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

Although availability is often considered one of the three key aspects of information security (along with confidentiality and integrity), availability assurance has been largely divorced from other security concerns. This paper starts to bridge the gap by giving a single, common framework for reasoning about confidentiality, integrity, and availability.

The first part of this framework is a language for specifying confidentiality, integrity, and availability policies. This policy language extends the decentralized label model [21], and thus is able to describe security policies involving mutually distrusting principals.

The second part of the framework is a semantics for the policy language, which characterizes precisely what it means for a system to enforce a policy. 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 for enforcing confidentiality, integrity, and availability policies. 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 [24]). 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 precisely defining what is meant by "availability"; then we define an expressive policy language for availability, and we 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. The output does not have to be

correct—that is the province of integrity.

There are two common ways to specify 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 [27]. The second approach is to specify *failure factors* (factors that could cause the system to fail), for example, the minimum number of host failures needed to bring down the system [25]. We adopt this second approach here.

The above description of availability glosses over another aspect of availability: timeliness. How soon does an output have to occur in order to be considered to be available? For real-time services, there may be hard time bounds beyond which a late output is useless. Reasoning about how long it takes to generate an output adds considerable complexity, 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 and principals

We assume that the unavailability of a system output is 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 policy language that can describe all these kinds of failures and how the availability of the system should be affected by them.

We consider a failure to be the malfunction of a *principal*, an entity that may affect the behavior of a system. Therefore, a 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`: the "superuser", which has the ability to control (or shut down) a system, and to act on behalf of users.
- $\texttt{DDoS}_{1000}$: a distributed denial of service attack launched from 1000 machines. This principal can be used to specify the availability of a system that tolerates DDoS attacks launched from fewer than 1000 machines.
- `puzzle`: the puzzle generated by a puzzle-based defense mechanism [13] for DoS attacks. This principal fails if attackers can feasibly solve the puzzle and launch DoS attacks successfully.

More complex failure scenarios are described by using *composite principals* [1]. 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 composite principal given by the conjunction $\texttt{power} \wedge \texttt{ups}$.

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

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

The notation a is an abstract name representing a principal. The composite principal $p_1 \wedge p_2$ 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): principal $p_1 \vee p_2$ represents a failure that happens if either p_1 or p_2 fails. For example, the principal $\texttt{root} \vee \texttt{power}$ can make a system fail if the superuser 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 [17]. 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, 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—that is, p_1 has all the powers of p_2 and is at least as trustworthy [21]. Interpreting principals as failure factors, this means the failure of p_1 is worse than the failure of p_2 (or the same). The acts-for relation is useful for analyzing availability, because $p_1 \succeq p_2$ means that the availability represented by p_1 is at least as high as the availability represented by p_2. For example, if hosts h_1 and h_2 are two principals, then $h_1 \wedge h_2 \succeq h_1$ holds because h_1 fails if both h_1 and h_2 fail. And information with the availability $h_1 \wedge h_2$ also achieves the availability h_1, because if h_1 does not fail, $h_1 \wedge h_2$ does not fail.

The acts-for relation between principals creates a *principal hierarchy* $\mathcal{H}$, an ordering (actually, a pre-order) on the set of principals. By the definition of acts-for, a principal hierarchy must satisfy the following 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 policies

An *end-to-end availability policy* specifies the availability that a user requires of a system input or output. In this work, availability is specified as a principal representing a failure factor. Accordingly, an availability policy has the form $u : p$, where principal u is the *policy owner* (the user who specifies the policy), and principal p represents the required availability. For example, if Alice specifies the availability policy $\texttt{Alice} : h_1 \wedge h_2$ on one of her files, it means that Alice requires the file to be available if hosts h_1 and h_2 do not both fail.

In general, security (including availability) rests on assumptions. In particular, the enforcement of a policy owned by user u is contingent on the assumptions made by u. For example, system security commonly depends on a trusted computing base (TCB). If the assumption that the TCB is trustworthy is false, security may not be enforced. In a system with mutual distrust, such as a distributed system crossing administrative domains, different users might assume different components of the system trustworthy. Thus it is important to specify policy owners explicitly to indicate whose assumptions are relevant to policy enforcement.

We build on the decentralized label model (DLM) [21], which applies the notion of policy ownership to confidentiality and integrity. In the DLM, a confidentiality or integrity policy has the form $u : p_1, \ldots, p_n$, meaning that u allows only principals $p_1, \ldots, p_n$ to read or update the information protected by the policy. Using disjunctive principals, the policy $u : p_1, \ldots, p_n$ can be written in the form $u : p_1 \vee \ldots \vee p_n$, just like an availability policy. Furthermore, for each security property (confidentiality, integrity or availability), a policy $u : p$ can be interpreted as an assumption by u that p does not fail. 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 if p does not fail.

Based on this commonality, we can separate a notion of *owned policies* from the security properties these policies apply to. Let π abstractly represent a security property of the system; it may be a confidentiality, integrity, or availability property. Formally, we treat π as an abstract proposition that is true if the corresponding security property holds, and false otherwise. In general, if the policy $u : p$ is applied to a security property π, it means that u requires π to hold if p does not fail.

Treating owned policies separately from the underlying, abstract security properties is useful for two reasons. First, it enables a uniform semantics for security policies. Second, it may in general be infeasible to formally specify or analyze what it means for a security property to hold, particularly if the security violation might occur outside the computing system; some form of abstraction is needed. This abstraction does not create a problem for security enforcement as long as the dependencies between security properties induced by a computing system can be analyzed precisely.

2.5. Policy semantics

Whether the policy $u : p$ is applied to confidentiality, integrity, or availability properties, it corresponds to two security assumptions: that p does not fail, and the assumptions made by u are true.

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

$$\sigma ::= ok\, p \mid \sigma_1 \wedge \sigma_2 \mid \sigma_1 \vee \sigma_2$$

where $ok\, p$ means that principal p does not fail. The properties of the acts-for relation can be captured formally using these propositions. If p_1 acts for p_2, this means the failure of p_1 implies the failure of p_2:

$$ok\, p_2 \Rightarrow ok\, p_1 \quad \text{iff} \quad p_1 \succeq p_2 \tag{1}$$

Consequently, composite principals satisfy the following conditions:

$$ok\, p_1 \vee ok\, p_2 \quad \text{iff} \quad ok\, (p_1 \wedge p_2)$$
$$ok\, p_1 \wedge ok\, p_2 \quad \text{iff} \quad ok\, (p_1 \vee p_2)$$

In addition, we assume there exists an *assumption configuration* Σ that maps each principal u to its assumptions $\sigma = \Sigma(u)$. In general, if $u_1 \succeq u_2$, then any assumption made by u_1 is considered an assumption made by u_2. Consequently, Σ must satisfy the following condition:

$$u_1 \succeq u_2 \quad \Rightarrow \quad (\Sigma(u_2) \Rightarrow \Sigma(u_1)) \tag{2}$$

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

$$[\![u : p]\!] = \Sigma(u) \wedge ok\, p$$

Suppose a policy P is applied to a security property π. Then the meaning of the policy is a characterization of when the property is guaranteed to hold. To enforce the policy is to guarantee π under the assumption that $[\![P]\!]$ is true, that is, to ensure $[\![P]\!] \Rightarrow \pi$.

Consider the example of enforcing the availability policy $\texttt{Alice} : h_1 \wedge h_2$ on Alice's file. The goal is to ensure that the file is available under the assumption that $\Sigma(\texttt{Alice}) \wedge ok(h_1 \wedge h_2)$ is true. Therefore, one way to enforce the policy is to replicate the file on hosts h_1 and h_2 because $ok\, (h_1 \wedge h_2)$ means that h_1 and h_2 cannot both fail, which ensures that at least one host is available to serve accesses to the file. Moreover, if $\Sigma(\texttt{Alice})$ implies $ok\, h_3$, storing the file on h_3 is another way to enforce the policy.

2.6. Dependency analysis and policy ordering

A system processes inputs and produces outputs, creating dependencies between security properties of those inputs and outputs. Such dependencies capture influences of the system on security and induce constraints on security policies. For example, consider a system running the following pseudo-code:

```
while (i > 0) skip;
send i to o;
```

This program sends the input i to the output o if the value of i is not positive. Otherwise, the program diverges, so the output is unavailable. Thus, the availability of o depends on the integrity of i. For simplicity, suppose there is only one policy applied to these properties, and let A_o represent the availability policy of o, and I_i represent the integrity policy of i. Then $[\![A_o]\!] \Rightarrow [\![I_i]\!]$ must hold in order to enforce A_o. If $[\![A_o]\!] \not\Rightarrow [\![I_i]\!]$, then $[\![A_o]\!]$ and $\neg[\![I_i]\!]$ may both hold. In this case the integrity of i is not guaranteed, and the program may compromise the availability of o. But this violates the availability policy A_o because $[\![A_o]\!]$ holds.

In general, given two security properties π_1 and π_2, and their policies P_1 and P_2, if π_1 depends on π_2, then $[\![P_1]\!] \Rightarrow [\![P_2]\!]$ must hold in order to enforce P_1. The constraint $[\![P_1]\!] \Rightarrow [\![P_2]\!]$ corresponds to a natural ordering on the two policies: P_2 is at least as strong as P_1, written $P_1 \leq P_2$, meaning that for any configuration Σ and any security property π, P_1 is enforced on π if P_2 is enforced on π. It is clear that $P_1 \leq P_2$ is equivalent to $\forall\Sigma.\, [\![P_1]\!] \Rightarrow [\![P_2]\!]$. The quantification over Σ ensures that analyses based on the policy ordering are insensitive to Σ.

From the semantics of policies and formulas (1) and (2) in Section 2.5, the following rule for ordering policies immediately follows:

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

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 $u_i : p_i$ applied to the same security property.

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 $\forall\Sigma.\, [\![\beta_1]\!] \Rightarrow [\![\beta_2]\!]$. 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 [7]. 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 $e_1 + e_2$. 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 of $e_1 + e_2$ 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

This section introduces two security assumptions that enables enforcing security policies in a system by noninterference. One assumption specifies which policies are already enforced, and the other designates the power of attackers.

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 generated 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. Therefore, 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 the system. This leads to our second security assumption:

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

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

Indeed, suppose the output o is made unavailable by attackers. By (SA2), it is because the availability or integrity of some input i is compromised by attackers. Without loss of generality, suppose the availability of i is compromised. Let A_i be the availability policy of i. By (SA1), A_i is enforced, which, plus the unavailability of i, implies that $[\![A_i]\!]$ is false. By the noninterference property, we have $A_o \leq A_i$, which implies $[\![A_o]\!] \Rightarrow [\![A_i]\!]$. Thus, $[\![A_o]\!]$ is false because $[\![A_i]\!]$ is false. Therefore, the unavailability of o implies that $[\![A_o]\!]$ is false. In other words, if $[\![A_o]\!]$ is true, then o must be available, which means that A_o is enforced.

One advantage of enforcing an availability policy by noninterference is to avoid proving that a program will eventually produce an output, which generally amounts to solving the halting problem.

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 [30, 11, 32]. Since availability policies also correspond to a noninterference property in our security model, a static program 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.

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

Figure 1. Syntax of Aimp

The Aimp language is a basic imperative language with assignments, sequential composition, conditionals and loops. What distinguishes Aimp from other security-typed imperative languages [30] is the 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 program code.
- 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 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 1. 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 program text. Statements include the empty statement `skip`, the assignment 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:

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

In Aimp, the only data type is $\texttt{int}_\ell$, an integer type annotated with security label ℓ, which contains three base labels as described in Section 2.8.

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 2. Let M represent a memory that is a finite map from locations to values (including `none`), and let $\langle s,\ M\rangle$ be a machine configuration. Then a small evaluation step is a transition from $\langle s,\ M\rangle$ to another configuration $\langle s',\ M'\rangle$, written $\langle s,\ M\rangle \longmapsto \langle s',\ M'\rangle$.

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.

3.4. Examples

By its simplicity, the Aimp language helps focus on the essentials of an imperative language. Figure 3 shows a few code segments that demonstrate various kind of availability dependencies, some of which are subtle. In all these examples, m_o represents an output, and its initial value is `none`. All other references represent inputs.

In code segment (A), if m_1 is unavailable, the execution gets stuck at the first assignment. Therefore, the availability of m_o depends on the availability of m_1.

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

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

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

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

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

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

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

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

Figure 2. Operational semantics for Aimp

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

Figure 3. Examples

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 m_o depends on the value of m_1. Moreover, the program executes a `while` statement that may diverge before m_o is assigned value 2. Therefore, for the whole program, the availability of m_o depends on the value of m_1.

4. Noninterference properties

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

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

As far as we are aware, no previous work has proposed a notion of noninterference between the availability of outputs and both the integrity and availability of inputs. This formulation of noninterference provides a separation of concerns (and policies) for availability and integrity, yet prevents the two attacks discussed above.

The intuitive concepts of high and low security are based on the power of the potential attacker, which is represented by a base label L. In the DLM, suppose the attacker is able to act for principals $p_1, \ldots, p_n$, and that there exists a top principal (denoted by $*$) that acts for every principal. Then 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 are just the initial memory, and the outputs are the observable aspects of a program execution, which is defined by the *observation model* of the language. In Aimp, we have the following observation model:

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

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

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

Definition 4.1 ($\Gamma \vdash M_1 \approx_{C\leq L} M_2$). The low-confidentiality parts of M_1 and M_2 are indistinguishable, written $\Gamma \vdash M_1 \approx_{C\leq L} M_2$, if for any $m \in \mathit{dom}(\Gamma)$, $C(\Gamma(m)) \leq L$ implies $M_1(m) \approx M_2(m)$.

Definition 4.2 ($\Gamma \vdash M_1 \approx_{I\not\leq L} M_2$). The high-integrity parts of M_1 and M_2 are indistinguishable, written $\Gamma \vdash M_1 \approx_{I\not\leq L} M_2$, 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 ($\Gamma \vdash M_1 \approx_{A\not\leq L} M_2$). The high-availability parts of M_1 and M_2 are equally available, written $\Gamma \vdash M_1 \approx_{A\not\leq L} M_2$, 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. First, we assume that users cannot observe timing. As a result, traces $[M, M]$ and $[M]$ look the same to a user. In general, two traces T_1 and T_2 are equivalent, written $T_1 \approx T_2$, if they are equal up to stuttering, which means the two traces obtained by eliminating repeated elements in T_1 and T_2 are equal. For example, $[M_1, M_2, M_2] \approx [M_1, M_1, M_2]$. Second, T_1 and T_2 are indistinguishable, if T_1 appears to be a prefix of T_2, because in that case, T_1 and T_2 may be generated by the same execution. Given two traces T_1 and T_2 of memories with respect to Γ, let $\Gamma \vdash T_1 \approx_{C\leq L} T_2$ denote that the low-confidentiality parts of T_1 and T_2 are indistinguishable, and $\Gamma \vdash T_1 \approx_{I\not\leq L} T_2$ denote that the high-integrity parts of T_1 and T_2 are indistinguishable. These two notions are defined as follows:

Definition 4.4 ($\Gamma \vdash T_1 \approx_{C\leq L} T_2$). Given two traces T_1 and T_2, $\Gamma \vdash T_1 \approx_{C\leq L} T_2$ 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 $\Gamma \vdash M_i \approx_{C\leq L} M_i'$ for any i in $\{1, \ldots, \texttt{min}(m, n)\}$.

Definition 4.5 ($\Gamma \vdash T_1 \approx_{I\not\leq L} T_2$). Given two traces T_1 and T_2, $\Gamma \vdash T_1 \approx_{I\not\leq L} T_2$ 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 $\Gamma \vdash M_i \approx_{I\not\leq L} M_i'$ for any i in $\{1, \ldots, \texttt{min}(m, n)\}$.

Note that two executions are indistinguishable if any two finite traces generated by those two executions are indistinguishable. Thus, we can still reason about the indistinguishability of two nonterminating executions, even though $\approx_{I\not\leq L}$ and $\approx_{C\leq L}$ 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 $\Gamma \vdash \texttt{NI}_C(s)$, if for any two traces T_1 and T_2 generated by evaluating $\langle s,\, M_1\rangle$ and $\langle s,\, M_2\rangle$, we have that $\Gamma \vdash M_1 \approx_{C\leq L} M_2$ implies $\Gamma \vdash T_1 \approx_{C\leq L} T_2$.

Note that this confidentiality noninterference property does not treat covert channels based on termination and timing. Static control of timing channels is largely orthogonal to this work, and has been partially addressed elsewhere [28, 2, 23].

Definition 4.7 (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 $\Gamma \vdash M_1 \approx_{I\not\leq L} M_2$ implies $\Gamma \vdash T_1 \approx_{I\not\leq L} T_2$.

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

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 $\Gamma\,;\mathcal{R} \vdash \texttt{NI}_A(s)$, if for any two memories M_1, M_2, the following statements

- $\Gamma \vdash M_1 \approx_{I\not\leq L} M_2$
- For $i \in \{1, 2\}$, $\forall m \in \mathit{dom}(\Gamma).\ A(\Gamma(m)) \not\leq L \wedge m \notin \mathcal{R} \Rightarrow M_i(m) \neq \texttt{none}$
- $\langle s,\, M_i\rangle \longmapsto^* \langle s_i',\, M_i'\rangle$ for $i \in \{1, 2\}$

imply that there exist $\langle s_i'',\, M_i''\rangle$ for $i \in \{1, 2\}$ such that $\langle s_i',\, M_i'\rangle \longmapsto^* \langle s_i'',\, M_i''\rangle$ and $\Gamma \vdash M_1'' \approx_{A\not\leq L} M_2''$.

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 [7, 30]. 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: $\Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash s : \texttt{stmt}_{\mathcal{R}'}$, where Γ is the typing assignment, and *pc* is the *program counter* label [6] used to track security levels of the program counter. The typing judgment for expressions has the form $\Gamma\,;\mathcal{R} \vdash e : \tau$

The typing rules are shown in Figure 5. Rules (INT) and (NONE) check constants. An integer n has type $\texttt{int}_\ell$ 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 $\texttt{int}_\ell$.

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

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

Rule (ADD) checks addition expressions. 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 $\Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash s_1 : \texttt{stmt}_{\mathcal{R}_1}$ 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 $\Gamma\,;\mathcal{R}_1\,;\mathit{pc}$. It is clear that s_2 and $s_1; s_2$ terminate at the same point. Thus, $s_1; s_2$ has the same type as s_2.

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

$$[\textit{INT}] \quad \Gamma\,;\mathcal{R} \vdash n : \mathtt{int}_\ell$$

$$[\textit{NONE}] \quad \Gamma\,;\mathcal{R} \vdash \mathtt{none} : \mathtt{int}_\ell$$

$$[\textit{REF}] \quad \frac{\Gamma(m) = \mathtt{int}_\ell}{\Gamma\,;\mathcal{R} \vdash m : \mathtt{int}_\ell\ \mathtt{ref}}$$

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

$$[\textit{ADD}] \quad \frac{\Gamma\,;\mathcal{R} \vdash e_1 : \mathtt{int}_{\ell_1} \qquad \Gamma\,;\mathcal{R} \vdash e_2 : \mathtt{int}_{\ell_2}}{\Gamma\,;\mathcal{R} \vdash e_1 + e_2 : \mathtt{int}_{\ell_1 \sqcup \ell_2}}$$

$$[\textit{SKIP}] \quad \Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash \mathtt{skip} : \mathtt{stmt}_{\mathcal{R}}$$

$$[\textit{SEQ}] \quad \frac{\Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash s_1 : \mathtt{stmt}_{\mathcal{R}_1} \quad \Gamma\,;\mathcal{R}_1\,;\mathit{pc} \vdash s_2 : \mathtt{stmt}_{\mathcal{R}_2}}{\Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash s_1; s_2 : \mathtt{stmt}_{\mathcal{R}_2}}$$

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

$$[\textit{IF}] \quad \frac{\begin{array}{c}\Gamma\,;\mathcal{R} \vdash e : \mathtt{int}_\ell \qquad A_\Gamma(\mathcal{R}) \le A(\ell) \\ \Gamma\,;\mathcal{R}\,;\mathit{pc} \sqcup \ell \vdash s_i : \tau \quad i \in \{1,2\}\end{array}}{\Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash \mathtt{if}\ e\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 : \tau}$$

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

$$[\textit{SUB}] \quad \frac{\Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash s : \tau \qquad \Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash \tau \le \tau'}{\Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash s : \tau'}$$

Figure 4. Typing rules for Aimp

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

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

Rule (IF) checks `if` statements. Consider the statement `if` e `then` s_1 `else` s_2. The value of e determines which branch is executed, so the program-counter labels for branches s_1 and s_2 subsume the label of e to protect e from implicit flows. As usual, the `if` statement has type τ if both s_1 and s_2 have type τ. As in rule (ASSIGN), the premise $A_\Gamma(R) \le A(\ell)$ ensures that e has sufficient availability.

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

For example, consider the code segments (B) and (C) in Figure 3, in which $\mathcal{R} = \{m_o\}$. Suppose $A(\Gamma(m_o)) \not\le L$. In (B), the constraint $A_\Gamma(\mathcal{R}) \le I(\ell)$ of rule (WHILE) ensures $I(\Gamma(m_1)) \not\le L$, so attackers cannot affect the value of m_1, and whether the `while` statement diverges. In (C), the constraint $A_\Gamma(\mathcal{R}) \le I(\mathit{pc})$ guarantees $I(\mathit{pc}) \not\le L$, and thus $I(\Gamma(m_1)) \not\le L$ holds because $I(\mathit{pc}) \le I(\Gamma(m_1))$. Therefore, attackers cannot affect which branch of the `if` statement would be taken, or whether control reaches the `while` statement.

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

$$[\textit{ST}] \quad \frac{\begin{array}{c}\mathcal{R}' \subseteq \mathcal{R}'' \subseteq \mathcal{R} \\ \forall m,\ \ m \in \mathcal{R}'' - \mathcal{R}' \Rightarrow A(\Gamma(m)) \le I(\mathit{pc})\end{array}}{\Gamma\,;\mathcal{R}\,;\mathit{pc} \vdash \mathtt{stmt}_{\mathcal{R}'} \le \mathtt{stmt}_{\mathcal{R}''}}$$

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

Consider the assignment $m_o := 1$ in code segment (D) of Figure 3. By rule (ASSIGN), $\Gamma\,;\{m_o\}\,;\mathit{pc} \vdash m_o := 0 : \mathtt{stmt}_\emptyset$. For the `else` branch of the `if` statement, we have $\Gamma\,;\{m_o\}\,;\mathit{pc} \vdash \mathtt{skip} : \mathtt{stmt}_{\{m_o\}}$. By rule (IF), $\Gamma\,;\{m_o\}\,;\mathit{pc} \vdash m_o := 0 : \mathtt{stmt}_{\{m_o\}}$ needs to hold, which requires $\Gamma\,;\{m_o\}\,;\mathit{pc} \vdash \mathtt{stmt}_\emptyset \le \mathtt{stmt}_{\{m_o\}}$. In this example, the availability of m_o depends on which branch is taken, and we need to ensure $A(\Gamma(m_o)) \le I(\Gamma(m_1))$. Indeed, if (D) is well typed, by rules (ST) and (IF), we have $A(\Gamma(m_o)) \le I(\mathit{pc}) \le I(\Gamma(m_1))$.

This type system satisfies the subject reduction property. Moreover, we can prove that any well-typed program has confidentiality, integrity and availability noninterference properties. These results are formalized in the following two theorems (see the technical report [34] for the proofs).

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 so 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 2 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:

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

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.

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

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

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

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 the following:

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

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:

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

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 that 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 $\Gamma \vdash \mathtt{NI}_I(s)$, if for any two traces T_1 and T_2 generated by evaluating $\langle s, M_1 \rangle$ and $\langle s, M_2 \rangle$, we have that $\Gamma \vdash M_1 \approx_{I \not\leq L} M_2$, $\Gamma \vdash M_1 \approx_{A \not\leq L} M_2$ 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:

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

The name x is used to range over a set of reference variables. The `new` statement $\mathtt{new}\, x{:}\ell_x = \mathtt{ref}(\ell)\, \mathtt{in}\, s$ creates a new reference m with type $\mathtt{int}_\ell\ \mathtt{ref}$, 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 $\mathtt{new}\, x{:}\ell_x = \mathtt{ref}(\ell)\, \mathtt{in}\, s$, 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}⟩) in
  x := 1;
```

6.2.1. Operational semantics. Formally, the following rule is used to evaluate the `new` statement:

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

The function $\mathit{newloc}(M, \ell_x)$ deterministically returns a fresh reference m such that $m \notin \mathit{dom}(M)$. 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 Ω from the memory space $\mathcal{M}$ (an infinite set of memory locations) to labels. Given a label ℓ, let $\mathcal{M}_\ell = \{m \mid m \in \mathcal{M} \wedge \Omega(m) = \ell\}$. In addition, we assume that for any ℓ, $\mathcal{M}_\ell$ is infinite. The function *newloc*(M, ℓ_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. We 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 $I(\Omega(m)) \not\leq L$ 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*(Γ) such that $I(\Omega(m)) \not\leq L$, 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, $\Omega(m) = \langle \bot_C, \top_I, \top_A \rangle$. 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.

6.2.2. Typing. 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 $\texttt{new}\ x : \ell_x = \texttt{ref}(\ell)\ \texttt{in}\ s$. Suppose the typing environment for the `new` statement is "$\Gamma\,;\mathcal{R}\,;$ *pc*". Then the typing environment for s should be "$\Gamma, x : \texttt{int}_\ell\,;\mathcal{R} \cup \{x\}\,;$ *pc*". Second, to control the implicit information flow arising from the creation of a new reference, the typing rule for checking the statement $\texttt{new}\ x : \ell_x = \texttt{ref}(\ell)\ \texttt{in}\ s$ 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 $A(\ell) \leq I(pc)$ by rule (WHILE). Therefore, for the above code to be well-typed, $A(\ell) \leq I(\Gamma(m))$ needs to hold, 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 track the program counter label for each reference variable x from the program point where x is created. Accordingly, 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 $\{\bot_C, \top_I, \top_A\}$.

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

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 bit, we use $\Delta \sqcup \ell$ to denote the program counter map Δ' that satisfies *dom*(Δ) = *dom*(Δ') and $\Delta'(r) = \Delta(r) \sqcup \ell$ for any $r \in$ *dom*(Δ). In addition, let $\Delta(r, pc)$ denote $\Delta(r)$ if $r \in$ *dom*(Δ), and *pc* if otherwise. The adjusted typing rules are shown as follows:

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

$$[\textit{IF}]\quad \frac{\Gamma\,;\mathcal{R} \vdash e : \texttt{int}_\ell \qquad A_\Gamma(\mathcal{R}) \leq A(\ell) \qquad \Gamma\,;\mathcal{R}\,;\Delta \sqcup \ell\,;pc \sqcup \ell \vdash s_i : \tau \quad i \in \{1, 2\}}{\Gamma\,;\mathcal{R}\,;\Delta\,;pc \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \tau}$$

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

6.2.3. Example: TCP handshake protocol. The TCP connection establishment process uses a three-step handshake protocol [26]. First, a client host h_c sends a $\texttt{SYN}(h)$

packet that contains the address of host h to a server s. Second, the server sends a SYN_ACK packet to host h. Third, host h sends an ACK or RST packet to the server, depending on whether h is h_c. An instance of this protocol can be simulated by the following code, in which message communications are modeled by assignments:

```
m := !m_{i,h_c};            // receive SYN from h_c
new x_end:⟨⊥,⊥,⊤⟩ = ref(⟨s:s, s:s, s:h_c⟩) in
  m_{o,h} := !m + 1;      // send SYN_ACK to h
  m := !m_{i,h};          // receive ACK/RST from h
  x_end := 1
```

The reference m_{i,h_c} represents the connection request from h_c. After the request is received, a new reference x_{end} is created to capture the availability requirement of the server: the protocol will terminate if the client h_c does not fail, which is specified by the availability label $s : h_c$ of x_{end}.

The statement $m := m_{i,h}$ represents the third step of the handshake protocol, and the reference $m_{i,h}$ represents the response from h. Intuitively, the availability of $m_{i,h}$ only depends on h, and thus we suppose that the availability label of $m_{i,h}$ is $s : h$. Then the above code is not well-typed because $s : h_c \not\leq s : h$. Interestingly, this reflects the problem with the handshake protocol that allows the SYN flooding attack: host h may be spoofed and cannot be trusted to establish the connection between s and h_c.

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 [25, 17, 4]; the other deals with faulty clients and defends denial of service attacks [31, 19, 13]. 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 [15]. 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 [9]. It is well known that a noninterference property is not a property on traces [18], 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.

Focardi and Gorrieri [8] provide a classification of security properties in the setting of a non-deterministic process algebra. In particular, the BNDC (*bisimulation-based nondeducibility on compositions*) property prevents attackers from affecting the availabilities of observable process actions. However, the BNDC property requires observational equivalence, making it difficult to separate the concerns for integrity and availability.

Yu and Gligor [31] develop a formal method for analyzing availability: a form of first-order temporal logic is used to specify safety and liveness constraints on the inputs and behaviors of a service, and then those constraints can be used to formally verify the availability guarantees of the service. The flexibility and expressiveness of first-order temporal logic come at a price: it is difficult to automate the verification process. The approach of formalizing and reasoning system constraints and guarantees in terms of logic resembles the rely-guarantee method [12], which was also applied to analyzing cryptographic protocols by Guttman et al. [10].

Lafrance and Mullins [14] define a semantic security property *impassivity* for preventing DoS attacks. Intuitively, impassivity means that low-cost actions cannot interfere with high-cost actions. In some sense, impassivity is an integrity noninterference property, if we treat low-cost as low-integrity and high-cost as high-integrity. With the implicit assumption that high-cost actions may exhaust system resources and render a system unavailable, impassivity corresponds to one part of our notion of availability noninterference: low-integrity inputs cannot affect the availabilities of highly available outputs.

Li et al. [16] formalize the notion that highly available data does not depend on low-availability data. However, their definition is *termination-insensitive* [24], which makes it inappropriate to model availability noninterference.

Volpano and Smith [29] introduce the notion of *termination agreement*, which requires two executions indistinguishable to low-confidentiality users to both terminate or both diverge. The integrity dual of termination agreement can be viewed as a special case of the availability noninterference in which termination is treated as the only output of a program.

Language-based information flow control techniques [7,

24, 30, 11, 32, 22, 3, 33] 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 [20]. This paper generalizes the DLM to provide a unified framework for specifying confidentiality, integrity and availability policies. The form of a combined security policy is an instance of an *owned policy* [5], though we give a different semantics here.

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.

An important direction for future work is to apply this static availability analysis framework to multithreaded programming models, and develop a notion of possibilistic (or probabilistic) availability noninterference.

Acknowledgements

The authors would like to thank Andrei Sabelfeld, Fred B. Schneider, Stephen Chong and Lorenzo Alvisi for their insightful suggestions and comments on this work. In addition to the anonymous reviewers, Nate Nystrom, Michael Clarkson, Kevin O'Neill and Jed Liu all helped improve this paper.

This research was supported in part by the Department of the Navy, Office of Naval Research, ONR Grant N00014-01-1-0968, and by National Science Foundation grants 0208642, 0133302, and 0430161. Andrew Myers is supported by an Alfred P. Sloan Research Fellowship. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright annotation thereon. The views and conclusions here are those of the authors and do not necessarily reflect the views of these sponsors.

References

[1] M. Abadi, M. Burrows, B. W. Lampson, and G. D. Plotkin. A calculus for access control in distributed systems. *TOPLAS*, 15(4):706–734, 1993.

[2] J. Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, Jan. 2000.

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

[4] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proc. 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999.

[5] H. Chen and S. Chong. Owned policies for information security. In *Proc. 17th IEEE Computer Security Foundations Workshop*, June 2004.

[6] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[7] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[8] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1995.

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

[10] J. D. Guttman and et al. Trust management in strand spaces: A rely-guarantee method. In *Proc. European Symposium on Programming*, pages 325–339, Apr. 2004.

[11] N. Heintze and J. 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, Jan. 1998.

[12] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Prog. Lang. Syst.*, 5(4):596–619, 1983.

[13] A. Juels and J. 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.

[14] S. Lafrance and J. Mullins. Using admissible interference to detect denial of service vulnerabilities. In *6th International Workshop on Formal Methods*, pages 1–19, July 2003.

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

[16] P. Li, Y. Mao, and S. Zdancewic. Information integrity policies. In *Proceedings of the Workshop on Formal Aspects in Security and Trust*, Sept. 2003.

[17] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. of the 29th ACM Symposium on Theory of Computing*, pages 569–578, El Paso, Texas, May 1997.

[18] J. 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.

[19] J. K. Millen. A resource allocation model for denial of service. In *Proc. IEEE Symposium on Security and Privacy*, pages 137–147, Oakland, CA, USA, May 1992.

[20] A. C. Myers and B. 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.

[21] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, Oct. 2000.

[22] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 319–330, 2002.

[23] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proceedings of the 9th International Static Analysis Symposium*, volume 2477 of *LNCS*, Madrid, Spain, Sept. 2002. Springer-Verlag.

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

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

[26] C. L. Schuba and et al. Analysis of a denial of service attack on TCP. In *Proc. IEEE Symposium on Security and Privacy*, pages 208–223, May 1997.

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

[28] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 355–364, San Diego, California, Jan. 1998.

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

[30] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[31] C.-F. Yu and V. D. Gligor. A specification and verification method for preventing denial of service. In *Proc. IEEE Symposium on Security and Privacy*, pages 187–202, Oakland, CA, USA, Apr. 1988.

[32] S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2–3):209–234, Sept. 2002.

[33] L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *Proc. 2nd Workshop on Formal Aspects in Security and Trust, IFIP TC1 WG1.7*. Springer, Aug. 2004.

[34] L. Zheng and A. C. Myers. End-to-end availability policies and noninterference. Technical Report 2005–xxxx, Cornell University Computing and Information Science, 2005.