Stateful Declassification Policies for Event-Driven Programs

Mathy Vanhoef, Willem De Groef, Dominique Devriese, Frank Piessens
iMinds-DistriNet, KU Leuven
{*firstname.lastname*}*@cs.kuleuven.be*

Tamara Rezk
INRIA
tamara.rezk@inria.fr

***Abstract*—We propose a novel mechanism for enforcing information flow policies with support for declassification on event-driven programs. Declassification policies consist of two functions. First, a *projection* function specifies for each confidential event what information in the event can be declassified directly. This generalizes the traditional security labelling of inputs. Second, a stateful *release* function specifies the aggregate information about all confidential events seen so far that can be declassified.**

We provide evidence that such declassification policies are useful in the context of JavaScript web applications. An enforcement mechanism for our policies is presented and its soundness and precision is proven. Finally, we give evidence of practicality by implementing and evaluating the mechanism in a browser.

Listing 1: Keylogger

```
var url = 'http://hacker.com/?=';
window.onkeypress = function(e) {
  var leak = e.charCode;
  new Image().src = url + leak; }
```

Listing 2: Shortkey usage

```
var d = 0, url = 'http://analytic.com/?=';
window.onkeypress = function(e) {
  if (e.charCode == 101) d = 1; }
window.onunload = function() {
  $.ajax(url + d); }
```

I. Introduction

Browsers commonly run untrusted JavaScript code. These scripts can handle user interface events such as key presses and mouse clicks, and network events such as the arrival of HTTP responses. By handling these events, scripts can interact with the user and one or more services on the network. Since scripts have, and *need*, access to both user information and to remote HTTP servers, they are commonly used to leak private information to untrusted network servers [1].

Listing 1 shows an example that implements a simple key logger in JavaScript. It installs an event handler to monitor key presses, and leaks every keystroke to `hacker.com`. It does this by encoding the character code of the key in an image URL and asking the browser to fetch that URL.

Researchers [2], [3], [4] have realized that mechanisms for information flow security are a promising countermeasure for such curious or malicious scripts, since such mechanisms allow the script to have access to private information but at the same time prevent it from leaking that information to untrusted servers. As a consequence, impressive progress has been made in dynamic mechanisms supporting information flow security for event-driven programs such as JavaScript web applications [5], [2], [3], [4]. Unfortunately, this strict information flow control breaks some functionality that is important for the web today. Consider the following simple variant of web analytics: a web-based application that wants to analyse which keyboard shortcuts are commonly used. It is common practice on the web to include third party analytics scripts to gather such information [6]. For this use case, there is no need to know in what order keys were pressed nor how many times a particular key was pressed, only whether a certain key was pressed at least once during the interaction with the web application. Releasing such limited information poses negligible security risks, and can be considered acceptable and even useful in many situations. Listing 2 shows the implementation of a script that sends this minimal amount of information to `analytic.com`. This is a simple example of a very broad set of practices on the web today, where third party web analytics companies monitor application usage and gather statistics (such as mouse heat maps or the geographical spread of users).

Another example where strict information flow control breaks functionality is when the labelling of incoming information is too coarse grained. For instance, when cookies are marked as confidential to defend against cross-site scripting attacks [7], also non-sensitive information stored in cookies (such as the preferred language) is no longer accessible to low observers, and this can break script functionality.

Existing information flow secure browsers such as FlowFox [3] do not distinguish between the malicious key logger from Listing 1 and the useful benign script from Listing 2: both scripts leak private information (key presses) to network servers. They also do not support a fine grained approach to label incoming event information: events are either high or low. What is needed is some form of declassification: a policy should specify what kind of aggregate, derived, or partial information is safe to release to low observers.

This paper proposes a specific type of stateful declassification policies for event-driven programs such as JavaScript applications, and develops an enforcement mechanism for

DOI 10.1109/CSF.2014.28

these policies on top of the existing secure multi-execution (SME) mechanism [8], [9]. The main contributions of this paper are:

- We propose a notion of declassification policy for event-driven programs that can handle web analytics style declassifications.
- We extend SME to enforce such declassification policies, develop a formal model of the essence of our approach, and prove its security and precision. The formal model has been mechanized in Redex [10] and is available for download [11].
- We implement the mechanism as an extension to FlowFox [3] and discuss its evaluation for performance and usefulness. Our implementation is also available for download [11].

The remainder of this paper is organized as follows. Section II specifies a simple formal event-driven programming language. Section III introduces stateful declassification policies, and Section IV develops an enforcement mechanism with security and precision proofs. Section V discusses our implementation of the mechanism for JavaScript in a web browser, and in Section VI we discuss limitations and strengths of our mechanism. Finally, Sections VII and VIII discuss related work and conclude.

II. Model Language

In this section we introduce a simple formal model language and define traditional information flow policies.

A. Syntax and semantics

Our formal language models the essence of an event-driven programming language [12]. The syntax of our model language is defined in Figure 1. The language supports five types of events: Load, MouseClick, KeyPress, GpsUpdate and Unload. Additionally it has two output channels: Send n outputs the integer n to the network, and Display n outputs n to the user. The set of event types and output channels are denoted by, respectively, ev and out. A program p consists of a sequence of handler declarations h, where each handler declaration defines a command that is to be run on occurrence of a specific event type. Events carry a single integer value containing event information, and the handler command can refer to that value by means of the formal parameter x. All handlers share a single store mapping global variables g to integers. Commands c and expressions e are completely standard. To avoid confusion between formal parameters x and global variables g, we make the simple assumption that a formal parameter x is always the identifier x. Since event handlers only bind one parameter and since nested bindings are not possible in our language, this is sufficient. Listing 3 shows the shortcut key monitoring example (Listing 2) in our model language.

The semantics of the model language is straightforward. A program configuration (μ, c) consists of a store μ, and the currently executing command c. The program p is clear from the context and is not explicitly included.

Listing 3: Monitoring if a shortkey has been pressed

```
on KeyPress(x) { if x = 101 then
                    { keyPressed = 1 } else { skip } }
on Unload(x) { Send(keyPressed) }
```

We use the following semantic judgements:

- $\mu \vdash e \Downarrow n$ says that the ground expression e evaluates under the variable store μ to the integer n. A *ground* expression does not contain any parameters x, but it can contain global variables g. A store μ is a mapping from global variable names g to integers n. The definition of this judgement is completely standard and hence omitted.
- $(\mu, c) \overset{\alpha}{\to} (\mu', c')$ says that program configuration (μ, c) can step to (μ', c') by performing transition α. A transition α can be a program action o, which can be an observable output $out\ n$ (for instance Send n) or an unobservable (silent) action denoted by $\cdot$. A transition α can also be an input action i, i.e. the occurrence of a new event $ev\ n$ (for instance KeyPress n).
The definition of this judgment is given in Figure 2, where the function $\mathrm{lookup}(p, ev)$ looks up the handler for ev in the program p. If no handler for ev is present it returns skip. We use the notation $\mu[g \mapsto n]$ for an update of the store μ that sets global variable g to n, and the notation $c[x \leftarrow n]$ for the substitution of integer n for formal parameter x in command c.

$$
\begin{array}{rcl}
ev & ::= & \text{GpsUpdate} \mid \text{MouseClick} \\
 & \mid & \text{KeyPress} \mid \text{Load} \mid \text{Unload} \\
out & ::= & \text{Send} \mid \text{Display} \\
p & ::= & h* \\
h & ::= & \text{on } ev(x)\ \{c\} \\
c & ::= & \text{skip} \\
 & \mid & c; c \\
 & \mid & g := e \\
 & \mid & \text{if } e \text{ then } \{c_1\} \text{ else } \{c_2\} \\
 & \mid & \text{while } e\ \{c\} \\
 & \mid & out(e) \\
e & ::= & x \mid n \mid g \mid e \odot e \\
\odot & ::= & + \mid - \mid = \mid < \\
o & ::= & out\ n \mid \cdot \\
i & ::= & ev\ n \\
\alpha & ::= & o \mid i
\end{array}
$$

Figure 1: Syntax

The *initial state* of a program is (μ_0, skip) where μ_0 maps every global variable to the integer 0. A program state is *passive* if it has the form (μ, skip). It is straightforward to prove

$$\frac{}{(\mu, \text{skip}; c) \overset{\cdot}{\to} (\mu, c)} \tag{1}$$

$$\frac{(\mu, c_1) \overset{o}{\to} (\mu', c_1')}{(\mu, c_1; c_2) \overset{o}{\to} (\mu', c_1'; c_2)} \tag{2}$$

$$\frac{\mu \vdash e \Downarrow n}{(\mu, g := e) \overset{\cdot}{\to} (\mu[g \mapsto n], \text{skip})} \tag{3}$$

$$\frac{c = \text{if } e \text{ then } \{c_1\} \text{ else } \{c_2\} \quad \mu \vdash e \Downarrow n \quad n \neq 0}{(\mu, c) \overset{\cdot}{\to} (\mu, c_1)} \tag{4}$$

$$\frac{c = \text{if } e \text{ then } \{c_1\} \text{ else } \{c_2\} \quad \mu \vdash e \Downarrow 0}{(\mu, c) \overset{\cdot}{\to} (\mu, c_2)} \tag{5}$$

$$\frac{c = \text{while } e\, \{c_{\text{loop}}\} \quad \mu \vdash e \Downarrow 0}{(\mu, c) \overset{\cdot}{\to} (\mu, \text{skip})} \tag{6}$$

$$\frac{c = \text{while } e\, \{c_{\text{loop}}\} \quad \mu \vdash e \Downarrow n \quad n \neq 0}{(\mu, c) \overset{\cdot}{\to} (\mu, c_{\text{loop}}; c)} \tag{7}$$

$$\frac{\mu \vdash e \Downarrow n}{(\mu, \mathit{out}(e)) \overset{\mathit{out}\ n}{\to} (\mu, \text{skip})} \tag{8}$$

$$\frac{\text{lookup}(p, ev) = c}{(\mu, \text{skip}) \overset{ev\ n}{\to} (\mu, c[x \leftarrow n])} \tag{9}$$

Figure 2: Small-step semantics.

that 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 event, and these are only possible from a passive state. Since μ is a total mapping from global variable names to integers, the only way a program could get stuck is by reading a formal parameter x during expression evaluation. However, because of our assumption that there is only a single parameter name x, and because on instantiation of a handler this single name always gets substituted, this case can not occur. (Alternatively, we could define expression evaluation to handle the case of reading from an unbound parameter by returning 0.)

An *execution* of a program is a sequence of transitions $\overline{\alpha}$ starting from the initial state:

$$(\mu_0, \text{skip}) \overset{\alpha_0}{\to} (\mu_1, c_1) \overset{\alpha_1}{\to} (\mu_2, c_2) \cdots$$

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.

Given a (finite) list of input events I, we say that $p(I) \to^* O$ iff there exists an event-complete execution $\overline{\alpha}$ of p where I is exactly the list of input events occurring in $\overline{\alpha}$ and O is the list of non-silent outputs occurring in $\overline{\alpha}$. This defines a partial function from lists of inputs to lists of outputs. It is a function because the only non-determinism in programs is input non-determinism. The function is partial because for some sequences of inputs the program may go into an infinite loop, and hence there is no event-complete execution (the program never reaches a passive state).

For the purposes of this paper, we limit our attention to event-complete executions, and we ignore programs that go into an infinite loop on handling one specific event (i.e. scripts can run arbitrary long while handling many events, but the handling of a single event is typically short). This is a realistic assumption for web scripts: because JavaScript is single-threaded, event-handlers that do not complete very quickly will "freeze" the browser, and most browsers will detect it if an event handler runs for more than a few seconds and offer the user the option to terminate the script. We discuss in Section VI how this assumption could be removed.

As an example of an event-complete execution, let p be the program from Listing 3 and let I = [KeyPress 101, KeyPress 102, Unload 0], then $p(I) \to^* O$ with output list O = [Send 1].

B. Noninterference

An information flow policy maps event types ev and output channels out to labels chosen from a lattice $\mathcal{L}$ of security levels. To keep definitions simple we specialize all definitions in this paper to the case of a two-element lattice with H (high, confidential) ordered above L (low, public). A policy then is just a function $\sigma\colon ev \cup out \to \{L, H\}$. We overload σ to also apply to events and outputs. For instance if $\sigma(\text{KeyPress}) = L$, we also define $\sigma(\text{KeyPress } n) = L$, and similarly for output channels out and corresponding outputs $out\ n$.

Definition 1 (L-similarity): Two lists S and S' are L-similar, denoted by $S \approx_L S'$, iff they are equal after removing all elements s with $\sigma(s) = H$.

Definition 2: A program p is *noninterferent* iff for all lists I, I', O, O' such that $p(I) \to^* O$ and $p(I') \to^* O'$ it holds that $I \approx_L I' \implies O \approx_L O'$.

This definition coincides with Bohannon et al.'s definition of ID-security [12], for the case of finite streams of events. The definition is *termination-insensitive*: for an input list I for which there is no event-complete execution (i.e. the program diverges on handling one of the events in I), there is no O such that $p(I) \to^* O$, and hence the definition does not impose any restrictions on the handling of such I. Because we only care about event-complete executions, this limitation is not important. Nevertheless, our enforcement mechanism also closes information leaks in such diverging event-handlers as we discuss in Section VI - even if they are not covered by our definition of non-interference.

If we label KeyPress as H and Unload and Send as L, then the program p in Listing 3 is *not* noninterferent, as:

p([KeyPress 101, KeyPress 102, Unload 0]) $\to^*$ Send 1
p([KeyPress 103, KeyPress 102, Unload 0]) $\to^*$ Send 0

and hence the program maps two L-similar inputs to outputs that are not L-similar. Obviously, the key logger program from Listing 1 (when rendered in our model language) would also be interferent.

Note that this traditional definition of noninterference cannot separate programs like the key logger from programs like the shortcut key monitor.

III. Declassification policies

In this section we design information flow policies supporting declassification for event-driven programs. They should be expressive enough to support for instance the following use cases:

1) **Declassification of approximate information.** A prototypical example is geolocation information: scripts may need access to precise GPS coordinates of a mobile browser, for instance to graphically display one's location on a map. Sending these precise coordinates to a map server is a potential violation of privacy, but sending approximate coordinates still allows a script to download the correct map. Also, even with approximate location information, web analytics services can still compute geographical spread of users.
2) **Declassification of aggregate or statistical information.** The shortcut key example in Listing 2 in the Introduction provides a prototypical example: we want to be able to set policies that make this program secure, yet still forbid key loggers such as the one in Listing 1.

In order to support declassification with such expressivity we propose the notion of *event projection* and *information release*, defined using a projection and release function, respectively. The intuition is that these functions, on each new input, define the information made available to low observers. Both functions are specified by means of a declarative, functional program.

Event projection can be used to specify that it is OK for low observables to depend on the occurrence of the event and on the *projected* (approximated) event information. Projection is done per individual event occurrence and thus stateless. As an example, specifying that low observers may depend on rounded GPS coordinates can be achieved by projecting GPS updates using a floor function. A policy writer can choose to only project events satisfying some condition, yet hide the occurrence of others. For example, we can define a projection which reveals shortcut key presses while hiding the occurrence of other keys. Declassifying only the occurrence of an event is possible by always projecting the event information to a constant value.

In contrast, information release can be used to specify which aggregate or derived information from past events can be declassified, and thus is stateful. For example, using it we can specify that the average of mouse click coordinates can be declassified after 100 clicks. It is defined by declaring the type of the state to be maintained by the policy, and by declaring a function which, on each input event, updates the state and specifies what information can be released to low observers.

A. Event Projection

Event projection is defined by a function *project* which takes an event $ev\ n$ as input argument, and returns either "Project n'" or "Nothing". A result of the form "Project n'" specifies that the event argument n should be projected to n', i.e., that low observers can depend on the occurrence of the event and on the value n'. Returning "Nothing" specifies that the event remains confidential (both the event argument as well as the occurrence of the event).

We consider only idempotent projections, which map the projected event to itself when applied twice. More precisely:

Definition 3: A function π is an *idempotent projection* iff for all events $ev\ n$, and for all n', it holds that: $\pi(ev\ n) =$ Project $n' \implies \pi(ev\ n') =$ Project n'.

The restriction to idempotent projections is in line with the intuition behind projections as functions that "remove the confidential information from the event argument". Moreover, limiting the policy to idempotent projections is required to prove precision of our enforcement mechanism (see Section IV). In case the policy does not define the *project* function, it is assumed to always return "Nothing".

As an example, the following policy specifies that key presses of the shortcut key (with key code 101) can be declassified, and that Unload events are L:

```
project(KeyPress 101) = Project 101
project(Unload x) = Project x
```

Intuitively, under this policy the shortcut key monitoring program in Listing 3 is secure, but a key logger is not (security under our policies is defined formally in section III-C).

The policy below declassifies the occurrence of key press events, but keeps the key code confidential:

```
project(KeyPress x) = Project 0
```

Under this policy, a program that counts key presses and sends the total number on a low output channel would be secure.

Finally, the policy below declassifies approximate GPS coordinates by projecting the location information using a floor function.

```
project(GpsUpdate x) = Project floor(x)
```

Under this policy a program requesting the map of a region (identified by rounded coordinates) in response to a GPS update is secure. That is, a web service can display the precise location to the user (as this is a high output), but it can only communicate the approximate, rounded coordinates to the service providing the map.

Projections generalize the labelling of event types. We can consider a labelling $\sigma(ev) = L$ to be syntactic sugar for $project(ev\ n)$ = Project n. Similarly, $\sigma(ev) = H$ is syntactic sugar for $project(ev\ n) =$ Nothing. Labellings of output channels of course remain relevant and are not subsumed by projections.

B. Information Release

We think of information release as specifying a *release channel* that can be sampled by low observers (on occurrence of a low observable event). The release channel always contains the latest value released by the policy. For simplicity, we only consider a single release channel, but extending to multiple release channels is straightforward. A released (declassified) value can depend on past events, and hence the policy needs some way of maintaining state.

Information release is specified by declaring (1) a state space; (2) initial values S_{init} and R_{init} for respectively the policy state and the value on the release channel; and (3) a function *release* that on each event occurrence updates the state and the value on the release channel.

The function *release* takes as input parameters the current value of the state, and an input event. It returns a new policy state, and either "Unchanged" or "Release x". If it returns "Unchanged" then no new information is released to low observers. Otherwise, when it returns "Release x", the value x is released to low observers and x becomes the new value on the release channel.

The default value for R_{init} is 0, for the policy state space the default value is a singleton state space $\{*\}$ with S_{init} the single state value $*$, and for *release* the default value is the function that always returns "Unchanged".

For example, the policy below (in Listing 4) uses information release to declassify whether a shortcut key was used so far. The state space is the set of booleans, and the initial value of the state is False. The value on the release channel (that is initially 0) switches to 1 after the shortcut key was pressed the first time. Note that the type of the release channel is always an integer (in line with the fact that our simple scripting language only handles integer values), but the type of the policy state can be anything (in this case Bool).

Listing 4: Declassifying shortcut key presses with release

```
state :: Bool = False
release(s, i) = if i = KeyPress 101 and not s
                then (True, Release 1)
                else (s, Unchanged)
```

Under this policy the shortcut key monitoring program in Listing 3 is secure, but a key logger is not. This policy releases less information than the projection-based policy above, as it does not release how often the shortcut key was pressed.

The following policy releases the average mouse click coordinate each time 100 mouse clicks have been observed:

```
state :: Int * Int = (0,0)
release( (n,sum), MouseClick x) =
    if (n+1 = 100) then ( (0, 0), Release (sum+x) / (n+1) )
    else ( (n+1, sum + x), Unchanged)
```

Finally, we give an example of a policy that combines projection and release. The policy below reveals the *occurrence* of all GPS updates by projecting them to the constant coordinate zero. Additionally, once the user has agreed to sharing his location (by pressing a button at mouse coordinate 45), rounded GPS coordinates are released. This allows low observers to depend on (rounded) GPS coordinates once the user has agreed to this.

```
state :: Bool = False
project(GpsUpdate x) = Project 0

release(x, MouseClick 45) = (True, Unchanged)
release(True, GpsUpdate x) = (True, Release (floor x))
release(s, x) = (s, Unchanged)
```

The state is a boolean indicating whether the user has clicked at coordinate 45. GPS updates under a True state result in the release of approximate coordinates.

C. Noninterference with Declassification

We combine the functions $project$ and $release$ in a single declassification policy function $\mathcal{D}$:

```
D(s, r, i) = let (s', d) = release(s, i) in case d of
                 Release x = (s', x, project i)
                 Unchanged = (s', r, project i)
```

We write $\mathcal{D}(s, r, i) = (s', r', pr)$ to say that processing the event i by policy $\mathcal{D}$ in the context of a state s and value r on the release channel results in the state s', value r' on the release channel, and a projection pr for event i.

Our policies define *what* can be declassified *when*, without restricting *where* something can be declassified. They are a black-box, extensional definition of secure information flow. Such extensional notions of noninterference can be formalized as: low-similar (i.e., low-indistinguishable) input event lists should lead to low-similar output lists [13]. In our simple definition of non-interference (Definition 2), low-similarity of input event lists was instantiated as "equality after removing all high input events". In the presence of declassification policies we refine the notion of low-similarity: the declassification policy says that low observers can also observe *some* aspects of high input events.

If we think of a declassification policy $\mathcal{D}$ as defining a function $\mathcal{D}^*(I)$ that returns all the information in input event list I that low observers can depend upon, then we can define noninterference as follows (we define the function $\mathcal{D}^*$ more precisely below).

Definition 4 (L-similarity under $\mathcal{D}$): Two lists I and I' are L-similar under $\mathcal{D}$ iff $\mathcal{D}^*(I) = \mathcal{D}^*(I')$. We denote this by $I \approx_L^{\mathcal{D}} I'$.

Definition 5 (Noninterference under $\mathcal{D}$): A program p is noninterferent under a declassification policy $\mathcal{D}$ iff for all lists I, I', O, O' such that $p(I) \to^* O$ and $p(I') \to^* O'$ it holds that: $I \approx_L^{\mathcal{D}} I' \implies O \approx_L O'$.

It remains to define the function $\mathcal{D}^*(I)$. On occurrence of an input event $ev\ n$, the policy defines that the following information is observable: (1) the projected event, and (2) the current value on the release channel if the event does not project to Nothing. Since release can be state dependent, we first define the function $\mathcal{D}^*(s, r, I)$ that specifies what information low observers can depend upon if inputs I are received when the policy state is s and the current value on the release channel is r.

$$\begin{aligned}
\mathcal{D}^*(s, r, []) &= [] \\
\mathcal{D}^*(s, r, (ev\ n) :: I) &= \mathcal{D}^*(s', r', I) \\
&\quad \text{if } \mathcal{D}(s, r, ev\ n) = (s', r', \text{Nothing}) \\
\mathcal{D}^*(s, r, (ev\ n) :: I) &= [ev\ x, r'] +\!\!+\ \mathcal{D}^*(s', r', I) \\
&\quad \text{if } \mathcal{D}(s, r, ev\ n) = (s', r', \text{Project } x)
\end{aligned}$$

where $+\!\!+$ is a notation for list concatenation. So, for an event that projects to Nothing, no information is made available to low observers. Otherwise, the projected event and current value on the release channel are made available.

Finally, we can define $\mathcal{D}^*(I) = \mathcal{D}^*(S_{\text{init}}, R_{\text{init}}, I)$. When $\mathcal{D}$ consists of only projections desugared from a labelling of event types, the new definition coincides with Definition 2 (then $\mathcal{D}^*(I)$ is the list of low events in I).

IV. Enforcing declassification policies

We now set out to design an enforcement mechanism. We extend the technique of Secure Multi-Execution (SME) to handle the declassification policies. First, we briefly recap how plain SME works on reactive programs [9]. Then we introduce our extensions and prove security and precision.

A. Plain SME

SME [8], [14], [9] is a dynamic enforcement mechanism for information flow security with practical advantages when applied in the context of JavaScript web applications [8, §VI.D].

The core idea of SME for reactive systems [9] is to maintain two executions of the program: a low and a high execution. Low events are handled by both executions, and high events are only handled by the high execution. Outputs on low channels are only performed in the low execution and outputs on high channels only in the high execution. This technique assures noninterference. Additionally, if the program is noninterferent and low and high executions are scheduled correctly, outputs remain the same [15], [16]. 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 first execute the low execution, 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.

For a detailed overview of SME the reader is referred to [8], [9], [17], [15].

Listing 5: Declassify in the shortkey usage program

```
on KeyPress(x) { if x = 101 then
                    { keyPressed := 1 } else { skip } }
on Unload(x) { r := declassify keyPressed; Send(r) }
```

B. Declassify annotations

The core idea of our extension is to first run the declassification policy on each input event. This determines what should be passed to the low and high executions. Events that project to Nothing are only given to the high execution. For all other events, the low execution receives the projected event, and the high execution receives the original event.

It remains to specify how the low execution can access the release channel. In order for the program to do this, we extend the language with a declassify annotation. To the syntax in Figure 1 we add the following clause:

$$c ::= \ldots \mid g := \text{declassify } e$$

The programmer has to use this annotation to indicate where in the program he has computed the value that the policy allows to be declassified. Our enforcement mechanism will then process declassify expressions by reading from the release channel.

The annotation only has effect when the program is run under SME. Otherwise it is an identity function: for the standard semantics, $\to$, the command g := declassify e is handled as $g := e$. The programmer writes the annotation to indicate that the expression e evaluates (under the standard semantics) to exactly the value that is declassified by the policy on the release channel.

As an example, consider again the program in Listing 3 under the policy of Listing 4. If the program is run under SME, it will only output a 0 in response to the Unload event, as the low execution never gets to see any of the key presses. In order to benefit from the declassification policy, the developer should change his program as shown in Listing 5. Our enforcement mechanism will plug in the correct value, denoting whether the shortcut key was used, during execution of the declassify statement in the low execution.

Note that security does not depend on how the program uses declassify expressions: the low execution will only ever get to see values that were declassified by the policy, meaning an adversary cannot abuse them to leak secret information. We will formally prove that in the security theorem (Theorem 3). Declassify annotations are only required to improve precision for release policies: they make sure that *secure* programs that run under our enforcement mechanism run unmodified.

When declassify annotations are used, it is important that the program is run under the declassification policy expected by the programmer. Otherwise the value computed by the declassification policy might differ from the value computed by the expression e in the declassify command, leading to unexpected results. For instance, a policy $\mathcal{D}$ might declassify a number rounded down to three decimal places, yet the program itself could round the number to only one decimal place in the declassify command. In this case the behaviour of the program will be modified under SME. To formalize

$$\frac{}{(s, r, \mu, g \coloneqq \text{declassify } e) \xrightarrow{\cdot}_{\mathcal{D}} (s, r, \mu[g \mapsto r], \text{skip})} \tag{1}$$

$$\frac{\mathcal{D}(s, r, i) = (s', r', pr) \quad i = ev\ n \quad \text{lookup}(p, ev) = c}{(s, r, \mu, \text{skip}) \xrightarrow{i}_{\mathcal{D}} (s', r', \mu, c[x \leftarrow n])} \tag{2}$$

Figure 3: Extended model language semantics handling declassification policies.

this requirement, we first define a variation on the standard semantics where declassify is treated as reading from the release channel. This is done by extending the program configuration to (s, r, μ, c) where r is the value on the release channel and s is the policy state. Declassification commands are handled by assigning r to the appropriate variable, leading to the semantics defined in Figure 3. We only show the rules for declassify and for handling a new event; all other rules remain as in Figure 2.

We can define the notion of execution, event-complete execution and the judgment $p(I) \rightarrow^*_{\mathcal{D}} O$ analogous to how we defined these notions for the standard semantics (with proper initial values for s and r). Now we can formally define what it means for a program with declassify annotations to be compatible with a policy:

Definition 6 (Compatible declassification policy): A program p is compatible with a declassification policy $\mathcal{D}$ if for all inputs I we have $p(I) \rightarrow^* O$ iff $p(I) \rightarrow^*_{\mathcal{D}} O$.

In other words: treating declassify as reading from the declassification channel should always give the same results as treating declassify as the identity function.

C. SME with declassification

We now turn to the formalization of our enforcement mechanism. SME enforces policies by executing programs under the SME semantics. Under this semantics, a program configuration consists of the tuple $(s, r, (\mu_L, c_L), (\mu_H, c_H))$ where s is the state of the declassification policy, r the value on the release channel, and (μ_L, c_L) and (μ_H, c_H) the execution states of the low and the high execution.

The definition of the SME transition relation $\overset{\alpha}{\Rightarrow}$ is shown in Figure 4. The first two rules define how a new event must be handled. In both cases, the policy will process the event, leading to a new policy state and released value.

- (Rule SME-1a) If the event is projected to Nothing by the policy, then it is handled by the high execution only.
- (Rule SME-1b) If the event is projected to Project n' by the policy, then the projected event is handled by the low execution, and the original event is handled by the high execution.

The last two rules say that first the low execution runs until it has processed the event completely, and then the high execution runs until it has processed the event completely. Each of the two executions progresses with its own store according to the standard semantics of commands where declassify is handled by reading from the release channel as defined in Figure 3. Outputs are suppressed (turned into unobservable outputs) unless they are of the same level as the execution.

An SME program configuration of the form $(s, r, (\mu_L, \text{skip}), (\mu_H, \text{skip}))$ is called passive. That is, both the low and high executions are done handling the last input event.

We again define the notion of execution, event-complete execution and the judgment $p(I) \Rightarrow^* O$ analogous to how we defined these notions for the standard semantics.

We are now ready to prove the two main properties of our enforcement mechanism: security and precision.

D. SME with declassification is secure

The security property says that the execution of any program under SME with declassification policy $\mathcal{D}$ is noninterferent under that policy.

To avoid repeating the definition of SME states we use the convention that $S_i = (s_i, r_i, L_i, H_i)$. We extend this convention to S'_i and S''_i as expected. An SME state is either passive or it can make a deterministic step. Adding an element i to the beginning of a list I is denoted by $i.I$. The proof uses the following equivalence relation:

Definition 7 (L-Equivalent States): Given two SME states S_1 and S_2 we write $S_1 \approx_L S_2$ iff $L_1 = L_2$ and L_1 not passive implies $r_1 = r_2$.

Next we need a lemma which states that if two inputs are currently similar under a policy, then we can always take a step while maintaining this similarity.

Lemma 1 (Policy Preservation): The following properties hold given that $\mathcal{D}^*(s_1, r_1, i_1.I'_1) = \mathcal{D}^*(s_2, r_2, I_2)$:

1) If $\mathcal{D}(s_1, r_1, i_1) = (s'_1, r'_1, \text{Nothing})$ then we know that $\mathcal{D}^*(s'_1, r'_1, I'_1) = \mathcal{D}^*(s_2, r_2, I_2)$.

$$\frac{\mathcal{D}(s,r,i) = (s',r',\text{Nothing}) \quad c = \text{lookup}(p, ev)}{(s,r,(\mu_L,\text{skip}),(\mu_H,\text{skip})) \overset{ev\ n}{\Rightarrow} (s',r',(\mu_L,\text{skip}),(\mu_H, c[x \leftarrow n]))} \quad \text{(SME-1a)}$$

$$\frac{\mathcal{D}(s,r,i) = (s',r',\text{Project } n') \quad c = \text{lookup}(p, ev)}{(s,r,(\mu_L,\text{skip}),(\mu_H,\text{skip})) \overset{ev\ n}{\Rightarrow} (s',r',(\mu_L, c[x \leftarrow n']),(\mu_H, c[x \leftarrow n]))} \quad \text{(SME-1b)}$$

$$\frac{(s,r,\mu_L,c_L) \xrightarrow{o}_{\mathcal{D}} (s,r,\mu_L',c_L') \quad \textbf{if } \sigma(o) = L \textbf{ then } o' = o \textbf{ else } o' = \cdot}{(s,r,(\mu_L,c_L),(\mu_H,c_H)) \overset{o'}{\Rightarrow} (s,r,(\mu_L',c_L'),(\mu_H,c_H))} \quad \text{(SME-2)}$$

$$\frac{(s,r,\mu_H,c_H) \xrightarrow{o}_{\mathcal{D}} (s,r,\mu_H',c_H') \quad \textbf{if } \sigma(o) = H \textbf{ then } o' = o \textbf{ else } o' = \cdot}{(s,r,(\mu_L,\text{skip}),(\mu_H,c_H)) \overset{o'}{\Rightarrow} (s,r,(\mu_L,\text{skip}),(\mu_H',c_H'))} \quad \text{(SME-3)}$$

Figure 4: Small-step semantics of SME with declassification.

2) If it holds that $\mathcal{D}(s_1, r_1, i_1) = (s_1', r_1', \text{Project } x_1)$ and $\mathcal{D}(s_2, r_2, i_2) = (s_2', r_2', \text{Project } x_2)$ with $I_2 = i_2.I_2'$ then $x_1 = x_2$ and $\mathcal{D}^*(s_1', r_1', I_1') = \mathcal{D}^*(s_2', r_2', I_2')$.

Proof: This follows directly from the definition of $\mathcal{D}^*$. ■

For the security proof we introduce a new notation denoting an execution starting from a given state S: we write $S(I) \Rightarrow^* O$ iff there exists an event-complete execution $\overline{\alpha}$ of S where I is exactly the list of input events occurring in $\overline{\alpha}$ and O is the list of output events occurring in $\overline{\alpha}$. An execution $\overline{\alpha}$ of state S is a sequence of transitions $\overline{\alpha}$ starting from S.

Lemma 2 (State Security): If $S_1 \approx_L S_2$, $S_1(I_1) \Rightarrow^* O_1$, $S_2(I_2) \Rightarrow^* O_2$, and $\mathcal{D}^*(s_1, r_1, I_1) = \mathcal{D}^*(s_2, r_2, I_2)$, then we have $O_1 \approx_L O_2$.

Proof: The proof is by induction on the execution $\overline{\alpha}$ of $S_1(I_1) \Rightarrow^* O_1$.

Base case. Given an empty execution $\overline{\alpha}$, I_1 and O_1 are also empty. Hence $\mathcal{D}^*(s_1, r_1, I_1)$ and $\mathcal{D}^*(s_2, r_2, I_2)$ are empty as well, meaning that I_2 can only contain non-projected inputs, and that rule SME-1b is never used in $S_2(I_2) \Rightarrow^* O_2$. Additionally we know that S_1 must be passive, so from $S_1 \approx_L S_2$ we know that L_2 is also passive. Since SME-1b is not used in $S_2(I_2) \Rightarrow^* O_2$, and L_2 is passive, SME-2b is also not used in $S_2(I_2) \Rightarrow^* O_2$. Therefore O_2 contains no low outputs, proving that $[] = O_1 \approx_L O_2$.

Inductive Step. We assume it holds for $\overline{\alpha}$ and prove it for $\alpha_0.\overline{\alpha}$. Either α_0 is an output o or an input i, leading to the following two cases:

$S_1 \overset{o}{\Rightarrow} S_1'$: If $\sigma(o) \neq$ L then we know rule SME-3 was used and only H_1 has changed, from this it follows that $S_1' \approx_L S_2$ and $\mathcal{D}^*(s_1', r_1', I_1) = \mathcal{D}^*(s_2, r_2, I_2)$. Since o is a high output we also have $S_1'(I_1) \Rightarrow^* O_1'$ with $O_1 \approx_L O_1'$. Applying the induction hypothesis gives $O_1' \approx_L O_2$ and thus $O_1 \approx_L O_2$.

If $\sigma(o) =$ L then rule SME-2 was used. From $S_1 \approx_L S_2$ we know that SME-2 can also be applied to S_2 and that it produces the same output and low execution state. Hence we have $S_2 \overset{o}{\Rightarrow} S_2'$ with $S_1' \approx S_2'$, $S_2'(I_2) \Rightarrow^* O_2'$, and $S_1'(I_1) \Rightarrow^* O_1'$ with $O_2 = o.O_2'$ and $O_1 = o.O_1'$. Since s and r are not modified by SME-2 we know that $\mathcal{D}^*(s_1', r_1', I_1) = \mathcal{D}^*(s_2', r_2', I_2)$. From the induction hypothesis we get $O_1' \approx_L O_2'$ and thus $O_1 \approx_L O_2$.

$S_1 \overset{i}{\Rightarrow} S_1'$: Assuming that $\mathcal{D}(s_1, r_1, i) = (s_1', r_1', \text{Nothing})$ we can apply Lemma 1 case 1 to obtain that $\mathcal{D}^*(s_1', r_1', I_1') = \mathcal{D}^*(s_2, r_2, I_2)$ with $I_1 = i.I_1'$. Since SME-1a must be used we have $S_1' \approx_L S_2$ and $S_1'(I_1') \Rightarrow^* O_1$. Applying the induction hypothesis results in $O_1 \approx_L O_2$.

If $\mathcal{D}(s_1, r_1, i) = (s_1', r_1', \text{Project } x)$ then rule SME-1b was used in $S_1 \overset{i}{\Rightarrow} S_1'$ and $\mathcal{D}^*(s_1, r_1, I_1)$ starts with a projected event $ev\ x$. Hence $\mathcal{D}^*(s_2, r_2, I_2)$ also contains the same projected event $ev\ x$, meaning at some point SME-1b must be used in $S_2(I_2) \Rightarrow^* O_2$. We now show that the first state S_2' to which SME-1b is applicable, is L-equivalent with S_2, and that no low outputs are generated in the execution from S_2 to S_2'. Since $S_1 \approx_L S_2$ we know that L_2 is passive and that SME-2 isn't applicable to S_2. Rules SME-1a and SME-3 don't change L_2, meaning after applying them rule SME-2 still isn't applicable. We conclude that, starting with S_2, we keep executing either SME-1a or SME-3 until we get to a state S_2' for which SME-1b is applicable.

In other words there exists an S_2' such that $S_2' \overset{i'}{\Rightarrow} S_2''$ with $S_2''(I_2'') \Rightarrow^* O_2''$, $S_1 \approx_L S_2'$, and $O_2 \approx_L O_2''$. Since SME-2 doesn't change s_2 and r_2, and we can apply Lemma 1 case 1 after using SME-1a, we also have $\mathcal{D}^*(s_2, r_2, I_2) = \mathcal{D}^*(s_2', r_2', I_2')$. Applying Lemma 1 results in $\mathcal{D}^*(s_1', r_1', I_1') = \mathcal{D}^*(s_2'', r_2'', I_2'')$. As the same value x is projected for both S_1 and S_2' we get $S_1' \approx_L S_2''$. We also know that $S_1'(I_1') \Rightarrow^* O_1$. From the induction hypothesis it follows that $O_1 \approx_L O_2''$ and hence $O_1 \approx_L O_2$. ■

With this lemma we can prove the main security theorem.

Theorem 3 (Security): Execution of any program p under SME with declassification policy $\mathcal{D}$ is noninterferent under $\mathcal{D}$.

Proof: This follows directly from lemma 2. ∎

E. SME with declassification is precise

The SME execution of a program can potentially change the behaviour of the program. For example, if the program is not noninterferent under $\mathcal{D}$, the security theorem tells us that the behaviour will definitely change, as it will become noninterferent. Hence it is important to show that the behaviour of *secure* programs does not change observably. As mentioned in Section IV-A, depending on the scheduler used, plain SME can change the behaviour of noninterferent programs in the sense that outputs can be reordered. But within a given security level outputs remain in the same order.

Definition 8 (Observer Indistinguishable): Let $\ell \in \{L, H\}$ and let S and S' be lists. We say $S =_\ell S'$ if the two lists are equal after dropping all elements having $\sigma(s) \neq \ell$. We say two lists S and S' are observer indistinguishable, denoted by $S \approx^{\text{obs}} S'$, iff $S =_L S'$ and $S =_H S'$.

As explained previously, in many cases maintaining order within security levels is sufficient.

Now we want to prove precision, the property that "good" programs produce identical outputs when run with or without our enforcement mechanism, i.e., that our mechanism is *transparent* for secure programs.

We begin by proving precision for high observers.

Lemma 4 (Precision for H observers): If $p(I) \rightarrow^* O$, $p(I) \Rightarrow^* O'$, and $\mathcal{D}$ is compatible with p, then $O =_H O'$.

Proof: Let $S = (s, r, (\mu_L, c_L), (\mu_H, c_H))$ be a state in the SME semantics, and let $S' = (s', r', \mu, c)$ be a state in the extended model semantics $\rightarrow_\mathcal{D}$ handling declassification policies. We say $\mathcal{R}(S, S')$ if and only if $c = c_H$, $\mu = \mu_H$, $s = s'$, and $r = r'$. It is easy to check that this relation is a simulation and that the produced high outputs are identical. Hence it follows from $p(I) \Rightarrow^* O'$ that $p(I) \rightarrow^*_\mathcal{D} O''$ with $O' =_H O''$. Applying definition 6 of a compatible declassification policy results in $O =_H O'$. ∎

For low observers, precision is more intricate. Ideally the mechanism enforcing a declassification policy $\mathcal{D}$ would be transparent for any program that is noninterferent under it. In case a projection-only policy is used, we can deliver such strong guarantees. A projection-only policy is one which does not define a *release* function. When such a policy is used the program requires no annotations in order to run properly under our enforcement mechanism. Intuitively, a program which is noninterferent under a projection-only policy internally "projects" the inputs itself. Since we require that such projections must be idempotent, letting SME project the input before giving it to the program will not change its behaviour.

Proving this requires the following lemma, describing the behaviour of the low execution using the $\rightarrow_\mathcal{D}$ semantics. The notation $\mathcal{D}^*_{ev}(I)$ denotes $\mathcal{D}^*(I)$ with all released values filtered, i.e., only containing events.

Lemma 5 (Behaviour Low Execution): Given $p(I) \Rightarrow^* O$, for all I' and $\mathcal{D}'$ with $I' = \mathcal{D}^*_{ev}(I)$ and $\mathcal{D}^*(I) = \mathcal{D}'^*(I')$, we have $p(I') \rightarrow^*_{\mathcal{D}'} O'$ with $O =_L O'$.

Proof: This can be proven by means of a straightforward simulation between the execution of $p(I) \Rightarrow^*_\mathcal{D} O$ and $p(I') \rightarrow^*_{\mathcal{D}'} O'$. The only non-trivial case is the handling of inputs, for which we need to rely on $I' = \mathcal{D}^*_{ev}(I)$ and $\mathcal{D}^*(I) = \mathcal{D}'^*(I')$. ∎

Using this Lemma we can prove precision for low observers when using a projection-only policy.

Lemma 6: If $p(I) \rightarrow^* O$ and $p(I) \Rightarrow^* O'$ with p compatible with, and noninterferent under, a projection-only policy $\mathcal{D}$, then $O =_L O'$.

Proof: Let $I' = \mathcal{D}^*_{ev}(I)$. We first prove $\mathcal{D}^*(I) = \mathcal{D}^*(I')$ using induction. The base case is when I is the empty list, for which it is trivially true. For the induction hypothesis we assume $\mathcal{D}^*(I) = \mathcal{D}^*(\mathcal{D}^*_{ev}(I))$ and prove $\mathcal{D}^*(i.I) = \mathcal{D}^*(\mathcal{D}^*_{ev}(i.I))$. Note that line 2 in the definition of $\mathcal{D}$ is never executed because $\mathcal{D}$ is a projection-only policy, and thus that r is never modified and always remains zero. We now consider the two cases in the definition of $\mathcal{D}^*$:

Input not Projected. In this case nothing is added to the output. Hence $\mathcal{D}^*(i.I_s) = \mathcal{D}(I_s)$, from which the desired result follows.

Input Projected. Now $ev\ x$ is added to the output followed by r, which we know is zero for both lists. In $\mathcal{D}^*(\mathcal{D}^*_{ev}(i.I))$ the projection function returns $ev\ x$ the second time it is applied, since it is a idempotent projection. We get $\mathcal{D}^*(\mathcal{D}^*_{ev}(i.I)) = ev\ x.0.\mathcal{D}^*(\mathcal{D}^*_{ev}(I))$ and $\mathcal{D}^*(i.I) = ev\ x.0.\mathcal{D}^*(I)$, applying the induction hypothesis completes this case.

We conclude that $\mathcal{D}^*(I) = \mathcal{D}^*(I')$. Applying Lemma 5 on $\mathcal{D}^*(I) = \mathcal{D}^*(I')$ and $p(I) \Rightarrow^*_\mathcal{D} O'$ results in $p(I') \rightarrow^*_\mathcal{D} O''$ with $O'' =_L O'$. Since p is compatible with $\mathcal{D}$ we get $p(I') \rightarrow^* O''$.

As p is noninterferent under $\mathcal{D}$, $p(I') \rightarrow^* O''$, $p(I) \rightarrow^* O$, and $\mathcal{D}^*(I) = \mathcal{D}^*(I')$, we get $O'' =_L O$. We conclude that $O =_L O'$. ∎

Theorem 7 (Projection Precision): If $p(I) \rightarrow^* O$, $p(I) \Rightarrow^* O'$, p has no declassify statements and is noninterferent under a projection-only policy $\mathcal{D}$, then $O \approx^{\text{obs}} O'$.

Proof: The policy is vacuously compatible with $\mathcal{D}$, as there are no declassify statements. Applying lemma 4 and lemma 6 completes the proof. ∎

Though this only covers projection-only policies, it is a powerful result. Most examples discussed so far can be handled using a projection-only policy.

For information release policies, precision for low observers is subject to more conditions. Consider for instance the program in Listing 3 running under the policy in Listing 4. The program is noninterferent under the policy, and the policy is vacuously compatible with the program since there are no declassify statements. But for low observers there will be a difference if the program is run with key presses of the shortcut key as input. It will perform Send 1 if it is run without enforcement mechanism and Send 0 if it is run under our enforcement mechanism. The problem is the lack of declassify annotations: the programmer should explicitly annotate information that he wants to declassify. The program in Listing 5 adds the necessary declassify annotation so the program runs correctly under SME.

Hence the additional requirement is that there must be "enough" declassify annotations. A first attempt to formalize this could be to require that the program must be noninterferent after removal of declassify statements: removing all the assignments with a declassify annotation should remove all information leaks from H to L. But that is not sufficient, as removal of the declassify statements may render some parts of the program unreachable. For instance, consider the program in Listing 6, where num can be any number. If in some run the policy would declassify num the program would be interferent as it leaks gps coordinates in the then-branch of the conditional on line 3.

Listing 6: Information leak outside of declassify

```
1 on GpsUpdate(x) { g = x }
2 on Unload(x) { d = declassify e;
3                  if d = num then { Send(g) } else { skip } }
```

Intuitively we want to assure that replacing the result of any reachable declassify command with any constant, always results in low-equivalent outputs (given equivalent inputs).

Definition 9 (No leaks outside declassify): A program p does not leak outside declassify under policy $\mathcal{D}$ if for all inputs I, I' and policies $\mathcal{D}', \mathcal{D}''$ having the same projection functions as $\mathcal{D}$, it holds that $\mathcal{D}'^*(I) = \mathcal{D}''^*(I')$, $p(I) \rightarrow^*_{\mathcal{D}'} O$, and $p(I') \rightarrow^*_{\mathcal{D}''} O'$ implies $O \approx_L O'$.

The reasoning behind this definition is that, if we quantify over all declassification policies, we are effectively replacing all reachable declassify commands with any constant. Since we want to do this without changing the security labellings of inputs we have to require that the projection functions are identical. Finally we can easily require that inputs are equivalent by stating that $\mathcal{D}'^*(I) = \mathcal{D}''^*(I')$. For example, consider again Listing 6, where num can be any number, under a policy $\mathcal{D}$ labelling GpsUpdates as high security inputs. Two policies $\mathcal{D}'$ and $\mathcal{D}''$ declassifying num at the appropriate time will detect that it leaks outside of declassify (given two inputs where one contains a GPS update and the other one does not). Furthermore, if a program tries to declassify information without using annotations, it would be detected when $\mathcal{D}'$ and $\mathcal{D}''$ are instantiated to policies without release functions.

With this definition precision for L observers can be proven.

Lemma 8 (Precision for L observers): If $p(I) \rightarrow^* O$, $p(I) \Rightarrow^* O'$, $\mathcal{D}$ compatible with p, and p does not leak outside declassify commands, then $O' \approx_L O$.

Proof: Let $I' = \mathcal{D}^*_{ev}(I)$. We construct a declassification policy $\mathcal{D}'$ such that $\mathcal{D}^*(I) = \mathcal{D}'^*(I')$. This can be done by using the same projection functions as in $\mathcal{D}$ and using the following release function:

```
1 state :: Int = 0
2 release(n, input) = (n+1, Release $\mathcal{D}^*(I)$[2n + 1])
```

Where $\mathcal{D}^*(I)$[i] returns the i-th entry in the list $\mathcal{D}^*(I)$ and zero if i is out of bounds (the index is zero based). Because projections must be idempotent, and I' has half the length of $\mathcal{D}^*(I)$, it is now straightforward to prove that $\mathcal{D}^*(I) = \mathcal{D}'^*(I')$. Applying Lemma 5 on $p(I) \Rightarrow^* O'$ and $\mathcal{D}^*(I) = \mathcal{D}'^*(I')$ results in $p(I') \rightarrow^*_{\mathcal{D}'} O''$ with $O'' \approx_L O'$.

From $p(I) \rightarrow^* O$ and the fact that $\mathcal{D}$ is compatible with p it follows that $p(I) \rightarrow^*_{\mathcal{D}} O$. We now have $p(I) \rightarrow^*_{\mathcal{D}} O$, $p(I') \rightarrow^*_{\mathcal{D}'} O''$, and $\mathcal{D}^*(I) = \mathcal{D}'^*(I')$ where $\mathcal{D}$ and $\mathcal{D}'$ share the same projection functions. These are precisely the conditions needed to apply definition 9, namely that p does not leak outside of declassify. We get $O \approx_L O''$.

From $O'' \approx_L O'$ and $O \approx_L O''$ it follows that $O \approx_L O'$. ■

Since precision for both low and high observers has been proven, we can conclude with:

Theorem 9 (Full Precision): Given that $p(I) \rightarrow^* O$ and $p(I) \Rightarrow^* O'$, $\mathcal{D}$ compatible with p, and that p does not leak outside declassify under $\mathcal{D}$, then $O \approx^{\text{obs}} O'$.

V. Implementation in a real browser

To show both practicality and usefulness of our approach, we have implemented the mechanism for full JavaScript in a real-world browser by extending and enhancing the SME-enabled browser FlowFox [3], [18]. Our implementation supports writing dynamic declassification policies in JavaScript and is successfully tested on real-life examples. The implementation, together with some examples and the Redex mechanization of the formal model are all available for download [11].

A. Implementation Details

FlowFox is a modified Mozilla Firefox browser where each JavaScript program is executed within a specific JavaScript context subject to the SME I/O rules. Each implementation of a JavaScript API method is wrapped with code that consults the policy and enforces the SME I/O rules. This section will cover our updates and modifications to FlowFox to enable stateful declassification support.

The original FlowFox handles events as follows: both the low and the high execution can set handlers. FlowFox saves the security level for each handler that gets installed. For example, if a JavaScript program installs an event handler for the onload event by setting window.onload, FlowFox will run this program for each available security level and as a result will store two event handlers (one to be executed in the low context and one in the high context). Whenever the onload event is triggered, FlowFox looks up all available event handlers for the event and executes each one – within the correct JavaScript context with the associated security level – sequentially. Hence, events can be made H by labelling the methods that set handlers for the event as H; in that case the low execution will skip setting the handler and hence will never receive the event.

To generalize this to our projection policies, we change the code responsible for handling events. For each event that enters the system, first the release function will be called so that it can update the state space and the value on the release channel. Before dispatching the event to its registered event handler(s), the policy must also be consulted about event projection. If the event projects to Nothing, the event is not dispatched to L handlers. All other types of projections are achieved by carefully modifying the return value of the relevant API methods of the associated event object in order to mimic the behavior of the projection function. Finally, we extended the global scope, available for JavaScript programs on web pages, with a new declassify operator that returns the value of the release channel contained within the policy.

The last place where we had to change FlowFox, is the code for events that are not strictly DOM events, but still call a callback function after a specific task has finished. This is for example the case with the getCurrentPosition method of the geolocation object. The method requires two callback functions – one for a successful lookup and one to handle a failure. In the background, the geolocation implementation will try several strategies to acquire the user's position. When successful the callback will be executed as if it was an event handler, so the same logic for event handlers needs to be applied. We implemented similar logic as for DOM event handlers for the geolocation events.

B. Examples

To test our implementation and to show both practicality and usefulness, we implemented some examples from this paper in full JavaScript, and we verified by means of testing that they behave as expected. As an example, an annotated shortkey monitoring script is given in Listing 7, along with the corresponding information release part of the FlowFox policy in Listing 8. The policy has the same behaviour as the one given in Listing 4 for the formal model. A policy written for FlowFox updates the state and release channel by setting stateSpace and releaseValue, respectively. A more detailed explanation of the policy syntax, together with more examples, are available for download [11].

Listing 7: JavaScript Program

```
var d=0, u='http://analytic.com/?=';
window.onkeypress = function(e) {
    if (e.charCode == 101) d = 1; }
window.onunload = function() {
    v = declassify(d);
    $.ajax(u + v);
}
```

Listing 8: Information Release Policy

```
SME.stateSpace = false;
SME.release = function (ev, arg) {
    if (ev == 'keypress' && !this.stateSpace
          && arg.charCode == 101) {
              this.stateSpace = true;
              this.releaseValue = 1;
} };
```

As with all policy-based mechanisms, an important question is: who will write the policies? It is not realistic to expect browser users to do this. In our view, at least two approaches are feasible. For simple and widely useful declassification policies (such as the approximate GPS location policy) the browser can offer them as part of a privacy profile. For more application-specific policies, policies can be server-driven: the integrator of a web mashup will set a policy and push it to the browser. The policy will need to be agreed upon between the integrator and the various script providers. Legacy scripts (not using declassify annotations) are enforced to be noninterferent (without declassification), whereas scripts that have been annotated are enforced to be noninterferent under the enforced declassification policy. The browser user no longer needs to trust third-party scripts to protect his data, but only the main site he interacts with.

Obviously, it is important to come up with a better surface syntax for the policy language; but this is an orthogonal problem not addressed in this paper.

C. Benchmarks

We performed benchmarks to give evidence that the performance cost of our approach is acceptable. All measurements we report below were done on a virtualized Ubuntu 11.04 with 2.6GHz and 3.5GB RAM.

The performance cost of our approach can be broken down in (1) additional processing for each event (consulting the policy, updating the state space, and checking if an event handler for an event must be skipped), and (2) the cost of calling the declassify function in a handler. The cost of updating the policy store depends on the policy being enforced, but we expect policies to be simple computations (e.g., computing an average or rounding a result) so the main costs are the additional context switches between JavaScript and C++.

To measure the cost of (1), we conducted an experiment in which FlowFox visits a web page that triggers 1000 rounds of 1000 clicks via JavaScript on an HTML button with an empty event handler, and we measure how long it takes to handle these 1000 events. The average time for handling 1000 clicks (averaged over 1000 runs) was 136 ms, i.e., handling a single event costs around 0.14 ms.

The cost of (2) can be expected to be fairly low, as it is just a getter of a value set by the policy. To measure it, we conduct an experiment that triggers an event handler that calls the declassify function 1000 times. The average cost of a single call to declassify was 5.5 μs, which is mostly due to the switching cost between JavaScript and C++ (and back).

The setup of our macro benchmark experiment consisted of an automated version of both the short key example from Listing 2 and a mouse click example that leaks the average mouse position after multiple clicks. The policy file consisted of an aggregation of the first two declassification policies from Section III-B. The timing results between FlowFox with and without declassification support, show an insignificant overhead.[1]

We believe all these results are evidence of the fact that adding declassification to FlowFox has a negligible impact on the overall performance of the browser.

VI. Discussion

Limitations of the approach: Our approach to declassification is the first extensional fine-grained approach to declassification for SME. We believe that our theoretical results and implementation experience show that it is a useful approach. However, there also some disadvantages or limitations with respect to competing approaches.

First, because declassification policies are essentially programs (in a declarative functional language), understanding what information a given policy leaks can be complex. So mistakes in policies are more likely than for approaches that use simpler (e.g. type based) specifications of information flow policies. There is an inherent trade-off between (1) supporting more expressive policies and (2) making policies easy to analyze. Our policies emphasize expressiveness over analyzability.

Second, by giving an *extensional* definition of secure information flow, our approach can not handle policies that depend on the structure of the program being monitored. Consider for instance, a policy such as: declassify all keypress events, except the ones that are sent to a credit-card number submit form. This can not be specified as an extensional policy, as only the program knows which form is the credit-card number submit form. Such policies can be specified in approaches that support *intensional* policies, usually by allowing trusted parts of the program to declassify values. For instance, in the approach of Rafnsson and Sabelfeld [15], such intensional policies can be supported. The flip-side of the coin is that this makes it harder to give guarantees on programs from untrusted sources (such as third party analytics scripts). Rafnsson and Sabelfeld's approach can only impose coarse grained extensional limitations on the power of declassification (essentially specifying between what labels of the security lattice information can potentially be declassified by means of their release policy ρ). In our approach the declassify annotations are not trusted, and we can enforce more fine grained policies on programs from untrusted sources.

It is interesting to note that – while our policies and our definition of noninterference are purely extensional – our enforcement mechanism in addition gives guarantees about *where* declassification happens: it can only happen in parts of the program that the programmer has annotated with declassify annotations. But these declassify annotations are *not* trusted: if the programmer would cheat, and leave out a declassify annotation, our enforcement mechanism would modify executions of the program so that they become secure. Similarly, if a programmer would cheat and use declassify annotations to leak information that is not allowed to leak by the policy, the enforcement mechanism would modify the execution of the program to make it secure. So there is no need to review untrusted code for appropriate use of declassify annotations to ensure security. Inappropriate use of declassify only harms precision.

Simplifications in the formal model: There are some simplifications in our formal model, and we discuss how they impact the practicality of our approach.

A first simplification is that we have restricted our attention to event-complete executions, essentially assuming that handling a single event always terminates. This allows us to work with a simple notion of termination-insensitive noninterference. However, it is well-known that programs that can do many outputs (as our event handlers) can leak a substantial amount of information even if they are termination-insensitively non-interferent [19]. Fortunately, it is easy to see that our enforcement mechanism only leaks one bit (whether the event handler terminates or not). This follows for instance from the fact that noninterference holds for any approximation of a non-terminating event handler that performs the same first n steps and then terminates. While it would be possible to prove security for a stronger notion of noninterference (that also puts demands on non-event-complete executions), this would complicate the theory without significant practical benefits.

Second, our model language is significantly simpler than JavaScript. Hence, an important questions is in how far our approach extends to more complex language features.

[1]The source code of the scenarios and the exact timing results and logs of the experiments can all be downloaded from [11].

Fortunately, a nice aspect of SME is that it is *language-independent*: it treats the language as a black box [15]. This language-independence makes it feasible to implement SME-based enforcement relatively easily even for real and complex languages like JavaScript.

Third, the event model and API available to scripts in our formal model is significantly simpler than the actual DOM API that JavaScript can use in web applications. This gap between the formal model and actual web applications is more important. SME has to deal with all I/O operations and so wherever the I/O behaviour is more complex in the real system than in the model there is a significant engineering challenge and a risk for introducing vulnerabilities.

From a foundational point of view: treating events and API calls as inputs and outputs with a specific noninterference policy only makes sense if the context that provides these events and implements these API calls does not itself leak information (e.g. if a high output API call triggers a low event, there is an information leak in the context, and scripts could use this to "launder" information).

From a practical point of view: ensuring that a real browser (the "context" for our scripts) is fine in this sense is non-trivial. The implementation currently addresses this in the same way as FlowFox does [18]: by imposing on the policy writer the obligation to write policies that are such that the context does not leak information with respect to these policies. The complexity of API and event model in real browsers makes this a difficult obligation for the policy writer.

There is no simple solution here: a browser is a complex beast, and any formal model must simplify tremendously. We believe our model makes the right simplifications to study the essence of declassification for event driven programs, but the gap with a real system is of course still significant. The model helps us engineer declassification into FlowFox in the right way, but our formal proofs apply only to the model not to the full browser.

VII. Related Work

A. Information flow security and declassification

Sabelfeld and Sands [20] survey information declassification policies and enforcement mechanisms, and provide a set of dimensions for declassification models. The identified dimensions take into account the nature of the released data, its location in the program, its time, and its owner (the so-called *what*, *where*, *when* and *who* dimensions). These dimensions have been extensively studied in the literature [21], [22], [23], [24], [25], [26], [27]. We discuss closely related work and refer to the survey for a wider overview.

Declassification policies: Li and Zdancewic [28] have proposed *relaxed noninterference*. Like our notion of noninterference, this is an extensional security property that specifies the declassification policy in a simple separate language. They consider such policies to be security levels and study enforcement by means of type systems. For enforcement by means of SME, an approach that considers policies as levels would likely be expensive, as the performance cost of SME grows linearly with the number of levels.

Sabelfeld and Myers [29] present *delimited release*, a security policy to declassify information through *escape hatches* specified using a special language operator declassify(e), where e is globally considered as declassified (*what* dimension). Our information release mechanism is similar to such escape hatches: one can think of the release function as specifying an escape hatch expression on the entire list of inputs seen so far. Later, Askarov and Sabelfeld [30] combined the *what* and *where* dimensions into the definition of *localized delimited release*. They use an instrumented semantics with the set of declassified expressions. Magazinius et al. [31] proposed decentralized delimited released, combining the *what* and *who* dimensions in decentralized settings and consider a web attacker model. In the context of multiple facets [5], Austin and Flanagan show how to perform robust declassification (*who* dimension) with integrity guarantees against active attackers.

Our security policy is inspired by *delimited release*, and thus released information is considered as declassified in the *what* dimension. However, SME enforcement releases information only where a declassify e expression appears. Thus released information can be considered as declassified in the *where* dimension. In fact, our precision theorem uses as hypothesis that the program does not leak outside of declassify expressions. Thus, information can only be released where the programmer explicitly marks the program.

There are some existing approaches in the literature where declassification policies can be computationally specified by means of a program, as in our work. Fournet and Planul [32] propose to implement declassification code in a trusted platform module in order to minimize trust assumptions. The idea of handling declassification in SME by means of an additional subprogram was mentioned by Kashyap et al. [17] as possible future work. Swamy and Hicks [33] specify information release policies as security automata in the AIR language, and use a combination of program transformation and static typing to ensure that programs in a core functional programming language comply with the policy.

Rafnsson and Sabelfeld [15] extend the technique of SME in many directions, one of them being declassification. Like us, they distinguish between occurrence and content of events. For input channels, our projection policies generalize their distinction. This generality allows us to write a policy to only project events satisfying some condition, yet hide the occurrence of others. For example, in our declassification framework we can express a policy that reveals shortcut key presses while hiding the occurrence of other keys (see Section 3.1). For output channels, Rafnsson and Sabelfeld also support the occurrence/content distinction, allowing them to support intensional declassification policies. In constrast, our declassification policies are extensional, enabling safe execution of programs from an untrusted source, as discussed in Section VI .

Information flow policies in web scripts: Information flow control in web scripts is usually proposed by means of dynamic mechanisms [34] due to the dynamic nature of the JavaScript language, the de facto programming language on the client side web applications. Several authors [5], [2], [4] study runtime monitors for noninterference in JavaScript-like languages. Bohannon et al. [12] propose reactive noninterference, a noninterference property for reactive programs such as web scripts that is proposed to replace the Same Origin Policy in browsers, and Bohannon and Pierce [35] developed Featherweight Firefox, a formal model of a simple browser, with the purpose of formally studying confidentiality and integrity policies for browsers, including reactive noninterference policies. Bielova et al. [9] later propose an enforcement mechanism for reactive noninterference based on secure multi-execution and implement it in the Featherweight Firefox browser model. Barthe et al. [36] implement SME by program transformation and report on Python and JavaScript prototypes. None of these works target declassification policies. Our proposal is based on dynamic enforcement by secure multi-execution [8]. Secure multi-execution was implemented in the Flowfox browser [3], [18], but could only handle noninterference policies before this work. To the best of our knowledge, decentralized delimited released [31] and multiple facets [5] are the only works that attempt to control declassication in real web scripts. In particular the work targets a subset of the JavaScript language.

B. Other web script security countermeasures

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 / sandboxing: 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 [37], ADSafe [38], Caja [39], and JSand [40]. Maffeis et al. [41] 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. Isolation-based systems do not deal with declassification: if access is given to information, the script can send any derived information through any output methods in the API made available to the script.

Taint tracking: 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. [42], who propose taint-enhanced policy enforcement as a general approach to mitigate implementation-level vulnerabilities, and Vogt et al. [43] who propose taint tracking to defend against cross-site scripting. In taint-tracking systems, declassification happens through trusted code that removes the taint. For instance, taint can be removed after passing the input through an input sanitization method.

VIII. Conclusion

Motivated by the desire to support web analytics in information flow secure browsers, we have proposed an approach to specify declassification policies for event-driven programs and we have developed an enforcement mechanism that we proved to be secure and precise. We implemented the mechanism in an existing information flow secure browser thus providing evidence of practicality.

The implementation, some example policies and scripts, and the Redex mechanization of the formal model are all available for download [11].

Acknowledgements

This research is partially funded by the Research Fund KU Leuven, by the EU FP7 projects WebSand and NESSoS, and by the IWT project SPION. 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] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications," in *ACM CCS*, 2010.

[2] D. Hedin and A. Sabelfeld, "Information-Flow Security for a Core of JavaScript," in *CSF*, 2012.

[3] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens, "FlowFox: a Web Browser with Flexible and Precise Information Flow Control," in *ACM CCS*, 2012.

[4] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer, "Information Flow Control in WebKit's JavaScript Bytecode," in *POST*, 2014.

[5] T. H. Austin and C. Flanagan, "Multiple Facets for Dynamic Information Flow," in *POPL*, 2012.

[6] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions," in *ACM CCS*, 2012.

[7] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna, "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis," in *NDSS*, 2007.

[8] D. Devriese and F. Piessens, "Noninterference Through Secure Multi-Execution," in *IEEE Symposium on Security and Privacy*, 2010.

[9] N. Bielova, D. Devriese, F. Massacci, and F. Piessens, "Reactive non-interference for a browser model," in *NSS*, 2011.

[10] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler, "Run Your Research: On the Effectiveness of Lightweight Mechanization," in *POPL*, 2012.

[11] https://distrinet.cs.kuleuven.be/software/FlowFox/.

[12] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, "Reactive Noninterference," in *ACM CCS*, 2009.

[13] A. Sabelfeld and D. Sands, "A Per Model of Secure Information Flow in Sequential Programs," *Higher Order Symbol. Comput.*, vol. 14, no. 1, pp. 59–91, Mar. 2001.

[14] R. Capizzi, A. Longo, V. Venkatakrishnan, and A. Sistla, "Preventing Information Leaks through Shadow Executions," in *ACSAC*, 2008.

[15] W. Rafnsson and A. Sabelfeld, "Secure multi-execution: fine-grained, declassification-aware, and transparent," in *CSF*, 2013.

[16] D. Zanarini, M. Jaskelioff, and A. Russo, "Precise enforcement of confidentiality for reactive systems," in *CSF*, 2013.

[17] V. Kashyap, B. Wiedermann, and B. Hardekopf, "Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach," in *IEEE Symposium on Security and Privacy*, 2011.

[18] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens, "Secure multi-execution of web scripts: Theory and practice," *Journal of Computer Security*, 2014.

[19] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *ESORICS*, 2008.

[20] A. Sabelfeld and D. Sands, "Dimensions and Principles of Declassification," in *CSF*, 2005.

[21] S. Chong and A. C. Myers, "Language-based Information Erasure," in *CSF*, 2005.

[22] A. A. Matos and G. Boudol, "On Declassification and the Non-Disclosure Policy," in *CSF*, 2005.

[23] N. Broberg and D. Sands, "Flow Locks: Towards a Core Calculus for Dynamic Flow Policies," in *ESOP*, 2006.

[24] H. Mantel and A. Reinhard, "Controlling the What and Where of Declassification in Language-Based Security," in *ESOP*, 2007.

[25] G. Barthe, S. Cavadini, and T. Rezk, "Tractable Enforcement of Declassification Policies," in *CSF*, 2008.

[26] S. Chong and A. C. Myers, "End-to-End Enforcement of Erasure and Declassification," in *CSF*, 2008.

[27] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Expressive Declassification Policies and Modular Static Enforcement," in *IEEE Symposium on Security and Privacy*, 2008.

[28] P. Li and S. Zdancewic, "Downgrading policies and relaxed noninterference," in *POPL*, 2005.

[29] A. Sabelfeld and A. C. Myers, "A model for delimited information release," in *International Symposium on Software Security*, 2003.

[30] A. Askarov and A. Sabelfeld, "Localized Delimited Release: Combining the What and Where Dimensions of Information Release," in *PLAS*, 2007.

[31] J. Magazinius, A. Askarov, and A. Sabelfeld, "Decentralized Delimited Release," in *APLAS*, 2011.

[32] C. Fournet and J. Planul, "Compiling Information-Flow Security to Minimal Trusted Computing Bases," in *ESOP*, 2011.

[33] N. Swamy and M. Hicks, "Verified enforcement of stateful information release policies," in *PLAS*, 2008.

[34] G. Le Guernic, "Confidentiality Enforcement Using Dynamic Information Flow Analyses," Ph.D. dissertation, Kansas State University, 2007.

[35] A. Bohannon and B. C. Pierce, "Featherweight firefox: Formalizing the core of a web browser," in *USENIX Conference on Web Application Development*, 2010.

[36] G. Barthe, J. M. Crespo, D. Devriese, F. Piessens, and E. Rivas, "Secure multi-execution through static program transformation," in *FMOODS/FORTE*, 2012.

[37] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen, "Webjail: Least-privilege integration of third-party components in web mashups," in *ACSAC*, 2011.

[38] D. Crockford, "Adsafe," http://www.adsafe.org/, 2009.

[39] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Caja: Safe active content in sanitized javascript," http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf.

[40] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, "JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications," in *ACSAC*, 2012.

[41] S. Maffeis, J. C. Mitchell, and A. Taly, "Object Capabilities and Isolation of Untrusted Web Applications," in *IEEE Symposium on Security and Privacy*, 2010.

[42] W. Xu, S. Bhatkar, and R. Sekar, "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," in *USENIX Security*, 2006.

[43] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna, "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis," in *NDSS*, 2007.