Eliminating Covert Flows with Minimum Typings[†]

Dennis Volpano
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943, USA
E-mail: volpano@cs.nps.navy.mil

Geoffrey Smith
School of Computer Science
Florida International University
Miami, FL 33199, USA
E-mail: smithg@cs.fiu.edu

Abstract

A type system is given that eliminates two kinds of covert flows in an imperative programming language. The first kind arises from nontermination and the other from partial operations that can raise exceptions. The key idea is to limit the source of nontermination in the language to constructs with minimum typings, and to evaluate partial operations within expressions of **try** *commands which also have minimum typings. A mutual progress theorem is proved that basically states that no two executions of a well-typed program can be distinguished on the basis of nontermination versus abnormal termination due to a partial operation. The proof uses a new style of programming language semantics which we call a natural transition semantics.*

1. Introduction

In [9], we gave a type system for secure information flow in a core imperative language. The type system is composed of a set of types and typing rules for deducing the types of expressions and commands. Types correspond to partially-ordered security classes like low (L) and high (H), where $L \leq H$. The ordering is the basis for a subtype relation which allows upward information flows. We proved a form of noninterference for the type system. However, the system does not address covert flows in programs that arise from nontermination and partial operations.

To illustrate these kinds of flows, we give part of the thread bodies of two Java applets that merely prompt a client for a password via a text field. The first applet creates an inspector thread for each character in the password. Part of the inspector thread body is given in Figure 1. It loops indefinitely when it discovers the character stored at position i. Until then, it records the characters it has examined by opening a socket connection back to another port on the server from which the applet originated. This connection is permitted under the current "sandbox" model of Java security. A similar inspector thread body can be designed to reveal a password using a partial operation. Part of such a body is given in Figure 2. It uses division and fails to catch the arithmetic exception. The thread bodies of the preceding examples are well typed in our original secure-flow type system.

```
while (p.charAt(i) == 'a')
      ;
ps.println(i + " not a");
while (p.charAt(i) == 'b')
      ;
ps.println(i + " not b");
```

Figure 1. Covert Flow from Nontermination

```
if (1/(p.charAt(i) - 'a') == 0)
      ;
ps.println(i + " not a");
if (1/(p.charAt(i) - 'b') == 0)
      ;
ps.println(i + " not b");
```

Figure 2. Covert Flow from a Partial Operation

We show how these kinds of covert flows can be handled with just a simple modification to our original type system based on the notion of a *minimum type*. We say that a type τ is minimum if $\tau \leq \tau'$ for every type τ'. To handle the covert flow arising from nontermination, we merely change

[†]This material is based upon activities supported by DARPA under contract BEA 96-1125 and by the National Science Foundation under grant CCR-9612176.

the typing rule for **while** e **do** c to require that e have minimum type. Similarly, we introduce a **try** command for each partial operation and type the command minimally. Now, the variable `c`, in the examples above, would not have minimum type, so the thread bodies would not be well typed since neither could be typed minimally.

The new typing rules allow us to prove theorems about covert flows. Our first covert-flow theorem establishes the property of *termination agreement* for well-typed programs. It is proved with respect to a natural, or "big-step", semantics. Termination agreement is a somewhat weaker statement about covert flow than we desire. This will lead us to a second theorem that establishes a stronger property for well-typed programs, namely *mutual progress*.

To prove mutual progress, we need a transitional, or "small-step", style of semantics in order to make statements about partial executions. We use a form of transition semantics for this purpose which we call a natural transition semantics (NTS) [7]. It is derivable directly from our natural semantics.

Soundness and completeness of the NTS, with respect to the natural semantics, allows us to switch from one semantic style to the other where appropriate. The proof of mutual progress, for instance, depends on termination agreement which can be proved more easily in the natural semantics than in the NTS since a natural semantics is well suited for reasoning about complete evaluation derivations. So we jump out of the progress proof, by NTS soundness, to get termination agreement, which is proved in the natural semantics, and then re-enter, by NTS completeness, to carry out the progress proof.

Finally, we consider a more restrictive type system that also requires conditionals to be typed minimally. Then we get an even stronger covert-flow result that basically rules out covert timing channels in programs. That is, no two executions of a well-typed program can be distinguished by timing differences.

2. The Type System

The core language we consider consists of phrases, each of which is either an expression or a command. We let metavariable p range over phrases, e over expressions, and c over commands:

$$
\begin{array}{lcl}
p & ::= & e \mid c \\
e & ::= & x \mid l \mid n \mid e + e' \mid e - e' \\
 & & \mid e = e' \mid e < e' \\
c & ::= & e := e' \mid c; c' \\
 & & \mid \textbf{if } e \textbf{ then } c \textbf{ else } c' \\
 & & \mid \textbf{while } e \textbf{ do } c \\
 & & \mid \textbf{letvar } x := e \textbf{ in } c \\
 & & \mid \textbf{try } x = e \div e' \textbf{ in } c
\end{array}
$$

Metavariable x ranges over identifiers, l over *locations*, and n over integer literals. Integers are the only values. We use 0 for false and 1 for true, and assume that locations are well ordered. All program I/O is done through free locations in a program. The core language includes a **try** command for one partial operation, namely, integer division. The scope of x in a **try** command is c. Other partial operations can be introduced in the same fashion. We want to consider only those programming constructs that are fundamental to a treatment of covert flows in an imperative language. For this reason, procedures and an assortment of other language features, such as arrays, are not included.

Notice that **try** commands do not have **catch** clauses for exception handling. A command like

$$\textbf{try } x = e \div e' \textbf{ in } c \textbf{ catch } c'$$

introduces an implicit flow from e and e' to c' that can be handled with a typing rule like those for any guarded commands. Here, we focus on the case where exceptions are not caught and therefore do not consider **try-catch** commands.

As in our earlier type system, the types of the core language are stratified:

$$
\begin{array}{lcl}
\tau & ::= & s \\
\rho & ::= & \tau \mid \tau \ \mathit{var} \mid \tau \ \mathit{cmd}
\end{array}
$$

Metavariable s ranges over security classes, which we assume are partially ordered by $\leq$. Type τ *var* is the type of a variable and τ *cmd* is the type of a command.

The typing rules for the core imperative language are given in Figure 3. They form a deductive proof system for assigning types to expressions and commands. They are given in a syntax-directed form and are equivalent to a more flexible system where coercions can be applied more freely. Typing rules for some expressions are omitted since they are similar to rule (ARITH).

Typing judgements have the form

$$\lambda; \gamma \vdash p : \rho$$

where λ is a *location typing* and γ is an *identifier typing*. The judgement means that phrase p has type ρ, assuming λ prescribes types for locations in p and γ prescribes types for any free identifiers in p. An identifier typing is a finite function mapping identifiers to ρ types; $\gamma(x)$ is the ρ type assigned to x by γ and $\gamma[x : \rho]$ assigns type ρ to x and to variable $x' \neq x$, type $\gamma(x')$. If γ is dropped from a judgement, as in $\lambda \vdash p : \rho$, then it is assumed to be empty. A location typing is also a finite function, but it maps locations to τ types. The notational conventions for location typings are similar.

One can understand the intuition behind our type system as follows: in a guarded command like **while** e **do** c, whenever c is executed, it is known that e was true. Hence, if

(INT) $\lambda; \gamma \vdash n : \tau$

(VAR) $\lambda; \gamma \vdash x : \tau \ \mathit{var} \quad \gamma(x) = \tau \ \mathit{var}$

(VARLOC) $\lambda; \gamma \vdash l : \tau \ \mathit{var} \quad \lambda(l) = \tau$

(ARITH)
$$\frac{\lambda; \gamma \vdash e : \tau, \quad \lambda; \gamma \vdash e' : \tau}{\lambda; \gamma \vdash e + e' : \tau}$$

(R-VAL)
$$\frac{\lambda; \gamma \vdash e : \tau \ \mathit{var}, \quad \tau \leq \tau'}{\lambda; \gamma \vdash e : \tau'}$$

(ASSIGN)
$$\frac{\lambda; \gamma \vdash e : \tau \ \mathit{var}, \quad \lambda; \gamma \vdash e' : \tau, \quad \tau' \leq \tau}{\lambda; \gamma \vdash e := e' : \tau' \ \mathit{cmd}}$$

(COMPOSE)
$$\frac{\lambda; \gamma \vdash c : \tau \ \mathit{cmd}, \quad \lambda; \gamma \vdash c' : \tau \ \mathit{cmd}}{\lambda; \gamma \vdash c; \ c' : \tau \ \mathit{cmd}}$$

(IF)
$$\frac{\lambda; \gamma \vdash e : \tau, \quad \lambda; \gamma \vdash c : \tau \ \mathit{cmd}, \quad \lambda; \gamma \vdash c' : \tau \ \mathit{cmd}, \quad \tau' \leq \tau}{\lambda; \gamma \vdash \textbf{if } e \textbf{ then } c \textbf{ else } c' : \tau' \ \mathit{cmd}}$$

(TRY)
$$\frac{\lambda; \gamma \vdash e : \tau, \quad \lambda; \gamma \vdash e' : \tau, \quad \lambda; \gamma[x : \tau] \vdash c : \tau \ \mathit{cmd}, \quad \tau \text{ is minimum}}{\lambda; \gamma \vdash \textbf{try } x = e \div e' \textbf{ in } c : \tau \ \mathit{cmd}}$$

(WHILE)
$$\frac{\lambda; \gamma \vdash e : \tau, \quad \lambda; \gamma \vdash c : \tau \ \mathit{cmd}, \quad \tau \text{ is minimum}}{\lambda; \gamma \vdash \textbf{while } e \textbf{ do } c : \tau \ \mathit{cmd}}$$

(LETVAR)
$$\frac{\lambda; \gamma \vdash e : \tau, \quad \lambda; \gamma[x : \tau \ \mathit{var}] \vdash c : \tau' \ \mathit{cmd}}{\lambda; \gamma \vdash \textbf{letvar } x := e \textbf{ in } c : \tau' \ \mathit{cmd}}$$

Figure 3. Typing Rules for Eliminating Covert Flows

$e : H$, then c must not assign to any variables of class L, for such assignments would constitute an illegal downward flow. The typing rule therefore requires that c in this case have type H *cmd*, which means that it only assigns to variables of class H. However, assigning to variables is not the only way for a command to transmit information—a command can also transmit information by failing to terminate or by aborting. Such failed executions transmit information (covertly) to an outside observer of the program's execution, who must be regarded as L. To prevent such downward covert flows, we require that the sources of failed executions (i.e. the guard of a **while** loop and the denominator of a division in a **try** command) have minimum type.[1] The new restrictions on **while** and **try** ensure that executing a command of type H *cmd* does not transmit covert information to an outside observer, because the command is guaranteed to terminate successfully.

Of course, this does not rule out *timing channels*, which use program execution time to transmit information to the outside observer. In our final covert-flow theorem in Section 5, we consider eliminating timing channels by also requiring the guard of conditional commands to have minimum type. But this may make the type system too restrictive to be practical. More experience is needed to be sure.

3. Our First Covert-Flow Theorem

Our first covert-flow theorem is expressed with respect to a natural semantics for closed phrases in the core language. A closed phrase is evaluated relative to a *memory* μ, which is a finite function from locations to values. The contents of a location $l \in \mathit{dom}(\mu)$ is the value $\mu(l)$, and we write $\mu[l := n]$ for the memory that assigns value n to location l, and value $\mu(l')$ to a location $l' \neq l$; $\mu[l := n]$ is an update of μ if $l \in \mathit{dom}(\mu)$ and an extension of μ otherwise.

The evaluation rules are given in Figure 4. They allow us to derive judgements of the form $\mu \vdash e \Rightarrow n$ for expressions and $\mu \vdash c \Rightarrow \mu'$ for commands. Evaluating a closed expression e in a memory μ results in an integer n. Expressions are pure in that they do not alter memory when evaluated. Evaluating a closed command c in a memory μ results in a new memory μ'. Commands do not yield values.

We write $[e/x]c$ to denote the substitution of e for all free occurrences of x in c, and let $\mu - l$ be memory μ with location l deleted from its domain. Note the use of substitution in rules (DIV) and (BINDVAR). It allows us to avoid using environments in the semantics.

3.1. Termination Agreement

Now we can state our first covert-flow theorem:

[1] For simplicity, we also require the numerator of a division to have minimum type. This restriction can be relaxed.

Theorem 3.1 (Termination Agreement) *Suppose*

(a) $\lambda \vdash c : \rho$,

(b) $\mu \vdash c \Rightarrow \mu'$,

(c) ν *is a memory such that* $\mathit{dom}(\mu) = \mathit{dom}(\nu) = \mathit{dom}(\lambda)$, *and*

(d) $\nu(l) = \mu(l)$ *for all* l *such that* $\lambda(l) \leq \tau$.

Then there is a memory ν' *such that* $\nu \vdash c \Rightarrow \nu'$ *and* $\nu'(l) = \mu'(l)$ *for all* l *such that* $\lambda(l) \leq \tau$.

An alternative statement of the theorem is if a command c is well typed, and μ and ν are memories such that (c) and (d) are true, then either

1. c fails to terminate successfully under μ and ν, or
2. c terminates successfully under μ and ν and the resulting memories agree on all locations whose types are bounded by τ.

The theorem departs from the noninterference theorem of [9] in that it does not require c to terminate successfully under both μ and ν. There is a hypothesis about the successful termination of c under μ only. With the remaining hypotheses, it is enough to ensure that c also terminates successfully under ν.

Before proving the theorem, we need a number of lemmas. The first four lemmas are taken from our earlier work [9]. They can be proved for the typing rules in Figure 3 as well.

Lemma 3.2 (Simple Security) *If* $\lambda \vdash e : \tau$, *then for every* l *in* e, $\lambda(l) \leq \tau$.

Lemma 3.3 (Confinement) *If* $\lambda; \gamma \vdash c : \tau$ *cmd, then for every* l *assigned to in* c, $\lambda(l) \geq \tau$.

Lemma 3.4 (Expression Substitution) *If* $\lambda; \gamma[x : \tau] \vdash p : \rho$, *then* $\lambda; \gamma \vdash [n/x]p : \rho$, *and if* $\lambda; \gamma \vdash l : \rho$ *and* $\lambda; \gamma[x : \rho] \vdash p : \rho'$, *then* $\lambda; \gamma \vdash [l/x]p : \rho'$.

Lemma 3.5 *If* $\mu \vdash c \Rightarrow \mu'$, *then* $\mathit{dom}(\mu) = \mathit{dom}(\mu')$.

We introduce the following lemmas, each of which can be proved by induction on phrase structure.

Lemma 3.6 (Determinism) *Suppose* $\nu(l) = \mu(l)$, *for every* l *in* e, $\mu \vdash e \Rightarrow n$, *and* $\nu \vdash e \Rightarrow n'$. *Then* $n = n'$.

Lemma 3.7 *Suppose* $\lambda \vdash e : \tau$ *and* μ *is a memory such that* $\mathit{dom}(\mu) = \mathit{dom}(\lambda)$. *Then there is an integer* n *such that* $\mu \vdash e \Rightarrow n$.

(VAL) $\mu \vdash n \Rightarrow n$

(CONTENTS) $\mu \vdash l \Rightarrow \mu(l) \qquad l \in \mathit{dom}(\mu)$

(ADD)
$$\frac{\mu \vdash e \Rightarrow n, \;\; \mu \vdash e' \Rightarrow n'}{\mu \vdash e + e' \Rightarrow n + n'}$$

(UPDATE)
$$\frac{\mu \vdash e \Rightarrow n}{\mu \vdash l := e \Rightarrow \mu[l := n]} \;\; l \in \mathit{dom}(\mu)$$

(SEQUENCE)
$$\frac{\mu \vdash c \Rightarrow \mu', \;\; \mu' \vdash c' \Rightarrow \mu''}{\mu \vdash c;\, c' \Rightarrow \mu''}$$

(BRANCH)
$$\frac{\mu \vdash e \Rightarrow n, \;\; (n \text{ nonzero}) \qquad \mu \vdash c \Rightarrow \mu'}{\mu \vdash \textbf{if } e \textbf{ then } c \textbf{ else } c' \Rightarrow \mu'}$$

$$\frac{\mu \vdash e \Rightarrow 0, \qquad \mu \vdash c' \Rightarrow \mu'}{\mu \vdash \textbf{if } e \textbf{ then } c \textbf{ else } c' \Rightarrow \mu'}$$

(DIV)
$$\frac{\mu \vdash e \Rightarrow n, \qquad \mu \vdash e' \Rightarrow n', \;\; (n' \text{ nonzero}) \qquad \mu \vdash [(n \div n')/x]c \Rightarrow \mu'}{\mu \vdash \textbf{try } x = e \div e' \textbf{ in } c \Rightarrow \mu'}$$

(LOOP)
$$\frac{\mu \vdash e \Rightarrow 0}{\mu \vdash \textbf{while } e \textbf{ do } c \Rightarrow \mu}$$

$$\frac{\mu \vdash e \Rightarrow n, \;\; (n \text{ nonzero}) \qquad \mu \vdash c \Rightarrow \mu', \qquad \mu' \vdash \textbf{while } e \textbf{ do } c \Rightarrow \mu''}{\mu \vdash \textbf{while } e \textbf{ do } c \Rightarrow \mu''}$$

(BINDVAR)
$$\frac{\mu \vdash e \Rightarrow n, \qquad l \text{ is the least location not in } \mathit{dom}(\mu), \qquad \mu[l := n] \vdash [l/x]c \Rightarrow \mu'}{\mu \vdash \textbf{letvar } x := e \textbf{ in } c \Rightarrow \mu' - l}$$

Figure 4. Core Language Natural Semantics

Lemma 3.8 *If $\lambda;\gamma \vdash c : \tau$ cmd and c contains an occurrence of* **while** *or* **try**, *then τ is minimum.*

Lemma 3.9 *Suppose $\lambda \vdash c : \tau$ cmd and c does not contain an instance of* **while** *or* **try**, *and μ is a memory such that dom(μ) = dom(λ). Then there is a memory μ' such that $\mu \vdash c \Rightarrow \mu'$.*

Notice the purely syntactic hypotheses under which termination is guaranteed in Lemma 3.9. Limiting *partial* recursion to typed commands in a language (*e.g.* **while** or **letrec**) makes it easier to get a sound and practical type system to control covert flows. Some programming language features make it much harder to achieve such a system. For example, some people have proposed extending Java with higher-order functions. In the context of an imperative language, such as Java, higher-order functions make recursion possible through circularity in memory: one can bind a variable to a function containing a free occurrence of that variable [8]. Such an extension makes it harder for the type system to be aware of potentially nonterminating programs, and yet be flexible.

Typing the **while** and **try** commands minimally prevents them from taking different execution paths under two memories that agree on locations with minimum type. A conditional, however, is still free to take different execution paths under two such memories.

The proof of the termination agreement theorem resembles the proof of noninterference in [9]. It proceeds by induction on the structure of $\mu \vdash c \Rightarrow \mu'$. We give the proof for one of the more interesting cases, namely, evaluation rule (BRANCH). The remaining evaluation rules are treated similarly.

(BRANCH). Suppose $\mu \vdash$ **if** e **then** c **else** $c' \Rightarrow \mu'$ and the typing derivation ends with an application of rule (IF):

$$\frac{\begin{array}{l}\lambda \vdash e : \tau',\\ \lambda \vdash c : \tau' \textit{ cmd},\\ \lambda \vdash c' : \tau' \textit{ cmd},\\ \tau'' \leq \tau'\end{array}}{\lambda \vdash \textbf{if } e \textbf{ then } c \textbf{ else } c' : \tau'' \textit{ cmd}}$$

There are two cases:

1. $\tau' \leq \tau$. Then suppose the evaluation under μ ends with the second rule for (BRANCH):

$$\frac{\begin{array}{l}\mu \vdash e \Rightarrow 0\\ \mu \vdash c' \Rightarrow \mu'\end{array}}{\mu \vdash \textbf{if } e \textbf{ then } c \textbf{ else } c' \Rightarrow \mu'}$$

   By the simple security lemma, $\lambda(l) \leq \tau'$ for every l in e and so $\lambda(l) \leq \tau$ for every l in e. By hypothesis (d) then, $\nu(l) = \mu(l)$ for every l in e, and thus $\nu \vdash e \Rightarrow 0$ by Lemmas 3.6 and 3.7. By induction there is a memory ν' such that $\nu \vdash c' \Rightarrow \nu'$ and $\nu'(l) = \mu'(l)$ for all l such that $\lambda(l) \leq \tau$. Then $\nu \vdash$ **if** e **then** c **else** $c' \Rightarrow \nu'$ by the second rule for (BRANCH). Evaluation under μ ending with the first rule for (BRANCH) is handled similarly.

2. $\tau' \not\leq \tau$. Then τ' is not minimum, and thus by Lemma 3.8, neither c nor c' contains an occurrence of **while** or **try**. So there is a memory ν' such that $\nu \vdash$ **if** e **then** c **else** $c' \Rightarrow \nu'$ by Lemma 3.9. By the Confinement Lemma, $\lambda(l) \geq \tau'$ for every l assigned to in c or c'. Thus for every l assigned to in c or c', $\lambda(l) \not\leq \tau$ since otherwise $\tau' \leq \tau$. So if $l \in dom(\lambda)$ and $\lambda(l) \leq \tau$, then l is not assigned to in c or c'. So $\mu'(l) = \mu(l)$ and $\nu'(l) = \nu(l)$ for all l such that $\lambda(l) \leq \tau$, and we're done by (d).

4. Our Second Covert-Flow Theorem

Termination agreement is still a somewhat weaker statement than we want about what the type system actually guarantees in terms of protection against covert flows. It says that if c does not terminate successfully under one memory then it doesn't terminate successfully under the other memory either. So the two executions cannot be distinguished by one of them terminating successfully and the other failing to do so. But what about distinguishing nontermination from abnormal termination? The theorem does not rule out the possibility that c fails to terminate under one memory and gets stuck (aborts) under the other.

For example, suppose location l ranges over 0 and 1 and that $l \in dom(\mu)$ and $l \in dom(\nu)$. Now if μ and ν agree on all locations of minimum type, then

$$\begin{array}{l}\textbf{try } z = 2 \div l \textbf{ in}\\ \quad \textbf{while } (l > 0) \textbf{ do } ;\end{array}$$

may get stuck under μ yet fail to terminate under ν if l does not have minimum type. These two executions can be distinguished. What we want to show yet is that if a command c is well typed and it fails to terminate successfully in some way under μ, then it also fails to terminate successfully in the same way under ν. Stated in another way, execution of c under ν makes progress iff its execution under μ does. This brings us to our second covert-flow theorem: the *mutual progress* theorem. However, before we can state and prove the theorem, we need another form of semantics.

A natural semantics allows us to state properties about successful or complete program executions, not partial ones. So it is not suited for proving properties about intermediate steps of a computation like progress theorems. For this, we use a new form of semantics which we call a *natural transition semantics* (NTS) because it is derived directly from the natural semantics [7]. Unlike the treatment

of NTS in [7], here it is formulated as a set of transition rules. These rules admit proofs of properties about a single transition by induction on the structure of its derivation.

4.1. Natural Transition Semantics

A traditional transition semantics for an imperative programming language defines transitions between configurations that involve memories and terms of the language [3]. Here we define transitions between *partial derivation trees* which represent partial derivations in the natural semantics.

Partial derivation trees are defined as follows. First, we add to the complete judgments $\mu \vdash e \Rightarrow n$ and $\mu \vdash c \Rightarrow \mu'$, a new kind of judgment called a *pending judgment* which has the form

$$\mu \vdash p \Rightarrow ?$$

where p is a phrase. Then partial derivation trees are defined inductively:

1. $[\mu \vdash e \Rightarrow n]$, $[\mu \vdash c \Rightarrow \mu']$ and $[\mu \vdash p \Rightarrow ?]$ are partial derivation trees.

2. if P is a predicate, then $[P]$ is a partial derivation tree.

3. if $T_1, \ldots, T_n$ are partial derivation trees, then $[\mu \vdash e \Rightarrow n](T_1, \ldots, T_n)$, $[\mu \vdash c \Rightarrow \mu'](T_1, \ldots, T_n)$, and $[\mu \vdash p \Rightarrow ?](T_1, \ldots, T_n)$ are partial derivation trees.

For example, $[\mu \vdash l \Rightarrow \mu(l)]([l \in \mathit{dom}(\mu)])$ is a partial derivation tree. We say that a partial derivation is *complete* if it has no subtree rooted at $[\mu \vdash p \Rightarrow ?]$. Every complete derivation tree is a partial derivation tree. We let I, J, and K range over complete derivation trees and T over partial derivation trees.

Rules of the natural transition semantics for expressions and the **while** and **try** commands, are given in Figures 5, 6, and 7. We use m, j, and k in the rules as indices that start at zero. Transition rules have been omitted for the other commands since their formulation from the natural semantics is similar. Let $\longrightarrow^*$ be the reflexive and transitive closure of $\longrightarrow$, that is, $T \xrightarrow{0} T$, for any T, $T \xrightarrow{k+1} T'$ if there exists T'' such that $T \xrightarrow{k} T''$ and $T'' \longrightarrow T'$, and $T \longrightarrow^* T'$ if $T \xrightarrow{k} T'$ for some $k \geq 0$.

The transition rules also include a rule (CONGRUENCE) which allows execution of compound phrases:

$$\frac{T \longrightarrow T'}{\begin{array}{c}[\mu \vdash p \Rightarrow ?](J_1, \ldots, J_n, T) \longrightarrow \\ [\mu \vdash p \Rightarrow ?](J_1, \ldots, J_n, T')\end{array}}$$

It allows the semantics to "scale up":

Lemma 4.1 *Suppose that T and T' are partial derivation trees, $n \geq 0$, and $k \geq 0$. Then $T \xrightarrow{k} T'$ iff*

$$\begin{array}{c}[\mu \vdash p \Rightarrow ?](J_1, \ldots, J_n, T) \xrightarrow{k} \\ [\mu \vdash p \Rightarrow ?](J_1, \ldots, J_n, T')\end{array}$$

Proof. Both directions can be proved by induction on k, using rule (CONGRUENCE). The (if) direction requires observing that if $T \longrightarrow^* T'$ then the number of children of the root of T' is at least that of the root of T, and if

$$T \longrightarrow^* [\mu \vdash p \Rightarrow ?](T_1, \ldots, T_m)$$

for $m \geq 0$, then T is rooted at $[\mu \vdash p \Rightarrow ?]$. □

It should be noted that controlling the lifetime of locations in a traditional transition semantics is tricky since one is limited to transitions between configurations involving language terms. But with transitions between partial derivation trees, we can exploit different tree structure and avoid introducing extra information into configurations like the number of "live" locations [6]. The transition rule that allocates a location for an instance of **letvar** is a transition from a tree whose root has exactly one child to one whose root has exactly three children. This different tree structure can be exploited in the rules to specify in a more natural way when locations should be deallocated.

We say that a partial derivation tree T is *sound* if for every node in T of the form $[\nu \vdash c \Rightarrow \nu']$, we have $\nu \vdash c \Rightarrow \nu'$, for every node of the form $[\nu \vdash e \Rightarrow n]$, we have $\nu \vdash e \Rightarrow n$, and for every node of the form $[P]$, P is true.

Lemma 4.2 *If T and T' are partial derivation trees such that T is sound and $T \longrightarrow T'$, then T' is sound.*

Proof. Induction on the structure of the derivation of $T \longrightarrow T'$. □

By an easy induction on the number of transitions, we have that if $T \longrightarrow^* T'$ and T is sound, then so is T'. This leads to the following corollary:

Proposition 4.3 (NTS Soundness) *If $[\mu \vdash e \Rightarrow ?] \longrightarrow^* [\mu \vdash e \Rightarrow n](J_1, \ldots, J_m)$ then $\mu \vdash e \Rightarrow n$. Further, if we have $[\mu \vdash c \Rightarrow ?] \longrightarrow^* [\mu \vdash c \Rightarrow \mu'](J_1, \ldots, J_m)$, then $\mu \vdash c \Rightarrow \mu'$.*

Completeness of the transition semantics is given by

Proposition 4.4 (NTS Completeness) *Suppose that $\mu \vdash e \Rightarrow n$ and that the judgment has a complete derivation tree J. Then $[\mu \vdash e \Rightarrow ?] \longrightarrow^* J$. Further, if $\mu \vdash c \Rightarrow \mu'$ and this judgment has a complete derivation tree J, then $[\mu \vdash c \Rightarrow ?] \longrightarrow^* J$.*

Proof. Induction on the structure of the derivation of $\mu \vdash e \Rightarrow n$ and of $\mu \vdash c \Rightarrow \mu'$, using Lemma 4.1. □

(T-VAL) $[\mu \vdash n \Rightarrow ?] \longrightarrow [\mu \vdash n \Rightarrow n]$

(T-CONTENTS) $$\frac{l \in \mathit{dom}(\mu)}{[\mu \vdash l \Rightarrow ?] \longrightarrow [\mu \vdash l \Rightarrow \mu(l)]([l \in \mathit{dom}(\mu)])}$$

(T-ADD) $[\mu \vdash e + e' \Rightarrow ?] \longrightarrow [\mu \vdash e + e' \Rightarrow ?]([\mu \vdash e \Rightarrow ?])$

(1)
$$\begin{array}{l}[\mu \vdash e + e' \Rightarrow ?]([\mu \vdash e \Rightarrow n](J_1, \ldots, J_k)) \longrightarrow \\ \quad [\mu \vdash e + e' \Rightarrow ?](\\ \qquad [\mu \vdash e \Rightarrow n](J_1, \ldots, J_k), \\ \qquad [\mu \vdash e' \Rightarrow ?] \\ \qquad)\end{array}$$

(2)
$$\begin{array}{l}[\mu \vdash e + e' \Rightarrow ?]([\mu \vdash e \Rightarrow n](J_1, \ldots, J_k), \\ \qquad\qquad [\mu \vdash e' \Rightarrow n'](K_1, \ldots, K_m) \\ \qquad\qquad) \longrightarrow \\ \quad [\mu \vdash e + e' \Rightarrow n + n'](\\ \qquad [\mu \vdash e \Rightarrow n](J_1, \ldots, J_k), \\ \qquad [\mu \vdash e' \Rightarrow n'](K_1, \ldots, K_m) \\ \qquad)\end{array}$$

Figure 5. Natural Transition Semantics for Expressions

4.2. Mutual Progress

Next we establish the mutual progress property for the type system.

Theorem 4.5 (Mutual Progress) *Suppose*

(a) $\lambda \vdash c : \rho$*,*

(b) ν *and* μ *are memories such that* $\mathit{dom}(\mu) = \mathit{dom}(\nu) = \mathit{dom}(\lambda)$*,*

(c) $\nu(l) = \mu(l)$ *for all* l *such that* $\lambda(l) \leq \tau$*,*

(d) $[\mu \vdash c \Rightarrow ?] \longrightarrow^* T$*, and*

(e) T *has a leaf of the form* $[\mu' \vdash c' \Rightarrow ?]$ *where* c' *is a* **try** *command.*

Then there is a location typing λ' *and partial derivation tree* T' *such that* T' *has a leaf of the form* $[\nu' \vdash c' \Rightarrow ?]$*,* $[\nu \vdash c \Rightarrow ?] \longrightarrow^* T'$*,* $\lambda \subseteq \lambda'$*,* $\mathit{dom}(\mu') = \mathit{dom}(\nu') = \mathit{dom}(\lambda')$*, and* $\mu'(l) = \nu'(l)$*, for all* l *such that* $\lambda'(l) \leq \tau$*.*

Proof. Induction on the number of transitions in $[\mu \vdash c \Rightarrow ?] \longrightarrow^* T$. Aside from the basis, we show only one case, namely (T-LOOP). It is a good representative case because it illustrates the key steps one needs in order to prove the theorem for all other rules of the transition semantics.

For zero transitions, we have

$$[\mu \vdash c \Rightarrow ?] \longrightarrow^* [\mu \vdash c \Rightarrow ?]$$

where c is a **try** command. Let $\lambda' = \lambda$ and we're done by hypotheses (b) and (c).

Now suppose c is **while** e **do** c''. There are two subcases to consider here. They correspond to whether the leaf of hypothesis (e) arises before or after c has made a transition according to rule (T-LOOP)(3). First we consider the case when it arises before. Since commands are not expressions, we have, by hypotheses (d) and (e), that c'' contains a **try** command c',

$$\begin{array}{l}[\mu \vdash c \Rightarrow ?] \longrightarrow [\mu \vdash c \Rightarrow ?]([\mu \vdash e \Rightarrow ?]) \longrightarrow^* \\ \qquad [\mu \vdash c \Rightarrow ?](J_1)\end{array}$$

where J_1 is a complete derivation tree rooted at $[\mu \vdash e \Rightarrow n]$ and n is nonzero, and finally that

$$\begin{array}{l}[\mu \vdash c \Rightarrow ?](J_1) \longrightarrow \\ \quad [\mu \vdash c \Rightarrow ?](J_1, [n \text{ nonzero}], [\mu \vdash c'' \Rightarrow ?]) \longrightarrow^* \\ \qquad [\mu \vdash c \Rightarrow ?](J_1, [n \text{ nonzero}], T)\end{array}$$

where T contains a leaf of the form $[\mu' \vdash c' \Rightarrow ?]$.

By rule (T-LOOP), we have

$$[\nu \vdash c \Rightarrow ?] \longrightarrow [\nu \vdash c \Rightarrow ?]([\nu \vdash e \Rightarrow ?])$$

(T-DIV) $[\mu \vdash \textbf{try}\ x = e \div e'\ \textbf{in}\ c \Rightarrow ?] \longrightarrow$
$[\mu \vdash \textbf{try}\ x = e \div e'\ \textbf{in}\ c \Rightarrow ?]([\mu \vdash e \Rightarrow ?])$

(1) $[\mu \vdash \textbf{try}\ x = e \div e'\ \textbf{in}\ c \Rightarrow ?]([\mu \vdash e \Rightarrow n](K_1, \ldots, K_m)) \longrightarrow$
$[\mu \vdash \textbf{try}\ x = e \div e'\ \textbf{in}\ c \Rightarrow ?]($
$[\mu \vdash e \Rightarrow n](K_1, \ldots, K_m),$
$[\mu \vdash e' \Rightarrow ?]$
$)$

(2) $\dfrac{n' \text{ nonzero}}{[\mu \vdash \textbf{try}\ x = e \div e'\ \textbf{in}\ c \Rightarrow ?]([\mu \vdash e \Rightarrow n](K_1, \ldots, K_m),}$
$[\mu \vdash e' \Rightarrow n'](J_1, \ldots, J_k)$
$) \longrightarrow$
$[\mu \vdash \textbf{try}\ x = e \div e'\ \textbf{in}\ c \Rightarrow ?]($
$[\mu \vdash e \Rightarrow n](K_1, \ldots, K_m),$
$[\mu \vdash e' \Rightarrow n'](J_1, \ldots, J_k)$
$[n' \text{ nonzero}],$
$[\mu \vdash [(n \div n')/x]c \Rightarrow ?]$
$)$

(3) $[\mu \vdash \textbf{try}\ x = e \div e'\ \textbf{in}\ c \Rightarrow ?]($
$[\mu \vdash e \Rightarrow n](K_1, \ldots, K_m),$
$[\mu \vdash e' \Rightarrow n'](J_1, \ldots, J_k),$
$[n' \text{ nonzero}],$
$[\mu \vdash [(n \div n')/x]c \Rightarrow \mu'](I_1, \ldots, I_j)$
$) \longrightarrow$
$[\mu \vdash \textbf{try}\ x = e \div e'\ \textbf{in}\ c \Rightarrow \mu']($
$[\mu \vdash e \Rightarrow n](K_1, \ldots, K_m),$
$[\mu \vdash e' \Rightarrow n'](J_1, \ldots, J_k),$
$[n' \text{ nonzero}],$
$[\mu \vdash [(n \div n')/x]c \Rightarrow \mu'](I_1, \ldots, I_j)$
$)$

Figure 6. Natural Transition Semantics for try

(T-LOOP) $[\mu \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow ?] \longrightarrow$
$[\mu \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow ?]([\mu \vdash e \Rightarrow ?])$

(1) $[\mu \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow ?]([\mu \vdash e \Rightarrow 0](J_1, \ldots, J_k)) \longrightarrow$
$[\mu \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow \mu]([\mu \vdash e \Rightarrow 0](J_1, \ldots, J_k))$

(2) $\dfrac{n \text{ nonzero}}{[\mu \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow ?]([\mu \vdash e \Rightarrow n](J_1, \ldots, J_k)) \longrightarrow}$
$[\mu \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow ?]($
$[\mu \vdash e \Rightarrow n](J_1, \ldots, J_k),$
$[n \text{ nonzero}],$
$[\mu \vdash c \Rightarrow ?]$
$)$

(3) $[\mu \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow ?]($
$[\mu \vdash e \Rightarrow n](J_1, \ldots, J_k),$
$[n \text{ nonzero}],$
$[\mu \vdash c \Rightarrow \mu'](K_1, \ldots, K_m)$
$) \longrightarrow$
$[\mu \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow ?]($
$[\mu \vdash e \Rightarrow n](J_1, \ldots, J_k),$
$[n \text{ nonzero}],$
$[\mu \vdash c \Rightarrow \mu'](K_1, \ldots, K_m),$
$[\mu' \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow ?]$
$)$

(4) $[\mu \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow ?]($
$[\mu \vdash e \Rightarrow n](J_1, \ldots, J_k),$
$[n \text{ nonzero}],$
$[\mu \vdash c \Rightarrow \mu'](K_1, \ldots, K_j),$
$[\mu' \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow \mu''](I_1, \ldots, I_m)$
$) \longrightarrow$
$[\mu \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow \mu''](}$
$[\mu \vdash e \Rightarrow n](J_1, \ldots, J_k),$
$[n \text{ nonzero}],$
$[\mu \vdash c \Rightarrow \mu'](K_1, \ldots, K_j),$
$[\mu' \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow \mu''](I_1, \ldots, I_m)$
$)$

Figure 7. Natural Transition Semantics for while

Now we have $\lambda \vdash e : \tau'$ and $dom(\nu) = dom(\lambda)$, so there is an integer n' such that $\nu \vdash e \Rightarrow n'$, by Lemma 3.7. Suppose this judgment has a complete derivation tree J_1' rooted at $[\nu \vdash e \Rightarrow n']$. By completeness of the transition semantics,

$$[\nu \vdash e \Rightarrow ?] \longrightarrow^* J_1'$$

At this point, we need to show that execution of c does not proceed with a transition by rule (T-LOOP)(1) since this rule cannot lead to a derivation tree with the desired leaf.

We have $\lambda \vdash e : \tau'$ and τ' is minimum by the typing rule (WHILE). So $\lambda(l) \leq \tau'$ for every l in e by the simple security lemma. Also, $\tau' \leq \tau$ since τ' is minimum. So $\lambda(l) \leq \tau$ for every l in e, and thus $\nu(l) = \mu(l)$ for every l in e, by hypothesis (c). Further, by soundness of the transition semantics, $\mu \vdash e \Rightarrow n$. Thus, $n' = n$ by Lemma 3.6. So n' is nonzero and we then have by rule (T-LOOP)(2) and Lemma 4.1, that

$$[\nu \vdash c \Rightarrow ?]([\nu \vdash e \Rightarrow ?]) \longrightarrow^* [\nu \vdash c \Rightarrow ?](J_1', [n' \text{ nonzero}], [\nu \vdash c'' \Rightarrow ?])$$

Now by Lemma 4.1,

$$[\mu \vdash c'' \Rightarrow ?] \longrightarrow^* T$$

and so by induction,

$$[\nu \vdash c'' \Rightarrow ?] \longrightarrow^* T'$$

T' has a leaf of the form $[\nu' \vdash c' \Rightarrow ?]$ and there is a location typing λ' such that $\lambda \subseteq \lambda'$, $dom(\mu') = dom(\nu') = dom(\lambda')$, and $\nu'(l) = \mu'(l)$ for all l such that $\lambda'(l) \leq \tau$. Finally, by Lemma 4.1 again,

$$[\nu \vdash c \Rightarrow ?](J_1', [n' \text{ nonzero}], [\nu \vdash c'' \Rightarrow ?]) \longrightarrow^* [\nu \vdash c \Rightarrow ?](J_1', [n' \text{ nonzero}], T')$$

Now consider the case when the leaf arises after the **while** command has made a transition according to rule (T-LOOP)(3). Suppose that

$$[\mu \vdash c \Rightarrow ?] \longrightarrow^* [\mu \vdash c \Rightarrow ?](J_1, [n \text{ nonzero}], J_2, [\mu' \vdash c \Rightarrow ?]) \longrightarrow^* [\mu \vdash c \Rightarrow ?](J_1, [n \text{ nonzero}], J_2, T)$$

where J_1 is a complete derivation tree rooted at $[\mu \vdash e \Rightarrow n]$, such that n is nonzero, J_2 is a complete derivation tree rooted at $[\mu \vdash c'' \Rightarrow \mu']$, and T has a leaf of the form $[\mu'' \vdash c' \Rightarrow ?]$.

By Lemma 3.7, $\nu \vdash e \Rightarrow n'$. Suppose this judgment has a complete derivation tree J_1' rooted at $[\nu \vdash e \Rightarrow n']$. We also have $\lambda \vdash e : \tau'$ where τ' is minimum by typing rule (WHILE). So by the simple security lemma and hypothesis (c), $\nu(l) = \mu(l)$ for every l in e. Thus, $n' = n$, by Lemma 3.6, and so n' is nonzero.

By the soundness of the transition semantics, we have $\mu \vdash c'' \Rightarrow \mu'$. So by the termination agreement theorem, there is a ν' such that $\nu \vdash c'' \Rightarrow \nu'$ and $\nu'(l) = \mu'(l)$ for all l such that $\lambda(l) \leq \tau$. Suppose this judgment has complete derivation tree J_2' rooted at $[\nu \vdash c'' \Rightarrow \nu']$. By the completeness of the transition semantics, Lemma 4.1, and rules (T-LOOP), (T-LOOP)(2) and (T-LOOP)(3), we have

$$[\nu \vdash c \Rightarrow ?] \longrightarrow^* [\nu \vdash c \Rightarrow ?](J_1', [n' \text{ nonzero}], J_2', [\nu' \vdash c \Rightarrow ?])$$

Now by Lemma 4.1,

$$[\mu' \vdash c \Rightarrow ?] \longrightarrow^* T$$

By Lemma 3.5, and since $dom(\mu) = dom(\nu) = dom(\lambda)$, we have $dom(\mu') = dom(\nu') = dom(\lambda)$. Thus, by induction,

$$[\nu' \vdash c \Rightarrow ?] \longrightarrow^* T'$$

T' has a leaf of the form $[\nu'' \vdash c' \Rightarrow ?]$ and there is a location typing λ' such that $\lambda \subseteq \lambda'$, $dom(\mu'') = dom(\nu'') = dom(\lambda')$ and $\nu''(l) = \mu''(l)$ for all l such that $\lambda'(l) \leq \tau$.

Finally, by Lemma 4.1,

$$[\nu \vdash c \Rightarrow ?](J_1', [n' \text{ nonzero}], J_2', [\nu' \vdash c \Rightarrow ?]) \longrightarrow^* [\nu \vdash c \Rightarrow ?](J_1', [n' \text{ nonzero}], J_2', T')$$

and we're done. □

Notice in the proof that we have the guard of a **while** command evaluating to the same value under μ and ν since the command is typed minimally by rule (WHILE). The proof also needs the guard of a conditional to evaluate to the same value under μ and ν, yet rule (IF) does not require a conditional to be minimally typed. Nevertheless, it will be minimally typed due to hypotheses (a) and (e) of the theorem, and Lemma 3.8.

The mutual progress theorem tells us that if execution of a command c in a memory μ depends on executing

$$\textbf{try } x = e \div e' \textbf{ in } c'$$

in some memory μ', then c's execution in ν also depends on executing the **try** command in some memory ν'. Furthermore, we have that $\lambda; \gamma \vdash e' : \tau$, for minimum type τ, since c is well typed. The theorem gives us a typing λ' that contains λ, and hence $\lambda'; \gamma \vdash e' : \tau$. The theorem also tells us that ν' and μ' agree on all locations in the domain of λ' with minimum type. Thus either both executions proceed (e' evaluates to the same nonzero integer in μ' and ν') or both get stuck (e' evaluates to zero in μ' and ν').

5. Our Third Covert-Flow Theorem

Looking at the mutual progress theorem more closely, if execution of a command c gets stuck under a memory μ,

then its execution also gets stuck under any other memory ν that agrees with μ on locations of minimum type. This says that executions of c under memories that differ only on locations of nonminimum type cannot be distinguished on the basis of abnormal termination versus nontermination. But the number of steps c takes under μ and ν may differ.

Consider a well-typed composition $c; c'$ where c contains a conditional, with a nonminimum guard, and only c' contains a **try** command. Then although c' may get stuck under μ and ν, more steps may be needed to do so under one memory than under the other due to different execution paths taken by the conditional in c. (Remember that conditionals with nonminimum guards can still be typed minimally by subtyping.) As long as conditionals are not typed minimally, we cannot say that if execution of a well-typed command c gets stuck after k steps under μ, then it does so after k steps under ν as well.

As our final covert-flow theorem, we prove a timing agreement theorem for a more restricted type system. The restricted system is the original type system with rule (IF) changed so that τ is required to be minimum. Assume, hereafter, that $\vdash$ now refers to the more restricted system.

First we need two lemmas:

Lemma 5.1 *If μ and ν are memories, dom(μ) = dom(ν) and $[\mu \vdash e \Rightarrow ?] \xrightarrow{k} [\mu \vdash e \Rightarrow n](J_1, \ldots, J_n)$, then*

$$[\nu \vdash e \Rightarrow ?] \xrightarrow{k} [\nu \vdash e \Rightarrow n'](J'_1, \ldots, J'_n)$$

Proof. Straightforward induction on k, using Lemma 4.1. □

The next lemma is a stronger form of termination agreement (Theorem 3.1) for the more restricted type system. It does not hold if conditionals are not minimally typed.

Lemma 5.2 *Suppose $\lambda \vdash c : \rho$, μ and ν are memories such that dom(μ) = dom(ν) = dom(λ), $\mu(l) = \nu(l)$ for all l such that $\lambda(l) \leq \tau$, and*

$$[\mu \vdash c \Rightarrow ?] \xrightarrow{k} [\mu \vdash c \Rightarrow \mu'](J_1, \ldots, J_n)$$

Then we have $[\nu \vdash c \Rightarrow ?] \xrightarrow{k} [\nu \vdash c \Rightarrow \nu'](J'_1, \ldots, J'_n)$ and $\nu'(l) = \mu'(l)$ for all l such that $\lambda(l) \leq \tau$.

Proof. Induction on k, using Lemmas 4.1 and 5.1. □

Theorem 5.3 (Timing Agreement) *Suppose*

(a) $\lambda \vdash p : \rho$,

(b) *ν and μ are memories such that dom(μ) = dom(ν) = dom(λ),*

(c) $\nu(l) = \mu(l)$ *for all l such that $\lambda(l) \leq \tau$,*

```
class TimingChannel implements Runnable {
   boolean val = false;
   TimingChannel()
           throws InterruptedException {
      new TimeSlicer(5);
      new Thread(this).start();
      try Thread.sleep(2);
      finally;
      System.out.println("val = true");
   }
   public void run() {
      double x;
      if (val)
          for (int i = 0; i < 64; i++)
              x = Math.exp(Math.PI) + i;
      System.out.println("val = false");
     }
   public static void main(String args[])
           throws InterruptedException {
      try new TimingChannel();
      finally;
   }
}
```

Figure 8. Timing Channel with Java Threads

(d) $[\mu \vdash p \Rightarrow ?] \xrightarrow{k} T$, *for $k \geq 0$, and*

(e) T *has a leaf of the form* $[\mu' \vdash p' \Rightarrow ?]$.

Then there is a location typing λ' and partial derivation tree T' such that T' has a leaf of the form $[\nu' \vdash p' \Rightarrow ?]$, $[\nu \vdash p \Rightarrow ?] \xrightarrow{k} T'$, $\lambda \subseteq \lambda'$, dom$(\mu')$ = dom(ν') = dom(λ'), and $\mu'(l) = \nu'(l)$, for all l such that $\lambda'(l) \leq \tau$.

Proof. Induction on k, using Lemmas 4.1 and 5.2. □

Clearly timing agreement is a stronger property than either termination agreement or mutual progress. But the cost for this added strength is a much more restrictive typing rule for conditionals. Though it might be argued the rule is impractical for writing systems software or TCB source code, it may be the kind of rule that should be used in writing "Web programs" like Java Applets. The reason is that with threads, timing differences become quite easy to observe from *within* programs.

For example, take the Java program in Figure 8. The idea is that we want to determine the contents of the boolean variable `val` by setting up two competing threads. The `main` thread creates another thread, the `TimingChannel` thread, whose `run` method checks whether `val` is true, doing some computation if it is and nothing otherwise, except print a string. Notice that the `run` method does not have any illegal implicit flows in the sense of Denning's program certification [2, 9]. There is a third thread,

called the `TimeSlicer`, which is a daemon thread running at a higher priority. It re-awakens every five milliseconds and immediately goes back to sleep which guarantees round-robin scheduling among the other two threads.[2] After creating the `TimingChannel` thread, the main thread sleeps for two milliseconds. If it awakens before the `TimingChannel` thread completes, then the first string output will be `val = true`, otherwise it will be `val = false`. The first string usually reflects the variable's contents accurately. This is not a completely reliable way of getting the contents due to thread scheduling variations, but it works often enough.

So it seems that conditionals should also be typed minimally. But this may not be the best way to deal with them. After all, unlike the earlier Java examples, threads here seem to have a critical role. Perhaps with a proper treatment of threads, conditionals won't need to be typed so restrictively.

6. Conclusion

The idea of analyzing source code for covert information flow is not new. He and Gligor, for example, informally describe analyzing TCB source code for such flows [4]. Others have recognized the need to augment Denning's original secure-flow certification with rules that deal with *global flows* arising from loops and possibly nonterminating programs [1, 5]. But these efforts provide no formal specification nor proof of the properties that are guaranteed to hold for programs that pass the analyses. In contrast, we have given a rigorous account of various properties that a program has if it is typeable in our covert-flow type system.

References

[1] G. Andrews and R. Reitman. An Axiomatic Approach to Information Flow in Programs. *ACM Trans. on Prog. Lang. and Systems*, 2(1):56–76, 1980.

[2] D. Denning and P. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 20(7):504–513, 1977.

[3] C. Gunter. *Semantics of Programming Languages*. The MIT Press, 1992.

[4] J. He and V. Gligor. Formal Methods and Automated Tool for Timing-Channel Identification in TCB Source Code. In *Proceedings 2nd European Symposium on Research in Computer Security*, pages 57–75, November 1992.

[5] M. Mizuno and A. Oldehoeft. Information Flow Control in a Distributed Object-Oriented System with Statically-Bound Object Variables. In *Proceedings 10th National Computer Security Conference*, pages 56–67, 1987.

[2] The timeslicer was needed because our example was developed using Solaris JDK 1.02 which, unlike the JDK for Windows NT, does not schedule threads in a round-robin fashion.

[6] M. Ozgen. A Type Inference Algorithm and Transition Semantics for Polymorphic C. Master's thesis, Naval Postgraduate School, 1996.

[7] G. Smith and D. Volpano. A Sound Polymorphic Type System for a Dialect of C. *Science of Computer Programming*, 1997. To appear.

[8] D. Volpano and G. Smith. A Type Soundness Proof for Variables in LCF ML. *Information Processing Letters*, 56(3):141–146, 1995.

[9] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(3):167–187, 1996.