Reactive Noninterference

Aaron Bohannon Benjamin C. Pierce Vilhelm Sjöberg
Stephanie Weirich Steve Zdancewic

University of Pennsylvania

Abstract

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

In this paper, we explore various definitions of noninterference for reactive programs and identify two of special interest—one corresponding to termination-insensitive noninterference for a standard 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.

1 Introduction

Reactive programs—programs that alternate between computing and interacting with one or more external agents—are ubiquitous. Any program that computes in response to events (the clicking of a GUI button, the issuing of a command at a terminal, the receipt of a network packet, the expiration of a timer, *etc.*) is reactive. The ability for multiple agents to interact with a single reactive program immediately raises questions of security and privacy.

Web browsers and the client-side web applications they host are prototypical cases of reactive programs. Because they interact with both a local user and remote, possibly untrusted, agents (e.g., web servers), it is not surprising that significant effort has been spent on making them secure [19, 15, 6, 7]. Despite significant progress on improving browser security, we still rely on ad-hoc mechanisms such as the "same-origin policy"[1] to protect the integrity and confidentiality of data processed by web applications. The fundamental problem is that we lack the tools (both theoretical and practical) to answer the question "What does it mean for a browser running a web application to be secure?"

This paper focuses on the issue of confidentiality—in particular, what it means to enforce information-flow properties in a reactive system like a web browser. Moreover, we are interested in using language-based techniques for enforcing such properties in reactive programs, with the eventual goal of applying similar techniques to JavaScript or other web scripting languages with comparable event-handling capabilities. To this end, we define a generic semantics of reactive computations.

Within the context of this computational model, we make the following contributions. First, we frame the question of reactive security so that we can give definitions of noninterference paralleling the standard definitions of noninterference for sequential languages [21, 18]. Second, we explore this space of possible definitions and find that, from a practical standpoint, there are two useful definitions: one corresponding to termination-sensitive security in sequential languages and one to termination-insensitive security. Third, we establish a bisimulation-based technique for proving the termination-insensitive definition of security. And fourth, we illustrate this technique by using it to prove the information security of a reactive extension of IMP, the familiar language of imperative while-programs.

There has, of course, been a great deal of prior work on using language-based techniques for enforcing information-flow properties [20]. In particular, a model of reactive programs was studied by Goguen and Meseguer [4], but it is not a good fit when program-

[1] `http://www.mozilla.org/projects/security/components/same-origin.html`

ming with possibly diverging languages, nor for ones that produce output incrementally. Focardi and Gorrieri [2] study security definitions for CCS, which is certainly reactive, but their definitions are designed specifically with a view toward nondeterministic behavior, while we are interested in the security of systems whose real-world behavior is predictable. O'Neill, Clarkson, and Chong [16], address "interactive" programs, which can be viewed as a subset of the reactive programs we have in mind; their notion of security corresponds to a termination sensitive definition of information-flow, whereas we focus here on a termination insensitive variant. More detailed comparisons and other related work are discussed in Section 6.

2 Reactive Computation

Our model of reactive computation is inspired by JavaScript[2], the de facto standard for implementing client-side components of web applications. JavaScript code consists of handlers that wait for specific events. In the JavaScript execution model, when an event triggers a handler, the code of the handler executes to completion without any form of preemptive multitasking. JavaScript's core functionality is simple: it may update the state of its internal environment; send messages on the network (e.g., using `XMLHttpRequest`); install new handlers to wait for new events, including responses to its network requests; or update the DOM tree of the HTML page being displayed.

Because JavaScript programs can produce many outputs after handling a single input—in fact, they may not terminate but just go on producing outputs forever—it is natural to model their behavior as an automaton that takes small, bounded steps:

2.1 Definition: A *reactive system* mapping *Input* to *Output* 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 $P \xrightarrow{o} Q$.

A few observations about this definition are in order. First, the definition implies that the system can always make some kind of progress unless it is blocking on input. However, this does not mean that it must always return to an accepting state: it can get into a loop producing outputs forever and never try to consume another input. Second, it is impossible for two inputs to be processed simultaneously. If an input arrives while the system is busy producing outputs, we assume it is transparently enqueued and processed later—our security properties are designed with asynchronous communication in mind. Third, this definition does not demand that reactive systems be deterministic.

We also need to explain how this definition models the communication we have in mind. The senders and recipients of inputs and outputs are not modeled explicitly: we assume there is a tag, such as a channel name, encoded in each element of the sets *Input* and *Output* that indicates its sender or receiver. More interestingly, the definition requires every small step to produce an output. This technical device does not constrain the model in practical terms because, when we talk about security, we assume that different outputs (and inputs) may be unobservable to different agents, so we can easily model the act of a machine taking a silent, internal step with a system transition having an output that is unseen by all observers. This assumption simplifies the definition of our system and allows us to treat silent divergence uniformly, as a behavior that is always relative to an observer. Similarly, there is no harm in forcing the system to go to a producer state after every input: if we wish to model a machine that ignores an input entirely, the system may simply produce a single invisible output and return to a consumer state. This assumption guarantees that an infinite stream of inputs will produce an infinite stream of outputs, which turns out to reduce a surprising amount of redundancy and clutter in some of our proofs.

Each state in a reactive system has a natural interpretation as a (nondeterministic) transducer between input streams and output streams. In this paper we take "streams" to be finite or infinite sequences of elements: formally, this is the coinductive interpretation of the grammar

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

where s ranges over stream elements. We use metavariables $I \in \mathit{Stream\ Input}$ and $O \in \mathit{Stream\ Output}$ to

[2] `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`

range over streams of inputs i and outputs o, respectively. The formal definition of how a a system state relates input streams and output stream is quite straightforward.

2.2 Definition: Coinductively[3] define $Q(I) \Rightarrow O$ (Q translates the input stream I to the output stream O) with the following rules:

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

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

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

2.3 Lemma: For every $Q \in \mathit{State}$ and $I \in \mathit{Stream\ Input}$, there exists an $O \in \mathit{Stream\ Output}$ such that $Q(I) \Rightarrow O$.

Although, we make no formal assumption that such transducers be deterministic, it will turn out that most of the notions of security we explore imply that each input stream is transduced to just one output stream.

In order to illustrate how a reactive system might be programmed, we now introduce the syntax of a very simplistic, imperative reactive programming language (RIMP). The full semantics are given in Section 5; here we rely on an intuitive explanation of RIMP's operational model. Inputs 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."

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

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

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

[3]With a coinductive definition, we take a greatest fixed-point interpretation of a set of inference rules. Intuitively, this simply means that derivations need not be finite. For background on coinductive reasoning, see the tutorial by Jacobs and Rutten [8].

A program consists of 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, after replacing the parameter x with the message contents. The handler code may examine and modify global state that is shared among all the handlers, output messages on channels, branch and loop. If the handler for an input terminates, the RIMP program returns to a state in which it can handle another input. Note that, since handlers share state, processing an input may affect how later inputs are handled.

The global state, called the *store*, is a mapping from variables r to natural numbers. We assume that each variable in the shared global state is initialized to 0 at the start of the program. A consumer state consists of the program text and the shared global state. 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 $\bullet$.

3 Security of Reactive Systems

Reactive systems may send messages to and receive messages from multiple agents, which we will call *principals*. The scenario of interest to us is when the principals would like to use the reactive system to communicate private data to other principals who are authorized to see it, while having a guarantee that unauthorized principals will not learn anything about the secret information via their interactions with the system. We assume there is a pre-order of security labels $(\mathcal{L}, \leq)$ and that all principals have a label corresponding to their level of authorization. We also assume that messages interchanged with the system have a label and that there is a mechanism (such as cryptography) restricting the observation of such communications to principals having equal or higher labels.

Now we can state an informal definition of information security: if an observer 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, 18], which say that, if the low-readable inputs to a program are equivalent, then the low-readable outputs must be equivalent when (and if) the program terminates. We can formalize our notion of security in the following way:

3.1 Definition: A state Q is *secure* if, whenever

$Q(I) \Rightarrow O$ and $Q(I') \Rightarrow O'$, it follows that, for all l, $I \approx_l I'$ implies $O \approx_l O'$.

The notation $S \approx_l S'$ is meant to stand for a similarity relation on streams parametrized by a label l—the inability of an observer at level l to draw a distinction between S and S'. Defining this relation is where things become interesting: it turns out that there are multiple, natural notions of similarity between streams relative to an observer who cannot see all of the elements, leading us to multiple notions 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 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 write $\mathit{fin}(S)$ when S is finite and $\mathit{inf}(S)$ when S is infinite.

Finally, we need notations for identifying the next stream element that is visible to an observer at level l (if one exists), and for characterizing streams that have no more elements visible to an observer at level l.

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

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

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

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

Nonconflicting Security The coarsest version of similarity, *nonconflicting similarity*, simply requires that the observer cannot find two distinct stream elements in corresponding positions in the streams:

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.

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⊤(x) {
  output ch⊥(x);
}
```

This event handler has an *explicit flow*, and 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⊤(x) {
  r := x;
}
input ch⊥(x) {
  if r = 0 then
    output ch⊥(0);
  else
    output ch⊥(1);
}
```

The second event handler in this example has an *implicit flow*. It is deemed insecure because the 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 $[\bullet, \bullet, \bullet, ch_\bot(1), \bullet]$ and 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 lost some of its strength by handling the input and output streams separately rather than as one interleaved stream. It turns out that this particular instantiation of our security definition is, in fact, rather weak in this regard.

3.8 Example: The following program is NC-secure:

```
input ch⊤(x) {
  r := x;
}
input ch⊥(x) {
```

```
    if r = 0 then
      output ch⊥(0);
    else
      output ch⊥(0);
      output ch⊥(0);
}
```

Although this example looks much like the previous one, it 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 that coincides with the obvious definition on finite streams (i.e., dropping invisible items and comparing what remains for equality).

Indistinguishable Security We modify the previous definition by allowing the observer 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 we would define distinctness of finite streams if we had to do so inductively, so its behavior on finite streams is very predictable. 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⊤(x) {
  r := x;
}
input ch⊥(x) {
  if r = 0 then
    output ch⊥(0);
  else
    while 1 do skip;
}
```

The second event handler in this example creates a *termination channel*. Observe that the input streams $[ch_\top(0), ch_\bot(0), ch_\bot(0)]$ and $[ch_\top(1), ch_\bot(0), ch_\bot(0)]$ are ID-similar at $\bot$ and the corresponding output streams $[\bullet, \bullet, \bullet, ch_\bot(0), \bullet, \bullet, ch_\bot(0), \bullet]$ and $[\bullet, \bullet, \bullet, \bullet, \ldots]$ are, in fact, also ID-similar at $\bot$. Termination-insensitive definitions of security are quite common in language-based information-flow because ruling out termination channels either requires the elimination of too many useful, secure programs, or else requires the use of static termination checking. It is perhaps even a bit surprising that a termination-insensitive definition of noninterference for reactive programs exists at all.

Standard definitions of noninterference [21, 18] usually imply some sort of functional dependency between the inputs and outputs of a program. The same is true here (and this is a useful fact 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 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 direct definition of similarity, called *coproductive similarity*, that is akin to a weak bisimulation between the two streams, where 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. CP-security corresponds closely to the definition of security for interactive programs given by O'Neill, Clarkson, and Chong [16].

The inductive definitions of NC-similarity and ID-similarity resemble one another, so it is easy to establish their relationship; 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? Since CP-similarity appears both covariantly and contravariantly in the definition of CP-security, their relationship is not clear. 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 Although CP-security is quite strong, 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:

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

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

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

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⊤(x) {
  r := x;
  if x = 0 then
    while 1 do skip;
}
```

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 such a choice cannot leak information 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 high input was nonzero can send a message to the machine and wait for a response (we assume that there is no other way to probe the system). Since a response would never be given to the low observer if the high

input were 0, CP-security guarantees that no response will be given if the high input were 1. Therefore the observer cannot learn anything about the value of the high input. Thus CP-security is weaker than CPCT-security only on paper. That being said, a type system for ensuring CP-security may naturally ensure CPCT-security just as one for ensuring NC-security naturally ensures ID-security.

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. Since enforcing CP-security through language-based techniques takes us into a rather complex design space that has already been partially explored by O'Neill, Clarkson, and Chong, we choose to focus on ID-security. 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 system.

4 A Method for Proving ID-Security

We now present a generic technique for proving the ID-security of a state in a reactive system.

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. What we are accomplishing here is really the same thing that Goguen and Meseguer were doing with their "unwinding lemma" [4]: we are taking a property of the input/output behavior of a system and reframing it in terms of the states and transitions of the system. What we end up with is more complex because we are interested in a termination-insensitive definition of security for systems that might diverge.

Others have investigated a variety of related (bisimulation-based) methods for establishing security properties [2]. However, the relations we have characterized here are different from standard weak bisimulations (taking our invisible inputs and outputs as analogous to internal τ actions). In one respect they are stronger: considering the first of the three cases under item (e), we see that, if one side can make a step with an output o, then *all* steps taken by the other side must produce the output o. On the other hand, the other two cases permit one side to take a silent step by itself, which allows one side to get infinitely far ahead of the other when this definition is used coinductively. An ID-bisimulation differs from standard bisimulations because it is tailored specifically to proving ID-security.

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:

$$\frac{}{[] \approx_l^{VS} []}$$

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

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

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

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 definition of 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 $Q \sim_l Q'$ where $Q(I) \Rightarrow O$ 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 intended goal.

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

5 RIMP

We now 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 are 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.
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 evaluation of expressions is completely standard for RIMP. We can use a big-step evaluation relation, since expressions cannot diverge nor have side-effects.

5.1 Definition: Inductively define $\mu \vdash e \Downarrow n$ with the following rules:

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

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

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

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

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

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

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

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

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.

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

$$\frac{}{(\mu, (\texttt{skip};\ c)) \xrightarrow{\bullet} (\mu, c)}$$

$$\frac{(\mu, c_1) \xrightarrow{o} (\mu', c_1')}{(\mu, (c_1;\ c_2)) \xrightarrow{o} (\mu', (c_1';\ c_2))}$$

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

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

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

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

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

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

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.

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

$$\frac{}{(\cdot)(i) \Downarrow \texttt{skip}}$$

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

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

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).

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

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

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

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

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 given a function *lbl* that associates a label with every channel and with every variable and define $visible_l(ch(n))$ to mean that $lbl(ch) \leq l$, for both inputs and outputs. Define $visible_l(\bullet)$ 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 have only one parameter x in this language, we write it this way for consistency of notation with standard typing judgments in more expressive languages.)

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

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

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

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.

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

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

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

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

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

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

$$\frac{\Gamma \vdash e : l \qquad \Gamma \vdash c_1 : pc_1 \qquad l \leq pc_1 \qquad pc \leq pc_1}{\Gamma \vdash \texttt{while}\ e\ \{c\} : pc}$$

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.

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

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

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.

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

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

We do not investigate decidability of typechecking formally, but note that this type system is just a small extension to a standard, termination-insensitive type system for an IMP; nothing fancy is needed to deal with RIMP's reactivity. Thus, it should be completely straightforward to write an algorithm to check the type of terms.

These definitions have the standard type preservation property.

5.9 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.

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

Next, to define when two commands are related, we must first define a predicate $high_L(c)$ 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.

5.11 Definition: Inductively define $high_L(c)$ with the following rules:

$$\frac{L \text{ is upward-closed}}{high_L(\texttt{skip})}$$

$$\frac{high_{L_1}(c_1) \qquad high_{L_2}(c_2)}{high_{L_1 \cup L_2}(c_1;\ c_2)}$$

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

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

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

$$\frac{high_L(c)}{high_L(\texttt{while}\ e\ \{c\})}$$

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.

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

$$\frac{}{\texttt{skip} \sim_l \texttt{skip}}$$

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

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

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

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

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

$$\frac{high_L(c) \qquad high_L(c') \qquad l \notin L}{c \sim_l c'}$$

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

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

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

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

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.

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

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

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

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

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

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 a ID-bisimulation.

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

5.15 Lemma: If $high_L(c)$ and $(\mu, c) \xrightarrow{o} (\mu', c')$, then $high_L(c')$ and, for all $l \notin L$, we have $\neg\ visible_l(o)$ 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.

5.16 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.

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

6 Related Work

There is a large body of research on using language-based techniques to enforce information-flow properties; see Sabelfeld and Myers' survey for an overview [20]. Much of this prior work has concentrated on batch-oriented, non-reactive programming models, though there is also much work on studying concurrent systems in various formalisms (see for example, the work by Pottier [17], or Mantel and Sabelfeld [9]). RIMP's event-handling model and interactive stream semantics distinguish it from most of these approaches, though its type system is similar to ones in the line of work initiated by Volpano, Smith, and Irvine [21]. Askarov, Hunt, Sabelfeld, and Sands [1] gave one of the first rigorous models of incremental output in sequential languages; our intention is to model incremental output *and* incremental input.[4] In what follows, we describe the approaches nearest to our reactive programming setting.

The prior research most closely related to ours is O'Neill, Clarkson, and Chong's work on interactive programs [16] (hereafter "OCC"), which builds upon Halpern and O'Neill's Multiagent Systems Framework [5]. The OCC paper focuses on a language with explicit input operations that block waiting for designated principals to respond. This differs significantly in technical detail from the input handlers in RIMP. These authors 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. In contrast, our definition of ID-security rules out such communication by requiring the output stream to be a function of the input stream. The OCC work also considers nondeterministic systems, which are ruled out by our definitions of security. We strongly suspect that when restricted to the deterministic subset of OCC, their definition of noninterference should be effectively equivalent to our notion of CP-security.[5] As a consequence, they are forced to use a very restrictive type system that prohibits looping on high data, which is a form of termination-*sensitive* security. In contrast, the ID-security enforced by RIMP is termination-*insensitive*.

Focardi and Gorrieri [2] consider the security of labeled transition systems in general, so their definitions could, in theory, be directly applied to our reactive systems. However, all of the definitions they study are tailored to allow the systems to behave nondeterministically. Their definitions (some of which are based on sets of traces, following on the work of McLean [12, 13, 14] and McCullough [11], and other of which are based on weak bisimulations) are fundamentally weaker than the assertion that ID-similar input streams must produce ID-similar output streams. Zakinthinos and Lee [22] also consider security definitions as properties of sets of traces (which, again, could be naturally applied to our reactive systems). They compare the relative strengths

[4] However, their results on the weaknesses of termination-insensitive definitions of security would apply equally well in our setting.

[5] We have not established this claim formally because the technical details of mapping their formal model into ours are quite complex.

of several definitions from the literature. However, our notion of ID-security, which we feel captures an important notion of security, isn't even a "security property" according to Zakinthinos and Lee because it can't be expressed using their trace closure predicates.

Goguen and Meseguer's seminal work [3, 4] gives a computation model that looks somewhat similar to our notion of reactive system and they study its security properties. The key difference between their model and ours is that they only allow a machine to take one small step after each input, whereas our model permits the transducer to generate many outputs or diverge in response to a single input. Moreover, we have given both a bisimulation-based proof technique and a sound type system for enforcing ID-security.

Matos, et. al. [10] give a type system and a proof of information-flow security for "reactive programs;" however, their notion of reactive programs is really a sequential language with some nonstandard programming constructs. Their programs (deterministically) run to completion without consuming any intermediate input or producing any intermediate output.

7 Conclusions and Future Work

In this paper, we have developed theoretical tools for guaranteeing the information-flow security of programs driven by event handling. This work involves technical issues on multiple levels, and we have factored our development so that pieces of our solution can be reused in different settings.

First, we designed a general model of reactive computation that is applicable to many systems that can be thought of as "reactive." Our model most closely corresponds to languages that lack preemptive multitasking, such as JavaScript. However, it does not preclude the possibility of preemptive multitasking as part of the system behavior. In that case, the model would have to be instantiated with a particular scheduling algorithm as part of the language semantics. It would be interesting to address the security of such a language in future work.

Second, we examined a spectrum of natural definitions for the information-flow security of reactive systems. Each of these definitions is applicable to any instantiation of our model of reactive behavior. We furthermore identified two definitions of particular interest: ID-security and CP-security, which correspond to termination-insensitive and termination-sensitive noninterference in the context of sequential languages. It is somewhat surprising that there is a termination-incentive definition since reactive programs are not intended to ultimately halt with a final result. Moreover, these two definitions do not share the weaknesses of a possibilistic notion of security because they impose functional behavior on the system.

Third, we designed a bisimulation-based proof technique for ensuring ID-security that decomposes the end-to-end notion of security into properties of the behavior of the individual transitions of this system. Although we demonstrated this technique using the simple RIMP language, this technique can be used to show the security of any language that behaves in a reactive manner. It should also be possible to design a similar, equally generic, bisimulation-based proof technique for ensuring CP-security, but we haven't pursued this direction because realistic CP-secure languages are difficult to design.

To demonstrate the viability of our proof technique and give examples of the differences between the security definitions, we instantiated our model with the RIMP language. Although event handling in RIMP was designed to model that of JavaScript, there is further work that must be done before we can apply our ideas to the domain of web scripting languages: RIMP does not include many features of JavaScript—in particular, timer events, first class functions, the ability to dynamically add and remove handlers, or dynamic evaluation of code. We would also need to address a mechanism for making the DOM interface secure. Additionally, the transfer structured data, especially where parts of the structure have different security annotations is something that will have an impact on our model because we currently require each stream element to be tagged with a single security label. There are also some practical issues to resolve, including an account of backwards-compatibility. However, we believe that this paper forms a firm foundation to addressing the problem of implementing a secure language for web programming and ultimately creating secure web systems.

References

[1] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David 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, October 2008.

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

[3] 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, April 1982.

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

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

[6] Collin Jackson, Andrew Bortz, Dan Boneh, and John 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.

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

[8] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.

[9] Heiko Mantel and Andrei Sabelfeld. A unifying approach to the security of distributed and multithreaded programs. *Journal of Computer Security*, 11(4):615–676, 2003.

[10] Ana Almeida Matos, Gérard Boudol, and Ilaria Castellani. Typing noninterference for reactive programs. In *In Proceeding of the Workshop on Foundations of Computer Security*, 2004.

[11] Daryl 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.

[12] John McLean. Reasoning about security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 123–131, Oakland, CA, 1988. IEEE Computer Society Press.

[13] John McLean. Security models and information flow. In *Proc. IEEE Symposium on Security and Privacy*, pages 180–187. IEEE Computer Society Press, 1990.

[14] 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. IEEE Computer Society Press, May 1994.

[15] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized javascript. A Google research project., January 2008.

[16] Kevin R. O'Neill, Michael R. Clarkson, and Stephen 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] François Pottier. A simple view of type-secure information flow in the π-calculus. In *Proc. of the 15th IEEE Computer Security Foundations Workshop*, 2002.

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

[19] Charles Reis, Steven D. Gribble, and Henry M. Levy. Architectural principles for safe web programs. Presented at the Sixth Workshop on Hot Topics in Networks (HotNets-VI), November 2007.

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

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

[22] Aris 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.