Reactive Noninterference

Aaron Bohannon
University of Pennsylvania

Benjamin C. Pierce
University of Pennsylvania

Vilhelm Sjöberg
University of Pennsylvania

Stephanie Weirich
University of Pennsylvania

Steve Zdancewic
University of Pennsylvania

ABSTRACT

Many programs operate reactively—patiently waiting for user input, running for a while producing output, and eventually returning to a state where they are ready to accept another input (or occasionally diverging). When a reactive program communicates with multiple parties, we would like to be sure that it can be given secret information by one without leaking it to others.

Motivated by web browsers and client-side web applications, we explore definitions of noninterference for reactive programs and identify two of special interest—one corresponding to termination-insensitive noninterference for a simple sequential language, the other to termination-sensitive noninterference. We focus on the former and develop a proof technique for showing that program behaviors are secure according to this definition. To demonstrate the viability of the approach, we define a simple reactive language with an information-flow type system and apply our proof technique to show that well-typed programs are secure.

Categories and Subject Descriptors

F.1.2 [**Modes of Computation**]: Interactive and reactive computation

General Terms

Languages, Security, Theory

Keywords

Noninterference, information flow, reactive programming, web browsers, web applications

1. INTRODUCTION

Reactive programs, which repeatedly perform single-threaded computations in response to events generated by external agents (GUI button clicks, commands issued at a terminal, receipt of network packets, timer events, *etc.*), are ubiquitous. When a single reactive program may interact with multiple agents, questions of security and privacy immediately arise.

Web browsers and the client-side web applications they run are an obvious case in point, because they interact with both a local user and mutually untrusting, remote agents (e.g., web servers). Web client security has attracted significant interest [18, 15, 10, 11]. However, we still lack the theoretical tools to answer the question "What does it mean for a browser, running a web application, to be secure?"

A web client may receive data and programs from many different servers, and each program may attempt to read any browser data and communicate with any remote server. Most browsers adhere to the "same-origin policy" [20], which is intended to enforce isolation among programs and data from different servers. However, the same-origin policy is a *policy* in name only: in fact, it is a set of rather complex and subtle rules, with no high-level statement of the security property they are intended jointly to enforce. Moreover, it has several practical problems: First, it prevents the client from calling useful third-party web services. Second, it prevents client-side data integration or collaborative behavior, even when the data being handled is guaranteed never to leave the user's machine. Third, its rules involving subdomain relationships are complex and ambiguous [11]. And finally, because of its rigid restrictions, the same-origin policy is not applied to browser extensions or plug-ins.

Our goal in this paper is to offer a more principled approach to the sort of attack scenario that is addressed by the same-origin policy and to lay a foundation for more flexible enforcement mechanisms. In this scenario, the attacker is positioned at a remote host and can only communicate with the user's client over HTTP. However, the attacker may have written some or all of the event handlers running on the client. The goal of the attacker is to retrieve some private piece of information that the user would not knowingly authorize the attacker to have.[1]

When compared with models of execution that have been previously studied in the information flow literature, reactive programs possess a novel combination of features: interactivity with buffered, asynchronous communication; incremental output; the requirement that the system respond in some way to arbitrary inputs; and the possibility of nontermination in handlers. Web client programs illustrate all

[1]The same-origin policy also addresses scenarios in which the attacker wants to corrupt data from other web servers or to interfere with the functionality of other web pages in the browser; we are focusing only on confidentiality here.

of these features. They have handlers that wait idly for user input or responses to HTTP requests. After an event has occurred, the appropriate event handler runs from start to completion, possibly sending one or more messages to remote hosts or to the user. (To streamline the model, we can consider all updates to the browser display as "messages" to the user.) Outgoing messages are sent as execution proceeds, so a script need not terminate to produce visible output. When the browser is idle, *any* event can be processed (this feature has been called *input-totality* [14, 4, 22]), although it will typically result in a no-op if no handler exists. However, when the browser is running a script, additional events cannot be immediately processed because there is no notion of preemptive multitasking. Instead, additional events must be buffered and processed when and if the current script terminates.

Our aim is to find a natural definition of information-flow security for this execution model and to design a language-based enforcement technique for reactive programming languages that execute under this model. Before coming to the details of our development, though, we briefly summarize its connections to the most closely related prior work.

Goguen and Meseguer [5, 6] gave one of the first formal accounts of noninterference, and they did so in a setting of machines that accept inputs and produce outputs. Their "MLS property" [6]—informally, "the low-visible outputs of a system remain unchanged after dropping the high-level inputs"—offers an attractive template for the sort of succinct, high-level specification we are looking for here. However, their model does not apply to nonterminating behaviors (they rely on a total function that immediately moves the system to a new state after each input); adapting the same intuition to our model involves an entirely new mathematical development.

By contrast, some information-flow research [4, 22] has addressed execution models of concurrent systems with input and output that are somewhat more general than our reactive model. In stating the security properties for these systems, the observable behaviors of a system are usually characterized by a set of traces, each trace being a finite prefix of some, possibly infinite, sequence of transitions that might occur. This is a convenient tool for handling nondeterministic behavior, and it naturally encompasses nonterminating behavior as well. However, sets of traces are too abstract to capture some distinctions that we would like to make: in the standard trace representation, a system with the sole behavior of repeatedly outputting a single message forever is represented by the same set of traces as a system that repeatedly outputs that message until it nondeterministically decides to halt. We address this issue by using streams rather than sets of traces to represent infinite behaviors, leading to a notion of security that is fundamentally not expressible as a "security property" in the sense of Zakinthinos and Lee [22].

Besides articulating a natural information-flow property for reactive systems, we want to be able to enforce it using a security type system. There is a large body of research on such type systems (see Sabelfeld and Myers [19] for an overview). In particular, Volpano, Smith, and Irvine [21] develop a basic information-flow type system for sequential while-programs and prove that it guarantees a high-level noninterference property. Notably, their noninterference property is *termination-insensitive*, which means that a program may diverge on some high inputs and terminate on others. Since this allows the possibility of a covert *termination channel* in a program, it is technically less secure than a *termination-sensitive* property, but it is more practical for language-based enforcement because ruling out these termination channels either requires the elimination of too many useful, secure programs, or else requires the use of static termination checking. Thus, a question of particular interest to us is whether there exists a termination-insensitive definition of security for reactive systems.

While most work on language-based security has ignored the issue of incremental inputs and outputs, it has been addressed by some recent work on "interactive programs" [16, 9, 2]. In contrast with our execution model, the execution model of these interactive programs is based on synchronous communication that is not input-total: the languages have explicit input operations that block, waiting for designated principals to respond. In stating its security properties, this line of work must tackle some of the same issues that we do; for instance, Hunt and Sands [9] make use of infinite streams. However, all of their definitions are termination-sensitive, and in order to pursue a termination-insensitive notion of noninterference, we have found it necessary to set up a more general framework for defining information security in the presence of inputs and outputs.

We offer the following contributions. First, we define an abstract model of execution that is applicable to web client behavior (and any form of reactive computation that does not rely on preemptive multitasking). Second, by appealing to a stream-based semantics of these abstract systems, we give a generic definition of security for our execution model that is high-level in the manner of Goguen and Meseguer's MLS property; then we show that this generic definition has specific instantiations corresponding to both termination-sensitive and to termination-insensitive definitions of noninterference. Third, we offer an "unwinding lemma" [6] for our termination-insensitive version of security, which gives rise to a lower-level, transition-based definition of security. Finally, we use this unwinding lemma to demonstrate how a security type system can be used to enforce security in a simple language with a reactive semantics.

2. REACTIVE COMPUTATION

We use a constrained labeled transition system with input and output actions to capture our execution model. Although it bears similarity to labeled transition systems in the information-flow literature [14, 7, 4], the constraints are notably different.

2.1 Definition: A *reactive system* is a tuple

$$(\mathit{ConsumerState}, \mathit{ProducerState}, \mathit{Input}, \mathit{Output}, \rightarrow)$$

where $\rightarrow$ is a labeled transition system whose states are $\mathit{State} = \mathit{ConsumerState} \cup \mathit{ProducerState}$ and whose labels are $\mathit{Act} = \mathit{Input} \cup \mathit{Output}$, subject to the following constraints:

- for all $C \in \mathit{ConsumerState}$, if $C \xrightarrow{a} Q$, then $a \in \mathit{Input}$ and $Q \in \mathit{ProducerState}$,
- for all $P \in \mathit{ProducerState}$, if $P \xrightarrow{a} Q$, then $a \in \mathit{Output}$,
- for all $C \in \mathit{ConsumerState}$ and $i \in \mathit{Input}$, there exists a $P \in \mathit{ProducerState}$ such that $C \xrightarrow{i} P$, and

- for all $P \in \mathit{ProducerState}$, there exists an $o \in \mathit{Output}$ and $Q \in \mathit{State}$ such that $P \xrightarrow{o} Q$.

In words, a reactive system is one that takes the next available input, produces one or more outputs in response, and repeats the process.

Of course, a reactive system is inert unless it exists in an environment that can supply and receive messages. We may view an environment as comprising multiple agents communicating with the system over different channels by assuming that all inputs i and outputs o are tagged with some channel name c. In this scenario, the environment performs the service of multiplexing multiple streams of messages into a single, buffered stream. In modeling a web application, channel names for inputs would be used to represent both the addresses of remote servers (e.g., domain names) and unique references to the input controls available to the user of the web client; channel names for outputs would be used to model server addresses and references to updateable browser display components.

It is useful to make a few more observations about how our formal definition relates to the real systems we are trying to model. First, the definition implies that the system can always make some kind of progress unless it is blocking on input, but we note that this does not mean that it must always return to an input-accepting state: it can get into a loop producing outputs forever and never try to consume another input. Second, the definition requires every small step to produce an output. This is a technical device that does have any practical implications in our particular setting because, when we talk about security, we will assume that different outputs (and inputs) may be invisible to different observers, so we can easily model the act of a machine taking a silent, internal step using a system transition whose requisite output is invisible to all observers. This assumption conveniently allows us to assume, in our theoretical development, that all programs that diverge are continuously producing output, instead of having to make a special case for programs that diverge silently. We also force the system to go to a producer state after every input. Similarly, this does no harm in a setting where the system can simply produce a single invisible output in response to an input; however, it is a technical device that greatly simplifies our proofs because it entails that an infinite stream of inputs will always generate an infinite stream of outputs.

Now that we have a machine-like notion of reactive systems, we need a higher-level representation of their behavior to achieve a high-level definition of security. As mentioned earlier, we choose a stream-based interpretation instead of a trace-based interpretation. Formally, we define a stream as the coinductive[2] interpretation of the grammar

$$S ::= [] \mid s :: S$$

where s ranges over stream elements. That is, a stream is a finite or infinite list of elements. We use metavariables I and O to range over streams of inputs i and outputs o, respectively. Now we can view the behavior of a reactive system in a state Q as a relation between input streams and output streams.

[2]See Appendix A for technical background on this way of presenting coinductive definitions.

2.2 Definition: Coinductively define $Q(I) \Rightarrow O$ (Q translates the input stream I to the output stream O) with the following rules:

$$\frac{}{C([]) \Rightarrow []}$$

$$\frac{C \xrightarrow{i} P \qquad P(I) \Rightarrow O}{C(i :: I) \Rightarrow O} \qquad \frac{P \xrightarrow{o} Q \qquad Q(I) \Rightarrow O}{P(I) \Rightarrow o :: O}$$

We observe that this definition associates at least one output stream with every input stream, given the constraints on our transition systems.

In order to illustrate how a reactive system might be programmed, we now introduce the syntax of the simple language RIMP—a reactive version of the IMP language of basic while-programs. The full semantics are given in Section 5; here we rely on an intuitive explanation of RIMP's operational model. Input messages in RIMP are natural numbers tagged with their channels, where we let n range over the set of natural numbers, and ch range over a set of channels. Outputs are either a natural number sent over a channel or a "tick" (which will be the default output on an internal step).

$$\begin{array}{rcrcl} \mathit{Input} & \ni & i & ::= & ch(n) \\ \mathit{Output} & \ni & o & ::= & ch(n) \mid \bullet \end{array}$$

The syntax of programs, handlers, commands, and expressions is defined as follows:

$$\begin{array}{rcl} p & ::= & \cdot \mid h;\ p \\ h & ::= & ch(x)\{c\} \\ c & ::= & \texttt{skip} \mid c;\ c \mid \texttt{output}\ ch(e) \mid r := e \\ & \mid & \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{while}\ e\ \{c\} \\ e & ::= & x \mid n \mid r \mid e \odot e \\ \odot & ::= & + \mid - \mid = \mid < \end{array}$$

A program is a collection of event handlers, each of which accepts a message (a natural number) on some channel and runs a simple imperative program in response. The handler code may examine and modify shared global state, send messages, branch, and loop. When the handler for an input terminates, the RIMP program returns to a state in which it can handle another input. Handlers persist after handling events. Note that, since handlers share a global state, processing one input may affect the behavior of handlers in the future. The global state, called the *store*, is a mapping from variables r to natural numbers. We assume that every variable in the global state is initialized to 0 at the start. In the machine that runs RIMP programs, a consumer state consists of the program text and the shared global state, and a producer state additionally includes the command that is currently being executed. If a producer state takes a step that does not otherwise generate an output message, we assume the label on that transition is $\bullet$.

Of course, RIMP is a long way from a full-featured web scripting language. Our goal with RIMP is to model only the event-handling mechanism of web application programming. Moreover, for the sake of simplicity, there is no mechanism for dynamically adding or removing handlers, which is characteristic of web programming; however, we believe our work on RIMP in this paper can be extended to account for this scenario once first-class functions and dynamic allocation are added to the language (which have been studied before in the context of information-flow type systems [17]). We leave that for future work. Finally, it is worth noting that the integration of secure web scripts into web documents also requires careful consideration, which is another topic we must defer to our future work.

3. SECURITY OF REACTIVE SYSTEMS

As described earlier, reactive systems may send messages to and receive messages from multiple agents, which we will call *principals*. We assume there is a pre-order of security labels $(\mathcal{L}, \leq)$ and that all principals have a label corresponding to a level of authorization. We also assume that messages interchanged with the system have a label indicating their level of confidentiality. We intend to derive this level from the channel that carries the message, and this can be done at the point where the message streams are multiplexed. We assume that principals at a level l may only view messages at or below the level l. This is reasonable if we assume that observers would be positioned at the endpoints of HTTP connections and view the elements of $\mathcal{L}$ as based on domain names.

It is important to remember that, in the particular setting of web applications, it is the web browser user's personal secrets and the user's shared secrets with other principals that are being protected. (We are in no way addressing the issue of protecting a web server's secrets from web clients.) As noted before, there must be some means to associate channels with user interface components. Since these channels determine the assigned security level of the data, it is necessary to have a user interface design that allows users of web applications to view the precise security level of any interface component.[3] We assume that any user input control with a channel labeled by $\top$ (a maximal element of $\mathcal{L}$) can be used to handle information that should never leave the user's computer.

Now we can state an informal definition of information security: if a principal at one level cannot draw a distinction between two streams of inputs given to a reactive system starting in a particular state, then the same observer must not be able to draw a distinction between the resulting streams of outputs. This is a natural generalization of standard definitions of noninterference for imperative and functional languages [21, 17], and corresponds closely to Goguen and Meseguer's MLS property [6]. We can state this definition formally in the following way:

3.1 Definition: A state Q is *secure* if, for all l, $I \approx_l I'$ implies $O \approx_l O'$ whenever $Q(I) \Rightarrow O$ and $Q(I') \Rightarrow O'$.[4]

The notation $S \approx_l S'$ is meant to stand for a similarity relation on streams that is parametrized by a label l—in other words, the inability of an observer at level l to draw a distinction between S and S'. Defining this relation precisely is where things become interesting: it turns out that there are many natural notions of similarity between streams relative to an observer who cannot see all of the elements, leading us to multiple notions of security. Moreover, there are multiple ways to define each of these notions of similarity, and it is often difficult to guess which definitions are precisely equivalent. In the remainder of this section, we present a definition for four, increasingly-refined notions of similarity, and consider the technical implications for the corresponding definitions of security.

Preliminaries. To discuss these notions of security precisely, we need a few auxiliary definitions. To determine whether a stream element s is visible to an observer at level l, we use the predicate $\mathit{visible}_l(s)$. We assume that the set of security labels $\mathcal{L}$ has a top element, $\top$, with $\mathit{visible}_\top(s)$ for all s. In examples, we assume there are labels $\top$ and $\bot$ and channels $ch_\top$ and $ch_\bot$, such that messages on channel $ch_\top$ are invisible to an observer at level $\bot$.

We also need some auxiliary definitions about streams. We write $\mathit{fin}(S)$ when S is finite and $\mathit{inf}(S)$ when S is infinite. Next, we need a relation that associates a stream with its next l-visible element (if such an element exists) and with the remainder of the stream thereafter.

3.2 Definition: Inductively define $S \rhd_l s :: S'$ (S l-reveals s followed by S') with the following rules:

$$\frac{\mathit{visible}_l(s)}{s :: S \rhd_l s :: S} \qquad \frac{\neg\ \mathit{visible}_l(s) \qquad S \rhd_l s' :: S'}{s :: S \rhd_l s' :: S'}$$

This predicate is inductively defined because we only want it to hold true if one can find an l-visible element in a finite prefix of the potentially infinite stream. On the other hand, we would also like to define a predicate asserting that a stream contains no more l-visible elements.

3.3 Definition: Coinductively define $\mathit{silent}_l(S)$ with the following rules:

$$\frac{}{\mathit{silent}_l([])} \qquad \frac{\neg\ \mathit{visible}_l(s) \qquad \mathit{silent}_l(S)}{\mathit{silent}_l(s :: S)}$$

This definition is coinductive because it is asserting a fact about all of the elements of a potentially infinite stream.

Nonconflicting Security. The first two versions of similarity that we present are each defined by taking the *negation* of a definition of stream distinctness. The coarsest version of similarity, *nonconflicting similarity*, just requires that the observer cannot find two distinct stream elements in corresponding positions in the streams. Since a conflict must be evident from some finite prefixes of two streams, an inductive definition of this notion of distinctness is appropriate.

3.4 Definition: Inductively define $\mathit{conflicting}_l(S, S')$ with the following rules:

$$\frac{S \rhd_l s :: S_1 \qquad S' \rhd_l s' :: S'_1 \qquad s \neq s'}{\mathit{conflicting}_l(S, S')}$$

$$\frac{S \rhd_l s :: S_1 \qquad S' \rhd_l s :: S'_1 \qquad \mathit{conflicting}_l(S_1, S'_1)}{\mathit{conflicting}_l(S, S')}$$

3.5 Definition: Define $S \approx_l^{NC} S'$ (S is NC-similar to S' at l) to mean $\neg\ \mathit{conflicting}_l(S, S')$. Define *NC-security* as Definition 3.1, instantiated with NC-similarity.

[3] Although this raises questions about human interface design that are quite important in practice, it does not affect the fundamental theory of browser security that we are developing here.

[4] In this definition, we do not assume that the reactive system under consideration is deterministic, but it can be shown that this definition does put very stringent restrictions on the sorts of nondeterminism that are deemed secure. This definition suffices for our purposes here, but a more lenient, possibilistic notion of security would demand an equivalence on the *sets* of output streams that might be produced by equivalent input streams.

There are other ways of defining NC-similarity. It turns out that S is NC-similar to S' at l if the sequence of visible elements of one stream is a prefix of the visible elements of the other, which may be a more intuitive way to think about this relation. Nonconflicting similarity is reflexive and symmetric, but *not* transitive—we have $[] \approx_l^{NC} S$ for any l and S.

3.6 Example: The following program is not NC-secure:

```
  input $ch_\top(x)$ { output $ch_\bot(x)$ }
```

This event handler has an *explicit flow*, and it is deemed insecure because the streams $[ch_\top(0)]$ and $[ch_\top(1)]$ are NC-similar at $\bot$ but the corresponding output streams, $[ch_\bot(0), \bullet]$ and $[ch_\bot(1), \bullet]$, are not NC-similar at $\bot$.

3.7 Example: The following program is not NC-secure:

```
input $ch_\top(x)$ { $r$ := $x$ }
input $ch_\bot(x)$ { if $r = 0$ then output $ch_\bot(0)$
                     else output $ch_\bot(1)$ }
```

The second event handler has an *implicit flow*. It is deemed insecure because the input streams $[ch_\top(0), ch_\bot(0)]$ and $[ch_\top(1), ch_\bot(0)]$ are NC-similar at $\bot$ but the corresponding output streams, $[\bullet, \bullet, \bullet, ch_\bot(0), \bullet]$ and $[\bullet, \bullet, \bullet, ch_\bot(1), \bullet]$, are not NC-similar at $\bot$.

It may not be immediately clear which $\bullet$ outputs go with which inputs in the previous example, and the reader may wonder at this point whether our formalization of security has an inherent weakness because it handles the input and output streams separately rather than as one interleaved stream. In fact, this is a weakness of NC-similarity (but it will be resolved by stricter notions of similarity).

3.8 Example: The following program *is* NC-secure:

```
input $ch_\top(x)$ { $r$ := $x$ }
input $ch_\bot(x)$ { if $r = 0$ then output $ch_\bot(0)$
                     else (output $ch_\bot(0)$; output $ch_\bot(0)$) }
```

This example is almost the same as the previous example. However, this one will map the input streams $[ch_\top(0), ch_\bot(0)]$ and $[ch_\top(1), ch_\bot(0)]$ to the output streams $[\bullet, \bullet, \bullet, ch_\bot(0), \bullet]$ and $[\bullet, \bullet, \bullet, ch_\bot(0), \bullet, ch_\bot(0), \bullet]$, which are NC-similar at $\bot$. We can see that the program is NC-secure, in general, because the only outputs it can produce are $\bullet$ and $ch_\bot(0)$, and any two streams of these elements are NC-similar at $\bot$. In order to strengthen our notion of security to deal with the synchronization behavior of inputs and outputs, we need a more refined notion of similarity—one that coincides with the obvious definition on finite streams (i.e., dropping invisible items and comparing what remains for equality) when both streams are finite.

Indistinguishable Security. We modify the previous definition by adding two inference rules that effectively grant an observer the power to distinguish finite silent streams from streams that still have observable elements. We call this *indistinguishable similarity.*

3.9 Definition: Define $\mathit{distinguishable}_l(S, S')$ inductively with the following rules:

$$\frac{S \rhd_l s :: S_1 \qquad \mathit{silent}_l(S') \qquad \mathit{fin}(S')}{\mathit{distinguishable}_l(S, S')}$$

$$\frac{\mathit{silent}_l(S) \qquad \mathit{fin}(S) \qquad S' \rhd_l s :: S'_1}{\mathit{distinguishable}_l(S, S')}$$

$$\frac{S \rhd_l s :: S_1 \qquad S' \rhd_l s' :: S'_1 \qquad s \neq s'}{\mathit{distinguishable}_l(S, S')}$$

$$\frac{S \rhd_l s :: S_1 \qquad S' \rhd_l s :: S'_1 \qquad \mathit{distinguishable}_l(S_1, S'_1)}{\mathit{distinguishable}_l(S, S')}$$

3.10 Definition: Define $S \approx_l^{ID} S'$ (S is ID-similar to S' at l) to mean $\neg\ \mathit{distinguishable}_l(S, S')$. Define *ID-security* as Definition 3.1, instantiated with ID-similarity.

Note that we defined $\mathit{distinguishable}_l(S, S')$ exactly as one would inductively define distinctness of finite streams, so its behavior on finite streams is the obvious one that simply filters out invisible elements and tests the remaining lists for equality. It immediately renders Example 3.8 insecure because, in general, if the high inputs differ, the output streams will not be equal after dropping the $\bullet$ outputs. Although ID-similarity gives an equivalence relation on finite streams, it is not transitive, in general, because of its subtle behavior on infinite streams. Observe that, if $\mathit{inf}(S)$ and $\mathit{silent}_l(S)$, then $S \approx_l^{ID} S'$ for all l and S'. This observation leads us to our next example.

3.11 Example: The following program is ID-secure.

```
input $ch_\top(x)$ { $r$ := $x$ }
input $ch_\bot(x)$ { if $r = 0$ then output $ch_\bot(0)$
                     else while 1 do skip }
```

The second event handler creates a *termination channel.* Observe that the input streams $[ch_\top(0), ch_\bot(0)]$ and $[ch_\top(1), ch_\bot(0)]$ are ID-similar at $\bot$ and the corresponding output streams $[\bullet, \bullet, \bullet, ch_\bot(0), \bullet]$ and $[\bullet, \bullet, \bullet, \bullet, \ldots]$ are, in fact, also ID-similar at $\bot$. Thus, this is a termination-insensitive definition of security.

Standard definitions of noninterference [21, 17] usually imply some sort of functional dependency between the inputs and outputs of a program. The same is true here (and this fact is convenient for proving subsequent properties of our system).

3.12 Lemma: If a state Q is ID-secure, then for all I, $Q(I) \Rightarrow O$ and $Q(I) \Rightarrow O'$ implies $O = O'$.

To be precise, this does not mean a reactive system must be deterministic in order to be ID-secure: state transitions can be nondeterministic as long as they do not affect the output behavior.

It is straightforward to demonstrate a relationship between ID-similarity and NC-similarity.

3.13 Lemma: $S \approx_l^{ID} S'$ implies $S \approx_l^{NC} S'$.

More interesting is the fact that ID-security is stronger than NC-security. (This is not as straightforward to show because ID-similarity appears contravariantly in the definition of security.)

3.14 Lemma: If a transducer in a state Q is ID-secure, then it is NC-secure.

From a practical standpoint, we don't see any setting where NC-security is preferable to ID-security. We will see later that ID-security for RIMP programs can be guaranteed with a simple and flexible type system, and it is not clear how one would weaken the type system to include programs that are NC-secure but not ID-secure.

Coproductive Security. ID-security is termination-insensitive because it does not give the observer the power to distinguish non-silent output streams from silent but infinite ones. We can ensure that such streams are always considered distinct with a more direct, coinductive definition of similarity, called *coproductive similarity*, which can be viewed as a weak bisimulation between the two streams, in which invisible elements correspond to internal τ actions.

3.15 Definition: Coinductively define $S \approx_l^{CP} S'$ (S is CP-similar to S' at l) with the following rules:

$$\frac{\mathit{silent}_l(S) \qquad \mathit{silent}_l(S')}{S \approx_l^{CP} S'}$$

$$\frac{S \rhd_l s :: S_1 \qquad S' \rhd_l s :: S_1' \qquad S_1 \approx_l^{CP} S_1'}{S \approx_l^{CP} S'}$$

Define *CP-security* as Definition 3.1, instantiated with CP-similarity.

Unlike the earlier definitions of similarity, this one *is* an equivalence relation. It is easy to check that Example 3.11 is not CP-secure, using the same input and output pairs mentioned above. Although we use a coinductive definition here, it should be possible to draw a very close correspondence between this definition and the ones used recently for "interactive programs" [16, 9, 2].

The inductive definitions of NC-similarity and ID-similarity resemble one another, so it was easy to prove Lemma 3.13; on the other hand, proving the following lemma requires a bit more work.

3.16 Lemma: $S \approx_l^{CP} S'$ implies $S \approx_l^{ID} S'$.

What is the relationship between CP-security and ID-security, though? Again, since CP-similarity appears both co- and contravariantly in the definition of CP-security, their relationship is not at all obvious. The proof of the following lemma rests on several auxiliary definitions and lemmas, and additionally makes use of the bisimulation-based technique we introduce in Section 4.

3.17 Lemma: If a state Q is CP-secure, then it is ID-secure.

Coproductive-Coterminating Security. CP-security is quite strong, but it is possible to go a step further by defining similarity in such a way that finite and infinite silent streams can be distinguished (*coproductive-coterminating similarity*).

3.18 Definition: Coinductively define $S \approx_l^{CPCT} S'$ (S is CPCT-similar to S' at l) with the following rules:

$$\frac{\begin{array}{cc}\mathit{silent}_l(S) & \mathit{fin}(S)\\ \mathit{silent}_l(S') & \mathit{fin}(S')\end{array}}{S \approx_l^{CPCT} S'} \qquad \frac{\begin{array}{cc}\mathit{silent}_l(S) & \mathit{inf}(S)\\ \mathit{silent}_l(S') & \mathit{inf}(S')\end{array}}{S \approx_l^{CPCT} S'}$$

$$\frac{S \rhd_l s :: S_1 \qquad S' \rhd_l s :: S_1' \qquad S_1 \approx_l^{CPCT} S_1'}{S \approx_l^{CPCT} S'}$$

Here is an example of a program that is secure by every other definition thus far but is not CPCT-secure.

3.19 Example: The following program is not CPCT-secure:

```
input $ch_\top(x)$ { $r$ := $x$;
              if $x = 0$ then while 1 do skip }
```

This is the entire program. Low inputs are consumed but produce no low-visible output because there is no handler for them. (If this were not the case, then this program would fail to be CP-secure.)

The definitions of CP-similarity and CPCT-similarity aren't too different; so the following results shouldn't be too surprising, although the latter one is still not trivial.

3.20 Lemma: $S \approx_l^{CPCT} S'$ implies $S \approx_l^{CP} S'$.

3.21 Lemma: If a transducer in a state Q is CPCT-secure, then it is CP-secure.

CPCT-security guarantees that a reactive system can never make a choice between entering a input-accepting state or silently diverging based on a high input. However, this additional guarantee over CP-security is unimportant in practice because an attacker does not have the power to observe the results of such a choice in a CP-secure system. Consider a CP-secure machine that will silently diverge upon receiving a high input of 0 but will immediately return to a consumer state upon receiving a nonzero high input. A low observer who wishes to determine if the first high input was nonzero can only send a message to the machine and wait for a response (in our attack model, there is no other way to probe the system). A response would not be given to the low observer if the high input were 0; thus, CP-security guarantees that, even if the machine eventually consumes the low input, no response will be given to the low observer after (or even before) any high input. Since there is no possibility for getting feedback, there is no way for the low observer to determine if the system accepted the low input or whether the input is sitting in a buffer while the machine runs forever. Thus, CP-security is weaker than CPCT-security only on paper.

Summary. We have presented four definitions of security based on four definitions of similarity. Of these, two appear to be of practical interest: ID-security and CP-security. Enforcing CP-security through language-based techniques involves difficult trade-offs. For instance, O'Neill, Clarkson, and Chong [16] choose to disallow looping over high-level data, a very severe restriction. Instead, we choose to focus on termination-insensitive ID-security at this point. Although the type system we'll use to enforce this looks quite standard, we first need to break down the definition of ID-security from a property on the input/output behavior of a system to a property on the states of a reactive system.

4. PROVING ID-SECURITY

We now present a generic technique for proving the ID-security of a state in a reactive system. This is an "unwinding lemma" in the sense of Goguen and Meseguer [6]: it is a logically sufficient condition on the states of a transition system to ensure a high-level property of the system's input/output behavior. One can alternatively view it as a bisimulation technique, given that it involves a binary relation that facilitates the coinductive proof of the unwinding lemma.

4.1 Definition: An *ID-bisimulation* on a reactive system is a label-indexed family of binary relations on states (written $\sim_l$) with the following properties:

(a) if $Q \sim_l Q'$, then $Q' \sim_l Q$;

(b) if $C \sim_l C'$ and $C \xrightarrow{i} P$ and $C' \xrightarrow{i} P'$, then $P \sim_l P'$;

(c) if $C \sim_l C'$ and $\neg\ \mathit{visible}_l(i)$ and $C \xrightarrow{i} P$, then $P \sim_l C'$;

(d) if $P \sim_l C$ and $P \xrightarrow{o} Q$, then $\neg\ \mathit{visible}_l(o)$ and $Q \sim_l C$;

(e) if $P \sim_l P'$, then either

- $P \xrightarrow{o} Q$ and $P' \xrightarrow{o'} Q'$ implies $o = o'$ and $Q \sim_l Q'$, or else
- $P \xrightarrow{o} Q$ implies $\neg\ \mathit{visible}_l(o)$ and $Q \sim_l P'$, or else
- $P' \xrightarrow{o'} Q'$ implies $\neg\ \mathit{visible}_l(o')$ and $P \sim_l Q'$.

We will see below that, if $Q \sim_l Q$ for all l, then Q is ID-secure. We do not use a standard form of bisimulation (as is done in [4]) because we need a technique that gives rise to a termination-insensitive security property. Note that, in the first of the three cases under item (e), if one side can make a step with an output o, then *all* steps taken by the other side must produce the same output o. On the other hand, the other two cases under item (e) permit one side to take a silent step without being matched by the other side, which allows one side to get infinitely far ahead of the other when this definition is used coinductively.

Before we can prove that this definition gives us the property we want, we need to introduce one more definition of similarity between streams.

4.2 Definition: Coinductively define $S \approx_l^{VS} S'$ (S is visibly l-similar to S') with the following rules:

$$\frac{}{[] \approx_l^{VS} []} \qquad \frac{\mathit{visible}_l(s) \qquad S \approx_l^{VS} S'}{s :: S \approx_l^{VS} s :: S'}$$

$$\frac{\neg\ \mathit{visible}_l(s) \qquad S \approx_l^{VS} S'}{s :: S \approx_l^{VS} S'} \qquad \frac{\neg\ \mathit{visible}_l(s) \qquad S \approx_l^{VS} S'}{S \approx_l^{VS} s :: S'}$$

Observe that this is a natural relation to define between two streams with invisible elements. It is easy to write down because it does not depend on auxiliary definitions such as l-reveals. Does this relation give rise to yet another notion of similarity and security? No, in fact, it coincides exactly with ID-similarity.

4.3 Lemma: $S \approx_l^{VS} S'$ iff $S \approx_l^{ID} S'$.

Visible similarity is an important technical tool in our development since it gives us a coinduction principle that can be used to prove the following key lemma.

4.4 Lemma: Suppose that $Q \sim_l Q'$, and that $Q(I) \Rightarrow O$ and $Q'(I') \Rightarrow O'$. Then $I \approx_l^{VS} I'$ implies $O \approx_l^{VS} O'$.

The previous two lemmas lead us directly to our goal.

4.5 Theorem: If $Q \sim_l Q$ for all l, then Q is ID-secure.

5. RIMP

Rather than using our technique from Section 4 to prove programs secure one at a time, we would like to demonstrate that we can use a type system to show that all of the well-typed programs in a language are secure, in line with the previous work on language-based security [19]. To this end, we complete our technical development with a formal presentation of the RIMP language, along with a static type system that will ensure that well-typed programs are secure. We will prove this result by defining a relation on program states and showing that it is an ID-bisimulation for which well-typed programs are related to themselves.

Operational Semantics. We first define consumer and producer states of the RIMP reactive system. A consumer state, C, is a store paired with a program. A producer state, P, also includes the currently executing command and is tagged by the channel that triggered the execution. Stores, μ, map global variables to the natural numbers they contain.

$$\begin{array}{rcl} C & ::= & (\mu, p) \\ P & ::= & (\mu, p, c)^{ch} \end{array}$$

The transition between states in the RIMP reactive system is defined by the following four judgments of the operational semantics, whose definitions appear below.

1. $\mu \vdash e \Downarrow n$, a big step evaluation of closed expressions to numeric values, using the store to look up variables. (This definition is an entirely straightforward one, in which we use 0 for the boolean value false and nonzero numbers for true. See Appendix B for the formal definition.)

2. $(\mu, c) \xrightarrow{o} (\mu', c')$, a small step execution of a closed command paired with a store, where each step produces an output.

3. $(p)(i) \Downarrow c$, the response to an input event, producing the command that will execute next.

4. $Q \xrightarrow{a} Q'$, the actual transitions of the reactive system.

The bulk of computation occurs when the commands in a handler are executed. Each step of computation produces an output, o, although many of those outputs will be the trivial output $\bullet$, which is visible only to the highest-security observer. The rules below are standard except for the final rule, which produces output.

5.1 Definition: Inductively define $(\mu, c) \xrightarrow{o} (\mu', c')$ with the following rules:

$$\frac{}{(\mu, (\texttt{skip};\ c)) \xrightarrow{\bullet} (\mu, c)} \qquad \frac{(\mu, c_1) \xrightarrow{o} (\mu', c_1')}{(\mu, (c_1;\ c_2)) \xrightarrow{o} (\mu', (c_1';\ c_2))}$$

$$\frac{\mu \vdash e \Downarrow n}{(\mu, (r := e)) \xrightarrow{\bullet} (\mu[r \mapsto n], \texttt{skip})}$$

$$\frac{\mu \vdash e \Downarrow n \qquad n \neq 0}{(\mu, (\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2)) \xrightarrow{\bullet} (\mu, c_1)}$$

$$\frac{\mu \vdash e \Downarrow 0}{(\mu, (\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2)) \xrightarrow{\bullet} (\mu, c_2)}$$

$$\frac{\mu \vdash e \Downarrow n \qquad n \neq 0}{(\mu, (\texttt{while } e \ \{c\})) \xrightarrow{\bullet} (\mu, (c;\ \texttt{while } e \ \{c\}))}$$

$$\frac{\mu \vdash e \Downarrow 0}{(\mu, (\texttt{while } e \ \{c\})) \xrightarrow{\bullet} (\mu, \texttt{skip})}$$

$$\frac{\mu \vdash e \Downarrow n}{(\mu, (\texttt{output } ch(e))) \xrightarrow{ch(n)} (\mu, \texttt{skip})}$$

Next, we need a definition that pairs an input with a program and builds the code that will be executed in response to that event. This will require a substitution of the message data for the parameter x in the body of the event handler. We assume a standard definition of substituting a value n for x in an expression e (written $e\{n/x\}$), extended to commands in the obvious way.

5.2 Definition: Inductively define $(p)(i) \Downarrow c$ with the following rules:

$$\frac{}{(ch(x)\{c\};\ p)(ch(n)) \Downarrow c\{n/x\}}$$

$$\frac{(p)(ch'(n)) \Downarrow c \qquad ch \neq ch'}{(ch(x)\{c\};\ p)(ch'(n)) \Downarrow c} \qquad \frac{}{(\cdot)(i) \Downarrow \texttt{skip}}$$

Finally, we give the labeled transition system corresponding to RIMP's semantics. This system either transitions a consumer state to a producer state by looking up the appropriate handler, steps a producer state to a new producer state (if there is computation remaining), or steps a producer state to a consumer state (if the handler has finished execution).

5.3 Definition: Define $Q \xrightarrow{a} Q'$ (where $a ::= i \mid o$) with the following rules:

$$\frac{(p)(ch(n)) \Downarrow c}{(\mu, p) \xrightarrow{ch(n)} (\mu, p, c)^{ch}}$$

$$\frac{(\mu, c) \xrightarrow{o} (\mu', c')}{(\mu, p, c)^{ch} \xrightarrow{o} (\mu', p, c')^{ch}} \qquad \frac{}{(\mu, p, \texttt{skip})^{ch} \xrightarrow{\bullet} (\mu, p)}$$

We can easily show that these rules define a reactive system, which is really just a matter of confirming that the RIMP execution will never halt if inputs are available.

Typing of RIMP Programs. Now we give a static type system to the RIMP language, whose purpose is to identify a subset of programs that are secure.

We assume there is a function *lbl* that associates a label with every channel and program variable, and we define $\mathit{visible}_l(ch(n))$ to mean that $\mathit{lbl}(ch) \leq l$, for both inputs and outputs. Define $\mathit{visible}_l(\bullet)$ to hold iff $l = \top$.

Expressions are typed with a single label, which can be interpreted as an upper bound on the secrecy level of the components of the expression. The typing judgment for expressions is parametrized by a mapping Γ from parameters to labels. (Even though we use only one formal parameter x in this language, we write it this way for consistency of notation with standard typing judgments in more expressive languages.)

5.4 Definition: Inductively define $\Gamma \vdash e : l$ with the following rules:

$$\frac{\Gamma(x) \leq l}{\Gamma \vdash x : l} \qquad \frac{}{\Gamma \vdash n : l} \qquad \frac{\mathit{lbl}(r) \leq l}{\Gamma \vdash r : l}$$

$$\frac{\Gamma \vdash e_1 : l_1 \qquad \Gamma \vdash e_2 : l_2 \qquad l_1, l_2 \leq l}{\Gamma \vdash e_1 \odot e_2 : l}$$

Commands are also typed with a single label, which can be interpreted as a lower bound on the secrecy of the effects that could occur during the execution of the command. Traditionally, this label is called the label of the "program counter," so we use *pc* to range over it. Again, we need a typing environment Γ for the parameters that might be present in commands.

5.5 Definition: Inductively define $\Gamma \vdash c : pc$ with the following rules:

$$\frac{}{\Gamma \vdash \texttt{skip} : pc}$$

$$\frac{\Gamma \vdash c_1 : pc_1 \qquad \Gamma \vdash c_2 : pc_2 \qquad pc \leq pc_1, pc_2}{\Gamma \vdash (c_1;\ c_2) : pc}$$

$$\frac{\Gamma \vdash e : l \qquad l \leq \mathit{lbl}(ch) \qquad pc \leq \mathit{lbl}(ch)}{\Gamma \vdash \texttt{output } ch(e) : pc}$$

$$\frac{\Gamma \vdash e : l \qquad l \leq \mathit{lbl}(r) \qquad pc \leq \mathit{lbl}(r)}{\Gamma \vdash (r := e) : pc}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : l \qquad \Gamma \vdash c_1 : pc_1 \qquad \Gamma \vdash c_2 : pc_2 \\ l \leq pc_1, pc_2 \qquad pc \leq pc_1, pc_2\end{array}}{\Gamma \vdash \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 : pc}$$

$$\frac{\Gamma \vdash e : l \qquad \Gamma \vdash c : pc_1 \qquad l \leq pc_1 \qquad pc \leq pc_1}{\Gamma \vdash \texttt{while } e \ \{c\} : pc}$$

The typing judgment for programs simply requires that each handler be well typed at the level of its channel, under the assumption that the message received is secret at the level of the channel.

5.6 Definition: Inductively define $\vdash p$ with the following rules:

$$\frac{}{\vdash \cdot} \qquad \frac{x : \mathit{lbl}(ch) \vdash c : \mathit{lbl}(ch) \qquad \vdash p}{\vdash ch(x)\{c\};\ p}$$

Finally, we may define a typing judgment for producer and consumer states. Note that typing programs does not depend on the store. The channel that triggered a producer state also constrains the type of the command in that state.

5.7 Definition: Define the judgment $\vdash Q$ with the following rules:

$$\frac{\vdash p}{\vdash (\mu, p)} \qquad \frac{\vdash p \qquad \vdash c : \mathit{lbl}(ch)}{\vdash (\mu, p, c)^{ch}}$$

These definitions have the standard type preservation property.

5.8 Lemma: If $\vdash Q$ and $Q \xrightarrow{a} Q'$, then $\vdash Q'$.

The standard progress theorem for well-typed terms is actually trivial here because by definition *every* term can make progress in a reactive system.

Bisimulation on RIMP Programs. We now turn to defining a label-indexed family of binary relations on program states and showing that it is a ID-bisimulation. This relation is built from relations on stores, commands, and programs.

First, two stores are related at label l if the contents visible to l are identical. This relation is an equivalence relation.

5.9 Definition: Define two stores μ and μ' to be related at l (written $\mu \sim_l \mu'$) if, for all r for which $\mathit{lbl}(r) \leq l$, we have $\mu(r) = \mu'(r)$.

Next, to define when two commands are related, we must first define a predicate $\mathit{high}_L(c)$ stating that the effects of a command are visible only within a certain upward-closed set L. In the following, we define the *downward closure* of a set of labels L (written $L^{\blacktriangledown}$) as $\{l \mid \exists l' \in L.\, l \leq l'\}$. Similarly, the *upward closure* of a set of labels L (written $L^{\blacktriangle}$) is $\{l \mid \exists l' \in L.\, l' \leq l\}$. (We write $l^{\blacktriangledown}$ and $l^{\blacktriangle}$ for $\{l\}^{\blacktriangledown}$ and $\{l\}^{\blacktriangle}$.) $\overline{L}$ is the complement of L.

5.10 Definition: Inductively define $\mathit{high}_L(c)$ with the following rules:

$$\frac{L \text{ is upward-closed}}{\mathit{high}_L(\texttt{skip})} \qquad \frac{\mathit{high}_{L_1}(c_1) \qquad \mathit{high}_{L_2}(c_2)}{\mathit{high}_{L_1 \cup L_2}(c_1;\ c_2)}$$

$$\frac{\mathit{lbl}(ch) \in L \qquad L \text{ is upward-closed}}{\mathit{high}_L(\texttt{output}\ ch(e))}$$

$$\frac{\mathit{lbl}(r) \in L \qquad L \text{ is upward-closed}}{\mathit{high}_L(r := e)}$$

$$\frac{\mathit{high}_{L_1}(c_1) \qquad \mathit{high}_{L_2}(c_2)}{\mathit{high}_{L_1 \cup L_2}(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)}$$

$$\frac{\mathit{high}_L(c)}{\mathit{high}_L(\texttt{while}\ e\ \{c\})}$$

Now we can define when two commands are related at a label. Intuitively, the commands must be identical, except for subcommands whose effects are invisible to an observer at level l.

5.11 Definition: Inductively define $c \sim_l c'$ as follows:

$$\frac{}{\texttt{skip} \sim_l \texttt{skip}} \qquad \frac{c_1 \sim_l c_1' \qquad c_2 \sim_l c_2'}{(c_1;\ c_2) \sim_l (c_1';\ c_2')}$$

$$\frac{\vdash e : l' \qquad l' \leq \mathit{lbl}(ch) \leq l}{\texttt{output}\ ch(e) \sim_l \texttt{output}\ ch(e)}$$

$$\frac{\vdash e : l' \qquad l' \leq \mathit{lbl}(r) \leq l}{(r := e) \sim_l (r := e)}$$

$$\frac{\begin{array}{cc}\vdash e : l' & l' \leq l \\ c_1 \sim_l c_1' & c_2 \sim_l c_2'\end{array}}{\begin{array}{c}\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \sim_l \\ \texttt{if}\ e\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'\end{array}}$$

$$\frac{\vdash e : l' \qquad l' \leq l \qquad c \sim_l c'}{\texttt{while}\ e\ \{c\} \sim_l \texttt{while}\ e\ \{c\}}$$

$$\frac{\mathit{high}_L(c) \qquad \mathit{high}_L(c') \qquad l \notin L}{c \sim_l c'}$$

(This relation is symmetric and transitive; however, it is not reflexive for untypeable commands. For example, consider $c = \texttt{output}\ ch(r)$ where $\mathit{lbl}(r) \not\leq \mathit{lbl}(ch)$.)

Next we define when two programs are related. As for commands, this is a partial equivalence relation.

5.12 Definition: Two programs p and p' are related at l (written $p \sim_l p'$) if

- for all ch for which $\mathit{lbl}(ch) \leq l$, if $(p)(ch(n)) \Downarrow c$ and $(p')(ch(n)) \Downarrow c'$, then $c \sim_l c'$, and
- for all ch for which $\mathit{lbl}(ch) \not\leq l$, if $(p)(ch(n)) \Downarrow c$, then $\mathit{high}_{\overline{l^{\blacktriangledown}}}(c)$, and
- for all ch for which $\mathit{lbl}(ch) \not\leq l$, if $(p')(ch(n)) \Downarrow c$, then $\mathit{high}_{\overline{l^{\blacktriangledown}}}(c)$.

Finally, we define when two program states are related. A consumer state is related to a producer state only when the outputs of the command in the producer state are invisible and the stores and programs are related. This relation is also a partial equivalence relation.

5.13 Definition: Two states Q and Q' are related at l (written $Q \sim_l Q'$) with the following inductive definition:

$$\frac{\mu \sim_l \mu' \qquad p \sim_l p'}{(\mu, p) \sim_l (\mu', p')}$$

$$\frac{\mu \sim_l \mu' \qquad p \sim_l p' \qquad \mathit{high}_{\overline{l^{\blacktriangledown}}}(c)}{(\mu, p) \sim_l (\mu', p', c)^{ch}}$$

$$\frac{\mu \sim_l \mu' \qquad p \sim_l p' \qquad \mathit{high}_{\overline{l^{\blacktriangledown}}}(c)}{(\mu, p, c)^{ch} \sim_l (\mu', p')}$$

$$\frac{\mu \sim_l \mu' \qquad p \sim_l p' \qquad c \sim_l c'}{(\mu, p, c)^{ch} \sim_l (\mu', p', c')^{ch}}$$

Security of RIMP Programs. Now that we have defined a label-indexed family of relations on program states, we need to show that it is an ID-bisimulation.

A key lemma is that high commands step to high commands and only produce invisible outputs.

5.14 Lemma: If $\mathit{high}_L(c)$ and $(\mu, c) \xrightarrow{o} (\mu', c')$, then $\mathit{high}_L(c')$ and, for all $l \notin L$, we have $\neg\ \mathit{visible}_l(o)$ and $\mu \sim_l \mu'$.

We use this lemma to verify the conditions of Definition 4.1 to show that $\sim_l$ is an ID-bisimulation. Since we carefully constructed our binary relation, we can also show that programs are related to themselves at every label l if they are well typed.

5.15 Lemma: If $\vdash p$, then $p \sim_l p$ for all l.

Combining the previous lemma with Theorem 4.5 gives us the security result we claimed. This result guarantees us that any well-typed program will be secure when it is initialized with any store.

5.16 Theorem: If $\vdash p$, then (μ, p) is an ID-secure transducer.

6. ADDITIONAL RELATED WORK

The introduction has already drawn comparisons with the lines of work most directly relevant to ours. Here, we survey some more distantly related work that may also be of interest.

McCullough's "restrictiveness" property [14] is a security policy for labeled transition systems with input and output. Rather than defining a "high-level policy" on traces or streams, he defines restrictiveness with a set of constraints on transitions. It is therefore comparable with Goguen and Meseguer's "unwound policy" [6] or our ID-bisimulation. McCullough's restrictiveness property requires input totality in a strict sense (without an allowance for input buffering) and is termination-sensitive.

The work of O'Neill, Clarkson, and Chong [16] (hereafter "OCC") builds upon Halpern and O'Neill's Multiagent Systems Framework [8]. They define noninterference in terms of user *strategies*, which are functions that map every history of l-visible events to the next action for each principal at level l. This framework allows their security definitions to consider the possibility of a high user revealing information to a low user indirectly via choice of strategy. This seems to be of little practical value in a setting where high users have more direct means to interact with low users than through the system in question. For instance, in the setting of the web client, if a banking application wants to reveal the user's account number to a third party, this can be done trivially on the server's side of the application rather than through the code running in the user's browser. Moreover, Clark and Hunt [2] demonstrate that the choice of strategy does not permit covert communication in deterministic programs that are interactive in the sense of OCC. Although our execution model is different, the same fact may well hold true in our setting. (Much of the focus in OCC is actually on dealing with probabilistic behaviors; since RIMP is deterministic, this aspect is orthogonal to our aims.)

As mentioned earlier, Hunt and Sands [9] define security of their interactive programs in terms of infinite input streams, but don't use streams for describing their outputs. On the other hand, Askarov, Hunt, Sabelfeld, and Sands [1] explicitly consider potentially infinite sequences of outputs in their assessment of the weaknesses of termination-insensitive security, but do not deal with any kind of input. Matos, et. al. [13] give a type system and a proof of information-flow security for a notion of "reactive programs" rather different from ours. Their programs run (deterministically) to completion without consuming any intermediate input or producing any intermediate output.

7. CONCLUSIONS AND FUTURE WORK

We have proposed a formal definition of information-flow security for programs driven by event handling and have shown that language-based enforcement is a viable means to guarantee this property. It is now natural to ask how well this maps onto the real world of web programming.

The first issue is whether our definitions of security correctly capture the sort of web browsing confidentiality policies we desire in practice. Of course, such definitions cannot guarantee confidentiality in an absolute sense, since there are covert channels, such as timing channels, that are outside of our model. Assessing these channels and taking steps to mitigate their effects is an important part of any real-world security implementation. At the same time, like other definitions of "pure noninterference," our definitions are too restrictive, in that they rule out programs that must release (parts of) secret information to properly function: consider a mashup that must reveal a private street address to Google in the course of locating it with a Google Maps component. Finding appropriate mechanisms for declassification is an important direction for future work.

Another question is whether our fundamental model of interaction is flexible enough to account for real-world web behavior. Real network messages may be structured and may include different substructures with different security levels. For the purpose of a noninterference analysis, one could easily model such a message as a sequence of messages at different levels. However, a naive labeling of web page structures and a strict adherence to reactive noninterference can lead to some surprising results. For instance, if an entire HTTP response containing a web page is given a non-public security label, then it would not be secure for the browser to load any images on that page from servers with incomparable security labels: there would be no direct flow, but a system input (the HTTP response in this case) at one security level must only cause system outputs at the same level or higher. One easy workaround for this scenario is to give the initial HTTP response a public label and to use a private label only for the body of the message.

The connection between our model and the user interface is also very important, and the design space is complex. Users need to be able to understand the security interpretation that the browser assigns to each event they generate; otherwise they have no way of understanding the model's guarantees about the confidentiality of their input and browsing actions. In particular, users need to understand the precise form of the "pseudo-messages" corresponding to their actions. The message content corresponding to the action of entering text in a text box is reasonably clear. However, the action of clicking on an HTML link is much more subtle: if it is interpreted as a message from the user to the browser that contains the entire URL of the link destination, then reactive noninterference actually puts the burden on the user for verifying that the URL does not contain any encoding of any piece of secret data. Obviously, the user cannot do this without assistance from the browser, but it must be noted that a reactive noninterference policy says nothing about the correctness of that assistance. In addition to the contents of these user-generated messages, their security levels must also be determined. This could be controlled with a global "secrecy mode" setting for the whole browser, a per-window secrecy mode, a per-DOM-element secrecy mode, a per-action dialog box, or some combination of these. Moreover, in principle, it is possible that some of these modes might cause a single user action, such as a button press, to be viewed as a sequence of messages with different security levels (this would be useful for the same reason that it might be useful to view an HTTP response as a sequence of messages, as described above). A complex security interface will be hard to understand; an overly simple one may not provide enough flexibility to support web pages that interact with multiple remote sites or may not provide as much confidentiality as the user would like. There seem to be fundamental tradeoffs between flexibility, complexity, and security in this design space.

Since our model rules out preemptive multitasking, one may wonder whether it can account for timer events, which are common in web programming. The information security of timer events can be understood by modeling them as AJAX requests to a remote server that sends back a response after a fixed amount of time. Of course, concerns about covert timing channels must still be handled separately, and timing channels that exploit timer events may have a much higher bandwidth than covert channels based solely upon the timing of real network messages.

An entirely different question is whether language-based security is the best mechanism for enforcing our noninterference properties in the setting of web browsers. Although its event handling follows the same basic model as JavaScript, RIMP is a long way from a web scripting language. First, one would want to add some of the key features of JavaScript, such as first class functions, the ability to dynamically add and remove handlers, and `eval`. Second, one would need to design a security-aware version of the DOM interface for this language to use. Finally, one would have to implement a method for type-checking and running secure scripts in a manner that is reasonably backwards-compatible with existing web pages and scripts. All of these are important topics for future research.

Acknowledgments

Damien Pous made a big contribution to the early discussions that eventually led to this paper. We gratefully acknowledge support from the National Science Foundation under grant number 0715936, *Manifest Security*.

8. REFERENCES

[1] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *In Proceedings of the 13th European Symposium on Research in Computer Security*, pages 333–348, Malaga, Spain, Oct. 2008.

[2] D. Clark and S. Hunt. Non-interference for deterministic interactive programs. In *Formal Aspects of Security and Trust (FAST) '08*, 2008.

[3] Coq Development Team. *The Coq Proof Assistant Reference Manual v8.1.* http://coq.inria.fr/.

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

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

[6] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *In Proceedings of the IEEE Symposium on Security and Privacy*, 1984.

[7] I. Gray, J.W. Probabilistic interference. pages 170–179, May 1990.

[8] J. Y. Halpern and K. R. O'Neill. Secrecy in multiagent systems. *ACM Transactions on Information and Systems Security*, 12(1):1–47, 2008.

[9] S. Hunt and D. Sands. Just forget it—the semantics and enforcement of information erasure. In *In Proceedings of the 17th European Symposium on Programming (ESOP'08)*. Springer-Verlag (LNCS), 2008.

[10] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 737–744, New York, NY, USA, 2006. ACM.

[11] C. Jackson and H. J. Wang. Subspace: Secure cross-domain communication for web mashups. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, 2007.

[12] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.

[13] A. A. Matos, G. Boudol, and I. Castellani. Typing noninterference for reactive programs. In *In Proceeding of the Workshop on Foundations of Computer Security*, 2004.

[14] D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symposium on Security and Privacy*, pages 177–186. IEEE Computer Society Press, May 1988.

[15] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. A Google research project., Jan. 2008.

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

[17] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, 2003.

[18] C. Reis, S. D. Gribble, and H. M. Levy. Architectural principles for safe web programs. Presented at the Sixth Workshop on Hot Topics in Networks (HotNets-VI), Nov. 2007.

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

[20] Same origin policy for JavaScript. http://www.mozilla.org/projects/security/components/same-origin.html.

[21] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

[22] A. Zakinthinos and E. S. Lee. A general theory of security properties. In *In Proceedings of the IEEE Symposium on Security and Privacy*, pages 94–102, Washington, DC, USA, 1997. IEEE.

APPENDIX

A. COINDUCTIVE DEFINITIONS

We have used the Coq proof assistant [3] to guide our intuition about coinduction and to check many of our definitions and proofs. Following Coq's type-theoretic notion of coinduction, we take coinductive definitions as a primitive notion. We view our inference rules, both inductive and coinductive, as definitions of logical propositions, although they have an obvious translation to a set-theoretic definition of mathematical relations.

A coinductive definition can be understood as taking the greatest fixed-point interpretation of a grammar or a set of inference rules. In the case of a grammar, a coinductive definition describes the set of all finite or infinite objects that can be built with repeated applications of the term constructors (instead of just the finite objects). In the case of a proposition defined by a set of inference rules, a coinductive definition means that we allow the proposition to be proved with a finite or infinite derivation. This is often (and only) necessary when defining predicates on infinite data. Note that it is also perfectly reasonable to use an inductively defined proposition over coinductively defined data, which will mean that the truth of the proposition can only depend on a finite portion of the potentially infinite data. Inductively defined propositions give rise to a principle of induction, which can be used to prove a statement in which such a proposition appears as a hypothesis. On the other hand, coinductive definitions give rise a principle of coinduction, which can be used to prove a statement in which such a proposition appears as a *conclusion*.

For further background on inductive and coinductive reasoning, see the tutorial by Jacobs and Rutten [12].

B. EVALUATION OF RIMP EXPRESSIONS

B.1 Definition: Inductively define $\mu \vdash e \Downarrow n$ with the following rules:

$$\frac{}{\mu \vdash n \Downarrow n}$$

$$\frac{}{\mu \vdash r \Downarrow \mu(r)}$$

$$\frac{\mu \vdash e_1 \Downarrow n_1 \qquad \mu \vdash e_2 \Downarrow n_2 \qquad n = n_1 + n_2}{\mu \vdash e_1 + e_2 \Downarrow n}$$

$$\frac{\mu \vdash e_1 \Downarrow n_1 \qquad \mu \vdash e_2 \Downarrow n_2 \qquad n = n_1 - n_2}{\mu \vdash e_1 - e_2 \Downarrow n}$$

$$\frac{\mu \vdash e_1 \Downarrow n_1 \qquad \mu \vdash e_2 \Downarrow n_2 \qquad n_1 = n_2}{\mu \vdash e_1 = e_2 \Downarrow 1}$$

$$\frac{\mu \vdash e_1 \Downarrow n_1 \qquad \mu \vdash e_2 \Downarrow n_2 \qquad n_1 \neq n_2}{\mu \vdash e_1 = e_2 \Downarrow 0}$$

$$\frac{\mu \vdash e_1 \Downarrow n_1 \qquad \mu \vdash e_2 \Downarrow n_2 \qquad n_1 < n_2}{\mu \vdash e_1 < e_2 \Downarrow 1}$$

$$\frac{\mu \vdash e_1 \Downarrow n_1 \qquad \mu \vdash e_2 \Downarrow n_2 \qquad n_1 \not< n_2}{\mu \vdash e_1 < e_2 \Downarrow 0}$$