if for any $m \in dom(\Gamma)$ such that $I(\Sigma(m)) \not\leq L$, we have $m \in dom(M_1) \cap dom(M_2)$, and $A(\Gamma(m)) \not\leq L$ implies that $M_1(m) = \texttt{none}$ if and only if $M_2(m) = \texttt{none}$.

Note that we assume that for any reference m in the initial memory of a program, $\Sigma(m) = \langle \bot_C, \top_I, \top_A \rangle$. As a result, if a program s does not contain any `new` statement, these new definitions of memory indistinguishability , when applied to the traces of s, are consistent with those original definitions in Section 3.

Typing rules for this extension are found in Appendix B.

7. Related work

There has been much research on ensuring high availability of a computer platform, or guaranteeing a server to carry out the computation requests from clients. Most of these work falls in two main categories: one is aimed at tolerating server-side failures, usually by using some replication techniques [16, 10, 2]; the other deals with faulty clients and defends denial of service attacks [3]. This work is concerned with the availability risks inherent to the computation that may process untrusted inputs, while the computation platform is assumed available.

Lamport first introduced the concepts of *safety* and *liveness* properties [9]. Being available is often characterized as a liveness property, which informally means "something good will eventually happen". In general, verifying whether a program will eventually produce an output is equivalent to solving the halting problem, and thus incomputable for a Turing-complete language. In this work, we propose a security model in which an availability policy can be enforced by a noninterference property [6]. It is well known that a noninterference property is not a property on traces [11], and unlike safety or liveness properties, cannot be specified by a trace set. However, a noninterference property can be treated as a property on pairs of traces. For example, consider a trace pair $\langle T_1, T_2 \rangle$. It has the confidentiality noninterference property if the first elements of T_1 and T_2 are distinguishable, or T_1 and T_2 are indistinguishable to low-confidentiality users. Therefore, a noninterference property can be represented by a set of trace pairs S, and a program satisfies the property if all the pairs of traces produced by the program belong to S. Interestingly, with respect to a trace pair, the confidentiality and integrity noninterference properties have the informal meaning of safety properties ("something bad will not happen"), and availability noninterference takes on the informal meaning of liveness.

Language-based information flow control techniques [5, 15, 19, 7, 20, 14, 1] can be used to enforce noninterference. But they mainly dealt with confidentiality and integrity. Our work focuses on applying the security-typed language approach to enforcing availability policies.

Myers and Liskov proposed the decentralized label model for specifying information flow policies [12]. This paper generalizes the DLM to provide a unified framework for specifying confidentiality, integrity and availability policies. In this framework, it is possible to compare an availability policy with an integrity policy, or a confidentiality policy with an integrity policy, making it convenient to study the interactions between different aspects of security.

Volpano and Smith introduced the notion of *termination agreement* [18], which requires two executions indistinguishable to low-confidentiality users to both terminate or both diverge. The integrity dual of termination agreement can be viewed as a special case of the availability noninterference in which termination is treated as the only output of a program.

8. Conclusions

This paper makes three contributions. First, it proposes a way to specify availability policies as an extension to the decentralized label model, including the added expressive power of conjunctive and disjunctive principals and a new semantics for policies and labels. Second, the paper presents a simple language that can explicitly specify security policies as type annotations and has a security type system to reason about end-to-end availability policies, along with confidentiality and integrity policies. Third, the paper formally defines an end-to-end availability property in terms of program traces and shows that the security type system enforces this property. As far as we know, this is the first security type system for reasoning about availability.

Acknowledgements

The authors would like to thank Andrei Sabelfeld, Steve Chong and Lorenzo Alvisi for their insightful suggestions and comments on this work. Nate Nystrom, Michael Clarkson, Kevin O'Neill and Jed Liu also helped improve the presentation of this work.

References

[1] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 2002.

[2] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proc. 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.

[3] CERT. Denial of service attacks. http://www.cert.org/tech_tips/denial_of_service.html, June 2001.

[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] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[7] 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.

[8] Ari Juels and John Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of NDSS'99 (Network and Distributed System Security Symposium)*, pages 151–165, 1999.

[9] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[10] 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.

[11] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Security and Privacy*, pages 79–93, May 1994.

[12] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, Saint-Malo, France, 1997.

[13] 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.

[14] 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.

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

[16] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[17] Trivedi Kishor Shridharbhai. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Englewood Cliffs, N.J. : Prentice-Hall, 1st edition, 1982.

[18] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proc. 10th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.

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

[20] Steve Zdancewic and Andrew C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2–3):209–234, September 2002.

A. Noninterference proof

The noninterference result for Aimp is proved by extending the language to a new language AimpX. Each configuration C in AimpX encodes two Aimp configurations C_1 and C_2. Moreover, the operational semantics of AimpX is consistent with that of Aimp in the sense that the result of evaluating C is an encoding of the results of evaluating C_1 and C_2 in Aimp. The type system of AimpX can guarantee that C is well-typed only if the low-confidentiality or high-integrity parts of C_1 and C_2 are equivalent. Intuitively, if the result of C is well-typed, then the results of evaluating C_1 and C_2 should also have equivalent low-confidentiality or high-integrity parts. Therefore, the preservation of type soundness in an AimpX evaluation implies the preservation of low-confidentiality or high-integrity equivalence between two Aimp evaluations. Thus, to prove the confidentiality and integrity noninterference theorems of Aimp, we only need to prove the subject reduction theorem of AimpX. This proof technique was first used by Pottier and Simonet to prove the noninterference result of a security-typed ML-like language [14].

Interestingly, the availability noninterference theorem of Aimp can by proved by a *progress* property of AimpX's type system. This appendix details the syntax and semantic extensions of AimpX, proves the key subject reduction and progress theorems of AimpX, and then proves the noninterference theorem of Aimp.

A.1. Syntax extensions

The syntax extensions of AimpX include the bracket constructs, which are composed of two Aimp terms and used to

capture the differences between two Aimp configurations.

$$\begin{array}{rcll} \text{Values} & v & ::= & \dots \mid (v_1 \mid v_2) \\ \text{Expressions} & e & ::= & \dots \mid (n_1 \mid n_2) \\ \text{Statements} & s & ::= & \dots \mid (s_1 \mid s_2) \end{array}$$

The bracket constructs cannot be nested, so the subterms of a bracket construct must be Aimp terms. Given an AimpX statement s, let $\lfloor s \rfloor_1$ and $\lfloor s \rfloor_2$ represent the two Aimp statements that s encodes. The projection functions satisfy $\lfloor (s_1 \mid s_2) \rfloor_i = s_i$ and are homomorphisms on other statement and expression forms. An AimpX memory M maps references to AimpX values that encode two Aimp values. Thus, the projection function can be defined on memories too. For $i \in \{1, 2\}$, $dom(\lfloor M \rfloor_i) = dom(M)$, and for any $m \in dom(M)$, $\lfloor M \rfloor_i(m) = \lfloor M(m) \rfloor_i$.

Since an AimpX term effectively encodes two Aimp terms, the evaluation of a AimpX term can be projected into two Aimp evaluations. An evaluation step of a bracket statement $(s_1 \mid s_2)$ is an evaluation step of either s_1 or s_2, and s_1 or s_2 can only access the corresponding projection of the memory. Thus, the configuration of AimpX has an index $i \in \{\bullet, 1, 2\}$ that indicates whether the term to be evaluated is a subterm of a bracket expression, and if so, which branch of a bracket the term belongs to. For example, the configuration $\langle s, M \rangle_1$ means that s belongs to the first branch of a bracket, and s can only access the first projection of M. We write "$\langle s, M \rangle$" for "$\langle s, M \rangle_\bullet$", which means s does not belong to any bracket.

The operational semantics of AimpX is shown in Figure 6. It is based on the semantics of Aimp and contains some new evaluation rules (S10–S12) for manipulating bracket constructs. Rules (E1) and (S1) are modified to access the memory projection corresponding to index i. The rest of the rules in Figure 3 are adapted to AimpX by indexing each configuration with i. The following adequacy and soundness lemmas state that the operational semantics of AimpX is adequate to encode the execution of two Aimp terms.

Let the notation $\langle s, M \rangle \longmapsto^T \langle s', M' \rangle$ denote that $\langle s, M \rangle \longmapsto \langle s_1, M_1 \rangle \longmapsto \dots \longmapsto \langle s_n, M_n \rangle \longmapsto \langle s', M' \rangle$ and $T = [M, M_1, \dots, M_n, M']$, or $s = s'$ and $M = M'$ and $T = [M]$. In addition, let $|T|$ denote the length of T, and $T_1 \oplus T_2$ denote the trace obtained by concatenating T_1 and T_2. Suppose $T_1 = [M_1, \dots, M_n]$ and $T_2 = [M'_1, \dots, M'_m]$. If $mem_n = M'_1$, then $T_1 \oplus T_2 = [M_1, \dots, M_n, M'_2, \dots, M'_m]$. Otherwise, $T_1 \oplus T_2 = [M_1, \dots, M_n, M'_1, \dots, M'_m]$.

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

Proof. By induction on the structure of e. □

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

Proof. By induction on the structure of s. □

Lemma A.3 (One-step adequacy). If for $i \in \{1, 2\}$, $\langle s_i, M_i \rangle \longmapsto \langle s'_i, M'_i \rangle$ is an evaluation in Aimp, and there exists $\langle s, M \rangle$ in AimpX such that $\lfloor s \rfloor_i = s_i$ and $\lfloor M \rfloor_i = M_i$, then there exists $\langle s', M' \rangle$ such that $\langle s, M \rangle \longmapsto^T \langle s', M' \rangle$, and one of the following statements holds:

i. For $i \in \{1, 2\}$, $\lfloor T \rfloor_i \approx [M_i, M'_i]$ and $\lfloor s' \rfloor_i = s'_i$.
ii. For $\{j, k\} = \{1, 2\}$, $\lfloor T \rfloor_j \approx [M_j]$ and $\lfloor s' \rfloor_j = s_j$, and $\lfloor T \rfloor_k \approx [M_k, M'_k]$ and $\lfloor s' \rfloor_k = s'_k$.

Proof. By induction on the structure of s.

- s is `skip`. Then s_1 and s_2 are also `skip` and cannot be further evaluated. Therefore, the lemma is correct because its premise does not hold.
- s is $m := e$. In this case, s_i is $m := \lfloor e \rfloor_i$, and $\langle m := \lfloor e \rfloor_i, M_1 \rangle \longmapsto \langle \texttt{skip}, M_i[m \mapsto v_i] \rangle$ where $\langle \lfloor e \rfloor_i, M_1 \rangle \Downarrow v_i$. By induction, we have $\langle e, M \rangle \Downarrow v$ and $\lfloor v \rfloor_i = v_i$. Therefore, $\langle m := e, M \rangle \longmapsto \langle \texttt{skip}, M[m \mapsto v] \rangle$. Since $\lfloor M \rfloor_i = M_i$, we have $\lfloor M[m \mapsto v] \rfloor_i = M_i[m \mapsto \lfloor v \rfloor_i]$. Finally, we have $\lfloor s' \rfloor_i = s'_i = \texttt{skip}$ for $i \in \{1, 2\}$.

[E1] $$\frac{\pi_i\, M(m) = v \qquad v \neq \texttt{none}}{\langle !m,\, M \rangle_i \Downarrow v}$$

[E2] $$\frac{\langle e_1,\, M \rangle_i \Downarrow v_1 \qquad \langle e_2,\, M \rangle_i \Downarrow v_2 \qquad v = v_1 \oplus v_2}{\langle e_1 + e_2,\, M \rangle \Downarrow v}$$

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

[S10] $$\frac{\langle e,\, M \rangle \Downarrow (n_1 \mid n_2)}{\begin{array}{l}\langle \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2,\, M \rangle \longmapsto \\ \quad \langle (\texttt{if}\ n_1\ \texttt{then}\ \lfloor s_1 \rfloor_1\ \texttt{else}\ \lfloor s_2 \rfloor_1 \mid \\ \quad\ \ \texttt{if}\ n_2\ \texttt{then}\ \lfloor s_1 \rfloor_2\ \texttt{else}\ \lfloor s_2 \rfloor_2),\, M \rangle\end{array}}$$

[S11] $$\frac{\langle s_i,\, M \rangle_i \longmapsto \langle s'_i,\, M' \rangle_i \qquad s_j = s'_j \qquad \{i, j\} = \{1, 2\}}{\langle (s_1 \mid s_2),\, M \rangle \longmapsto \langle (s'_1 \mid s'_2),\, M' \rangle}$$

[S12] $\langle (\texttt{skip} \mid \texttt{skip}),\, M \rangle \longmapsto \langle \texttt{skip},\, M \rangle$

[Auxiliary functions]

$$\begin{array}{ll} v[v'/\pi_\bullet] = v' & \pi_\bullet\ v = v \\ v[v'/\pi_1] = (v' \mid \lfloor v \rfloor_2) & \pi_1\ v = \lfloor v \rfloor_1 \\ v[v'/\pi_2] = (\lfloor v \rfloor_1 \mid v') & \pi_2\ v = \lfloor v \rfloor_2 \end{array}$$

Figure 6. The operational semantics of AimpX

- s is $\texttt{if}\ e\ \texttt{then}\ s''_1\ \texttt{else}\ s''_2$. Suppose $\langle e, M\rangle \Downarrow n$. Then $\langle s, M\rangle \longmapsto \langle s''_j, M\rangle$ for some j in $\{1,2\}$, and s_i is $\texttt{if}\ e\ \texttt{then}\ \lfloor s''_1\rfloor_i\ \texttt{else}\ \lfloor s''_2\rfloor_i$ for $i \in \{1,2\}$. Therefore, $\langle s_i, \lfloor M\rfloor_i\rangle \longmapsto \langle\lfloor s''_j\rfloor_i, \lfloor M\rfloor_i\rangle$ holds because $\langle e, \lfloor M\rfloor_i\rangle \Downarrow n$. It is clear that for $i \in \{1,2\}$, $\lfloor T\rfloor_i = [\lfloor M\rfloor_i, \lfloor M\rfloor_i] = [M_i, M_i]$ and $\lfloor s'\rfloor_i = \lfloor s''_j\rfloor_i = s'_i$.

  Suppose $\langle e, M\rangle \Downarrow (n_1 \mid n_2)$. Then $\langle s, M\rangle \longmapsto^T \langle(\lfloor s''_{j_1}\rfloor_1 \mid \lfloor s''_{j_2}\rfloor_2), M\rangle$ where $j_1, j_2 \in \{1,2\}$. Because $\langle e, M\rangle \Downarrow (n_1 \mid n_2)$, we have $\langle\lfloor e\rfloor_i, \lfloor M\rfloor_i\rangle \Downarrow n_i$ for $i \in \{1,2\}$, which implies $\langle s_i, M_i\rangle \longmapsto \langle s''_{j_i}, M_i\rangle$ for $i \in \{1,2\}$. Therefore, $\lfloor T\rfloor_i \approx \lfloor M\rfloor_i \approx [M_i, M_i]$ and $\lfloor s\rfloor_i = s'_i = \lfloor s''_{j_i}\rfloor_i$ hold for $i \in \{1,2\}$.

- s is $\texttt{while}\ e\ \texttt{do}\ s''$. Then $\langle s, M\rangle \longmapsto^* \langle\texttt{if}\ e\ \texttt{then}\ s; \texttt{while}\ e\ \texttt{do}\ s''\ \texttt{else}\ \texttt{skip}, M\rangle$. Furthermore, $\langle s_i, \lfloor M\rfloor_i\rangle \longmapsto^*$ $\langle\texttt{if}\ \lfloor e\rfloor_i \texttt{then} \lfloor s\rfloor_i; \texttt{while} \lfloor e\rfloor_i \texttt{do} \lfloor s''\rfloor_i \texttt{elseskip}, \lfloor M\rfloor_i\rangle$ for $i \in \{1,2\}$. It is clear that $\lfloor T\rfloor_i = [\lfloor M\rfloor_i, \lfloor M\rfloor_i]$ and $\lfloor s'\rfloor_i = s'_i$ hold for $i \in \{1,2\}$.

- s is $s_3; s_4$. There are three cases:
  - s_3 is $\texttt{skip}$ or $(\texttt{skip} \mid \texttt{skip})$. Then $\langle s, M\rangle \longmapsto^T \langle s_4, M\rangle$, and $T \approx [M]$. For $i \in \{1,2\}$, since $s_i = \texttt{skip}; \lfloor s_4\rfloor_i$, $\langle\lfloor s\rfloor_i, \lfloor M\rfloor_i\rangle \longmapsto^* \langle\lfloor s_4\rfloor_i, \lfloor M\rfloor_i\rangle$. Therefore, the lemma holds for this case.
  - s_3 is $(s_5 \mid \texttt{skip})$ or $(\texttt{skip} \mid s_5)$ where s_5 is not $\texttt{skip}$. Without loss of generality, suppose s_3 is $(s_5 \mid \texttt{skip})$. Then s_1 is $s_5; \lfloor s_4\rfloor_1$, and s_2 is $\texttt{skip}; \lfloor s_4\rfloor_1$. Since $\langle s_5; \lfloor s_4\rfloor_1, \lfloor M\rfloor_1\rangle \longmapsto \langle s'_1, M'_1\rangle$, we have $\langle s_5, \lfloor M\rfloor_1\rangle \longmapsto \langle s'_5, M'_1\rangle$ and s'_1 is $s'_5; \lfloor s_4\rfloor_1$. By (S11) and Lemma A.2, $\langle s, M\rangle \longmapsto \langle(s'_5 \mid \texttt{skip}); s_4, M'\rangle$, and $\lfloor M'\rfloor_1 = M'_1$, and $\lfloor M'\rfloor_2 = \lfloor M\rfloor_2 = M_2$. It is clear that statement (ii) holds.
  - For $i \in \{1,2\}$, $\lfloor s_3\rfloor_i$ is not $\texttt{skip}$. For $i \in \{1,2\}$, because $\langle s_i, M_i\rangle \longmapsto \langle s'_i, M'_i\rangle$ and $s_i = \lfloor s_3\rfloor_i; \lfloor s_4\rfloor_i$, we have $\langle\lfloor s_3\rfloor_i, M_i\rangle \longmapsto \langle s_{3i}, M'_i\rangle$. By induction, $\langle s_3, M\rangle \longmapsto^T \langle s'_3, M'\rangle$, and statement (i) or (ii) holds for T and s'_3. Suppose statement (i) holds for T and s_3. Then for $i \in \{1,2\}$, $\lfloor T\rfloor_i \approx [M_i, M'_i]$ and $\lfloor s'_3\rfloor_i = s_{3i}$. By evaluation rule (S2), $\langle s, M\rangle \longmapsto^T \langle s'_3; s_4, M'\rangle$. Moreover, we have $\lfloor s'_3; s_4\rfloor_i = s_{3i}; \lfloor s_4\rfloor_i = s'_i$ for $i \in \{1,2\}$. Therefore, the lemma holds. For the case that statement (ii) holds for T and s_3, the same argument applies.

- s is $(s_3 \mid s_4)$. In this case, $s_1 = s_3$ and $s_2 = s_4$. Since $\langle s_i, M\rangle \longmapsto \langle s'_i, M'\rangle$ for $i \in \{1,2\}$, we have $\langle s_3, M\rangle_1 \longmapsto \langle s'_1, M''\rangle_1$ and $\langle s_4, M''\rangle_2 \longmapsto \langle s'_2, M'\rangle_2$. Therefore, $\langle s, M\rangle \longmapsto^T \langle(s'_1 \mid s'_2), M'\rangle$ where $T = [M, M'', M']$. By Lemma A.2, $\lfloor T\rfloor_i \approx [M_i, M'_i]$ for $i \in \{1,2\}$. Thus, the lemma holds for this case.

□

Lemma A.4 (Adequacy). Suppose $\langle s_i, M_i\rangle \longmapsto^{T_i} \langle s'_i, M'_i\rangle$ for $i \in \{1,2\}$ are two evaluations in Aimp. Then for an AimpX configuration $\langle s, M\rangle$ such that $\lfloor s\rfloor_i = s_i$ and $\lfloor M\rfloor_i = M_i$ for $i \in \{1,2\}$, we have $\langle s, M\rangle \longmapsto^T \langle s', M'\rangle$ such that $\lfloor T\rfloor_j \approx T_j$ and $\lfloor T\rfloor_k \approx T'_k$, where T'_k is a prefix of T_k and $\{k,j\} = \{1,2\}$.

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

- $|T_1| + |T_2| \le 3$. Without loss of generality, suppose $|T_1| = 1$. Then $T_1 = [M_1]$. Let $T = [M]$. We have $\langle s, M\rangle \longmapsto^T \langle s, M\rangle$. It is clear that $\lfloor T\rfloor_1 = T_1$, and $\lfloor T\rfloor_2 = [M_2]$ is a prefix of T_2.
- $|T_1| + |T_2| > 3$. If $|T_1| = 1$ or $|T_2| = 1$, then the same argument in the above case applies. Otherwise, we have $\langle s_i, M_i\rangle \longmapsto \langle s''_i, M''_i\rangle \longmapsto^{T'_i} \langle s'_i, M'_i\rangle$ and $T_i = [M_i] \oplus T'_i$ for $i \in \{1,2\}$. By Lemma A.3, $\langle s, M\rangle \longmapsto^{T'} \langle s'', M''\rangle$ such that
  1. For $i \in \{1,2\}$, $\lfloor T'\rfloor_i \approx [M_i, M''_i]$ and $\lfloor s''\rfloor_i = s''_i$. Since $|T'_1| + |T'_2| < |T_1| + |T_2|$, by induction we have $\langle s'', M''\rangle \longmapsto^{T''} \langle s', M'\rangle$ such that for $\{k,j\} = \{1,2\}$, $\lfloor T''\rfloor_j \approx T'_j$ and $\lfloor T''\rfloor_k \approx T''_k$, and T''_k is a prefix of T'_k. Let $T = T' \oplus T''$. Then $\langle s, M\rangle \longmapsto^T \langle s', M'\rangle$, and $\lfloor T\rfloor_j \approx T_j$, and $\lfloor T\rfloor_k \approx T'_k$ where $T'_k = [M_k, M''_k] \oplus T''_k$ is a prefix of T_k.
  2. For $\{j,k\} = \{1,2\}$, $\lfloor T'\rfloor_j \approx [M_j]$ and $\lfloor s\rfloor_j = s_j$, and $\lfloor T'\rfloor_k \approx [M_k, M''_k]$ and $\lfloor s\rfloor_k = s''_k$. Without loss of generality, suppose $j = 1$ and $k = 2$. Since $\langle s_1, M_1\rangle \longmapsto^{T_1} \langle s'_1, M'_1\rangle$ and $\langle s''_2, M''\rangle \longmapsto^{T'_2} \langle s'_2, M'_2\rangle$, and $\lfloor s'\rfloor_1 = s_1$ and $\lfloor s'\rfloor_2 = s''_2$, and $|T'_2| < |T_2|$, we can apply the induction hypothesis to $\langle s'', M''\rangle$. By the similar argument in the above case, this lemma holds for this case.

□

Lemma A.5 (Soundness). Suppose $\langle s, M\rangle \longmapsto \langle s', M'\rangle$. Then $\langle\lfloor s\rfloor_i, \lfloor M\rfloor_i\rangle \longmapsto^* \langle\lfloor s'\rfloor_i, \lfloor M'\rfloor_i\rangle$.

Proof. By induction on the derivation of $\langle s, M\rangle \longmapsto \langle s', M'\rangle$. □

A.2. Typing rules

The type system of AimpX includes all the typing rules in Figure 5 and has two additional rules for typing bracket constructs. In general, both confidentiality and integrity noninterference properties are instantiations of an abstract noninterference property: inputs with security labels that sat-

isfy a condition V cannot affect outputs with security labels that do not satisfy V. Two Aimp configurations are called V-equivalent if they differ only at terms and memory locations with security labels that satisfy V. The abstract noninterference property means that the V-equivalence relationship between two configurations is preserved during evaluation.

The bracket constructs captures the differences between two Aimp configurations. As a result, any effect and result of a bracket construct should have a security label that satisfies V. Let $V(\ell)$ and $V(\mathtt{int}_\ell)$ denote that ℓ satisfies V. If v_1 and v_2 are not `none`, rule (V-PAIR) ensures that the value $(v_1 \mid v_2)$ has a label that satisfies V; otherwise, there is no constraint on the label of $(v_1 \mid v_2)$, because `none` is indistinguishable from other values. In rule (S-PAIR), the premise $V(pc')$ ensures that the statement $(s_1 \mid s_2)$ may have only effects with security labels that satisfy V.

$$[\textit{V-PAIR}] \quad \frac{\Gamma \vdash v_1 : \tau \qquad \Gamma \vdash v_2 : \tau \qquad V(\tau) \text{ or } v_1 = \mathtt{none} \text{ or } v_2 = \mathtt{none}}{\Gamma \vdash (v_1 \mid v_2) : \tau}$$

$$[\textit{S-PAIR}] \quad \frac{\Gamma ; \lfloor \mathcal{R} \rfloor_1 ; pc' \vdash s_1 : \tau \qquad \Gamma ; \lfloor \mathcal{R} \rfloor_2 ; pc' \vdash s_2 : \tau \qquad V(pc')}{\Gamma ; \mathcal{R} ; pc \vdash (s_1 \mid s_2) : \tau}$$

Intuitively, noninterference between the inputs with labels satisfying V and the outputs with labels that does not satisfying V is achieved as long as all the bracket constructs are well-typed.

An important constraint that condition V needs to satisfy is that $V(\ell)$ implies $V(\ell \sqcup \ell')$ for any ℓ'. In AimpX, if expression e is evaluated to a bracket value $(n_1|n_2)$, statement `if` e `then` s_1 `else` s_2 would be reduced to a bracket statement $(s'_1 \mid s'_2)$ where s'_i is either s_1 or s_2. To show $(s'_1 \mid s'_2)$ is well-typed, we need to show that s_1 and s_2 are well-typed under a program-counter label that satisfying V, and we can show it by using the constraint on V. Suppose e has type $\mathtt{int}_\ell$, then we know that s_1 and s_2 are well-typed under the program counter label $pc \sqcup \ell$. Furthermore, ℓ satisfies V because the result of e is a bracket value. Thus, by the constraint that $V(\ell)$ implies $V(\ell \sqcup \ell')$, we have $V(pc \sqcup \ell)$.

Suppose $\Gamma ; \mathcal{R} ; pc \vdash (s_1 \mid s_2) : \tau$, and $m \in \mathcal{R}$. By the evaluation rule (S11), it is possible that $\langle (s_1 \mid s_2),\, M \rangle \longmapsto^* \langle (s'_1 \mid s_2),\, M' \rangle$ and $M'(m) = (n \mid \mathtt{none})$, which means that m still needs to be assigned in s_2, but not in s'_1. Assume there exists $\mathcal{R}'$ such that $\Gamma ; \mathcal{R}' ; pc \vdash (s'_1 \mid s_2) : \tau$. Then by rule (S-PAIR), we have $\Gamma ; \lfloor \mathcal{R}' \rfloor_1 ; pc \vdash s'_1 : \tau$ and $\Gamma ; \lfloor \mathcal{R}' \rfloor_2 ; pc \vdash s_2 : \tau$. Intuitively, we want to have $m \notin \lfloor \mathcal{R}' \rfloor_1$ and $m \in \lfloor \mathcal{R}' \rfloor_2$, which are consistent with M'. To indicate such a situation, a reference m in $\mathcal{R}$ may have an index: m^1 or m^2 means that m needs to be assigned only in the first or second component of a bracket statement, and $m^\bullet$ is the same as m. The projection of $\mathcal{R}$ is computed in the following way:

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

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

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

A.3. Subjection reduction

Lemma A.6 (Update). If $\Gamma ; \mathcal{R} \vdash v : \tau$ and $\Gamma ; \mathcal{R} \vdash v' : \tau$, then $\Gamma ; \mathcal{R} \vdash v[v'/\pi_i] : \tau$.

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

Lemma A.7 (Relax). If $\Gamma ; \mathcal{R} ; pc \sqcup \ell \vdash s : \tau$, then $\Gamma ; \mathcal{R} ; pc \vdash s : \tau$.

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

Lemma A.8. Suppose $\Gamma ; \mathcal{R} \vdash e : \tau$, and $\Gamma \vdash M$, and $\langle e,\, M \rangle \Downarrow v$. Then $\Gamma ; \mathcal{R} \vdash v : \tau$.

Proof. By induction on the structure of e. □

Lemma A.9. Suppose $\Gamma ; \mathcal{R} ; pc \vdash s : \mathtt{stmt}_{\mathcal{R}'}$. If $m^i \in \mathcal{R}$ where $i \in \{1, 2\}$, then $m \notin \mathcal{R}'$.

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

Definition A.1 ($\Gamma \vdash M$). $\Gamma \vdash M$ if $dom(\Gamma) = dom(M)$, and for any $m \in dom(\Gamma)$, $\Gamma ; \mathcal{R} \vdash M(m) : \Gamma(m)$.

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

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

i. $\Gamma ; \mathcal{R}' ; pc \vdash s' : \tau$, and $\mathcal{R}' \le \mathcal{R}$, and $\Gamma \vdash M'$.

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

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

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

Proof. By induction on the evaluation step $\langle s,\, M\rangle_i \longmapsto \langle s',\, M'\rangle_i$. Without loss of generality, we assume that the derivation of $\Gamma\,;\mathcal{R}\,;pc \vdash s : \tau$ does not end with using the (SUB) rule. Indeed, if $\Gamma\,;\mathcal{R}\,;pc \vdash s : \texttt{stmt}_{\mathcal{R}_2}$ is derived by $\Gamma\,;\mathcal{R}\,;pc \vdash s : \texttt{stmt}_{\mathcal{R}_1}$ and $\Gamma\,;\mathcal{R}\,;pc \vdash \texttt{stmt}_{\mathcal{R}_1} \leq \texttt{stmt}_{\mathcal{R}_2}$, and there exists $\mathcal{R}''$ such that statements (i)–(iv) hold for $\Gamma\,;\mathcal{R}\,;pc \vdash s : \texttt{stmt}_{\mathcal{R}_1}$, then by Lemma A.9, we can show that $\mathcal{R}' = \mathcal{R}'' \cup (\mathcal{R}_2 - \mathcal{R}_1)$ satisfies statements (i)–(iv) for $\Gamma\,;\mathcal{R}\,;pc \vdash s : \texttt{stmt}_{\mathcal{R}_2}$.

- **Case (S1)**. In this case, s is $m := e$, s' is $\texttt{skip}$, and τ is $\texttt{stmt}_{\mathcal{R}-\{m\}}$. By (S1), M' is $M[m \mapsto M(m)[v/\pi_i]]$. By Lemma A.8, we have $\Gamma \vdash v : \Gamma(m)$, which implies that $M(m)[v/\pi_i]$ has type $\Gamma(m)$. Therefore, $\Gamma \vdash M'$. The well-formedness of τ implies that $\mathcal{R}$ does not contain any indexed references. Let $\mathcal{R}'$ be $\mathcal{R} - \{m\}$. It is clear that $\mathcal{R}' \leq \mathcal{R}$. By rule (SKIP), $\Gamma\,;\mathcal{R}'\,;pc \vdash \texttt{skip} : \texttt{stmt}_{\mathcal{R}'}$. Because $\lfloor M' \rfloor_i(m) = v \neq \texttt{none}$, and $\mathcal{R} - \mathcal{R}' = \{m\}$, statement (ii) holds. Since $\lfloor M' \rfloor_i(m) = n$ and $\mathcal{R} - \mathcal{R}' = \{m\}$, we have that $\Gamma\,;\mathcal{R} \vdash \lfloor M \rfloor_i$ implies $\Gamma\,;\mathcal{R}'\,;L \vdash \lfloor M' \rfloor_i$.

- **Case (S2)**. Obvious by induction.

- **Case (S3)**. Trivial.

- **Case (S4)**. In this case, s is $\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2$. By the typing rule (IF), we have $\Gamma\,;\mathcal{R}\,;pc \sqcup \ell_e \vdash s_1 : \tau$. By Lemma A.7, $\Gamma\,;\mathcal{R}\,;pc \vdash s_1 : \tau$. In this case, $M' = M$ and $\mathcal{R}' = \mathcal{R}$, so statements (ii) and (iii) immediately hold.

- **Case (S5)**. By the similar argument of case (S4).

- **Case (S6)**. In this case, s is $\texttt{while}\ e\ \texttt{do}\ s_1$, and τ is $\texttt{stmt}_{\mathcal{R}}$. By rule (WHILE), $\Gamma\,;\mathcal{R}\,;pc \sqcup \ell \vdash s_1 : \tau$, where ℓ is the label of e. Then we have $\Gamma\,;\mathcal{R}\,;pc \sqcup \ell \vdash s_1; \texttt{while}\ e\ \texttt{do}\ s_1 : \tau$. Furthermore, $\Gamma\,;\mathcal{R}\,;pc \sqcup \ell \vdash \texttt{skip} : \texttt{stmt}_{\mathcal{R}}$. By rule (IF), $\Gamma\,;\mathcal{R}\,;pc \vdash \texttt{if}\ e\ \texttt{then}\ s; \texttt{while}\ e\ \texttt{do}\ s\ \texttt{else}\ \texttt{skip} : \tau$. Since $M' = M$ and $\mathcal{R}' = \mathcal{R}$, statements (ii) and (iii) hold.

- **Case (S10)**. In this case, s is $\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2$, and i must be $\bullet$. Suppose $\Gamma \vdash e : \texttt{int}_\ell$. By Lemma A.8, $\Gamma \vdash (n_1 \mid n_2) : \texttt{int}_\ell$. By rule (V-PAIR), $V(\ell)$ holds, which implies $V(pc \sqcup \ell)$. By rule (IF), $\Gamma\,;\mathcal{R}\,;pc \sqcup \ell \vdash s_i : \tau$, which implies $\Gamma\,;\mathcal{R}\,;pc \sqcup \ell \vdash \texttt{if}\ n_i\ \texttt{then}\ \lfloor s_1 \rfloor_i\ \texttt{else}\ \lfloor s_2 \rfloor_i : \tau$. By rule (S-PAIR), $\Gamma\,;\mathcal{R}\,;pc \vdash s' : \tau$. Again, since $M' = M$ and $\mathcal{R}' = \mathcal{R}$, statements (ii) and (iii) hold.

- **Case (S11)**. In this case, s is $(s_1 \mid s_2)$. Without loss of generality, suppose $\langle s_1,\, M\rangle_1 \longmapsto \langle s_1',\, M'\rangle_1$, and $\langle s,\, M\rangle \longmapsto \langle (s_1' \mid s_2),\, M'\rangle$. By rule (S-PAIR), $\Gamma\,;\lfloor \mathcal{R} \rfloor_1\,;pc \vdash s_1 : \tau$. By induction, there exists $\mathcal{R}_1'$ such that $\Gamma\,;\mathcal{R}_1'\,;pc \vdash s_1' : \tau$, and $\mathcal{R}_1' \subseteq \lfloor \mathcal{R} \rfloor_1$, and $\Gamma \vdash M'$. Let $\mathcal{R}'$ be $\mathcal{R}_1' \bullet \lfloor \mathcal{R} \rfloor_2$, which is computed by the formula:

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

  Since $\lfloor \mathcal{R}' \rfloor_1 = \mathcal{R}_1'$ and $\lfloor \mathcal{R}' \rfloor_2 = \lfloor \mathcal{R} \rfloor_2$, we have $\Gamma\,;\lfloor \mathcal{R}' \rfloor_1\,;pc \vdash s_1' : \tau$. By rule (S-PAIR), $\Gamma\,;\mathcal{R}'\,;pc \vdash s' : \tau$ holds. Since $\lfloor \mathcal{R}' \rfloor_2 = \lfloor \mathcal{R} \rfloor_2$, for any $m^j \in \mathcal{R} - \mathcal{R}'$, it must be the case that $j = 1$, and $m \in \lfloor \mathcal{R} \rfloor_1 - \mathcal{R}_1'$. By induction, $\lfloor M' \rfloor_1(m) \neq \texttt{none}$. Therefore, statements (ii) holds.

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

- **Case (S12)**. In this case, s is $(\texttt{skip} \mid \texttt{skip})$. We have $\Gamma\,;\lfloor \mathcal{R} \rfloor_i\,;pc \vdash \texttt{skip} : \texttt{stmt}_{\lfloor \mathcal{R} \rfloor_i}$ for $i \in \{1, 2\}$. By rule (S-PAIR), $\Gamma\,;\lfloor \mathcal{R} \rfloor_i\,;pc' \vdash \texttt{skip} : \tau$. Therefore, $\Gamma\,;\lfloor \mathcal{R} \rfloor_i\,;pc' \vdash \texttt{stmt}_{\lfloor \mathcal{R} \rfloor_i} \leq \tau$. By the subtyping rule, $\tau = \texttt{stmt}_{\lfloor \mathcal{R} \rfloor_i}$. So $\lfloor \mathcal{R} \rfloor_1 = \lfloor \mathcal{R} \rfloor_2 = \mathcal{R}$ and $\tau = \texttt{stmt}_{\mathcal{R}}$. By rule (SKIP), $\Gamma\,;\mathcal{R}\,;pc \vdash \texttt{skip} : \tau$.

□

A.4. Progress

Theorem A.2 (Progress). Let $V(\ell)$ be $I(\ell) \leq L$, and let $|s|$ represent the size of the statement s, i.e. the number of syntactical tokens in s. Suppose $\Gamma\,;\mathcal{R}\,;pc \vdash s : \texttt{stmt}_{\mathcal{R}'}$ and $\Gamma\,;\mathcal{R} \vdash M$. If $A_\Gamma(\mathcal{R}) \not\leq L$ then there exists $\langle s',\, M'\rangle$ such that $\langle s,\, M\rangle \longmapsto \langle s',\, M'\rangle$. Furthermore, if $\lfloor \mathcal{R} \rfloor_1 \neq \lfloor \mathcal{R} \rfloor_2$, then $|s'| < |s|$.

Proof. By induction on the structure of s. □

A.5. Noninterference

Theorem A.3 (Confidentiality noninterference). If $\Gamma\,;\mathcal{R}\,;pc \vdash s : \tau$, then $\Gamma \vdash \texttt{NI}_C(s)$.

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

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

Let $V(\ell)$ be $C(\ell) \not\leq L$. Then $\Gamma \vdash M_1 \approx_{C \leq L} M_2$ implies that $\Gamma \vdash M$. Suppose $\langle s_i,\, M_i\rangle \longmapsto^{T_i} \langle s_i',\, M'\rangle$ for $i \in \{1, 2\}$. Then by Lemma A.4, there exists $\langle s',\, M'\rangle$ such that $\langle s,\, M\rangle \longmapsto^{T} \langle s',\, M'\rangle$, and $\lfloor T \rfloor_j \approx T_j$ and $\lfloor T \rfloor_k \approx T_k'$ where $\{j, k\} = \{1, 2\}$ and T_k' is a prefix of T_j. By Theorem A.1, for each M' in T, $\Gamma \vdash M'$, which implies that $\lfloor M' \rfloor_1 \approx_{C \leq L} \lfloor M' \rfloor_2$. Therefore, we have $\Gamma \vdash T_j \approx_{C \leq L} T_k'$. Thus, $\Gamma \vdash \texttt{NI}_C(s)$. □

Theorem A.4 (Integrity noninterference). If $\Gamma\,;\mathcal{R}\,;pc \vdash s : \tau$, then $\Gamma \vdash \mathtt{NI}_I(s)$.

Proof. Let $V(\ell)$ be $I(\ell) \leq I$. By the same argument as in the proof of the confidentiality noninterference theorem. □

Lemma A.10 (Balance). Let $V(\ell)$ be $I(\ell) \leq L$. Suppose $\Gamma\,;\mathcal{R}\,;pc \vdash s : \tau$, and $\Gamma\,;\mathcal{R} \vdash M$. There exists $\langle s',\, M'\rangle$ such that $\langle s,\, M\rangle \longmapsto^* \langle s',\, M'\rangle$, and $\Gamma \vdash \lfloor M'\rfloor_1 \approx_{A\not\leq L} \lfloor M'\rfloor_2$.

Proof. By induction of on the size of s.

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

□

Theorem A.5 (Availability noninterference). If $\Gamma\,;\mathcal{R}\,;pc \vdash s : \tau$, then $\Gamma\,;\mathcal{R} \vdash \mathtt{NI}_A(s)$.

Proof. Let $V(\ell)$ be $I(\ell) \leq L$. Given two memories M_1 and M_2 in Aimp such that $\Gamma \vdash M_1 \approx_{I\not\leq L} M_2$ and for any m in $dom(\Gamma)$, $m \notin \mathcal{R}$ and $A(\Gamma(m)) \not\leq L$ imply $M_i(m) \neq \mathtt{none}$. To prove $\Gamma \vdash \mathtt{NI}_A(s)$, we only need to show that there exists $\langle s'_i,\, M'_i\rangle$ such that $\langle s,\, M_i\rangle \longmapsto^* \langle s'_i,\, M'_i\rangle$, and for any $\langle s''_i,\, M''_i\rangle$ such that $\langle s'_i,\, M'\rangle \longmapsto^* \langle s''_i,\, M''\rangle$, $\Gamma \vdash M''_1 \approx_{A\not\leq L} M''_2$ holds.

Let $M = M_1 \uplus M_2$. Intuitively, by Lemma A.10, evaluating $\langle s,\, M\rangle$ will eventually result in a memory M' such that $\Gamma \vdash \lfloor M'\rfloor_1 \approx_{A\not\leq L} \lfloor M'\rfloor_2$, and if any high-availability reference m is unavailable in M', m will remain unavailable. This conclusion can be projected to $\langle s,\, M_i\rangle$ for $i \in \{1,2\}$ by Lemma A.5.

Suppose there exists $\langle s',\, M'\rangle$ such that $\langle s,\, M\rangle \longmapsto^* \langle s',\, M'\rangle$, and for any m with $A_\Gamma(m) \not\leq L$, $\lfloor M'\rfloor_i \neq \mathtt{none}$ for $i \in \{1,2\}$. By Lemma A.5, $\langle s,\, M_i\rangle \longmapsto^* \langle\lfloor s'\rfloor_i,\, \lfloor M'\rfloor_i\rangle$. Moreover, for any $\langle s'_i,\, M'_i\rangle$ such that $\langle\lfloor s'\rfloor_i,\, \lfloor M'\rfloor_i\rangle \longmapsto^* \langle s'_i,\, M'_i\rangle$, and any m with $A_\Gamma(m) \not\leq L$, it must be the case that $M'_i(m) \neq \mathtt{none}$. Therefore, $\Gamma \vdash M'_1 \approx_{A\not\leq L} M'_2$.

Otherwise, there exists $\langle s',\, M'\rangle$ such that $\langle s,\, M\rangle \longmapsto^* \langle s',\, M'\rangle$, there exists m such that $A(\Gamma(m)) \not\leq L$ and $\lfloor M'\rfloor_i(m) \approx_{\mathtt{none}}$ for some $i \in \{1,2\}$, and for any $\langle s'',\, M''\rangle$ such that $\langle s',\, M'\rangle \longmapsto^* \langle s'',\, M''\rangle$, $\Gamma \vdash \lfloor M'\rfloor_i \approx_{A\not\leq L} \lfloor M''\rfloor_i$. By Lemma A.10, $\Gamma \vdash \lfloor M'\rfloor_1 \approx_{A\not\leq L} \lfloor M'\rfloor_2$ must hold. Otherwise, assume $\Gamma \vdash \lfloor M'\rfloor_1 \approx_{A\not\leq L} \lfloor M'\rfloor_2$ does not hold. Then there exists $\langle s'',\, M''\rangle$ such that $\langle s',\, M'\rangle \longmapsto^* \langle s'',\, M''\rangle$ and $\Gamma \vdash \lfloor M''\rfloor_1 \approx_{A\not\leq L} \lfloor M''\rfloor_2$. Because for $i \in \{1,2\}$, $\Gamma \vdash \lfloor M'\rfloor_i \approx_{A\not\leq L} \lfloor M''\rfloor_i$, we have $\Gamma \vdash \lfloor M'\rfloor_1 \approx_{A\not\leq L} \lfloor M'\rfloor_2$, which contradicts the original assumption. In addition, we can show that $\langle s',\, M'\rangle$ would diverge and generate a trace of infinite size. Indeed, by Theorem A.1, there exists $\mathcal{R}'$ such that $\Gamma\,;\mathcal{R}'\,;pc \vdash s' : \tau$, and $\Gamma\,;\mathcal{R}'\,;L \vdash M'$. Then $A(\mathcal{R}') \not\leq L$, because there exists m such that $A(\Gamma(m)) \not\leq L$ and $\lfloor M'\rfloor_i(m) \approx_{\mathtt{none}}$ for some $i \in \{1,2\}$. By Theorem A.2, there exists $\langle s'',\, M''\rangle$ such that $\langle s',\, M'\rangle \longmapsto \langle s'',\, M''\rangle$. Since $\Gamma \vdash \lfloor M'\rfloor_i \approx_{A\not\leq L} \lfloor M''\rfloor_i$ for $i \in \{1,2\}$, $\langle s'',\, M''\rangle$ can make progress by the same argument. Therefore, evaluating $\langle s',\, M'\rangle$ will generate a trace of infinite size. For $i \in \{1,2\}$, suppose there exists $\langle s'_i,\, M'_i\rangle$ such that $\langle\lfloor s'\rfloor_i,\, \lfloor M'\rfloor_i\rangle \longmapsto^* \langle s'_i,\, M'_i\rangle$. Since the trace from evaluating $\langle s',\, M'\rangle$ is infinitely long, for $i \in \{1,2\}$, there exists $\langle s''_i,\, M''_i\rangle$ such that $\langle s',\, M'\rangle \longmapsto^* \langle s''_i,\, M''\rangle$ and $\lfloor M''_i\rfloor_i = M'_i$. Therefore, $\Gamma \vdash M'_i \approx_{A\not\leq L} \lfloor M'\rfloor_i$ for $i \in \{1,2\}$. Thus, $\Gamma \vdash M'_1 \approx_{A\not\leq L} M'_2$.

□

B. Typing the `new` statement

The type system of Aimp needs to be extended to manipulate reference variables and check the `new` statement. First, variable x represents a reference that can be used in the typing environment: the typing assignment Γ may map x to a type, and the reference set $\mathcal{R}$ may contain x. For example, consider the statement $\mathtt{new}\ x : \ell_x = \mathtt{ref}(\ell)\ \mathtt{in}\ s$. Suppose the typing environment for the `new` statement is "$\Gamma\,;\mathcal{R}\,;pc$". Then the typing environment for s should be "$\Gamma, x : \mathtt{int}_\ell\,;\mathcal{R}\cup\{x\}\,;pc$". Second, to control the implicit information flow arising from the creation of a new reference, the typing rule for checking the statement $\mathtt{new}\ x : \ell_x = \mathtt{ref}(\ell)\ \mathtt{in}\ s$ needs to ensure that the confidentiality and integrity components of ℓ_x are bounded by the current program counter label pc. Formally, the corresponding constraints are $C(pc) \leq C(\ell_x)$ and $I(\ell_x) \leq I(pc)$.

Intuitively, the value or availability of a reference created at a program point is not affected by whether control reaches this point, because the reference itself does not exist if control does not reach the point. As a result, the typing rules in Figure 5 may be over-restrictive for reasoning about the security policies of a reference created at run time. For example, consider the following code:

```
if (!m) then
  new x:ℓx = ref(ℓ) in
    while !m1 do m1 := m1 - 1;
    x := 1
else
  skip
```

Suppose $\Gamma\,;\mathcal{R}\,;pc$ is the typing environment for the `while` statement in the above code. Then $I(pc) \leq I(\Gamma(m))$ holds by the typing rule (IF). Furthermore, we have $x \in \mathcal{R}$, which requires that $A(\ell) \leq I(pc)$ by the typing rule (WHILE). Therefore, $A(\ell) \leq I(\Gamma(m))$ needs to be satisfied for the above code to be well-typed, which contradicts the intuition that the availability of x is not affected by whether control reaches the `new` statement. To increase the precision of the static security analysis, we extend the type system to keep track of the program counter label for each reference variable x from the program point where x is created. As a result, the typing environment is extended with a new component Δ that maps references to program count labels.

The typing rule (NEW) is used to check the `new` statement $\texttt{new}\ x : \ell_x = \texttt{ref}(\ell)\ \texttt{in}\ s$. In this rule, statement s is checked with variable x in scope. In the typing environment of s, the program counter label mapped to x is $\bot_{pc}$, which is $\{\bot_C, \top_I, \top_A\}$.

$$[\textit{NEW}]\quad \frac{\begin{array}{c}\Gamma, x{:}\texttt{int}_\ell\,;\mathcal{R}\cup\{x\}\,;\Delta, x{:}\bot_{pc}\,;pc \vdash s : \tau \\ C(pc) \leq C(\ell_x) \qquad I(\ell_x) \leq I(pc)\end{array}}{\Gamma\,;\mathcal{R}\,;\Delta\,;pc \vdash \texttt{new}\ x{:}\ell_x = \texttt{ref}(\ell)\ \texttt{in}\ s : \tau}$$

In addition, typing rules (ASSIGN), (IF) and (WHILE) need to take into account the Δ component in the typing environment. To abuse the notation a little bit, we use $\Delta \sqcup \ell$ to denote the program counter map Δ' that satisfies $dom(\Delta) = dom(\Delta')$ and $\Delta'(r) = \Delta(r) \sqcup \ell$ for any $r \in dom(\Delta)$. In addition, let $\Delta(r, pc)$ denote $\Delta(r)$ if $r \in dom(\Delta)$, and pc if otherwise. The adjusted typing rules are shown as follows:

$$[\textit{ASSIGN}]\quad \frac{\begin{array}{c}\Gamma\,;\mathcal{R} \vdash r : \texttt{int}_\ell\ \texttt{ref} \qquad \Gamma\,;\mathcal{R} \vdash e : \texttt{int}_{\ell'} \\ C(\Delta(r,pc)) \sqcup C(\ell') \leq C(\ell) \\ I(\ell) \leq I(\Delta(r,pc)) \sqcap I(\ell')\end{array}}{\Gamma\,;\mathcal{R}\,;\Delta\,;pc \vdash r := e : \texttt{stmt}_{\mathcal{R}-\{r\}}}$$

$$[\textit{IF}]\quad \frac{\begin{array}{c}\Gamma\,;\mathcal{R} \vdash e : \texttt{int}_\ell \\ \Gamma\,;\mathcal{R}\,;\Delta \sqcup \ell\,;pc \sqcup \ell \vdash s_i : \tau \quad i \in \{1,2\}\end{array}}{\Gamma\,;\mathcal{R}\,;\Delta\,;pc \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \tau}$$

$$[\textit{WHILE}]\quad \frac{\begin{array}{c}\Gamma \vdash e : \texttt{int}_\ell \quad \Gamma\,;\mathcal{R}\,;\Delta \sqcup \ell\,;pc \sqcup \ell \vdash s : \texttt{stmt}_{\mathcal{R}} \\ A_\Gamma(\mathcal{R}) \leq I(\ell) \qquad \forall r \in \mathcal{R},\ A(r) \leq I(\Delta(r,pc))\end{array}}{\Gamma\,;\mathcal{R}\,;\Delta\,;pc \vdash \texttt{while}\ e\ \texttt{do}\ s : \texttt{stmt}_{\mathcal{R}}}$$