Information-Flow Security for Interactive Programs

Kevin R. O'Neill   Michael R. Clarkson   Stephen Chong
{oneill,clarkson,schong}@cs.cornell.edu

Department of Computer Science
Cornell University

Abstract. Interactive programs allow users to engage in input and output throughout execution. The ubiquity of such programs motivates the development of models for reasoning about their information-flow security, yet no such models seem to exist for imperative programming languages. Further, existing language-based security conditions founded on noninteractive models permit insecure information flows in interactive imperative programs. This paper formulates new strategy-based information-flow security conditions for a simple imperative programming language that includes input and output operators. The semantics of the language enables a fine-grained approach to the resolution of nondeterministic choices. The security conditions leverage this approach to prohibit refinement attacks while still permitting observable nondeterminism. Extending the language with probabilistic choice yields a corresponding definition of probabilistic noninterference. A soundness theorem demonstrates the feasibility of statically enforcing the security conditions via a simple type system. These results constitute a step toward understanding and enforcing information-flow security in real-world programming languages, which include similar input and output operators.

1 Introduction

Secure programs should maintain the secrecy of confidential information. For sequential imperative programming languages, this principle has led to a variety of information-flow security conditions which assume that all confidential information is supplied as the initial values of a set of program variables. This assumption reflects an idealized *batch-job* model of input and output, whereby all inputs are obtained (as initial values of program variables) from users before the program begins execution, and all outputs are provided (as final values of program variables) after program termination. Accordingly, these security conditions aim to protect the secrecy only of initial values.

Many real-world programs are *interactive*, sending output to and receiving input from their external environment throughout execution. Examples of such programs include web servers, GUI applications, and some command-line applications. The batch-

This work was supported in part by NSF under grants CTC-0208535 and 0430161, by ONR under grant N00014-01-10-511, by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the ONR under grants N00014-01-1-0795 and N00014-04-1-0725, and by AFOSR under grants F49620-02-1-0101 and FA9550-05-1-0055.

job model is unable to capture the behavior of interactive programs because of dependencies between inputs and outputs. For example, a program implementing a challenge/response protocol must first output a challenge to the user and then accept the user's response as input; clearly, the user cannot supply the response as the initial value of a program variable. In contrast, the interactive model generalizes the batch-job model: any batch-job program can be simulated by an interactive program that reads the initial values of all relevant variables, executes the corresponding batch-job program, and finally outputs the values of all variables.

Given the prevalence of interactive programs, it is important to be able to reason about their security properties. Traditionally, researchers have reasoned about information flow in interactive systems by encoding them as state machines (e.g., Mantel [19] and McLean [22, 23]) or as concurrent processes (e.g., Focardi and Gorrieri [6]) and applying trace-based information-flow security conditions. But since implementors usually create *imperative programs*, not abstract models, a need exists for tools that enable direct reasoning about the security of such programs. This paper addresses that need by developing a model for reasoning about the information-flow security of interactive imperative programs. Our model achieves a clean separation of user behavior from program code by employing *user strategies*, which describe how agents interact with their environment. Strategies are closely related to processes described in a language like CCS [24] or CSP [17]. We give novel strategy-based semantic security conditions similar to Wittbold and Johnson's definition of *nondeducibility on strategies* [38], which ensure that confidential information cannot flow from high-confidentiality users to low-confidentiality users. We also leverage previous work on static analysis techniques by adapting the type system of Volpano, Smith, and Irvine [37] to an interactive setting.

Our language and security conditions synthesize two branches of information-flow security research, in that we leverage the trace-based definitions that have been proposed for interactive systems to provide novel security conditions for imperative programs. Furthermore, our interactive programming language can be viewed as a specification language for interactive systems that more closely approximates the implementation of real programs than the abstract system models that have previously been used.

Nondeterminism arises in real-world systems for a number of reasons, including concurrency and probabilistic randomization. It is therefore an important consideration when reasoning about imperative programs. Nondeterminism is orthogonal to interactivity, but the interplay between information flow and nondeterminism is often quite subtle. We examine two kinds of nondeterministic choices: those which we assume are made probabilistically, and those which we are unable or unwilling to assign probabilities. We refer to the former as *probabilistic choice*, and to the latter as *nondeterministic choice*. Following Halpern and Tuttle [15], we factor out nondeterministic choice so that we can reason about it in isolation from probabilistic choice. By explicitly representing the resolution of nondeterministic choice in the language semantics, we adapt our security condition to rule out *refinement attacks* in which the resolution of nondeterministic choice results in insecure information flows. Finally, we give a security condition, based on Gray and Syverson's definition of *probabilistic noninterference* [11], that rules out probabilistic information flows in randomized interactive programs.

In Section 2 we develop our system model and introduce mathematical structures for reasoning about the behavior and observations of users. We proceed to instantiate the model on a simple language of while-programs in Section 3 and to give an operational semantics and security condition for the language. We then incorporate language features for nondeterministic choice (Section 4) and probabilistic choice (Section 5) and adapt our security conditions accordingly. In Section 6 we demonstrate the feasibility of statically enforcing our security condition by presenting a sound type system. Section 7 discusses related work, and Section 8 concludes.

2 User Strategies

It might seem at first that information-flow security for interactive programs can be obtained by adopting the same approach used for batch-job programs, that is, by preventing low-confidentiality users from learning anything about high-confidentiality inputs. (Hereafter we use the more concise terms "high" and "low" when describing the confidentiality level associated with inputs, users, and so on.) However, several papers, starting with Wittbold and Johnson [38], have described systems in which high users can transmit information to low users even though low users learn nothing about the high inputs. This is demonstrated by Program P_1 below, an insecure one-time pad implementation described by Wittbold and Johnson. Command **input** x **from** C reads a value from a channel named C and stores it in variable x; similarly, **output** e **to** C outputs the value of expression e on a channel named C. Assume that low users may use only channel L, that high users may use channel H, and that no users may observe the values of program variables. Infix operator $[\!]$ nondeterministically chooses to execute one of its two operands.

$$\begin{array}{ll} P_1: & \textbf{while}\ (\textbf{true})\ \textbf{do} \\ & \quad x := 0 \ [\!]\ x := 1; \\ & \quad \textbf{output}\ x\ \textbf{to}\ H; \\ & \quad \textbf{input}\ y\ \textbf{from}\ H; \\ & \quad \textbf{output}\ x\ \textbf{xor}\ (y\ \textbf{mod}\ 2)\ \textbf{to}\ L \end{array}$$

If nondeterminism is resolved in a way that is unpredictable to the low user, he will be unable to determine the inputs on channel H: for any output on L, the input on H could have been either 0 or 1. Yet the high user can still communicate an arbitrary confidential bit z to channel L at each iteration of the loop by choosing z **xor** x as input on H.

The confidential information z is never directly acquired by the program: it is neither the initial value of a program variable nor an input supplied on a channel. As Wittbold and Johnson observe, maintaining the secrecy of all high inputs (and even the initial values of program variables) is therefore insufficient to preserve the secrecy of confidential information.

In Program P_1, the high user is able to communicate arbitrary confidential information by selecting his next input as a function of outputs he has previously received. This suggests that if we want to prevent confidential information from flowing to low users, we should protect the secrecy of the function that high users employ to select inputs. Following Wittbold and Johnson's terminology, we call this function a *user strategy*. In the remainder of this section we develop the mathematical structures needed to define user strategies formally.

2.1 Types, Users, and Channels

We assume a set $\mathcal{L}$ of security types with ordering relation $\leq$ and use metavariable τ to range over security types. For simplicity, we assume that $\mathcal{L}$ equals $\{L, H\}$ with $L \leq H$. (Our results generalize to partial orders of security types.) Security type L represents low confidentiality, and H represents high confidentiality. The ordering $\leq$ indicates the relative restrictiveness of security types: high-confidentiality information is more restricted in its use than low-confidentiality information.

Users are agents (including humans and programs) that interact with executing programs. We associate with each user a security type indicating the highest level of confidential information that the user is permitted to read. Conservatively, we assume that users of the same security type may collaborate while attempting to subvert the security of a program. We can thus simplify our security analyses by reasoning about exactly two users, one representing the pooled knowledge of low users and another representing the pooled knowledge of high users.

We also assume the existence of *channels* with blocking input and nonblocking output. Although input is blocking, we assume that all inputs prompted for are eventually supplied. Each channel is associated with a security type τ, and only users of that type are permitted to use the channel. For simplicity, we assume that there are exactly two channels, L and H. We also assume that the values that are input and output on channels are integers. These are not fundamental restrictions; our results could be extended to allow multiple channels of each type, to allow high users to observe low channels, and to allow more general data types.

2.2 Traces

An *event* is the transmission of an input or output on a channel. Denote the input of value v on the channel of type τ as $in(\tau, v)$ and the output of v on τ as $out(\tau, v)$. Let $\mathbf{Ev}(\tau)$ be the set of all events that could occur on channel τ:

$$\mathbf{Ev}(\tau) \quad \triangleq \quad \bigcup_{v \in \mathbb{Z}} \{in(\tau, v), out(\tau, v)\}.$$

Let $\mathbf{Ev}$ be the set of all events:

$$\mathbf{Ev} \quad \triangleq \quad \bigcup_{\tau \in \mathcal{L}} \mathbf{Ev}(\tau).$$

We use metavariable α to range over events in $\mathbf{Ev}$.

A *trace* is a finite list of events. Given $E \subseteq \mathbf{Ev}$, an *event trace on* E is a finite, possibly empty list $\langle \alpha_1, \ldots, \alpha_n \rangle$ such that $\alpha_i \in E$ for all i. The empty trace is written $\langle\rangle$. The set of all traces on E is denoted $\mathbf{Tr}(E)$, and we abbreviate the set of all traces $\mathbf{Tr}(\mathbf{Ev})$ as $\mathbf{Tr}$. Trace equality is defined pointwise, and the concatenation of two traces t and t' is denoted $t \hat{} t'$. A trace t' *extends* trace t if there exists a trace t'' such that

$t' = t\hat{\ }t''$. The *restriction of t to E*, denoted $t \restriction E$, is the trace that results from removing all events not contained in E from t. We write $t \restriction \tau$ as shorthand for $t \restriction \mathbf{Ev}(\tau)$. A *low trace* is the low restriction $t \restriction L$ of a trace t.

2.3 User Strategies

As demonstrated by Program P_1, the input supplied by a user may depend on past events observed by that user. To capture this dependence we employ a *user strategy*, which determines the input for a particular channel as a function of the events that occur on the channel. Because events on a channel include both inputs and outputs, this function depends on both the user's observations and previous actions. Formally, a user strategy for a channel with security type τ is a function of type $\mathbf{Tr}(\mathbf{Ev}(\tau)) \to \mathbb{Z}$. Let **UserStrat** be the set of all user strategies. (Note that, to simulate the batch-job model, the initial inputs provided by users can be represented by a constant strategy that selects inputs without regard for past inputs or outputs. Also, high user strategies can be extended to depend on observation of the low channel, as described at the end of Section 3.)

As an example, we present a strategy that a high user could employ to transmit an arbitrary stream of bits $z_1 z_2 \ldots$ to the low user in Program P_1. This user strategy, g, ensures that if b was the previous output on H, then the next input on H is the bitwise exclusive-or of b and z_i. Note that every second event on channel H is an input event $in(H, v)$.

$$g(\langle \alpha_1, \ldots, \alpha_n \rangle) = \begin{cases} z_i \text{ xor } b & \text{if } \alpha_n = out(H, b) \\ & \quad \text{and } n = 2i - 1 \\ 0 & \text{otherwise} \end{cases}$$

A *joint strategy* is a collection of user strategies, one for each channel. Formally, a joint strategy ω is a function of type $\mathcal{L} \to$ **UserStrat**, that is, a function from security types to user strategies. Let **Strat** be the set of all joint strategies.

3 Noninterference for Interactive Programs

While-programs, extended with commands for input and output, constitute our core interactive programming language. The syntax of this language is:

$$\begin{array}{ll} \text{(expressions)} & e ::= n \mid x \mid e_0 \oplus e_1 \\ \text{(commands)} & c ::= \textbf{skip} \mid x := e \mid c_0; c_1 \mid \\ & \qquad \textbf{input } x \textbf{ from } \tau \mid \textbf{output } e \textbf{ to } \tau \mid \\ & \qquad \textbf{if } e \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{while } e \textbf{ do } c \end{array}$$

Metavariable x ranges over **Var**, the set of all program variables. Variables take values in $\mathbb{Z}$, the set of integers. Literal values n also range over integers. Binary operator $\oplus$ denotes any total binary operation on the integers.

3.1 Operational Semantics

The execution of a program modifies the values of variables and produces events on channels. A *state* determines the values of variables. Formally, a state is a function

(ASSIGN)

$$\frac{}{(x := e, \sigma, t, \omega) \longrightarrow (\textsf{\textbf{skip}}, \sigma[x := \sigma(e)], t, \omega)}$$

(SEQ-1)

$$\frac{}{(\textsf{\textbf{skip}}; c, \sigma, t, \omega) \longrightarrow (c, \sigma, t, \omega)}$$

(SEQ-2)

$$\frac{(c_0, \sigma, t, \omega) \longrightarrow (c_0', \sigma', t', \omega)}{(c_0; c_1, \sigma, t, \omega) \longrightarrow (c_0'; c_1, \sigma', t', \omega)}$$

(IN)

$$\frac{\omega(\tau)(t \upharpoonright \tau) = v}{(\textsf{\textbf{input}}\ x\ \textsf{\textbf{from}}\ \tau, \sigma, t, \omega) \longrightarrow (\textsf{\textbf{skip}}, \sigma[x := v], t\hat{\ }\langle in(\tau, v)\rangle, \omega)}$$

(OUT)

$$\frac{\sigma(e) = v}{(\textsf{\textbf{output}}\ e\ \textsf{\textbf{to}}\ \tau, \sigma, t, \omega) \longrightarrow (\textsf{\textbf{skip}}, \sigma, t\hat{\ }\langle out(\tau, v)\rangle, \omega)}$$

(IF-1)

$$\frac{\sigma(e) \neq 0}{(\textsf{\textbf{if}}\ e\ \textsf{\textbf{then}}\ c_0\ \textsf{\textbf{else}}\ c_1, \sigma, t, \omega) \longrightarrow (c_0, \sigma, t, \omega)}$$

(IF-2)

$$\frac{\sigma(e) = 0}{(\textsf{\textbf{if}}\ e\ \textsf{\textbf{then}}\ c_0\ \textsf{\textbf{else}}\ c_1, \sigma, t, \omega) \longrightarrow (c_1, \sigma, t, \omega)}$$

(WHILE)

$$\frac{}{(\textsf{\textbf{while}}\ e\ \textsf{\textbf{do}}\ c, \sigma, t, \omega) \longrightarrow (\textsf{\textbf{if}}\ e\ \textsf{\textbf{then}}\ (c; \textsf{\textbf{while}}\ e\ \textsf{\textbf{do}}\ c)\ \textsf{\textbf{else skip}}, \sigma, t, \omega)}$$

Fig. 1. Operational semantics

of type **Var** $\rightarrow$ $\mathbb{Z}$. Let σ range over states. A *configuration* is a 4-tuple (c, σ, t, ω) representing a system about to execute c with state σ and joint strategy ω. Trace t is the history of events produced by the system so far. Let m range over configurations. Terminal configurations, which have no commands remaining to execute, have the form $(\textsf{\textbf{skip}}, \sigma, t, \omega)$.

The operational semantics for our language is a small-step relation $\longrightarrow$ on configurations. Membership in the relation is denoted

$$(c, \sigma, t, \omega) \longrightarrow (c', \sigma', t', \omega),$$

meaning that execution of command c can take a single step to command c', while updating the state from σ to σ'. Trace t' extends t with any events that were produced during the step. Note that joint strategy ω is unchanged when a configuration takes a step; we include it in the configuration only to simplify notation and presentation.

The inductive rules defining relation $\longrightarrow$ are given in Figure 1. The rules for commands other than input and output are all standard. In Rule ASSIGN, $\sigma(e)$ denotes the value of expression e in state σ, and state update $\sigma[x := v]$ changes the value of variable x to v in σ. Rule IN uses the joint strategy ω to determine the next input event and appends it to the current trace, and rule OUT simply appends the output event to the current trace.

Let $\longrightarrow^*$ be the reflexive transitive closure of $\longrightarrow$. Intuitively, if

$$(c, \sigma, t, \omega) \longrightarrow^* (c', \sigma', t', \omega),$$

then configuration (c, σ, t, ω) can reach configuration $(c', \sigma', t', \omega)$ in zero or more steps. Configuration m *emits* t, denoted $m \rightsquigarrow t$, when there exists a configuration (c, σ, t, ω) such that $m \longrightarrow^* (c, \sigma, t, \omega)$. Note that emitted events may include both inputs and outputs.

3.2 A Strategy-Based Security Condition

We now develop a security condition which ensures that users with access only to channel L do not learn anything about the strategies employed by users interacting with channel H. Since strategies encode the possible actions that users may take as they interact with the system, protecting the secrecy of high strategies ensures that the actions taken by high users cannot affect (or "interfere with") the observations of low users. The security condition can be seen as an instance of *nondeducibility on strategies* as defined by Wittbold and Johnson [38] or as an instance of definitions of secrecy given by Halpern and O'Neill [13, 14].

Informally, a program is secure if, for every initial state σ, any trace of events seen on channel L is consistent with every possible user strategy for channel H. This ensures that low users cannot learn any information, including inputs, that high users attempt to convey—even if low users know the program text.

Definition 1 (Noninterference). A command c satisfies *noninterference* exactly when:

For all $m = (c, \sigma, \langle\rangle, \omega)$ and $m' = (c, \sigma, \langle\rangle, \omega')$
such that $\omega(L) = \omega'(L)$,
and for all t such that $m \rightsquigarrow t$,
there exists a t' such that $t \upharpoonright L = t' \upharpoonright L$ and $m' \rightsquigarrow t'$.

According to this condition, the high strategy $\omega(H)$ in m can be replaced by any other high strategy without affecting the low traces emitted. Although the condition assumes that programs begin with an empty trace of prior events, it can be generalized to account for arbitrary traces. (See Appendix A.) Some additional implications of this security condition are discussed below.

Initial variable values. The security condition does not protect the secrecy of the initial values of variables. More concretely, the program **output** x **to** L is considered secure for any $x \in$ **Var**, whereas the program **input** x **from** H; **output** x **to** L is obviously considered insecure. The definition thus reflects our intuition that high users interact with the system only via input and output events on the high channel and have no control over the initialization of variables. Systems in which the high user controls the initial values of some or all variables can be modeled by prepending commands that read inputs from the high user and assign them to variables.

Variable typings. It is not necessary to assign security types to program variables in order to determine whether a program is secure. (A program with no high inputs, for example, is secure regardless of its variables or their types.) Accordingly, our security condition makes no reference to the security types of variables. This distinguishes our work from most batch-job conditions, where variable typings are fundamental. We do, however, employ variable typings for the static analysis technique presented in Section 6.

Timing sensitivity. Our observational model is *asynchronous*: users do not observe the time when events occur or the time that passes while a program is blocking on an input command. The security condition is thus timing-insensitive. We could incorporate timing sensitivity into the model by assuming that users observe a "tick" event at each execution step or by tagging events with the time at which they occur; strategies could then make use of this additional temporal information.

Termination sensitivity. We make the standard assumption that users are unable to observe the nontermination of a program. Nonetheless, our security condition is termination-sensitive when low events follow commands that may not terminate. Consider the following program:

$$\begin{array}{ll} P_2: & \textbf{input } x \textbf{ from } H; \\ & \textbf{if } (x = 0) \textbf{ then } \{\textbf{while } (\textbf{true}) \textbf{ do skip}\} \textbf{ else skip}; \\ & \textbf{output } 1 \textbf{ to } L \end{array}$$

A high user can cause this program to transmit the value 1 to a low user. Since this would allow the low user to infer something about the high strategy, this program is insecure according to our security condition.

We do not assume that users are able to observe the termination of a program directly, but it would be easy to make termination observable by adding a distinguished termination event that is broadcast on all channels when execution reaches a terminal configuration.

Observation of channels. We have assumed that high users cannot observe the low channel, but this restriction can be removed in several ways. For example, it is straightforward to amend the operational semantics to echo low events to high channels by adding an additional high output event (prepended with a label to distinguish it from a regular high output events) to the trace every time a low input or output event occurs.

4 Nondeterministic Programs

We distinguish two kinds of nondeterminism that appear in programs: *probabilistic choice* and *nondeterministic choice*. Intuitively, probabilistic choice represents explicit use of randomization, whereas nondeterministic choice represents program behavior that is underspecified (perhaps due to unpredictable factors such as the scheduler in a concurrent setting). Following the approach of previous work [15, 35], we factor out the latter kind of nondeterminism by assuming that all nondeterministic choices are made as if they were specified before the program began execution. (The implications of this approach are discussed at the end of the section.) This allows reasoning about nondeterministic choice and probabilistic choice in isolation, and our definitions of noninterference reflect the resulting separation of concerns. In this section we extend our model to include nondeterministic choice. We return to probabilistic choice in Section 5.

4.1 Refiners

We extend the language of Section 3 with nondeterministic choice:

$$c \quad ::= \quad \ldots \mid c_0 \ [\!]_\tau \ c_1$$

Each nondeterministic choice is annotated with a security type τ that is used in the operational semantics. The need for the annotation is explained below; we remark, however, that the type system described in Section 6 could be used to infer annotations automatically, so that programmers need not specify them.

To factor out the resolution of nondeterminism, we introduce infinite lists of binary values called *refinement lists*. Denote the set of all such refinement lists as **RefList**. Informally, when a nondeterministic choice is encountered during execution, the head element of a refinement list is removed and used to resolve the choice. The program executes the left command of the nondeterministic choice if the element is 0 and the right command if the element is 1. Refinement lists are an operational analog of Milner's *oracle domains* [25] for denotational semantics.

Nondeterministic choices should not cause insecure information flows, even if low users can predict how the choices will be made. While it might seem that using a single refinement list would suffice to ensure that no insecure information flows arise as a result of the resolution of nondeterministic choice, the following program demonstrates that this is not the case:

$$\begin{array}{l} \textbf{input } x \textbf{ from } H; \\ \textbf{if } (x = 0) \textbf{ then } \{\textbf{skip } [\!]_H \textbf{ skip}\} \textbf{ else skip}; \\ \textbf{output } 0 \textbf{ to } L \ [\!]_L \textbf{ output } 1 \textbf{ to } L \end{array}$$

If the refinement list $\langle 1, 0, \ldots \rangle$ is used to execute this program, the output on channel L will equal the input on channel H. An insecure information flow arises because the same refinement list is used to make both low and high choices. To eliminate this flow, we identify the security type of a choice based on its annotation and require that different lists be used to resolve choices at each type. This ensures that the number of choices made at a given security level cannot become a covert channel. (Note that this requirement lends itself to natural implementation techniques. For example, if choices are made by using a stream of pseudorandom numbers, then different streams should be used to resolve high and low choices. Or if $[\!]$ represents scheduler choices, then the scheduler should resolve choices at each security type independently.)

A *refiner* is a function $\psi : \mathcal{L} \to$ **RefList** that associates a refinement list with each security type. Let **Ref** denote the set of all refiners. Denote the standard list operations of reading the first element of a list and removing the first element of a list as $head$ and $tail$, respectively. Given a refiner ψ, the value $head(\psi(\tau))$ is used to resolve the next choice annotated with type τ.

$$\text{(SEQ-2)}\quad \frac{(c_0, \sigma, \psi, t, \omega) \longrightarrow (c_0', \sigma', \psi', t', \omega)}{(c_0; c_1, \sigma, \psi, t, \omega) \longrightarrow (c_0'; c_1, \sigma', \psi', t', \omega)}$$

$$\text{(CHOICE)}\quad \frac{head(\psi(\tau)) = i}{(c_0 \ [\!]_\tau \ c_1, \sigma, \psi, t, \omega) \longrightarrow (c_i, \sigma, \psi[\tau := tail(\psi(\tau))], t, \omega)}$$

Fig. 2. Operational semantics for nondeterministic choice

4.2 Operational Semantics

Using refiners, we extend the operational semantics of Section 3 to account for nondeterministic choice. A command c is now executed with respect to a refiner ψ, in addition to a state σ, trace t, and joint strategy ω. We thus modify configurations to be 5-tuples $(c, \sigma, \psi, t, \omega)$; terminal configurations now have the form $(\textbf{skip}, \sigma, \psi, t, \omega)$.

All of the operational rules from Figure 1 are adapted in the obvious way to handle the new configurations. The only interesting change is SEQ-2, which is restated in Figure 2. Nondeterministic choice is evaluated by the new rule CHOICE, which uses refiner ψ to resolve the choice and specifies how the refiner changes as a result. Refiner $\psi[\tau := tail(\psi(\tau))]$ is the refiner ψ with the refinement list for τ replaced by $tail(\psi(\tau))$.

Note that a refiner factors out all nondeterminism in the program: once a refiner, state, and joint strategy have been fixed, execution is completely determined.

4.3 A Security Condition for Nondeterministic Programs

A well-known problem arises with nondeterministic programs: they are vulnerable to *refinement attacks*, in which a seemingly secure program can be refined to an insecure program. For example, whether the input from H is kept secret in the following program depends on how the nondeterministic choice is resolved:

$$\begin{aligned} P_3 : \quad & \textbf{input}\ x\ \textbf{from}\ H; \\ & \textbf{output}\ 0\ \textbf{to}\ L\ \ [\!]\ \ \textbf{output}\ 1\ \textbf{to}\ L \end{aligned}$$

If the choice is made independently of the current state of the program, say by tossing a coin, the program is secure. But if the choice is made as a function of x, the program may leak information about the high input.

To ensure that a program is resistant to refinement attacks, we insist that, for all possible resolutions of nondeterminism, the program does not leak any confidential information. Our model allows this quantification to be expressed cleanly, since refiners encapsulate the resolution of nondeterministic choice. We adapt the security condition of Section 3.2 to ensure that, for any refinement of the program, users with access only to channel L do not learn anything about the strategies employed by users of channel H.

Definition 2 (Noninterference Under Refinement). A command c satisfies *noninterference under refinement* exactly when:

For all $m = (c, \sigma, \psi, \langle\rangle, \omega)$ and $m' = (c, \sigma, \psi, \langle\rangle, \omega')$
such that $\omega(L) = \omega'(L)$,
and for all t such that $m \rightsquigarrow t$,
there exists a t' such that $t \restriction L = t' \restriction L$ and $m' \rightsquigarrow t'$.

Some implications of this definition are discussed below.

Low-observable nondeterminism. This security condition rules out refinement attacks but allows programs that appear nondeterministic to a low user. For example, Program P_3 (with $[]$ replaced by $[]_L$) satisfies noninterference under refinement, yet repeated executions may reveal different program behavior to the low user.

Initial refinement lists. The security condition does not require the secrecy of the initial refinement list for H. More concretely, the program

$$\textsf{\textbf{output}}\ 0\ \textsf{\textbf{to}}\ L\ \ []_H\ \ \textsf{\textbf{output}}\ 1\ \textsf{\textbf{to}}\ L$$

is considered secure even though it reveals information about the first value of $\psi(H)$. The definition thus reflects our intuition that high users interact with the system only via input and output events on the high channel, which gives them no control over refinement lists. The definition of noninterference under refinement could be adapted to systems where high users may exert control over refinement lists.

Expressivity of refiners. Our model can represent only those refinements that appear as if they were made before the program began execution. Refinements that may depend upon dynamic factors, such as the values of variables or the current program counter, cannot be represented. Our model therefore captures *compiler-time nondeterminism* but not *runtime nondeterminism* [16]. We leave development of more sophisticated refiners as future work.

5 Probabilistic Programs

Probabilistic choice can be seen as refinement of arbitrary nondeterministic choice. Now that we have shown how refiners can be used to factor out the nondeterministic choices to which we are unable or unwilling to assign probabilities, we can model probabilistic choice explicitly.

We begin by extending the nondeterministic language of Section 4 with probabilistic choice:

$$c \quad ::= \quad \dots \ \mid \ c_0 \ {}_p[] \ \ c_1$$

Informally, probabilistic choice $c_0 \ {}_p[] \ \ c_1$ executes command c_0 with probability p and command c_1 with probability $1-p$. The probability annotation p must be a real number such that $0 \leq p \leq 1$. We assume that probabilistic choices are made independently of one another.

(PROB-1)

$$\overline{(c_0\ {}_p[\!]\ \ c_1, \sigma, \psi, t, \omega) \xrightarrow{p} (c_0, \sigma, \psi, t, \omega)}$$

(PROB-2)

$$\overline{(c_0\ {}_p[\!]\ \ c_1, \sigma, \psi, t, \omega) \xrightarrow{1-p} (c_1, \sigma, \psi, t, \omega)}$$

Fig. 3. Operational semantics for probabilistic choice

5.1 Operational Semantics

To incorporate probability in the operational semantics we extend the small-step relation $\longrightarrow$ of previous sections to include a label for probability. We denote membership in the new relation by

$$m \xrightarrow{p} m',$$

meaning that configuration m steps with probability p to configuration m'. Configurations remain unchanged from the nondeterministic language of Section 4. The new operational rules defining this relation are given in Figure 3. To facilitate backwards-compatibility with the operational rules of previous sections, we interpret $m \longrightarrow m'$ as shorthand for $m \xrightarrow{1} m'$. The operational rules previously given in Figures 1 and 2 thus remain unchanged.

5.2 A Probabilistic Security Condition

It is well-known that probabilistic programs may be secure with respect to nonprobabilistic definitions of noninterference but leak confidential information with high probability. As an example, consider the following program:

$$\begin{array}{ll} P_4: & \textbf{input}\ x\ \textbf{from}\ H; \\ & \textbf{if}\ x\ \textbf{mod}\ 2 = 0\ \textbf{then} \\ & \quad \textbf{output}\ 0\ \textbf{to}\ L\ {}_{0.99}[\!]\ \ \textbf{output}\ 1\ \textbf{to}\ L \\ & \textbf{else} \\ & \quad \textbf{output}\ 0\ \textbf{to}\ L\ {}_{0.01}[\!]\ \ \textbf{output}\ 1\ \textbf{to}\ L \end{array}$$

If we regard probabilistic choice ${}_p[\!]$ as identical to nondeterministic choice $[\!]_L$, then this program satisfies noninterference under refinement. Yet with high probability, the program leaks the parity of the high input to channel L.

Toward preventing such *probabilistic information flows*, observe that if a low trace t is likely to be emitted with one high user strategy and unlikely with another, then the low user learns something about the high strategy by observing the occurrence of t. We thus conclude that our security condition should require that the probability with which low traces are emitted be independent of the strategy employed on the high channel, that is, that low-equivalent configurations should produce particular low traces with the same probability. This intuition is consistent with security conditions given by Gray and Syverson [11] and Halpern and O'Neill [14].

More formally, let $\mathcal{E}_m(t)$ represent the event that configuration m emits low trace t. Suppose that we had a probability μ_m on such events. Then our security condition should require, for all configurations m and m' that are equivalent except for the choice of high strategy, and all low traces t, that $\mu_m(\mathcal{E}_m(t)) = \mu_{m'}(\mathcal{E}_{m'}(t))$. The remainder of this section is devoted to defining μ_m and $\mathcal{E}_m(t)$.

We begin with two additional intuitions. First, since probabilistic choices are made independently, the probability of an *execution sequence*

$$m_0 \xrightarrow{p_0} m_1 \xrightarrow{p_1} \ldots \xrightarrow{p_{n-1}} m_n$$

is equal to the product of the probabilities p_i of the individual steps. Second, a configuration m could emit the same trace t along multiple sequences, so the probability that m emits t should be the sum of the probabilities associated with those sequences.

Based on these intuitions, we now construct probability measure μ_m by adapting a standard approach for reasoning about probabilities on trees [12]. For any configuration m, relation $\xrightarrow{p}$ gives rise to a rooted directed *probability tree* whose vertices are labeled with configurations, edges are labeled with probabilities, and root is m. Denote the probability tree for m by $\mathcal{T}_m$ and the set of vertices of $\mathcal{T}_m$ by $\mathcal{V}_m$. A *path* in the tree is a sequence of vertices, starting with the root, where each successive pair of vertices is an edge. Given a vertex v, let $tr(v)$ be the trace of events in the configuration with which v is labeled. We say that t *appears at* v when $tr(v) = t$ but $tr(v') \neq t$ for all ancestors v' of v. Let $ap(t)$ be the set of vertices where t appears. In accordance with the intuitions described above, let $\pi(v)$ be the product of the probabilities on the path to v.

A *ray* is an infinite path or a finite path whose terminal node has no descendants, Rays therefore represent maximal execution sequences. Let $\mathcal{R}_m$ denote the set of rays of $\mathcal{T}_m$. Let $R_m(v)$ be the set of rays that go through vertex v:

$$R_m(v) \quad \triangleq \quad \{r \in \mathcal{R}_m \mid v \text{ is on } r\}.$$

Let $\mathcal{A}_m$ be the σ-algebra on $\mathcal{R}_m$ generated by sets of rays going through particular vertices, that is, by the set $\{R_m(v) \mid v \in \mathcal{V}_m\}$.[1] The following result yields a probability measure on sets of rays. It is a consequence of elementary results in probability theory, and we omit the proof.

Theorem 1. For any configuration m, there exists a unique probability measure μ_m on $\mathcal{A}_m$ such that for all $v \in \mathcal{V}_m$ we have $\mu_m(R_m(v)) = \pi(v)$.

Now that we have constructed μ_m, we must show how to use it to obtain the probability of a set of traces in terms of the probability of a corresponding set of rays. For a set T of traces, let $R_m(T)$ be the set of rays on which a trace in T appears. Let $em_m(T) = \{t \in T \mid m \rightsquigarrow t\}$ be the set of traces in T emitted by m, and note that

$$R_m(T) \quad \triangleq \quad \bigcup_{t \in em_m(T)} \; \bigcup_{v \in ap(t)} R_m(v),$$

[1] A σ-algebra on a set X is a nonempty collection of subsets of X that contains X and is closed under complements and countable unions [2]. (The σ has no connection to states, although we also use σ as a metavariable that ranges over states.) A σ-algebra generated by a set $\mathcal{C}$ of subsets of X is defined as the intersection of all σ-algebras on X, including 2^X, that contain $\mathcal{C}$.

because a trace appears on a ray r if and only if it appears at a vertex v on r. The set $R_m(T)$ is measurable with respect to $\mathcal{A}_m$ because both $em_m(T)$ and $\mathcal{V}_m$ are countable sets. Given a trace t, the set $\{R_m(v) \mid v \in ap(t)\}$ is a partition of the set of rays on which t appears. It follows that

$$\begin{array}{rcl} \mu_m(R_m(\{t\})) & = & \mu_m(\bigcup_{v \in ap(t)} R_m(v)) \\ & = & \sum_{v \in ap(t)} \mu_m(R_m(v)) \\ & = & \sum_{v \in ap(t)} \pi(v), \end{array}$$

that is, that the probability that m emits t is equal to the sum of the values $\pi(v)$ for vertices v where t appears, as desired.

We can now define $\mathcal{E}_m(t)$. Given a security type τ and a trace t, let $[t]_\tau$ be the equivalence class of traces that are equal to t when restricted to τ:

$$[t]_\tau \quad \triangleq \quad \{t' \in \mathbf{Tr} \mid t' \upharpoonright \tau \,=\, t \upharpoonright \tau\}.$$

Finally, let $\mathcal{E}_m(t)$ be the set of rays on which there is some vertex v such that $tr(v) \upharpoonright L = t \upharpoonright L$:

$$\mathcal{E}_m(t) \quad \triangleq \quad R_m([t]_L).$$

The set $\mathcal{E}_m(t)$ is in $\mathcal{A}_m$. By Theorem 1, $\mu_m(\mathcal{E}_m(t))$ is equal to the sum of values $\pi(v)$ for vertices v such that $tr(v) \upharpoonright L = t \upharpoonright L$ and $tr(v') \upharpoonright L \neq t \upharpoonright L$ for any ancestor v' of v.

We are now ready to formalize our security condition.

Definition 3 (Probabilistic Noninterference). A command c satisfies *probabilistic noninterference* exactly when:

For all $m = (c, \sigma, \psi, \langle\rangle, \omega)$ and $m' = (c, \sigma, \psi, \langle\rangle, \omega')$
  such that $\omega(L) = \omega'(L)$,
and for all $t \in \mathbf{Tr}(\mathbf{Ev}(L))$,
we have $\mu_m(\mathcal{E}_m(t)) = \mu_{m'}(\mathcal{E}_{m'}(t))$.

Returning to Program P_4 at the start of this section, it is easy to check that the probability of the low trace $\langle out(L, 0)\rangle$ is 0.99 when the high strategy is to input an even number, and 0.01 when the high strategy is to input an odd number. Clearly, the program does not satisfy probabilistic noninterference.

If we interpret the nondeterministic choice in Program P_1 as ${}_{0.5}[]$ (a fair coin toss), the program does not satisfy probabilistic noninterference. However, if the output to H is removed, the resulting program

```
while (true) do
  x := 0 0.5[] x := 1;
  input y from H;
  output x xor (y mod 2) to L
```

does satisfy noninterference. The probability of low outputs is independent of the high strategy, which can no longer exploit knowledge of the value of one-time pad x.

User strategies as defined thus far are deterministic. However, our approach to reasoning about probability applies to randomized user strategies as well as to randomized programs, so it would be straightforward to adapt our model to handle randomized strategies.

6 A Sound Type System

The problem of characterizing programs that satisfy noninterference is, for many definitions of noninterference, intractable. For definitions appearing in the previous sections, there is a straightforward reduction from the halting problem to the noninterference problem. It follows that no decision procedure for certifying the information-flow security of programs can be both sound and complete with respect to our definitions of noninterference. The goal of this section is to demonstrate that static analysis techniques can be used to identify secure programs.

We use a type system based on that of Volpano, Smith, and Irvine [37]. It is interesting to note that a type-system designed to enforce batch-job noninterference conditions also enforces our interactive conditions, including probabilistic noninterference, even though the type system is oblivious to the subtleties of probability, interactivity, and user strategies. We believe that other type systems for information flow (e.g., [3, 18, 32, 34]) can also be easily adapted for our interactive model, and thus that advances in precision and expressiveness can be applied to our work.

The type system consists of a set of axioms and inference rules for deriving *typing judgments* of the form $\Gamma \vdash p : \kappa$, meaning that phrase p has phrase type κ under variable typing Γ. A *phrase* is either an expression or a command. A *phrase type* is either a security type τ or a command type τ *cmd*, where $\tau \in \mathcal{L}$. A *variable typing* is a function $\Gamma : \textbf{Var} \to \mathcal{L}$ mapping from variables to security types. Informally, a command c has type τ *cmd* when τ is a lower-bound on the effects that c may have, that is, when the types (under Γ) of any variables that c updates are bounded below by τ, and any input or output that c performs is on channels whose security type is bounded below by τ.

Axioms and inference rules for the type system are given in Figure 4. There are two types of rules: typing rules (prefixed with "T") and subtyping rules (prefixed with "ST"). Typing rules can be used to infer the type of an expression or command directly. Subtyping rules allow a low-typed expression to be treated as a high-typed expression and a high-typed command to be treated as a low-typed command. (It is safe, for example, to store a low-typed expression in a high variable, or to output data to a high user in the body of a loop with a low-typed guard.)

Most of the rules in this type system are standard. Rules T-IN and T-OUT are both similar to T-ASSIGN: T-IN ensures that values read from the τ channel are stored in variables whose type is bounded below by τ, whereas T-OUT ensures that only τ-typed expressions are output on the τ channel. Rules T-CHOICE and T-PROB are similar to T-SEQ, except that T-CHOICE also checks that the typing is consistent with the syntactic type annotation. Rule T-WHILE forbids high-guarded loops, ensuring that loop termination does not depend on the high user's strategy. This prohibits insecure programs such as P_2 (in Section 3.2). We believe this rule could be relaxed using techniques described by Boudol and Castellani [3] and Smith [34].

The following theorem states that this type system soundly enforces noninterference. Recall that our security conditions do not depend on the security types of vari-

$$\text{(T-LIT)}\ \frac{}{\Gamma \vdash n : \tau} \qquad \text{(T-VAR)}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{(T-OP)}\ \frac{\Gamma \vdash e_0 : \tau \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0 \oplus e_1 : \tau} \qquad \text{(T-ASSIGN)}\ \frac{\Gamma(x) = \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \tau\ \mathit{cmd}}$$

$$\text{(T-SKIP)}\ \frac{}{\Gamma \vdash \textbf{skip} : \tau\ \mathit{cmd}}$$

$$\text{(T-IF)}\ \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c_0 : \tau\ \mathit{cmd} \quad \Gamma \vdash c_1 : \tau\ \mathit{cmd}}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_0\ \textbf{else}\ c_1 : \tau\ \mathit{cmd}}$$

$$\text{(T-SEQ)}\ \frac{\Gamma \vdash c_0 : \tau\ \mathit{cmd} \quad \Gamma \vdash c_1 : \tau\ \mathit{cmd}}{\Gamma \vdash c_0; c_1 : \tau\ \mathit{cmd}}$$

$$\text{(T-WHILE)}\ \frac{\Gamma \vdash e : L \quad \Gamma \vdash c : \tau\ \mathit{cmd}}{\Gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : L\ \mathit{cmd}} \qquad \text{(T-CHOICE)}\ \frac{\Gamma \vdash c_0 : \tau\ \mathit{cmd} \quad \Gamma \vdash c_1 : \tau\ \mathit{cmd}}{\Gamma \vdash c_0 \ []_{\tau}\ c_1 : \tau\ \mathit{cmd}}$$

$$\text{(T-PROB)}\ \frac{\Gamma \vdash c_0 : \tau\ \mathit{cmd} \quad \Gamma \vdash c_1 : \tau\ \mathit{cmd}}{\Gamma \vdash c_0\ {}_{p}[]\ c_1 : \tau\ \mathit{cmd}}$$

$$\text{(T-IN)}\ \frac{\Gamma(x) = \tau' \quad \tau \leq \tau'}{\Gamma \vdash \textbf{input}\ x\ \textbf{from}\ \tau : \tau\ \mathit{cmd}} \qquad \text{(T-OUT)}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{output}\ e\ \textbf{to}\ \tau : \tau\ \mathit{cmd}}$$

$$\text{(T-SUBTYPE)}\ \frac{\Gamma \vdash p : \kappa_0 \quad \kappa_0 \leq \kappa_1}{\Gamma \vdash p : \kappa_1}$$

$$\text{(ST-BASE)}\ \frac{}{L \leq H} \qquad \text{(ST-REFL)}\ \frac{}{\kappa \leq \kappa} \qquad \text{(ST-CMD)}\ \frac{\tau_0 \leq \tau_1}{\tau_1\ \mathit{cmd} \leq \tau_0\ \mathit{cmd}}$$

Fig. 4. Typing rules

ables. Noninterference is enforced provided there exists some variable typing under which the program is well-typed.[2] The proof is in Appendix A.

Theorem 2 (Soundness). For any command c, if there exists a variable typing Γ and a security type τ such that $\Gamma \vdash c : \tau\ cmd$, then

(a) if c does not contain nondeterministic or probabilistic choice, then c satisfies noninterference;
(b) if c does not contain probabilistic choice, then c satisfies noninterference under refinement; and
(c) c satisfies probabilistic noninterference.

7 Related Work

Definitions of information-flow security for imperative programs began with the work of Denning [5]. Many subsequent papers define information-flow security for various sequential imperative languages, but nearly all of these papers assume a batch-job model of computation. Therefore, they attempt to ensure the secrecy of high-typed program variables rather than of the behavior of high users who interact with the system. See Sabelfeld and Myers [30] for a survey of language-based information-flow security.

Another line of work considers end-to-end information-flow restrictions for nondeterministic systems that provide input and output functionality for users. Definitions of noninterference exist both for abstract systems (such as finite state machines) that include input and output operations (Goguen and Meseguer [10], McCullough [21], McLean [23], Mantel [19]), and for systems described using process algebras such as CCS, the π-calculus, and related formalisms (Focardi and Gorrieri [6], Ryan and Schneider [28], Zdancewic and Myers [39]).

Definitions of noninterference based on process algebras typically require that the observations made by a public user are the same regardless of which high processes (if any) are interacting with the system. These definitions are thus similar in spirit to our definitions of noninterference. Indeed, there is a close connection between strategies and processes: both can be viewed as description of how an agent will behave in an interactive setting. A formal comparison with process-based definitions (such as [8]) may uncover further connections between process-based system models and imperative programs.

Wittbold and Johnson [38] give the first strategy-based definition of information-flow security, and Gray and Syverson [11] give a strategy-based definition of probabilistic noninterference. Halpern and O'Neill [14] generalize the definitions of Gray and Syverson to account for richer system models and more general notions of uncertainty. Our definitions of noninterference, which are instances of Halpern and O'Neill's definitions of secrecy, are the first strategy-based security conditions for an imperative programming language of which we are aware. Our work can thus be viewed as a unification of two distinct strands of the information-flow literature. In this sense our work

[2] Because the security types of variables can be inferred, programmers need not specify them. In a (trivially secure) program with no high inputs, for example, all variables can be assigned type L.

is similar to that of Mantel and Sabelfeld [20], who demonstrate a connection between security predicates taken from the *MAKS* framework of Mantel [19] and bisimulation-based definitions of security for a concurrent imperative language due to Sabelfeld and Sands [31]. However, Mantel and Sabelfeld do not consider interactive programs.

Our probabilistic noninterference condition can be interpreted as precluding programs that allow low users to make observations that improve the accuracy of their *beliefs* about high behavior, that is, their beliefs about which high strategy is used. Halpern and O'Neill [14] prove a result which implies that our probabilistic security condition suffices to ensure that low users cannot improve the accuracy of their subjective beliefs about high behavior by interacting with a program. Our probabilistic security condition also ensures that the quantity of information flow due to a secure program is exactly zero bits in the belief-based quantitative information-flow model of Clarkson, Myers, and Schneider [4].

The bisimulation-based security condition of Sabelfeld and Sands [31] can be viewed as a relaxation of the batch-job model. However, as Mantel and Sabelfeld [20] point out, bisimulation-based definitions are difficult to relate to trace-based conditions when a nondeterministic choice operator is present in the language. The following program, for example, satisfies both noninterference under refinement and probabilistic noninterference (for suitable interpretations of the $[]$ operator), but it is not secure with respect to a bisimulation-based definition of security:

$$
\begin{array}{l}
\textbf{input } x \textbf{ from } H;\\
\textbf{if } (x = 0)\\
\quad \textbf{output } 0 \textbf{ to } L;\\
\quad \{\textbf{output } 1 \textbf{ to } L \;\; [] \;\; \textbf{output } 2 \textbf{ to } L\}\\
\textbf{else}\\
\quad \{\textbf{output } 0 \textbf{ to } L;\ \textbf{output } 1 \textbf{ to } L\} \; []\\
\quad \{\textbf{output } 0 \textbf{ to } L;\ \textbf{output } 2 \textbf{ to } L\}
\end{array}
$$

Bisimulation-based security conditions implicitly assume that users can observe internal choices made by a program. When users observe only inputs and outputs on channels, our observational model is more appropriate.

Interactivity between users and a program is similar to message-passing between threads. Sabelfeld and Mantel [29] present a multi-threaded imperative language with explicit send, blocking receive, and non-blocking receive operators for communication between processes. They describe a bisimulation-based security condition and a type system to enforce it. However, it is not clear how to model user behavior in their setting. Users cannot be modeled as processes since user behavior is unknown, and their security condition applies only if the entire program is known.

Almeida Matos, Boudol, and Castellani [1] state a bisimulation-based security condition for *reactive programs*, which allow limited communication between processes, and they give a sound type system to enforce the condition. In their language, programs react to the presence and absence of named broadcast signals and can emit signals to other programs in a "local area." It is possible to implement our higher-level channels and events within a local area, using their lower-level reactivity operators. However, it is unclear how to use reactivity to model interactions with unknown users who are not part of a local area.

Focardi and Rossi [7] study the security of processes in *dynamic contexts* where the environment, including high processes, can change throughout execution. This is similar to how high user strategies describe changing inputs throughout execution. However, user strategies depend upon the history of the computation, whereas dynamic contexts do not, so it is unclear how to encode a user strategy using dynamic contexts.

Previous work dealing with the susceptibility of possibilistic noninterference to refinement attacks takes one of two approaches to specifying how nondeterministic choice is resolved. One approach is to assume that choices are made according to fixed probability distributions, as we do in Section 5. Volpano and Smith [36], for example, describe a scheduler for a multithreaded language that chooses threads to execute according to a uniform probability distribution. A second approach is to insist that programs be *observationally deterministic* for low users. McLean [22] and Roscoe [27] both advocate observational determinism as an appropriate security condition for nondeterministic systems, and Zdancewic and Myers [39] give a security condition based on observational determinism for a concurrent language based on the join calculus [9].

Observational determinism implies noninterference under refinement and thus immunity to refinement attacks. In settings where the resolution of nondeterministic choice may depend on confidential information, we conjecture that observational determinism and noninterference under refinement are equivalent. However, when the resolution of some choices is independent of confidential information, observational determinism is a stronger condition: any program that is observationally deterministic satisfies noninterference under refinement, but the converse does not hold.

8 Conclusion

This paper examines information flow in a simple imperative language that includes primitives for communication with program users. In this setting, it is not the initial values of variables or the inputs from high users that must be kept secret, but rather the high users' strategies. We present a trace-based noninterference condition which ensures that low users do not learn anything about the strategies employed by high users. Incorporating nondeterministic and probabilistic choice in the language leads to corresponding security conditions: noninterference under refinement and probabilistic noninterference. We prove that a type system conservatively enforces these security conditions.

This work is a step toward understanding and enforcing information-flow security in real-world programs. Many programs interact with users, and the behavior of these users will often be dependent on previous inputs and outputs. Also, many programs, especially servers, are intended to run indefinitely rather than to perform some computation and then halt. Our model of interactivity is thus more suitable for analyzing real-world systems than the batch-job model. In addition, our imperative language approximates the implementation of real-world interactive programs more closely than abstract system models such as the π-calculus. This paper thereby contributes to understanding the security properties of programs written in languages with information flow control, such as Jif [26] or Flow Caml [33], that support user input and output.

Acknowledgments

We thank Dan Grossman, Joseph Halpern, Gregory Lawler, Jed Liu, Andrew Myers, Nathaniel Nystrom, Fred B. Schneider, and Lantian Zheng for discussions, technical suggestions, and comments on this paper.

A Proof Sketch for Theorem 2

For the proof of Theorem 2, we treat the proof for nonprobabilistic noninterference—that is, the proof of parts (a) and (b)—separately from the proof of probabilistic noninterference. This is technically unnecessary, because the necessary lemmas for the probabilistic proof are generalizations of the lemmas for the nonprobabilistic proof. We take this approach so that we can prove the nonprobabilistic lemmas, which are simpler to understand, without the overhead of probability trees and probability distributions. The nonprobabilistic lemmas are proven in Section A.1, whereas the probabilistic lemmas and Theorem 2 are proven in Section A.2.

All of the results in this section assume the existence of a single variable typing Γ. When convenient, we avoid specifying Γ and assume that the typing is given.

We heavily overload the symbol $\sim_L$ to represent low equivalence relations. We write $\sigma \sim_L \sigma'$ to denote that states σ and σ' are low-equivalent with respect to Γ, that is, if $\sigma(x) = \sigma'(x)$ whenever $\Gamma(x) = L$. Refiners $\psi, \psi' \in$ **Ref** are low-equivalent, written $\psi \sim_L \psi'$, if $\psi(L) = \psi'(L)$. Similarly, joint strategies $\omega, \omega' \in$ **Strat** are low-equivalent, written $\omega \sim_L \omega'$ if $\omega(L) = \omega'(L)$. Traces t and t' are low-equivalent, written $t \sim_L t'$, if $t \restriction L = t' \restriction L$.

The low-equivalence relation on well-typed commands, denoted $c \sim_L c'$, is defined by the following rules:

(a) $c \sim_L c$ for all commands c;
(b) if $\Gamma \vdash c_1 : H\ cmd$ and $\Gamma \vdash c_2 : H\ cmd$, then $c_1 \sim_L c_2$;
(c) if $\Gamma \vdash c_1 : H\ cmd$ and $\Gamma \vdash c_2 : H\ cmd$, then $c_1; c \sim_L c_2; c$ for all commands c; and
(d) if $\Gamma \vdash c_H : H\ cmd$, then $c_H; c \sim_L c$ and $c \sim_L c_H; c$ for all commands c.

Property 1. The relation $\sim_L$ is an equivalence relation on well-typed commands.

Proof. Reflexivity is immediate by rule (a), and symmetry follows because the rules themselves are symmetric. Transitivity follows by a straightforward analysis of each pair of rules. □

Two configurations $m = (c, \sigma, \psi, t, \omega)$ and $m' = (c', \sigma', \psi', t', \omega')$ are low-equivalent, written $m \sim_L m'$, if $c \sim_L c'$, $\sigma \sim_L \sigma'$, $\psi \sim_L \psi'$, $t \sim_L t'$, and $\omega \sim_L \omega'$.

A.1 Nonprobabilistic Proof Details

The following lemma, an analogue of the "Simple Security" lemma of [37], demonstrates that low-typed expressions have the same values in low-equivalent states.

Lemma 1. If $\Gamma \vdash e : L$, then $\Gamma(x) = L$ for every variable x appearing in e. In particular, if $\Gamma \vdash e : L$ and $\sigma \sim_L \sigma'$, then $\sigma(e) = \sigma'(e)$.

Proof. By induction on the structure of e. $\square$

The following lemma demonstrates that configurations with high-typed commands take steps that preserve low-equivalence (in the sense that no low events are emitted and the resulting configuration is low-equivalent to the initial configuration).

Lemma 2. If $\Gamma \vdash c : H$ *cmd*, then for all σ, ψ, t, and ω, if

$$(c, \sigma, \psi, t, \omega) \longrightarrow (c', \sigma', \psi', t', \omega'),$$

then $(c, \sigma, \psi, t, \omega) \sim_L (c', \sigma', \psi', t', \omega')$, and moreover $\Gamma \vdash c' : H$ *cmd*.

Proof. By induction on the derivation of $(c, \sigma, \psi, t, \omega) \longrightarrow (c', \sigma', \psi', t', \omega')$. $\square$

The following lemma demonstrates that if the first command in a sequence terminates with some configuration, then the sequence eventually steps to an identical configuration with **skip** replaced by the second command in the sequence.

Lemma 3. For all $c_0, c_1, \sigma, \sigma', \psi, \psi', t, t', \omega$, and ω', if

$$(c_0, \sigma, \psi, t, \omega) \longrightarrow^* (\textbf{skip}, \sigma', \psi', t', \omega'),$$

then

$$(c_0; c_1, \sigma, \psi, t, \omega) \longrightarrow^* (c_1, \sigma', \psi', t', \omega').$$

Proof. By induction on the length of the derivation of

$$(c_0, \sigma, \psi, t, \omega) \longrightarrow^* (\textbf{skip}, \sigma', \psi', t', \omega'),$$

using rule SEQ-1 for the base case and rule SEQ-2 for the inductive case. $\square$

The following lemma demonstrates that high-typed commands always terminate, and that the resulting terminal configuration is low-equivalent to the initial configuration.

Lemma 4. If $\Gamma \vdash c : H$ *cmd*, then for any σ, ψ, t and ω there exists σ', ψ', t' and ω' such that

$$(c, \sigma, \psi, t, \omega) \longrightarrow^* (\textbf{skip}, \sigma', \psi', t', \omega'),$$

and

$$(c, \sigma, \psi, t, \omega) \sim_L (\textbf{skip}, \sigma', \psi', t', \omega').$$

Proof. Note that a high-typed command cannot contain a **while**-statement. The result follows by structural induction on c, using Lemma 2 to demonstrate low equivalence for the base cases. For sequences we appeal to Lemma 3. $\square$

The following lemma demonstrates that low-equivalent configurations with the same command take steps that preserve low equivalence.

Lemma 5. For all $c, \sigma_1, \sigma_2, \psi_1, \psi_2, t_1, t_2, \omega_1, \omega_2$, and m_1, if

$$(c, \sigma_1, \psi_1, t_1, \omega_1) \sim_L (c, \sigma_2, \psi_2, t_2, \omega_2), \text{ and } (c, \sigma_1, \psi_1, t_1, \omega_1) \longrightarrow m_1,$$

then there exists a configuration m_2 such that

$$(c, \sigma_2, \psi_2, t_2, \omega_2) \longrightarrow m_2, \text{ and } m_1 \sim_L m_2.$$

Proof. By induction on the derivation $(c, \sigma_1, \psi_1, t_1, \omega_1) \longrightarrow m_1$, using Lemma 1 for the rules ASSIGN, OUT, IF-1, and IF-2. □

The main nonprobabilistic lemma demonstrates that the traces emitted by low-equivalent configurations are low-equivalent.

Lemma 6. For all configurations m_1, m_2, and m_1', if

$$m_1 \sim_L m_2 \text{ and } m_1 \longrightarrow^* m_1',$$

then there exists a configuration m_2' such that

$$m_2 \longrightarrow^* m_2' \text{ and } m_1' \sim_L m_2'.$$

Proof. By induction on the length of the derivation of $m_1 \longrightarrow^* m_1'$. The base case is trivial. Otherwise, write $m_1 = (c_1, \sigma_1, \psi_1, t_1, \omega_1)$ and $m_2 = (c_2, \sigma_2, \psi_2, t_2, \omega_2)$, and consider the cases for $c_1 \sim_L c_2$:

(a) If $c_1 = c_2$, then suppose that $m_1 \longrightarrow m_1''$ and $m_1'' \longrightarrow^* m_1'$. By Lemma 5, there is a state m_2'' such that $m_2 \longrightarrow m_2''$ and $m_1'' \sim_L m_2''$. We can then apply the inductive hypothesis.

(b) If c_1 and c_2 are both high-typed, suppose that $m_1 \longrightarrow m_1''$ and $m_1'' \longrightarrow^* m_1'$. By Lemma 2, m_1'' is low equivalent to m_2, and we can apply the inductive hypothesis.

(c) If $c_1 = c_{H_1}; c$ and $c_2 = c_{H_2}; c$ for some command c and high-typed commands c_{H_1} and c_{H_2}, then consider the form of c_{H_1}. If $c_{H_1} = \textbf{skip}$, then $m_1 \longrightarrow \hat{m}$, where $\hat{m} = (c, \sigma_1, \psi_1, t_1, \omega_1)$, and since $(c, \sigma_1, \psi_1, t_1, \omega_1)$ is low equivalent to m_2, we can apply the inductive hypothesis. Otherwise, by Lemma 2 and SEQ-2, $m_1 \longrightarrow m_1''$ for some m_1'' such that is low equivalent to m_2, and we can apply the inductive hypothesis.

(d) If $c_1 = c_{H_1}; c_2$, then consider the form of c_{H_1}. If $c_{H_1} = \textbf{skip}$, then $m_1 \longrightarrow (c_2, \sigma_1, \psi_1, t_1, \omega_1)$, and since $(c_2, \sigma_1, \psi_1, t_1, \omega_1)$ is low equivalent to m_2, we can apply the inductive hypothesis. Otherwise, by Lemma 2 and SEQ-2, $m_1 \longrightarrow m_1''$ such that m_1'' is low equivalent to m_2, and we can apply the inductive hypothesis.
If $c_2 = c_{H_2}; c_1$, then by Lemma 4 and Lemma 3, there is a configuration $m_2'' = (c_1, \sigma_2'', \psi_2'', t_2'', \omega_2'')$ such that $m_2 \longrightarrow^* m_2''$ and $m_2 \sim_L m_2''$, and thus $m_1 \sim_L m_2''$. Suppose $m_1 \longrightarrow m_1''$ and $m_1'' \longrightarrow^* m_1'$. Then by Lemma 5, there is a configuration m_2''' such that $m_2'' \longrightarrow m_2'''$ such that $m_1'' \sim_L m_2'''$, and we can apply the inductive hypothesis.

□

The first two cases of Theorem 2 follow directly from this result.

A.2 Probabilistic Proof Details

We now generalize the results of the previous section to account for probabilistic programs. The structure of the proof is similar to the nonprobabilistic results.

Given a vertex v of a probability tree $\mathcal{T}_m$, let $\mathcal{T}_v$ denote the subtree of $\mathcal{T}_m$ rooted at v, and let $\mathcal{V}_v$ and $\mathcal{R}_v$ denote the sets of vertices and rays of $\mathcal{T}_v$, respectively. We denote the configuration with which v is labeled as $cf(v)$, and we write $v \sim_L v'$ if $cf(v) \sim_L cf(v')$.

Let a *frontier set* of a vertex v be a finite set of vertices $S \subseteq \mathcal{V}_v$ such that for every ray $r \in \mathcal{R}_v$ there exists exactly one vertex from S on r. Given a frontier set S of v, we call $F = (v, S)$ a *frontier*. Note that $\{v\}$ is a frontier set of v, and that given any frontier F we can obtain a new frontier F' by replacing any vertex in the frontier set with all of its descendants. Note also that a frontier set S partitions $\mathcal{R}_v$ into sets of rays that go through particular vertices in S.

Define the *depth* of a frontier (v, S) to be the length of the longest path (that is, the number of edges in the longest path) between v and vertices in S.

Because the vertices in a frontier $F = (v, S)$ induce a partition on the sets of rays going through v, the function π on vertices gives rise to a discrete probability measure on sets of vertices on S, normalized by the value of $\pi(v)$. More concretely, for any vertex v' in a frontier set S of v, let $\pi_v(v')$ be the product of the probabilities on the path from v to v'. We can now compare the distribution of low-equivalent configurations in two different frontiers. Given a frontier $F = (v, S)$ and a configuration m, let

$$[m]_F \quad \triangleq \quad \{v' \in S \mid cf(v') \sim_L m\}$$

be the subset of S whose configurations are low-equivalent to m. Define two frontiers $F = (v, S)$ and $F' = (v', S')$ to be low-equivalent, denoted $F \sim_L F'$, if for any configuration m we have

$$\sum_{v'' \in [m]_F} \pi_v(v'') = \sum_{v'' \in [m]_{F'}} \pi_{v'}(v'').$$

The following lemma, which generalizes Lemma 3, demonstrates that if the first command in a sequence terminates in all execution paths, then the sequence eventually steps, in all execution paths, to the second command, while preserving other aspects of the original terminal configuration.

Lemma 7. If v is a vertex in a probability tree such that $cf(v) = (c_0; c_1, \sigma, \psi, t, \omega)$, and $F_0 = (v_0, S_0)$ is a frontier such that

- $cf(v_0) = (c_0, \sigma, \psi, t, \omega)$,
- the subtree rooted at v_0 is finite, and
- S_0 consists of the root vertices of the subtree rooted at v_0,

then there exists a frontier $F = (v, S)$ and a one-to-one mapping $g : S_0 \to S$ such that if $v_0' \in S_0$ and $cf(v_0') = (\textsf{skip}, \sigma', \psi', t', \omega')$, we have $cf(g(v_0')) = (c_1, \sigma', \psi', t', \omega')$.

Proof. By induction on the depth of F_0, using rules SEQ-1 and SEQ-2. □

The following lemma, which generalizes Lemma 4, demonstrates that high-typed commands terminate in all execution paths and that terminal configurations are low-equivalent to the initial configuration.

Lemma 8. If v is a vertex in a probability tree such that $cf(v) = (c, \sigma, \psi, t, \omega)$ and $\Gamma \vdash c : H\ cmd$, then the subtree rooted at v is finite and that for any leaf vertex v' of that subtree we have $v' \sim_L v$.

Proof. By structural induction on c, using Lemma 7 for sequences and Lemma 2 to demonstrate low-equivalence for the base cases. □

The following lemma, generalizing Lemma 5, demonstrates that low-equivalent vertices have low-equivalent sets of children.

Lemma 9. If $F_1 = (v_1, S_1)$ and $F_2 = (v_2, S_2)$ are frontiers such that

- $v_1 \sim_L v_2$,
- S_1 and S_2 are the sets of children of v_1 and v_2, and
- $cf(v_1)$ and $cf(v_2)$ share the same command c,

then $F_1 \sim_L F_2$.

Proof. By structural induction on c, using Lemma 1 for the rules ASSIGN, OUT, IF-1, and IF-2. □

The following lemma is useful for the inductive cases of Lemma 11. It states that we can combine frontiers of vertices in low-equivalent frontiers to obtain deeper low-equivalent frontiers.

Lemma 10. If $F_1 = (v_1, S_1)$ and $F_2 = (v_2, S_2)$ are frontiers such that

- $F_1 \sim_L F_2$;
- g_1 is a mapping from S_1 to frontier sets such that for any $v \in S_1$, $(v, g_1(v))$ is a frontier;
- g_2 is a mapping from S_2 to frontier sets such that for any $v \in S_2$, $(v, g_2(v))$ is a frontier; and
- for all $v_1' \in S_1$ and $v_2' \in S_2$ such that $v_1' \sim_L v_2'$, we have $(v_1', g_1(v_1')) \sim_L (v_2', g_2(v_2'))$;

then $F_1' = (v_1, \cup_{v \in S_1} g_1(v))$ and $F_2' = (v_2, \cup_{v \in S_2} g_2(v))$ are frontiers such that $F_1' \sim_L F_2'$.

Proof. F_1' and F_2' are frontiers, which follows directly from the definition of a frontier set. To demonstrate the low-equivalence of F_1' and F_2', we must establish that for all configurations m we have

$$\sum_{v \in [m]_{F_1'}} \pi_{v_1}(v) = \sum_{v \in [m]_{F_2'}} \pi_{v_2}(v).$$

Let $\mathcal{M}$ denote a set of class representatives (for the equivalence relation $\sim_L$) of the set of configurations with which the elements of $S_1 \cup S_2$ are labeled. We have

$$\begin{array}{rcl}
\sum_{v\in[m]_{F_1'}} \pi_{v_1}(v) & = & \sum_{v_1'\in F_1} \sum_{v\in[m]_{g_1(v_1')}} \pi_{v_1}(v_1') \cdot \pi_{v_1'}(v) \\
& = & \sum_{m'\in\mathcal{M}} \sum_{v_1'\in[m']_{F_1}} \sum_{v\in[m]_{g_1(v_1')}} \pi_{v_1}(v_1') \cdot \pi_{v_1'}(v) \\
& = & \sum_{m'\in\mathcal{M}} \sum_{v_1'\in[m']_{F_1}} \pi_{v_1}(v_1') \cdot \sum_{v\in[m]_{g_1(v_1')}} \pi_{v_1'}(v).
\end{array}$$

However, by assumption, for any configuration m and for any $v_1' \in S_1$ and $v_2' \in S_2$ such that $v_1' \sim_L v_2'$, we have $(v_1', g_1(v_1')) \sim_L (v_2', g_2(v_2'))$, and thus

$$\sum_{v\in[m]_{g_1(v_1')}} \pi_{v_1'}(v) = \sum_{v\in[m]_{g_2(v_2')}} \pi_{v_2'}(v).$$

In fact, for any $w \in S_1$ such that $v_1' \sim_L w$, we also have

$$\sum_{v\in[m]_{g_1(v_1')}} \pi_{v_1'}(v) = \sum_{v\in[m]_{g_1(w)}} \pi_{w}(v).$$

Thus, this sum depends only on the low-equivalence class of v_1'. We capture this by defining $s(m', m)$ to be equal to this sum, resulting in the following equalities:

$$\begin{array}{rcll}
s(m', m) & = & \sum_{v\in[m]_{g_1(v_1')}} \pi_{v_1'}(v) & \text{for any } v_1' \in S_1 \text{ such that } v_1' \sim_L m' \\
& = & \sum_{v\in[m]_{g_2(v_2')}} \pi_{v_2'}(v) & \text{for any } v_2' \in S_2 \text{ such that } v_2' \sim_L m'.
\end{array}$$

We therefore have

$$\begin{array}{rcll}
\sum_{v\in[m]_{F_1'}} \pi_{v_1}(v) & = & \sum_{m'\in\mathcal{M}} \sum_{v_1'\in[m']_{F_1}} \pi_{v_1}(v_1') \cdot s(m', m) & \\
& = & \sum_{m'\in\mathcal{M}} s(m', m) \cdot \sum_{v_1'\in[m']_{F_1}} \pi_{v_1}(v_1') & \\
& = & \sum_{m'\in\mathcal{M}} s(m', m) \cdot \sum_{v_2'\in[m']_{F_2}} \pi_{v_2}(v_2') & [\text{as } F_1 \sim_L F_2] \\
& = & \sum_{v\in[m]_{F_2'}} \pi_{v_2}(v), & [\text{similarly}]
\end{array}$$

as desired. □

We can now state the main lemma, which generalizes Lemma 6.

Lemma 11. If $F_1 = (v_1, S_1)$ is a frontier and $v_2 \sim_L v_1$, then there exists a frontier set S_2 of v_2 such that $F_1 \sim_L (v_2, S_2)$.

Proof. By induction on the depth of F_1. The base case is trivial, and cases (a)–(d) are analogues of the cases in the nonprobabilistic proof. As before, write $cf(v_1) = (c_1, \sigma_1, \psi_1, t_1, \omega_1)$ and $cf(v_2) = (c_2, \sigma_2, \psi_2, t_2, \omega_2)$, and consider the cases for $c_1 \sim_L c_2$:

(a) Suppose that $c_1 = c_2$. If S_1' and S_2' are the sets of children of v_1 and v_2, we have $(v_1, S_1') \sim_L (v_2, S_2')$ by Lemma 9. The inductive hypothesis applies to $v_1' \in S_1'$ (with appropriate frontier sets $S_{v_1'} \subseteq S_1$) and elements $v_2' \in S_2'$ such that $v_1' \sim_L v_2'$, and the result follows by Lemma 10.

(b) If c_1 and c_2 are both high-typed, the result follows by Lemma 2, the inductive hypothesis applied to the children of v_1, and Lemma 10.
(c) If $c_1 = c_{H_1}; c$ and $c_2 = c_{H_2}; c$ for some command c and high-typed commands c_{H_1} and c_{H_2}, let S'_1 be the set of children of v_1. For each $v'_1 \in S'_1$ we have $v'_1 \sim_L v_1 \sim_L v_2$ (by SEQ-1 if c_{H_1} is **skip**, or by Lemma 2 and SEQ-2 otherwise), and we can therefore apply the inductive hypothesis. The result follows from Lemma 10.
(d) If $c_1 = c_{H_1}; c_2$, we can apply the inductive hypothesis using the same reasoning used for case (c). Otherwise we have $c_2 = c_{H_1}; c_1$. Let S'_1 be the set of children of v_1. By Lemma 7 and Lemma 8, there exists a frontier set S'_2 of v_2 such that for every element $v'_2 \in S'_2$, v'_2 is labeled with a configuration whose command is c_1, and $v'_2 \sim_L v_2 \sim_L v_1$. Given any $v'_2 \in S'_2$, let $S_{v'_2}$ be the set of children of v'_2. By Lemma 9 we have $(v_1, S'_1) \sim_L (v'_2, S_{v'_2})$, and the inductive hypothesis applies to elements $v'_1 \in S'_1$ as in case (a). By Lemma 10 we can combine the frontiers of the elements of $S_{v'_2}$ to get a frontier $F_{v'_2} = (v'_2, S_{v'_2})$ such that $F_{v'_2} \sim_L F_1$. Let $S_2 = \cup_{v'_2 \in S'_2} S_{v'_2}$. We have $(v_2, S_2) \sim_L F_1$ by Lemma 10.

□

We are now ready to prove Theorem 2. We do so by establishing a connection between $\mu_m(\mathcal{E}_m(t))$, the probability that configuration m emits trace t, and the probabilities of vertices of arbitrarily deep frontiers of $\mathcal{T}_m$. Given a trace t and probability tree $\mathcal{T}_m$ with root vertex v_r and frontier (v_r, S), define:

$$\mathcal{E}_S(t) \triangleq \{r \in \mathcal{R}_m \mid \text{there exists a vertex } v \in S \text{ on } r \text{ and a trace } t' \text{ such that } tr(v) \text{ extends } t' \text{ and } t' \upharpoonright L = t \upharpoonright L\}.$$

The set $\mathcal{E}_S(t)$ consists of those rays on which traces that are low-equivalent to t appear at vertices that are ancestors of elements in S. Intuitively, $\mu_m(\mathcal{E}_S(t))$ is an approximation of $\mu_m(\mathcal{E}_m(t))$.

Theorem 2 (Soundness). For any command c, if there exists a variable typing Γ and a security type τ such that $\Gamma \vdash c : \tau$ *cmd*, then

(a) if c does not contain nondeterministic or probabilistic choice, then c satisfies noninterference;
(b) if c does not contain probabilistic choice, then c satisfies noninterference under refinement; and
(c) c satisfies probabilistic noninterference.

Proof. That c satisfies noninterference and noninterference under refinement follows from Lemma 6. To demonstrate that c satisfies probabilistic noninterference, we must show that, for all low-equivalent configurations m and m' and traces t, that $\mu_m(\mathcal{E}_m(t)) = \mu_{m'}(\mathcal{E}_{m'}(t))$. We demonstrate that $\mu_m(\mathcal{E}_m(t)) \leq \mu_{m'}(\mathcal{E}_{m'}(t))$; the reverse inequality is symmetric. Suppose, by way of contradiction, that $\mu_m(\mathcal{E}_m(t)) > \mu_{m'}(\mathcal{E}_{m'}(t))$. We demonstrate below that there exists a sequence of frontier sets $\{S_i\}$ of the root vertex v_r of $\mathcal{T}_m$ such that $\mu_m(\mathcal{E}_{S_i}(t))$ converges to $\mu_m(\mathcal{E}_m(t))$. It follows there exists a frontier set S of $\mathcal{T}_m$ such that $\mu_m(\mathcal{E}_S(t)) > \mu_{m'}(\mathcal{E}_{m'}(t))$. But by Lemma 11, there exists a frontier $F' = (v'_r, S')$ of $\mathcal{T}_{m'}$ such that $F' \sim_L (v_r, S)$. Thus $\mu_{m'}(\mathcal{E}_{S'}(t)) = \mu_m(\mathcal{E}_S(t))$, and $\mu_{m'}(\mathcal{E}_{S'}(t)) > \mu_{m'}(\mathcal{E}_{m'}(t))$. This is a contradiction, because $\mathcal{E}_{S'}(t) \subseteq \mathcal{E}_{m'}(t)$.

We now exhibit a sequence of frontier sets $\{S_i\}$ such that $\mu_m(\mathcal{E}_{S_i}(t))$ converges to $\mu_m(\mathcal{E}_m(t))$. Consider the sequence $S_0, S_1, \ldots, S_i, \ldots$ of frontier sets of v_r that comprise all the vertices at depth i of $\mathcal{T}_m$. We have $\mathcal{E}_m(t) = \cup_{i \geq 0} \mathcal{E}_{S_i}(t)$, and for all i we have $\mathcal{E}_{S_i}(t) \subseteq \mathcal{E}_{S_{i+1}}(t)$. Convergence follows due to a standard result in probability theory [2]. $\square$

References

1. Ana Almeida Matos, Gérard Boudol, and Ilaria Castellani. Typing noninterference for reactive programs. In *Proc. Workshop on Foundations of Computer Security*, 2004.
2. Patrick Billingsley. *Probability and Measure*. Wiley-Interscience, 3rd edition, April 1995.
3. Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Lecture Notes in Computer Science*, 281(1):109–130, 2002.
4. Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Belief in information flow. In *Proc. 18th IEEE Computer Security Foundations Workshop*, pages 31–45, June 2005.
5. Dorothy E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, 1976.
6. Riccardo Focardi and Roberto Gorrieri. Classification of security properties (Part I: Information flow). In *Foundations of Security Analysis and Design*, pages 331–396. Springer, 2001.
7. Riccardo Focardi and Sabina Rossi. Information flow security in dynamic contexts. In *Proc. 15th IEEE Computer Security Foundations Workshop*, pages 307–319, 2002.
8. Riccardo Focardi, Sabina Rossi, and Andrei Sabelfeld. Bridging language-based and process calculi security. In *Proc. of Foundations of Software Science and Computation Structures (FOSSACS'05)*, volume 3441 of *LNCS*, April 2005.
9. Cédric Fournet and Georges Gonthier. The Reflexive CHAM and the Join-Calculus. In *Conf. Record 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, 1996.
10. Joseph A. Goguen and José Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
11. James W. Gray III and Paul F. Syverson. A logical approach to multilevel security of probabilistic systems. *Distributed Computing*, 11(2):73–90, 1998.
12. Joseph Y. Halpern. *Reasoning About Uncertainty*. MIT Press, Cambridge, Mass., 2003.
13. Joseph Y. Halpern and Kevin R. O'Neill. Secrecy in multiagent systems. In *Proc. 15th IEEE Computer Security Foundations Workshop*, pages 32–46, 2002.
14. Joseph Y. Halpern and Kevin R. O'Neill. Secrecy in multiagent systems. Available at `http://arxiv.org/pdf/cs.CR/0307057`, 2005.
15. Joseph Y. Halpern and Mark Tuttle. Knowledge, probability, and adversaries. *Journal of the ACM*, 40(4):917–962, 1993.
16. Jifeng He, K. Seidel, and A. McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28:171–192, 1997.
17. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
18. Sebastian Hunt and Dave Sands. On flow-sensitive security types. In *Conf. Record 33rd ACM Symposium on Principles of Programming Languages*, 2006.
19. Heiko Mantel. *A uniform framework for the formal specification and verification of information flow security*. PhD thesis, Universität des Saarlandes, 2003.
20. Heiko Mantel and Andrei Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *Journal of Computer Security*, 11(4):615–676, September 2003.

21. Daryl McCullough. Specifications for multi-level security and a hook-up property. In *Proc. IEEE Symposium on Security and Privacy*, pages 161–166, 1987.
22. John McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–58, 1992.
23. John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Security and Privacy*, pages 79–93, 1994.
24. R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Volume 92. Springer-Verlag, Berlin/New York, 1980.
25. Robin Milner. Processes: A mathematical model of computing agents. In H. E. Rose and J. C. Shepherdson, editors, *Proceedings of the Logic Colloquium, Bristol, July 1973*, pages 157–173, New York, 1975. American Elsevier Pub. Co.
26. Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, 2001–2005.
27. A. W. Roscoe. CSP and determinism in security modeling. In *Proc. IEEE Symposium on Security and Privacy*, 1995.
28. Peter Y. A. Ryan and Steve A. Schneider. Process algebra and non-interference. In *Proc. 12th IEEE Computer Security Foundations Workshop*, pages 214–227, 1999.
29. Andrei Sabelfeld and Heiko Mantel. Static confidentiality enforcement for distributed programs. In *Proceedings of the 9th International Static Analysis Symposium*, volume 2477 of *LNCS*, Madrid, Spain, September 2002. Springer-Verlag.
30. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.
31. Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. 13th IEEE Computer Security Foundations Workshop*, pages 200–214. IEEE Computer Society Press, July 2000.
32. Vincent Simonet. Fine-grained information flow analysis for a lambda-calculus with sum types. In *Proc. 15th IEEE Computer Security Foundations Workshop*, pages 223–237, June 2002.
33. Vincent 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.
34. Geoffrey Smith. A new type system for secure information flow. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 115–125, 2001.
35. Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proc. 26th IEEE Symp. on Foundations of Computer Science*, pages 327–338, 1985.
36. Dennis Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, November 1999.
37. Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
38. J. Todd Wittbold and Dale M. Johnson. Information flow in nondeterministic systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 144–161, May 1990.
39. Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *Proc. 16th IEEE Computer Security Foundations Workshop*, pages 29–43, Pacific Grove, California, June 2003.