Tainted Secure Multi-Execution to Restrict Attacker Influence

McKenna McCall
Carnegie Mellon University
Pittsburgh, USA
mckennak@cmu.edu

Abhishek Bichhawat
Indian Institute of Technology Gandhinagar
Gandhinagar, India
abhishek.b@iitgn.ac.in

Limin Jia
Carnegie Mellon University
Pittsburgh, USA
liminjia@cmu.edu

ABSTRACT

Attackers can steal sensitive user information from web pages via third-party scripts. Prior work shows that secure multi-execution (SME) with declassification is useful for mitigating such attacks, but that attackers can leverage dynamic web features to declassify more than intended. The proposed solution of disallowing events from dynamic web elements to be declassified is too restrictive to be practical; websites that declassify events from dynamic elements cannot function correctly.

In this paper, we present SME^T, a new information flow monitor based on SME which uses taint tracking within each execution to remember what has been influenced by an attacker. The resulting monitor is more permissive than what was proposed by prior work and satisfies both knowledge- and influence-based definitions of security for confidentiality and integrity policies (respectively). We also show that robust declassification follows from our influence-based security condition, for free. Finally, we examine the performance impact of monitoring attacker influence with SME by implementing SME^{T}on top of Featherweight Firefox.

CCS CONCEPTS

• **Security and privacy** → **Formal security models**; **Web application security**.

KEYWORDS

information flow; knowledge-based noninterference; robust declassification; secure multi-execution; taint tracking

ACM Reference Format:
McKenna McCall, Abhishek Bichhawat, and Limin Jia. 2023. Tainted Secure Multi-Execution to Restrict Attacker Influence. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3576915.3623110

1 INTRODUCTION

Online services, for example, banking, social media, and shopping, typically require access to a user's personal information, such as their phone number, location, or credit card details. Web attackers have been known to steal sensitive user data [25], sometimes via third-party scripts, which have been observed indiscriminately collecting data, including personal information, from web forms [39].

Information flow control (IFC) monitors are a promising way to prevent sensitive information from leaking to attackers [21, 31, 35]. They have been used to secure applications in many domains [20, 22, 26, 27, 40, 44]. The canonical IFC security property is *noninterference*. The simplest form of noninterference says that public outputs (least privileged) should never be influenced by secret inputs (requiring the most privilege). However, in many real-world applications, this definition is too restrictive to be practical. Supporting principled *declassification*, which allows selected sensitive information to be leaked while maintaining an otherwise provably secure system, is important for many useful web services like website analytics. For instance, if a company wants to know which products are being clicked on, they may want to track some of their customers' interactions on their site. Declassification can ensure that these third-party analytics will have access to the information they need (e.g., which products are clicked on), without releasing *other* sensitive information.

Prior work that allowed declassification by web scripts either did not prove formal properties about declassification [11, 12], or used a simplified model missing some dynamic JavaScript features that could be leveraged by an attacker to leak information [41]. Later work explored the threat posed by declassifying events associated with dynamically added page elements and developed a technique using secure multi-execution (SME) to prevent these leaks [30] (detailed discussion in Section 2.2). However, the proposed solution disallows all events from dynamically generated web elements from being declassified. While this technique is provably secure, it risks altering the behavior of secure programs and could prevent declassification in the benign example described above.

This paper aims to develop an IFC monitor that allows flexible declassification without sacrificing security. Since SME enjoys strong security guarantees and do not need to abort the program (as opposed to NSU [7]), which is desirable for web applications, we build on prior work on securing dynamic secrets with SME [30] to develop a more fine-grained technique for protecting dynamic features from leaking secrets due to declassification.

One key insight is that leaks caused by attackers' interactions with declassification can be stopped if the monitor tracks attacker influence on the page, only preventing declassification when it involves code added by the attacker. We provide more detailed examples in Section 3.

We design SME^{T}by extending prior work [30] with techniques based on integrity labels to enforce robust declassification [19, 43]. Specifically, SME^{T}uses taint tracking within *each* execution to remember the trustworthiness of page elements and their event handlers via integrity labels. These integrity labels indicate attacker influence and decide whether declassification is allowed.

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3623110

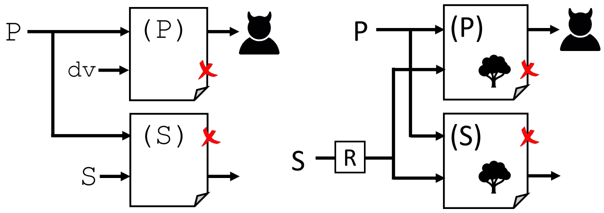


Figure 1: Standard SME (left) and SME with declassification and multiple DOMs (right)

We define our security conditions based on knowledge-based noninterference [1, 4, 6, 9, 10]. We present a novel knowledge-based security condition where robust declassification follows from influence-based security for free.

The same techniques may be applied to knowledge-based security conditions to prove transparent endorsement [18] (the integrity dual of robust declassification), but in this paper, we focus on robust declassification for ease of understanding. We prove that the design of SME^T is secure. We implement our model on top of Featherweight Firefox as a sanity check on our semantics and to understand the impact of our more complex security lattice on performance.

This paper makes the following technical contributions.

- A novel IFC monitor design that combines SME and taint tracking for more permissive declassification
- Novel security conditions that capture confidentiality, integrity, and robust declassification.
- Proofs that SME^T is secure.
- A prototype implementation in Featherweight Firefox.

Detailed definitions and proofs be found in our companion technical report [29]. The implementation of our model on top of Featherweight Firefox is available at https://github.com/CompIFC/tainted-sme.git.

2 BACKGROUND AND RELATED WORK

2.1 Reactive systems and IFC monitors

Reactive systems have been widely used to model web applications. A reactive program is a set of event handlers which execute when they are triggered by events [16]. We consider a single-threaded model where event handlers execute one at a time. While an event handler is running, the system is in Producer state. After the event handlers finish execution, the system waits in Consumer state for more events to process.

Secure multi-execution (SME) enforces IFC policies in reactive programs by executing event handlers multiple times—once at each security level. Each execution only receives the inputs it has privilege to see, and only outputs to channels matching the security level of the execution [23, 24]. Consider a two-point security lattice with labels P (Public) and S (Secret), and the ordering $P \sqsubseteq S$ (meaning information can flow from P to S but not vice versa). As shown in Figure 1, SME would run event handlers twice, where both copies of the execution see P-labeled data. The S copy of the execution can see all of the data, but can only output to privileged (Secret) channels. In the P copy of the execution, Secrets are replaced with a default value (dv) and can only output to Public channels.

Faceted execution [8, 14] is a similar multi-execution technique. Rather than running all of the code multiple times, this approach creates "facets" of values for every level in the lattice, only when they depend on a secret. The code runs once until the control flow depends on a faceted value and the execution splits to evaluate each facet. Later work combines SME and faceted execution [37] (and an optimization [2]) and proposes "generalized" multiple facets [33] to balance the security and performance tradeoffs of the two multi-execution techniques (in the first two cases), consider a more general security lattice (in the last case), and each achieves stronger (termination-sensitive) security guarantees than offered by traditional faceted execution. SME^T also uses a general security lattice. The techniques SME^T uses may be relevant to faceted execution, but we focus on SME because the semantics are simpler.

Taint tracking approaches enforce IFC policies by attaching labels to the data in the system, which indicates their secrecy and trustworthiness. The label on the data determines if an output is permitted (if the channel trusts the data and has enough privilege to receive it) or not. Taint tracking is susceptible to implicit leaks when branching on a secret. One solution is to abort the execution when updating public data in secret contexts (called *no sensitive upgrade* [7]), or simply permit the leaks and block only explicit leaks that output secret information to public channels (satisfying a weaker security condition called *explicit secrecy* [28, 38]). SME^T is a new monitor which composes SME with taint tracking so that we can keep track of the trustworthiness of the event handlers within each execution. These labels are determined when the event handler is initially registered and remain fixed throughout execution, so we don't need to worry about sensitive upgrades, nor do we have to resort to an explicit secrecy security condition.

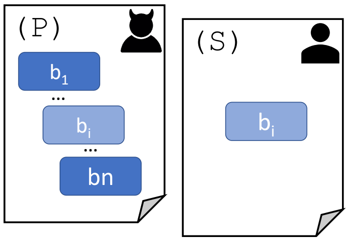
2.2 Declassification with dynamic features

The monitors described above enforce strict *noninterference*, where secret inputs are never allowed to influence public outputs. But this is often too restrictive for common use cases such as analytics where an online shop wants to learn which products users are clicking on most, or user authentication where a bank wants to know a user's location. Declassification offers a principled way to release some information. Vanhoef et al. developed an approach to *stateful* declassification in SME [41], where declassification policies are flexible enough to release events, as well as aggregated/approximated data. For instance, "the user's approximate location may be released after they give permission" and "the average location of every 100 mouse clicks may be released" are both stateful policies.

However, our prior work [30] showed that dynamic features can be used by an attacker to leak more than is allowed by these declassification policies via the following attack, illustrated in Figure 2. Consider the 2-point security lattice from before and a web page with the policy: all user events are secret, click events and the occurrence of keypress events may be declassified (however, *which* key was pressed should remain secret). The P copy of the page is visible to the attacker, and receives only the public (or declassified) events, while the S copy is visible to the user and receives all of the events. Suppose the attacker registers the event handler shown in Figure 2a which runs whenever a key is pressed and adds a different button to the page, depending on what is typed (stored in *secret*).

$$\begin{array}{lll}
\multicolumn{3}{l}{\text{onKeypress}(\textit{secret}) = \text{case } \textit{secret} :} \\
\quad \mid 1 & \Longrightarrow & \text{new}(b_1); \text{addEH}(b_1, \text{on}\textit{Click}\{\text{output}_P(1)\}); \\
\quad \dots & & \\
\quad \mid \text{n} & \Longrightarrow & \text{new}(b_n); \text{addEH}(b_n, \text{on}\textit{Click}\{\text{output}_P(n)\}); \\
\quad \mid \text{dv} & \Longrightarrow & \text{new}(b_1); \text{addEH}(b_1, \text{on}\textit{Click}\{\text{output}_P(1)\}); \\
 & & \dots \\
 & & \text{new}(b_n); \text{addEH}(b_n, \text{on}\textit{Click}\{\text{output}_P(n)\});
\end{array}$$

(a) Event handler added by the attacker.



(b) Resulting attacker view (*P*) and user view (*S*) of the page.

Figure 2: Example of dynamic features causing leaks. The dv case guarantees that the attacker copy will have a matching button (colored light blue) to capture the declassified event and leak the secret.

If the user types i, this event handler would add button b_i to the S copy of the page based on the actual value of the secret. The P copy of the page receives the event with a default value dv to hide what was typed, so the event handler adds *all* possible buttons to the page (shown in Figure 2b). When the user clicks on b_i (S copy of the page), the click is declassified to the P copy, which is guaranteed to have a matching button to capture the event. The on*Click* event handler executes the statement $\text{output}_P(i)$. Since outputs to P channels are allowed in the P execution, this leaks what the user typed to the attacker.

To prevent this leak, our prior work [30] proposes an additional label S_Δ for dynamically-generated elements, and *restrict* declassification to only apply to elements labeled S, i.e., events dispatched on elements labeled S_Δ are never declassified to P. Accordingly, in the previous example, the button b_i is labeled S_Δ; hence, the mouse click on b_i is not declassified to P, which prevents the attacker from learning which key was pressed. While this prevents unintentional leaks, it can be too restrictive to be practical, which is one of the motivations for this work.

3 MOTIVATING EXAMPLES

Recall the scenario from Section 1 where an online shop wants to know which of their products are receiving the most attention. They use JavaScript to dynamically display products on their site depending on what the user has searched. To measure product popularity, they use a third-party analytics library to track where users are clicking on their site. Because they do not want the third-party to have access to *all* of the user's private information, they treat the script as Public. To give the library access to the relevant click information, the shop employs a policy where the coordinates of each click are Secret, but which product is clicked may be declassified. With the solution described above, everything dynamically loaded to the page (even by code not controlled by the attacker) will be labeled S_Δ and excluded from declassification, and thus, the online shop won't be able to perform their analytics.

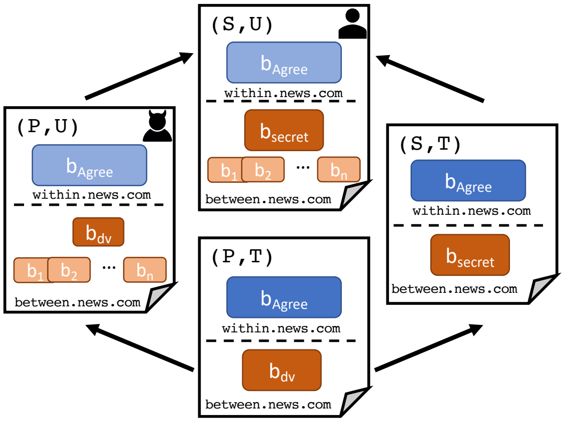


Figure 3: Information is allowed to flow in the direction of the arrows. The attacker can influence Untrusted executions to add page elements or event handlers to try to manipulate declassification directly within an execution (blue case) or indirectly between executions (orange case).

The reason the earlier example (Figure 2b) leaked more than intended is that the *attacker* leveraged the declassification policy to leak information by adding buttons to the page. Meanwhile, the products added to the web page described above are added by the shop itself, who should be trusted to trigger declassification. The underlying problem is not the dynamic page elements, but their *source*. Instead of disallowing any dynamic features to influence declassification, an intuitive fix would simply restrict the attacker's influence. This involves protecting the *integrity* of the data, which is dual to the *confidentiality* policies we have discussed so far.

3.1 Examples with integrity labels

Consider a 4-point security lattice with 2 confidentiality labels (Public and Secret) and 2 integrity labels (Trusted and Untrusted). Information is allowed to flow from Public to Secret and Trusted to Untrusted. The complete security lattice is a diamond with (P, T) at the bottom, (S, U) at the top, and the other labels (P, U) and (S, T) in between. SME can enforce information flow policies drawn from this lattice by running one execution for each of these 4 security levels as shown in Figure 3. In this model, the attacker and other Untrusted parties, like *ad.com*, are only able to influence the code running on the Untrusted executions, while Trusted parties (like *news.com*) may influence code running in any execution. In our examples, the (P, U) execution communicates with the attacker via *ad.com* and the user is shown the (S, U) version of the webpage.

In the following examples, we show that attackers can influence declassification irrespective of whether the user interacts with attacker code directly (similar to the leak from [30]) or indirectly (if a declassification triggers attacker code in *another* execution).

Example 3.1. **Leaks within an execution.** Suppose a user visits a webpage (*within.news.com*) which explains that it will share their account preferences with advertisers (*ad.com*), but only if they click the "Agree" button (identified in the code as b_{Agree}) to consent. When the page loads, *ad.com* adds a large b_{Agree} button at the top of the page with the text "Click me!", as in Figure 3, where the buttons coming from *ad.com* are light blue and the ones from *within.news.com* are dark blue. A user may click the button, not realizing it will declassify their preferences. We call this a leak *within* an execution because the user is interacting directly with attacker-controlled code. This is similar to the attacks from prior work [30], where the user interacts directly with the page element.

Example 3.2. **Leaks between executions.** Consider another webpage (*between.news.com*) which has the policy that keypress events are Secret, but clicks may be declassified from Secret to Public. *news.com* installs an event handler which adds a different button to the page, depending on which key the user presses (similar to onKeypress in Figure 2, without the dv case). Meanwhile, *ad.com* adds all possible buttons to the page and registers an event handler which is triggered by a click to send them a message, telling them which button was clicked (similar to the dv case from the onKeypress event handler in Figure 2). The resulting page is shown in Figure 3, where the dark orange buttons were added by *news.com* and the light orange buttons were added by *ad.com.* Note that because *news.com* is Trusted, the dark orange buttons are added to all copies of the webpage, including the Untrusted ones.

Like the leak from prior work [30], if the user clicks the b_{secret} button on the (S,U) page, the event will be declassified to the (P,U) execution, which is guaranteed to have a matching button to capture the event and leak the keypress to the attacker. We call this a leak *between* executions because the user is interacting with code added by the host page which triggers attacker-controlled code in *another* execution. This example highlights that it is not enough to only look at the page the user is interacting with, we also need to consider the executions capturing the declassified events.

To prevent the attacker from influencing declassification, one approach would be to extend the solution from prior work [30] to apply to events *originating from* dynamic elements in Untrusted executions (which might include attacker-controlled code), as well as events being *released to* dynamic elements in the Untrusted executions. But as we described above, this would also prevent innocent declassifications, like in the online shop. Likewise, it wouldn't be enough to prevent the user from interacting directly with attacker-controlled code by showing them the (S,T) copy of the page instead of the (S,U) copy, because this would still be susceptible to the leaks between executions.

3.2 Our approach: Tracking integrity in SME

To prevent these leaks without sacrificing functionality, we develop SME^T (Section 5.2), which is SME with taint tracking to reflect the trustworthiness of the source of the code adding new page elements and event handlers. We check that the user trusts the code they're interacting with directly to decide if a declassification should be triggered (preventing leaks *within* executions), as well as the code in other executions to decide whether they should receive the event (preventing leaks *between* executions).

4 SME WITH DYNAMIC FEATURES

We first describe the syntax and semantics of SME for reactive systems with dynamic features (declassification will be added in the next section). Our semantics are flexible enough to work with any finite security lattice of confidentiality and integrity labels. Following prior work [30], we organize our SME semantics into three levels: the top-most level is responsible for processing inputs and outputs, looking up event handlers, and switching between executions. The mid-level manages the execution for a particular execution. The lowest level runs the current event handler.

4.1 I/O Processing and EH Lookup

The syntax for these rules is summarized in Figure 4. The security lattice includes confidentiality labels, $l_c \in \mathcal{L}_c$, which specifies the privilege needed to access data, and integrity labels, $l_i \in \mathcal{L}_i$, which specifies how trusted a component is. Information may flow from (l_c, l_i) to (l_c', l_i') if l_c' has privilege to see data from l_c ($l_c \sqsubseteq l_c'$) and l_i' trusts data from l_i ($l_i \sqsubseteq l_i'$). Our earlier example used a security lattice with $\mathcal{L}_c = \{P, S\}$ and $\mathcal{L}_i = \{T, U\}$ for $P \sqsubseteq S$ and $T \sqsubseteq U$, but our rules are general enough to accommodate any (finite) lattice.

Events are associated with elements given by unique identifiers id. Event handlers of the form on$Ev(x)\{c\}$ run command c with argument x when the system receives event Ev (such as a click). The security label (l_c, l_i) of an event is determined by the security policy $\mathcal{P}$. An execution trace T is zero or more steps of the top-level system. An SME configuration K is a snapshot of the system including the SME state Σ and the configuration stack ks.

Σ keeps track of the persistent state for each execution; each security level $pc = (l_c, l_i)$ has its own store $\sigma^{EH}_{(l_c,l_i)}$ which is the event handler storage (i.e., the DOM) for each execution. The event handler storage maps identifiers id to attributes v and event handler maps M, which maps events Ev to their respective event handlers $eh_1, ..., eh_n$. This model allows each execution to have its own copy of the DOM, whose contents may vary in privilege and trust. Each execution runs its event handlers separately, beginning at the top of the configuration stack ks. Each element of the configuration stack determines what event handler to run, given by configuration κ, and in which execution, given by the security level pc.

As the system runs, it may react to/emit various actions, α. In the reactive setting, the system waits until it receives an input which is an event triggering (zero or more) event handlers which may produce some outputs. In our case, inputs are user interactions $id.Ev(v)$ which are events Ev associated with an element id (possibly) carrying some argument (e.g., which key is pressed for a keyPress event or the location of a click). Outputs are given by values sent along a channel ch. The other actions are silent $\bullet$.

The semantics for the top-most level are shown in Figure 5. Rule In receives an event Ev for page element id with parameter v from the principal with privilege and trustworthiness given by pc. The security policy tells us the label on the event is pc'. We run the event handlers associated with the event in each execution with enough privilege to see the event and who trust the event, i.e., at all executions at or above $pc \sqcup pc'$ in the security lattice. The lookup semantics ($\Sigma, E \rightsquigarrow$ ks) looks up the event handlers in Σ and constructs a configuration for each execution in E, resulting in ks.

$$
\begin{array}{llll}
\textit{Security lattice:} & \mathcal{L} & ::= & \mathcal{L}_c \times \mathcal{L}_i \\
\textit{Event:} & Ev & ::= & \mathsf{click} \mid \mathsf{keyPress} \mid \ldots \\
\textit{Event handler:} & eh & ::= & \mathsf{on}Ev(x)\{c\} \\
\textit{Security policy:} & \mathcal{P} & & \\
\text{Individual event handler} & & & \\
\textit{Expression:} & e & ::= & x \mid v \mid id \mid \mathsf{uop}\ e \mid e_1\ \mathsf{bop}\ e_2 \\
\textit{Command:} & c & ::= & \mathsf{skip} \mid c_1; c_2 \mid x := e \mid id := e \\
 & & \mid & \mathsf{while}\ e\ \mathsf{do}\ c \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \\
 & & \mid & \mathsf{output}\ ch\ e \mid \mathsf{new}(id, e) \\
 & & \mid & \mathsf{addEh}(id, eh) \mid \mathsf{trigger}\ id.Ev(e)
\end{array}
\qquad
\begin{array}{llll}
\textit{Single configuration:} & \kappa & ::= & \sigma^v, c, s, E \\
\textit{Execution state:} & s & ::= & \mathsf{P} \mid \mathsf{C} \\
\textit{SME traces:} & T & ::= & K \mid \mathcal{P} \vdash T \overset{\alpha_I}{\Longrightarrow} K \\
\textit{Event queue:} & E & ::= & \cdot \mid E, (id.Ev(v), pc) \\
\textit{SME configuration:} & K & ::= & \Sigma; \mathsf{ks} \\
\textit{SME state:} & \Sigma & ::= & \cdot \mid \Sigma, pc \mapsto \sigma^{EH}_{pc} \\
\textit{EH state:} & \sigma^{EH} & ::= & \cdot \mid \sigma^{EH}, id \mapsto (v, M) \\
\textit{EH map:} & M & ::= & \cdot \mid M, Ev \mapsto \{eh_1, \ldots, eh_n\} \\
\textit{Configuration stack:} & \mathsf{ks} & ::= & \cdot \mid (\kappa, pc) :: \mathsf{ks} \\
\textit{Actions:} & \alpha & ::= & id.Ev(v) \mid ch(v) \mid \bullet
\end{array}
$$

Figure 4: SME Syntax

$$\boxed{\mathcal{P} \vdash K \overset{(\alpha,pc)}{\Longrightarrow} K'}$$

$$\frac{\mathcal{P}(id.Ev(v)) = pc' \qquad E = ((id.Ev(v), pc'') \mid pc'' \in \mathcal{L} \text{ s.t. } pc \sqcup pc' \sqsubseteq pc'') \qquad \Sigma, E \rightsquigarrow \mathsf{ks}}{\mathcal{P} \vdash \Sigma; \cdot \overset{(id.Ev(v),pc)}{\Longrightarrow} \Sigma; \mathsf{ks}}\ \textsc{In}$$

$$\frac{\mathsf{producer}(\kappa) \qquad \Sigma, \kappa \xrightarrow{ch(v)}_{pc} \Sigma', \mathsf{ks}' \qquad \alpha = ch(v) \text{ if } \mathcal{P}(ch) = pc \qquad \alpha = \bullet \text{ otherwise}}{\mathcal{P} \vdash \Sigma; (\kappa, pc) :: \mathsf{ks} \overset{(\alpha,pc)}{\Longrightarrow} \Sigma'; \mathsf{ks}' :: \mathsf{ks}}\ \textsc{Out}$$

$$\frac{\mathsf{producer}(\kappa) \qquad \Sigma, \kappa \xrightarrow{\bullet}_{pc} \Sigma', \mathsf{ks}'}{\mathcal{P} \vdash \Sigma; (\kappa, pc) :: \mathsf{ks} \overset{(\bullet,pc)}{\Longrightarrow} \Sigma'; \mathsf{ks}' :: \mathsf{ks}}\ \textsc{Out-Silent}$$

$$\frac{\mathsf{consumer}(\kappa)}{\mathcal{P} \vdash \Sigma; (\kappa, pc) :: \mathsf{ks} \overset{(\bullet,pc)}{\Longrightarrow} \Sigma; \mathsf{ks}}\ \textsc{Out-Next}$$

$$\boxed{\Sigma, E \rightsquigarrow \mathsf{ks}}$$

$$\frac{\Sigma(pc)(id.Ev(v)) = c \qquad \kappa = \cdot, c, P, \cdot \qquad \Sigma, E \rightsquigarrow \mathsf{ks}}{\Sigma, (id.Ev(v), pc) :: E \rightsquigarrow (\kappa, pc) :: \mathsf{ks}}\ \textsc{lookup}$$

$$\frac{}{\Sigma, \cdot \rightsquigarrow \cdot}\ \textsc{lookup-empty}$$

Figure 5: Top-level SME rules for processing inputs and outputs, and looking up event handlers

The output rules run event handlers one at a time. When an event handler is running, the configuration at the top of the stack is in producer state, producer(κ). Rule Out handles outputs produced by the event handler. An execution performs outputs to channels only if the label on the channel matches the execution context, i.e., $\mathcal{P}(ch(v)) = pc$. Otherwise, the output is suppressed. Rule Out-Silent handles steps which don’t produce outputs. When the event handler finishes running, the configuration at the top of the stack is in consumer state, consumer(κ), and rule Out-Next pops the configuration off the stack to run the next event handler. The execution state is managed by the mid-level semantics, described next.

Example: Example 3.1 of leaks *within* an execution uses the security policy that click events are considered secret and trusted, $\mathcal{P}(\_.\mathsf{Click}(\_)) = (S, T)$[1] and page load events are public and trusted, $\mathcal{P}(\_.\mathsf{load}(\_)) = (P, T)$. The user interacts with the (S, U) copy of the page and the attacker who serves ads from *ad.com* is listening on (P, U) channels.

Initially, before any events have been triggered, we assume that the SME state is well-formed, meaning the source of the code (l_i) loaded to each execution (l'_i) is trusted ($l_i \sqsubseteq l'_i$). The attacker-controlled code from *ad.com* only appears in Untrusted executions, while the code from Trusted *news.com* will appear in all of the executions. For our example, we also assume that *ad.com* registers an onLoad^U function to add the “Click me!” button (from Figure 3), and *news.com* registers onLoad^T to add the “Agree” button. We use the superscript U and T to distinguish the event handler added by *ad.com* from the one added by *news.com.*

Then, the initial SME configuration is $K_0 = \Sigma_0; \mathsf{ks}_0$ where ks_0 runs *body*.load for each execution (ks_0 will be described in more detail in the next section) in the following SME state:

$$
\begin{array}{rll}
\Sigma_0 = & (S, U) \mapsto & body \mapsto (\_, \mathsf{load} \mapsto \{\mathsf{onLoad}^U, \mathsf{onLoad}^T\}), \\
 & (P, U) \mapsto & body \mapsto (\_, \mathsf{load} \mapsto \{\mathsf{onLoad}^U, \mathsf{onLoad}^T\}), \\
 & (S, T) \mapsto & body \mapsto (\_, \mathsf{load} \mapsto \{\mathsf{onLoad}^T\}), \\
 & (P, T) \mapsto & body \mapsto (\_, \mathsf{load} \mapsto \{\mathsf{onLoad}^T\}),
\end{array}
$$

Next, the (S, U) execution runs the onLoad^U event handler. Rule Out-Silent applies and makes a step: $K_0 \overset{(\bullet,(S,U))}{\Longrightarrow} K_1$. The new configuration K_1 has a new button in the (S, U) copy of the store and the other copies remain unchanged:

$$
\begin{array}{rll}
\Sigma_1 = & (S, U) \mapsto & body \mapsto (\ldots), b_{\mathsf{Agree}} \mapsto (\textit{“Click me!”}, \cdot) \\
 & \text{the rest are the same as } \Sigma_0 &
\end{array}
$$

The same process will repeat to add the “Click me!” button to the (P, U) store and the “Agree” button to the other executions. Now that the event handlers have finished running, rule Out-Next pops the event handler from ks and the system waits for user input.

Suppose the attacker also installed an event handler in the (S, U) and (P, U) executions which directly sends them the user’s account preferences. Since they are listening on a (P, U) channel, the rule out would suppress the output from the (S, U) execution which knows the real preferences (since $\mathcal{P}(ch) \neq (S, U)$). The same rule allows the output from the (P, U) execution, which would instead output a default value dv, with no access to the real preferences.

[1] Not to be confused with the isTrusted property distinguishing events which come from a user from events which were generated by an event handler (see [42]).

$$\boxed{\Sigma, \sigma, c \xrightarrow{\alpha}_{pc} \Sigma', \sigma', c', E}$$

$$\frac{[\![e]\!]^{pc}_{\sigma,\Sigma} = v}{\Sigma, \sigma, \mathsf{output}\ ch\ e \xrightarrow{ch(v)}_{pc} \Sigma, \sigma, \mathsf{skip}, \cdot}\ \textsc{output}$$

$$\frac{[\![e]\!]^{pc}_{\sigma,\Sigma} = v \qquad E = (id.Ev(v), pc)}{\Sigma, \sigma, \mathsf{trigger}\ id.Ev(e) \xrightarrow{\bullet}_{pc} \Sigma, \sigma, \mathsf{skip}, E}\ \textsc{trigger}$$

$$\frac{\begin{array}{c}[\![e]\!]^{pc}_{\sigma,\Sigma} = v \qquad \Sigma(pc) = \sigma^{EH} \\ id \notin \sigma^{EH} \qquad \Sigma' = \Sigma[pc \mapsto \sigma^{EH}[id \mapsto (v, \cdot)]]\end{array}}{\Sigma, \sigma, \mathsf{new}(id, e) \xrightarrow{\bullet}_{pc} \Sigma', \sigma, \mathsf{skip}, \cdot}\ \textsc{new}$$

$$\frac{\begin{array}{c}\Sigma(pc) = \sigma^{EH} \qquad \sigma^{EH}(id) = (v, M) \\ \sigma^{EH'} = \sigma^{EH}[id \mapsto (v, M[Ev \mapsto M(Ev) \cup eh])] \\ \Sigma' = \Sigma[pc \mapsto \sigma^{EH'}]\end{array}}{\Sigma, \sigma, \mathsf{addEh}(id, Ev, eh) \xrightarrow{\bullet}_{pc} \Sigma', \sigma, \mathsf{skip}, \cdot}\ \textsc{add-eh}$$

Figure 6: (Selected) rules for running event handlers

4.2 Execution State and EH Queue

A single configuration κ is a snapshot of one execution, including the local variables σ^v (which are only accessible to the event handler currently running), the current command c being executed, the execution state s of the event handler, and the event queue E. The execution state is either P for producer (meaning an event handler is running) or C for consumer (meaning the event handlers have finished and the execution is ready to process a new event). Here, the event queue, E, is a list of the events triggered by other event handlers. The events will run in the same execution, so the pc on each event in the queue will match the current execution context. Due to space constraints, the semantics for managing the event handler queue and execution state may be found in the companion technical report [29].

4.3 Individual Event Handlers

Expressions in the body of an event handler include variables, values (integers and booleans), page element identifiers, id, unary, and binary operators. Commands are mostly standard and include outputs to channels and dynamic behaviors for adding new page elements ($\mathsf{new}(id, e)$), registering new event handlers ($\mathsf{addEh}(id, eh)$), and triggering event handlers ($\mathsf{trigger}\ id.Ev(e)$).

Selected event handler operational semantic rules are in Figure 6. Expression evaluation is denoted $[\![e]\!]^{pc}_{\sigma,\Sigma}$ where pc tells us which copy of the shared storage to access in Σ and σ is the store local to the current event handler. Candidate outputs are produced by rule OUTPUT. The other rules are for handling dynamic elements, including triggering event handlers (rule TRIGGER), generating new page elements (rule NEW), and registering a new event handler (rule ADD-EH). In each of these rules, we interact with the copy of the global storage that matches the current execution context. Event handlers run in the same context they were triggered in, denoted by pc. New page elements must have a unique identifier, $id \notin \sigma^{EH}$, and are initialized with the given attribute and no event handlers, $M = \cdot$. When registering a new event handler, the existing event handlers associated with the event are looked up in the event handler map, $M(Ev)$. The event handler map is updated to include the original event handlers plus the new one, $M[Ev \mapsto M(Ev) \cup eh]$.

5 DECLASSIFICATION AND SME^T

We extend the syntax and semantics from Section 4 to include declassification. Due to space constraints, we describe the changes to the rules in this section and present the full rules in our companion technical report [29].

$$\boxed{\mathcal{P}, \mathcal{D} \vdash K \overset{(\alpha,pc)}{\Longrightarrow} K'}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.Ev(v)) = pc' \\ E = ((id.Ev(v), pc'') \mid pc'' \in \mathcal{L} \text{ s.t. } pc \sqcup pc' \sqsubseteq pc'') \\ (\mathcal{R}', E_d) = \mathsf{declassify}(\mathcal{D}, \mathcal{R}, \Sigma, (id.Ev(v), pc), pc') \\ \Sigma, E :: E_d \rightsquigarrow \mathsf{ks}\end{array}}{\mathcal{P}, \mathcal{D} \vdash \mathcal{R}; \Sigma; \cdot \overset{(id.Ev(v),pc)}{\Longrightarrow} \mathcal{R}'; \Sigma; \mathsf{ks}}\ \textsc{In}$$

$$\boxed{\mathsf{declassify}(\mathcal{D}, \mathcal{R}, \Sigma, (id.Ev(v), pc)) = (\mathcal{R}', E)}$$

$$\frac{\begin{array}{c}\mathcal{R} = (\rho, d) \qquad \mathcal{D}((id.Ev(v), pc), pc', \rho) = (\rho', v_d, E_d) \\ d' = \mathsf{update}(d, v_d)\end{array}}{\mathsf{declassify}(\mathcal{D}, \mathcal{R}, \Sigma, (id.Ev(v), pc), pc') = ((\rho', d'), E_d)}\ \textsc{declassify}$$

Figure 7: Updated input rule for declassification. Key changes are shown in red text.

5.1 Stateful Declassification

We use *stateful* declassification [30, 41]. A stateful policy is one that may involve the system state when deciding whether to declassify. Here, we describe the syntax for declassification, shown below.

$$\begin{array}{llcl}
\textit{Declass. policy:} & \mathcal{D} & & \\
\textit{Declass. module:} & \mathcal{R} & ::= & (\rho, d) \\
\textit{Declass. state:} & \rho & ::= & \cdot \mid \rho, (id_1.Ev_1, n_1) \\
\textit{Declass. channel:} & d & ::= & (\iota_1, v_1), \cdots, (\iota_n, v_n)
\end{array}$$

The declassification policy is given by $\mathcal{D}$. Given an event and the current state, as well as information from the security policy, $\mathcal{P}$, $\mathcal{D}$ updates the current state and decides whether the event should be declassified. The declassification module $\mathcal{R}$ keeps track of the current state for making decisions about declassification as well as channels for event handlers to access released values. A declassification state ρ keeps track of relevant state conditions, such as the number of times an event has been seen, and the declassification channel d associates locations ι (such as a line number in the code) with the released value accessible by that location.

Rules for the I/O semantics is updated to include $\mathcal{D}$ and $\mathcal{R}$:

$$\mathcal{P}, \mathcal{D} \vdash \mathcal{R}; \Sigma; \mathsf{ks} \overset{\alpha_I}{\Longrightarrow} \mathcal{R}'; \Sigma'; \mathsf{ks}'$$

A declassification function (declassify), shown in Figure 7 is added to the input rule. It uses the declassification policy $\mathcal{D}$ to determine whether the new event should be released to run event handlers in additional execution contexts E_d, whether the system state ρ should be updated, and what values should be updated on the declassification channel d (if any).

Example: Recall Example 3.1 of leaks *within* an execution, where the security policy says that clicks are (S, T), and the declassification policy says that the user's preferences may be declassified from S to P when b_{Agree} is clicked.

When the user clicks b_{Agree} in the (S, U) execution, In will share the event with all the executions with enough privilege and trust the user (just (S, U)), but we also use declassify to determine whether the event should be declassified to additional executions:

$$\mathcal{D}((b_{\text{Agree}}.\text{Click}(\ ), (S, U)), (S, T), (b_{\text{Agree}}.\text{Click}, n)) = \\ ((b_{\text{Agree}}.\text{Click}, n + 1), \mathit{pref}, \cdot)$$

This indicates that the state ρ has been updated to reflect that one more click has been seen (n becomes $n + 1$), the user's preferences should be released on the declassification channel (*pref*), and the click event should not be released to any additional executions.

For Example 3.2 of leaks *between* executions, the security policy says that button clicks and keypresses are both (S, T), but now, the declassification policy says that button clicks may be released from S to P. When the user clicks b_{secret}, In runs the event as-is in the (S, U) execution and declassifies the event as follows:

$$\mathcal{D}((b_{\text{secret}}.\text{click}(\ ), (S, T)), (S, T), (b_{\text{secret}}.\text{click}, m)) = \\ ((b_{\text{secret}}.\text{click}, m + 1), \text{none}, (b_{\text{secret}}.\text{click}(\ ), (P, U)))$$

Here, ρ is updated to reflect the click, nothing is updated on the declassification channel (none), and the click event is released to the P executions who trust the event. That is, the event is released to all executions with label l_i s.t. l_i trusts the event l_i' (determined by the security policy) and the source of the event l_i'' (formally, $l_i' \sqcup l_i'' \sqsubset l_i$). Here, this is just $(b_{\text{secret}}.\text{Click}(\ ), (P, U))$. The result is that the onClick event handler will run in both the (S, U) and (P, U) executions. The rule Out will suppress the output from the (S, U) execution, but permit the output from the (P, U) execution, which is guaranteed to have a matching button to capture the event.

5.2 Robust Declassification in SME^T

In the presence of an active attacker who may control some of the code, we need to ensure that they do not control what/whether data is declassified [43]. For the declassifications to be robust against attacker influence, we need to ensure that the source of the event l_i trusts the code l_i' on the *same* execution they're interacting with ($l_i' \sqsubseteq l_i$). Additionally, we need to check that the source of the event trusts the code which added the page element in the *other* execution receiving the declassified event.

SME^T composes taint tracking with the SME semantics presented in the previous section to also keep track of the source of the page elements in each execution. First, we modify the event handler storage σ^{EH} so that page elements and event handlers have labels indicating the trustworthiness of their source:

$$\begin{array}{lll} \textit{EH state: } \sigma^{EH} & ::= & \cdot \mid \sigma^{EH}, id \mapsto (v, M)^l \\ \textit{EH map: } \ M & ::= & \cdot \mid M, Ev \mapsto \{eh_1^{l_1}, ..., eh_n^{l_n}\} \end{array}$$

The input rules prevent leaks *within* executions by using the labels in σ^{EH} to decide whether to proceed with a declassification. In order to declassify, the source of the event must trust the source of the page element. We use the shorthand $\text{labelOf}(\sigma^{EH}(id))$ to represent the label on the element identified by id in σ^{EH}, and we write $pc \downarrow^i$ to mean the integrity label in pc. Then, an event from a user at security level pc associated with a page element given by id in σ^{EH} is allowed to be declassified when the following holds: $\text{labelOf}(\sigma^{EH}(id)) \sqsubseteq pc \downarrow^i$ (rule In-Release). Otherwise, rule In-No-Release only runs the event in the executions which have enough privilege to see the event and who trust the user.

We use the declassification function described in Section 5.1 to prevent leaks *between* executions. The updated declassification rules are shown in Figure 8. In addition to looking up the declassified event(s) and the execution(s) they will run in, robust throws out any executions where the source of the event doesn't trust the source of the page element. Rule robust handles the case where the user trusts the source of the code (the event is sent to the execution), and rule not-robust handles the case where they do not (the execution does not receive the event). Then, the lookup semantics (judgement $l, \mathsf{r} \vdash \Sigma, E \rightsquigarrow \mathsf{ks}$) ensure only the trusted event handlers run. We define $(eh, l') \downarrow_l$ as eh when $l' \sqsubseteq l$ and $\cdot$ otherwise. When there is at least one event handler the user trusts ($\Sigma(pc)(id.Ev(v)) \downarrow_l = c$), rule lookup-R adds the trusted event handlers to ks and attaches a label $\Sigma(pc) \sqcup \Sigma(pc)(id.Ev(v))$ reflecting the source of the code. When there are no trusted event handlers ($\Sigma(pc)(id.Ev(v)) \downarrow_l = \cdot$), rule lookup-notR moves to the next execution receiving the declassified event. The rules adding a new page element (new) or event handler (add-eh) from the command semantics are responsible for assigning the labels in the event handler store, where l_{src} is the label from rule lookup-R.

Example: We assume that the initial SME state is well-formed, i.e., page elements and event handlers are trusted by the execution context they appear in: execution (l_c, l_i) should trust the page elements and their event handlers, from source l_i', that is, $l_i' \sqsubseteq l_i$.

For our example of leaks *within* an execution, there are three event handlers. onLoad^U is added by the attacker via *ad.com*, who is Untrusted, and onLoad^T is added by the host via *news.com*, who is Trusted. These event handlers are associated with the *body* of the page, which we treat as Trusted. Recall that we assume that the source of the code is trusted by the execution, meaning code from *ad.com* only runs in the Untrusted executions and code from *news.com* runs in both the Untrusted and Trusted executions. Then, the initial SME state with integrity labels is:

$$\begin{array}{lll} \Sigma_0 = & (S, U) \mapsto & \mathit{body} \mapsto (\_, \text{load} \mapsto \{\text{onLoad}^U, \text{onLoad}^T\})^T \\ & (P, U) \mapsto & \mathit{body} \mapsto (\_, \text{load} \mapsto \{\text{onLoad}^U, \text{onLoad}^T\})^T \\ & (S, T) \mapsto & \mathit{body} \mapsto (\_, \text{load} \mapsto \{\text{onLoad}^T\})^T \\ & (P, T) \mapsto & \mathit{body} \mapsto (\_, \text{load} \mapsto \{\text{onLoad}^T\})^T \end{array}$$

Now, when the onLoad^U event handler runs, the execution knows the code came from an Untrusted source because of the label U. When the event handler adds the "Click me!" button, rule new uses the label on the page element T and event handler U to determine the trustworthiness of the new button $T \sqcup U = U$. The state after adding the "Click me!" button to the (S, U) execution is:

$$\begin{array}{lll} \Sigma_1 = & (S, U) \mapsto & \mathit{body} \mapsto (...)^T, b_{\text{Agree}} \mapsto (\textit{“Click me!”}, \{\ \})^U \\ & \cdots & \end{array}$$

$$\boxed{\mathcal{P}, \mathcal{D} \vdash K \overset{(\alpha, pc)}{\Longrightarrow} K'}$$

$$\frac{\begin{array}{c} \mathcal{P}(id.Ev(v)) = pc' \qquad \mathsf{labelOf}(\Sigma(pc)(id)) \sqsubseteq pc\downarrow^i \\ E = ((id.Ev(v), pc'') \mid pc'' \in \mathcal{L} \text{ s.t. } pc \sqcup pc' \sqsubseteq pc'') \\ (\mathcal{R}', E_d) = \mathsf{declassify}(\mathcal{D}, \mathcal{R}, \Sigma, (id.Ev(v), pc), pc') \\ \Sigma, E \rightsquigarrow \mathsf{ks} \qquad pc\downarrow^i, \mathsf{r} \vdash \Sigma, E_d \rightsquigarrow \mathsf{ks}_d \end{array}}{\mathcal{P}, \mathcal{D} \vdash \mathcal{R}; \Sigma; \cdot \overset{(id.Ev(v), pc)}{\Longrightarrow} \mathcal{R}'; \Sigma; \mathsf{ks} :: \mathsf{ks}_d} \ \textsc{In-Release}$$

$$\frac{\begin{array}{c} \mathcal{P}(id.Ev(v)) = pc' \qquad \mathsf{labelOf}(\Sigma(pc)(id)) \not\sqsubseteq pc\downarrow^i \\ E = ((id.Ev(v), pc'') \mid pc'' \in \mathcal{L} \text{ s.t. } pc \sqcup pc' \sqsubseteq pc'') \\ \Sigma, E \rightsquigarrow \mathsf{ks} \end{array}}{\mathcal{P}, \mathcal{D} \vdash \mathcal{R}; \Sigma; \cdot \overset{(id.Ev(v), pc)}{\Longrightarrow} \mathcal{R}'; \Sigma; \mathsf{ks}} \ \textsc{In-No-Release}$$

$$\boxed{\mathsf{declassify}(\mathcal{D}, \mathcal{R}, \Sigma, (id.Ev(v), pc), pc_{Ev}) = (\mathcal{R}', E)}$$

$$\frac{\begin{array}{c} \mathcal{D}((id.Ev(v), pc), pc', \rho) = (\rho', v_d, E_d) \\ d' = \mathsf{update}(d, v_d) \qquad E = \mathsf{robust}(\Sigma, E_d, pc\downarrow^i) \end{array}}{\mathsf{downgrade}_{\mathcal{D}}((\rho, d), \Sigma, (id.Ev(v), pc), pc') = ((\rho', d'), E)} \ \textsc{declassify}$$

$$\boxed{\mathsf{robust}(\Sigma, E, pc_{Ev}) = E'}$$

$$\frac{\mathsf{labelOf}(\Sigma(pc)(id)) \sqsubseteq l}{\begin{array}{c} \mathsf{robust}(\Sigma, ((id.Ev(v), pc) :: E), l) = \\ (id.Ev(v), pc) :: \mathsf{robust}(\Sigma, E, l) \end{array}} \ \textsc{robust}$$

$$\frac{\mathsf{labelOf}(\Sigma(pc)(id)) \not\sqsubseteq l}{\begin{array}{c} \mathsf{robust}(\Sigma, ((id.Ev(v), pc) :: E), l) = \\ \mathsf{robust}(\Sigma, E, l) \end{array}} \ \textsc{not-robust}$$

$$\boxed{pc, \mathsf{r} \vdash \Sigma, E \rightsquigarrow \mathsf{ks}}$$

$$\frac{\Sigma(pc)(id.Ev(v)) \downarrow_l = c \qquad \kappa = \cdot, c, P, \cdot \qquad l, \mathsf{r} \vdash \Sigma, E \rightsquigarrow \mathsf{ks}}{\begin{array}{c} l, \mathsf{r} \vdash \Sigma, (id.Ev(v), pc) :: E \rightsquigarrow \\ (\kappa, pc, \Sigma(pc) \sqcup \Sigma(pc)(id.Ev(v))) :: \mathsf{ks} \end{array}} \ \textsc{lookup-R}$$

$$\frac{\Sigma(pc)(id.Ev(v)) \downarrow_l = \cdot \qquad l, \mathsf{r} \vdash \Sigma, E \rightsquigarrow \mathsf{ks}}{l, \mathsf{r} \vdash \Sigma, (id.Ev(v), pc) :: E \rightsquigarrow \mathsf{ks}} \ \textsc{lookup-notR}$$

$$\frac{}{l, \mathsf{r} \vdash \Sigma, \cdot \rightsquigarrow \cdot} \ \textsc{lookup-Remp}$$

$$\boxed{l_{src}, d \vdash \Sigma, \sigma, c \xrightarrow{\alpha}_{pc} \Sigma', \sigma', c', E}$$

$$\frac{\Sigma(pc) = \sigma^{EH} \qquad id \notin \sigma^{EH} \qquad \begin{array}{c} [\![e]\!]^{pc}_{\sigma,\Sigma} = v \\ \Sigma' = \Sigma[pc \mapsto \sigma^{EH}[id \mapsto (v, \cdot)^{l_{src}}]] \end{array}}{l_{src}, d \vdash \Sigma, \sigma, \mathsf{new}(id, e) \xrightarrow{\bullet}_{pc} \Sigma', \sigma, \mathsf{skip}, \cdot} \ \textsc{new}$$

$$\frac{\begin{array}{c} \Sigma(pc) = \sigma^{EH} \qquad \sigma^{EH}(id) = (v, M)^{l_{id}} \\ \Sigma' = \Sigma[pc \mapsto \sigma^{EH}[id \mapsto (v, M[Ev \mapsto M(Ev) \cup eh^{l_{src}}])^{l_{id}}]] \end{array}}{l_{src}, d \vdash \Sigma, \sigma, \mathsf{addEh}(id, Ev, eh) \xrightarrow{\bullet}_{pc} \Sigma', \sigma, \mathsf{skip}, \cdot} \ \textsc{add-EH}$$

Figure 8: Robust declassification. Key changes are in red.

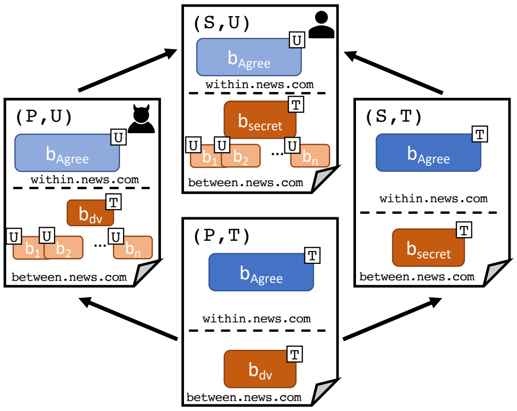


Figure 9: Insecure example from Section 3 with robustness checks. The labels tell us the trustworthiness of the source of the page elements and event handlers, depicted here as small white labels on each page element.

Figure 9 shows the resulting page after all of the buttons are loaded, including their labels. When the user clicks the "Click me!" button on the (S, U) copy of the page, the input rules will use the label on the button to determine if the declassification is allowed. The user is treated as a Trusted source of events, so because $U \not\sqsubseteq T$, rule In-No-Release prevents the event from being declassified and the attacker doesn't learn the user's settings.

For our example of leaks *between* executions, the host installs an onKeypress event handler to some field which adds a different button to the page depending on what the user types, and the attacker adds all possible buttons to the page. After the user presses a key, the SME store has one Trusted button per execution, and several Untrusted buttons in the Untrusted executions:

$$\begin{aligned} \Sigma_0 = \ & (S, U) \mapsto b_{\text{secret}} \mapsto (\ldots)^T, b_1 \mapsto (\ldots)^U, \ldots, b_n \mapsto (\ldots)^U \\ & (P, U) \mapsto b_{\text{dv}} \mapsto (\ldots)^T, b_1 \mapsto (\ldots)^U, \ldots, b_n \mapsto (\ldots)^U \\ & (S, T) \mapsto b_{\text{secret}} \mapsto (\ldots)^T \\ & (P, T) \mapsto b_{\text{dv}} \mapsto (\ldots)^T \end{aligned}$$

When the user clicks the b_{secret} button on the (S, U) copy of the page, rule In-Release attempts to declassify the event to the (P, U) execution since the button is Trusted. Next, the robust rules use the labels on the button capturing the event to determine if the (P, U) execution should receive the declassified event. In this case, the button b_i capturing the event was added by the attacker. Since $U \not\sqsubseteq T$, rule not-robust skips the (P, U) execution and the attacker does not learn which key the user pressed.

6 SECURITY

We define two security conditions and prove that SME^T satisfies them. First, we define a knowledge-based progress-insensitive noninterference with declassification (Section 6.1) which ensures that the attacker's knowledge of the secret inputs is not refined as the system runs outside of what is declassified (and the fact that the system makes progress). Second, we describe a novel influence-based progress-insensitive noninterference (Section 6.2) which is the integrity dual to the knowledge-based security condition to demonstrate that SME^T do not allow the attacker to influence the more trusted components of the system (except the fact that the system makes progress). Finally, we show if we treat declassification as a trusted behavior, the influence-based security condition may be extended so that robust declassification follows.

6.1 Knowledge-based security (confidentiality)

Knowledge-based security conditions allow precise specification of what information (if any) is leaked. We informally define several knowledge conditions (summarized in Figure 10) to set up our knowledge-based progress-insensitive noninterference definition. Formal definitions and proofs may be found in our companion technical report [29].

For someone with enough privilege to observe data up to label l, their knowledge is the set of all possible inputs which might have produced the observations they made. Knowledge can also be thought of as a measure of *uncertainty*. As the attacker learns more, they will become more confident about the inputs received by the system and the knowledge set will become smaller (i.e., the attacker has become more certain about what the inputs might have been). We define the knowledge of an observer with privilege $l \in \mathcal{L}_c$:

$$\begin{aligned}\mathcal{K}(T, \Sigma_0, \mathcal{R}, \mathcal{P}, l) = \{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{P}),\\ T \approx_l^c T', \tau = \mathsf{in}(T')\}\end{aligned}$$

The knowledge of an observer with privilege l is the set of all inputs from execution traces T' ($\tau = \mathsf{in}(T')$) that have *observationally equivalent at l* to T ($T \approx_l^c T'$) and start from the same initial state with the same security and declassification policies ($T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{P})$). For now, an input is a user-generated event ($id.Ev(v)$). We say that two runs are *observationally equivalent at l*, $T \approx_l^c T'$, if they look the same to an observer with privilege l (i.e., they make the same outputs on any l-visible channel and the l-visible executions behave the same) $T \downarrow_l^c = T' \downarrow_l^c$. The *observation* of a trace is the sequence of actions observable by an attacker and include inputs, outputs, silent actions,[2] and declassifications $\mathsf{rls}(...)$.

$$\textit{Sequence of actions} : \tau \quad ::= \quad \cdot \mid \tau :: \alpha \mid \tau :: \mathsf{rls}(id.Ev(v), \mathcal{R}, E)$$

The rules for the observation of a trace are shown in Figure 11. Note that $T \downarrow_l^p$ is parameterized by p, where $p = c$ is for confidentiality and $T \downarrow_l^c$ is the observation of a trace at l, and $p = i$ will be for integrity (Section 6.2) and $T \downarrow_l^i$ is the behavior of a trace at l. The observation of an output is $ch(v)$ if the output is made on an observable channel $\mathcal{P}(ch) \sqsubseteq l$ or by an observable execution $pc \downarrow^p \sqsubseteq l$ (rule TP-Out2), otherwise the output is skipped (rule TP-Out-S1). Inputs are observable if the security policy and source is observable (rule TP-In), and declassifications are observable if they are successful (rule TP-In-R). Other actions are observable if they happen in an observable execution (rules TP-Out1); otherwise, they are skipped (rule TP-Out-S2).

A knowledge-based progress-sensitive noninterference says that an attacker should not be able to refine their knowledge of the secret inputs by watching the system run:

$$\mathcal{K}(T, \Sigma_0, \mathcal{R}, \mathcal{P}, l) \subseteq_{\preceq} \mathcal{K}(T \Longrightarrow K, \Sigma_0, \mathcal{R}, \mathcal{P}, l)$$

We write $A \subseteq_{\preceq} B$ to mean that each element of A is a prefix of an element in B (since the last step of $T \Longrightarrow K$ may be an input). When the system takes a step ($T \Longrightarrow K$), the attacker's knowledge should not be refined; they should be equally uncertain about the possible secret inputs before and after the step. Because we run event handlers in a single-thread, it is possible for an event handler to get "stuck" in an infinite loop, which could leak something to the attacker if the loop condition is secret. Therefore, we will permit this leak and prove *progress-insensitive* noninterference instead. We define *progress knowledge* as the set of traces producing the same outputs *and* making enough progress to accept another input.

A knowledge-based progress-insensitive security condition is:

$$\mathcal{K}_p(T, \Sigma_0, \mathcal{R}, \mathcal{P}, l) \subseteq_{\preceq} \mathcal{K}(T \Longrightarrow K, \Sigma_0, \mathcal{R}, \mathcal{P}, l)$$

When the system takes a step, the attacker's knowledge should not be refined (except that they learn the system makes progress).

This definition has yet to capture declassification. For example, if a user's click on a hat b_{Hat} is declassified for analytics (like for the shop from Section 3), the attacker's knowledge would be refined to inputs that include the click on b_{Hat}. This leak is permitted by declassification, but not by the definition above. Therefore, we define *release knowledge* as the set of traces producing the same outputs, making progress, *and* releasing *the same event*. Our definition for knowledge-based progress-insensitive noninterference with declassification says that, outside of declassification, the attacker should not learn anything by watching the system take a step (except that the system has made progress) and when something is declassified, the attacker should only learn what is declassified. We say $\mathsf{releaseA}(T \Longrightarrow K)$ if $(\mathsf{last}(T) \Longrightarrow K) \downarrow_l^c = \mathsf{rls}(...)$, where $\mathsf{last}(T)$ is the last state in T. That is, $\mathsf{releaseA}(T \Longrightarrow K)$ means something was declassified in the last step.

Definition 1 (PINI with Declassification). A system satisfies progress-insensitive noninterference, outside of what is declassified, against l-observers for $l \in \mathcal{L}_c$ iff given any initial global store Σ_0, security policy $\mathcal{P}$, and declassification policy $\mathcal{R}$, it is the case that for all traces T, actions α, and configurations K s.t. $(T \overset{\alpha}{\Longrightarrow} K) \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{P})$, then, the following holds

- *If* $\mathsf{releaseA}(T \overset{\alpha}{\Longrightarrow} K)$*:* $\mathcal{K}(T \overset{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{K}_{rp}(T, \Sigma_0, \mathcal{R}, \mathcal{P}, \alpha, l)$
- *Otherwise:* $\mathcal{K}(T \overset{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{P}, l) \supseteq_{\preceq} \mathcal{K}_p(T, \Sigma_0, \mathcal{R}, \mathcal{P}, l)$

Example Recall Example 3.2 of leaks *between* executions. The security policy is that keypress events should be *S*ecret, but clicks may be declassified from *S*ecret to *P*ublic. Which button is added by the host depends on what the user types. The attacker adds all buttons $b_1, ..., b_n$ and registers an onClick event handler to each button which outputs i to a (P, U) channel if registered for b_i. When the user types, the attacker isn't sure which key is pressed. Their knowledge at this point includes all possible keypresses:

$$\mathcal{K}(K, \Sigma_0, \mathcal{R}, \mathcal{P}, P) = \{f.\mathsf{keyPress}(1), ..., f.\mathsf{keyPress}(n)\}$$

The keypress triggers the onKeypress event handler which adds button b_i to the user's page if they pressed i. Suppose the attacker also registered a Click event handler to b_{secret} to directly leak the user's keypress through a (P, U) channel. If the output were allowed, the attacker would be able to eliminate the traces where the user pressed a different key which refines their knowledge.

$$\begin{aligned}\mathcal{K}(K \overset{ch(i)}{\Longrightarrow} K', \Sigma_0, \mathcal{R}, \mathcal{P}, P) = \{\cancel{f.\mathsf{keyPress}(1)}, ...,\\ f.\mathsf{keyPress}(\textit{secret}) :: b_{\mathsf{secret}}.\mathsf{Click}(\_), ..., \cancel{f.\mathsf{keyPress}(n)}\}\end{aligned}$$

[2] We consider silent actions observable only when they come from an observable execution, which makes proofs for observable executions more uniform. Since our equivalence definitions force observable executions to be the same anyway, this choice does not affect our security conditions.

Knowledge	$\mathcal{K}(T, \Sigma_0, \mathcal{R}, \mathcal{P}, l) =$ $\{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{P}), T \approx^c_l T', \tau = \mathsf{in}(T')\}$	All possible inputs producing the same observations
Progress Knowledge	$\mathcal{K}_p(T, \Sigma_0, \mathcal{R}, \mathcal{P}, l) =$ $\{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{P}), T \approx^c_l T', \tau = \mathsf{in}(T'), \mathsf{prog}(T')\}$	All possible inputs producing the same observations *and* accept another input: $\mathsf{prog}(T')$ holds if T' can reach the consumer state
Release Knowledge	$\mathcal{K}_{rp}(T \xRightarrow{\alpha} K, \Sigma_0, \mathcal{R}, \mathcal{P}, l) =$ $\{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{P}), T \approx^c_l T',$ $\tau = \mathsf{in}(T'), \mathsf{prog}(T'), \mathsf{releaseT}(T', \alpha))\}$	All possible inputs producing the same observations, accept another input, *and* release the same event: $\mathsf{releaseT}(T', \alpha)$ holds if T' can be extended to release the same event α

Figure 10: Knowledge definitions. Knowledge and progress knowledge are for defining a knowledge-based progress-insensitive noninterference. Release knowledge accounts for what is leaked to the attacker through declassification.

$$\boxed{T \downarrow^p_l = \tau}$$

$$\frac{}{\mathcal{P} \vdash K \downarrow^p_l = \cdot}\ \text{TP-Base}$$

$$\frac{pc \downarrow^p \sqsubseteq l \qquad \alpha \notin \{id.Ev(v), ch(v)\}}{(\mathcal{P}, \mathcal{D} \vdash K \xRightarrow{(\alpha,pc)} T') \downarrow^p_l = \alpha :: T' \downarrow^p_l}\ \text{TP-Out1}$$

$$\frac{pc \downarrow^p \sqsubseteq l \vee \mathcal{P}(ch) \downarrow^p \sqsubseteq l}{(\mathcal{P}, \mathcal{D} \vdash K \xRightarrow{(ch(v),pc)} T') \downarrow^p_l = ch(v) :: T' \downarrow^p_l}\ \text{TP-Out2}$$

$$\frac{pc \downarrow^p \not\sqsubseteq l \downarrow^p \qquad \mathcal{P}(ch) \downarrow^p \not\sqsubseteq l}{(\mathcal{P}, \mathcal{D} \vdash K \xRightarrow{(ch(v),pc)} T') \downarrow^p_l = T' \downarrow^p_l}\ \text{TP-Out-S1}$$

$$\frac{\alpha \notin \{id.Ev(v), ch(v)\} \qquad pc \downarrow^p \not\sqsubseteq l}{(\mathcal{P}, \mathcal{D} \vdash K \xRightarrow{(\alpha,pc)} T') \downarrow^p_l = T' \downarrow^p_l}\ \text{TP-Out-S2}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.Ev(v)) = pc' \qquad \Sigma(pc)(id) \not\sqsubseteq pc \downarrow^i \\ \tau = id.Ev(v) \text{ if } pc' \downarrow^p \sqcup pc \downarrow^p \sqsubseteq l \qquad \tau = \cdot \text{ otherwise}\end{array}}{(\mathcal{P}, \mathcal{D} \vdash \_; \Sigma; \_ \xRightarrow{(id.Ev(v),pc)} T') \downarrow^p_l = \tau :: T' \downarrow^p_l}\ \text{TP-In}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.Ev(v)) = pc' \qquad \Sigma(pc)(id) \sqsubseteq pc \downarrow^i \\ (\mathcal{R}', E) = \mathsf{declassify}(\mathcal{D}, \mathcal{R}, \Sigma, (id.Ev(v), pc), pc') \\ \tau = \mathsf{rls}(id.Ev(v), \mathcal{R}', E \downarrow^p_l) \text{ if } \mathcal{R} \neq \mathcal{R}' \\ \tau = id.Ev(v) \text{ if } \mathcal{R} = \mathcal{R}' \wedge pc \downarrow^p \sqcup pc \downarrow^p \sqsubseteq l \\ \tau = \cdot \text{ otherwise}\end{array}}{(\mathcal{P}, \mathcal{D} \vdash \mathcal{R}; \Sigma; \_ \xRightarrow{(id.Ev(v),pc)} T') \downarrow^p_l = \tau :: T' \downarrow^p_l}\ \text{TP-In-R}$$

Figure 11: The observation (p=c) or behavior (p=i) of T at l

Our knowledge-based security condition would correctly identify this output as insecure because: $\mathcal{K}(K \xRightarrow{ch(i)} K', \ldots) \not\supseteq \mathcal{K}(K, \ldots)$

In reality, the SME monitor would prevent the output from the (S, U) execution to the (P, U) channel. The user's click would not be able to directly leak their keypress to the attacker, but it could be declassified to the (P, U) execution. Since the attacker added all possible buttons $b_1, \ldots, b_n$, they are guaranteed to trigger the leaky output and learn which key the user pressed. Because the releaseA condition allows the attacker's knowledge to be refined by declassifications, our security condition for confidentiality does not catch this leak. Next, we describe our security condition for integrity and how this condition can be used to describe both progress-insensitive noninterference as well as robust declassification.

6.2 Influence-based security (integrity)

We measure the attacker's ability to change the behavior of the system with a dual condition to knowledge called *influence*, (based on attacker power [5]). At a high level, an attacker's influence is the set of all untrusted inputs which might have produced the same trusted behaviors. The attacks in the influence set have the same relative ability to influence the system's behavior. If the attacker has no influence over the system, then the set should include all possible attacks: all of the attacks are equally powerless. As the system runs, the refinement of attacker's influence indicates that some attacks are more powerful than the others because the ones eliminated couldn't have led to the observed behavior. We define the attacker's influence over behaviors at l (for $l \in \mathcal{L}_i$) below:

$$\mathcal{I}(T, \Sigma_0, \mathcal{R}, \mathcal{P}, l) = \{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{P}), T \approx^i_l T' \wedge \tau = \mathsf{in}(T')\}$$

The influence of an attacker over behaviors at l is the set of all τ which are inputs from execution traces T' ($\tau = \mathsf{in}(T')$) that are *behaviorally equivalent at* l to T ($T \approx^i_l T'$) and start from the same initial state with the same security and declassification policies ($T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{P})$). We say that two runs are *behaviorally equivalent at* l if they produce the same l-trusted actions (i.e., they make the same outputs on any l-trusted channel and the l-trusted executions behave the same). $T \downarrow^p_l$ is defined in Figure 11. We summarize our influence definitions in Figure 12.

An influence-based progress-sensitive noninterference says the attacker's influence over a system should never be refined:

$$\mathcal{I}(T, \Sigma_0, \mathcal{R}, \mathcal{P}, l) \subseteq_{\preceq} \mathcal{I}(T \Longrightarrow K, \Sigma_0, \mathcal{R}, \mathcal{P}, l)$$

Similar to progress knowledge, we define *progress influence* as the set of traces producing the same behaviors *and* making enough progress to accept another input. Then, an influence-based progress-insensitive security condition states that when the system takes a step, the attacker's influence should not be refined, outside of what control they have over whether the system makes progress:

$$\mathcal{I}_p(T, \Sigma_0, \mathcal{R}, \mathcal{P}, l) \subseteq_{\preceq} \mathcal{I}(T \Longrightarrow K, \Sigma_0, \mathcal{R}, \mathcal{P}, l)$$

6.3 Robust declassification

In addition to showing that the attacker doesn't have influence over trusted behaviors, we also want to show that the attacker doesn't

Influence	$\mathcal{I}(T, \Sigma_0, \mathcal{R}, \mathcal{P}, l) =$ $\{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{P}), T \approx^i_l T', \tau = \mathsf{in}(T')\}$	All possible inputs producing the same trusted actions
Progress Influence	$\mathcal{I}_p(T, \Sigma_0, \mathcal{R}, \mathcal{P}, l) =$ $\{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{P}), T \approx^i_l T', \tau = \mathsf{in}(T'), \mathsf{prog}(T')$	All possible inputs producing the same trusted actions *and* accept another input: $\mathsf{prog}(T')$ holds if T' can reach the consumer state
Robust Influence	$\mathcal{I}_{rp}(T \overset{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{P}, l) =$ $\{\tau \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{P}), T \approx^i_l T',$ $\tau = \mathsf{in}(T'), \mathsf{prog}(T'), \mathsf{robustT}(T', \alpha))\}$	All possible inputs producing the same trusted actions, accept another input, *and* capable of the same robust declassifications: $\mathsf{robustT}(T', \alpha)$ holds if T' can be extended to create the same trusted page event α

Figure 12: Influence definitions. Influence and progress influence are for defining an influence-based progress-insensitive noninterference. Robust influence is for defining robust declassification.

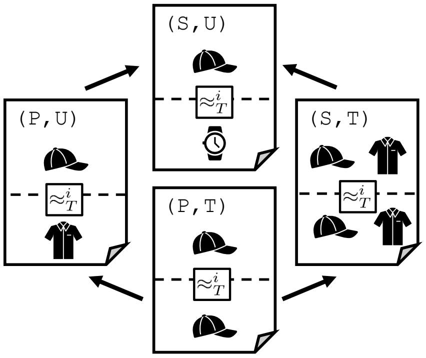


Figure 13: The states above and below the dotted line are behaviorally equivalent at T even there are different products in the (P, U) and (S, U) states.

influence declassification. We can define robust declassification by extending our influence-based security condition.

A naïve formalization of robust declassification is as follows. We model an *active attacker* by treating the addition of a page element or event handler ($\mathsf{new}(id, pc)$, $\mathsf{new}(id, eh, pc)$) as an input. A system is robust if any of these *attacks* have equivalent power. That is, when a new declassification happens, we will know the attacker's code influenced the declassification if the set of attacks *without* their code could not have led to the same declassification. However, it turns out that this definition is too strong and leads to false positives.

Consider the online shop described in Section 3. The buttons are all loaded by the Trusted host, so they can safely influence declassification: the declassifications in this example are robust. The issue is that behavioral equivalence at T only guarantees that the Trusted executions behave the same. See the example of two equivalent traces in Figure 13. The (S, T) execution has the same products in both traces, as does the (P, T) shop, but even among two equivalent runs, the (S, U) and (P, U) executions may have different products. When the user clicks b_{Hat} in the (S, U) execution, the click is declassified. But it isn't possible to produce the same declassification in the equivalent state because there is no b_{Hat} for the user to click on. This makes it appear as though the attacker had some influence over the declassification, even though the declassification is actually robust against their influence.

$$\frac{\begin{array}{c}\alpha \in \{\mathsf{new}(id, l_{src}), \mathsf{new}(id, eh, l_{id}, l_{src})\} \\ pc \downarrow^i \not\sqsubseteq l \qquad \tau = \mathsf{r}(id, pc) \text{ if } l_{src} \sqsubseteq pc \downarrow^i \\ \tau = \mathsf{r}(id, eh, pc) \text{ if } l_{id} \sqcup l_{src} \sqsubseteq pc \downarrow^i \qquad \tau = \cdot \text{ otherwise}\end{array}}{(\mathcal{P}, \mathcal{D}, \vdash K \overset{(\alpha,pc)}{\Longrightarrow} T') \downarrow^i_l = \tau :: T' \downarrow^i_l} \text{TP-NEW}$$

Figure 14: New rule for the behavior of a trace for robust declassification.

To make these benign influence refinements concrete, we introduce *robust influence* for when trusted page elements are created. Robust influence is the set of traces producing the same elements, making progress, *and* capable of producing the same robust declassifications in the untrusted executions. This is similar to release knowledge from Section 6.1. We say $\mathsf{robustA}(T)$ if $(\mathsf{last}(T) \Longrightarrow K) \downarrow^i = \mathsf{r}(...)$, where $\mathsf{last}(T)$ is the last state in T. That is, $\mathsf{robustA}(T \Longrightarrow K)$ means something capable of robust declassification was added to an Untrusted execution.

To model an *active* attacker's ability to add code to the page, we emit an action for dynamically-generated elements and event handlers. $\mathsf{new}(id, pc)$ is a new page element identified, id, added to the pc execution, while $\mathsf{new}(id, eh, pc)$ is a new event handler eh registered to the element identified by id in the execution at security level pc. Sequences of actions include page elements/event handlers *which are capable of robust declassification* $\mathsf{r}(...)$.

$$\begin{array}{llll} \textit{Actions:} & \alpha & ::= & id.Ev(v) \mid ch(v) \\ & & \mid & \mathsf{new}(id, pc) \mid \mathsf{new}(id, eh, pc) \mid \bullet \\ \textit{Sequence of actions}: \tau & & ::= & \cdot \mid \tau :: \alpha \mid \tau :: \mathsf{rls}(id.Ev(v), \mathcal{R}, E) \\ & & & \mid \tau :: \mathsf{r}(id, pc) \mid \tau :: \mathsf{r}(id, eh, pc) \end{array}$$

We modify the behavior of a trace as shown in Figure 14. When a new page element is created or event handler is registered, this is not considered an observable action unless it is capable of a robust declassification (rule TP-New).

Definition 2 (Influence-based PINI with Robust Declassification). A system satisfies progress-insensitive noninterference with robust declassification for behaviors at $l \in \mathcal{L}_i$ iff given any initial global store Σ_0, security policy $\mathcal{P}$, and declassification policy $\mathcal{R}$, it is the case that for all traces T, actions α, and configurations K s.t. $(T \overset{\alpha}{\Longrightarrow} K) \in$ $\mathsf{runs}(\Sigma_0, \mathcal{R}, \mathcal{P})$, then, the following holds

- *If* $\mathsf{robustA}(T \overset{\alpha}{\Longrightarrow} K)$*:* $\mathcal{I}(T \overset{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{P}, l) \supseteq_{\leq} \mathcal{I}_{rp}(T, \Sigma_0, \mathcal{R}, \mathcal{P}, \alpha, l)$
- *Otherwise:* $\mathcal{I}(T \overset{\alpha}{\Longrightarrow} K, \Sigma_0, \mathcal{R}, \mathcal{P}, l) \supseteq_{\leq} \mathcal{I}_p(T, \Sigma_0, \mathcal{R}, \mathcal{P}, l)$

Example: To illustrate how this new definition is sufficient for defining robust declassification, we will walk through examples from Section 3. In Example 3.1 of a leak *within* an execution, the Untrusted attacker registers the event handler onLoad^U and the Trusted host registers onLoad^T to add buttons to the page.

After the page finishes loading, we know that the Trusted *"Agree"* button, b_{Agree}, must have been dynamically loaded because all of the behaviorally-equivalent Trusted executions have run onLoad^T. On the other hand, we aren't sure whether the Untrusted "Click me!" button, was added because the Untrusted pages are equivalent whether or not onLoad^U has run. At this point, the attacks where the *"Click me!"* button has been added are equally as powerful as the attacks without it:[3]

$$\mathcal{I}(K, \Sigma_0, \mathcal{R}, \mathcal{P}, T) = \{\mathsf{new}(b)^T, \mathsf{new}(b)^U :: \mathsf{new}(b)^T, ...\}$$

If the system allowed the click on the Untrusted b_{Agree} to be declassified, it would mean there *must* be a *"Click me!"* button on the (S, U) copy of the page. Therefore, the only viable attack leading to this behavior are the ones including the Untrusted b_{Agree} button:

$$\begin{aligned}\mathcal{I}(T \overset{b^U}{\Longrightarrow} K, \mathcal{R}, \mathcal{P}, T) = \{&\cancel{\mathsf{new}(b)^T},\\ &\mathsf{new}(b)^U :: \mathsf{new}(b)^T :: b^U.\mathsf{Click}(\,), ...\}\end{aligned}$$

Because $\mathcal{I}(T \overset{b^U}{\Longrightarrow} K, \mathcal{R}, \mathcal{P}, T) \not\supseteq_{\leq} \mathcal{I}_p(T, \mathcal{R}, \mathcal{P}, T)$, the attacker must have had influence over the declassification, so it isn't robust.

Example 3.2 of leaks *between* executions is similar. The Trusted host adds a different button to the page depending on what the user has typed, and the Untrusted attacker adds all possible buttons.

After the user presses a key on their keyboard, we know that there is one button on the (S, T) page (based on the actual *secret* value) and another button on the (U, T) page (based on the default value dv) because all of the behaviorally-equivalent Trusted executions have run the Trusted event handler in response to the user's keypress. We also know that the (S, U) and (P, U) copies of the page must include b_{secret} and b_{dv} (respectively) because those buttons are capable of robust declassification since they were added by the host. On the other hand, we aren't sure whether the attacker has added their buttons, because the Untrusted pages are equivalent with or without those buttons:

$$\begin{aligned}\mathcal{I}(K, \Sigma_0, \mathcal{R}, \mathcal{P}, l) = \{&\mathsf{new}(b_{\text{secret}})^S :: \mathsf{new}(b_{\text{dv}})^P,\\ &\mathsf{new}(b_{\text{secret}})^S :: \mathsf{new}(b_{\text{dv}})^P :: \mathsf{new}(b_1)^U :: ... :: \mathsf{new}(b_n)^U, ...\}\end{aligned}$$

Now, when the user's click on b_{secret} in the (S, U) page is declassified to the matching button b_i in the (P, U) page, we know there must be a b_i button on the (P, U) copy of the page to capture the event. Then, the only viable attack is the one where b_i has been added to the page:

$$\begin{aligned}\mathcal{I}(K, \Sigma_0, \mathcal{R}, \mathcal{P}, l) = \{&\cancel{\mathsf{new}(b_{\text{secret}})^S :: \mathsf{new}(b_{\text{dv}})^P},\\ &\mathsf{new}(b_{\text{secret}})^S :: \mathsf{new}(b_{\text{dv}})^P :: \mathsf{new}(b_1)^U :: ... :: \mathsf{new}(b_n)^U, ...\}\end{aligned}$$

Since the attacker's influence has been refined we know this example is not robust either.

Finally, consider the secure web shop where the host adds products to the page and declassifies click counts so that a (P, U) library can do analytics for them. All of the elements are added by the Trusted host, so they are capable of robust declassification. From the robustA case in Definition 2, the attacker's influence can be refined by the addition of these elements to include only the traces that load the same products on the web store. This means that declassifying a user's click won't refine the attacker's influence and our security condition correctly identifies this as robust.

6.4 Metatheory

We prove that our semantics are sound. Formally:

Theorem 3 (Soundness). $\forall \mathcal{P}, \mathcal{D}, \Sigma_0$, the SME state Σ_0 satisfies knowledge-based progress-insensitive noninterference with declassification at $l_c \in \mathcal{L}_c$ and influence-based progress-insensitive noninterference with robust declassification at $l_i \in \mathcal{L}_i$ w.r.t. the security policy $\mathcal{P}$ and declassification policy $\mathcal{D}$.

Complete proofs may be found in our companion technical report [29]. Robust declassification follows from influence-based progress-insensitive noninterference. If declassifications are trusted and we prove that untrusted sources cannot influence trusted behaviors, then it must be the case that the declassifications are robust.

Corollary 4 (Robust Declassification). $\forall \mathcal{P}, \mathcal{D}, \Sigma_0$ s.t. Σ_0 satisfies influence-based progress-insensitive noninterference at l_i w.r.t. the security policy $\mathcal{P}$ and declassification policy $\mathcal{D}$, then an attacker at $l_i' \in \mathcal{L}_i$ with $l_i' \not\sqsubseteq l_i$ has no influence over whether the user's events at l_i are declassified.

7 IMPLEMENTATION AND EVALUATION

We have prototyped SME^T in OCaml on top of Featherweight Firefox [15], which is a lightweight implementation of the web browser model. The implementation provides a sanity check on the semantics and helps understand the behavior of programs that restrict declassification to certain cases. The original Featherweight Firefox does not include recent browser features, but is expressive enough to enforce all features of our formalization and demonstrate its feasibility in a browser-like setting. Our implementation is available at **https://github.com/CompIFC/tainted-sme.git**. We leave enforcement in a real browser to future work.

We modify the model to attach labels to nodes on a web page. The host page has higher integrity than the user and third-party integrity. We leverage the implementation from prior work [13] to label input events and the outputs generated. The labels of the nodes are fixed across all executions of the browser model. We omit the trigger command from the semantics and assume that all events are user-generated. We implement the release module to perform declassification as per the release policies. When an input event is received, we check the context (label) of the event (which is user in our case) and compare it against the label of the node on which the event was triggered. If the label on the node is not trusted by the source of the event, then the release module is not called. Otherwise, the release module writes the declassified value to a shared channel. For simplicity, we declassify all values in the release module to the confidentiality level P. The declassify command reads the last declassified value for that level.

Evaluation: To compare against SME models with declassification, we also implement versions of the model with the original stateful declassification approach [41] and the one that prohibits declassification on dynamically created elements [30]; we modify the release module to declassify without checking the node label

[3] Due to space constraints, we write $\mathsf{new}(b)^T$ instead of $\mathsf{new}(b_{\text{Agree}}, (S, T))$ and $\mathsf{new}(b_{\text{Agree}}, (P, T))$, and likewise for $\mathsf{new}(b)^U$ for the Untrusted executions

(for the former) and assigning a special label SD, which never declassifies (for the latter). We observe that the example programs presented earlier leak information with unrestricted declassification while our approach is more permissive compared to the approach where declassification is never allowed for events from dynamic elements. In terms of performance overhead, our monitor performs worse compared to the existing approaches due to the operations involving multiple levels and the additional integrity label. More concretely, the overhead of running our monitor as compared to the prior approaches is around 22% and 30%, respectively, for the example presented earlier.

8 DISCUSSION

Robust declassification and attacker control Prior work on robust declassification that is most similar to our setting involves attacker control [5] which is the set of attacks (i.e., untrusted inputs) with a similar effect on knowledge. They say declassification is robust if the attacker control (which are the possible attacks resulting in the *same declassification*) includes all of the attacks *reaching* the declassification; otherwise, the attacker must have influenced the declassification. Our definition is similar. We relate the set of possible attacks before and after declassification and consider the declassification robust if attacks reaching the declassification could also result in the same declassification. The key benefit of our condition over prior work is that robust declassification follows from our influence-based security condition which makes the definitions more uniform and simplifies the proofs.

(Transparent) endorsement and qualified robustness The focus of this work is robust declassification, but like our "influence-based" security condition is the integrity dual of "knowledge-based" security conditions for confidentiality, *(transparent) endorsement* is the integrity dual of *(robust) declassification*. Endorsement allows a program to treat untrusted data as if it were more trusted, and transparent endorsement ensures that the data is sufficiently public before endorsing. The idea being that if the attacker supplies information they do not actually have the privilege to see, we should not trust it. For example, prior work [18] proposing transparent endorsement explains that without restricting endorsement to what data the attacker has the privilege to see, they could cheat in a sealed-bid auction by simply bidding "one more than the other person" (even though they don't know what the other person bid).

In our companion technical report [29], we include (transparent) endorsement by adding an endorsement policy $\mathcal{E}$ and module $\mathcal{S}$, which functions similarly to $\mathcal{D}$ and $\mathcal{R}$. We update the input rules to ensure the source of the event has enough privilege to see the page element ($\Sigma(pc)(id) \downarrow^c \sqsubseteq pc \downarrow^c$). An event may be both declassified and endorsed as long as the *original* event is both robust and transparent (we do not declassify before checking for transparency or vice versa).

The changes to the security conditions are similarly straightforward. We add *sanitized influence* to prove an influence-based progress-insensitive noninterference *with endorsement*. Sanitized influence measures the amount of influence the attacker gains through endorsement and is defined as the set of all possible inputs producing the same trusted actions, accepting another input, and capable of the same endorsements (similar to release knowledge). If we treat endorsements as public, transparent endorsement follows from our knowledge-based security condition if we add *transparent knowledge* which captures the information leaked by adding an element to a secret execution that is capable of transparent endorsement (similar to robust influence). The supporting definitions for these security conditions may be found in the technical report [29].

Note that because an event associated with an attacker-controlled page element might be endorsed, we are actually proving a *qualified robustness* condition [32] (and *qualified transparency*) which says that the attacker does not have influence over declassifications, outside of what has been endorsed (and we do not endorse what the attacker does not have privilege to see, outside of what has been declassified). This does not change our security conditions because sanitized influence (and release knowledge) already capture this, but it does give the attacker more power over what is declassified since untrusted code could be endorsed and then be permitted to influence declassification.

Alternative DOM models: In our model, each execution has its own copy of the DOM, similar to prior work [13, 17, 30]. Another option would be to have a single DOM [23, 41]. In these models, the security policy would determine which API calls would succeed and which would be replaced with a default value. Yet other work looks at the possibility of using a single DOM with SME by tracking secrets (taint) through the nodes, attributes, and event handlers [28]. It would be challenging to allow similar fine-grained declassifications of events related to dynamically generate elements in the first model, and the second model is susceptible to implicit leak through control flow decisions.

Our "DOM" is a flat structure with few APIs since the structure of the DOM did not contribute directly to the relationship between attacker influence and robust declassification. As future work, it would be interesting to have a more realistic tree-structured DOMs [28, 36] to model more complex DOM features [3, 34] to explore whether attacker influence over event bubbling order and pre-emptive event scheduling (for instance) yields new attacks.

9 CONCLUSION

We developed SME^T, an IFC monitor, which combines SME and taint tracking to prevent attackers from influencing declassification. SME^T permits the benign declassifications involving trusted dynamic features—without sacrificing security. We proved that SME^T satisfies progress-insensitive noninterference for both confidentiality and integrity using knowledge-based and influence-based security conditions, respectively. We showed that robust declassification follows from our novel influence-based security condition.

ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation via grants CNS1704542 and CNS2245115, the CyLab Presidential Fellowship at Carnegie Mellon University, and the DST-INSPIRE Faculty grant DST/INSPIRE/04/2020/002092. We would also like to thank the anonymous reviewers for their feedback on our paper.

REFERENCES

[1] Amir A. Ahmadian and Musard Balliu. 2022. Dynamic Policies Revisited. In *Proceedings of the 2022 IEEE 7th European Symposium on Security and Privacy*

(EuroS&P).

[2] Maximilian Algehed, Alejandro Russo, and Cormac Flanagan. 2019. Optimising Faceted Secure Multi-Execution. In *Proceedings of the 2019 IEEE Computer Security Foundations Symposium (CSF).*

[3] Ana Gualdina Almeida Matos, José Fragoso Santos, and Tamara Rezk. 2014. An Information Flow Monitor for a Core of DOM: Introducing References and Live Primitives. In *Proceedings of the International Symposium on Trustworthy Global Computing (TGC).*

[4] Aslan Askarov and Stephen Chong. 2012. Learning is Change in Knowledge: Knowledge-Based Security for Dynamic Policies. In *Proceedings of the 2012 IEEE Computer Security Foundations Symposium (CSF).*

[5] Aslan Askarov and Andrew Myers. 2011. Attacker Control and Impact for Confidentiality and Integrity. *Logical Methods in Computer Science (LMCS)* 7 (2011).

[6] Aslan Askarov and Andrei Sabelfeld. 2007. Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP).*

[7] Thomas H. Austin and Cormac Flanagan. 2009. Efficient purely-dynamic information flow analysis. In *ACM Workshop on Programming Languages and Analysis for Security (PLAS).*

[8] Thomas H. Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. In *Proceedings of the ACM Principles of Programming Languages (POPL).*

[9] Musard Balliu. 2013. A logic for information flow analysis of distributed programs. In *Proceedings of the Nordic Conference on Secure IT Systems (NordSec).*

[10] Anindya Banerjee, David A Naumann, and Stan Rosenberg. 2008. Expressive declassification policies and modular static enforcement. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP).*

[11] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. 2015. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS).*

[12] Abhishek Bichhawat, Vineet Rajani, Jinank Jain, Deepak Garg, and Christian Hammer. 2017. WebPol: Fine-grained Information Flow Policies for Web Browsers. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS).*

[13] Nataliia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. 2011. Reactive non-interference for a browser model. In *Proceedings of the International Conference on Network and System Security (NSS).*

[14] Nataliia Bielova and Tamara Rezk. 2016. Spot the Difference: Secure Multi-execution and Multiple Facets. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS).*

[15] Aaron Bohannon and Benjamin C. Pierce. 2010. Featherweight Firefox: Formalizing the Core of a Web Browser. In *USENIX Conference on Web Application Development (WebApps 10).*

[16] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. 2009. Reactive noninterference. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security (CCS).*

[17] Roberto Capizzi, Antonio Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. 2008. Preventing Information Leaks through Shadow Executions. In *Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC).*

[18] Ethan Cecchetti, Andrew C. Myers, and Owen Arden. 2017. Nonmalleable Information Flow Control. In *Proceedings of the 2017 ACM Conference on Computer and Communications Security (CCS).*

[19] Stephen Chong and Andrew C. Myers. 2006. Decentralized Robustness. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW).*

[20] Stephen Chong, K. Vikram, and Andrew C. Myers. 2007. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of the 16th USENIX Security Symposium (USENIX).*

[21] Andrey Chudnov and David A. Naumann. 2010. Information Flow Monitor Inlining. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium (CSF).*

[22] Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. 2009. Cross-tier, Label-based Security Enforcement for Web Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD).*

[23] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS).*

[24] Dominique Devriese and Frank Piessens. 2010. Noninterference through Secure Multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP).*

[25] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2010. An Empirical Study of Privacy-violating Information Flows in JavaScript Web Applications. In *Proceedings of the 2010 ACM Conference on Computer and Communications Security (CCS).*

[26] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. 2013. Run-Time Enforcement of Information-Flow Properties on Android. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS).*

[27] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information flow control for standard OS abstractions. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP).*

[28] McKenna McCall, Abhishek Bichhawat, and Limin Jia. 2022. Compositional Information Flow Monitoring for Reactive Programs. In *Proceedings of the 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P).*

[29] McKenna McCall, Abhishek Bichhawat, and Limin Jia. 2023. Tainted Secure Multi-Execution to Restrict Attacker Influence. https://doi.org/10.1184/R1/22296628.v1 Technical Report.

[30] McKenna McCall, Hengruo Zhang, and Limin Jia. 2018. Knowledge-based Security of Dynamic Secrets for Reactive Programs. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF).*

[31] Scott Moore and Stephen Chong. 2011. Static Analysis for Efficient Hybrid Information-Flow Control. In *Proceedings of the 24TH IEEE Computer Security Foundations Symposium (CSF).*

[32] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. 2006. Enforcing Robust Declassification and Qualified Robustness. *Journal of Computer Security (JCS)* 14, 2 (2006).

[33] Minh Ngo, Nataliia Bielova, Cormac Flanagan, Tamara Rezk, Alejandro Russo, and Thomas Schmitz. 2018. A Better Facet of Dynamic Information Flow Control. In *Proceedings of The Web Conference (WWW).*

[34] Vineet Rajani, Abhishek Bichhawat, Deepak Garg, and Christian Hammer. 2015. Information flow control for event handling and the DOM in web browsers. In *Proceedings of the 2015 IEEE Computer Security Foundations Symposium (CSF).*

[35] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium (CSF).*

[36] Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. 2009. Tracking Information Flow in Dynamic Tree Structures. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS).*

[37] Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. 2018. Faceted Secure Multi Execution. In *Proceedings of the 2018 ACM Conference on Computer and Communications Security (CCS).*

[38] Daniel Schoepe, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. 2016. Explicit Secrecy: A Policy for Taint Tracking. In *Proceedings of the 2016 IEEE 1st European Symposium on Security and Privacy (EuroS&P).*

[39] Steven Sprecher, Christoph Kerschbaumer, and Engin Kirda. 2022. SoK: All or Nothing - A Postmortem of Solutions to the Third-Party Script Inclusion Permission Model and a Path Forward. In *Proceedings of the 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P).*

[40] Deian Stefan, Edward Z. Yang, Brad Karp, Petr Marchenko, Alejandro Russo, and David Mazières. 2014. Protecting Users by Confining JavaScript with COWL. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI).*

[41] Mathy Vanhoef, Willem De Groef, Dominique Devriese, Frank Piessens, and Tamara Rezk. 2014. Stateful declassification policies for event-driven programs. In *Proceedings of the 2014 IEEE Computer Security Foundations Symposium (CSF).*

[42] MDN web docs. 2023. Event.isTrusted. https://developer.mozilla.org/en-US/docs/Web/API/Event/isTrusted [Online; accessed 9-January-2023].

[43] Steve Zdancewic and Andrew C Myers. 2001. Robust Declassification.. In *Proceedings of Computer Security Foundations Workshop (CSFW).*

[44] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI).*