Monitoring Reactive Systems with Dynamic Channels

Dante Zanarini Mauro Jaskelioff
CIFASIS - CONICET
Universidad Nacional de Rosario
{zanarini, jaskelioff}@cifasis-conicet.gov.ar

Abstract

Given the increasingly sensitive data that web applications deal with, a lot of attention has been put into their security. Dynamic methods for ensuring confidentiality of secret data, such as monitors, are usually preferred due to their permissiveness and ability to adapt to dynamic features of web languages. One dynamic approach to confidentiality is through secure multi-execution, a technique which transforms programs into secure ones. A recent refinement of this technique led to a monitor for reactive systems such as web applications which is precise, in the sense that it raises an alarm exactly when a security condition is violated, and transparent, in the sense that the semantics of secure programs is preserved. A limitation of this and other approaches based on secure multi-execution is that there is a fixed set of channels with a fixed security level. However, most web applications create channels dynamically, even by doing something as trivial as adding a button to a page. Moreover, the security level of such new channel would be chosen dynamically. In this work, we overcome the limitation of assuming a fixed set of channels and introduce a model of reactive systems with dynamic channels and present a precise and transparent monitor for it.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.4.6 [*Operating Systems*]: Security and Protection—Information flow controls

General Terms Languages, Security

Keywords Information Flow, Reactive Systems, Dynamic Channels

PLAS 2014, July 29th, 2014, Uppsala, Sweden.
Copyright © 2014 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

1. Introduction

Web applications are pervasive and deal with data of increasing significance. Therefore, many researchers have been working towards the goal of securing them. In particular, there have been many efforts to apply information flow techniques to web applications as a general mechanism to enforce confidentiality [7, 18, 20, 24, 35]. Given that several of the OWASP top-ten web vulnerabilities [37] can be rephrased in terms of information-flow problems, a general information-flow technique would be a substantial improvement over ad-hoc and purpose-specific approaches such as architectures to contain advertisement scripts [23], and browser extensions to control cache-based leaks [19], to name just a few.

However, it is difficult to obtain a general solution for controlling information flow in web applications. There are millions of deployed web pages. A good solution should raise a minimum of false alarms when applied to them, and hence it must be permissive and allow as many secure web pages as possible. Moreover, the advanced language features of web applications make them difficult to analyse. Therefore, researchers tend to work with simplified models of real web applications in order for the problem to be tractable.

Static Denning-style [14, 30, 36] enforcements of information flow, can reject secure programs, as they over-approximate the problem. Moreover, dynamic features of web applications can be difficult to work with in this style. Hence, in order to increase permissiveness and be able to cope with dynamic features of web applications, many approaches tend to be dynamic, usually in the form an execution monitor [1, 3, 4, 31], despite their inability to enforce sound and precise standard notions of information-flow policies, such as non-interference [25, 32].

Recently, Zanarini, Jaskelioff, and Russo [39] introduced a monitor for general reactive systems (such as web applications) which can precisely detect when a program leaks information, where the information leak is defined with respect to an input without secrets. The monitor is based on the idea of secure multi-execution (SME) [15], a technique which by executing the same program several times, once for each security level, transforms an arbitrary program into a secure one. By improving SME so that the order of events is preserved across security levels, one can compare the original program with the transformed one and see if they differ, and construct in this way a security monitor which is both precise (only raises alarms when there is an information leak) and transparent (it does not affect the execution of secure programs.)

In reactive systems one has input channels on which events arrive, and output channels on which events are outputted. For example, in a web application, every element of the DOM tree has events associated to it (such as `onmouseover`), and therefore we can think that every element has an input channel associated to it. Having all events arrive through a single channel is not an option since we expect different nodes to have different security levels. An output event can be, for instance, a modification of an attribute of an existing node of the DOM, or the addition of a new one.

One limitation of the approach in [39] and of other approaches based on SME [5, 6, 15, 21, 28] is that there is a fixed set of channels, each with a fixed security level. However, web applications may modify the DOM during execution [8], and therefore the set of possible input channels may be modified during execution. That is, the set of channels is *dynamic*. For example, when adding a button to a web page, a new set of events will be possible (such as an `onclick` event). Such events must come through a new channel, which will have an application and context-specific security level. That is, depending on the context, the events from a newly created channel might be secret or public, and what's more, the security level of an existing channel might change during execution. Therefore, in order to model more realistic reactive systems, a way of dealing with dynamic channels is needed. Note that, one could improvise a way of dealing with dynamic channels by assumming a countably infinite set of channels and assigning a "fresh" channel name to a new channel, but this approach has clear diasvantages: first and foremost, the security level for each channel is fixed and cannot be changed dynamically; second, the interpretation of a source language into a model with a fixed set of channels is more complex, and hence more error-prone.

Adding dynamic channels to a model has the disadvantage of making it more complex. The very question of what is a secure program in this context becomes non-trivial, as the security level of a channel might change during execution. However, if we are to tackle realistic web applications, it is a complexity that needs to be dealt with.

In this article, we extend the monitor for reactive systems introduced in [39] to account for dynamic channels. In order to do this, we need to extend the model of reactive systems and adapt the notion of non-interference. Notably, our extended monitor offers the same security and transparency guarantees as the one for static channels.

In Section 2, we introduce a model of reactive systems with dynamic channels, and show how a JavaScript-like language can be interpreted into such a model. In Section 3 we introduce notions of security for programs and for runs. In Section 4, we introduce our monitor for reactive systems with dynamic channels. In Section 5, we state the precision and transparency guarantees of the monitor. In Section 6 we discuss related work. Finally, in Section 7, we conclude. Proofs for all the results can be found in an online extended version [38].

2. Modelling Reactive Systems

We want to provide an abstract model for reactive systems. The model should be as abstract as possible in order to simplify away irrelevant features, but powerful enough so as to model all the pertinent features. The main example of a reactive system we have in mind is web applications. Therefore we will use this example to drive intuitions.

A reactive system consists of input events and of reactions to those events, which may be output events or simply an invisible change of the internal state of the system. Each input or output event occurs in a given channel. When thinking about web applications, a typical input event is a mouse click on an element of a web page, and a typical output event is putting some text on the screen. More precisely, a web page is represented by a DOM tree, input events are associated to elements of the DOM, and output events modify the DOM. An obvious way of adding security level information to a DOM, is adding an attribute to indicate the security level of the element, which may be modified during execution.

We model a reactive program as an interaction structure [17]. Our model is based on the model in [39], but extended to account for the dynamic opening and closure of channels. That is, we model reactive systems using an interaction structure indexed by a mapping of open channels to their corresponding security levels.

Throughout this article, we organise security levels in a lattice $(\mathcal{L}, \sqsubseteq)$, with the intention to express that data at level ℓ_1 can securely flow into data at level ℓ_2 when $\ell_1 \sqsubseteq \ell_2$. Let Chan be the set of channel names, and let $\mathsf{CS} = \mathsf{Chan} \rightharpoonup \mathcal{L}$ be the type of channel systems, i.e. a channel system $cs : \mathsf{CS}$ is a partial function from channels to security levels. If $c \in \mathrm{dom}(cs)$, we say that c is open in cs.

An event is a piece of data paired with the name of the communication channel associated to it. Given a channel system cs, the domain of cs indicates which channels are open, and therefore which events are compatible with cs. Take A to be a set; the set of A-events compatible with cs is:

$$E_A\ cs = \{(c, a) \in \mathsf{Chan} \times A \mid c \in \mathrm{dom}(cs)\}\,.$$

That is, $E_A\ cs$ denotes the set of events on open channels of the channel system cs. Equivalently, the set of compatible events can be defined as

$$E_A\ cs = \{(c, a, \ell) \in \mathsf{Chan} \times A \times \mathcal{L} \mid cs(c) = \ell\}\,.$$

It is not difficult to see that the two definitions are equivalent. However, as discussed later in the article, the latter is more convenient for analysing confidentiality in a dynamic-channel context. We denote with I the set of input values, and with O the set of output values.

Definition 2.1 (Reactive Behaviour)**.** A reactive behaviour $\mathsf{React} : \mathsf{CS} \rightarrow \mathsf{Set}$ is the interaction structure given by the following coinductive datatype indexed by a channel system.

$$\begin{array}{l}
\mathsf{React} : \mathsf{CS} \rightarrow \mathsf{Set} = \\
\quad \mathsf{Read} : \forall cs.\, (E_I\ cs \rightarrow \mathsf{React}\ cs) \rightarrow \mathsf{React}\ cs \\
\quad \mathsf{Write} : \forall cs.\, E_O\ cs \rightarrow \mathsf{React}\ cs \rightarrow \mathsf{React}\ cs \\
\quad \mathsf{Step} : \forall cs.\, \mathsf{React}\ cs \rightarrow \mathsf{React}\ cs \\
\quad \mathsf{Stop} : \forall cs.\, \mathsf{React}\ cs \\
\quad \mathsf{New} : \forall cs.\, ((c, \ell) : \mathsf{Chan} \times \mathcal{L}) \rightarrow \mathsf{React}\ (cs \oplus (c, \ell)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \mathsf{React}\ cs \\
\quad \mathsf{Close} : \forall cs.\, (c : \mathsf{Chan}) \rightarrow \mathsf{React}\ cs \\
\qquad\qquad\qquad\qquad\qquad \rightarrow \forall \ell.\, \mathsf{React}\ (cs \oplus (c, \ell))
\end{array}$$

The idea is that $\mathsf{React}\ cs$ describes the execution behaviour of a reactive system whose open channels are described by a channel system cs. Because the behaviour datatype is defined coinductively, it is perfectly suited to model the behaviour of programs that run indefinitely.

Intuitively, constructor Read denotes a behaviour $\mathsf{React}\ cs$ that receives an input from an open channel determined by the environment ($E_I\ cs$) and, based on that, decides how to continue ($\mathsf{React}\ cs$). Constructor Write represents a behaviour $\mathsf{React}\ cs$ which writes an output in a chosen channel ($E_O\ cs$) and continues with another computation ($\mathsf{React}\ cs$). Constructor Step corresponds to a silent step, that is, a computation which does not affect the environment. Silent steps allows us to model divergence. Constructor Stop models termination. Reactive systems usually run forever, but we use this constructor to model abnormal termination as triggered by an unrecoverable error condition. Constructor New denotes a behaviour $\mathsf{React}\ cs$ which opens a new channel c of security level ℓ, and continues with a behaviour on a channel system which extends cs with (c, ℓ). Note that in a behaviour $r' = \mathsf{New}\ (c, \ell)\ r : \mathsf{React}\ cs$ the new channel c is open only in the continuation r and not on r'. Finally, constructor Close represents a behaviour on a channel system with an open channel c and continues with a behaviour with that channel closed. That is, the behaviour $r' = (\mathsf{Close}\ c\ r)$ has channel c open, but its continuation r must have it closed.

The operation $\oplus$ adds a channel to a channel system. More precisely, given a channel c and a channel system cs such that $c \notin \mathrm{dom}(cs)$, $cs \oplus (c, \ell) = cs \cup \{(c, \ell)\}$. As a consequence of this definition, in a behaviour $r' = \mathsf{New}\ (c, l)\ r : \mathsf{React}\ cs$, we know that c is not open in cs, and r' can interact with c only in the continuation r. A similar observation applies to constructor Close. If $r' = \mathsf{Close}\ c\ r : \mathsf{React}\ (cs \oplus (c, \ell))$, then c is closed in cs, and the continuation r cannot interact with c.

By modelling reactive systems with an interaction structure we obtain a black-box model of the system. The constructors represent only the features which are relevant to our analysis of security and therefore we can abstract away all the rest of the complex features of the system. In this manner, we gain modularity since new programming features which are not related to I/O do not affect our formal results. Indexing by the channel system means that behaviours cannot do nonsense such as writing to a closed channel or opening an open channel. Therefore, working with behaviours is simplified since we do not need to consider absurd cases. Of course, a JavaScript program might be erronous and try to perform an illegal operation, but this can be handled as an error, perhaps behaving as a Stop.

2.1 Semantics

A reactive behaviour is essentially a static description of the possible interactions that might occur during execution. In order to know what happens in a given run, we need to give reactive behaviours some semantics. We start by defining possibly infinite sequences.

Definition 2.2 (Colists)**.** Let A be a set. Consider the type of possibly infinite sequences of A to be coinductively defined as follows.

$$Colist_A = \ [] \mid A :: Colist_A$$

We write $[a, b, c]$ to denote a finite colist $a :: b :: c :: []$.

The structure of input and output events of the system is given by colists. The set of all events with data from a set A is defined as $\mathcal{E}_A = \mathsf{Chan} \times A \times \mathcal{L}$. It is easy to see that for any channel system cs, the set $E_A\ cs$ of compatible events is included in $\mathcal{E}_A$.

Definition 2.3 (Colists of input and output events)**.** We define the set of colists of inputs $\mathcal{I}$ and the set of colists of outputs $\mathcal{O}$ for reactive systems as follows.

$$\mathcal{I} = Colist_{\mathcal{E}_I} \qquad\qquad \mathcal{O} = Colist_{\mathcal{E}_O \cup \{\bullet, \circ\}}$$

The elements of an output colist can be events in $\mathcal{E}_O$, an invisible output ($\bullet$), or an abnormal termination event ($\circ$).

The rules in Figure 1 coinductively define an evaluation relation for reactive behaviours. This is really a *family* of relations indexed by a channel system. More precisely, we define a family of relations $\Rightarrow_{cs} \subseteq (\mathsf{React}\ cs \times \mathcal{I} \times \mathcal{O})$, and write $(t, i) \Rightarrow_{cs} o$ for $(t, i, o) \in \Rightarrow_{cs}$. Intuitively, feeding a colist of inputs i to a reactive behaviour t yields the output colist $o \in \mathcal{O}$ iff $(t, i) \Rightarrow_{cs} o$.

Rule R_1 produces no outputs ($[]$) when no input events are present. Rule R_2 consumes the first available input event (e) if the event e is compatible with cs, and based on that, produces the output o ($(f\ e, i) \Rightarrow_{cs} o$). Rule W outputs an event e ($e :: o$), followed by the outputs triggered by program t ($(t, i) \Rightarrow_{cs} o$). Rule N continues execution in the channel system obtained after opening a new channel ($cs \oplus (c, \ell)$).

$$\mathrm{R}_1\ \frac{}{(\mathsf{Read}\ f, []) \Rightarrow_{cs} []}$$

$$\mathrm{R}_2\ \frac{e \in E_I\, cs \qquad (f\ e, i) \Rightarrow_{cs} o}{(\mathsf{Read}\ f, e :: i) \Rightarrow_{cs} \bullet :: o}$$

$$\mathrm{W}\ \frac{(t, i) \Rightarrow_{cs} o}{(\mathsf{Write}\ e\ t, i) \Rightarrow_{cs} e :: o}$$

$$\mathrm{N}\ \frac{(t, i) \Rightarrow_{cs \oplus (c,\ell)} o}{(\mathsf{New}\ (c, \ell)\ t, i) \Rightarrow_{cs} \bullet :: o}$$

$$\mathrm{C}\ \frac{cs = cs' \oplus (c, \ell) \qquad (t, i) \Rightarrow_{cs'} o}{(\mathsf{Close}\ c\ t, i) \Rightarrow_{cs} \bullet :: o}$$

$$\mathrm{S}\ \frac{(t, i) \Rightarrow_{cs} o}{(\mathsf{Step}\ t, i) \Rightarrow_{cs} \bullet :: o} \qquad \mathrm{E}\ \frac{}{(\mathsf{Stop}, i) \Rightarrow_{cs} [\circ]}$$

Figure 1. Evaluation relation for reactive behaviours

Rule C continues execution in a channel system obtained after closing channel c (that is, cs'). Rule S simply outputs $\bullet$ when a silent computation step is performed. Finally, rule E signals an abnormal termination event and ends. Here we see the role of the termination event $\circ$: to indicate a termination different from $[]$.

It might seem strange that input events carry a security level. It would be more natural to consider input events as pairs (c, d) of a channel and a datum, and assign the security level of the event according to the mapping of c in the channel system at the time the event arrives. As a matter of fact, this is exactly the way we expect to *implement* the semantics of behaviours. Nevertheless, this "natural" semantics would be equivalent to the one in Fig. 1 as the predicate $e \in E_I\, cs$ in R_2 filters all non-compatible events. Of the two equivalent semantics, we have chosen the less intuitive semantics of Fig. 1 as it is better suited to formal analysis. In fact, considering input events equipped with a security level is precisely what is needed in order to define a notion of confidentiality for systems with dynamic channels (see Section 3).

2.2 Reactive Behaviour of a JavaScript-like Language

Reactive behaviours are independent of the features of the programming languages that generate them, which may be, for example, imperative or functional, call-by-name or call-by-value, etc.

In this section, we show how to construct reactive behaviours from programs in a JavaScript-like language. Figure 2 presents the syntax of a language, which is inspired by JavaScript models found in previous works [7, 27, 39].

Programs are defined as a list of event handlers. A handler has the form $(ch(x)\{c\})$, and associates a channel ch to a command c to be executed when an input message (stored in variable x) is received on channel ch. A command c is an imperative program augmented with output messages (`out`), handler creation (`new`), and dynamic creation (`open`) and deletion (`close`) of communication channels.

$$
\begin{array}{rl}
p ::= & \cdot \mid h; p \\
h ::= & ch(x)\, \{c\} \\
c ::= & \texttt{skip} \\
 & \mid c; c \\
 & \mid x\ \texttt{:=}\ e \\
 & \mid \texttt{if}\ e\ \{c\}\, \{c\} \\
 & \mid \texttt{while}\ e\ \{c\} \\
 & \mid \texttt{out}(ch, e) \\
 & \mid \texttt{new}\ h \\
 & \mid \texttt{open}(ch, \ell) \\
 & \mid \texttt{close}(ch)
\end{array}
$$

Figure 2. A JavaScript-like language.

Command $\texttt{open}(ch, \ell)$ opens a new communication channel ch at security level ℓ. Command $\texttt{close}(ch)$ closes channel ch. Command `new` h associates to an open channel a new event handler (possibly replacing a previous handler.) Opening an open channel, closing a closed channel, or associating a handler to a closed channel, results in a run-time error.

To simplify the construction of the reactive behaviour, we assume that every handler uses variable x to store the message received, and we extend the set of commands $\mathbb{C}$ with a special symbol $\cent$.

A program state is a tuple (cs, hs, μ, c), where:

- $cs : \mathsf{CS} \rightharpoonup \mathcal{L}$ is a partial mapping from channels to security levels: the set of channels being used by the program to interact with the environment;
- $hs : \mathrm{dom}(cs) \rightharpoonup \mathbb{C} \cup \{\cent\}$ is the set of event handlers associated to input channels in cs; if an input channel ch does not have an event handler associated to it, then a default handler $hs(ch) = \cent$ is assumed;
- μ is a mapping from variables to values, the store; and
- c is the command being executed in response to an input event; or $\cent$ if the system is in a consumer state (the next interaction will be an input event).

We write $(\mathsf{State}\ cs)$ for the set of states whose first component is cs. Given a program state $s = (cs, hs, \mu, c)$, we can compute the next interaction of s with the environment. The set of possible interactions is defined as

$$\mathsf{Interaction} = \left\{\downarrow, \uparrow_v^{ch}, \odot, \Uparrow_\ell^{ch}, \Downarrow^{ch}, \circledcirc\right\}.$$

Interaction $\downarrow$ is raised in a consumer state, i.e. when $c = \cent$. Interaction $\uparrow_v^{ch}$ is raised when s will produce an output message to an open channel ch with value v. Interactions $\Uparrow_\ell^c$

$$
\begin{array}{ll}
\mathsf{step}\ (cs, hs, \mu, c) = & \textbf{case}\ c\ \textbf{of} \\
\quad \not c & \rightarrow (\downarrow, (cs, hs, \mu, \not c)) \\
\quad \texttt{out}(ch,e) & \rightarrow \textbf{if}\ ch \in \mathrm{dom}(cs) \\
 & \quad \textbf{then}\ \left(\uparrow^{ch}_{[\![e]\!]_\mu}, (cs, hs, \mu, \not c)\right) \\
 & \quad \textbf{else}\ (\circledcirc, (cs, hs, \mu, \not c)) \\
\quad \texttt{open}(ch, \ell) & \rightarrow \textbf{if}\ ch \notin \mathrm{dom}(cs) \\
 & \quad \textbf{then}\ (\Uparrow^{ch}_{\ell}, (cs[ch \mapsto \ell], hs, \mu, \not c)) \\
 & \quad \textbf{else}\ (\circledcirc, (cs, hs, \mu, \not c)) \\
\quad \texttt{close}(ch) & \rightarrow \textbf{if}\ ch \in \mathrm{dom}(cs) \\
 & \quad \textbf{then}\ (\Downarrow^{ch}, (\mathsf{del}(cs, ch), \mathsf{del}(hs, ch), \mu, \not c)) \\
 & \quad \textbf{else}\ (\circledcirc, (cs, hs, \mu, \not c)) \\
\quad \texttt{new}\ ch(x)\ \{c'\} & \rightarrow \textbf{if}\ ch \in \mathrm{dom}(cs) \\
 & \quad \textbf{then}\ (\odot, (cs, hs[ch \mapsto c'], \mu, \not c)) \\
 & \quad \textbf{else}\ (\circledcirc, (cs, hs, \mu, \not c)) \\
\quad \texttt{skip} & \rightarrow (\odot, (cs, hs, \mu, \not c)) \\
\quad \texttt{x :=}\ e & \rightarrow (\odot, (cs, hs, \mu[x \mapsto [\![e]\!]_\mu], \not c)) \\
\quad \texttt{if}\ \ e\ \{c_0\}\ \{c_1\} & \rightarrow \textbf{if}\ [\![e]\!]_\mu = 0 \\
 & \quad \textbf{then}\ (\odot, (cs, hs, \mu, c_1)) \\
 & \quad \textbf{else}\ (\odot, (cs, hs, \mu, c_0)) \\
\quad \texttt{while}\ \ e\ \ \{c'\} & \rightarrow \textbf{if}\ [\![e]\!]_\mu = 0 \\
 & \quad \textbf{then}\ (\odot, (cs, hs, \mu, \not c)) \\
 & \quad \textbf{else}\ (\odot, (cs, hs, \mu, c'\,\texttt{;}\ \texttt{while}\ \ e\ \ \{c'\})) \\
\quad c_0\,\texttt{;}\ c_1 & \rightarrow \textbf{let}\ (i, (cs', hs', \mu', c')) = \mathsf{step}\,(cs, hs, \mu, c_0) \\
 & \quad \textbf{in if}\ c' = \not c\ \textbf{then}\ (i, (cs', hs', \mu', c_1)) \\
 & \quad \quad \textbf{else}\ (i, (cs', hs', \mu, c'\,\texttt{;}\ c_1))
\end{array}
$$

Figure 3. Next interaction for a program state

and $\Downarrow^c$ are used to signal opening and closure of channels. A silent interaction is represented by $\odot$, and interaction $\circledcirc$ represents termination. Our language has no instruction to signal that execution should terminate. However, interaction $\circledcirc$ is used to denote an abnormal termination, as it may happen, for example, if a program tries to output a message on a closed channel. Figure 3 defines a function

$$\mathsf{step} : \forall cs, cs'.\, \mathsf{State}\ cs \rightarrow \mathsf{Interaction} \times \mathsf{State}\ cs',$$

which computes the next interaction with the environment of a state $s : \mathsf{State}\ cs$, together with the new state $s' : \mathsf{State}\ cs'$ that results from such interaction.

The definition of step makes use of an evaluation function for expressions; $[\![e]\!]_\mu = v$ iff expression e evaluates to value v in memory μ. We assume that $[\![\ ]\!]$ is a side-effect free function. We write $f[x \mapsto y]$ for the mapping that behaves like f except on x, where it maps to y, and $\mathsf{del}(f, a)$ for $f \setminus \{(a, f\ a)\}$.

If there is no command to execute, the system is ready to process the next input (interaction $\downarrow$). Assignments, conditionals, loops and `skip` generate internal activities, raising interaction ($\odot$). A sequence $c_0\,\texttt{;}\, c_1$ of instructions generates the interaction of c_0. If ch is open in cs, command `out`$(ch$`,`$e)$ generates an output message interaction ($\uparrow^{ch}_{[\![e]\!]_\mu}$); otherwise an error condition ($\circledcirc$) is raised, since the program is trying to send a message to a closed channel. Command `new` $(ch)\{c\}$ interacts silently ($\odot$), and associates command c to channel ch in the handler set, provided that ch is open in the actual state. Otherwise, ($\circledcirc$) is raised. If ch is closed in cs, command `open`(ch, ℓ) adds (ch, ℓ) to the channel system and generates interaction $\Uparrow^{ch}_{\ell}$. Command `close`(ch) raises interaction $\Downarrow^{ch}$, provided that ch is open in the current state. If ch is open (close) in the current state, `open`(ch, ℓ) (`close`(ch)) raises $\circledcirc$.

$$
\begin{array}{l}
[\![-]\!] : \forall cs.\mathsf{State}\ cs \rightarrow \mathsf{React}\ cs \\
[\![s]\!] = \textbf{let}\ (i, s') = \mathsf{step}(s)\ \textbf{in} \\
\quad \textbf{case}\ i\ \textbf{of} \\
\qquad \odot \quad\ \ \rightarrow \mathsf{Step}\ ([\![s']\!]) \\
\qquad \circledcirc \quad\ \ \rightarrow \mathsf{Stop} \\
\qquad \uparrow^{ch}_{v} \ \ \rightarrow \mathsf{Write}\ (ch, v, cs(ch))\ [\![s']\!] \\
\qquad \Uparrow^{ch}_{\ell} \ \ \rightarrow \mathsf{New}\ (ch, \ell)\ [\![s']\!] \\
\qquad \Downarrow^{ch} \rightarrow \mathsf{Close}\ ch\ [\![s']\!] \\
\qquad \downarrow \quad\ \ \rightarrow \mathsf{Read}\ (\lambda e.[\![\mathsf{update}(s', e)]\!]) \\
\\
\mathsf{update} : \forall cs.\mathsf{State}\ cs \times E_I\ cs \rightarrow \mathsf{State}\ cs \\
\mathsf{update}\,((cs, hs, \mu, c), (ch, v, \ell)) = (cs, hs, \mu[x \mapsto v], hs(ch))
\end{array}
$$

Figure 4. Reactive behaviour for program states

Iterating over step, one can generate a reactive behaviour $t : \mathsf{React}\ cs$ from a state $s : \mathsf{State}\ cs$, as it is shown in Fig. 4.

Definition 2.4 (Reactive behaviour for a JavaScript-like program)**.** Given a channel system cs and a memory μ, the interpretation of p in (cs, μ) is defined as

$$[\![p]\!]_{cs,\mu} = [\![(cs, \mathsf{handlers}(cs, p), \mu, \not c)]\!],$$

where

$$
\begin{array}{ll}
\mathsf{handlers} & : \mathsf{CS} \rightharpoonup \mathbb{C} \cup \{\not c\} \\
\mathsf{handlers}(cs, \cdot) & = \emptyset \\
\mathsf{handlers}(cs, ch(x)\{c\}; p) & = \textbf{let}\ hs = \mathsf{handlers}(cs, p) \\
 & \quad \textbf{in if}\ ch \in \mathrm{dom}(cs) \\
 & \qquad \textbf{then}\ \{(ch, c)\} \cup hs \\
 & \qquad \textbf{else}\ hs
\end{array}
$$

is the set of handlers in p with open channels in cs.

3. Security Conditions for Reactive Systems

Confidentiality for a reactive system can be defined in terms of a notion of similarity of colists of input and output events. In a system with dynamic channels, the security level of an event depends on the state of the system at the time of the arrival of the event. For example, an `onclick` event can be classified as public at first and as secret later. In terms of reactive behaviours, the security level of a channel might be changed by closing the channel and opening it at a different security level. In this context, the analysis of similarity of two colists for an observer at a certain security level is simplified when the security level is part of the event as we do not need to depend on the system state. This is the main reason for considering events on A as triples $\mathsf{Chan} \times A \times \mathcal{L}$. Once the security level is included in the events, the definitions are analogous to those of systems with static channels. We recap the concepts about security of reactive systems presented in previous work [39], slightly modified to work in our setting.

3.1 Reactive Noninterference

The security level of an event is defined as $\mathsf{lvl}\,(c, v, \ell) = \ell$. The predicate $\mathsf{visible}_\ell$ on events determines when an event is observable for an observer at level ℓ:

$$\frac{\mathsf{lvl}(e) \sqsubseteq \ell}{\mathsf{visible}_\ell(e)} \qquad \frac{}{\mathsf{visible}_\ell(\circ)}$$

Termination ($\circ$) is visible at all levels. Silent steps ($\bullet$), on the other hand, are not visible at any security level.

A reactive behaviour is non-interferent when similar inputs produce similar outputs for observers at any security level.

Definition 3.1 (Security for Reactive Behaviours)**.** Given a family of similarity relations $\sim_\ell$ on colists, we say that a reactive behaviour t : React cs is secure iff, for all ℓ and input colists i, i' such that $i \sim_\ell i'$, if $(t, i) \Rightarrow_{cs} o$ and $(t, i') \Rightarrow_{cs} o'$, then $o \sim_\ell o'$.

Note that the definition depends on a notion of similarity. Depending on the power of the attacker one wants to model, one chooses a notion of similarity that makes more or less distinctions and, as a result, one obtains stronger or weaker notions of security. Bohannon et al. [7] identify two notions of security with practical interest: ID- and CP-security.

Definition 3.2 (ID-similarity)**.** ID-similarity between colists for an observer at level ℓ is formalised coinductively by the following rules.

$$\frac{}{[] \sim_\ell^{\mathsf{ID}} []} \qquad \frac{\neg\mathsf{visible}_\ell(e) \quad s \sim_\ell^{\mathsf{ID}} s'}{e :: s \sim_\ell^{\mathsf{ID}} s'}$$

$$\frac{\neg\mathsf{visible}_\ell(e) \quad s \sim_\ell^{\mathsf{ID}} s'}{s \sim_\ell^{\mathsf{ID}} e :: s'} \qquad \frac{\mathsf{visible}_\ell(e) \quad s \sim_\ell^{\mathsf{ID}} s'}{e :: s \sim_\ell^{\mathsf{ID}} e :: s'}$$

Intuitively, two colists are ID-similar at level ℓ if there is no evidence that they produce different events for an observer at level ℓ. In particular, an infinitely silent colist –such as one produced by a divergent computation– is ID-similar to any other colist because we never find evidence against it. A feature of ID-similarity is that it is symmetric and reflexive, but not transitive. If it were, we could not have an infinitely silent colist ID-similar to any other as transitivity would imply that every colist is similar to each other.

In order to take progress into account, and distinguish a productive colist from a silent one, we consider a stronger notion of non-interference for reactive systems called CP-security [7].

We coinductively define when a colist of events is not visible (silent) for an observer at level ℓ as follows.

$$\frac{\neg\mathsf{visible}_\ell(e) \quad \mathsf{silent}_\ell(s)}{\mathsf{silent}_\ell(e :: s)} \qquad \frac{}{\mathsf{silent}_\ell([])}$$

We define a relation that identifies the next event that is visible to an observer at security level ℓ (if it exists). Intuitively, we say that $s \rhd_\ell e :: s'$ when e is the next event in s visible at level ℓ. The following rules inductively define the relation $\rhd_\ell$.

$$\frac{\mathsf{visible}_\ell(e)}{e :: s \rhd_\ell e :: s} \qquad \frac{\neg\mathsf{visible}_\ell(e) \quad s \rhd_\ell e' :: s'}{e :: s \rhd_\ell e' :: s'}$$

Note that the relation is inductively defined, which means that when $s \rhd_\ell e :: s'$, the next ℓ-visible event e of the colist s must come after a finite sequence of ℓ-invisible events.

Definition 3.3 (CP-similarity)**.** CP-similarity between colists is defined coinductively by the following rules.

$$\frac{\mathsf{silent}_\ell(s) \quad \mathsf{silent}_\ell(s')}{s \sim_\ell^{\mathsf{CP}} s'}$$

$$\frac{s \rhd_\ell e :: s_1 \quad s' \rhd_\ell e :: s_1' \quad s_1 \sim_\ell^{\mathsf{CP}} s_1'}{s \sim_\ell^{\mathsf{CP}} s'}$$

As opossed to ID-similarity, the notion of CP-similarity requires proof that if one colist produces a visible event e, the next observable event in the other will be e and moreover, it will be produced in a finite number of steps. Also, unlike ID-similarity, CP similarity is an equivalence relation.

By instantiating Definition 3.1 to ID- and CP-similarity we obtain the notions of ID- and CP-security for reactive behaviours.

3.2 Run-based Security

It is well known that non-interference is not a safety property and cannot be precisely enforced by execution monitors [16, 32]. Therefore, we introduce a security condition which is defined on runs, and characterises the set of *secure inputs*, i.e. inputs for which a program does not leak information. One can think that if a program p receives an input i with no secrets, then executing p by feeding it i is secure, since there are no secrets to leak[1]. Using this intuition, we classify an input i as secure if it reveals the same information as the input where secrets have been erased.

We coinductively define the relation $\blacktriangleright_\ell$ responsible for removing all the events unobservable at level ℓ.

$$\frac{\mathsf{silent}_\ell(s)}{s \blacktriangleright_\ell []} \qquad \frac{s \rhd_\ell e :: s' \quad s' \blacktriangleright_\ell s''}{s \blacktriangleright_\ell e :: s''}$$

Observe that given a colist s, there is a unique colist s' such that $s \blacktriangleright_\ell s'$. We will write $s_{\blacktriangleright_\ell}$ for this unique colist, and refer to it as the *restriction* of s at level ℓ.

Let us assume a level-indexed similarity relation $\sim_\ell$ between colists. Two inputs for a program reveal the same secrets at a given security level ℓ if, for an observer at level ℓ, they are similar and induce similar outputs.

Definition 3.4 ($\approx_{\ell,t}$)**.** Let $t \in$ React, $\ell \in \mathcal{L}$, and i, i' input colists such that $i \sim_\ell i'$, $(t, i) \Rightarrow o$ and $(t, i') \Rightarrow o'$. We say

[1] This observation is valid if we ignore covert channels, in particular, the termination channel.

that the program t reveals the same ℓ-secrets when given the inputs i, i', noted $i \approx_{\ell,t} i'$, iff $o \sim_\ell o'$.

Similarly to [6], we consider an input to be *secure* for a program t if it reveals the same information about the secrets as the input where secrets have been erased.

Definition 3.5 (Secure input)**.** Let t be a reactive behaviour. An input colist i is *secure for t* iff $\forall \ell.\, i \approx_{\ell,t} i_{\blacktriangleright_\ell}$. We say that the input i is ID-secure (CP-secure) for t when $\sim_\ell$ is instantiated to $\sim_\ell^{\mathsf{ID}}$ ($\sim_\ell^{\mathsf{CP}}$) in Definition 3.4.

Example 3.6. Consider the following program

```
p =  c?₀(x){ r := x };

     c?₁(x){ if r ≥ 1 ∨ x = 0
               {new c?₂(x){out(c!₀,r)}}
               {while 1 {skip} }};
     c?₂(x){skip}
```

Let us assume an initial memory ($\mu_0 = \lambda x.0$) and an initial channel system

$$cs = \{(\mathtt{c?}_0, \mathsf{H}), (\mathtt{c?}_1, \mathsf{L}), (\mathtt{c?}_2, \mathsf{L}), (\mathtt{c!}_0, \mathsf{L})\},$$

where L and H are security levels such that $\mathsf{L} \sqsubseteq \mathsf{H}$. Let t : React cs be the interpretation of program p in cs. That is, $t = [\![p]\!]_{cs,\mu_0}$. Consider an input i with three messages as follows:

$$i = [(\mathtt{c?}_0, v, \mathsf{H}), (\mathtt{c?}_1, u, \mathsf{L}), (\mathtt{c?}_2, 42, \mathsf{L})]$$

If $v = 1$ and $u = 0$, then i is neither ID- nor CP-secure for t. However, if $v = u = 1$, we have that i is ID-secure, but not CP-secure for t. Taking $v = u = 0$, input i becomes ID- and CP-secure for t.

The evaluation relation defined in Figure 1 is only defined for compatible inputs, which means that for incompatible inputs evaluation will get stuck. However, as shown by the following example, erasing the secrets can turn a compatible input into an incompatible one.

Example 3.7. Consider the following program:

```
p =  c?₀(x){ r := x };

     c?₁(x){ if r ≥ 1
               {open(c?₂, L);
                new c?₂(x){out(c!₀,r)}}
               {skip }};
```

Let $cs = \{(\mathtt{c?}_0, \mathsf{H}), (\mathtt{c?}_1, \mathsf{L}), (\mathtt{c!}_0, \mathsf{L})\}$, be the initial channel system, let t : React cs be such that $t = [\![p]\!]_{cs,\mu_0}$. The insecure input $i = [(\mathtt{c?}_0, 1, \mathsf{H}), (\mathtt{c?}_1, 1, \mathsf{L}), (\mathtt{c?}_2, 1, \mathsf{L})]$ is compatible with t:

$$(t, i) \Rightarrow_{cs} [\bullet, \bullet, \bullet, \bullet, \bullet, (\mathtt{c!}_0, 1, \mathsf{L})]$$

However, its restriction $i_{\blacktriangleright_\mathsf{L}} = [(\mathtt{c?}_1, 1, \mathsf{L}), (\mathtt{c?}_2, 1, \mathsf{L})]$ is not, as channel $\mathtt{c?}_2$ is never opened.

Hence, the filtering of a compatible input may be incompatible. Moreover, the fact that a program may get stuck with a filtered input does not imply that the (unfiltered) input is insecure for that program.

Therefore, in order to be able to use the notion of secure-run we need to account for incompatible input events in the evaluation relation. We extend the evaluation relation with the following rule:

$$\mathrm{R}_3\,\frac{e \notin E_I\, cs \qquad (\mathsf{Read}\ f, i) \Rightarrow_{cs} o}{(\mathsf{Read}\ f, e :: i) \Rightarrow_{cs} \bullet :: o}$$

Intuitively, Rule R_3 simply discards the next input event if it is not compatible with the actual channel system. With this additional rule, the evaluation relation is defined for every input and it becomes a function.

We can now prove the following relation between ID-secure and CP-secure inputs:

Lemma 3.8. *Let t : React cs and i an input colist. If i is CP-secure for t, then i is ID-secure for t.*

3.3 From Run-based Security to Noninterference and Back

We have the notions of ID-security and CP-security which characterise the security of programs, and the notions of ID- and CP-secure inputs which characterise the security of runs. As it is shown in [39], the security for programs and runs is closely related for CP-similarity.

Lemma 3.9 (Secure inputs and CP-security)**.** *A reactive behaviour $t \in$ React cs is CP-secure iff $\forall i \in \mathcal{I}$. i is CP-secure for t.*

It is easy to see that all inputs are CP-secure for a CP-secure program. In order to see why a program such that every input is CP-secure is a CP-secure program, observe that whenever two inputs i, i' are similar at some level ℓ they have exactly the same restriction at that level.

$$i \sim_\ell^{\mathsf{CP}} i' \quad \Longrightarrow \quad i_{\blacktriangleright_\ell} = i'_{\blacktriangleright_\ell}$$

Using the transitivity of $\sim_\ell^{\mathsf{CP}}$ and the definition of secure input, one obtains that the output colists produced by t when executed on i and i' must be CP-similar, provided that i and i' are CP-secure.

The strong correspondence between security of programs and security of runs does not hold when we consider ID-security, and we only have the property that all inputs are secure for secure programs.

Lemma 3.10 (Secure inputs and ID-security)**.** *If a reactive behaviour $t \in$ React cs is ID-secure, then every input $i \in \mathcal{I}$ is ID-secure for t.*

Even though all inputs may be ID-secure for a program, the program might be interferent, as shown by the following example.

Example 3.11. Consider the following program

$$\begin{array}{rl} p = & \texttt{c?}_0\texttt{(x)}\{\ \texttt{r := x}\ \};\\ \\ & \texttt{c?}_1\texttt{(x)}\{\ \texttt{if r} \geq \texttt{1}\\ & \qquad\qquad \{\texttt{out(c!}_0\texttt{,r)}\}\\ & \qquad\qquad \{\texttt{while 1 \{skip\}}\ \}\};\end{array}$$

where we assume an initial channel system

$$cs = \{(\texttt{c?}_0, \textsf{H}), (\texttt{c?}_1, \textsf{L}), (\texttt{c!}_0, \textsf{L})\}\,.$$

Every input is ID-secure for p but p is not an ID-secure program, as the following two ID-similar inputs show.

$$\begin{array}{c} i = [(\texttt{c?}_0, 1, \textsf{H}), (\texttt{c?}_1, 0, \textsf{L})]\\ \sim^{\textsf{ID}}_{\textsf{L}}\\ i' = [(\texttt{c?}_0, 2, \textsf{H}), (\texttt{c?}_1, 0, \textsf{L})]\end{array}$$

Let $t = [\![p]\!]_{cs,\mu_0}$, the reactive behaviour obtained from p and μ_0 in cs. Then, we see that t is not ID-secure, since i and i' are ID-similar at level L, but their outputs are not ID-similar at level L.

$$\begin{array}{rcl} (t, i) & \Rightarrow & [\bullet, \bullet, \bullet, \bullet, (\texttt{c!}_0, 1, \textsf{L})]\\ & & \qquad\qquad \not\sim^{\textsf{ID}}_{\textsf{L}}\\ (t, i') & \Rightarrow & [\bullet, \bullet, \bullet, \bullet, (\texttt{c!}_0, 2, \textsf{L})]\end{array}$$

Nevertheless, all inputs i are ID-secure for t (Definition 3.5). The key observation here is that t diverges for $i_{\blacktriangleright \textsf{L}}$ (which coincides with $i'_{\blacktriangleright \textsf{L}}$), since r was initially zero. In other words, the input colist without events on channel $\texttt{c?}_0$ produces an output which is silent and infinite, and hence ID-similar to every other output.

4. Monitoring Reactive Behaviours

In this section, we present the main contribution of the paper: a monitor for reactive behaviours. The monitor is based on a multi-execution of a reactive behaviour: for each security level in $\ell \in \mathcal{L}$, a modified reactive behaviour is created, named the *producer* at level ℓ. A producer at level ℓ is fed only input events visible at security level ℓ, and it produces outputs only at level ℓ. Therefore, it is easy to see that each producer cannot leak secrets, since it never has any secrets available.

In order to detect if a run is secure or not, the following procedure is followed: if the original program produces an output event at security level ℓ, then:

- only the producer at level ℓ can produce an output at such security level;
- if such producer outputs the same event, it is safe, since producers do not leak secrets;
- if such producer outputs a different event, then the run is unsafe.

Hence by the coordinated execution of the original program and the producers, one can replicate the behaviour of the original program for as long as the program is secure, and detect when a run becomes insecure.

$$\begin{array}{lll}
\textsf{prd} & : & \forall \ell, cs.\ \textsf{React}\ cs \rightarrow \textsf{React}\ cs_{\downarrow\ell}\\
\textsf{prd}_{\ell,cs}\ (\textsf{Read}\ f) & = & \textsf{Read}\ (\lambda e.\ \textsf{prd}_{\ell,cs}\ (f\ e))\\
\textsf{prd}_{\ell,cs}\ (\textsf{Write}\ e\ t) & = & \textsf{if}\ \ell = \textsf{lvl}(e)\\
 & & \quad \textsf{then Write}\ e\ (\textsf{prd}_{\ell,cs}\ t)\\
 & & \quad \textsf{else Step}\ (\textsf{prd}_{\ell,cs}\ t)\\
\textsf{prd}_{\ell,cs}\ (\textsf{Step}\ t) & = & \textsf{Step}\ (\textsf{prd}_{\ell,cs}\ t)\\
\textsf{prd}_{\ell,cs}(\textsf{New}\ (c,\ell')\ t) & = & \textsf{if}\ \ell' \sqsubseteq \ell\\
 & & \quad \textsf{then New}\ (c,\ell')\ (\textsf{prd}_{\ell,cs\oplus(c,\ell')}\ t)\\
 & & \quad \textsf{else Step}\ (\textsf{prd}_{\ell,cs\oplus(c,\ell')}\ t)\\
\textsf{prd}_{\ell,cs}(\textsf{Close}\ c\ t) & = & \textsf{if}\ cs\ c \sqsubseteq \ell\\
 & & \quad \textsf{then Close}\ c\ (\textsf{prd}_{\ell,cs\setminus\{(c,\ell)\}}\ t)\\
 & & \quad \textsf{else Step}\ (\textsf{prd}_{\ell,cs\setminus\{(c,\ell)\}}\ t)\\
\textsf{prd}_{\ell,cs}(\textsf{Stop}) & = & \textsf{Stop}
\end{array}$$

Figure 5. Producer creation at level ℓ

4.1 Creation of Producers

A producer at level ℓ is a modification of a reactive behaviour in such a way that it only outputs events at level ℓ. Moreover, the channel system of the producer at level ℓ is restricted to channels of at most level ℓ. More precisely, the *restriction of a channel system cs* to level ℓ is given by:

$$cs_{\downarrow\ell} = \{(c, \ell') \in cs \mid \ell' \sqsubseteq \ell\}$$

The creation of producers is defined in Figure 5. When the original behaviour does a Read, the producer reads the event and continues execution with the producer corresponding to the reaction to that event. Because the channel system of a producer is restricted to ℓ, the security level of this input event is necessarily less or equal to ℓ. In the case of a Write, the producer will perform it only when the level of the channel is ℓ, and do a silent step otherwise. Silent steps (Step) and stops (Stop) are simply copied. The creation (New) or closure (Close) of a channel will be replicated, as long as it is compatible with the restricted channel system (i.e. the security level ℓ' of the channel being created or closed is such that $\ell' \sqsubseteq \ell$.) On the contrary, if the security level of the channel being created or closed is not compatible with the restricted channel system, a silent step is generated and the modification to the channel system is ignored.

Note how the type of the transformation ensures that initially the reactive behaviour does not has access to secrets since only channels with public events are open, and how this invariant is preserved by the function prd which creates producers. Also, it is easy to see that a producer at security level ℓ will only perform Writes at level ℓ by simple inspection of the definition of prd.

4.2 Small-step Semantics of Reactive Behaviours

The monitor works by analising the next step that would be performed by a reactive behaviour and comparing it with a producer. In order to be able to refer easily to the next step of a behaviour we introduce a small-step execution

$$\text{sR}_1\ \frac{e \in E_I(cs)}{(cs, \mathsf{Read}\ f, e :: i) \xrightarrow{\bullet} (cs, f\ e, i)}$$

$$\text{sR}_2\ \frac{e \notin E_I(cs)}{(cs, \mathsf{Read}\ f, e :: i) \xrightarrow{\bullet} (cs, \mathsf{Read}\ f, i)}$$

$$\text{sW}\ \frac{}{(cs, \mathsf{Write}\ e\ t, i) \xrightarrow{e} (cs, t, i)}$$

$$\text{sS}\ \frac{}{(cs, \mathsf{Step}\ t, i) \xrightarrow{\bullet} (cs, t, i)}$$

$$\text{sN}\ \frac{cs' = cs \oplus (c, \ell)}{(cs, \mathsf{New}\ (c, \ell)\ t, i) \xrightarrow{\bullet} (cs', t, i)}$$

$$\text{sC}\ \frac{cs(c) = \ell \qquad cs' = cs \setminus \{(c, \ell)\}}{(cs, \mathsf{Close}\ c\ t, i) \xrightarrow{\bullet} (cs', t, i)}$$

Figure 6. Small step semantics for React

relation for React. The rules in Figure 6 define a relation $\longrightarrow\ \subseteq \mathsf{Conf} \times \mathcal{E}_{O\cup\{\bullet\}} \times \mathsf{Conf}$, where a configuration is a (dependent) tuple $\mathsf{Conf} = (cs : \mathsf{CS}) \times \mathsf{React}\ cs \times \mathcal{I}$ consisting of a channel system, a reactive behaviour for that channel system, and a colist of input events. As usual, we write $\sigma \xrightarrow{e} \sigma'$ for $(\sigma, e, \sigma') \in \longrightarrow$.

Intuitively, a Read is only actually performed when the next input event is compatible (sR$_1$). If the event is incompatible it is discarded without reading (sR$_2$). A Write $e\ t$ outputs an event e and continues executing t (sW). A Step t performs a silent step and continues with t (sS). A New $(c, \ell)\ t$ performs a silent step and continues with t on a channel system extended with c open at level ℓ (sN). A Close $c\ t$ performs a silent step and continues with t on a channel system with channel c closed.

There are two ways in which a configuration may be final: when doing a Read on an empty input colist, and on a Stop. The predicates on configurations end$_\bullet$ and end$_\circ$ capture these two situations.

$$\frac{}{\mathsf{end}_\bullet(cs, \mathsf{Read}\ f, [])} \qquad \frac{}{\mathsf{end}_\circ(cs, \mathsf{Stop}, i)}$$

4.3 A Precise Monitor

We define the monitor for reactive behaviours by a family of evaluation relations $\Downarrow_{cs}$ indexed by a channel system. Like the evaluation relation in Figure 1, the monitor evaluation relation tell us which outputs a reactive behaviour might generate given a certain input. However, the monitor may also rise an alarm if it detects an insecure run. More precisely the monitor is given by a relation

$$\Downarrow_{cs} \subseteq \mathsf{React}\ cs \times \mathcal{I} \times (\mathcal{L} \to \mathsf{Conf}) \times \mathcal{O}_\epsilon$$

where $\mathcal{O}_\epsilon$ is similar to the colist of output events $\mathcal{O}$, but it may also end with an alarm ϵ. Formally, we coinductively define $\mathcal{O}_\epsilon$ as follows.

$$\mathcal{O}_\epsilon = [] \mid \epsilon \mid (\mathcal{E}_O \cup \{\bullet, \circ\}) :: \mathcal{O}_\epsilon$$

We write $(t, i, \overline{\sigma}) \Downarrow_{cs} o$ for $(t, i, \overline{\sigma}, o) \in \Downarrow_{cs}$ indicating that the reactive behaviour t, with input i, and producers $\overline{\sigma}$ (we have one configuration for each producer at each security level, hence the vector notation) produce an output o.

The monitor evaluation relation is given by the rules in Figure 7. Rule SIL indicates it is always safe for a reactive behaviour to perform a silent step. Note that in doing the silent step the channel system may change, as it happens when opening or closing channels (see Fig. 6.) The other rules concern the case where a reactive behaviour wants to perform an operation which might produce a visible event, in which case the monitor needs to check that the corresponding producer outputs the same event. The most obvious case of visible event is when a reactive behaviour wants to output an event by doing a Write (rules VIS$_{1-4}$.) Rule VIS$_1$ applies when the event to be written coincides with the output of the producer at the corresponding security level. In this case, the output is safe and the event is outputted. It might happen that the producer needs to do some silent steps before it outputs an event. Rule VIS$_2$ takes care of this case by advancing the computation on the producer. If the producer outputs an event, but it does not coincide with the one the reactive behaviour wants to output, we are in presence of an insecure run and an alarm is raised (rule VIS$_3$.) If the producer configuration at the event level is final, then it cannot produce such an event and an alarm is raised (rule VIS$_4$.) Finally, the last two rules (END$_1$ and END$_2$) concerns the case where the reactive behaviour would finish executing. Before concluding that finishing would not leak information, the monitor needs to make sure that every producer finishes silently. Since we distinguish between finishing by emptying the input colist and finishing by a Stop (see Fig. 1) we need to treat the two cases differently. Therefore we need two rules with two different functions (done and stop) that will execute producers until they finish in the appropriate manner, or until they produce some visible output, raising an alarm. Note that even if the reactive behaviour terminates, some producer may diverge, and hence, the monitor may diverge. As we will see in the next section, this may only happen if the run is insecure.

The following example shows why, even if the reactive behaviour was going to end, rules END$_1$ and END$_2$ are needed.

Example 4.1. Consider the following program:

```
p =  c?₀(x){ r := x };

     c?₁(x){ if r = 0
               {out(c!₀,1)}
               {skip} };
```

$$\text{SIL}\ \frac{(cs,t,i) \xrightarrow{\bullet} (cs',t',i') \qquad (t',i',\overline{\sigma}) \Downarrow_{cs'} o}{(t,i,\overline{\sigma}) \Downarrow_{cs} \bullet :: o}$$

$$\text{VIS}_1\ \frac{\begin{array}{c}\mathsf{lvl}(e)=\ell \quad \overline{\sigma}(\ell) \xrightarrow{e} \sigma' \\ (t,i,\overline{\sigma}[\ell \mapsto \sigma']) \Downarrow_{cs} o\end{array}}{(\mathsf{Write}\ e\ t,i,\overline{\sigma}) \Downarrow_{cs} e :: o}$$

$$\text{VIS}_2\ \frac{\begin{array}{c}\mathsf{lvl}(e)=\ell \qquad \overline{\sigma}(\ell) \xrightarrow{\bullet} \sigma' \\ (\mathsf{Write}\ e\ t,i,\overline{\sigma}[\ell \mapsto \sigma']) \Downarrow_{cs} o\end{array}}{(\mathsf{Write}\ e\ t,i,\overline{\sigma}) \Downarrow_{cs} \bullet :: o}$$

$$\text{VIS}_3\ \frac{\begin{array}{c}\mathsf{lvl}(e)=\ell \quad \overline{\sigma}(\ell) \xrightarrow{e'} \sigma' \\ e' \neq \bullet \qquad\quad e' \neq e\end{array}}{(\mathsf{Write}\ e\ t,i,\overline{\sigma}) \Downarrow_{cs} \epsilon}$$

$$\text{VIS}_4\ \frac{\mathsf{lvl}(e)=\ell \quad (\mathsf{end}_\bullet(\overline{\sigma}(\ell)) \vee \mathsf{end}_\circ(\overline{\sigma}(\ell)))}{(\mathsf{Write}\ e\ t,i,\overline{\sigma}) \Downarrow_{cs} \epsilon}$$

$$\text{END}_1\ \frac{}{(\mathsf{Read}\ f,[\,],\overline{\sigma}) \Downarrow_{cs} \mathsf{done}(\overline{\sigma})}$$

$$\text{END}_2\ \frac{}{(\mathsf{Stop},i,\overline{\sigma}) \Downarrow_{cs} \mathsf{stop}(\overline{\sigma})}$$

Figure 7. Monitor for Reactive Behaviours

Let t be the reactive behaviour obtained from it for the initial memory μ_0 and channel system

$$cs = \{(\mathtt{c?}_0, \mathsf{H}), (\mathtt{c?}_1, \mathsf{L}), (\mathtt{c!}_0, \mathsf{L})\}$$

and let $i = [(\mathtt{c?}_0, 1, \mathsf{H}), (\mathtt{c?}_1, 0, \mathsf{L})]$. The evaluation of t with input i results in a finite sequence of invisible events.

$$(t,i) \Rightarrow_{cs} [\bullet,\bullet,\bullet,\bullet,\bullet]$$

However, the evaluation of t when fed with the restriction of i at level L, i.e. $i_{\blacktriangleright_\mathsf{L}} = [(\mathtt{c?}_1, 0, \mathsf{L})]$, has an observable event.

$$(t, i_{\blacktriangleright_\mathsf{L}}) \Rightarrow_{cs} [\bullet,\bullet,(\mathtt{c!}_0, 1, \mathsf{L})]$$

Therefore, the outputs are distinguishable at level L, and we conclude that i is not ID-secure for t.

The above example illustrates the need to run all producers and check that they terminate in the same manner as the reactive behaviour. In the monitor, this check is performed by the following functions.

$$\mathsf{done}(\overline{\sigma}) = \begin{cases} [\,] & \text{if } \forall\ell.\ \mathsf{end}_\bullet(\overline{\sigma}(\ell)) \\ \epsilon & \text{if } \exists\ell.\ \mathsf{isWrite}(\overline{\sigma}(\ell)) \\ & \qquad \vee\ \mathsf{end}_\circ(\overline{\sigma}(\ell)) \\ \bullet :: \mathsf{done}(\mathsf{next}(\overline{\sigma})) & \text{otherwise} \end{cases}$$

$$\mathsf{stop}(\overline{\sigma}) = \begin{cases} \epsilon & \text{if } \exists\ell.\ \mathsf{isWrite}(\overline{\sigma}(\ell)) \\ & \qquad \vee\ \mathsf{end}_\bullet(\overline{\sigma}(\ell)) \\ \bullet :: \mathsf{stop}(\mathsf{next}(\overline{\sigma})) & \text{otherwise} \end{cases}$$

The predicate on configurations isWrite tell us if a configuration is a Write:

$$\frac{}{\mathsf{isWrite}(cs, \mathsf{Write}\ e\ t, i)}$$

Function next advances each producer a silent step (when possible):

$$\begin{array}{lcl} \mathsf{next} & : & (\mathcal{L} \to \mathsf{Conf}) \to (\mathcal{L} \to \mathsf{Conf}) \\ \mathsf{next}(\overline{\sigma})(\ell) & = & \begin{cases} \sigma' & \text{if } \overline{\sigma}(\ell) \xrightarrow{\bullet} \sigma' \\ \overline{\sigma}(\ell) & \text{otherwise.} \end{cases} \end{array}$$

Given a channel system cs, a reactive behaviour t, and an input colist i, the monitor is initialised in the following manner

$$\mathsf{monitor}(cs,t,i) = \big(t, i, \lambda\ell.\ (cs_{\downarrow\ell}, \mathsf{prd}_{\ell,cs}(t), i)\big)$$

That is, producers are created with function prd in a restricted channel system. Since inputs on closed channels are ignored there is no need to filter the input colist of each producer.

5. Properties of the Monitor

Ideally, a monitor should be precise and transparent. Precision means that the monitor should raise an alarm only for insecure runs. Transparency, on the other hand, means that secure runs should be indistinguishable from an ordinary execution (i.e. with the monitor not present.)

In order to state these properties formally, we define the predicate $\mathsf{ok} \subseteq \mathcal{O}_\epsilon$ which identifies output colists where the monitor has not raised an alarm. The predicate is defined coinductively by the following rules.

$$\frac{}{\mathsf{ok}([\,])} \qquad\qquad \frac{\mathsf{ok}(s)}{\mathsf{ok}(e :: s)}$$

The monitor for reactive behaviours of the previous section is precise with respect to ID-security.

Theorem 5.1 (Precision for ID-secure runs)**.** *Let cs be a channel system, t* : React *cs a reactive behaviour and i an input colist such that* $\mathsf{monitor}(cs,t,i) \Downarrow_{cs} o$*. Then,*

$$i \textit{ is ID-secure for } t \iff \mathsf{ok}(o)$$

Moreover, the monitor is transparent with respect to both ID- and CP-security.

Theorem 5.2 (Transparency for secure runs)**.** *Let* cs : CS *be a channel system,* t : React *cs a reactive behaviour on cs, and i an input colist, such that* $(t,i) \Rightarrow_{cs} o$ *and* $\mathsf{monitor}(cs,t,i) \Downarrow_{cs} o'$*. Then,*

$$i \textit{ is ID-secure for } t \implies \mathsf{ok}(o') \wedge \forall\ell.\ o \sim_\ell^{\mathsf{ID}} o', \quad (1)$$

and

$$i \textit{ is CP-secure for } t \implies \mathsf{ok}(o') \wedge \forall\ell.\ o \sim_\ell^{\mathsf{CP}} o'. \quad (2)$$

However, the monitor is not precise with respect to CP-security: it could happen that an insecure run causes the monitor to diverge instead of raising an alarm.

As a corollary of Theorem 5.2, Lemma 3.10, and Lemma 3.9, we have the following result on behaviours.

Corollary 5.3 (Transparency for secure behaviours)**.** *Let cs be a channel system, t an (ID/CP)-non-interferent reactive behaviour on cs, and i an input colist i. Then*

$$\left.\begin{array}{r}(t,i) \Rightarrow_{cs} o \\ \mathsf{monitor}(cs,t,i) \Downarrow_{cs} o'\end{array}\right\} \Longrightarrow o \sim^{\mathsf{ID}/\mathsf{CP}} o'.$$

Therefore, the monitor will always preserve the semantics of secure programs.

6. Related Work

There are many security policies enforceable by execution monitoring [16, 22, 32], but non-interference is not one of them [25, 32]. Therefore, monitors often enforce properties stronger than non-interference [1, 3, 4, 29, 31] losing precision, and hence raising false alarms.

Our approach to monitoring is an extension of the monitor by Zanarini, Jaskelioff, and Russo [39]. The idea of transforming programs according to their security level is inspired by secure multi-execution as introduced by Devriese and Piessens [15] for interactive systems. However, the idea of having different execution threads according to their security level had been used before [9, 11]. Barthe et al. [5] show that secure multi-execution can also be achieved by code transformation. Rafnsson and Sabelfeld [28] make several improvements to secure multi-execution of interactive systems, such as lifting the totality assumptions on input channels and distinguishing between presence and content of messages. Differently from the above mentioned works which focused on interactive systems, Bielova et al. [6] adapt secure multi-execution for reactive systems and, similarly to this work, consider security in terms of runs.

Zheng and Myers [40] consider dynamic security labels for a statically typed language. Nevertheless, none of the works based on secure multi-execution described above [5, 6, 15, 28, 39] consider dynamic channels.

De Groef et al. [12] implement FlowFox, a web browser that supports an information flow control mechanism based on SME. The actual implementation of FlowFox allows for policies that assign labels to API method calls based on a state maintained by the security policy. Hence, the policy can record in its state which channels are open. However, this support for dynamic channels is not present in the formal model of FlowFox [13], and hence no formal guarantees are provided when using this feature.

The notions for reactive systems of ID- and CP-non-interference were introduced by Bohannon et al. [7]. As opposed to ID-security, the notion of CP-security is progress-sensitive. However, it is more difficult to enforce. Rafnsson and Sabelfeld [27] show that ID- and CP-security are termination-insensitive, and therefore susceptible to brute force attacks that exploit the termination channel. Our proposal, similarly to other modern information-flow tools [10, 26, 33] cannot avoid this kind of leaks. However, for deterministic systems, the bandwidth of leaking information by exploiting outputs in combination with termination is logarithmic in the size of the secret [2] and can be reduced by applying buffering techniques [27].

Our model of reactive behaviours is an extension of the description of interactions in [21] which in turn is inspired by [34]. In order to cope with dynamic channels it has been extended to use the expressive power of interaction structures [17].

7. Conclusions

We propose a monitor for reactive systems in which channels can be opened and closed dynamically. Our approach extends the multi-execution monitor of Zanarini, Jaskelioff, and Russo [39] which could only handle a fixed set of channels. Due to the generality of the approach in [39] and a careful choice of how to extend it, we were able to adapt many notions to our more complex setting of dynamic channels with only slight modifications. The main issue to consider was how to model non-interference. Once we decided that the right way was to pair each event with a security level, the rest of the design decisions followed up naturally. Interestingly, we obtained similar results as the previous monitor: our monitor is precise with respect to ID-security and transparent with respect to both ID and CP-security.

The addition of dynamic channels means that a channel may be closed and then opened at a different security level, and therefore we need to treat security levels dynamically. This opens the door for an approach to declassification: by adding a declassification primitive, we may allow the temporary change of the security level of a channel and still be able to provide formal security guarantees.

Finally, we would like to implement the monitor in a real browser and develop some applications for it. This would provide a reality check for our model of reactive systems and suggest areas where improvement is needed.

Acknowledgments

We thank the anonymous reviewers for their constructive criticism, and Exequiel Rivas and Alejandro Russo for their insightful comments and generous help. This work was funded by the *Agencia Nacional de Promoción Científica y Tecnológica* (PICT 2009-15) and *Consejo Nacional de Investigaciones Científicas y Técnicas* (CONICET).

References

[1] A. Askarov and A. Sabelfeld. Tight Enforcement of Information-Release Policies for Dynamic Languages. In *Proceedings of the 22nd IEEE Computer Security Foundations*

Symposium, Washington, DC, USA, 2009. IEEE Computer Society.

[2] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08. Springer-Verlag, 2008.

[3] T. H. Austin and C. Flanagan. Efficient Purely-Dynamic Information Flow Analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.

[4] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10. ACM, 2010.

[5] G. Barthe, J. M. Crespo, D. Devriese, F. Piessens, and E. Rivas. Secure multi-execution through static program transformation. In *Formal Techniques for Distributed Systems (FMOODS/FORTE 2012)*, June 2012.

[6] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *Proceedings of the 5th International Conference on Network and System Security (NSS 2011)*, Sept. 2011.

[7] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09. ACM, 2009.

[8] S. B. Byrne, R. S. Sutor, J. Robie, G. Nicol, M. Champion, S. Isaacson, I. Jacobs, L. Wood, C. Wilson, and A. L. Hors. Document Object Model (DOM) Level 1. W3C recommendation, W3C, Oct. 1998. http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001.

[9] R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. P. Sistla. Preventing Information Leaks through Shadow Executions. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08. IEEE Computer Society, 2008.

[10] R. Chapman and A. Hilton. Enforcing security and safety models with an information flow analysis tool. In *Proc. of the 2004 annual ACM SIGAda international conference on Ada*. ACM, 2004.

[11] M. Cristiá and P. Mata. Runtime Enforcement of Noninterference by Duplicating Processes and their Memories. In *Workshop de Seguridad Informática WSEGI 2009, Argentina*, 38 JAIIO, 2009.

[12] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12. ACM, 2012.

[13] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Secure multi-execution of web scripts: Theory and practice. *Journal of Computer Security*, 22(4):469–509, 2014. ISSN 0926-227X. . URL `https://lirias.kuleuven.be/handle/123456789/442492`.

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

[15] D. Devriese and F. Piessens. Noninterference through Secure Multi-execution. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, SP '10. IEEE Computer Society, 2010.

[16] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, Jan. 2006.

[17] P. Hancock and A. Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic. 14th international workshop, CSL 2000*, Springer Lecture Notes in Computer Science, Vol. 1862, pages 317 – 331, 2000.

[18] D. Hedin and A. Sabelfeld. Information-Flow Security for a Core of JavaScript. In *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE Computer Society, 2012.

[19] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06. ACM, 2006.

[20] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10. ACM, 2010.

[21] M. Jaskelioff and A. Russo. Secure multi-execution in Haskell. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2011.

[22] J. Ligatti, L. Bauer, and D. Walker. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *International Journal of Information Security*, Feb. 2005.

[23] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan. AdJail: practical enforcement of confidentiality and integrity policies on web advertisements. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, Berkeley, CA, USA, 2010. USENIX Association.

[24] J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10. ACM, 2010.

[25] J. McLean. A General Theory of Composition for Trace Sets Closed under Selective Interleaving Functions. In *Proceedings of the 1994 IEEE Symposium on Security and Privacy*, SP '94. IEEE Computer Society, 1994.

[26] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java Information Flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[27] W. Rafnsson and A. Sabelfeld. Limiting information leakage in event-based communication. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, PLAS '11. ACM, 2011. ISBN 978-1-4503-0830-4.

[28] W. Rafnsson and A. Sabelfeld. Secure Multi-execution: Fine-Grained, Declassification-Aware, and Transparent. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, pages 33–48, June 2013.

[29] A. Russo and A. Sabelfeld. Securing Timeout Instructions in Web Applications. In *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE Computer Society, July 2009.

[30] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[31] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 0 of *LNCS*. Springer-Verlag, June 2009.

[32] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 2000.

[33] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible Dynamic Information Flow Control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011.

[34] W. Swierstra and T. Altenkirch. Beauty in the Beast: A Functional Semantics of the Awkward Squad. In *Haskell '07: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 25–36, 2007.

[35] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proc. Network and Distributed System Security Symposium*, Feb. 2007.

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

[37] Williams and D. Wichers. OWASP Top 10 2010. `http://www.owasp.org/index.php/Top\_10\_2010`, 2010.

[38] D. Zanarini and M. Jaskelioff. Monitoring Reactive Systems with Dynamic Channels: Extended Version, 2014. Available at `http://www.fceia.unr.edu.ar/~dante`.

[39] D. Zanarini, M. Jaskelioff, and A. Russo. Precise Enforcement of Confidentiality for Reactive Systems. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, pages 18–32, June 2013.

[40] L. Zheng and A. C. Myers. Dynamic security labels and noninterference (extended abstract). In T. Dimitrakos and F. Martinelli, editors, *Formal Aspects in Security and Trust*, volume 173 of *IFIP International Federation for Information Processing*, pages 27–40. Springer US, 2005. ISBN 978-0-387-24050-3.