Learning is Change in Knowledge:
Knowledge-based Security for Dynamic Policies

Aslan Askarov
Harvard University

Stephen Chong
Harvard University

***Abstract*—In systems that handle confidential information, the security policy to enforce on information frequently changes: new users join the system, old users leave, and sensitivity of data changes over time. It is challenging, yet important, to specify what it means for such systems to be secure, and to gain assurance that a system is secure.**

We present a language-based model for specifying, reasoning about, and enforcing information security in systems that dynamically change the security policy. We specify security for such systems as a simple and intuitive extensional knowledge-based semantic condition: an attacker can only learn information in accordance with the current security policy.

Importantly, the semantic condition is parameterized by the ability of the attacker. Learning is about change in knowledge, and an observation that allows one attacker to learn confidential information may provide a different attacker with no new information. A program that is secure against an attacker with perfect recall may not be secure against a more realistic, weaker, attacker.

We introduce a compositional model of attackers that simplifies enforcement of security, and demonstrate that standard information-flow control mechanisms, such as security-type systems and information-flow monitors, can be easily adapted to enforce security for a broad and useful class of attackers.

I. Introduction

Given the wealth of confidential information handled by many different computer systems, it is important to ensure that systems enforce appropriate security on the information that they manipulate. But what constitutes "appropriate security" changes over time, even during a single execution of the system: users join the system and can now view confidential information; users leave the system and are no longer allowed to view confidential information; the sensitivity of data changes over time, affecting the set of users that can view the data.

In the presence of such dynamic changes to the desired security policy, it is challenging to specify what it means for a system to be secure, let alone to gain assurance that a system correctly enforces security. Previous approaches that aim to provide strong information security either ignore the dynamic nature of security policies in their security guarantee (e.g., [32, 33]), or introduce complex and unintuitive definitions of security and/or use non-standard enforcement mechanisms (e.g., [7, 17, 34]).

We present a language-based model for specifying, reasoning about, and enforcing information security in systems that dynamically change the security policy. Our semantic security condition is simple, intuitive, extensional, and defined in terms of an attacker's knowledge: an attacker that can observe program execution should learn information only in accordance with the current security policy.

Importantly, the semantic security condition is parameterized on the abilities of the attacker. Knowledge-based semantic security conditions are intended to restrict what and when an attacker learns information. Learning is about change in knowledge, and so we must consider how the knowledge of an attacker changes as it makes observations. In many practical scenarios, the perfect recall attacker is too strong. An observation that gives a weak attacker new information may be "old news" for a strong attacker with perfect recall. We enforce our semantic security condition for a broad and practical set of attackers using straightforward adaptations of standard information-flow security mechanisms (e.g., [4, 28]).

We regard a security policy $\sqsubseteq$ as a relation over a set L of security levels. Intuitively, security policies specify what information flows are permitted between security levels. Policies must be reflexive, and may be (but are not required to be) partial orders or lattices. Policies can encode information-flow lattices [12] and intransitive relations such as those used for intransitive noninterference [23, 35]. We assume that the set of security levels L is fixed, but allow the policy to change during program execution.

Motivating scenario. Consider a company's document-management system that is accessible by all of the company's employees. It contains many documents, some of which are sensitive, meaning that only certain employees may use them. As an employee joins or leaves the company, or is promoted or transferred, the set of documents that the employee may access changes. The information security policy for this document-management system specifies who may use which documents, and it changes over time. Indeed, the security policy itself is part of the state of the system, and part of the system's functionality is the ability for some users to modify the security policy.

Even a relatively simple document-management system can reveal information about documents in unexpected ways. For example, suppose the documents are indexed to facilitate search. Because the index contains information about keywords that appear in sensitive documents, the index contains sensitive information: if a user is given unrestricted access to the index, the user may learn confidential information she is not permitted to learn. Traditional access control mecha-

Appeared in CSF 2012

nisms would not suffice to enforce the desired security, since a user may learn information about documents through the index without ever attempting to access the document.

We thus seek to enforce strong information security, defined in terms of the knowledge of agents interacting with the system. The definition of security, and the enforcement of it, is complicated by the dynamic nature of security policies. However, our definition is simple and intuitive: an agent that observes program execution should learn information only if permitted by the current security policy.

The following program allows user U to learn about documents classified as $Nuclear$ (that is, changes the current security policy $\sqsubseteq$ so that $(Nuclear, U) \in \sqsubseteq$), and outputs the keywords of document $nuke_1$ to user U. (We assume that the indexing functionality of the system can determine the keywords of a document, and the keywords of a document reveal information about the document's content.) The program then removes permission for U to learn about $Nuclear$ documents (that is, changes the current security policy $\sqsubseteq$ so that $(Nuclear, U) \notin \sqsubseteq$, perhaps due to U being reassigned to a different department), and outputs the keywords of document $nuke_2$ to U. Both documents are classified as *Nuclear*.

P_1 : *Allow info flow from* $Nuclear$ *to* U
Output $\mathsf{keywords}(nuke_1)$ *to* U
Disallow info flow from $Nuclear$ *to* U
Output $\mathsf{keywords}(nuke_2)$ *to* U

This program is insecure, in that user U learns information about document $nuke_2$ at a time when the security policy does not allow it. User U also learns about document $nuke_1$, but does so when the security policy permits it.

Attacker model. Consider the following code, that outputs confidential information to user U at a time when U is permitted to learn the information, and the same information again at a time when U is not permitted.

P_2 : *Allow info flow from* $Nuclear$ *to* U
Output $\mathsf{keywords}(nuke_1)$ *to* U
...
Disallow info flow from $Nuclear$ *to* U
Output $\mathsf{keywords}(nuke_1)$ *to* U

Should this program be regarded as secure or not? At the first output, U learns information about document $nuke_1$. If U is a powerful attacker who remembers the first output, then when the confidential information is output again, U learns nothing new. From our description of the security condition, this program is secure: the attacker learns information only in accordance with the currently enforced policy. However, intuitively, we would like to regard this program as insecure: it outputs information about $nuke_1$ to U at a time when this is not permitted. Indeed, for a more realistic attacker, who may not remember every output it has observed, the last output may enable the attacker to learn information about $nuke_1$ at a time when this isn't permitted.

A suitable definition of security should permit us to reject both programs P_1 and P_2, even though one of them is secure against a powerful attacker. We therefore parameterize our definition of security with respect to an attacker: a program is secure against attacker A if A learns information only in accordance with the current security policy.

Security thus depends on the ability of the attacker. Rather than being an artifact of our technical development, we believe that this is a fundamental and important notion. Our intuitions about information security revolve around the idea of restricting what an attacker is permitted to learn. Learning is about *change in knowledge*: an attacker learns something from an event if the attacker's knowledge after the event is more precise than the attacker's knowledge before. In the presence of dynamic security policies, it is not sufficient to simply consider the most powerful possible attacker: we must also consider changes in the knowledge of weaker, more realistic, attackers.

Ideally we would like to ensure that a program is secure against as many attackers as possible. However, it may not be possible to be secure against all possible attackers (as we will discuss later). This begs the question "which set of attackers we should strive to ensure security against?"

We develop a compositional theory of attackers that gives insight into the security condition, and simplifies enforcement. Specifically, we show that there is a class of *simple attackers* that are easy to reason about, and if a program is secure against this class of simple attackers then the program is secure against many other attackers, including logically-omniscient attackers with perfect recall, and with bounded memory. This compositional theory both enables reasoning about the security of programs, and simplifies enforcement.

Contributions. This work makes three key contributions.

- **We present a novel semantic security condition suitable for dynamic security policies.** The security condition is intuitive, extensional, and knowledge-based, and is suitable for a language that permits arbitrary changes to the security policy. The semantic security condition is parameterized on an attacker, and a program may be secure against a powerful attacker and insecure against a weaker attacker.
- **We present both static and dynamic techniques to enforce this semantic security condition.** These techniques elegantly extend existing information-flow control techniques to handle dynamic security policies in a language that can be extended with expressive security-relevant features.
- **We introduce a compositional model of attackers, which simplifies both the security condition and the enforcement mechanisms.** We prove that enforcing security for a simple and intuitive class of attackers implies security for a much broader class of attackers.

$$
\begin{array}{lll}
\text{Values} & v & ::= n \\
\text{Expressions} & e & ::= v \mid x \mid e_1 \oplus e_2 \\
\text{Commands} & c & ::= \mathsf{skip} \mid x := e \mid c_1; c_2 \\
 & & \quad \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c \\
 & & \quad \mid \mathsf{input}\ x\ \mathsf{from}\ \ell \mid \mathsf{output}\ e\ \mathsf{to}\ \ell \\
 & & \quad \mid \mathsf{setPolicy}(\sqsubseteq)
\end{array}
$$

Figure 1. Language syntax

II. Language

We present a simple imperative interactive language that contains an explicit command for changing the current security policy. The language can perform input and output on channels. We assume that there is one channel for each security level in L. Our semantic security conditions will be concerned with protecting the confidentiality of inputs received on channels.

Syntax. Figure 1 presents the language syntax. Language commands are standard, with the exception of input and output commands, and command $\mathsf{setPolicy}(\sqsubseteq)$, which sets the current security policy to $\sqsubseteq$. We do not specify how security policies are denoted, but assume that some mechanism exists. In later sections, we will extend the language to allow security policies to be specified using language mechanisms. Input command $\mathsf{input}\ x\ \mathsf{from}\ \ell$ receives an input from channel ℓ and assigns the value to variable x. Output command $\mathsf{output}\ e\ \mathsf{to}\ \ell$ evaluates expression e and outputs the resulting value on channel ℓ.

Expressions e consist of program variables x, values v, and binary operations over expressions. We use $\oplus$ to range over total binary relations over values. For simplicity, we restrict values to integers n.

Semantics. A *memory* is a function from program variables to values. We use metavariable m to range over memories.

An *input stream* is a sequence of values representing the pending inputs on a channel. We use metavariable vs to range over input streams, and write $v : vs$ for the input stream with first element v, and remaining elements vs. An *input environment* is a function from L to input streams. Metavariable w ranges over input environments. For input environment w and security level $\ell \in L$, $w(\ell)$ is the input stream for channel ℓ. Note that, because our language is deterministic, it is sufficient to model input via streams—Clark and Hunt show that, for deterministic programs, quantification over all streams expresses arbitrary interactive input strategy [11].

A *configuration* is a tuple $\langle c, m, w, \sqsubseteq \rangle$ consisting of command c, memory m, input environment w, and security policy $\sqsubseteq$. Command c is the remainder of the program to execute, m is the current memory, w is the current input environment, and $\sqsubseteq$ is the current security policy. Figure 2 presents an operational semantics for the language. Judgment $\langle c, m, w, \sqsubseteq \rangle \longrightarrow_\alpha \langle c', m', w', \sqsubseteq' \rangle$ means that configuration $\langle c, m, w, \sqsubseteq \rangle$ can take a single step to configuration $\langle c', m', w', \sqsubseteq' \rangle$, optionally emitting an *event* α. Events are

$$\frac{m(e) = v}{\langle x := e, m, w, \sqsubseteq \rangle \longrightarrow_\epsilon \langle \mathsf{skip}, m[x \mapsto v], w, \sqsubseteq \rangle}$$

$$\frac{\langle c_1, m, w, \sqsubseteq \rangle \longrightarrow_\alpha \langle c_1', m', w', \sqsubseteq' \rangle}{\langle c_1; c_2, m, w, \sqsubseteq \rangle \longrightarrow_\alpha \langle c_1'; c_2, m', w', \sqsubseteq' \rangle}$$

$$\frac{}{\langle \mathsf{skip}; c, m, w, \sqsubseteq \rangle \longrightarrow_\epsilon \langle c, m, w, \sqsubseteq \rangle}$$

$$\frac{m(e) \neq 0}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, m, w, \sqsubseteq \rangle \longrightarrow_\epsilon \langle c_1, m, w, \sqsubseteq \rangle}$$

$$\frac{m(e) = 0}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, m, w, \sqsubseteq \rangle \longrightarrow_\epsilon \langle c_2, m, w, \sqsubseteq \rangle}$$

$$\frac{}{\langle \mathsf{while}\ e\ \mathsf{do}\ c, m, w, \sqsubseteq \rangle \longrightarrow_\epsilon \langle \mathsf{if}\ e\ \mathsf{then}\ (c; \mathsf{while}\ e\ \mathsf{do}\ c)\ \mathsf{else}\ \mathsf{skip}, m, w, \sqsubseteq \rangle}$$

$$\frac{}{\langle \mathsf{setPolicy}(\sqsubseteq'), m, w, \sqsubseteq \rangle \longrightarrow_\epsilon \langle \mathsf{skip}, m, w, \sqsubseteq' \rangle}$$

$$\frac{w(\ell) = v : vs}{\langle \mathsf{input}\ x\ \mathsf{from}\ \ell, m, w, \sqsubseteq \rangle \longrightarrow_{i(v,\ell)} \langle \mathsf{skip}, m[x \mapsto v], w[\ell \mapsto vs], \sqsubseteq \rangle}$$

$$\frac{m(e) = v}{\langle \mathsf{output}\ e\ \mathsf{to}\ \ell, m, w, \sqsubseteq \rangle \longrightarrow_{o(v,\ell)} \langle \mathsf{skip}, m, w, \sqsubseteq \rangle}$$

Figure 2. Language semantics

either *input events* $i(v, \ell)$ or *output events* $o(v, \ell)$, indicating, respectively, the input or output of value v on channel ℓ. We use $\alpha = \epsilon$ to indicate that no event was emitted during the execution step. We use $\mathbb{E}$ to denote the set of all possible events, and $\mathbb{E}(\ell)$ to denote the set of possible events on channel ℓ.

$$\mathbb{E}(\ell) = \{i(v, \ell) \mid v \text{ is a value}\} \cup \{o(v, \ell) \mid v \text{ is a value}\}$$
$$\mathbb{E} = \bigcup_{\ell \in L} \mathbb{E}(\ell)$$

We write $m(e) = v$ to indicate that expression e evaluates to value v using memory m to look up the value of program variables. We write $m[x \mapsto v]$ for the memory that maps program variable x to value v and otherwise behaves the same as memory m. Similarly, we write $w[\ell \mapsto vs]$ for the input environment that maps channel ℓ to input stream vs and otherwise behaves the same as w.

The inference rules for the semantics are mostly standard. Command $\mathsf{setPolicy}(\sqsubseteq)$ modifies the configuration to make policy $\sqsubseteq$ the current policy. Input command $\mathsf{input}\ x\ \mathsf{from}\ \ell$ inputs value v from input stream $w(\ell)$, updates the memory to map x to v, updates the input environment to remove v from input stream $w(\ell)$, and emits event $i(v, \ell)$. Output command $\mathsf{output}\ e\ \mathsf{to}\ \ell$ evaluates e to value v, and emits

event $o(v, \ell)$.

We assume that there is a distinguished memory m_{init} and a distinguished security policy $\sqsubseteq_{init}$ that are used as the initial memory and security policy, respectively, for any program execution. For concreteness, we assume in the rest of the paper that the initial security policy is the identity relation over security levels: $\sqsubseteq_{init} = \sqsubseteq_{Id} = \{(\ell, \ell) \mid \ell \in L\}$.

Our semantic security conditions will be concerned with the confidentiality of initial input environments. As such, there is no distinguished initial input environment.

Traces. Traces are finite sequences of events. We use metavariable t to range over traces, and write $t_1 \cdot t_2$ for the concatenation of traces t_1 and t_2. We write ϵ for the empty trace (and also use it to denote the absence of an event in an execution step). We write $|t|$ for the length of trace t.

We write $t \restriction \ell$ for the restriction of trace t to events on channel ℓ. More formally, we have

$$\epsilon \restriction \ell = \epsilon$$
$$(\alpha \cdot t) \restriction \ell = \begin{cases} \alpha \cdot (t \restriction \ell) & \text{if } \alpha \in \mathbb{E}(\ell) \\ t \restriction \ell & \text{if } \alpha \notin \mathbb{E}(\ell). \end{cases}$$

We say that configuration $\langle c_0, m_0, w_0, \sqsubseteq_0 \rangle$ *emits trace t on channel ℓ ending with policy $\sqsubseteq_k$* (written $\langle c_0, m_0, w_0, \sqsubseteq_0 \rangle \Downarrow_\ell (t, \sqsubseteq_k)$) if there are $k+1$ configurations $\langle c_i, m_i, w_i, \sqsubseteq_i \rangle$ for $i \in 0..k$ such that

$$\langle c_{i-1}, m_{i-1}, w_{i-1}, \sqsubseteq_{i-1} \rangle \longrightarrow_{\alpha_i} \langle c_i, m_i, w_i, \sqsubseteq_i \rangle$$

for all $i \in 1..k$, and $t = (\alpha_1 \cdot \ldots \cdot \alpha_k) \restriction \ell$, and $t \neq (\alpha_1 \cdot \ldots \cdot \alpha_{k-1}) \restriction \ell$.

Intuitively, if $\langle c, m_{init}, w, \sqsubseteq_{init} \rangle \Downarrow_\ell (t, \sqsubseteq)$ then an observer of channel ℓ may observe trace t during the execution of command c with initial input environment w, and policy $\sqsubseteq$ is the policy enforced when the last event of t was emitted.

III. Security

We define security of a program in terms of the knowledge of an attacker that observes program execution. Conceptually, the definition of security is straightforward: an execution of a program is secure if an attacker learns information about the initial input environment only in accordance with the current security policy.

In this section, we define attackers as entities that observe the execution of a program, and define the knowledge of an attacker. We then state two versions of the semantic security condition and explore some of the consequences.

A. Attackers and attacker knowledge

As discussed in the introduction, a program may be secure against a powerful attacker, but insecure against a weaker attacker. We thus define attackers, and will parameterize our definition of security with respect to the attacker that is observing program execution.

An *attacker* is a state-based machine that observes a subset of events during a program's execution, and updates its state accordingly. We assume that all attackers know the source code of the program generating the events, and that attackers are logically omniscient. Attackers differ in their ability to remember the observations they have made. We will define the attacker's knowledge to be the set of initial input environments that could have resulted in a sequence of observations that caused the attacker to be in its current state.

Formally, attacker A is a tuple $A = (S_A, s_{init}, \delta_A)$ where

- S_A is a set of attacker states;
- $s_{init} \in S_A$ is the initial attacker state; and
- $\delta_A : S_A \times \mathbb{E} \to S_A$ is the transition function that describes how the attacker's state changes due to events the attacker observes. Note that δ_A is a function, and so state transitions are deterministic.

Given trace t and attacker $A = (S_A, s_{init}, \delta_A)$, we write $A(t)$ to denote the attacker's state after observing trace t.

$$A(\epsilon) = s_{init}$$
$$A(t \cdot \alpha) = \delta_A(A(t), \alpha)$$

We assume that attacker A is able to observe only events on a single channel, and refer to that channel as the *level of A*.

Note while this is not uncommon in literature [27], this is different from conventional approaches where attacker observes events from all channels ℓ' such that $\ell' \sqsubseteq \ell$. In our case, the choice is more than a matter of preference—the conventional definition would be unsuitable because of the dynamic nature of $\sqsubseteq$.

Example attackers. We give four examples of attackers that will be of later interest. The "perfect attacker" A_{Per} has perfect recall: it remembers all observations. The set of states for A_{Per} is the set of traces. The attacker's initial state is the empty trace, and the transition function concatenates the latest observable event to the attacker's state: $\delta_{A_{Per}}(s, \alpha) = s \cdot \alpha$. The perfect attacker knows more than any other attacker (which will be stated and proved later).

A significantly weaker class of attackers are "bounded memory" attackers, $A_{last\text{-}i}$, that remember the last i observed events (and the total number of events observed). More formally, the set of attacker states is the set of pairs of natural numbers (counting the number of observations), and traces of up to length i. The initial state is the pair $(0, \epsilon)$, and the transition function is defined as $\delta_{A_{last\text{-}i}}((j, t), \alpha) = (j + 1, t')$ where t' is equal to the last k events of $t \cdot \alpha$, where k is the minimum of i and the length of $t \cdot \alpha$.

Another class of attackers that will be of interest are the "i-th event only" attackers $A_{i\text{-}only}$ which simply count the number of observed events, and remember only the ith event. More formally, the set of attacker states is the set of pairs of natural numbers (counting the number of observations), and, optionally, events (recording the ith event): $S_{A_{i\text{-}only}} = \mathbb{N} \times \mathbb{E}$. The initial state is $(0, \epsilon)$. The transition function is defined

as follows.

$$\delta_{A_{i\text{-}only}}((j,\alpha),\alpha') = \begin{cases} (i,\alpha') & \text{if } j+1=i \\ (j+1,\alpha) & \text{otherwise} \end{cases}$$

An even weaker attacker is the "no memory" attacker $A_\emptyset$ who has only a single state s_{init}, and thus pays no attention to any observations: $\delta_{A_\emptyset}(s_{init},\alpha) = s_{init}$.

Attacker knowledge. Given program c, we define the *knowledge* of attacker A with current state s and level ℓ to be the set of initial input environments that could have resulted in the attacker's current state by observing execution of command c. We write $k(c,A,\ell,s)$ for the attacker's knowledge, and define it as follows.

$$k(c,A,\ell,s) = \{w \mid \exists t.\ \langle c, m_{init}, w, \sqsubseteq_{init}\rangle \Downarrow_\ell (t,\sqsubseteq) \text{ and } A(t) = s\}$$

Intuitively, the set $k(c,A,\ell,s)$ is the set of initial input environments that attacker A with state s believes are possible. Thus, a smaller set means that the attacker has better, or more precise, knowledge.

The perfect attacker A_{Per} has the most precise knowledge out of any possible attacker, since A_{Per} remembers all observable events.

Theorem 1. *Let $A = (S_A, s_{init}, \delta_A)$ be an attacker. Then for all commands c, all security levels ℓ, all initial input environments w, and all traces t such that $\langle c, m_{init}, w, \sqsubseteq_{init}\rangle \Downarrow_\ell (t,\sqsubseteq)$ we have*

$$k(c,A,\ell,A(t)) \quad \supseteq \quad k(c,A_{Per},\ell,A_{Per}(t)).$$

Proofs of this and other theorems are available in the accompanying technical report [2]

B. Security definition

Our definition of security is, intuitively, that the attacker learns information only in accordance with the current security policy. Since we are interested in protecting the initial input environment, we need to define what the attacker is permitted to learn about the initial input environment. Towards this end, we define an equivalence relation over input environments: *ℓ-equivalence according to $\sqsubseteq$*, written $\approx_\ell^\sqsubseteq$. Two input environments are related to each other by $\approx_\ell^\sqsubseteq$ if the two input environments have identical input streams for all security levels ℓ' that are permitted to flow to level ℓ according to policy $\sqsubseteq$. Intuitively, if $w \approx_\ell^\sqsubseteq w'$ then policy $\sqsubseteq$ would not allow an attacker with level ℓ to distinguish the input environments w and w'. More formally,

$$w \approx_\ell^\sqsubseteq w' \iff \forall \ell'.(\ell',\ell) \in \sqsubseteq \Rightarrow w(\ell') = w'(\ell').$$

We write $[w]_\ell^\sqsubseteq$ to denote the equivalence class of w under relation $\approx_\ell^\sqsubseteq$.

Given this interpretation of how security policies are intended to restrict attacker knowledge of initial input environments, we can now state the definition of security.

Definition 1 (Security for input environment w)**.** Command c is *secure* against attacker $A = (S_A, s_{init}, \delta_A)$ with level ℓ for initial input environment w if for all traces t, events α, and policies $\sqsubseteq$ such that $\langle c, m_{init}, w, \sqsubseteq_{init}\rangle \Downarrow_\ell ((t\cdot\alpha),\sqsubseteq)$ we have

$$k(c,A,\ell,A(t\cdot\alpha)) \quad \supseteq \quad k(c,A,\ell,A(t)) \ \cap\ [w]_\ell^\sqsubseteq.$$

Security requires that, for each observation α that the attacker makes, the attacker's new knowledge—$k(c,A,\ell,A(t\cdot\alpha))$, the left-hand side of the equation—is no more precise than the combination of the attacker's previous knowledge and the knowledge about the initial input environment allowed by the current policy—$k(c,A,\ell,A(t)) \cap [w]_\ell^\sqsubseteq$, the right-hand side of the equation.

C. Security against attackers

The definition of security depends on the knowledge of the attacker. Thus, different programs will be secure against different attackers. For example, consider the following program, P_3, a version of the pseudocode program from the introduction. There are three security levels $L = \{A, B, C\}$, policy $\sqsubseteq_{AB\to C}$ allows information flow from level A to C and from level B to C, and $\sqsubseteq_{B\to C}$ allows information flow from B to C, but not from A to C.

P_3 : $\sqsubseteq_{AB\to C} = \sqsubseteq_{Id} \cup \{(A,C),(B,C)\}$
$\sqsubseteq_{B\to C} = \sqsubseteq_{Id} \cup \{(B,C)\}$

input a from A;
input b from B;
setPolicy($\sqsubseteq_{AB\to C}$);
output $a+b$ to C;
setPolicy($\sqsubseteq_{B\to C}$);
output b to C

This program reads inputs from channels A and B, stores them in variables a and b, sets the policy to $\sqsubseteq_{AB\to C}$, and outputs $a+b$ to channel C. It then sets the policy to $\sqsubseteq_{B\to C}$ and outputs variable b to channel C. It is secure against the "no memory" attacker $A_\emptyset$ with any level ℓ. Indeed, *every* program is secure against $A_\emptyset$, since the knowledge of $A_\emptyset$ never changes.

All executions of P_3 are secure against A_{Per} with level C. Consider an execution with A_{Per} observing channel C. The attacker observes the sum $a+b$, say 12. At this point in the execution, the attacker's knowledge is the set of all initial input environments w such that the first elements of input streams $w(A)$ and $w(B)$ sum to 12. This change in knowledge is permitted by the current security policy $\sqsubseteq_{AB\to C}$ which allows the attacker to learn anything and everything about the initial input streams $w(A)$ and $w(B)$ (since $(A,C) \in \sqsubseteq_{AB\to C}$ and $(B,C) \in \sqsubseteq_{AB\to C}$).

The attacker next observes the value of variable b, say 7. The attacker's knowledge has improved: it now knows the exact values for the first elements of input streams $w(A)$ and $w(B)$. The current security policy is $\sqsubseteq_{B\to C}$. Clearly

the attacker has learned the first element of the initial input stream $w(B)$, which is permitted by policy $\sqsubseteq_{B\to C}$. But the attacker has also learned the first element of the initial input stream $w(A)$, which is not permitted by the policy. Nonetheless, the execution is secure, since the attacker's new knowledge is no more precise than what may be obtained by combining the attacker's previous knowledge with information it is permitted to learn according to the current security policy.

Security against the perfect attacker A_{Per} does not imply security against all attackers. Consider program P_4 below, which differs from P_3 only in the final command, which outputs variable a to channel C instead of b.

$$
\begin{array}{ll}
P_4: & \mathsf{input}\ a\ \mathsf{from}\ A;\\
& \mathsf{input}\ b\ \mathsf{from}\ B;\\
& \mathsf{setPolicy}(\sqsubseteq_{AB\to C});\\
& \mathsf{output}\ a+b\ \mathsf{to}\ C;\\
& \mathsf{setPolicy}(\sqsubseteq_{B\to C});\\
& \mathsf{output}\ a\ \mathsf{to}\ C
\end{array}
$$

During execution of P_4, the knowledge of A_{Per} is identical to the knowledge of A_{Per} during an execution of P_3. Thus, program P_4 is secure against A_{Per} with attacker level C. But intuitively the program is insecure! It outputs variable a to channel C even though the current security policy does not allow information flow from A to C.

Indeed, this program is insecure against attacker $A_{2\text{-}only}$ with level C, which remembers only the second observed event and whose knowledge suddenly improves from the set of all possible initial input environments to only those input environments where the first element of input stream $w(A)$ matches the observed output.

Although security against perfect attacker A_{Per} does not imply security against all attackers, security against some attackers does imply security against other attackers. Specifically, given some some set of attackers $\{A_j\}_{j\in J}$, if a program is secure against A_j for all $j \in J$, then the program is also secure against an attacker that combines the knowledge of all A_j. That is, the program is also secure against an attacker whose knowledge equals the intersection of the knowledge of A_j for all $j \in J$.

Theorem 2. *Let $\{A_j\}_{j\in J}$ be a set of attackers, and for all $j \in J$, let c be secure against attacker A_j with level ℓ. Let A be an attacker such that for all initial input environments w and traces t such that $\langle c, m_{init}, w, \sqsubseteq_{init}\rangle \Downarrow_\ell (t, \sqsubseteq)$ we have*

$$k(c, A, \ell, A(t)) \quad = \quad \bigcap_{j\in J} k(c, A_j, \ell, A_j(t)).$$

Then c is secure against attacker A with level ℓ.

Theorem 2 implies that if a program is secure against $A_{i\text{-}only}$ with level ℓ for all possible values of i, then c is secure against many other attackers with level ℓ, including the perfect attacker A_{Per}, and all bounded memory attackers $A_{last\text{-}j}$ for all j.

$\sqsubseteq_{H\to A} = \sqsubseteq_{Id} \cup \{(H, A)\}$

$\mathsf{input}\ x\ \mathsf{from}\ H;$
$\mathsf{setPolicy}(\sqsubseteq_{H\to A});$
$\mathsf{output}\ (x > 0\ ?\ 1 : 2)\ \mathsf{to}\ A;$
$\mathsf{setPolicy}(\sqsubseteq_{Id});$
$\mathsf{output}\ 3\ \mathsf{to}\ A$

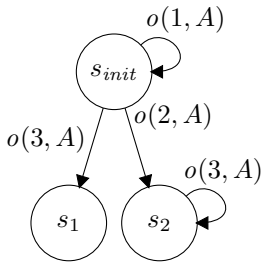

Figure 3. Example program and attacker A_{insec}

Theorem 3. *For all commands c, if for all i, c is secure against $A_{i\text{-}only}$ with level ℓ, then c is secure against attacker A_{Per} with level ℓ.*

Theorem 4. *For all commands c, if for all i, c is secure against $A_{i\text{-}only}$ with level ℓ, then for all j, c is secure against attacker $A_{last\text{-}j}$ with level ℓ.*

These results (Theorems 2, 3, and 4) simplify both security reasoning and security enforcement. It allows both a system developer and an enforcement mechanism to consider the security of a program just with respect to attacker $A_{i\text{-}only}$ for all i: a set of simple attackers that are easy to understand. Security against attackers $A_{i\text{-}only}$ will imply security for a large and practical set of attackers.

However, security against all $A_{i\text{-}only}$ does not imply security against all attackers. For example, Figure 3 shows a program and an attacker A_{insec} such that the program is secure against $A_{i\text{-}only}$ with level ℓ for all possible values of i, but is insecure against A_{insec}. In an execution where the input from H is positive, then once A_{insec} observes the output $o(3, A)$, the attacker learns that the input was positive at a time when it is not permitted to learn this information. Intuitively, this is because the attacker's state machine does not record that it saw the event $o(1, A)$, which would let it learn that the input was positive at an appropriate time.

Attacker A_{insec} is, in some ways, being willfully stupid: it remembers observing the value 2 (by transitioning to state s_2) but ignores observing the value 1. It is not an attacker for which we particularly care whether our system is secure against. By contrast, we believe that the attackers A_{Per} and $A_{last\text{-}i}$ are a useful, relevant, and realistic set of attackers, for which it is reasonable to ensure a system is secure against. Thus, enforcing security against $A_{i\text{-}only}$ is a worthwhile endeavor, even if it does not imply security against all attackers.

Characterizing such "willfully stupid" attackers is future work, as is the identification of a definitive class of attackers against which programs should be secure.

D. Progress-insensitive security

The definition of security given in Definition 1 is *progress sensitive* [4]: it accounts for knowledge the attacker may gain

by observing progress of program execution. For example, consider the following program.

$$\begin{array}{ll} P_5: & \mathsf{output}\ 0\ \mathsf{to}\ L; \\ & \mathsf{input}\ x\ \mathsf{from}\ H; \\ & \mathsf{while}\ x > 0\ \mathsf{do\ skip}; \\ & \mathsf{output}\ 1\ \mathsf{to}\ L \end{array}$$

An attacker observing channel L sees the second output event if and only if the while loop terminates. Since the loop guard depends on an input from channel H (which policy $\sqsubseteq_{init}$ does not allow to be learned by an observer of channel L), program P_5 is insecure according to Definition 1.

However, many practical enforcement techniques ignore information flow via the observation of progress, and enforce a definition of security that allows attackers to learn arbitrary information by observing progress. We present a *progress insensitive* version of the security definition to enable standard enforcement techniques. Indeed, the enforcement techniques we present in Section V enforce progress-insensitive security, defined below.

We define *progress knowledge* [4] as the set of initial input environments that could have led to the attacker's current state s, *and* can produce another event observable by the attacker (which may or may not change the attacker's state).

$$\begin{aligned} k^+(c, A, \ell, s) = \{w \mid \exists t.\ \exists \alpha \in \mathbb{E}(\ell).\ \\ \langle c, m_{init}, w, \sqsubseteq_{init}\rangle \Downarrow_\ell ((t \cdot \alpha), \sqsubseteq) \text{ and } A(t) = s\} \end{aligned}$$

Progress knowledge is more precise than the attacker's knowledge, in that for all commands c, attackers A, attacker levels ℓ, and attacker states s, we have $k(c, A, \ell, s) \supseteq k^+(c, A, \ell, s)$.

We modify our previous definition of security to *progress-insensitive security*, which allows the attacker to learn that the program makes progress. That is, the attacker knowledge at each step is no more precise than the combination of the attacker's previous knowledge, the information the current security policy allows to be learned, and the knowledge that the program makes progress. The definition of progress-insensitive security differs from the definition of progress-sensitive security (Definition 1) only in the replacement of $k(c, A, \ell, A(t))$ with $k^+(c, A, \ell, A(t))$.

Definition 2 (Progress-insensitive security for input environment w)**.** Command c is *progress-insensitively secure* against attacker $A = (S_A, s_{init}, \delta_A)$ with level ℓ for initial input environment w if for all traces t, events α, and policies $\sqsubseteq$ such that $\langle c, m_{init}, w, \sqsubseteq_{init}\rangle \Downarrow_\ell ((t \cdot \alpha), \sqsubseteq)$ we have

$$k(c, A, \ell, A(t \cdot \alpha)) \quad \supseteq \quad k^+(c, A, \ell, A(t)) \ \cap\ [w]^{\sqsubseteq}_{\ell}.$$

As with progress-sensitive security, the left-hand side of the equation is the attacker's new knowledge, and the right-hand side of the equation provides a bound on how precise the attacker's knowledge is allowed to be.

Program P_5 satisfies progress-insensitive security (Definition 2) against A_{Per} with level L for all input environments, but does not satisfy progress-sensitive security (Definition 1) against A_{Per} with level L for an initial input environment w where the first element of $w(H)$ is not positive.

The results of this section (specifically, Theorems 2, 3 and 4) hold for both progress-insensitive and progress-sensitive security.

E. Noninterference, declassification, and revocation

Noninterference [15] is a semantic security condition that requires that "high security" events do not interfere with, or affect, "low security" events. While there are many definitions of noninterference, most relevant here are knowledge-based definitions for interactive models (e.g., [4, 5, 10]).

The definitions of security, both progress sensitive and progress insensitive, generalize noninterference. More precisely, if the security policy never changes from the initial security policy, then progress-(in)sensitive security implies progress-(in)sensitive noninterference [3, 4].

Declassification weakens noninterference, to allow some high-security information to be observed by low-security observers. There are at least two ways that declassification can be viewed or incorporated into our model. First, if the security policy changes from $\sqsubseteq$ to $\sqsubseteq'$, where $\sqsubseteq' \supseteq \sqsubseteq$, then the new security policy allows more information flows. This can be viewed as a coarse-grained form of declassification, since previously disallowed information flows are now permitted. Second, finer-grained policies, described in Section VI, can be added into our model to allow partial flows between security levels.

Revocation occurs when security privileges are removed. In our model this corresponds to removing a flow that was previously allowed. Our security conditions allow previous knowledge of the attacker to persist, but allows new knowledge to be acquired only in accordance with the current security policy.

IV. First-class security policies

Systems with dynamic security policies typically have a run-time representation of the security policy. For example, in a document management system that attempts to restrict which users may use which documents, users' permissions would be represented in a data structure, and the data structure examined and modified at runtime.

In this section, we extend our language to encode security policies at run time, which allows the program to inspect and manipulate security policies. The extension simplifies our semantics, requires no changes to the security definitions of Section III, and does not require complex enforcement mechanisms.

We encode the security policy in memory using a subset of the program variables. Let Λ be a function from $L \times L$ to variables, so that variable $\Lambda(\ell, \ell')$ encodes whether the

current security policy allows information flow from ℓ to ℓ'. For brevity, we write $\Lambda_{\ell,\ell'}$ to denote variable $\Lambda(\ell,\ell')$. Flow is permitted from ℓ to ℓ' if either $\ell = \ell'$ (since policies must be reflexive) or the current memory maps variable $\Lambda_{\ell,\ell'}$ to a non-zero value. Given memory m, we write $\sqsubseteq_m$ for the security policy represented by m, defined as follows.

Definition 3 (Interpretation of the security policies $\sqsubseteq_m$)**.**

$$\sqsubseteq_m = \{(\ell,\ell') \mid \ell = \ell' \vee m(\Lambda_{\ell,\ell'}) \neq 0\}$$

The program may modify the current security policy by updating the variables $\Lambda_{\ell,\ell'}$. Thus, the encoding removes the need for a specialized syntax for policy changes, and $\mathsf{setPolicy}(\sqsubseteq)$ can now be dropped from the language syntax. Moreover, the program can perform run-time tests to determine whether information flow is permitted from ℓ to ℓ' simply by inspecting variable $\Lambda_{\ell,\ell'}$.

Program configurations $\langle c, m, w, \sqsubseteq\rangle$ no longer need to explicitly represent the current security policy $\sqsubseteq$, since it can be inferred from the current memory. Of course, given a semantics over configurations $\langle c, m, w\rangle$ that do not explicitly represent the current security policy, we can lift the semantics to those of Section II, and can thus carry over the security definitions of Section III with no modifications.

Consider the following example program where communication occurs between a server, abbreviated to S and a client, abbreviated to C, until the server chooses to terminate communication. This program satisfies both progress-sensitive and progress-insensitive security for attacker $A_{i\text{-}only}$ with any level for all i.

$$\begin{array}{ll} P_6: & \Lambda_{C,S} := 1; \Lambda_{S,C} := 1; \\ & \mathsf{input}\ y\ \mathsf{from}\ S; \\ & \mathsf{while}\ \Lambda_{S,C} \neq 0\ \mathsf{do} \\ & \quad \mathsf{input}\ x\ \mathsf{from}\ C; \\ & \quad \mathsf{output}\ x > y\ \mathsf{to}\ C; \\ & \quad \mathsf{input}\ t\ \mathsf{from}\ S; \\ & \quad \mathsf{output}\ t\ \mathsf{to}\ C; \\ & \quad \Lambda_{S,C} := t; \\ & \mathsf{output}\ 0\ \mathsf{to}\ C \end{array}$$

The first line of this program establishes a security policy in which information can flow from C to S, and vice-versa. The first input reads variable y from S. In the body of the loop, we input values from C into variable x. Because information can flow from S to C, the result of the expression $x > y$ can also flow to C, and therefore the corresponding output statement is allowed. We input variable t from S, and use that value to update the security policy, possibly disallowing flow from S to C. However, before updating the security policy ($\Lambda_{S,C} := t$) we output the value of t to C, so that C learns whether the security policy will change. This notification is needed; otherwise, when C observes the output of 0 at the end of the program, C would learn information about inputs from S at a time when it is not allowed by the current security policy.

As another example, suppose user U wants to upload content to web server W. Moreover, the server decides whether the user is allowed to upload content. Program P_7 models this scenario.

$$\begin{array}{ll} P_7: & \Lambda_{W,U} := 1; \\ & \mathsf{input}\ \Lambda_{U,W}\ \mathsf{from}\ W; \\ & \mathsf{if}\ \Lambda_{U,W} \neq 0\ \mathsf{then} \\ & \quad \mathsf{input}\ \mathit{file}\ \mathsf{from}\ U; \mathsf{output}\ \mathit{file}\ \mathsf{to}\ W \\ & \mathsf{else} \\ & \quad \mathsf{skip} \end{array}$$

This program is both progress-sensitively and progress-insensitively secure against attacker $A_{i\text{-}only}$ with any level for all i. Variable $\Lambda_{U,W}$ is updated based on input from W. The conditional statement checks whether flow is allowed from U to W before transferring the file. This program is secure, because the upload occurs only when the information flow is permitted by the current security policy.

V. Enforcement

Sections III and IV present an expressive language and knowledge-based security conditions. The security conditions describe permitted information flow in the presence of dynamic security policies, even when the security policy is derived from runtime constructs of the program.

In this section we present both static and dynamic enforcement techniques for progress-insensitive security against attacker $A_{i\text{-}only}$ with any level, for all i. By the results of Section III, this implies progress-insensitive security against many other attackers, including the perfect attacker A_{Per} and all bounded memory attackers $A_{last\text{-}j}$. Progress-sensitive security can be enforced by more restrictive variations of these enforcement techniques (e.g., [4, 27]).

Our enforcement techniques are adapted from existing information-flow control mechanisms. Like previous work, we track information flow using a security lattice [12]. However, we use the powerset of security levels to track information flow, and check that information flows conform to the current enforced security policy only at program locations that generate observable events.

The absence of an event on a channel may reveal information to an observer. To control this information flow in the presence of dynamic security policies, we introduce a novel mechanism—*channel context bounds*—that tracks bounds on information flows arising from decisions to produce events on channels, a form of *implicit information flow* [12].

We note that, apart from channel context bounds, the enforcement mechanisms described in this section are largely standard. Soundness of these mechanisms is, nevertheless, an important contribution because it shows that dynamic security policies can be enforced using known techniques. Throughout the section we point out to a range of extensions that can improve accuracy of these techniques.

$$\frac{}{\Gamma, \Delta, pc \vdash \mathsf{skip}} \qquad \frac{pc \cup \Gamma(e) \subseteq \Gamma(x)}{\Gamma, \Delta, pc \vdash x := e}$$

$$\frac{\Gamma, \Delta, pc \vdash c_i, i = 1, 2}{\Gamma, \Delta, pc \vdash c_1; c_2} \qquad \frac{\Gamma, \Delta, pc \cup \Gamma(e) \vdash c_i, i = 1, 2}{\Gamma, \Delta, pc \vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2}$$

$$\frac{\Gamma, \Delta, pc \cup \Gamma(e) \vdash c}{\Gamma, \Delta, pc \vdash \mathsf{while}\ e\ \mathsf{do}\ c} \qquad \frac{\begin{array}{c} pc \cup \{\ell\} \subseteq \Gamma(x) \quad pc \subseteq \Delta(\ell) \\ \forall \ell' \in \Delta(\ell).\ \mathit{may\text{-}flow}(\ell', \ell) \end{array}}{\Gamma, \Delta, pc \vdash \mathsf{input}\ x\ \mathsf{from}\ \ell}$$

$$\frac{pc \subseteq \Delta(\ell) \qquad \forall \ell' \in \Delta(\ell) \cup \Gamma(e).\ \mathit{may\text{-}flow}(\ell', \ell)}{\Gamma, \Delta, pc \vdash \mathsf{output}\ e\ \mathsf{to}\ \ell}$$

Figure 4. Typing rules

A. Static enforcement

We present a type system that enforces security for $A_{i\text{-}only}$ with any level, for all i. The type system is a mostly standard information-flow control security-type system (e.g., [28, 36]). A *security-type context* Γ is a function from program variables to sets of security levels. Intuitively, if the value of program variable x at any point in the program's execution may reveal information about the initial input stream for channel ℓ, then $\ell \in \Gamma(x)$. We introduce *channel context bounds* Δ that map channels to sets of security levels, and for each channel provide an upper bound on decisions to produce an observable event on that channel. Intuitively, if a decision to produce an input or output on channel ℓ may reveal information about the initial input stream for channel ℓ', then $\ell' \in \Delta(\ell)$.

Type judgment $\Gamma, \Delta, pc \vdash c$ means that command c is well-typed under security-type context Γ, channel context bounds Δ, and *program counter levels* pc, which is a set of security levels such that if information at level ℓ might have influenced control flow reaching command c, then $\ell \in pc$. Program counter levels, in conjunction with channel context bounds, control *implicit information flows* [12]: information flow through the control flow structure of a program.

Inference rules for judgment $\Gamma, \Delta, pc \vdash c$ are presented in Figure 4. We write $\Gamma(e)$ for the set of security levels of variables occurring in e: $\Gamma(e) = \{\Gamma(x) \mid x \text{ appears in } e\}$. The rules are standard for security type systems except for the use of channel context bounds and the use of sets of security levels to track information flows. We explain these differences in more detail below.

Powerset of security levels. We use sets of security levels to track information flow instead of taking upper bounds of security levels. This is because the security policy, which provides an ordering over security levels, may change during execution, and thus it is difficult to determine statically an "upper bound" of a given set of security levels. By tracking information flow using sets of security levels, we avoid needing to commit to any particular security policy at the time of analysis. At program points that may produce observable events on channel ℓ (i.e., input and output commands), we compute a set of security levels that is an upper bound on the information that the event may reveal ($\Delta(\ell)$ for input statements, and $\Delta(\ell) \cup \Gamma(e)$ for output statements). For every security level ℓ' in the upper bound we check that ℓ' is allowed to flow to level ℓ at that program point, using the function $\mathit{may\text{-}flow}(\ell', \ell)$, described below.

Static approximation of runtime security policy. As in security-type systems with dynamic security levels (e.g., [16, 17, 25, 34, 37]), we use a static analysis to track which information flows are permitted at a given program point. Instead of conflating this analysis with the type system, we assume that this analysis is specified separately, and the results of the analysis are available via the function $\mathit{may\text{-}flow}(\ell_1, \ell_2)$. Note that the analysis needs to be flow sensitive, and we assume the function $\mathit{may\text{-}flow}(\ell_1, \ell_2)$ takes as an implicit argument the program point for which we are querying the analysis results. This could be made explicit by adding labels to all program points (e.g., [26]). The static approximation of the runtime security policy is used to check information flow only at observable events: input and output commands. We assume that this analysis is sound: at a given program point, for some security levels ℓ_1 and ℓ_2, if $\mathit{may\text{-}flow}(\ell_1, \ell_2)$ is true, then in any execution, whenever that program point is reached, we have $(\ell_1, \ell_2) \in \sqsubseteq$, where $\sqsubseteq$ is the current security policy in the configuration at the queried program point. Since the policy is interpreted from the current memory, this implies that the current value in variable Λ_{ℓ_1,ℓ_2} is non-zero. Thus, the analysis could be implemented as a standard constant propagation analysis.

Channel context bounds. We use channel context bounds to track and control information flow arising from the decision to perform an input or output. Channel context bounds restrict the contexts in which channels may be used. Consider the following program.

$$\begin{array}{ll} P_8: & \Lambda_{A,B} := 1; \\ & \mathsf{input}\ x\ \mathsf{from}\ A; \\ & \mathsf{if}\ x > 0\ \mathsf{then} \\ & \quad \mathsf{output}\ 1\ \mathsf{to}\ B \\ & \mathsf{else}\ \mathsf{skip}; \\ & \Lambda_{A,B} := 0; \\ & \mathsf{output}\ 2\ \mathsf{to}\ B; \end{array}$$

This program inputs a value from channel A, and, if that value is positive, outputs 1 to channel B. Information flow is then disallowed from A to B, and the value 2 is output on channel B. Suppose that the first event an attacker with level B sees is $o(2, B)$. At that time, the attacker learns that the input from A was not positive, and thus learns information about A at a time when it is not permitted, violating security. Observation of event $o(2, B)$ informed the attacker that event $o(1, B)$ did not occur. We track this

information flow using channel context bounds: $\Delta(\ell)$ is an upper bound on the information that may be learned by the occurrence or non-occurrence of any event on channel ℓ. It is a superset of the program counter levels at all input and output events on that channel. The use of channel context bounds in our type system ensures that insecure program P_8 above does not type-check.

The type system enforces progress-insensitive security against attacker $A_{i\text{-}only}$ with any level, for all i.

Theorem 5 (Soundness of type system)**.** *For all commands c, security levels ℓ, and $i \in \mathbb{N}$, if there exists a security-type context Γ and a channel context bounds Δ such that $\Gamma, \Delta, \emptyset \vdash c$ then c is progress-insensitively secure against attacker $A_{i\text{-}only}$ with level ℓ for all input environments w.*

For example, Program P_5 (Section III-D) is well-typed under security-type context Γ such that $\Gamma(x) = \{H\}$, and thus progress-insensitively secure against attacker $A_{i\text{-}only}$ with any level, for all i. Similarly, Program P_6 (Section IV) is well-typed if $\Gamma(y) = \Gamma(t) = \{S\}$, $\Gamma(x) = \{R, S\}$, $\Gamma(\Lambda_{R,S}) = \emptyset$, and $\Gamma(\Lambda_{S,R}) = \{S\}$.

The type system presented here assumes a security-type context Γ that describes the levels of information that may be found in variables at any point in the program's execution. This context could be specified in advance, or could be inferred using a standard type inference algorithm. The type system could easily be adapted to a more precise *flow-sensitive security-type system* [18, 19].

B. Dynamic enforcement

Dynamic information-flow monitors (e.g., [4, 13, 20]) control the flow of information in a system by monitoring the system execution, and intervening when necessary. We describe a *purely dynamic* information-flow monitor [30] based closely on the monitor of Askarov and Sabelfeld [4].

To enforce progress-insensitive security, we need to define configurations and inference rules for *monitored execution* of program. A *monitored configuration* has the form $(\langle c, m, w\rangle, st)$, where $\langle c, m, w\rangle$ is a program configuration (where the memory encodes the current security policy), and st is a monitor state. A monitor state is a stack of sets of security levels, and is used to track implicit information flows. The monitor state is analogous to the program counter levels pc used in the security-type system.

We extend our original program semantics to issue *monitor events* to the monitor for assignments, branching, input and output. The monitor can decide whether to accept or reject a monitor event based on the current monitor state. A monitored configuration makes a transition only when the monitor event is accepted by the monitor. When the event is not accepted, program execution halts. This extension of the semantics follows the presentation of Askarov and Sabelfeld [4]. We omit most of the details here, except the monitor rules for input and output monitor events, presented in Figure 5, which describe when the monitor is prepared to accept input and output monitor events.

$$\frac{\bigcup st \subseteq \Delta(\ell) \quad (\bigcup st) \cup \{\ell\} \subseteq \Gamma(x) \quad \forall \ell' \in \Delta(\ell) \;.\; \ell' \sqsubseteq_m \ell}{st \longrightarrow^{i(x,\ell)} st}$$

$$\frac{\bigcup st \subseteq \Delta(\ell) \quad \forall \ell' \in \Gamma(e) \cup \Delta(\ell) \;.\; \ell' \sqsubseteq_m \ell}{st \longrightarrow^{o(e,\ell)} st}$$

Figure 5. Dynamic enforcement of input and output

As in the security-type system, we assume a security-type context Γ and channel context bounds Δ. The monitor ensures that Γ and Δ are bounds on the information that may be learned by examining the values of variables and performing input and output. That is, a value that may reveal information about the initial input stream $w(\ell)$ may only be stored in variable x if $\ell \in \Gamma(x)$. In Figure 5, we write $\bigcup st$ for the union of all elements of the stack st. Recall that a stack element is a set of security levels. The set $\bigcup st$ describes which security levels could have influenced control flow reaching the current program point.

Interestingly, to check whether information flow is permitted from ℓ to ℓ', the monitor must inspect the current security policy $\sqsubseteq_m$. Such inspection may reveal sensitive information, since it depends on the value of variable $\Lambda_{\ell,\ell'}$, which may reveal information about levels $\Gamma(\Lambda_{\ell,\ell'})$. However, the resulting information channel cannot be magnified [5].

In dynamic enforcement, unlike static enforcement, Γ and Δ need to be available at runtime. We call such executions Γ, Δ-*monitored* executions. This also requires lifting the definitions of knowledge and progress knowledge to be parameterized over Γ and Δ. Using these definitions of knowledge it is straightforward to lift the definition of progress-insensitive security to Γ, Δ-*monitored progress-insensitive security*.

Soundness of the dynamic enforcement states that monitored executions satisfy the lifted progress-insensitive security for attacker $A_{i\text{-}only}$ with any level, for all i.

Theorem 6 (Soundness of dynamic enforcement)**.** *For all commands c, security levels ℓ, input environments w, security-type contexts Γ, channels contexts bounds Δ, and $i \in \mathbb{N}$, it holds that Γ, Δ-monitored executions of c is progress-insensitively security against attacker $A_{i\text{-}only}$ with level ℓ for input environment w.*

VI. Extensions

Our language-based model for information security in the presence of dynamic security policies is simple, expressive, and can be enforced using practical information-flow control techniques. In this section we highlight the simplicity by sketching two extensions to the model: the addition of first-class security levels, and of fine-grained security policies.

Values	$v ::= \ldots \mid \ell$
Expressions	$e ::= \ldots \mid e_1 \,\mathsf{flows\text{-}to}\, e_2$
Commands	$c ::= \ldots \mid \mathsf{input}\, x \,\mathsf{from}\, e \mid \mathsf{output}\, e_1 \,\mathsf{to}\, e_2$

Figure 6. Modified language syntax for first-class security levels

These security-relevant extensions require only little change to the definitions of security and enforcement techniques.

A. First-class security levels

We add security levels as first-class values to the language, and extend the type system of Section V-A to enforce progress-insensitive security for this new language. This increases the expressiveness of the language, bringing it closer to realistic systems that represent and manipulate both security levels and security policies at run time.

The addition of first-class security levels does *not* change either the observational model or the definitions of security. We believe that this supports our design choice in the observational model and definitions of security: the addition of a security-relevant language feature changes just the enforcement mechanisms.

We extend the language to allow values to include security levels $\ell \in L$, and add expression $e_1 \,\mathsf{flows\text{-}to}\, e_2$ to test whether flow is permitted between levels. We extend input and output commands to specify the channel using arbitrary expressions. Figure 6 presents the modified syntax.

Language semantics remain the same, modulo extending the set of values to include security levels, and changing the evaluation of input and output commands to evaluate the channel expression. Definitions of attacker knowledge, progress knowledge, and progress-sensitive and progress-insensitive security remain unchanged.

The following program demonstrates the increased expressiveness of the language. Provided the security policy allows the appropriate flows, the program inputs a security level x from A, reads a value from channel x, and echoes it to channel B. We assume that expression $e_1 \,\mathsf{flows\text{-}to}\, e_2$ evaluates to one if the current security policy permits information flow from e_1 to e_2 and zero otherwise.

P_9 :
```
input x from A;
if A flows-to x then input y from x else skip;
if (x flows-to B and A flows-to B) then
  output y to B
else skip
```

This program satisfies progress-sensitive and progress-insensitive security for attacker $A_{i\text{-}only}$ with any level, for all i. An attacker with level x upon observing the input event may learn the first input from channel A. Thus, the program must check that flow is permitted from A to x. An attacker with level B upon observing the output event may learn both the value input from x and that the first input from channel A was x. Thus the program must check that flow is permitted both from x to B and from A to B.

We can modify the type system of Section V-A to enforce progress-insensitive security for the extended language. The modified type system is described in Appendix A.

B. Fine-grained policies

The definitions of security and progress-insensitive security allow an attacker's knowledge to improve only in accordance with the current security policy. So far we have considered only coarse-grained security policies $\sqsubseteq$ such that if $(\ell', \ell) \in \sqsubseteq$ then an attacker at security level ℓ is allowed to learn everything about the initial input stream of channel ℓ'. This is expressed in the security conditions by the set $[w]_\ell^\sqsubseteq$ of input environments that policy $\sqsubseteq$ does not allow an attacker at security level ℓ to distinguish from the initial input environment w.

It is natural and straightforward to consider information flows at finer granularity. Equivalence relations over the initial input environments provide a flexible and expressive way of specifying what information an attacker is permitted to learn [14, 31].

Let a *fine-grained policy* $P = \{ \approx_\ell \}_{\ell \in L}$ be a family of equivalence relations over initial input environments, indexed by $\ell \in L$, and write $[w]_\ell^P$ for the equivalence class of input environment w under equivalence relation $\approx_\ell \in P$.

Intuitively, fine-grained policy P describes for each security level ℓ what information an observer at level ℓ is allowed to learn. Fine-grained policies generalize security policies.

We modify language configurations to replace policies $\sqsubseteq$ with fine-grained policies P, and assume there is some distinguished fine-grained policy P_{init} used as the initial fine-grained policy for every execution.

The definition of security generalizes in the obvious way for fine-grained security policies, where equivalence class $[w]_\ell^P$ is used to restrict how the knowledge of an attacker is allowed to improve. We give the definition of *fine-grained progress-sensitive security* below; the definition of *fine-grained progress-insensitive security* is similar, except that progress knowledge is used instead of attacker knowledge.

Definition 4 (Fine-grained progress-sensitive security)**.** Command c is *fine-grained progress-sensitively secure* against attacker $A = (S_A, s_{init}, \delta_A)$ with level ℓ for initial input environment w if for all traces t, events α, and fine-grained policies P such that

$$\langle c, m_{init}, w, P_{init} \rangle \Downarrow_\ell ((t \cdot \alpha), P)$$

we have

$$k(c, A, \ell, A(t \cdot \alpha)) \quad \supseteq \quad k(c, A, \ell, A(t)) \;\cap\; [w]_\ell^P .$$

Enforcement. There are existing enforcement mechanisms for certain classes of fine-grained policies, such as security-type systems (e.g., [29]) and information-flow monitors (e.g., [4, 21]). These mechanisms can be adapted to enforce fine-grained progress-sensitive security by modifications similar

to those described in Section V. To wit, checks that information flow conforms to policy should be performed at the program locations that generate observable events, and otherwise information flow should be tracked without any assumptions as to which policy is currently enforced.

VII. Related work

Broberg and Sands define *flow locks* [7–9] which specify conditions when information may flow between security levels. A flow lock may be explicitly opened or closed by a program, enabling or disabling information flow between levels. As such, flow locks allow for the dynamic updating of security policy. The semantic security condition for flow locks is knowledge based and quantifies over attackers with different observational abilities (which is analogous to our quantification over output channels ℓ). The type system is of similar precision to ours. The key distinction between flow locks and this work is the semantic security condition: we consider security with respect to attackers with different abilities, while the work on flow locks is only concerned with perfect recall attackers. We believe that this is a crucial step forward in knowledge-based semantic security conditions. Such conditions are intended to restrict what an attacker learns, and when. Since learning is about change in knowledge, one must consider how the knowledge of different attackers change with observations: an attacker with perfect recall may not learn anything new from an observation, but the same observation may allow a more realistic weaker attacker to learn confidential information.

Hicks et al. [17] consider dynamic updating of information-flow policies. They permit arbitrary updates to the security policy, and introduce the semantic security condition *noninterference between updates*, which requires that the program satisfies a form of noninterference between any two consecutive updates to the security policy. There is no security guarantee across security updates. By contrast, our semantic security condition provides a guarantee across arbitrary updates. They enforce their security condition using nonstandard mechanisms, including *permission tags*, a form of type coercion between security levels that are inserted into execution only in accordance with the current policy, but may be reduced at any time. By contrast, our choice of observational model and security condition allows us to use simple adaptations of standard information-flow control mechanisms to enforce security. We adapt the enforcement mechanisms to delay checking of permitted information flows until the production of observable events.

The Rx programming language [34], by Swamy et al., extends the policy update work of Hicks et al. [17] to make it more practical. They represent security policies using *owned roles*, derived from the RT role-based trust-management frameworks [22]. They identify the problem of *transitive flows*, which inadvertently allow information flows that are permitted by neither the old nor new policy. They prevent transitive flows by using a transactional mechanism to ensure that if the security of a program fragment depends upon flow allowed by the current security policy, then those portions of the current security policy are not modified until the fragment finishes execution. Our static enforcement mechanism does not exhibit any transitive flows, but a transactional mechanism would potentially enable the $may\text{-}flow(\cdot,\cdot)$ analysis to be more precise.

Rx uses *metapolicies* to control information that may be revealed through policy updates. This is needed because policies are not first class in Rx. Our model does not require metapolicies since the program is able to query and manipulate the security policy using standard program constructs (see Section IV). This justifies our design choice of cleanly separating the run-time behavior of the language from security policies: the definition of security and the semantics of security policies depend on the run-time behavior of programs, but not vice versa.

The Jif programming language [25] represents security principals at run time, and allows security principals to dynamically delegate their authority to other principals. Since delegation between security principals defines the security policy, Jif allows dynamic security updates. Broberg and Sands [9] encode Jif's Decentralized Label Model [24] in flow locks, and thus provide a semantic security condition for Jif. We believe that this work, provides a promising alternative approach to modeling Jif's run-time representation of security principals, and providing a semantic security condition for Jif that includes reasoning about the different abilities of different principals.

As discussed in Section III-E, dynamically updatable security policies can enable a coarse-grained form of declassification, in which all flows between two security levels are permitted (possibly temporarily) after being previously disallowed. Other work has considered coarse-grained declassification. Mantel and Sands [23] present a language in which a more permissive security policy is used when control flow is within a *downgrading command*. Their semantic security condition is bisimulation-based, and the intuition is that in each step, information flows according to the current security policy. Almeida Matos and Boudol [1] also present a language where lexical scope enables more flows with a bisimulation-based security condition. Information flow from security level A to security level B is allowed within the scope of a $\mathsf{flow}\ A \prec B\ \mathsf{in}\ c$ command.

Balliu et al. [6] point out to the connection of the set-based definitions of attacker knowledge (that our definition is an instance of) to the epistemic account of knowledge. We believe that our intuition on security in the presence of dynamic policies also applies to when information flow policies are expressed using temporal epistemic logic; formal development of such result may provide for an interesting and fruitful future research direction.

VIII. Conclusion

We have presented a simple, elegant, and extensible model for reasoning about information security in the presence of dynamic security policies. We use an extensional knowledge-based semantic security condition, which can be enforced using adaptations of standard information-flow control techniques, in a language that permits arbitrary changes to the security policy.

The semantic security condition is straightforward and intuitive: an attacker should learn information only in accordance with the current security policy. The semantic security condition is parameterized on an attacker, and a program may be secure for a powerful attacker, yet insecure for a weaker attacker. We identify a class of simple attackers such that if a program is secure against all members of this class, the program is secure against many more attackers, including the most powerful possible attacker, and attackers with bounded memory. We present mechanisms that enforce security for this class of simple attackers, and thus also enforce security for a useful and realistic set of attackers.

The language can be easily extended with expressive security-relevant features, such as run-time representation of the security policy, first-class security levels, and fine-grained security policies, without significant change to the semantic security condition.

We believe that this language-based model provides a promising platform on which to build practical systems with strong information security guarantees.

Acknowledgements

We thank Niklas Broberg, Scott Moore, Andrew Myers, Danfeng Zhang, and the anonymous reviewers for detailed feedback about earlier versions of this work. We also thank the Chalmers/KTH Security Workshop for feedback and lively discussion related to this work. This research is supported by the National Science Foundation under Grants No. 1054172 and No. 0964409, and by the Office of Naval Research under Grant No. N000140910652.

References

[1] A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 226–240, Washington, DC, USA, 2005. IEEE Computer Society.

[2] A. Askarov and S. Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. Technical Report TR-02-12, Harvard School of Engineering and Applied Sciences, 2012.

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

[4] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2009.

[5] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security*, Oct. 2008.

[6] M. Balliu, M. Dam, and G. Le Guernic. Epistemic temporal logic for information flow security. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, PLAS 2011. ACM, June 2011.

[7] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Proceedings of the 15th European Symposium on Programming*, pages 180–196. Springer, 2006.

[8] N. Broberg and D. Sands. Flow-sensitive semantics for dynamic information flow policies. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, June 2009.

[9] N. Broberg and D. Sands. Paralocks – role-based information flow control and beyond. In *Proceedings of the 37th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2010.

[10] S. Chong. Required information release. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium*, pages 215–227, July 2010.

[11] D. Clark and S. Hunt. Non-interference for deterministic interactive programs. In *Proc. 5th International Workshop on Formal Aspects in Security and Trust (FAST2008)*, Lecture Notes in Computer Science, Malaga, Spain, October 2008. Springer-Verlag.

[12] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[13] J. S. Fenton. Memoryless subsystems. *Computer Journal*, 17(2):143–147, May 1974.

[14] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 186–197, New York, NY, USA, 2004. ACM Press.

[15] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, Apr. 1982.

[16] R. Grabowski and L. Beringer. Noninterference with dynamic security domains and policies. In *13th Asian Computing Science Conference, Focusing on Information Security and Privacy*, 2009.

[17] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Proceedings of the Foundations of Computer Security*

Workshop, pages 7–18, June 2005.

[18] S. Hunt and D. Sands. On flow-sensitive security types. In *Conference Record of the Thirty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 79–90, New York, NY, USA, Jan. 2006. ACM Press.

[19] S. Hunt and D. Sands. From exponential to polynomial-time security typing via principal types. In *Proceedings of the 20th European Symposium on Programming*, 2011.

[20] G. Le Guernic. Automaton-based Confidentiality Monitoring of Concurrent Programs. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 218–232, 2007.

[21] G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. *Proceedings of the 11th Annual Asian Computing Science Conference*, pages 75–89, 2006.

[22] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society, May 2002.

[23] H. Mantel and D. Sands. Controlled Declassification based on Intransitive Noninterference. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems*, volume 3303 of *Lecture Notes in Computer Science*, pages 129–145. Springer-Verlag, Nov. 2004.

[24] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 186–197. IEEE Computer Society, May 1998.

[25] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, 2001–2008.

[26] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2nd edition, 2005.

[27] K. R. O'Neill, M. R. Clarkson, and S. Chong. Information-flow security for interactive programs. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, pages 190–201. IEEE Computer Society, June 2006.

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

[29] A. Sabelfeld and A. C. Myers. A model for delimited release. In *Proceedings of the 2003 International Symposium on Software Security*, number 3233 in Lecture Notes in Computer Science, pages 174–191. Springer-Verlag, 2004.

[30] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proceedings of Andrei Ershov International Conference on Perspectives of System Informatics*, pages 352–365, 2009.

[31] A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. In *Proceedings of the 8th European Symposium on Programming*, pages 40–58, London, UK, 1999. Springer.

[32] V. Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.

[33] D. Stefan, A. Russo, J. Mitchell, and D. Mazieres. Flexible dynamic information flow control in haskell. In *ACM SIGPLAN Haskell Symposium 2011*, 2011.

[34] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, pages 202–216. IEEE Computer Society, 2006.

[35] R. van der Meyden. What, indeed, is intransitive noninterference? In *Proceedings of the 12th European Symposium On Research In Computer Security*, volume 4734 of *Lecture Notes in Computer Science*, pages 235–250. Springer, Sept. 2007.

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

[37] L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *Formal Aspects in Security and Trust*, Toulouse, France, Aug. 2004.

APPENDIX

A. Static enforcement for first-class security levels

We modify the type system of Section V-A to enforce progress-insensitive security for the extended language. The codomain of security-type contexts Γ is now *labeled types* τ_S where $\tau \in \{\mathbf{int}, \mathbf{lev}\}$ indicates whether the variable contains integer values or security levels, and S is a set of expressions. Intuitively, if $\Gamma(x) = \tau_S$ and at some point in the program's execution, the value of variable x may reveal information about the initial input stream of the security level that expression e evaluates to, then $e \in S$.

Typing rules for the new type system are presented in Figure 7. Judgment $\Gamma, \Delta, pc \vdash_{\text{dep}} c$ means that command c is well-typed under security-type context Γ, channel context bounds Δ, and program counter levels pc. Program counter levels and the codomain of channel context bounds are now sets of expressions instead of sets of security levels. Intuitively, if information at the security level to which e evaluates might influence control flow reaching command c, then $e \in pc$, and if information at the security level to which e evaluates to might influence whether an input or output event occurs on channel ℓ, then $e \in \Delta(\ell)$.

$$\frac{}{\Gamma, \Delta, pc \vdash_{\text{dep}} \mathsf{skip}} \qquad \frac{\Gamma, \Delta, pc \vdash_{\text{dep}} c_1 \quad \Gamma, \Delta, pc \vdash_{\text{dep}} c_2}{\Gamma, \Delta, pc \vdash_{\text{dep}} c_1; c_2}$$

$$\frac{\Gamma(x) = \tau_S \quad \Gamma(e) = \tau_{S'} \quad pc \cup S' \subseteq S}{\Gamma, \Delta, pc \vdash_{\text{dep}} x := e} \qquad \frac{\Gamma(e) = \mathbf{int}_S \quad \Gamma, \Delta, pc \cup S \vdash_{\text{dep}} c_1 \quad \Gamma, \Delta, pc \cup S \vdash_{\text{dep}} c_2}{\Gamma, \Delta, pc \vdash_{\text{dep}} \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2}$$

$$\frac{\Gamma(e) = \mathbf{int}_S \qquad \Gamma, \Delta, pc \cup S \vdash_{\text{dep}} c}{\Gamma, \Delta, pc \vdash_{\text{dep}} \mathsf{while}\ e\ \mathsf{do}\ c}$$

$$\frac{\begin{array}{c}\Gamma(x) = \tau_S \qquad \Gamma(e) = \mathbf{lev}_{S'} \qquad pc \cup \{e\} \cup S' \subseteq S \\ \forall \ell \in \mathit{values}(e).\ pc \subseteq \Delta(\ell) \\ \forall \ell \in \mathit{values}(e).\ \forall e' \in \Delta(\ell) \cup S'. \\ \forall \ell' \in \mathit{all\text{-}values}(e').\ \mathit{may\text{-}flow}(\ell', \ell)\end{array}}{\Gamma, \Delta, pc \vdash_{\text{dep}} \mathsf{input}\, x\, \mathsf{from}\, e}$$

$$\frac{\begin{array}{c}\Gamma(e_1) = \tau_{S_1} \qquad \Gamma(e_2) = \mathbf{lev}_{S_2} \\ \forall \ell \in \mathit{values}(e_2).\ pc \subseteq \Delta(\ell) \\ \forall \ell \in \mathit{values}(e_2).\ \forall e' \in \Delta(\ell) \cup S_1 \cup S_2. \\ \forall \ell' \in \mathit{all\text{-}values}(e').\ \mathit{may\text{-}flow}(\ell', \ell)\end{array}}{\Gamma, \Delta, pc \vdash_{\text{dep}} \mathsf{output}\, e_1\, \mathsf{to}\, e_2}$$

Figure 7. Typing rules for dependent type system

The type system is very similar to that presented in Section V-A. The typing rules for if and while commands now check that the guard evaluates to an integer value. Otherwise, the only significant changes are to the rules for input and output.

Consider the rule for input command $\mathsf{input}\, x\, \mathsf{from}\, e$. The value that will be stored in x may reveal that the command was executed, and so the typing rule ensures that the program counter levels pc are a subset of S, the levels for x. In addition, the value to be stored in x will reveal information about the input stream of channel e, and so the rule requires $e \in S$. Also, the evaluation of e may reveal information, and so the rule requires $S' \subseteq S$, where S' is the set of level expressions for e. The channel context bounds must be an upper bound for the decision to perform any input or output, and so for all security levels $\ell \in \mathit{values}(e)$, we must have $pc \subseteq \Delta(\ell)$, where $\mathit{values}(e)$ is a conservative approximation of possible security levels that e could evaluate to at this program point. Finally, an observer of channel e observes that an input event occurred, potentially allowing the observer to infer both that the command executed, and the value of expression e. Thus, for all $\ell \in \mathit{values}(e)$, all $e' \in \Delta\ell \cup S'$ and all $\ell' \in \mathit{all\text{-}values}(e')$, flow must be allowed from ℓ' to ℓ, where $\mathit{all\text{-}values}(e')$ is a conservative approximation of possible security levels that e could evaluate to at any program point.

Note that while we consider possible evaluations of expression e at just the queried program point ($\mathit{values}(e)$), we must consider possible evaluations of expressions e' at any point in the program's execution ($\mathit{all\text{-}values}(e')$), since expressions e' are taken from type information, and the type system is flow insensitive.

Execution of an output command $\mathsf{output}\, e_1\, \mathsf{to}\, e_2$ reveals information to an observer of channel e_2, including that the command executed, the value of both expression e_1 and e_2. The typing rule for output commands requires that that $\mathit{may\text{-}flow}(\ell', \ell)$ is true for all $\ell \in \mathit{values}(e_2)$, all $e' \in \Delta\ell \cup S_1 \cup S_2$ and all $\ell' \in \mathit{all\text{-}values}(e')$.

The modified type system enforces security for the extended language.

Theorem 7. *For all commands c, security levels ℓ, and $i \in \mathbb{N}$, if there exists a security-type context Γ and a channel context bound Δ such that $\Gamma, \Delta, \emptyset \vdash_{\text{dep}} c$ then c is progress-insensitively secure against attacker $A_{i\text{-}only}$ with level ℓ.*

The proof of Theorem 7 is similar in structure to the proof of Theorem 5.

Program P_9 is well-typed under a security-type context Γ such that $\Gamma(x) = \{A\}$ and $\Gamma(y) = \{x, A\}$, and provided $\mathit{may\text{-}flow}(A, x)$ evaluates to true at command $\mathsf{input}\, y\, \mathsf{from}\, x$, and $\mathit{may\text{-}flow}(x, B)$ and $\mathit{may\text{-}flow}(A, B)$ both evaluate to true at the output command.