Information Flow Control for Web Scripts

Willem De Groef, Dominique Devriese, Mathy Vanhoef, and Frank Piessens

iMinds-DistriNet,
KU Leuven,
3001 Leuven, Belgium.
`firstname.lastname@cs.kuleuven.be`

Abstract. Modern web applications heavily rely on JavaScript code executing in the browser. These web scripts are useful for instance for improving the interactivity and responsiveness of web applications, and for gathering web analytics data. However, the execution of server-provided code in the browser also brings substantial security and privacy risks. Web scripts can access a fair amount of sensitive information, and can leak this information to anyone on the Internet. This tutorial paper discusses information flow control mechanisms for countering these threats. We formalize both a static, type-system based and a dynamic, multi-execution based enforcement mechanism, and show by means of examples how these mechanisms can enforce the security of information flows in web scripts.

Keywords: web scripts, JavaScript, security, information flow control

1 Introduction

Modern interactive web applications heavily rely on browser-side scripts in languages such as JavaScript, for instance to propose completions while a user is typing into a text field. These scripts are usually event-driven programs that can react to user interface events such as key presses or mouse clicks, or to network events such as the arrival of HTTP responses. While handling events, scripts can display output to the user or send output on the network in the form of HTTP requests.

Listing 1.1 shows a simplified example of such a program that interactively proposes possible completions for a string that the user is typing into a textfield.

The first three lines declare an event handler for the *key up* event. That handler takes the current contents of the textfield with ID *field1*, and invokes a helper function that computes the possible completions (for instance by contacting a remote server). These possible completions are finally displayed in the text area with ID *suggestions*.

While browser-side scripts are very useful for building responsive interactive web applications, they also come with substantial security and privacy risks. Scripts have, and *need*, access to both user information and to remote HTTP servers. The completion example above can only perform its function if it can

Listing 1.1. Suggesting completions

```
window.onkeyup = function(e) {
    suggestions.value = completions(field1.value);
  }

function completions(s) {
// return possible completions of s
}
```

read what the user is typing, and if it can contact the remote server to retrieve possible completions. Unfortunately, a consequence of these capabilities of scripts is that they are commonly used to leak private information to untrusted network servers [18, 26]. To illustrate these risks, Listing 1.2 shows an example of a script that implements a simple key logger in JavaScript. It installs an event handler to monitor key presses, and leaks every keystroke to `hacker.com`, using the jQuery ajax() function that sends an HTTP request to the URL provided as a parameter. The similarity of this example with the earlier completion example shows that it is a thin line between useful and dangerous scripts. The fact that many web sites include scripts from third parties [25] further amplifies the need for protective countermeasures.

Listing 1.2. Keylogger

```
var u = 'http://hacker.com/?=';
window.onkeypress = function(e) {
  var leak = e.charCode;
   $.ajax(u + leak);
}
```

Researchers have realized that mechanisms for information flow security are a promising countermeasure for web script-related threats, since such mechanisms allow the scripts to have access to private information but at the same time prevent it from leaking that information to untrusted servers.

Information flow security can be enforced *statically* or *dynamically*. The purpose of this tutorial article is to explain the essence of two techniques for enforcing information flow security for event-driven programs: a static technique based on typing, and a dynamic technique based on secure multi-execution.

The remainder of this tutorial paper is structured as follows. First, in Section 2, we define a formal scripting language that is a simple model of JavaScript. Then, in Section 3, we define information flow control and give both examples of scripts that are information flow secure and scripts that are insecure. Sections 4 and 5 define a static, respectively dynamic enforcement mechanism for information flow security and illustrate the mechanisms by means of examples. Section 6 provides a brief overview of the existing research in this area, and Section 7 concludes.

2 Formal Model of Web Scripts

2.1 Syntax

For the purpose of this tutorial paper, we use a very simple model of a web scripting language, strongly inspired by the model language introduced by Bohannon et al. [8]. The syntax is specified in Figure 1. We assume certain given disjoint sets of identifiers: *GVars* is the set of identifiers for mutable global variables, *Chan* is the set of identifiers for communication channel names, and *Var* is the set of identifiers for bound variables.

$$
\begin{array}{rrr}
n \in & \mathbb{Z} & \text{(Integer number literals)} \\
\odot \in & \{+, -, =, <\} & \text{(Primitive operations on numbers)} \\
r \in & \mathit{GVars} & \text{(Global mutable variables)} \\
ch \in & \mathit{Chan} & \text{(I/O Channels)} \\
x \in & \mathit{Var} & \text{(Variables)} \\
p ::= & h; p & \text{(Reactive Programs)} \\
\mid & \cdot & \\
h ::= & \texttt{on}\ ch(x)\ c & \text{(Event handlers)} \\
c ::= & \texttt{skip} & \text{(Commands)} \\
\mid & c; c & \\
\mid & r := e & \\
\mid & \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c & \\
\mid & \texttt{while}\ e\ \texttt{do}\ c & \\
\mid & \texttt{output}\ e\ \texttt{on}\ ch & \\
e ::= & n & \text{(Expressions)} \\
\mid & r & \\
\mid & x & \\
\mid & e \odot e &
\end{array}
$$

Fig. 1. Formal syntax of our web scripting language

A program p is essentially a list of handlers, where each handler h specifies a command c to be executed on occurrence of an input event on channel ch. An input event always carries a single integer value, and that integer value is bound to the formal parameter x before the command c is executed. The syntax of commands is standard, with syntactic forms for the empty command (*skip*), sequential composition, assignment to global variables, conditional and looping constructs, and performing output on a channel. Expressions e are standard integer arithmetic expressions that can refer to formal parameters x declared in a handler definition, or to global variables r.

In examples we will assume the existence of I/O channels such as *KeyPress*, *Network*, *Display*, *MouseClick*, ... Scripts can output integers on these channels with the `output` command. For instance, `output` 10 `on` *Network* will output the integer 10 on the *Network* channel. They can react to inputs arriving on these channels by declaring event handlers. For instance, key presses are modeled as input events on the *KeyPress* channel carrying a single integer value that represents the scan code of the key that was pressed. For simplicity, we assume that all input events carry a single integer value, and that all output events are outputs of a single integer value.

As an example, the JavaScript key logger program from Listing 1.2 is rendered in our model language as:

$$\texttt{on}\ \mathit{KeyPress}(x)\ \texttt{output}\ x\ \texttt{on}\ \mathit{Network}$$

This script declares an event handler that upon each key press sends the key scan code on the network.

If we want to distinguish different network destinations (for instance communication to the same network origin the web page was loaded from and other network origins) we can use two separate channel identifiers *Network* and *SameOriginNetwork*. The completions example from Listing 1.1 could then be rendered as:

$$\texttt{on}\ \mathit{KeyPress}(x)\ \texttt{output}\ x\ \texttt{on}\ \mathit{SameOriginNetwork}$$

We could even use parameterized channel names such as $\mathit{Network}(o)$ with o an *origin* of the form `http://www.kuleuven.be` for instance.

2.2 Semantics

To define the semantics of the model language, we define stores μ (assigning a current (integer) value to all global variable names) and outputs o (either the special "no output" constant $\bullet$ or an output of a number n on channel ch ($\texttt{out}_{ch(n)}$) (Figure 2). For updating of the store, we use the notation $\mu[r \mapsto n]$: it denotes the store equal to μ except that the global variable r now maps to the value n.

Using stores, we define a big-step operational semantics judgement for expressions $\mu \vdash e \downarrow n$ (in store μ, expression e evaluates to value n). This definition is completely standard (Figure 3).

Programs are event-driven. The judgement $(p)(i) \Downarrow c$ defines formally what command c program p will execute to handle the input event i. It is defined by the rules in Figure 4. Essentially, this looks up the handler for handling the input event, and substitutes the integer n received on the input channel for the

$$
\begin{array}{rrr}
\mu \in & \mathit{Ref} \to \mathbb{Z} & \text{(Stores)} \\
o ::= & \bullet & \text{(Outputs)} \\
\mid & \mathtt{out}_{ch(n)} & \\
i ::= & \mathtt{in}_{ch(n)} & \text{(Input Events)} \\
ev ::= & i \mid o & \text{(Reactive Events)}
\end{array}
$$

Fig. 2. Semantic structures

$$\frac{}{\mu \vdash n \downarrow n}\ (\text{E-Expr-Lit}) \qquad \frac{}{\mu \vdash r \downarrow \mu(r)}\ (\text{E-Expr-Ref})$$

$$\frac{\mu \vdash e_1 \downarrow n_1 \qquad \mu \vdash e_2 \downarrow n_2}{\mu \vdash e_1 + e_2 \downarrow n_1 + n_2}\ (\text{E-Expr-Plus}) \qquad \frac{\mu \vdash e_1 \downarrow n_1 \qquad \mu \vdash e_2 \downarrow n_2}{\mu \vdash e_1 - e_2 \downarrow n_1 - n_2}\ (\text{E-Expr-Minus})$$

$$\frac{\mu \vdash e_1 \downarrow n_1 \qquad \mu \vdash e_2 \downarrow n_2 \qquad n_1 \neq n_2}{\mu \vdash e_1 = e_2 \downarrow 0}\ (\text{E-Expr-Eq1})$$

$$\frac{\mu \vdash e_1 \downarrow n_1 \qquad \mu \vdash e_2 \downarrow n_2 \qquad n_1 = n_2}{\mu \vdash e_1 = e_2 \downarrow 1}\ (\text{E-Expr-Eq2})$$

$$\frac{\mu \vdash e_1 \downarrow n_1 \qquad \mu \vdash e_2 \downarrow n_2 \qquad n_1 \not< n_2}{\mu \vdash e_1 < e_2 \downarrow 0}\ (\text{E-Expr-Lt1})$$

$$\frac{\mu \vdash e_1 \downarrow n_1 \qquad \mu \vdash e_2 \downarrow n_2 \qquad n_1 < n_2}{\mu \vdash e_1 < e_2 \downarrow 1}\ (\text{E-Expr-Lt2})$$

Fig. 3. Semantics of expressions

formal parameter x in that handler. If no handler is defined in the program p for input i on this channel, we have $(p)(i) \Downarrow \texttt{skip}$.

$$\frac{}{(\texttt{on}\ ch(x)\ \texttt{do}\ c; p)(\texttt{in}_{ch(n)}) \Downarrow [x \mapsto n]c}$$

$$\frac{(p)(\texttt{in}_{ch(n)}) \Downarrow c \qquad ch \neq ch'}{(\texttt{on}\ ch'(x)\ \texttt{do}\ c'; p)(\texttt{in}_{ch(n)}) \Downarrow c}$$

$$\frac{}{(\cdot)(\texttt{in}_{ch(n)}) \Downarrow \texttt{skip}}$$

Fig. 4. Determining the event handling command

Finally, the semantics of commands is given as a small-step operational semantics judgement $(\mu, c) \xrightarrow{o} (\mu', c')$ (executing command c in store μ produces an updated store μ' and new command c' producing output o). They are defined by the rules in Figure 5.

The *initial program state* is $(\mu_0, \texttt{skip})$ where μ_0 maps all global variables on 0. A program state is *passive* if it has the form $(\mu, \texttt{skip})$. We say a program is *well-formed* if it has no unbound variables (i.e. the only variable $x \in \mathit{Var}$ occurring in the body of a handler is the formal parameter of the handler – of course, the handler can also use global variables $r \in \mathit{GVars}$) . It is straightforward to prove that well-formed programs that are not in a passive state can always make a deterministic step. The only non-deterministic transitions are transitions that consume a new input, and these are only possible from a passive state.
An *execution* of a script is a finite or infinite sequence of events $\overline{ev}$:

$$\overline{ev} = (\mu_0, \texttt{skip}) \xrightarrow{ev_0} (\mu_1, c_1) \xrightarrow{ev_1} (\mu_2, c_2) \xrightarrow{ev_2} \ldots$$

We say an execution is *event-complete* if it ends in a passive state: this means that all the input events the program has received have been fully handled, and that the only way to further extend the execution is by giving it a new input event.

2.3 Examples

Consider the following script:

$$\begin{array}{l}\texttt{on}\ \mathit{KeyPress}(x)\ \mathit{total} := \mathit{total} + x;\\ \texttt{on}\ \mathit{MouseClick}(x)\ \texttt{output}\ \mathit{total}\ \texttt{on}\ \mathit{Display}\end{array}$$

The script keeps a running total of the key scan codes of all key presses it has seen, and on a *MouseClick* input event, it displays the total on the *Display* channel. This models a simple JavaScript calculator.

$$\frac{}{(\mu, (\mathtt{skip}; c)) \overset{\bullet}{\to} (\mu, c)}\ (\textsc{E-Stmt-SeqSkip})$$

$$\frac{(\mu, c_1) \overset{o}{\to} (\mu', c_1')}{(\mu, (c_1; c_2)) \overset{o}{\to} (\mu', (c_1'; c_2))}\ (\textsc{E-Stmt-Seq})$$

$$\frac{\mu \vdash e \downarrow n}{(\mu, (r := e)) \overset{\bullet}{\to} (\mu[r \mapsto n], \mathtt{skip})}\ (\textsc{E-Stmt-Assign})$$

$$\frac{\mu \vdash e \downarrow n \qquad n \neq 0}{(\mu, \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2) \overset{\bullet}{\to} (\mu, c_1)}\ (\textsc{E-Stmt-If1})$$

$$\frac{\mu \vdash e \downarrow n \qquad n = 0}{(\mu, \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2) \overset{\bullet}{\to} (\mu, c_2)}\ (\textsc{E-Stmt-If2})$$

$$\frac{\mu \vdash e \downarrow n \qquad n \neq 0}{(\mu, \mathtt{while}\ e\ \mathtt{do}\ c) \overset{\bullet}{\to} (\mu, (c; \mathtt{while}\ e\ \mathtt{do}\ c))}\ (\textsc{E-Stmt-While1})$$

$$\frac{\mu \vdash e \downarrow n \qquad n = 0}{(\mu, \mathtt{while}\ e\ \mathtt{do}\ c) \overset{\bullet}{\to} (\mu, \mathtt{skip})}\ (\textsc{E-Stmt-While2})$$

$$\frac{\mu \vdash e \downarrow n}{(\mu, \mathtt{output}\ e\ \mathtt{to}\ ch) \xrightarrow{\mathtt{out}_{ch(n)}} (\mu, \mathtt{skip})}\ (\textsc{E-Stmt-Out})$$

$$\frac{(p)(\mathtt{in}_{ch(n)}) \Downarrow c}{(\mu, \mathtt{skip}) \xrightarrow{\mathtt{in}_{ch(n)}} (\mu, c)}\ (\textsc{E-Input})$$

Fig. 5. Semantics of commands

The following is an example execution of the script. We denote a memory μ as the set that has an element $r \mapsto \mu(r)$ for every global variable r that has a non-zero value in μ. Hence, μ_0 is denoted as the empty set $\{\}$.

$$\begin{aligned}
(\{\}, \texttt{skip}) &\xrightarrow{\texttt{in}_{\mathit{KeyPress(10)}}} (\{\}, \mathit{total} := \mathit{total} + 10) \\
&\overset{\bullet}{\rightarrow} (\{\mathit{total} \mapsto 10\}, \texttt{skip}) \\
&\xrightarrow{\texttt{in}_{\mathit{KeyPress(20)}}} (\{\mathit{total} \mapsto 10\}, \mathit{total} := \mathit{total} + 20) \\
&\overset{\bullet}{\rightarrow} (\{\mathit{total} \mapsto 30\}, \texttt{skip}) \\
&\xrightarrow{\texttt{in}_{\mathit{MouseClick(0)}}} (\{\mathit{total} \mapsto 30\}, \texttt{output } \mathit{total} \texttt{ on } \mathit{Display}) \\
&\xrightarrow{\texttt{out}_{\mathit{Display(30)}}} (\{\mathit{total} \mapsto 30\}, \texttt{skip})
\end{aligned}$$

For the remainder of this paper, we will usually not show the silent actions ($\bullet$) in example executions.

As a second example, consider again the key logger script:

$$\texttt{on } \mathit{KeyPress}(x) \texttt{ output } x \texttt{ on } \mathit{Network}$$

The following is an example execution of the script.

$$\begin{aligned}
(\{\}, \texttt{skip}) &\xrightarrow{\texttt{in}_{\mathit{KeyPress(10)}}} (\{\}, \texttt{output } 10 \texttt{ on } \mathit{Network}) \\
&\xrightarrow{\texttt{out}_{\mathit{Network(10)}}} (\{\}, \texttt{skip}) \\
&\xrightarrow{\texttt{in}_{\mathit{KeyPress(20)}}} (\{\}, \texttt{output } 20 \texttt{ on } \mathit{Network}) \\
&\xrightarrow{\texttt{out}_{\mathit{Network(20)}}} (\{\}, \texttt{skip}) \\
&\xrightarrow{\texttt{in}_{\mathit{MouseClick(0)}}} (\{\}, \texttt{skip})
\end{aligned}$$

Key press events get echoed on the network, while a mouse click event is just silently absorbed (there is no handler for these events).

3 Information flow control

3.1 Introduction

Web scripts can receive and send information on a variety of channels.

They need to *receive* information from both sensitive and less sensitive channels. For instance, a script may need to read a password from the user in order to estimate the strength of the password (clearly a sensitive piece of information). Scripts may also need to receive advertisements to be displayed from the network (an example of a script reading less sensitive information).

Scripts also need to *send* information both to trustworthy output channels as well as to less trustworthy output channels. For instance, in a web based document processor, scripts will send document content to the site hosting the document processing application (an output to a trustworthy channel). But scripts may also collect user interaction data to be sent to a web analytics site (outputs to a less trustworthy channel).

The key idea of information flow control is to allow scripts to perform all these inputs and outputs, as long as *no information received from a sensitive input channel leaks to a non-trustworthy output channel.* Let us assume for the sake of the following examples that:

- *Display* is a trustworthy output channel: the outputs on that channel can only be seen by a trusted observer – the user of the web application.
- *Network* is a non-trustworthy output channel: the outputs can possibly be seen by untrusted obervers, for instance attackers.
- *KeyPress* is a sensitive input channel: we do not want untrusted observers to know anything about what keys we press.
- *MouseClick* is a non-sensitive input channel: we do not care that information leaks about when and where we click the mouse.

To enforce information flow control, these assumptions are formalized in a *policy* that assigns a *security label* to each of the channels. These security labels should be thought of as *confidentiality levels.* For the purpose of this paper, we use only two such levels: H for high confidentiality and L for low confidentiality. The set of security labels is an ordered set: for our two element set, $H > L$.

For input channels, the label defines the level of confidentiality of information received on that channel. Hence, in our examples the label of *KeyPress* will be H and the label of *MouseClick* will be L.

For output channels, the label defines the trustworthiness of the observers of the output channel. A H observer is trusted and it is OK if that observer sees information of confidentiality levels H or L. A L observer is untrusted and should only ever see information of level L.

With these intuitions in mind, we can discuss some examples of secure and insecure scripts.

Consider again the JavaScript calculator:

$$\begin{array}{l}\texttt{on}\ \mathit{KeyPress}(x)\ \mathit{total} := \mathit{total} + x;\\ \texttt{on}\ \mathit{MouseClick}(x)\ \texttt{output}\ \mathit{total}\ \texttt{on}\ \mathit{Display}\end{array}$$

This script is secure: it reads sensitive information from *KeyPress* but only discloses it to the trustworthy *Display* channel.

On the other hand, the key logger script:

$$\texttt{on}\ \mathit{KeyPress}(x)\ \texttt{output}\ x\ \texttt{on}\ \mathit{Network}$$

is an example of an insecure script. It discloses information read from a H input channel (*KeyPress*) to a L output channel (*Network*). The variant of the script that outputs these key presses to a trustworthy network channel *SameOriginNetwork* would be secure.

The key logger script above has a blatant leak: it just copies information from a H channel to a L channel and hence is obviously insecure. But it is important to note that scripts can also leak information in more subtle ways. Consider for instance the following script:

```
on $KeyPress(x)$ $r := x$;
on $MouseClick(x)$ output $r$ on $Network$
```

This script leaks information from *KeyPress* to *Network* by first storing the information in memory, and sending it out at a later moment in time. Hence this script is also insecure, but in a somewhat less obvious way.

Leaks can be even more indirect. Consider for instance:

```
on $KeyPress(x)$  if $x = 100$ then $r := 1$ else skip;
on $MouseClick(x)$ output $r$ on $Network$
```

This script leaks whether the user ever pressed a key with scan code 100: it outputs 1 on the *Network* in case it has ever seen a *KeyPress*(100) event. Hence, it is also insecure but in an even more indirect way. Flows of information that, as in the example above, leak information by using the control flow of the program are often called *implicit* flows.

The objective of information flow security is to formalize the distinction between secure and insecure programs that we informally discussed in this section, and to develop enforcement mechanisms that prevent such insecure information leaks. We want to prevent both explicit and implicit flows.

3.2 Formal definitions

The notion of information flow security discussed above can be formalized as *noninterference*, which roughly says that there should not be two executions of the program that (1) receive the same L inputs, but (2) produce different L outputs. The intuition is that if L outputs are always the same given the same L inputs, then the L outputs could not have been influenced in any way by the H inputs, and hence do not leak any information about the H inputs.

To make this formal for web scripts, we need a few definitions. We assume as given a *policy* that assigns security labels to channels in the form of a function *lbl* from *Chan* to $\{L, H\}$.

For a sequence of events $\overline{ev} = ev_1 \cdots ev_n$, we define

- $\lfloor \overline{ev} \rfloor_I$ the subsequence of all input events.
- $\lfloor \overline{ev} \rfloor_O$ the subsequence of all output events.
- $\lfloor \overline{ev} \rfloor_{I,L}$ the subsequence of all input events $\texttt{in}_{ch(n)}$ such that $lbl(ch) = L$.
- $\lfloor \overline{ev} \rfloor_{O,L}$ the subsequence of all output events $\texttt{out}_{ch(n)}$ such that $lbl(ch) = L$.

Definition 1. *A program p is* noninterferent *iff for any two event-complete executions $\overline{ev_1}$ and $\overline{ev_2}$ it holds that:*

$$\lfloor \overline{ev_1} \rfloor_{I,L} = \lfloor \overline{ev_2} \rfloor_{I,L} \implies \lfloor \overline{ev_1} \rfloor_{O,L} = \lfloor \overline{ev_2} \rfloor_{O,L}$$

I.e. any two event-complete executions that receive the same L inputs will produce the same L outputs.

Example 1. The key logger script:

$$\texttt{on}\ \mathit{KeyPress}(x)\ \texttt{output}\ x\ \texttt{on}\ \mathit{Network}$$

is insecure according to this definition, because of the following two event-complete executions.

$$\begin{aligned}\overline{ev_1} = &\quad (\{\}, \texttt{skip}) \xrightarrow{\texttt{in}_{\mathit{KeyPress}(10)}} (\{\}, \texttt{output}\ 10\ \texttt{on}\ \mathit{Network}) \xrightarrow{\texttt{out}_{\mathit{Network}(10)}} (\{\}, \texttt{skip})\\ \overline{ev_2} = &\quad (\{\}, \texttt{skip}) \xrightarrow{\texttt{in}_{\mathit{KeyPress}(20)}} (\{\}, \texttt{output}\ 20\ \texttt{on}\ \mathit{Network}) \xrightarrow{\texttt{out}_{\mathit{Network}(20)}} (\{\}, \texttt{skip})\end{aligned}$$

For these two executions, $\lfloor \overline{ev_1} \rfloor_{I,L} = \lfloor \overline{ev_2} \rfloor_{I,L}$ (both executions have no L input events), but $\lfloor \overline{ev_1} \rfloor_{O,L} \neq \lfloor \overline{ev_2} \rfloor_{O,L}$ (both executions have different L outputs on *Network*).

Example 2. Also the script with the more subtle leaks:

$$\begin{aligned}&\texttt{on}\ \mathit{KeyPress}(x)\ \ \texttt{if}\ x = 100\ \texttt{then}\ r := 1\ \texttt{else}\ \texttt{skip};\\ &\texttt{on}\ \mathit{MouseClick}(x)\ \texttt{output}\ r\ \texttt{on}\ \mathit{Network}\end{aligned}$$

can be seen to be insecure by considering the following two event-complete executions (we do not show the silent output events):

$$\begin{aligned}\overline{ev_1} = (\{\}, \texttt{skip}) &\xrightarrow{\texttt{in}_{\mathit{KeyPress}(10)}} (\{\}, \texttt{skip})\\ &\xrightarrow{\texttt{in}_{\mathit{MouseClick}(10)}} (\{\}, \texttt{output}\ 0\ \texttt{on}\ \mathit{Network})\\ &\xrightarrow{\texttt{out}_{\mathit{Network}(0)}} (\{\}, \texttt{skip})\\ \overline{ev_2} = (\{\}, \texttt{skip}) &\xrightarrow{\texttt{in}_{\mathit{KeyPress}(100)}} (\{r \mapsto 1\}, \texttt{skip})\\ &\xrightarrow{\texttt{in}_{\mathit{MouseClick}(10)}} (\{\}, \texttt{output}\ 1\ \texttt{on}\ \mathit{Network})\\ &\xrightarrow{\texttt{out}_{\mathit{Network}(1)}} (\{\}, \texttt{skip})\end{aligned}$$

It is easy to check that $\lfloor \overline{ev_1} \rfloor_{I,L} = \lfloor \overline{ev_2} \rfloor_{I,L}$ but $\lfloor \overline{ev_1} \rfloor_{O,L} \neq \lfloor \overline{ev_2} \rfloor_{O,L}$.

3.3 Enforcement

Roughly speaking, there are two classes of approaches to enforce noninterference. We can *statically* check that a program is secure, by using techniques such as type systems or program verification. Or we can *dynamically* enforce that no information leaks by using techniques such as monitoring or multi-execution.

In the following two sections, we focus on one static enforcement technique (based on typing) and on one dynamic technique (based on multi-execution).

4 Static enforcement

The idea of using static techniques to check noninterference was pioneered in the seventies by Denning and Denning [14]. There is a huge body of literature on static enforcement of information flow security. The survey by Sabelfeld and Myers [30] provides an excellent overview. We illustrate static enforcement by means of typing by showing a type system that is very similar to a type system proposed by Bohannon et al. [8].

Types are just security labels (hence, in our case, there are only two types: H and L). Programmers have to declare a type for every global variable and the type checker will enforce that the information stored in global variables of type L will only depend on L information.

We first define a typing judgment for expressions (Figure 6). The intuition is that the type l of an expression e is an upper bound for the level of the information that could have influenced e. The judgement $(x : l_c) \vdash e : l$ defines the type l of expression e in context $(x : l_c)$. The context $(x : l_c)$ defines the level of the bound variable x; for an expression that is part of a handler definition on a channel ch, the variable bound by the handler will get as type the level of the channel ch.

$$\frac{l_c \leq l}{(x : l_c) \vdash x : l}\ (\text{T-Expr-Var}) \qquad \frac{}{(x : l_c) \vdash n : l}\ (\text{T-Expr-Lit})$$

$$\frac{\mathit{lbl}(r) \leq l}{(x : l_c) \vdash r : l}\ (\text{T-Expr-Ref})$$

$$\frac{(x : l_c) \vdash e_1 : l_1 \qquad (x : l_c) \vdash e_2 : l_2 \qquad l_1 \leq l \qquad l_2 \leq l}{(x : l_c) \vdash e_1 \odot e_2 : l}\ (\text{T-Expr-Op})$$

Fig. 6. Typing of expressions

The type system is polymorphic: an expression can have multiple types. Any type that is an upper bound for the level of the information that could have influenced e is a valid type. With our restriction to two security levels H and L, expressions can either have the H type, or both the L and H type. This simple form of polymorphism in the type system will make some of the typing rules for commands simpler.

The rule (T-Expr-Var) says that a variable assumed to have level l_c in the context can be given as type any level above or equal to l_c, and rule (T-Expr-Ref) says that global variables can be given as type any level above or equal to the level assigned to them by the programmer. Literals can have any level (T-Expr-Lit), and in a binary expression, the level of the result can be any level that is above or equal to the levels of the two operands (T-Expr-Op).

Next we turn to typing of commands (Figure 7). We define a typing judgement $(x : l_c) \vdash c : l$, expressing that command c is well-typed with type l in context $(x : l_c)$. The intuition is that the type of a command is a *lower* bound for the level of the side-effects (either assignments to global variables or outputs on channels) that a command can have. Hence typing is again polymorphic. In our system, a well-typed command can have both type H and L, meaning it definitely only performs H side effects, or a well-typed command can have type L only, if the command possibly performs some L side effect.

$$\frac{}{(x : l_c) \vdash \texttt{skip}\colon l} \quad (\text{T-Cmd-Skip})$$

$$\frac{(x : l_c) \vdash c_1 : l_1 \qquad (x : l_c) \vdash c_2 : l_2 \qquad l \leq l_1 \qquad l \leq l_2}{(x : l_c) \vdash (c_1; c_2) : l} \quad (\text{T-Cmd-Seq})$$

$$\frac{(x : l_c) \vdash e : l_e \qquad l_e \leq \mathit{lbl}(ch) \qquad l \leq \mathit{lbl}(ch)}{(x : l_c) \vdash \texttt{output}\ e\ \texttt{to}\ ch : l} \quad (\text{T-Cmd-Out})$$

$$\frac{(x : l_c) \vdash e : l_e \qquad l_e \leq \mathit{lbl}(r) \qquad l \leq \mathit{lbl}(r)}{(x : l_c) \vdash (r := e) : l} \quad (\text{T-Cmd-Assign})$$

$$\frac{\begin{array}{ccc} (x : l_c) \vdash e : l_e & (x : l_c) \vdash c_1 : l_1 & (x : l_c) \vdash c_2 : l_2 \\ l \leq l_1 \qquad l \leq l_2 & l_e \leq l_1 & l_e \leq l_2 \end{array}}{(x : l_c) \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : l} \quad (\text{T-Cmd-If})$$

$$\frac{(x : l_c) \vdash e : l_e \qquad (x : l_c) \vdash c : l' \qquad l \leq l' \qquad l_e \leq l'}{(x : l_c) \vdash \texttt{while}\ e\ \texttt{do}\ c : l} \quad (\text{T-Cmd-While})$$

Fig. 7. Typing of commands

Rule (T-Cmd-Skip) says that `skip` can be given any level. The sequential composition of two commands must have a level that is below or equal to the levels of the two commands that are composed (T-Cmd-Seq). Rule (T-Cmd-Out) ensures two things. First, for an output command, the level of the expression that is output must be below or equal to the level of the channel on which it is output. Since l_e is an upper bound for the level the information that could have influenced e, this ensures no information leaks with this output. Second, the level l of the command itself must be a lower bound for the effects, and hence musy be below or equal to the level of the output channel. The rule for assignments (T-Cmd-Assign) is very similar: assignment to global variables is an effect that is similar to the effect of producing output. The rules for conditionals (T-Cmd-If) and (T-Cmd-While) make sure that no information leaks through the control flow and are needed to prevent implicit flows. Parts of the program that can only be reached dependent on information of level l_e (like the branches of an if-statement or the body of a while-statement) should only have effects that have l_e as lower bound. In other words, there should be no L effects in parts of the program whose reachability depends on H information.

Finally, we turn to programs (Figure 8). Each declared handler `on` $ch(x)$ c

$$\frac{}{\vdash \cdot}\ (\textsc{T-Pgm-Empty}) \qquad \frac{(x : \mathit{lbl}(\mathit{ch})) \vdash c : \mathit{lbl}(\mathit{ch}) \qquad \vdash p}{\texttt{on}\ \mathit{ch}(x)\ c; p}\ (\textsc{T-Pgm})$$

Fig. 8. Typing of programs

must be type-checked in a context that assigns the label of the channel *ch* to the bound variable x, and the side-effects of the resulting commands must be bounded by that same label; a H input event should not lead to L side effects.

Example 3. Consider again the JavaScript calculator:

$$\texttt{on}\ \mathit{KeyPress}(x)\ \mathit{total} := \mathit{total} + x;$$
$$\texttt{on}\ \mathit{MouseClick}(x)\ \texttt{output}\ \mathit{total}\ \texttt{on}\ \mathit{Display}$$

If we define $\mathit{lbl}(\mathit{total}) = H$, this script passes type checking. The expression $\mathit{total} + x$ gets type H and the assignment $\mathit{total} := \mathit{total} + x$ can be given both types H and L (both these are lower bounds for the single effect of assigning to *total*). In a similar way, the output command in the second handler can be given both types H and L.

Example 4. The key logger script:

$$\texttt{on}\ \mathit{KeyPress}(x)\ \texttt{output}\ x\ \texttt{on}\ \mathit{Network}$$

does not type check. The expression x must be given type H because it arrives on a H channel (rules (T-Pgm) and (T-Expr-Var)). As a consequence, the output command fails to type check as the level of *Network* is L and rule (T-Cmd-Out) requires that the type of the expression being output is below or equal to the level of the output channel.

For similar reasons, the script below also does not type check:

$$\texttt{on}\ \mathit{KeyPress}(x)\ r := x;$$
$$\texttt{on}\ \mathit{MouseClick}(x)\ \texttt{output}\ r\ \texttt{on}\ \mathit{Network}$$

The first handler can only be type checked of r is given type H by defining $\mathit{lbl}(r) = H$. But then the second handler can not be type checked for the same reason as above.

Example 5. Finally, let us consider an example with conditionals. The program below:

$$\texttt{on}\ \mathit{KeyPress}(x)\ \ \texttt{if}\ x = 100\ \texttt{then}\ r := 1\ \texttt{else}\ \texttt{skip};$$
$$\texttt{on}\ \mathit{MouseClick}(x)\ \texttt{output}\ r\ \texttt{on}\ \mathit{Network}$$

can not be type checked. Since the conditional of the if-statement depends on H information (x has type H), effects in the then and else branch must be bounded below by H (rule(T-Cmd-If)) . Hence, the r variable must be made H by defining $\mathit{lbl}(r) = H$. But then the second handler can not type check anymore.

One can prove that any program that type checks is noninterferent.

Theorem 1. *Suppose we are given a policy lbl that assigns security levels to I/O channels, and suppose that lbl can be extended to assign security levels to global variables such that* $\vdash p$*, then* p *is noninterferent under that policy.*

We refer the reader to [8] for a proof.

Note that the type system is *conservative*. It is easy to come up with example programs that are noninterferent but that fail to type check. For instance:

$$\begin{array}{l} \texttt{on}\ \mathit{KeyPress}(x)\ r := x; \\ \texttt{on}\ \mathit{MouseClick}(x)\ \texttt{output}\ r - r\ \texttt{on}\ \mathit{Network} \end{array}$$

is noninterferent since the expression $r - r$ always evaluates to 0. But the type system will treat the expression as H. The fact that type systems (and other static approaches) reject some good programs is one of the main motivations to also consider dynamic methods that can be more permissive [31].

5 Dynamic enforcement

The first attempts at dynamically enforcing information flow also date back to the seventies [16], but for many years static enforcement techniques were considered more promising. The impression was that dynamic mechanisms are not a good match for information flow security, as they monitor only a single execution, and the definition of noninterference talks about two executions. Hence, for many years the emphasis was on the development of static methods.

In the last decade, we have seen a renewed interest in dynamic methods [31]. The most obvious dynamic approach is *monitoring* where the enforcement mechanism monitors an execution and blocks it as soon as it detects an information leak. Such a monitor for JavaScript was for instance developed by Hedin and Sabelfeld [17]. An alternative approach is the approach of *secure multi-execution (SME)* [15]. We illustrate dynamic enforcement by means of a secure multi-execution mechanism that is very close to the mechanism proposed by Bielova et al. [6].

The core idea of SME for reactive systems is to maintain two executions of the program (one for each security level, i.e. a low (L) and a high (H) execution), and to implement the following rules on the I/O performed by these executions. L input events are handled by both executions, and H input events are only handled by the H execution. Outputs on L channels are only performed in the L execution and outputs on H channels only in the H execution.

It is relatively easy to see that executing a program under this SME regime will guarantee non-interference: the execution that does output at level L only sees inputs of level L and hence the output could not have been influenced by inputs of level H.

Similarly, it is relatively easy to see that non-interferent programs run unmodified: if L outputs indeed only depend on L inputs, then the L execution

$$\frac{\mathit{lbl}(ch) = H \qquad (p)(\mathtt{in}_{ch(n)}) \Downarrow c}{((\mu_L, \mathtt{skip}), (\mu_H, \mathtt{skip})) \xRightarrow{\mathtt{in}_{ch(n)}} ((\mu_L, \mathtt{skip}), (\mu_H, c))}\ (\textsc{New-H-Input})$$

$$\frac{\mathit{lbl}(ch) = L \qquad (p)(\mathtt{in}_{ch(n)}) \Downarrow c}{((\mu_L, \mathtt{skip}), (\mu_H, \mathtt{skip})) \xRightarrow{\mathtt{in}_{ch(n)}} ((\mu_L, c), (\mu_H, c))}\ (\textsc{New-L-Input})$$

$$\frac{(\mu_L, c_L) \overset{o}{\to} (\mu'_L, c'_L) \qquad o = \bullet \vee \mathit{lbl}(o) = H}{((\mu_L, c_L), (\mu_H, c_H)) \overset{\bullet}{\Rightarrow} ((\mu'_L, c'_L), (\mu_H, c_H))}\ (\textsc{L-Internal})$$

$$\frac{(\mu_L, c_L) \overset{o}{\to} (\mu'_L, c'_L) \qquad \mathit{lbl}(o) = L}{((\mu_L, c_L), (\mu_H, c_H)) \overset{o}{\Rightarrow} ((\mu'_L, c'_L), (\mu_H, c_H))}\ (\textsc{L-Output})$$

$$\frac{(\mu_H, c_H) \overset{o}{\to} (\mu'_H, c'_H) \qquad o = \bullet \vee \mathit{lbl}(o) = L}{((\mu_L, \mathtt{skip}), (\mu_H, c_H)) \overset{\bullet}{\Rightarrow} ((\mu_L, \mathtt{skip}), (\mu'_H, c'_H))}\ (\textsc{H-Internal})$$

$$\frac{(\mu_H, c_H) \overset{o}{\to} (\mu'_H, c'_H) \qquad \mathit{lbl}(o) = H}{((\mu_L, \mathtt{skip}), (\mu_H, c_H)) \overset{o}{\Rightarrow} ((\mu_L, \mathtt{skip}), (\mu'_H, c'_H))}\ (\textsc{H-Output})$$

Fig. 9. Semantics of SME.

will still perform the same outputs. The H execution gets all events and behaves exactly as the program would behave without SME so also the H outputs will remain the same. The only net effect that SME has on noninterferent programs is that – depending on how both executions are scheduled – outputs may happen in a different order (but outputs at the same security level remain in the same order). This is the *precision* property of SME. Rafnsson et al.[27] have shown that, if the program is noninterferent *and* low and high executions are scheduled correctly, then even ordering of outputs remains the same. However, in many cases it is sufficient to maintain order only within security levels. For instance, in the case of web scripts, if graphical outputs to the browser user are H and outputs to the network are L, it is sufficient to maintain order per security level. That is, the relative order of graphical outputs in relation to networks outputs is not important. This observation allows for simple schedulers which, for each input event, first perform the low execution (if the input event was low), and then the high execution. In this paper we focus on the case where the scheduler is simple, and only maintain output order per security level.

We formalize SME for web scripts by defining how to execute a script under SME. A program state under SME contains two program states of the original program $((\mu_L, c_L), (\mu_H, c_H))$, the state of the L execution (μ_L, c_L) and the state of the H execution (μ_H, c_H). We define the judgement $((\mu_L, c_L), (\mu_H, c_H)) \overset{ev}{\Rightarrow} ((\mu'_L, c'_L), (\mu'_H, c'_H))$ in Figure 9.

The rules (New-H-Input) and (New-L-Input) formalize that H inputs are only given to the H execution and L inputs are given to both executions. Then rules (L-Internal) and (L-Output) are applicable until the L execution is finished with the current input event. These rules let the L execution run but suppress any H output events.

When the L execution is done, rules (H-INTERNAL) and (H-OUTPUT) kick in. The H execution can now run, but L outputs will be suppressed.

The *initial program state* is $((\mu_0, \texttt{skip}), (\mu_0, \texttt{skip}))$ and a program state is *passive* if it has the form $((\mu, \texttt{skip}), (\mu, \texttt{skip}))$. We define the notions of *execution under SME* and *event-complete* execution under SME in the obvious way (similar to how they were defined for the standard semantics in Section 2.2).

Note that SME does not *detect* insecure scripts, it automatically *fixes* them as they execute.

Example 6. Consider again the key logger script:

$$\texttt{on } \mathit{KeyPress}(x) \texttt{ output } x \texttt{ on } \mathit{Network}$$

If this script is executed under SME, the occurrence of an input event on the *KeyPress* channel will be handled by the H execution only (rule NEW-H-INPUT). When that execution performs the output command on the *Network* channel, this output will be suppressed (rule H-INTERNAL). SME fixes this example by never performing any of the insecure outputs.

Example 7. Consider again the script with the more subtle leaks:

$$\begin{aligned}&\texttt{on } \mathit{KeyPress}(x)\ \ \texttt{if } x = 100 \texttt{ then } r := 1 \texttt{ else skip};\\ &\texttt{on } \mathit{MouseClick}(x) \texttt{ output } r \texttt{ on } \mathit{Network}\end{aligned}$$

For this script, inputs on the *KeyPress* channel are only delivered to the H execution (rule NEW-H-INPUT). Hence the value of the global variable r can become 1 in μ_H, but it will always remain 0 in μ_L. On occurrence of an input on the *MouseClick* channel, this input is delivered to both executions (rule NEW-L-INPUT). The L execution runs first and outputs a 0 on *Network*. Then the H execution runs, but when it performs the output on *Network* (that could be 0 or 1 in this execution), this output is suppressed (rule H-INTERNAL).

So we see that SME again fixes this example. The program becomes equivalent to the secure program that always outputs 0 to *Network* on a mouse click:

$$\begin{aligned}&\texttt{on } \mathit{KeyPress}(x)\ \ \texttt{if } x = 100 \texttt{ then } r := 1 \texttt{ else skip};\\ &\texttt{on } \mathit{MouseClick}(x) \texttt{ output } 0 \texttt{ on } \mathit{Network}\end{aligned}$$

The main security theorem about SME says that *any* script, when executed under the SME regime, is non-interferent.

Theorem 2. *Any program is noninterferent when executed under the SME semantics.*

For a proof, we refer the reader to [6].

Of course, since SME can change the behaviour of programs, we have to check that it does not change the behaviour of *secure* programs. We do not want an enforcement mechanism to do arbitrary changes to the semantics of secure programs. Fortunately, secure programs are more or less untouched.

Example 8. Consider again the (secure) JavaScript calculator:

$$\texttt{on}\ \mathit{KeyPress}(x)\ \mathit{total} := \mathit{total} + x;$$
$$\texttt{on}\ \mathit{MouseClick}(x)\ \texttt{output}\ \mathit{total}\ \texttt{on}\ \mathit{Display}$$

If we execute this script under SME, the behaviour remains the same. Key presses are only delivered to the H execution, and the total is correctly computed in μ_H (The value of total remains 0 in μ_L). When an input arrives on the *MouseClick* channel, it is delivered to *both* executions (rule NEW-L-INPUT). The output produced on the *Display* channel by the L execution is suppressed (rule L-INTERNAL). The (correct) output produced by the H execution is performed (rule H-OUTPUT). We see that SME leaves the behaviour of this secure program untouched.

However, with the simple scheduling approach of first running the L execution and then running the H execution, it might happen that the order of outputs is changed even for secure programs.

Example 9. Consider the (secure) program:

$$\texttt{on}\ \mathit{MouseClick}(x)\ \texttt{output}\ x\ \texttt{on}\ \mathit{Display}; \texttt{output}\ x\ \texttt{on}\ \mathit{Network}$$

If we execute this script under SME, an input that arrives on the *MouseClick* channel, is delivered to *both* executions (rule NEW-L-INPUT). The L execution runs first (rules H-INTERNAL and H-OUTPUT can only fire once the L execution has finished; they state that the L execution must be passive). The output produced on the *Display* channel by the L execution is suppressed (rule L-INTERNAL), and the output on the *Network* channel is performed. Then the H execution runs. It performs the output on the *Display* channel, and the output on the *Network* channel is suppressed. The net effect is that the order of the *Display* and *Network* outputs is reversed.

Fortunately, this kind of reordering is the only change that SME does to secure programs. The *precision* theorem for SME says that the output, when projected on an arbitrary security level, remains the same. So the relative order of outputs on different security levels is the only thing that can change. For an exact statement of the theorem, and a proof, we refer the reader to [6].

If even this kind of reordering is undesirable, it is possible to schedule the L and H executions in a more interleaved way so that absolute ordering of outputs can be maintained for secure programs. We refer the reader to [37] for details.

6 Related work

Information flow security is an established research area, and too broad to survey here. For many years, it was dominated by research into static enforcement techniques. We point the reader to the well-known survey by Sabelfeld and Myers [30] for a discussion of general, static approaches to information flow enforcement.

Several static or hybrid techniques specifically for information flow security in web scripts or in browsers have been proposed. Bohannon et al. [8, 7] define a notion of non-interference for reactive systems, and show how a model browser can be formalized as such a reactive system. Chugh et al. [10] have developed a novel multi-stage static technique for enforcing information flow security in JavaScript. Just et al. [19] propose a hybrid combination of dynamic information flow tracking and a static analysis to capture implicit flows within full (excluding exceptions) JavaScript programs, including programs calling eval.

Dynamic techniques have seen renewed interest in the last decade. Le Guernic's PhD thesis [22] gives an extensive survey up to 2007, but since then, significant new results have been achieved. Recent works propose run time monitors for information flow security, often with a particular focus on on the web scripts. Sabelfeld et al. have proposed monitoring algorithms that can handle DOM-like structures [29], dynamic code evaluation [2] and timeouts [28]. In a recent paper, Hedin and Sabelfeld [17] propose dynamic mechanisms for all the core JavaScript language features. Austin and Flanagan [3] have developed alternative, sometimes more permissive techniques.

Secure multi-execution (SME) was developed independently by several researchers [21, 36, 15]. Khatiwala et al. [21] proposed a technique called *Data Sandboxing*. They partition a program in two programs at source code level and use system call interposition to implement the SME I/O rules. In followup work, Capizzi et al. [9] avoid the need for source level partitioning by means of *shadow executions*: they run two executions of processes for the H (secret) and L (public) security level to provide strong confidentiality guarantees. Devriese and Piessens [15] independently came up with the closely related technique they called SME, and they were the first to prove the strong soundness and precision guarantees that SME offers.

These initial results were improved and extended in several ways: Kashyap et al. [20], generalize the technique of secure multi-execution to a family of techniques that they call *the scheduling approach to non-interference*, and they analyze how the scheduling strategy can impact the security properties offered. Barthe et al.[5] propose a program transformation that simulates SME. Bielova et al. [6] propose a variant of secure multi-execution suitable for reactive systems such as browsers. An implementation of SME in a real browser was done by De Groef et al. [12, 13]. Austin and Flanagan [4] propose a more efficient implementation technique called multi-faceted evaluation. In a recent paper, Rafnsson and Sabelfeld [27] extend SME in several ways by showing (1) how to support policies that can distinguish presence of messages from content of messages, (2) how to perform declassification under SME and (3) how to make SME precise (or transparent in their terminology) even for observers that can observe more than one level. An alternative, more black-box approach to declassification was developed by Vanhoef et al. [33].

Information flow security is one promising approach to web script security, but two other general-purpose approaches have been applied to script security as well: isolation and taint-tracking.

Isolation or *sandboxing* based approaches develop techniques where scripts can be included in web pages without giving them (full) access to the surrounding page and the browser API. Several practical systems have been proposed, including Webjail [32], ADSafe [11], Caja [24], and JSand [1]. Maffeis et al. [23] formalize the key mechanisms underlying these techniques and prove they can be used to create secure sandboxes. They also discuss several other existing proposals, and we point the reader to their paper for a more extensive discussion of work in this area. Isolation is easier to achieve than non-interference, but it is also more restrictive: often access needs to be denied to make sure the script cannot leak the information, but it would be perfectly fine to have the script use the information locally in the browser.

Taint tracking is an approximation to information flow security, that only takes explicit flows into account. Several authors have proposed taint tracking systems for web security. Two representative examples are Xu et al. [35], who propose taint-enhanced policy enforcement as a general approach to mitigate implementation-level vulnerabilities, and Vogt et al. [34] who propose taint tracking to defend against cross-site scripting.

7 Conclusions

Information flow control is a widely studied information security mechanism. It makes sure that programs can not leak sensitive information that they receive for processing to output channels that might be observable by opponents that should not learn such sensitive information. Information flow control is an interesting security mechanism for web scripts, since these scripts need to process sensitive information as well as need to communicate over untrustworthy output channels.

We have described two information flow control mechanisms for web scripts, one static mechanism based on typing, and one dynamic mechanism based on multi-execution. We have explained the intuitions behind these mechanisms and illustrated them on examples.

Acknowledgments. This research is partially funded by the Research Fund KU Leuven, by the EU FP7 projects WebSand, Strews and NESSoS, and by the IWT project SPION. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE). Mathy Vanhoef and Dominique Devriese hold a PhD fellowship of the Research Foundation - Flanders (FWO). Willem De Groef holds a PhD grant from the Agency for Innovation by Science and Technology in Flanders (IWT).

References

1. Agten, P., Van Acker, S., Brondsema, Y., Phung, P.H., Desmet, L., Piessens, F.: JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications. In: Proceedings of the Annual Computer Security Applications Conference. pp. 1–10 (2012)

2. Askarov, A., Sabelfeld, A.: Tight Enforcement of Information-Release Policies for Dynamic Languages. In: Proceedings of the IEEE Computer Security Foundations Symposium. pp. 43–59 (2009)
3. Austin, T.H., Flanagan, C.: Permissive Dynamic Information Flow Analysis. In: Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security. pp. 3:1–3:12 (2010)
4. Austin, T.H., Flanagan, C.: Multiple Facets for Dynamic Information Flow. In: Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 165–178 (2012)
5. Barthe, G., Crespo, J.M., Devriese, D., Piessens, F., Rivas, E.: Secure Multi-Execution through Static Program Transformation. Proceedings of the International Conference on Formal Techniques for Distributed Systems pp. 186–202 (2012)
6. Bielova, N., Devriese, D., Massacci, F., Piessens, F.: Reactive non-interference for a browser model. In: Proc. of the International Conference on Network and System Security. pp. 97–104 (2011)
7. Bohannon, A., Pierce, B.C.: Featherweight firefox: Formalizing the core of a web browser. In: Proceedings of the 2010 USENIX Conference on Web Application Development. pp. 11–11. WebApps'10, USENIX Association, Berkeley, CA, USA (2010)
8. Bohannon, A., Pierce, B.C., Sjöberg, V., Weirich, S., Zdancewic, S.: Reactive Noninterference. In: Proceedings of the ACM Conference on Computer and Communications Security. pp. 79–90 (2009)
9. Capizzi, R., Longo, A., Venkatakrishnan, V., Sistla, A.: Preventing Information Leaks through Shadow Executions. In: Proc. of the Annual Computer Security Applications Conference. pp. 322–331 (2008)
10. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged Information Flow for JavaScript. ACM SIGPLAN Notices 44(6), 50–62 (2009)
11. Crockford, D.: Adsafe. `http://www.adsafe.org/` (December 2009)
12. De Groef, W., Devriese, D., Nikiforakis, N., Piessens, F.: FlowFox: a Web Browser with Flexible and Precise Information Flow Control. In: Proc. of the ACM Conference on Computer and Communications Security. pp. 748–759 (2012)
13. De Groef, W., Devriese, D., Nikiforakis, N., Piessens, F.: Secure multi-execution of web scripts: Theory and practice. Journal of Computer Security (2014)
14. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM 20(7), 504–513 (Jul 1977)
15. Devriese, D., Piessens, F.: Noninterference Through Secure Multi-Execution. In: Proc. of the IEEE Symposium on Security and Privacy. pp. 109–124 (2010)
16. Fenton, J.S.: Memoryless subsystems. Comput. J. 17(2), 143–147 (1974)
17. Hedin, D., Sabelfeld, A.: Information-Flow Security for a Core of JavaScript. In: Proc. of the IEEE Computer Security Foundations Symposium. pp. 3–18 (2012)
18. Jang, D., Jhala, R., Lerner, S., Shacham, H.: An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications. In: Proc. of the ACM Conference on Computer and Communications Security. pp. 270–283 (2010)
19. Just, S., Cleary, A., Shirley, B., Hammer, C.: Information Flow Analysis for JavaScript. In: Proc. of the ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients. pp. 9–18 (2011)
20. Kashyap, V., Wiedermann, B., Hardekopf, B.: Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. In: Proc. of the IEEE Conference on Security and Privacy. pp. 413–428 (2011)

21. Khatiwala, T., Swaminathan, R., Venkatakrishnan, V.: Data Sandboxing: A Technique for Enforcing Confidentiality Policies. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC). pp. 223–234 (2006)
22. Le Guernic, G.: Confidentiality Enforcement Using Dynamic Information Flow Analyses. Ph.D. thesis, Kansas State University (2007)
23. Maffeis, S., Mitchell, J.C., Taly, A.: Object Capabilities and Isolation of Untrusted Web Applications. In: Proceedings of the IEEE Symposium on Security and Privacy. pp. 125–140 (2010)
24. Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Caja: Safe active content in sanitized javascript. `http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf` (January 2008)
25. Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., Vigna, G.: You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In: Proc. of the ACM Conference on Computer and Communications Security. pp. 736–747 (2012)
26. Nikiforakis, N., Kapravelos, A., Joosen, W., Kruegel, C., Piessens, F., Vigna, G.: Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In: Proceedings of the 34th IEEE Symposium on Security & Privacy. pp. 541–555 (May 2013)
27. Rafnsson, W., Sabelfeld, A.: Secure multi-execution: fine-grained, declassification-aware, and transparent. In: Proc. of the IEEE Computer Security Foundations Symposium (CSF) (2013)
28. Russo, A., Sabelfeld, A.: Securing Timeout Instructions in Web Applications. In: Proceedings of the IEEE Computer Security Foundations Symposium. pp. 92–106 (2009)
29. Russo, A., Sabelfeld, A., Chudnov, A.: Tracking Information Flow in Dynamic Tree Structures. In: Proceedings of the European Symposium on Research in Computer Security. pp. 86–103 (2009)
30. Sabelfeld, A., Myers, A.C.: Language-Based Information-Flow Security. IEEE Journal on Selected Areas of Communications 21(1), 5–19 (January 2003)
31. Sabelfeld, A., Russo, A.: From dynamic to static and back: Riding the roller coaster of information-flow control research. In: Ershov Memorial Conference. pp. 352–365 (2009)
32. Van Acker, S., De Ryck, P., Desmet, L., Piessens, F., Joosen, W.: Webjail: Least-privilege integration of third-party components in web mashups. In: ACSAC (2011), `https://lirias.kuleuven.be/handle/123456789/316291`
33. Vanhoef, M., De Groef, W., Devriese, D., Piessens, F., Rezk, T.: Stateful declassification policies for event-driven programs. In: Proc. of the IEEE Computer Security Foundations Symposium (CSF) (2014)
34. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Krügel, C., Vigna, G.: Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In: Proceedings of the Network & Distributed System Security Symposium (2007)
35. Xu, W., Bhatkar, S., Sekar, R.: Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In: Proceedings of the USENIX Security Symposium. pp. 121–136 (2006)
36. Yumerefendi, A.R., Mickle, B., Cox, L.P.: TightLip: Keeping Applications from Spilling the Beans. In: Proceedings of the USENIX Symposium on Network Systems Design & Implementation. pp. 159–172 (2007)
37. Zanarini, D., Jaskelioff, M., Russo, A.: Precise enforcement of confidentiality for reactive systems. In: Proc. of the IEEE Computer Security Foundations Symposium (CSF) (2013)