2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)

Compositional Information Flow Monitoring for Reactive Programs

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

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

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

2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-1614-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/EUROSP53844.2022.00036

***Abstract*—To prevent applications from leaking users' private data to attackers, researchers have developed runtime information flow control (IFC) mechanisms. Most existing approaches are either based on taint tracking or multi-execution, and the same technique is used to protect the entire application. However, today's applications are typically composed of multiple components from heterogenous and unequally trusted sources. The goal of this paper is to develop a framework to enable the flexible composition of IFC enforcement mechanisms. More concretely, we focus on reactive programs, which is an abstract model for event-driven programs including web and mobile applications. We formalize the semantics of existing IFC enforcement mechanisms with well-defined interfaces for composition, define knowledge-based security guarantees that can precisely quantify the effect of implicit leaks from taint tracking, and prove sound all composed systems that we instantiate the framework with. We identify requirements for future enforcement mechanisms to be securely composed in our framework. Finally, we implement a prototype in OCaml and compare the effects of different compositions.**

1. Introduction

Applications such as web and mobile phone apps collect a huge amount of user data. Such *event-driven* applications are typically modeled as reactive programs [24], where a program is a set of event handlers, triggered by corresponding user input events. Shared storage allows the same data to be accessed by different event handlers (e.g., cookies) and organizes the event handlers themselves (e.g., the DOM). These applications often include code from heterogeneous and untrusted sources and could potentially leak the users' sensitive data to an adversary. To prevent such leaks, many runtime mechanisms have been developed for enforcing information flow control (IFC) policies [15]–[17], [20], [30], [33], [39], [69], most of which guarantee (variants of) *noninterference*, i.e., private data should not influence data sent on channels that are publicly observable [34]. Broadly, these approaches can be classified into *multi-execution approaches* [10], [11], [31], and *taint tracking approaches* [8], [9], [66], [71].

Multi-execution-based approaches, like secure multi-execution (SME) [31], and faceted execution [10], execute code multiple times at different security levels. These ensure that the code executing at a particular level only outputs data at the same security level, and replace sensitive data from higher security levels with "default" values. *Taint tracking approaches* annotate data with *labels* to indicate its security level, and can suppress outgoing sensitive data to publicly observable channels to prevent leaks. These approaches differ in performance, how much they alter the semantics of safe programs (transparency), and the relative strength of their security guarantees.

Most of these approaches use the same enforcement mechanism for all components in an application. Given the heterogeneity of applications, a compositional enforcement mechanism where different components execute under different IFC enforcement mechanisms could offer an attractive solution to the tradeoffs of each approach.

In this work, we motivate the usefulness of composition (Section 3), build a *framework* for composing different IFC enforcement mechanisms, and explore which security properties can be proven for which compositions. One of the challenges is to build a unified framework so that different styles of enforcement (taint tracking-based and multi-execution-based) can interact smoothly and two distinct elements of reactive systems (event handlers and shared storage) can interface nicely. To do so, our formalism identifies the common elements between all techniques, as well as the interfaces between the event handler execution component and the shared storage components, and converts values between mechanisms securely.

As we show in Section 4, the compositional semantics are cleanly separated into a top-level component to trigger event handlers and a low-level component that executes individual event handlers and interacts with shared storage.

To reason about security guarantees and their relative strengths, we define, in Section 5, security properties based on *attacker knowledge*, which is the set of all possible (secret) inputs that the attacker believes could have produced the public outputs observed by the attacker [5]. As the attacker makes more observations, they become more certain about the possible secret inputs. We also define a weaker security condition which extends prior work on weak and explicit secrecy [66], [71] that permits the attacker to additionally learn what is implicitly leaked by taint tracking. We propose a set of security requirements that describe what is required by each system component and could be used to securely instantiate our framework with additional enforcement mechanisms in the future.

We implement the enforcement mechanisms in OCaml (Section 6) to validate the model and show an empirical comparison of the different compositions.

Contributions. We develop a framework to enable the flexible composition of dynamic IFC enforcement mech-

DOI 10.1109/EuroSP53844.2022.00036

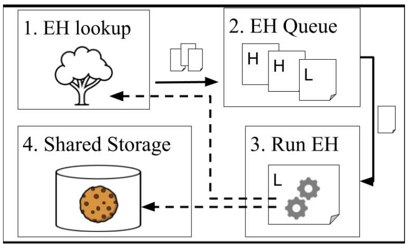


Figure 1: High-level depiction of a reactive system

anisms for reactive programs with provable security guarantees and model a simple web environment. We use a knowledge-based security condition to compare the relative security of different compositions. We extend prior work on weak secrecy to reason about implicit flows of information due to control flow decisions within as well as between event handlers and show that the overall security of a composed system may depend more on the security of the data structures shared between event handlers than the security of the event handler execution. Finally, we implement the framework in OCaml[1] to validate our model and to compare the tradeoffs of different compositions. Detailed definitions, lemmas, and proofs can be found in our companion technical report [48].

2. Background

IFC monitors for reactive programs typically associate security labels with input events, data, output channels, and event handlers to reflect the security policy. Labels are elements of a security lattice. In what follows, we consider the two-point security lattice $L \sqsubseteq H$ meaning that information labeled L (public) is allowed to flow to H (secret), but not the other way. We briefly review noninterference and weak secrecy, standard runtime IFC mechanisms, and knowledge-based security.

2.1. Reactive programming

A simple reactive system is shown in Figure 1. First, a user input triggers an event (1) which causes corresponding event handlers (EH) to run. Event handlers may be stored in a tree (such as the DOM on a webpage) or a simpler data structure, like an unordered list. Event handlers wait in a queue (2) to be run. The runtime manages a single-threaded event loop to run all of the event handlers in the queue. The runtime also keeps track of the system state which is *producer* (while an event handler is running) or *consumer* (when an event handler finishes). While an event handler is running (3), it may trigger new events or register new event handlers by interacting with the event handler storage. An event handler may also interact with other types of storage (like cookies or bookmarks) that persist after the event handler finishes (4). A new event is processed when all event handlers have finished running.

2.2. Noninterference and Weak secrecy

Noninterference [34] ensures that secret inputs do not affect public outputs. It prohibits *explicit* leaks, like output (L, h) where h contains secret data and L is a public channel, as well as *implicit* leaks via control flow like the following, where h is secret and l is public:

if $h = 0$ then $l := 1$ else $l := 0; output(L, l)$

A weaker form of noninterference that allows implicit leaks is *weak secrecy* [71] or *explicit secrecy* [66]. Weak secrecy only allows information leaks through branch predicates. The program above satisfies weak secrecy, as it can be re-written as a secure program without "high" branches: $l := 1; output(L, l)$ and $l := 0; output(L, l)$. Because both of these programs are secure, the original program satisfies weak secrecy.

2.3. Standard IFC Enforcement Mechanisms

We illustrate different enforcement mechanisms developed to enforce noninterference for reactive systems via an example event handler:

onKeyPress(k) { if $k == 42$ then $l := k$ }

Assume that initially $l = 0$. The event handler runs for keypress events and receives as a parameter the key pressed. The keypress events are public, while the parameter k, is secret. This means an attacker is allowed to learn that a key was pressed but not *which* key.

Multi-execution based approaches. These include *secure multi-execution* [31] and *faceted execution* [10].

Secure multi-execution (SME) executes the example program twice, once for each security level L and H, maintaining separate memory stores for each copy of the execution. Each execution receives only the data that is "visible" at its level, otherwise it is replaced with a *default* value. Thus, when a key is pressed, the execution at the H level reads the value of the key press k (the H input), and assigns 42 to the H copy of l if $k \mapsto 42$. In the L execution, the H input k is replaced with the default value, say 0, so the branch is never taken. Thus, at the end of the L execution, l remains unchanged in the L store, irrespective of the value of k.

Faceted execution (MF) simulates multiple executions while avoiding unnecessary re-execution by creating *facets* of a value only when the value contains a secret, e.g., $v = \langle v_h | v_l \rangle$ where v_h is the value of v observable at H and v_l is the value of v observable at L[2]. In the above example, $l \mapsto 0$ initially and is observable at all levels. If the secret input k is 42, l is assigned the faceted value $\langle 42 | 0 \rangle$, meaning H observers see 42, while L observers see l's original value.

Taint tracking approaches. *Taint tracking* (TT) approaches carry and propagate taint, or security labels, along with the data. In the example, suppose $l \mapsto 0^L$ initially, where L is the label of l. If $k \mapsto 42^H$, then $l \mapsto 42^H$ at the end. Otherwise, the branch is not taken, and l remains 0^L. Since the value of l depends on the branch condition, the branch condition is leaked *implicitly* through l. To securely handle such implicit leaks, some approaches maintain a program context (pc) label which keeps track of the context of the control flow decisions. In the example above, the context is H when assigning to l because of the branch predicate.

[1] The code repository is available at https://github.com/CompIFC/comp-model/releases/tag/eurosp22.

[2] The original faceted values [10] have the format $\langle k \,?\, v : v' \rangle$, where those that can read principal k's private data see v and others see v'.

```
1  onClick( ) {
2    if (strength > 5) {
3       p = pwdNode.value;
4       u = unameNode.value;
5       output (H, u+p); }};
```

Listing 1: Event handler to send username and password to host server

```
1  onInput(e) {
2    p = e.target.value; /* get password value */
3    if (p.match(/[0−9]/)) { strength+=1; }
4    if (p.match(/[A−Z]/)) { strength+=4; }
5    ...
6    output (L, p); }; /* Explicit leak */
```

Listing 2: Third-party event handler to check password strength; “strength” is a global variable

Then, these approaches abort the execution or diverge when assigning to public variables in secret contexts [8]. These approaches are called *no-sensitive-upgrade* (NSU) and satisfy termination-insensitive noninterference. Some other approaches satisfy weak secrecy by allowing implicit leaks and blocking only explicit leaks that output secret information to public channels [66], [71]. Here, we only consider approaches that do not abort or diverge.

3. Motivating Example

We demonstrate the usefulness of composing enforcement mechanisms via a web example in which event handlers run under different enforcement mechanisms.

Consider a website with a sign-up form including username and password fields and a submit button. There is also a third-party password strength-checking script which registers an event handler to the password field for the onInput event. The event handler is triggered whenever the user changes the password. It checks the password strength based on some algorithm (e.g., count the character classes of the password) and writes a numeric representation of the strength to a global variable *strength* (illustrated in Listing 2). The main page registers (among others) an event handler for the onClick event associated with the submit button (as shown in Listing 1). This event handler reads the global variable *strength*, and either allows the form submission (if the strength reaches a certain threshold) or displays a pop-up suggesting adding character classes, such as numbers and symbols.

The third-party script should compute the strength of the password locally without sending it on the network. A malicious script might try to send the password to their servers (line 6). The output command models sending a message to the third-party site. Let us see how taint tracking and multi-execution would enforce IFC in this scenario, and why composing them might be desirable. We will use this as a running example in the paper to describe our framework and later as a case study for evaluation.

Taint-tracking enforcement. Suppose we execute the event handlers with a taint tracking enforcement mechanism. NSU would terminate the execution of the entire page if any script attempts to assign to a public variable in a secret branch. This effectively opens up all pages to denial of service attacks, so we do not use NSU here (more discussion can be found in Section 7). Let’s consider naive taint tracking [33] without NSU, instead.

```
1  onInputLeak(e) {
2    p1 = e.target.value.charAt[0];
3    present_a = true; /* labeled L */
4    detected_a = false; /* labeled L */
5    if (p1 = 'a') {
6      detected_a = true;} /* tainted H if p1 is 'a' */
7    if (!detected_a) {
8      present_a = false;} /* still labeled L if p1 is 'a' */
9    output (L, present_a) };
```

Listing 3: Malicious third-party event handler

If the thid-party checker tries to directly leak the password on line 6 in Listing 2, the output will be suppressed because the output requires the value’s label to be lower than or equal to that of the channel, which does not hold.

A well-known limitation of naive taint tracking without NSU is that it allows the script to leak information via implicit flows [8]. Listing 3 is adapted from a classic example of implicit leaks. Here, the variable $detected\_a$ is only tainted if the first character is ‘a’. In this case, the assignment on line 8 is not executed as the branch is not taken. As a result, $present\_a$ remains true (and labeled L). On the other hand, if the first character is not ‘a’, the assignment on line 6 will not be taken. Then, the condition on line 8 will branch on an L value and therefore, $present\_a$ remains L and the value updated to false. Finally, the output on line 9 will successfully notify the attacker whether the first character is ‘a’. We can expand the program to test the password character-by-character for every ASCII symbol and thus leak the entire password [6]. Thus, taint tracking has weaker security guarantees, which we later formalize using *weak secrecy* [66], [71], that allows attackers to learn which H branches are taken and which L variables are upgraded to H. Because we allow branch conditions to be leaked, anyway, we simplify our semantics by not upgrading the pc when branching on secrets.

Multi-execution enforcement. To prevent the above-mentioned leaks, we can instead execute the event handlers using a multi-execution mechanism like SME [31]. The event handlers would then execute twice: once for the secret and once for the public level, where the secret execution would allow only H outputs while the public execution would allow only L outputs. The secret execution would see the actual value of the password but the public execution would get a default value instead. If the script sends the password on an L channel (line 6 in Listing 2), the public execution would send the default value instead of the actual password, while the secret execution would skip the output altogether. This also prevents the implicit leaks shown in Listing 3. Although SME securely computes accurate information, it runs the event handlers multiple times and stores multiple copies of data, which is resource intensive.

Composing taint-tracking and multi-execution. In this example, a desirable approach would be to execute the third-party script and store the global variable *strength* using a multi-execution approach so that it can correctly

Execution contexts:

$$\begin{array}{llll}
\textit{Sec. label set:} & \mathcal{L} & ::= & \{\cdot, L, H\} \\
\textit{Program counter:} & pc & \in & \mathcal{L} \\
\textit{Security label:} & l & \in & \mathcal{L} \\
\textit{Policy context:} & \mathcal{P} & & \\
\textit{Global storage enf.:} & G & & \\
\textit{EH enforcement:} & \mathcal{V} & &
\end{array}$$

Program syntax:

$$\begin{array}{llll}
\textit{Value:} & v & ::= & n \mid b \mid \mathsf{dv} \\
\textit{Expression:} & e & ::= & x \mid v \mid \mathsf{uop}\, e \mid e_1 \,\mathsf{bop}\, e_2 \mid \mathsf{ehAPIe}(\ldots) \\
\textit{Command:} & c & ::= & \mathsf{skip} \mid c_1; c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c\ \mid x := e \\
 & & \mid & id := e \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \\
 & & \mid & \mathsf{output}\ ch\ e \mid \mathsf{ehAPIc}(\ldots)
\end{array}$$

Runtime configurations:

$$\begin{array}{llll}
\textit{Global state:} & \sigma^G & & \\
\textit{Local state:} & \sigma^{\mathcal{V}} & & \\
\textit{Execution state:} & s & ::= & P \mid C \\
\textit{Events:} & E & ::= & \cdot \mid E, (id.Ev(v), l) \\
\textit{Configuration:} & \kappa^{\mathcal{V}} & ::= & \sigma^{\mathcal{V}}, c, s, E \\
\textit{Config. stack:} & ks & ::= & \cdot \mid (\mathcal{V}; \kappa^{\mathcal{V}}; pc) :: ks \\
\textit{Comp. config.:} & K^G & ::= & \sigma^G; ks \\
\textit{Actions:} & \alpha & ::= & \mathsf{in} \mid ch(v) \mid \bullet
\end{array}$$

Figure 2: Syntax for the compositional framework

$$\boxed{G, \mathcal{P} \vdash K \overset{\alpha}{\Longrightarrow} K'}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.Ev(v)) = H \\ G, \mathcal{P}, \sigma \vdash \mathsf{lookupEHAll}(id.ev(v)) \rightsquigarrow_H ks\end{array}}{G, \mathcal{P} \vdash \sigma; \cdot \overset{id.Ev(v)}{\Longrightarrow} \sigma; ks}\ \textsc{In-H}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.Ev(v)) = L \\ G, \mathcal{P}, \sigma \vdash \mathsf{lookupEHAll}(id.ev(v)) \rightsquigarrow_{\cdot} ks\end{array}}{G, \mathcal{P} \vdash \sigma; \cdot \overset{id.Ev(v)}{\Longrightarrow} \sigma; ks}\ \textsc{In-L}$$

(a) Simplified input rules

$$\frac{\begin{array}{c}\mathsf{producer}(\kappa) \qquad \alpha_l = \mathsf{out}_{\mathcal{V}}(\mathcal{P}, ch(v), pc) \\ G, \mathcal{P}, \mathcal{V} \vdash \sigma, \kappa \xrightarrow{ch(v)}_{pc} \sigma', ks'\end{array}}{G, \mathcal{P} \vdash \sigma; ((\mathcal{V}; \kappa; pc) :: ks) \overset{\alpha_l}{\Longrightarrow} \sigma', ks' :: ks}\ \textsc{Out}$$

$$\frac{\begin{array}{c}\mathsf{producer}(\kappa) \\ G, \mathcal{P}, \mathcal{V} \vdash \sigma, \kappa \xrightarrow{\alpha}_{pc} \sigma', ks' \qquad \bullet = \mathsf{out}_{\mathcal{V}}(\mathcal{P}, \alpha, pc)\end{array}}{G, \mathcal{P} \vdash \sigma; ((\mathcal{V}; \kappa; pc) :: ks) \overset{(\bullet, pc)}{\Longrightarrow} \sigma', ks' :: ks}\ \textsc{Out-Skip}$$

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

(b) Simplified output rules

Figure 3: Simplified semantics for processing inputs (user events) and performing outputs (communications on channels).

compute the strength of the password without compromising its secrecy. Meanwhile the event handlers on the main page could execute with a taint tracking mechanism as they do not purposely exploit implicit leaks and will be more performant than a multi-execution approach. In this particular example, for the main page event handlers to access precise information, the event handler will run in the H context to access the H copy of *strength*. Composition allows us to balance good security for the untrusted third-party scripts with good performance for the more trustworthy first-party scripts.

Another interesting composition question arising from this example is whether it is necessary to store shared variables such as *strength* twice as is typically done with SME and MF or is it sufficient to merely taint the variables and execute the script with SME? In this example, when onInput runs, the L copy runs first and sets the imprecise value for *strength* based on the default value for the password. The H copy runs next and sets the precise value for *strength* based on the real password with label H, as it is written from the H execution context. Is this secure? We take the first steps to explore different ways of storing data and executing scripts (Section 4), as well as what type of security each composition achieves (Section 5).

4. Compositional Enforcement Framework

One of our observations is that the semantics of reactive programs necessitate a high-level event handling loop that processes inputs and outputs, leading to the high-level semantics of dynamic IFC enforcement for these programs behaving similarly, regardless of the mechanism (e.g., SME or taint tracking). We design a framework that is flexible enough to incorporate all of the dynamic enforcement techniques described in Section 2. We describe the components from Figure 1, each of which has its own semantics (Section 4.2). The topmost level of semantics is responsible for processing inputs and outputs, and looking up event handlers. The next level manages the event handler queue, and another level describes how individual event handlers are run, according to the selected enforcement mechanism. Finally, the lowest level semantics are described in Section 4.3 determines how event handlers interact with shared storage (such as the DOM).

4.1. Syntax

The syntax for our compositional enforcement framework is shown in Figure 2. We organize our security labels, l, in a three-point security lattice which is the standard two-point security lattice with an additional label '$\cdot$'. At a high-level, $\cdot$ means "no (pc) context" and is neither public nor private, so we put it at the bottom of the security lattice. This is used by MF to differentiate a standard execution from one which has split into an L and H copy. The program context label indicates the context under which the event handlers execute, denoted as pc.

The policy context $\mathcal{P}$ keeps track of the labels assigned to input events and output channels, and is also responsible for deciding which event handlers run with which enforcement mechanism. For example, $\mathcal{P}$ might mark the onInput event for the password field as secret (H), output channels that belong to an attacker as public (L), and the enforcement of the onClick event handler to be TT and the third-party onInputCk event handler to be SME. We discuss considerations for making such decisions in Section 7. The enforcement for the global store is denoted G. The enforcement for a particular event handler is denoted $\mathcal{V}$.

Values include integers (n), booleans (b), and a predetermined default value dv, which is used to replace the public copy of private data in multi-execution [35]. Each value type can have a distinct default value; for simplicity we use a single default value. Commands and expressions are mostly standard in our framework. The event handler APIs ehAPIe (e.g., look up a DOM node's attribute) and ehAPIc (e.g., create a new child node in the DOM) interact with the event handler store, and $id := e$ updates the attributes in the event handler store.

A single configuration κ contains a local store for storing local script variables, $\sigma^{\mathcal{V}}$ (whose structure is determined by the enforcement mechanism $\mathcal{V}$), the command (event handler) being executed, the state of the execution (either Producer (P) or Consumer (C)), and a list of events triggered by that event handler, E. The compositional configuration K^G is a snapshot of the current system state. It maintains the global store σ^G and the configuration stack, ks. The global store inclues variables shared between scripts and event handler storage. The structure of the global store depends on the enforcement mechanism. Each element of the configuration stack includes one of the event handlers pending execution in κ, as well as the enforcement mechanism it should run under, $\mathcal{V}$, and the context in which it should run, pc. The enforcement ($\mathcal{V}$) used for each event handler in the stack is determined by $\mathcal{P}$ and may be different for different events. Actions emitted by the execution, α, include user-generated input events, outputs on channels and silent actions, denoted $\bullet$.

4.2. Framework Semantics

We organize our semantics into several layers to match the components illustrated in Figure 1.

Input/Output, EH Lookup. The top-most level for our compositional framework processes user input events and outputs to channels. These rules govern how inputs trigger event handlers and how outputs are processed and use the judgement $G, \mathcal{P} \vdash K \overset{\alpha}{\Longrightarrow} K'$, meaning the compositional configuration K can step to K' given input α or producing output α under the compositional enforcement G and label context $\mathcal{P}$. Simplified rules are shown in Figure 3.

Regardless of how the event handlers or global variables are stored, or how the policy determines to enforce IFC on individual event handlers, the logic for looking up event handlers is the same. In each case, the label context, $\mathcal{P}$, tells us whether the event is secret (H) or public (L). The EH lookup semantics, given by the judgement $G, \mathcal{P}, \sigma \vdash ks; \mathsf{lookupEH}(...) \rightsquigarrow_{pc} ks'$ return the stack of event handlers to run.

The label of an input event $id.Ev(v)$ is given by the policy $\mathcal{P}$. For secret events as in IN-H, all event handlers visible to H are run in the H context by using lookupEHAll with $pc = H$ to build ks. When the input is a public event as in IN-L, all event handlers are run in whatever context they are visible in by using the $\cdot$ pc for the lookup.

Similar to the input rules, the output rules shown in Figure 3b are the same regardless of the enforcement mechanism or event handler storage. The mid-level semantics are of the form: $G, \mathcal{P}, \mathcal{V} \vdash \sigma_1^G, \kappa \xrightarrow{\alpha}_{pc} \sigma_2^G, ks$ (omitted due to space constraints) and run a single event handler κ with the given enforcement mechanism $\mathcal{V}$ and produce some output α. producer(κ) and consumer(κ) tell us whether the execution state of κ is producer or consumer (respectively). When an event handler is currently running, the system is in producer state (OUT and OUT-SKIP) and when the event handler has finished, the system is in consumer state (OUT-NEXT) and the current event handler can be popped off ks. out$_{\mathcal{V}}(...)$ determines if an output should be allowed (OUT) or suppressed (OUT-SKIP) which is determined by whether the value being output is visible to the channel receiving the output and varies depending on the enforcement mechanism ($\mathcal{V}$).

EH Queue. The mid-level semantics control the execution state (P for Producer, when an event handler is running, and C for consumer, when it has finished) as well as adding event handlers for locally-triggered events (i.e., not triggered by a user) to the resulting configuration stack. After an event handler finishes running, these semantics check for any locally-triggered events. If there are some, their corresponding event handlers are added to ks. Finally, the current event handler enters consumer state to tell OUT-NEXT to run the next event handler.

Running EHs. The lower-level semantic rules for evaluating individual event handlers are triggered by the mid-level semantics in the "producer" state. These rules are mostly standard and enforcement-independent, except for interactions with the store. The rules in Figure 4 highlight the way our framework handles these differences. ASSIGN-G performs an assignment to a global variable while ASSIGN-D performs an assignment to an attribute in the event handler storage. Expressions are evaluated using the judgment $G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{pc} v$. This also ensures v is in the format expected by the enforcement when different mechanisms are composed. For instance, to convert a tainted value (v, H) to a value used by SME, we check that the label on the value is visible to the execution. The L execution would receive the default value dv instead of something tainted (v, H), while the H execution would receive the real value. This is reminiscent of the way SME replaces secret inputs with dv for the L execution. More discussion on conversion can be found in Section 4.3.

The assignment is performed using enforcement-specific helper functions. assign$_G(...)$ assigns global variables or event handler attributes, depending on whether a variable or node id is passed as an argument. The pc ensures that the assignments are performed securely (i.e., in the correct copy of the store, facet, or with the correct label, depending on the type of enforcement).

4.3. Shared storage

Event handlers may interact with each other through shared storage. To introduce the storage techniques, we describe the syntax for both variable and event handler storage (using the DOM as a case study) and describe their semantics at a high-level, then we explain how shared storage with one type of enforcement may be composed with an event handler running with a different type of enforcement. Finally, we illustrate these interactions by returning to our example from Section 3.

Variable storage syntax. We refer to shared storage techniques using similar terms as the enforcement mech-

$$\boxed{G, \mathcal{V} \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1 \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2^{\mathcal{V}}, c_2, E}$$

$$\frac{\begin{array}{c} G, \mathcal{V}, \sigma_1^G, \sigma_1^{\mathcal{V}} \vdash e \Downarrow_{pc} v \\ x \in \sigma_1^G \qquad \mathsf{assign}_G(\sigma_1^G, pc, x, v) = \sigma_2^G \end{array}}{G, \mathcal{V} \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, x := e \xrightarrow{\bullet}_{pc} \sigma_2^G, \sigma_1^{\mathcal{V}}, \mathsf{skip}, \cdot} \text{ ASSIGN-G}$$

$$\frac{\begin{array}{c} G, \mathcal{V}, \sigma_1^G, \sigma_1^{\mathcal{V}} \vdash e \Downarrow_{pc} v \\ x \notin \sigma_1^G \qquad \mathsf{assign}_{\mathcal{V}}(\sigma_1^{\mathcal{V}}, pc, x, v) = \sigma_2^{\mathcal{V}} \end{array}}{G, \mathcal{V} \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, x := e \xrightarrow{\bullet}_{pc} \sigma_1^G, \sigma_2^{\mathcal{V}}, \mathsf{skip}, \cdot} \text{ ASSIGN-L}$$

$$\frac{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{pc} v}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{output}\ ch\ e \xrightarrow{ch(v)}_{pc} \sigma^G, \sigma^{\mathcal{V}}, \mathsf{skip}, \cdot} \text{ OUTPUT}$$

Figure 4: Selected command semantics

Shared Storage

$$\textit{Shared storage: } \sigma^G \quad ::= \quad \sigma_g^G, \sigma_{EH}^G$$

$\mathsf{SME/SMS}$ **Variable Storage**

$$\begin{array}{lcll} \textit{Single store:} & \sigma_{pc} & ::= & \cdot \mid x \mapsto v \\ \mathsf{SME/SMS}\ \textit{Storage:} & \sigma^{\mathsf{SME}}, \sigma_g^{\mathsf{SMS}} & ::= & \sigma_H, \sigma_L \end{array}$$

$\mathsf{MF/FS}$ **Variable Storage**

$$\begin{array}{lcll} \textit{Faceted value:} & v^{\mathsf{MF}}, v^{\mathsf{FS}} & ::= & v \mid \langle v_H | v_L \rangle \mid \langle \cdot | v \rangle \mid \langle v | \cdot \rangle \\ \mathsf{MF}\ \textit{Storage:} & \sigma^{\mathsf{MF}} & ::= & \cdot \mid \sigma^{\mathsf{MF}}, x \mapsto v^{\mathsf{MF}} \\ \mathsf{FS}\ \textit{Storage:} & \sigma_g^{\mathsf{FS}} & ::= & \cdot \mid \sigma_g^{\mathsf{FS}}, x \mapsto v^{\mathsf{FS}} \end{array}$$

$\mathsf{TT/TS}$ **Variable Storage**

$$\begin{array}{lcll} \textit{Labeled value:} & v^{\mathsf{TT}}, v^{\mathsf{TS}} & ::= & (v, l) \\ \mathsf{TT}\ \textit{Storage:} & \sigma^{\mathsf{TT}} & ::= & \cdot \mid \sigma^{\mathsf{TT}}, x \mapsto v^{\mathsf{TT}} \\ \mathsf{TS}\ \textit{Storage:} & \sigma_g^{\mathsf{TS}} & ::= & \cdot \mid \sigma_g^{\mathsf{TS}}, x \mapsto v^{\mathsf{TS}} \end{array}$$

Figure 5: Storage syntax

EH Storage:

$$\textit{EH map: } M \quad ::= \quad \cdot \mid M, \mathit{Ev} \mapsto \{(eh_1, l_1), ..., (eh_n, l_n)\}$$

Unstructured SMS **DOM:**

$$\begin{array}{lcll} \textit{Single store:} & \sigma_{pc} & ::= & \cdot \mid id \mapsto (v, M) \\ \textit{DOM:} & \sigma_{EH}^{\mathsf{SMS}} & ::= & \sigma_H, \sigma_L \end{array}$$

Unstructured FS **DOM:**

$$\textit{DOM: } \sigma_{EH}^{\mathsf{FS}} \quad ::= \quad \cdot \mid \sigma^{\mathsf{FS}}, id \mapsto (v^{\mathsf{FS}}, M)$$

Unstructured TS **DOM:**

$$\textit{DOM: } \sigma_{EH}^{\mathsf{TS}} \quad ::= \quad \cdot \mid \sigma^{\mathsf{TS}}, id \mapsto (v^{\mathsf{TS}}, M, l)$$

DOM addresses:

$$\begin{array}{lcll} \textit{Location:} & loc & \in & \mathsf{Address} \\ \textit{Address:} & a & ::= & loc \mid \mathsf{NULL} \\ \textit{Root address:} & a^{\mathsf{rt}} & ::= & loc \\ \textit{Address list:} & A & ::= & \cdot \mid A, a \end{array}$$

Tree-structured SMS **DOM:**

$$\begin{array}{lcll} \textit{Node:} & \phi^{\mathsf{SMS}} & ::= & (id, v, M, a_p, A) \\ \textit{Single store:} & \sigma_{pc} & ::= & a^{\mathsf{rt}} \mapsto \phi^{\mathsf{SMS}} \mid \sigma_{pc}, loc \mapsto \phi^{\mathsf{SMS}} \\ \textit{DOM:} & \sigma_{EH}^{\mathsf{SMS}} & ::= & \sigma_H, \sigma_L \end{array}$$

Tree-structured FS **DOM:**

$$\begin{array}{lcll} \textit{Faceted address:} & a^{\mathsf{FS}} & ::= & a \mid \langle a_H | a_L \rangle \mid \langle \cdot | a \rangle \mid \langle a | \cdot \rangle \\ \textit{Faceted address list:} & A^{\mathsf{FS}} & ::= & \cdot \mid a^{\mathsf{FS}} :: A^{\mathsf{FS}} \\ \textit{Node:} & \phi^{\mathsf{FS}} & ::= & (id, v^{\mathsf{FS}}, M, a_p^{\mathsf{FS}}, A^{\mathsf{FS}}) \\ \textit{DOM:} & \sigma_{EH}^{\mathsf{FS}} & ::= & a^{\mathsf{rt}} \mapsto \phi^{\mathsf{FS}} \mid \sigma^{\mathsf{FS}}, loc \mapsto \phi^{\mathsf{FS}} \end{array}$$

Figure 6: Event handler storage syntax for the DOM

anisms for code execution: secure multi-storage, SMS, stores each item multiple times (once per security level), faceted storage, FS, stores multiple copies only when necessary, and tainted storage, TS, tracks labels for every item in the store. Storage syntax is shown in Figure 5. For SME/SMS, variables are stored twice: once at each security level. Observers at H will interact with the H copy of the store (σ_H) and observers at L with interact with the L copy of the store (σ_L). For MF/FS, variables are also stored twice, but only when the value depends on a secret. A faceted value such as $\langle v_H | v_L \rangle$ depends on a secret. H observers will interact with the H facet (v_H) and L observers interact with the L facet (v_L). Empty facets (such as the L facet of $\langle v | \cdot \rangle$) are treated as a default value. Finally, for TT/TS, values have an accompanying label to reflect whether they have been influenced by a secret (label H) or not (label L).

EH storage syntax. Event handler storage associates events with the appropriate event handlers. The DOM is one type of event handler storage, which links event handlers to elements on a webpage. We explain how to model event handler storage in our framework by considering both an unstructured DOM, where nodes are organized as an unordered list [49], which is useful for reactive systems like OS processes, as well as a more traditional tree-structure [62], which is useful for modeling the DOM. For brevity, we refer both the unstructured and tree-structured event handler storage as the "DOM." The syntax for both structures are shown in Figure 6.

In the unstructured DOM, elements are identified by a unique identifier (id) and contain both an attribute (whose structure is determined by the type of enforcement, to be described next) and an event handler map (M), which maps events (Ev) to a list of event handlers (eh) and the context they were registered in (l). M is the same for all enforcement mechanisms, except that event handlers in FS may have any label in $\mathcal{L}$ ("$\cdot$" means the event handler can be triggered by either L- or H-labeled events) but SMS and TS event handlers may only be labeled L or H.

Similar to variable storage, the unstructured [49] SMS DOM has two copies. H observers interact with the H copy of the DOM and likewise for L observers. Attributes are standard values (v), including integers and booleans. Initially, the H and L copies of the DOM will be identical. As events are triggered, new elements may be added to the DOM, event handlers registered, or attributes updated in one or both copies. The unstructured FS DOM is a single structure whose attributes are duplicated when they have been influenced by secrets. Here, attributes are standard when the value does not depend on a secret (v) or faceted values when the value appears different to H observers than L observers ($\langle v_H | v_L \rangle$). Initially, all the attributes are standard values in the FS DOM. A DOM element which has been added in only the H context will have an attribute with an empty L facet (i.e., $\langle v | \cdot \rangle$) and likewise for the H facet of an element added in only the L context. The TS DOM will associate labels with both attributes ((v, l)) and DOM elements ((v^{TS}, M, l)). The label on the element reflects the context the element was created in, while the label on the attribute reflects whether the attribute has been influenced by a secret ($l = H$) or not ($l = L$).

In the tree-structured [62] DOM, each element on the page has a matching DOM node (ϕ) which is stored by reference (loc). Nodes have a unique identifier (id), an attribute, and an event handler map, like in the unstructured DOM. They also contain a pointer to their parent (a_p), and a list of pointers to their children (A) (if any). The root of the DOM is at a^{rt}. The node at this address cannot be replaced with another node, but its attribute may be updated and children can be added to it. Since we later prove that compositions involving the unstructured TS DOM only satisfy weak secrecy, we only formalize the more complex tree-structured DOM for SMS and FS.

The tree-structured SMS DOM has two copies and behaves similarly to the unstructured SMS DOM. The tree-structured FS DOM supports faceted attributes, as well as a faceted parent pointer (a^{FS}) and list of faceted pointers to children (A^{FS}). Because nodes are uniquely identified by their ID, a node may have a faceted parent pointer, for instance, if a node is created as a child of ϕ_H in the H context and then a node with the same ID is created as a child of ϕ_L in the L context. A node might have a faceted pointer in its list of children if a child is added in the H context, but not the L context. In this case, if the child is at address a, the node would have $\langle a | \cdot \rangle$ in its list of children.

Storage composition. Since different event handlers running with different enforcement mechanisms may interact through shared storage, values may need to be "converted" from the format for one enforcement mechanism (i.e., a standard, faceted, or labeled value) to another. When converting data, we follow three high-level guidelines to ensure the composition is secure:

1. The pc context determines which copy to access in multi- storage. If a value is coming from SMS or FS, there may be two copies to pick from. When the context (i.e., the pc) is H, we access the H copy, and likewise for L. If the value does not exist in that copy of the store (in the case of SMS) or is an empty facet (in the case of FS), we use a default value.

2. The pc context and destination determines whether to replace a labeled value with a default value. If the value is coming from TS, we need to decide if we take the actual value or use a default value. If the context is H, we take the real value without leaking any information. If the context is L and the destination is a multi-storage (SMS, FS) or multi-execution (SME, MF) technique, we replace tainted values (with label H) with a default value since the L copy of the store/execution should never be influenced by a secret. On the other hand, if the destination is TS or TT, we use the original, tainted value, and propagate the taint through the resulting label.

3. The destination and pc context determines the ultimate format. Multi-storage and multi-execution techniques use the context to determine which copy of the store/which facet to update. For taint tracking techniques, the context is also used to determine the final label on the data (e.g., public data is labeled H if it is computed in the H context). Consider a public event handler running with SME. It would run first in the L context and then in the H context. The L execution would interact with the L copy of store secured with SMS, or with the L facets for a store secured with FS. The H execution would interact with the H copy (respectively, H facets). On the other hand, if the store is secured with TS, any changes made by the L execution would be labeled L and ultimately be overwritten by the H execution (which would have label H). A table summarizing how data is converted for every combination of enforcement is shown in Figure 7.

Examples. We describe how the example from Section 3 works in our framework, using the configuration in Figure 8. For illustrative purposes, we describe both SMS and TS shared storage with an unstructured DOM.

For TS storage, everything maps to a value and a label, including both variables and attributes and elements in the DOM. SMS involves an H and L copy of both the shared variables and DOM. The onInput event handler is public, so it exists in both the H copy of the SMS event handler storage and is labeled L in the TS storage. The contents of the field id_p are secret, so for SMS, the contents are replaced with a default value in the L copy of the DOM, and for TS the contents are labeled H. The onClick event is secret, so it is only registered in the H copy of the SMS DOM and is labeled H in the TS DOM. The policy is that onInput event handlers should be run under SME. We trust the first-party event handler onClick to not misbehave, so the policy is to run this event handler with TT.

The ks in Figure 8 is the result of looking up event handlers for the input event on the password field and the public click event on the "Submit" button. Note that ks will be the same whether we use SMS or TS for shared storage (more details on this to follow). For illustrative purposes, ks is the *result* of running all three event handlers. In reality, the local stores would initially be empty and the input event handlers would run to completion before the click event was triggered.

Rule IN-L is used to process the public Input event. It will run all of the registered event handlers in whatever context they are visible. Since the event handler is registered in both the L and H copies of the SMS DOM, and with label L in the TS DOM, it is visible to both the L and H context. Since we are running this event handler with SME, the ks has two onInput event handlers: one running in the L context and one in the H context (note that SMS and TS produce the same ks).[3] The onInput event handler attempts to output to an L channel. In the H execution, this output is suppressed (OUT-SKIP) because the output condition for SME requires that the label on the channel matches the label of the pc. On the other hand, the same output in the L execution would succeed (OUT). Recall that event handlers running in the L context interact with the L copy of the SMS DOM and receive default values instead of tainted values from the TS DOM. Therefore, this output does not leak anything to the attacker since the L copy of the execution receives a default value for the password from the DOM in both cases.

For the Click event, IN-H runs all of the event handlers visible to H (i.e. only those labeled H). This is the third element in ks (note that, like above, SMS and TS produce the same ks). When this event handler runs it will run in the H context, so it will interact with the H copy of the

[3] If the same event handler were to run under TT, we can output to both L and H channels from the L context, but only H channels from the H context. We only want to run the event handler once to avoid duplicated outputs to H channels, and we don't want to suppress all the L outputs, so we would run the event handler in the L context only.

		Destination and pc								
		SME, SMS			MF, FS			TT, TS		
		$\cdot$	L	H	$\cdot$	L	H	$\cdot$	L	H
Source	v^{std}	v^{std}	v^{std}	v^{std}	v^{std}	$\langle \cdot \| v^{\mathsf{std}} \rangle$	$\langle v^{\mathsf{std}} \| \cdot \rangle$	(v^{std}, L)	(v^{std}, L)	(v^{std}, H)
	$\langle v_H \| v_L \rangle$	$\langle v_H \| v_L \rangle$	v_L	v_H	$\langle v_H \| v_L \rangle$	v_L	v_H	$\langle v_H \| v_L \rangle$	(v_L, L)	(v_H, H)
	$\langle v \| \cdot \rangle$	$\langle v \| \cdot \rangle$	dv	v	$\langle v \| \cdot \rangle$	dv	v	$\langle v \| \cdot \rangle$	(dv, L)	(v, H)
	$\langle \cdot \| v \rangle$	$\langle \cdot \| v \rangle$	v	dv	$\langle \cdot \| v \rangle$	v	dv	$\langle \cdot \| v \rangle$	(v, L)	(dv, H)
	(v, L)	–	v	v	v	v	v	–	(v, L)	(v, H)
	(v, H)	–	dv	v	$\langle v \| \mathsf{dv} \rangle$	dv	v	–	(v, H)	(v, H)

Figure 7: Conversion between standard, tainted, and faceted values.

$$\begin{array}{lcl}
\mathsf{TS}\text{ Shared storage} \quad \sigma_g & = & \mathit{strength} \mapsto (40, H), \mathit{username} \mapsto (\text{“}\mathit{bob}\text{”}, L) \\
\sigma_{EH} & = & (id_p \mapsto ((\text{“}aKUd?mdu5GHa\&l7gHJ5\text{”}, H), \mathsf{input} \mapsto \{(\mathsf{onInput}(x)\{c_{in}\}, L)\}, L)), \\
 & & \quad (id_b \mapsto (\_, \mathsf{click} \mapsto \{(\mathsf{onClick}(\ )\{c_{clk}\}, L)\}, H)) \\
\mathsf{SMS}\text{ Shared storage} \quad \sigma_{g,L} & = & \mathit{strength} \mapsto \mathsf{dv}, \mathit{username} \mapsto \text{“}\mathit{bob}\text{”} \\
\sigma_{EH,L} & = & (id_p \mapsto\mapsto (\mathsf{dv}, \mathsf{input} \mapsto \{(\mathsf{onInput}(x)\{c_{in}\}, L)\})), (id_b \mapsto (\_, \cdot)) \\
\sigma_{g,H} & = & \mathit{strength} \mapsto 40, \mathit{username} \mapsto \text{“}\mathit{bob}\text{”} \\
\sigma_{EH,H} & = & (id_p \mapsto (\text{“}aKUd?mdu5GHa\&l7gHJ5\text{”}, \mathsf{input} \mapsto \{(\mathsf{onInput}(x)\{c_{in}\}, H)\}), \\
 & & \quad (id_b \mapsto (\_, \mathsf{click} \mapsto \{(\mathsf{onClick}(\ )\{c_{clk}\}, H)\})) \\
\text{Configuration stack} \quad ks & = & (\mathsf{SME}, ((\mathit{p1} \mapsto \mathsf{dv}[0], \mathit{present}_a \mapsto \mathsf{false}, \mathit{detected}_a \mapsto \mathsf{false}), [id_p/x]c_{in}, P, \cdot), L) \\
 & & \quad :: (\mathsf{SME}, ((\mathit{p1} \mapsto \text{“}a\text{”}, \mathit{present}_a \mapsto \mathsf{true}, \mathit{detected}_a \mapsto \mathsf{true}), [id_p/x]c_{in}, P, \cdot), H) \\
 & & \quad :: (\mathsf{TT}, ((p \mapsto (\text{“}aKUd?mdu5GHa\&l7gHJ5\text{”}, H), u \mapsto (\text{“}\mathit{bob}\text{”}, L)), c_{clk}, P, \cdot), H)
\end{array}$$

Figure 8: Example configuration

SMS σ_g and runs the risk of upgrading public variables in the TS σ_g. In this case, Listing 1 only reads from the shared storage, so nothing is leaked through TS. Recall from above that everything from the H copy of the SMS storage will be labeled H, and everything that comes from the TS storage will keep its label. An output for TT succeeds if the pc (H) joined with the label on the value being output ($p + u$, so, $H \sqcup H = H$ in the case of SMS or $H \sqcup L = H$ in the case of TS) is at or below the label on the channel (H). Therefore, this output succeeds.

This example shows that our framework can seamlessly compose enforcement mechanisms and securely convert data between different enforcement mechanisms, like SMS and TT.

5. Security and Weak Secrecy

Next we present two security definitions of different strengths, compare these two definitions, and prove that the techniques from Section 2 may be composed to enforce varying levels of security.

5.1. Attacker Observation

To quantify how much an attacker learns by interacting with our framework, we first define what the attacker can observe from an execution trace. A trace T is a sequence of execution steps, inductively defined as $T = G, \mathcal{P} \vdash T' \overset{\alpha_l}{\Longrightarrow} K$ where an empty trace is the initial state $G, \mathcal{P} \vdash K_0$. An attacker's observation of T, denoted $T \downarrow_L$, is the sequence of L-observable inputs and outputs in T. Two execution traces are L-equivalent if their L observations are the same: $T \approx_L T'$ iff $T \downarrow_L = T' \downarrow_L$. Key rules defining L observation of an execution trace are in Figure 9.

$$\frac{\mathcal{P}(id.Ev(v)) = L}{(G, \mathcal{P} \vdash K \overset{id.Ev(v)}{\Longrightarrow} T') \downarrow_L = id.Ev(v) :: T' \downarrow_L} \ \textsc{Trace-In-L}$$

$$\frac{\mathcal{P}(\alpha) = L \quad \mathsf{or} \quad l \sqsubseteq L}{(G, \mathcal{P} \vdash K \overset{(\alpha, l)}{\Longrightarrow} T') \downarrow_L = \alpha :: T' \downarrow_L} \ \textsc{Trace-L}$$

$$\frac{\mathcal{P}(id.ev(v)) = H}{(G, \mathcal{P} \vdash K \overset{id.ev(v)}{\Longrightarrow} T') \downarrow_L = T' \downarrow_L} \ \textsc{Trace-In-H}$$

$$\frac{\mathcal{P}(\alpha) = H \quad \mathsf{or} \quad \alpha = \bullet}{(G, \mathcal{P} \vdash K \overset{(\alpha, H)}{\Longrightarrow} T') \downarrow_L = T' \downarrow_L} \ \textsc{Trace-H}$$

Figure 9: Rules for projecting execution traces to L

Low inputs (Trace-In-L) and other low actions (Trace-L) are observable. Trace-L defines a "low action" as one produced in the low context ($l \sqsubseteq L$) or an L-labeled action ($\mathcal{P}(\alpha) = L$). L-labeled inputs and outputs to L-labeled channels are all L-labeled actions. Secret events, are not observable (Trace-In-H). Finally, secret actions ($\mathcal{P}(\alpha) = H$) performed in the H context are not observable as shown in Trace-H.

Two configurations K_1 and K_2 are L-equivalent if their global stores σ_1^G and σ_2^G and their configuration stacks ks_1 and ks_2 are L-equivalent. Configuration stacks are L-equivalent if all of the L configurations have L-equivalent local stores and they agree on commands. Most of these definitions are straightforward. The most interesting definition is L-equivalence of the tree-structured DOM, which is defined inductively over the structure of the tree beginning with the root nodes.

5.2. Progress-Insensitive Security

We first define attacker's knowledge assuming that the attacker can view all of the publicly-observable inputs and outputs, as well as the initial state of the system (this includes the initial global variables and DOM upon page load which contains no secrets). The attacker's knowledge given a trace T is *what they believe the secret inputs might have been*, which is the set of inputs from L-equivalent execution traces starting from the same initial state:

$$\mathcal{K}(T, \sigma_0^G, \mathcal{P}) = \{\tau_i \mid \exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{P}),\\ T \approx_L T' \wedge \tau_i = \mathsf{in}(T')\}$$

We define $\mathsf{runs}(...)$ as the set of possible execution traces resulting from the shared state σ_0^G under the policy $\mathcal{P}$. The set of inputs from a trace T is denoted $\mathsf{in}(T)$, while τ is a sequence of actions.

Intuitively, the system is secure if the attacker does not refine their knowledge. However, this definition is too strong for our system because it is progress-sensitive. An infinite loop that depends on a secret will allow the attacker to refine their knowledge based on whether the system makes progress to accept another low input. Instead, we define a weaker, progress-*in*sensitive security property, by introducing the following progress-insensitive attacker's knowledge below:

$$\mathcal{K}_p(T, \sigma_0^G, \mathcal{P}) = \{\tau_i \mid \exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{P}),\\ T \approx_L^{\mathcal{P}} T' \wedge \tau_i = \mathsf{in}(T') \wedge \mathsf{prog}(T')\}$$

The attacker is allowed to distinguish between traces which do and do not make progress so we add $\mathsf{prog}(T')$ as a condition on T' to consider only the traces which produce the same L-observations *and* make progress.

Using these knowledge definitions, we define what it means for a program to be secure: when the system takes a step, the attacker's *confidence* about the secret inputs should not increase; they should not be able to distinguish between any more traces than before, other than through whether the system makes progress. We use $\preceq$ subscript for the subset relation ($\supseteq_{\preceq}$) to say that the input sequences after the step may be longer.

Definition 1 (Progress-Insensitive Security). *The compositional framework is progress-insensitive secure iff given any initial global store σ_0^G and policy $\mathcal{P}$, it is the case that for all traces T, actions α, and configurations K s.t. $(T \xRightarrow{\alpha} K) \in \mathsf{runs}(\sigma_0^G, \mathcal{P})$, then $\mathcal{K}(T \xRightarrow{\alpha} K, \sigma_0^G, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_p(T, \sigma_0^G, \mathcal{P})$.*

We can prove that any combination of enforcement mechanisms SME, SMS, MF, and FS satisfy this progress-insensitive security condition:

Theorem 2 (Soundness). *If event handlers are enforced with $\mathcal{V} \in \{\mathsf{SME}, \mathsf{MF}\}$ and the global storage is enforced with $G \in \{\mathsf{SMS}, \mathsf{FS}\}$, then the composition of these event handlers and global stores in our framework satisfies progress-insensitive security.*

We prove our framework secure with these enforcement mechanisms by defining a series of "requirements" for the framework (called *Trace* and *Expression* requirements), variable stores (called *Variable* requirements), and event handler store (called *Event Handler* requirements). These requirements are described in Figure 10. For the most part, these requirements follow a similar structure to other knowledge-based security proofs from prior work. The most noteworthy difference is the notion of "strong equivalence" for values. Traditionally, noninterference only requires that values are equivalent (i.e., they are the same public values, *or* both values are secret) but here we require that values are both equivalent and publicly observable (i.e., they are equivalent only if they are the same public values; they cannot be tainted). This distinction is important for highlighting the difference between progress-insensitive security and weak secrecy.

5.3. Weak Secrecy

As discussed in Section 3, NSU semantics are too rigid for our setting. Unfortunately, without NSU semantics, taint tracking techniques are susceptible to implicit leaks. Namely, branching on a secret in the L context may result in different public behavior for different secrets. We can also see implicit leaks through global store: suppose a secret event handler *upgrades* a public value stored in the global variable x. If the attacker successfully output x in the past, but cannot output x now, they can conclude that a secret event handler which writes to x must have ran recently. For example, the leaky third-party script shown in Listing 3 violates Definition 1 when the script is enforced with TT and the global storage with TS. Consider the scenario where the user inputs a password "abcd". Before the output (true, L), the attacker knows the input was *some* password, but they are not sure which one, so their knowledge set is all possible passwords. After the output, the attacker learns that the input password must start with an 'a', thus refining the set of possible inputs to only the passwords beginning with 'a', which violates the security condition. Branching on a secret implicitly leaked information to the attacker.

Instead, we prove a weaker security condition called *weak secrecy* [66], [71] which allows *implicit* leaks through control flow but still ensures that *explicit* leaks via outputs are still prevented.

Additional attacker observations. We modify our semantics with additional outputs to capture both types of implicit leaks described above: $\mathsf{br}(\_)$ when branching on a tainted value in the L context, and $\mathsf{gw}(\_)$ when a L-labeled value is upgraded in the H context.

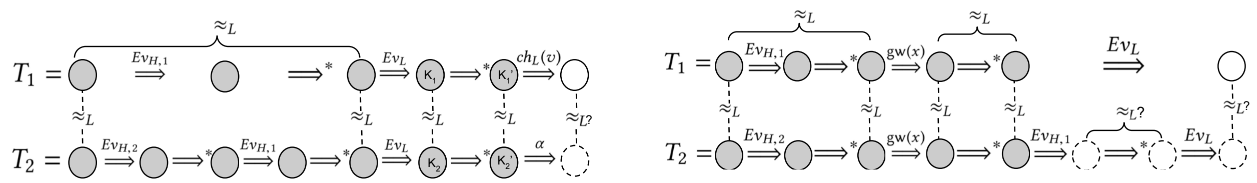
Knowledge-based weak secrecy definition. Since we allow information to leak through control flow decisions, we define another form of knowledge to capture this:

$$\mathcal{K}_{wp}(T, \Sigma_0, \mathcal{P}, \alpha_l, \mathcal{I}) = \{\tau_i \mid \exists T' \in \mathsf{runs}(\Sigma_0, \mathcal{P}, \mathcal{I}), T \approx_L T'\\ \wedge \tau_i = \mathsf{in}(T') \wedge \mathsf{prog}(T') \wedge \mathsf{wkTrace}(T', \alpha')\\ \text{where } \alpha' = (\mathsf{last}(T) \xRightarrow{\alpha_l} K)) \downarrow_L\}$$

$\mathsf{last}(T)$ returns the last configuration in a trace. Here, $\approx_L$ ensures the implicit leaks *up to this point* were the same and $\mathsf{wkTrace}$ ensures *the next* implicit leak is the same. If T is about to output $\mathsf{br}(b)$ or $\mathsf{gw}(x)$, then T' can be extended to produce the same output. We also need to make sure that when T receives a public input, T' does not leak anything until the next public input. Because inputs come nondeterministically, and we only want to consider traces which produce the same implicit leaks, we don't want T' to leak anything extra in a secret event handler before the next public input. This ensures that if T and T' were $\approx_L$ *up to this point*, they will continue to be equivalent *after the next step*. Maintaining equivalence like this is important for proving security.

Trace Requirements		
(W)T1	$\approx_L$ traces, $\approx_L$ states	Equivalent traces starting in equivalent states lead to equivalent states
(W)T2	Empty traces, $\approx_L$ states	Traces producing no public events produce equivalent states
T3	Secret pc's, empty traces	Steps under a secret pc produce no public events
(W)T4	Strong one-step	If a trace takes a step, then an equivalent trace can take an equivalent step
(W)T5	Weak one-step	Equivalent traces taking steps producing equivalent public observations lead to equivalent states
Expression Requirements		
(W)E1	L-expressions are $\approx_L$	Evaluating an expression under equivalent stores with public pc's results in *(strong)* equivalent values
Variable Requirements		
(W)V1	L-lookups are $\approx_L$	Lookups of the same variable under public pc's in equivalent stores result in *(strong)* equivalent values
(W)V2	H-assignments are $\approx_L$	Assignments to stores under a secret pc result in an equivalent store
(W)V3	L-assignments are $\approx_L$	Assignments to equivalent stores under public pc's result in equivalent stores
Event Handler Storage Requirements		
(W)EH1	L-lookups are $\approx_L$	Lookups in equivalent DOM's under public pc's result in *(strong)* equivalent values
(W)EH2	H EH lookups empty	Event handler lookups under a secret pc produce no public event handlers
EH3	H-updates are $\approx_L$	Updates under a secret pc results in an equivalent store
(W)EH4	L-updates are $\approx_L$	Updates under public pc's in equivalent stores result in equivalent stores

Figure 10: Requirements for Progress-Insensitive Security and Weak Secrecy



(a) Progress-insensitive Security: Each ●in the H context before Ev_L is L-equivalent, even though T_2 sees different H events than T_1. From $T_1 \approx_L T_2$, T_1 and T_2 see the same public input: Ev_L. We show that each step in the L context (K_1 to K_1' and K_2 to K_2') produces $\approx_L$ states and from this, we prove that T_2 can take step ● $\xRightarrow{\alpha}$ ◌ producing the same output $\alpha = ch(v)$ and equivalent states ○$\approx_L$ ◌.

(b) Weak Security: The proof is similar to above except that T_1 and T_2 are also synchronized on gw(_) and br(_) actions. Because of this, when T_1 takes a step to accept a low event ● $\xRightarrow{Ev_L}$ ○, we need to know that running the event handler for $Ev_{H,1}$ in T_2 (○$\Longrightarrow^*$ ○) will not produce any gw(_) actions. This is guaranteed by the wkTrace condition in $\mathcal{K}_{wp}()$.

Figure 11: Comparison of Progress-insensitive security (top) and Weak Security (bottom) proofs. Given $T_1 \approx_L T_2$, where T_1 takes a step to ○, we want to show that T_2 can take equivalent steps ◌, and that trace equivalence maintains state equivalence ●.

Consider, again, our leaky third-party script in Listing 3 where the user inputs the password "abcd". In our weak secrecy semantics, the execution of that event handler would generate br(true) when branching on the secret. The wkTrace predicate in the weak secrecy definition allows the attacker to refine their knowledge to include the fact that the branch condition must evaluate to true by throwing out all the traces which do not generate this branch condition. Only passwords starting with 'a' cause the branch condition to be true, so at this step, the attacker is allowed to learn that the password must begin with 'a' (i.e. the knowledge set is refined from all possible passwords to all possible passwords starting with 'a'). Therefore, the output does not further refine the attacker's knowledge, so this program satisfies weak secrecy.

Definition 3 (Progress-insensitive Weak Secrecy). *The compositional framework satisfies progress-insensitive weak secrecy in our framework iff given any initial global store, σ_0^G, and policy $\mathcal{P}$, it is the case that for all traces T, actions α, and configurations K s.t. $(T \xRightarrow{\alpha} K) \in$ runs$(\sigma_0^G, \mathcal{P})$, the following holds*

- *If* wkAction(last$(T) \xRightarrow{\alpha} K$)*:*
  $\mathcal{K}(T \xRightarrow{\alpha} K, \sigma_0^G, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{P}, \alpha)$
- *Otherwise:*
  $\mathcal{K}(T \xRightarrow{\alpha} K, \sigma_0^G, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_p(T, \sigma_0^G, \mathcal{P})$.

Meta-theory. We prove that any combination of enforcement mechanisms that we instantiated our framework with, including TT and TS, satisfy Definition 3:

Theorem 4 (Soundness-Weak Secrecy). *If event handlers are enforced with $\mathcal{V} \in$ {SME, MF, TT} and the global storage is enforced with $G \in$ {SMS, FS, TS}, then the composition of these event handlers and global stores in our framework satisfies progress-insensitive weak secrecy.*

We prove weak secrecy using a similar technique to progress-insensitive security. The requirements are nearly the same and are shown in Figure 10 with a **(W)**. Requirements **T3** and **EH3** cannot be proven in the presence of implicit leaks (upgrades to global variables in the H context is publicly observable). However, they are not needed to prove weak secrecy. The requirements mentioning "strong equivalence" are weakened to "equivalence" since leaking branch conditions is permitted.

Further comparisons of the proof techniques behind these two security definitions are shown in Figure 11. The events that two equivalent traces T_1 and T_2 have to agree on for the weak secrecy definition are a superset set of those required by the regular security definition, so the set of traces in the equivalent class (knowledge set) of the former is a subset of the latter. Consequently, attackers know more in the system that allows implicit leaks. We prove that our weak secrecy security condition is weaker than our standard security condition, in general:

Theorem 5 (PI Security implies PI Weak Secrecy)**.** *If the composition of event handlers and global storage enforcement are progress-insensitive secure, then they are also progress-insensitive weak secure.*

5.4. Securing TT

We can prove that in the presence of a secure global storage, using taint tracking for the event handler *is* secure, even without NSU semantics.

Theorem 6 (Soundness (TT))**.** *If event handlers are enforced with* $\mathcal{V} \in \{\mathsf{TT}, \mathsf{SME}, \mathsf{MF}\}$ *and the global storage is enforced with* $G \in \{\mathsf{SMS}, \mathsf{FS}\}$*, then the composition of these event handlers and global stores in our framework satisfies progress-insensitive security.*

The proof deviates from the requirements shown in Figure 10. We cannot prove the variable requirements for TT because looking up a tainted value violates requirement **V1**. However these requirements are stronger than necessary. The proof is intuitive: from the requirements, a secure global store will not allow a public event handler to access secrets, nor will it let secret event handlers modify public values. Recall that the local variable storage is cleared between event handlers, so there is no way for public event handlers to branch on secret values because the local storage will only contain public values. This means that **WV1** is sufficient to prove the stronger security condition and taint tracking techniques can be used securely, without NSU semantics, as long as the global structures satisfy strong security guarantees.

Going back to our example, the event handler in Listing 1 is secure, even though it is enforced with TT because it does not have implicit leaks. On the other hand, code with implicit leaks (Listing 2 and 3) can be secured by connecting the taint tracking script enforcement with a secure storage like SMS or FS, as shown by Theorem 6. This is noteworthy because it suggests that the selection of script enforcement is not as relevant to security as the selection of the global storage enforcement. Furthermore, the effects of TT are not manifested in this setting (since tainted variables never appear in the L context), meaning that as long as the shared structures are secure, the event handlers execution may require *no additional enforcement*.

6. Prototype Implementation and Evaluation

We discuss our prototype implementation of the compositional IFC monitoring framework and present evaluation results.

6.1. Prototype Implementation

We implemented our compositional semantics in OCaml 4.06.1 to validate the semantic rules and to study the results of composition. Our implementation consists of 2400 lines of OCaml code (including comments and blank lines) that is parametrized over the execution mechanism and the global store type. We optimized some of the semantics; for instance, instead of splitting the program when branching on a faceted value, our implementation only splits the execution *of the branch*, after which the program executes normally.

Limitations. Since the main goal of our prototype implementation is to understand and study the behavior of different compositions and to evaluate their security and performance, we do not aim to include all features of a browser (e.g., DOM APIs, cookies, localStorage, event handling logic), and restrict our implementation to the generic features modeled in Section 4. As future work, we would like to explore integrating our current model with Featherweight Firefox [23], an OCaml model of the browser, and study the behavior of the framework in a more realistic browser setting.

6.2. Evaluation Setup

We model web clients (specifically, a basic tree-structured DOM) by creating one root node with other nodes as children. For all experiments, we created 10 nodes in each of the stores with event handlers registered on some of these nodes. We emulate the execution of scripts on real-world websites by triggering events on some of these nodes and running the associated event handlers. To evaluate the performance of composing one script with varying enforcement mechanisms (Section 6.3.1), we install a keypress event handler on the password node that checks for the presence of certain characters in the entered password and returns the computed strength as the result (à la Listing 2). The program automatically inputs 100 character keypresses on the password node, which then runs the event handler producing some output depending on the mechanism used.

To evaluate the effects of composing different enforcement mechanisms for different scripts (Section 6.3.2), we implement a host page script, which installs an event handler that sends the password to the host server based on the strength of the password when the submit button is clicked (à la Listing 1). To emulate the behavior of multiple host scripts (and the possibility that more host scripts run than third-party scripts), we run more instances of the event handler installed by the host. The third-party script is the same as above.

For studying the security and accuracy of various enforcement mechanisms (Section 6.4), we implemented an analytics tracking script that tracks mouse clicks and keypresses by a user using the three event handlers shown in Figure 12. The aim is to implicitly leak the key pressed by the user through the global variable o.

6.3. Performance Evaluation

We compare the performance overhead of running a single event handler with different shared storage enforcement in Section 6.3.1. In Section 6.3.2, we compare the performance of composing multiple event handlers and show that the performance improves (while providing the same guarantees) when using a combination of enforcement mechanisms as opposed to a single enforcement mechanism for all event handlers.

Execution Mechanisms	Shared State		
	SMS	FS	TS
SME	12.69	13.24	12.51
MF	9.47	9.13	9.48
TT	7.72	7.79	7.35

TABLE 1: Time taken for different compositions for the example shown in Listing 2, measured in milliseconds with 100 characters as input, by the user

Host Script Exec. Mechanism	Third-party Script Exec. Mechanism		
	SME	MF	TT
SME	28.73	27.87	26.32
MF	24.63	23.77	23.12
TT	21.57	21.28	20.07

TABLE 2: Time taken for different script compositions for the examples shown in Listings 1 and 2, measured with secure multi-storage DOM

6.3.1. Composing shared storage and script enforcement. The execution times in milliseconds for all compositions are shown in Table 1. As expected, SME is less performant because it executes event handlers multiple times for publicly visible events. Our implementation does not parallelize the L and H executions; prior work has shown that parallelism helps SME's performance considerably [35]. MF's performance is better than SME as there are fewer commands that MF executes multiple times, while TT is the fastest of the three. MF spends extra time creating, projecting and removing facets in values, depending on the context of the execution.

We also measure the total shared memory usage using OCaml's garbage collector API; they are 168 kB for SMS, 166.4 kB for FS and 165 kB for TS. These results match our intuition. SMS stores multiple copies of data, so it uses the most memory, followed by FS, which needs to store additional data when facets are created, while TS requires additional memory to store labels.

6.3.2. Composing script enforcement. The time taken for different combinations of script enforcement mechanisms with the SMS DOM are shown in Table 2. Down each column and across each row from left to right, the execution time decreases, as the enforcement mechanisms run fewer copies of the code. Since the host page has more scripts than third-party scripts, the script enforcement mechanism of the host page is the dominating factor of the execution time. The time taken for TT as the host script's enforcement mechanism and SME as the third-party script's enforcement mechanism is considerably shorter when compared to SME being used for both. Further, under the assumption that host page scripts are to be trusted not to have malicious implicit leaks, the security offered by the composition is similar to the one where SME is used for all the scripts. More generally, it may suffice to run trusted scripts with TT enforcement and selected scripts from untrusted sources under SME and MF to prevent malicious implicit leaks.

6.4. Security and Accuracy Evaluation

We evaluate the effects of different compositions on security guarantees and accuracy where "accuracy" is comparing the behavior of the program with enforcement to the behavior without enforcement[4].

$$\begin{aligned}
&\mathsf{onKeyPress}(k): && \mathsf{if}\ k = 42\ \mathsf{then}\ l := k\\
&\mathsf{onClick}(c): && \mathsf{if}\ l = 42\ \mathsf{then}\ o := 1;\\
& && \mathsf{output}\ (L, o); \mathsf{output}\ (H, o)\\
&\mathsf{onMouseOver}(c): && \mathsf{output}(L, o)
\end{aligned}$$

Figure 12: Analytics script event handlers for comparing security and precision. The keypress event is secret while click and mouseover events are public.

Execution Mechanisms	Shared State		
	SMS	FS	TS
SME	Secure Accurate	Secure Accurate	Weak Secrecy Inaccurate
MF	Secure Accurate	Secure Accurate	Weak Secrecy Inaccurate
TT	Secure Inaccurate	Secure Inaccurate	Weak Secrecy Inaccurate

TABLE 3: Security and precision for different compositions for the example shown in Figure 12.

The execution traces of our case study shown in Figure 12 with different compositions of SME, MF and TT with each of the stores are shown in Figures 13, 14 and 15 in the Appendix. The time taken for different compositions are shown in Table 4 in the Appendix. The performance numbers are similar to the numbers obtained above. The security and accuracy between the various compositions vary as shown in Table 3. In the table, "Secure" indicates that the composition does not leak information through the execution while "Weak Secrecy" indicates that some information is leaked via implicit flows. Similarly, "Accurate" means the execution produced correct outputs as per the user expectation while "Inaccurate" means the output is not consistent with what the user might have expected.

Summarizing results from Section 5, both SME and MF guarantee progress-insensitive security with all stores except for TS. For TS, we can show only weak secrecy, as global upgrades may cause leaks. Interestingly TT guarantees noninterference with SMS and FS storage, and with lower overheads. This is because only public event handlers can output to L channels and the only way public event handlers can access a secret is through the global store. SMS and FS replace secrets with dv in the L context, so TT does not leak secrets, even implicitly.

Figures 13, 14 and 15 show that TT and TS affect the accuracy of the outputs of program execution. While SME and MF guarantee similar results in most scenarios, SME is sometimes more accurate [22], illustrated by the example below when x is secret and l is public:

$$l := 0; \mathsf{if}\ x = 1\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 2; \mathsf{output}(L, l)$$

While SME outputs either 1 or 2 (depending on the default value of x), which are the possible values of l after the branch, MF outputs 0 as the value of l to L channel because the public part of the facet is 0 irrespective of

[4] This is similar to "transparency" which says the behavior of non-leaky programs should not be altered by enforcement, except, here, some of the event handlers we evaluate do have leaks so we use the term "accuracy", instead.

which branch is taken. This makes SME apt for systems where accuracy is critical. If a small loss of accuracy is permissible, MF would be a more performant option.

7. Discussion

Declassification. The framework we have presented thus far does not allow practical scenarios where some secret information needs to be released to public channels (e.g., releasing the last four digits of a credit-card number). Luckily, *declassification* [64] of secret information can be incorporated into our framework without extensive modification. Based on prior work on stateful declassification for SME [25], [49], [70], we can lift the declassification components to the top-level of the framework so it applies uniformly to all enforcement mechanisms, similar to the way that we process inputs and outputs the same way for each mechanism.

To allow the attacker's knowledge to be refined due to declassification, we would also define *release knowledge*, similar to implicit knowledge, which distinguishes between traces which perform different declassifications. Definitions 1 and 3 would then be extended to include an additional case for released events (similar to the wkA condition for weak secrecy). Modified semantics and security conditions which account for declassification may be found in the Appendix.

Other reactive settings. Our framework can be applied to different reactive settings, such as web apps with a full DOM, OS processes [43], [75], mobile phone applications [33], [39], [56], and serverless computing [4]. We consider only a few dynamic enforcement mechanisms, but our framework could be easily extended to accommodate others. To add another event handler enforcement mechanism, the local storage and output conditions would need to be defined. Rules for interacting with the local storage and any other special rules (for instance, for switching executions in SME or branching on faceted values in MF) would also need to be added. For global variable storage, only the storage syntax and rules for accessing the store would be necessary. The event handler storage is by far the most involved, likely requiring both new structures and rules.

The other reactive systems mentioned above typically have less sophisticated storage than the DOM and more complex scheduling compared to JavaScript's single-threaded execution. We would need to modify the semantics to accommodate different schedulers and ensure they do not become a source of information leakage.

NSU semantics in reactive systems. Traditional taint tracking upgrades the pc when branching on a tainted value, whereas our semantics do not. We made this choice for two reasons: First, this choice is consistent with prior work on weak secrecy [66], [71]. Second, upgrading the pc on tainted branches adds complexity to the semantics, but still leaks information. *No sensitive upgrade* (NSU) semantics that halt the execution of the entire page are problematic, as discussed in Section 3. More flexible variants of NSU, like permissive upgrades [9], or terminating the execution of individual event handlers [61], or simply skipping problematic assignments, can be adapted to the reactive setting. Adapting these mechanisms for our framework is straightforward: low-level rules for commands need to be defined. Variants of NSU techniques may achieve a stronger security guarantee, but run the risk of altering the behavior of non-leaky programs if they prevent upgrades to variables which never affect outputs to public channels. The focus of our work is on the effect of composition on security and we leave the investigation of additional mechanisms to future work.

Compositional security. So far, we developed a compositional framework to combine multi-execution techniques (strong security guarantees) with taint tracking (weaker security guarantees). One question that remains is whether we can use a compositional definition and proof infrastructure of the form, "If A is secure and B is secure, then their composition is secure". This is challenging in our setting because the security of event handlers often depends on the security of the global store. Instead, we define compositional security based on the interfaces between event handlers and global storage in a rely-guarantee style using "requirements" on execution traces, variable storage, and event handler storage.

Selecting desired composition. Decisions about which enforcement mechanisms to use depend on the desired trade-offs between security, accuracy, and performance: the main factors considered for IFC. Our evaluation shows that different compositions can guarantee different security properties with varying overheads and accuracy. SMS and FS provide the same security guarantees but because SMS is complex and difficult to maintain in systems with multiple security levels, FS might be a better choice. More specifically, SME with FS could be used when accuracy is important, while MF with FS/SMS can be used to balance security, accuracy and performance. TT approaches are useful in systems where accuracy is not as important; TT could be composed with FS or SMS without incurring high runtime overheads or sacrificing security but could double the storage needed for shared structures.

8. Related Work

Information flow security has been explored extensively for reactive systems. Prior work in this area, to the best of our knowledge, has focused on the formalization and enforcement of either IFC in scripts, IFC in DOM or only one of the compositions described before. We discuss three classes of closely related work: formalization of information flow security properties in reactive settings, enforcement of IFC in reactive systems, and composition of security properties in (reactive) systems.

Austin and Flanagan proposed purely dynamic IFC for dynamically-typed languages based on TT [8], [9] and, later, using MF [10]. Subsequent work [11], [72] discusses its extension to applications where the policy is specified separately from the code. Ngo et al. [57] generalize MF to multi-level lattices. Stefan et al. [68] present a dynamic IFC approach for functional languages and propose the LIO monad. Secure multi-execution [31] is another approach to enforce IFC in dynamic and reactive systems at runtime. Our formalism uses these three approaches to protect event handler execution.

Bielova and Rezk [22] point out the similarities and differences between SME and MF enforcement, while Zanarini et al. [74] have extended SME to be more precise. Bohannon et al. [24] present a formalization of a reactive system and introduce several definitions of reactive noninterference, some of which we have used in our formalism. Ngo et al. [58] study a different runtime enforcement for reactive programs by treating the program as a black-box and monitoring only the input and output events. Recently, Algehed and Flanagan [1] proved the impossibility of building a transparent and efficient black-box runtime monitor. Our framework is quite different from theirs because we are in the reactive setting. Moreover, we handle declassification and do not treat the enforcement mechanisms as black-boxes.

Schmitz et al. [65] combine MF with SME to provide better guarantees and performance and develop a framework that is parametric and can provide MF, SME, or MF − SME enforcement based on whether a program may diverge to guarantee termination-sensitive noninterference. Later, Algehed et al. [2] presented an approach to optimize the data and performance overheads in the earlier technique by joining or shrinking facets whenever possible. Our work includes more diverse and general composition of different enforcement mechanisms with different shared states. It would be an interesting direction for future research to incorporate into our framework the option to switch enforcement mechanisms within an event handler's execution.

Declassification [64] in reactive systems is an interesting problem. Various approaches have been proposed for declassifying information in reactive systems that employ SME [25], [49], [60], [70] and TT with NSU [20]. Our declassification module in the Appendix uses some of these formalisms to release information about secret events to public scripts. While our current formalism does not account for integrity and robust declassification [27], [55], it is an interesting area of future work.

Our knowledge-based security definitions are based on the gradual release property [7], which ensures that the knowledge of the adversary stays unchanged outside of released events. Banerjee et al. [13] proposed a type system for enforcing knowledge-based declassification defined as conditioned gradual release. Askarov and Chong [5] further define progress knowledge to reason about initial configurations. Balliu [12] defines abstract knowledge-based security, and studies the relationship between knowledge and trace-based definitions.

Volpano [71] originally defined weak secrecy as a means to formalize data-dependent flows as opposed to the stronger property of noninterference. Schoepe et al. [66] generalize this property as a knowledge-based property, *explicit secrecy*, to adapt to different semantics used by different languages. Our definition of weak secrecy is a specific instance of the explicit secrecy and control-flow gradual release property for the imperative language, and is defined on traces of input and output events instead of the complete state. Later work by Schoepe et al. [67] improves the precision of TT by using faceted values in the memory, which is similar in flavor to the composition of TT with FS in this paper.

Many projects have developed IFC enforcement in JavaScript and browsers. Most of the prior work on IFC in JavaScript [10], [19], [28], [29], [32], [36]–[38], [40] use dynamic or hybrid enforcements because of its dynamic features. Several IFC approaches for browsers have been proposed that build on top of these mechanisms [17], [20], [30], [42], [44], [69] and allow information to be declassified. The focus of that work is on composing mechanisms and our composition includes TT and MF from those works.

Prior work has also developed IFC enforcement mechanisms in the DOM and event-handling logic of the browser [3], [21], [61], [62]. We reason about simpler DOMs with essential functionality but consider multiple enforcements that are reasoned about individually by prior work. We additionally show how they compose with different IFC enforcement mechanisms of script executions.

Composition of information flow properties has been studied in the setting of event-based systems [45], [50]. McCullough, further, defined the property of restrictiveness for security of systems [52] based on what a user can infer about sensitive data, which is composable. Zakinthinos and Lee [73] showed important results about the composition of generalized noninterference, which was earlier proven to be not fully compositional [51]. Mantel [45] designed the modular assembly kit for security properties (MAKS) framework for composing information flow properties to reason about complex properties. Compositional methods for proving information flow properties of concurrent programs have also been extensively studied [14], [26], [41], [46], [47], [53], [54], [63] given the complexity that concurrency introduces. Bauereiss et al. [18] verify the security of a distributed social media platform by composition. Rafnsson and Sabelfeld [59] explore the composition of PINI and progress-sensitive noninterference in the context of interactive programs. Similar to existing work, we explore the composition of information flow security properties across various types of mechanisms for event handlers and shared storage. Because event handling and accesses to shared storage are not symmetric, we stipulate requirements on each component, but cannot directly compose them as homogeneously defined secure components.

9. Conclusion

We develop a framework to enable the flexible composition of dynamic IFC enforcement mechanisms for reactive programs with provable security guarantees. We use a knowledge-based security condition to compare the relative security of different compositions. We extend weak secrecy to reason about implicit flows of information due to control flow decisions within as well as between event handlers. Finally, we implement the framework in OCaml to validate the semantic rules and show the tradeoffs of different compositions.

Acknowledgement

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

References

[1] M. Algehed and C. Flanagan. Transparent IFC enforcement: Possibility and (in)efficiency results. In *IEEE CSF*, 2020.

[2] M. Algehed, A. Russo, and C. Flanagan. Optimising faceted secure multi-execution. In *IEEE CSF*, 2019.

[3] A. Almeida-Matos, J. Fragoso Santos, and T. Rezk. An information flow monitor for a core of DOM. In *TGC*, 2014.

[4] K. Alpernas, C. Flanagan, S. Fouladi, L. Ryzhyk, M. Sagiv, T. Schmitz, and K. Winstein. Secure serverless computing using dynamic information flow control. In *ACM OOPSLA*, 2018.

[5] A. Askarov and S. Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *IEEE CSF*, 2012.

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

[7] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *IEEE SP*, 2007.

[8] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *ACM PLAS*, 2009.

[9] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *ACM PLAS*, 2010.

[10] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *ACM POPL*, 2012.

[11] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama. Faceted execution of policy-agnostic programs. In *ACM PLAS*, 2013.

[12] M. Balliu. A logic for information flow analysis of distributed programs. In *NordSec*, 2013.

[13] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *IEEE SP*, 2008.

[14] G. Barthe and L. P. Nieto. Formally verifying information flow type systems for concurrent and thread systems. In *ACM FMSE*, 2004.

[15] I. Bastys, M. Balliu, and A. Sabelfeld. If this then what? controlling flows in IoT apps. In *ACM CCS*, 2018.

[16] I. Bastys, F. Piessens, and A. Sabelfeld. Tracking information flow via delayed output. In *NordSec*, 2018.

[17] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian. Run-time monitoring and formal analysis of information flows in Chromium. In *NDSS*, 2015.

[18] T. Bauereiss, A. P. Gritti, A. Popescu, and F. Raimondi. CoSMeDis: A distributed social media platform with formally verified confidentiality guarantees. In *IEEE SP*, 2017.

[19] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in WebKit's JavaScript bytecode. In *POST*, 2014.

[20] A. Bichhawat, V. Rajani, J. Jain, D. Garg, and C. Hammer. WebPol: Fine-grained information flow policies for web browsers. In *ESORICS*, 2017.

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

[22] N. Bielova and T. Rezk. Spot the difference: Secure multi-execution and multiple facets. In *ESORICS*, 2016.

[23] A. Bohannon and B. C. Pierce. Featherweight Firefox: Formalizing the core of a web browser. In *USENIX WebApps*, 2010.

[24] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *ACM CCS*, 2009.

[25] I. Bolosteanu and D. Garg. Asymmetric secure multi-execution with declassification. In *POST*, 2016.

[26] A. Bossi, C. Piazza, and S. Rossi. Compositional information flow security for concurrent programs. *Journal of Computer Security*, 15(3), 2007.

[27] E. Cecchetti, A. Myers, and O. Arden. Nonmalleable information flow control. In *ACM CCS*, 2017.

[28] A. Chudnov and D. A. Naumann. Inlined information flow monitoring for JavaScript. In *ACM CCS*, 2015.

[29] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *ACM PLDI*, 2009.

[30] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *ACM CCS*, 2012.

[31] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *IEEE SP*, 2010.

[32] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *ACSAC*, 2009.

[33] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX OSDI*, 2010.

[34] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[35] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Secure multi-execution of web scripts: Theory and practice. *Journal of Computer Security*, 22(4), 2014.

[36] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for JavaScript and its APIs. *Journal of Computer Security*, 24(2), 2016.

[37] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *IEEE CSF*, 2012.

[38] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *ACM CCS*, 2010.

[39] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake. Run-time enforcement of information-flow properties on android (extended abstract). In *ESORICS*, 2013.

[40] S. Just, A. Cleary, B. Shirley, and C. Hammer. Information flow analysis for JavaScript. In *PLASTIC*, 2011.

[41] A. Karbyshev, K. Svendsen, A. Askarov, and L. Birkedal. Compositional non-interference for concurrent programs via separation and framing. In *POST*, 2018.

[42] C. Kerschbaumer, E. Hennigan, P. Larsen, S. Brunthaler, and M. Franz. Towards precise and efficient information flow control in web browsers. In *TRUST*, 2013.

[43] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *ACM SOSP*, 2007.

[44] Z. Li, K. Zhang, and X. Wang. Mash-IF: Practical information-flow control within client-side mashups. In *IEEE/IFIP DSN*, 2010.

[45] H. Mantel. On the composition of secure systems. In *IEEE SP*, 2002.

[46] H. Mantel and A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *Journal of Computer Security*, 11(4), 2003.

[47] H. Mantel, D. Sands, and H. Sudbrock. Assumptions and guarantees for compositional noninterference. In *CSF*, 2011.

[48] M. McCall, A. Bichhawat, and L. Jia. Compositional information flow monitoring for reactive programs. Technical report, Carnegie Mellon University, 2022.

[49] M. McCall, H. Zhang, and L. Jia. Knowledge-based security of dynamic secrets for reactive programs. In *2018 IEEE CSF*, 2018.

[50] D. McCullough. Specifications for multi-level security and a hook-up. In *IEEE SP*, 1987.

[51] D. McCullough. Noninterference and the composability of security properties. In *IEEE SP*, 1988.

[52] D. McCullough. A hookup theorem for multilevel security. *IEEE Transactions on Software Engineering*, 16(6), 1990.

[53] T. Murray, R. Sison, and K. Engelhardt. COVERN: A logic for compositional verification of information flow control. In *IEEE EuroSP*, 2018.

Execution Mechanisms	Shared State		
	SMS	FS	TS
SME	0.184 ms	0.185 ms	0.179 ms
MF	0.142 ms	0.135 ms	0.140 ms
TT	0.135 ms	0.139 ms	0.130 ms

TABLE 4: Time taken for different compositions for the example shown in Figure 12, measured in ms

[54] T. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *IEEE CSF*, 2016.

[55] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *IEEE CSFW*, 2004.

[56] A. Nadkarni, B. Andow, W. Enck, and S. Jha. Practical DIFC enforcement on android. In *USENIX Security*, 2016.

[57] M. Ngo, N. Bielova, C. Flanagan, T. Rezk, A. Russo, and T. Schmitz. A better facet of dynamic information flow control. In *WWW*, 2018.

[58] M. Ngo, F. Massacci, D. Milushev, and F. Piessens. Runtime enforcement of security policies on black box reactive programs. In *ACM POPL*, 2015.

[59] W. Rafnsson and A. Sabelfeld. Compositional information-flow security for interactive systems. In *IEEE CSF*, 2014.

[60] W. Rafnsson and A. Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. *Journal of Computer Security*, 24(1), 2016.

[61] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer. Information flow control for event handling and the DOM in web browsers. In *IEEE CSF*, 2015.

[62] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *ESORICS*, 2009.

[63] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *IEEE CSFW*, 2000.

[64] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5), 2009.

[65] T. Schmitz, M. Algehed, C. Flanagan, and A. Russo. Faceted secure multi execution. In *ACM CCS*, 2018.

[66] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld. Explicit secrecy: A policy for taint tracking. In *IEEE EuroSP*, 2016.

[67] D. Schoepe, M. Balliu, F. Piessens, and A. Sabelfeld. Let's face it: Faceted values for taint tracking. In *ESORICS*, 2016.

[68] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell*, 2011.

[69] D. Stefan, E. Z. Yang, B. Karp, P. Marchenko, A. Russo, and D. Mazières. Protecting users by confining JavaScript with COWL. In *USENIX OSDI*, 2014.

[70] M. Vanhoef, W. De Groef, D. Devriese, F. Piessens, and T. Rezk. Stateful declassification policies for event-driven programs. In *IEEE CSF*, 2014.

[71] D. M. Volpano. Safety versus secrecy. In *SAS*, 1999.

[72] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong. Precise, dynamic information flow for database-backed applications. In *ACM PLDI*, 2016.

[73] A. Zakinthinos and E. S. Lee. The composability of non-interference [system security]. In *IEEE CSFW*, 1995.

[74] D. Zanarini, M. Jaskelioff, and A. Russo. Precise enforcement of confidentiality for reactive systems. In *IEEE CSF*, 2013.

[75] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

Appendix A. Weak Secrecy Semantics

The weak secrecy semantics are mostly straightforward: we modify the existing semantics so that upgrades to L global variables or L attributes in L DOM nodes emit a $\mathsf{gw}(\_)$ action and branches in the L context result in $\mathsf{br}(\_)$ actions. Note that our trace equivalence definitions already account for these actions.

Appendix B. Composing Shared Storage

We can allow shared storage techniques to differ between shared variables and event handler storage. Instead of tracking a single enforcement, $G \in \{\mathsf{SMS}, \mathsf{TS}, \mathsf{FS}\}$, G is a pair $G = (\mathcal{G}_g, \mathcal{G}_{EH})$ where $\mathcal{G}_g$ is the shared variable enforcement mechanism and $\mathcal{G}_{EH}$ is the event handler storage enforcement mechanism with $\mathcal{G}_g, \mathcal{G}_{EH} \in \{\mathsf{SMS}, \mathsf{TS}, \mathsf{FS}\}$. The changes to the semantics to support this composition are straightforward. Values may be converted between mechanisms using the same conversion rules discussed in Section 4.3. The security guarantees follow from Section 5: compositions involving SMS and/or FS for either shared variable or event handler storage satisfy progress-insensitive noninterference, while those involving TS satisfy progress-insensitive weak secrecy.

Appendix C. Declassification

The additional syntax for the framework to support declassification is based on prior work [49], shown below:

$$\begin{array}{rcll} \textit{Release} & \mathcal{R} & ::= & (\rho, \mathcal{D}) \\ \textit{Released value} & r & ::= & \mathsf{none} \,|\, \mathsf{some}(\iota, v) \\ \textit{Release Channel} & d & ::= & \cdot \,|\, d, (\iota, v) \end{array}$$

We add a release channel $\mathcal{R}$ which maintains the state (ρ) and a declassification function ($\mathcal{D}$). $\mathcal{D}$ takes an input event and the current state, ρ, and returns a tuple containing the value to be released (r), a parameter to the event to be released (v or emp if the event is not released), and the new state. We also add a declassify command which reads from the declassification channel d, where ι is the identifier of the declassification (e.g., a unique location in the code). The values on d are updated to value r by $\mathsf{update}(d, r)$ and read by $\mathsf{read}(d, \iota)$.

The complete rules for processing inputs and outputs, with support for declassifying secret inputs, is shown in Figure 16. Note that these rules emit α_l which are labeled actions. The label tells us the context the action was generated in, which is helpful for proofs. Rule In-NR1 handles secret input events which are not declassified (i.e., no public event handlers will be triggered). $\mathcal{D}(\rho, id.Ev(v))$ returns (r, emp, ρ') which is the value to update on the declassification channel (using $\mathsf{update}(d, r)$), emp to indicate that the event should not be declassified to a public event handler, and the new state ρ'. Like In-H from Section 4.2, event handlers are looked up using $\mathsf{lookupEHAll}$ in the H context to return all of the event handlers visible from H, configured to run in the H context. In-NR1 is similar, except that it processes events associated with

Global Stores	SMS		FS		TS	
Initial State	$\sigma_L[l \mapsto 0; o \mapsto 2]$ $\sigma_H[l \mapsto 0; o \mapsto 2]$		$\sigma[l \mapsto 0; o \mapsto 2]$		$\sigma[l \mapsto 0^L; o \mapsto 2^L]$	
	keypress $k = 42$ and no release to L; click; mouseover					
	L	H	L	H	L	H

onKeyPress(k):
if $k = 42$ then $l := k$
		$\sigma_H[l \mapsto 42]$		$\sigma[l \mapsto \langle 42 \vert 0 \rangle]$		$\sigma[l \mapsto 42^H]$

onClick(c):
if $l = 42$ then $o := 1$;
	-	$\sigma_H[o \mapsto 1]$	-	$\sigma[o \mapsto \langle 1 \vert 2 \rangle]$	-	$\sigma[o \mapsto 1^H]$

output (L, o); output (H, o)	2; ●	●; 1	2; ●	●; 1	2; ●	●; 1
onMouseOver(): output (L, o)	2	●	2	●	dv	●
	keypress $k \neq 42$ and no release to L; click; mouseover					
Execution Context	L	H	L	H	L	H

onKeyPress(k):
if $k = 42$ then $l := k$
		-		-		-

onClick(c):
if $l = 42$ then $o := 1$;
	-	-	-	-	-	-

output (L, o); output (H, o)	2; ●	●; 2	2; ●	●; 2	2; ●	●; 2
onMouseOver(): output (L, o)	2	●	2	●	2	●

Figure 13: Execution of SME with different global shared states (Figure 12). L execution runs first; H execution second. The keypress events are secret while click and mouseover events are public. Default values are indicated as dv in the outputs while ● indicates that the output is suppressed. The L outputs and public events are shown in the blue color while H outputs and events are shown in red.

Global Stores	SMS	FS	TS
Initial State	$\sigma_L[l \mapsto 0; o \mapsto 2]$ $\sigma_H[l \mapsto 0; o \mapsto 2]$	$\sigma[l \mapsto 0; o \mapsto 2]$	$\sigma[l \mapsto 0^L; o \mapsto 2^L]$
	keypress $k = 42$ and no release to L; click; mouseover		

onKeyPress(k):
if $k = 42$ then $l := k$
	$\sigma_H[l \mapsto 42]$	$\sigma[l \mapsto \langle 42 \vert 0 \rangle]$	$\sigma[l \mapsto 42^H]$

onClick(c):
if $l = 42$ then $o := 1$;
	$\sigma_H[o \mapsto 1]$	$\sigma[o \mapsto \langle 1 \vert 2 \rangle]$	$\sigma[o \mapsto 1^H]$

output (L, o); output (H, o)	2; 1	2; 1	dv; 1
onMouseOver(): output (L, o)	2	2	dv
	keypress $k \neq 42$ and no release to L; click; mouseover		

onKeyPress(k):
if $k = 42$ then $l := k$
	-	-	-

onClick(c):
if $l = 42$ then $o := 1$;
	-	-	-

output (L, o); output (H, o)	2; 2	2; 2	2; 2
onMouseOver(): output (L, o)	2	2	2

Figure 14: Execution of MF with global shared states and events (Figure 12). The keypress events are secret while click and mouseover events are public. Default values are indicated as dv in the outputs. The L outputs are shown in the blue color while H outputs are shown in red.

Global Stores	SMS	FS	TS
Initial State	$\sigma_L[l \mapsto 0; o \mapsto 2]$ $\sigma_H[l \mapsto 0; o \mapsto 2]$	$\sigma[l \mapsto 0; o \mapsto 2]$	$\sigma[l \mapsto 0^L; o \mapsto 2^L]$
	keypress $k = 42$ and no release to L; click; mouseover		

onKeyPress(k):
if $k = 42$ then $l := k$
	$\sigma_H[l \mapsto 42]$	$\sigma[l \mapsto \langle 42 \vert 0 \rangle]$	$\sigma[l \mapsto 42^H]$

onClick(c):
if $l = 42$ then $o := 1$;
			$\sigma[o \mapsto 1^L]$

output (L, o); output (H, o)	2; 2	2; 2	1; 1
onMouseOver(): output (L, o)	2	2	1
	keypress $k \neq 42$ and no release to L; click; mouseover		

onKeyPress(k):
if $k = 42$ then $l := k$
	-	-	-

onClick(c):
if $l = 42$ then $o := 1$;
	-	-	-

output (L, o); output (H, o)	2; 2	2; 2	2; 2
onMouseOver(): output (L, o)	2	2	2

Figure 15: Execution of TT with global shared states and events (Figure 12). The keypress events are secret while click and mouseover events are public. The L outputs are shown in the blue color while H outputs are shown in red.

$$\boxed{G, \mathcal{P} \vdash K_1^G \overset{\alpha_l}{\Longrightarrow} K_2^G}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.Ev(v)) = H \\ \mathcal{D}(\rho, id.Ev(v)) = (r, \mathsf{emp}, \rho') \qquad d' = \mathsf{update}(d, r) \qquad G, \mathcal{P}, \sigma \vdash \cdot; \mathsf{lookupEHAll}(id.ev(v)) \rightsquigarrow_H ks\end{array}}{G, \mathcal{P} \vdash (\rho, \mathcal{D}), d; \sigma; \cdot \overset{id.Ev(v)}{\Longrightarrow} (\rho', \mathcal{D}), d'; \sigma; ks} \ \textsc{In-NR1}$$

$$\frac{\mathcal{P}(id.Ev(v)) = H_\Delta \qquad G, \mathcal{P}, \sigma \vdash \cdot; \mathsf{lookupEHAll}(id.ev(v)) \rightsquigarrow_H ks}{G, \mathcal{P} \vdash \mathcal{R}, d; \sigma; \cdot \overset{id.Ev(v)}{\Longrightarrow} \mathcal{R}, d; \sigma; ks} \ \textsc{In-NR2}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.Ev(v)) = H \qquad \mathcal{D}(\rho, id.Ev(v)) = (r, v', \rho') \qquad v' \neq v \qquad d' = \mathsf{update}(d, r) \\ G, \mathcal{P}, \sigma \vdash \cdot; \mathsf{lookupEHAt}(id.ev(v')) \rightsquigarrow_L ks \qquad G, \mathcal{P}, \sigma \vdash ks; \mathsf{lookupEHAt}(id.ev(v)) \rightsquigarrow_H ks'\end{array}}{G, \mathcal{P} \vdash (\rho, \mathcal{D}), d; \sigma; \cdot \overset{id.Ev(v)}{\Longrightarrow} (\rho', \mathcal{D}), d'; \sigma; ks'} \ \textsc{In-R-Diff}$$

$$\frac{\begin{array}{c}\mathcal{P}(id.Ev(v)) = H \\ \mathcal{D}(\rho, id.Ev(v)) = (r, v, \rho') \qquad d' = \mathsf{update}(d, r) \qquad G, \mathcal{P}, \sigma \vdash \cdot; \mathsf{lookupEHAll}(id.ev(v)) \rightsquigarrow_. ks\end{array}}{G, \mathcal{P} \vdash (\rho, \mathcal{D}), d; \sigma; \cdot \overset{id.Ev(v)}{\Longrightarrow} (\rho', \mathcal{D}), d'; \sigma; ks} \ \textsc{In-R-Same}$$

$$\frac{\mathcal{P}(id.Ev(v)) = L \qquad G, \mathcal{P}, \sigma \vdash \cdot; \mathsf{lookupEHAll}(id.ev(v)) \rightsquigarrow_. ks}{G, \mathcal{P} \vdash \mathcal{R}, d; \sigma; \cdot \overset{id.Ev(v)}{\Longrightarrow} \mathcal{R}, d; \sigma; ks} \ \textsc{In-L}$$

Figure 16: Input rules with declassification

dynamically-generated elements. From prior work [49], these events should never influence declassification so they do not result in a public event, nor do they result in changes to the declassification channel or the state.

In-R-Diff and In-R-Same are for secret events that are declassified to a public event. Here, $\mathcal{D}(\rho, id.Ev(v))$ returns (r, v', ρ') where v' (instead of emp) indicates that the event will be declassified. If v' is the same as the original argument to the event, v, then In-R-Same applies and otherwise, In-R-Diff applies. If the released event is different (In-R-Diff) then lookupEHAt in the L context returns event handlers labeled L (or $\cdot$) to run in the L context, and lookupEHAt in the H context returns event handlers labeled H (or $\cdot$) to run in the H context. If the released event is the same (In-R-Same) then lookupEHAll in the $\cdot$ context returns all of the event handlers to run in whatever context they are visible in.

In-L processes public events. Like for In-R-Same, it runs all event handlers associated with the event in whatever context they are visible in.

$$\boxed{G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa \overset{\alpha}{\Longrightarrow} \sigma_2^G, ks}$$

$$\frac{\begin{array}{c}E \neq \cdot \qquad G, \mathcal{P}, \mathcal{V}, \sigma^G \vdash \kappa; \mathsf{lookupEHs}(E) \rightsquigarrow_{pc} ks \\ \kappa = (\mathcal{V}; (\sigma, \mathsf{skip}, C, \cdot); pc)\end{array}}{G, \mathcal{P}, \mathcal{V}, d \vdash \sigma^G, \sigma, \mathsf{skip}, P, E \xrightarrow{\bullet}_{pc} \sigma^G, ks} \ \text{LC}$$

$$\frac{\kappa = (\mathcal{V}; (\sigma, \mathsf{skip}, C, \cdot); pc)}{G, \mathcal{P}, \mathcal{V}, d \vdash \sigma^G, \sigma, \mathsf{skip}, P, \cdot \xrightarrow{\bullet}_{pc} \sigma^G, \kappa} \ \textsc{PtoC}$$

$$\frac{\begin{array}{c}G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c_1 \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2, c_2, E_2 \\ \kappa = (\mathcal{V}; (\sigma_2, c_2, P, (E_1, E_2)); pc)\end{array}}{G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \sigma_1, c_1, P, E_1 \xrightarrow{\alpha}_{pc} \sigma_2^G, \kappa} \ \text{P}$$

Figure 17: Rules for managing the event handler queue

Appendix D. EH Queue Semantics

The rules for processing the event handler queue are shown in Figure 17. LC ("local consumer") processes simulated events, which were generated by an event handler rather than a user. This rule is triggered when an event handler has finished running (the current command is skip) and the local event queue is not empty ($E \neq \cdot$). Event handlers are looked up using lookupEHs which returns the event handlers with label matching the current pc (or $\cdot$) and runs them at the current pc, or in the case of TT, returns all event handlers and runs them at the current pc if the event being triggered is public, or runs them at H if the event being triggered is secret. The resulting configuration stack ks includes the current event handler (now in consumer state) on top of all of the event handlers triggered by lookupEHs.

PtoC switches the execution state from producer to consumer when the current event handler finishes running (i.e., the command is skip) and there are no simulated events triggered by the event handler to process ($E = \cdot$). P runs the current event handler for one step using the event handler semantics.

Appendix E. Enforcement-Specific Semantics

Figure 18 contains expression semantics and conversion rules. Note that pc_l denotes a pc which cannot be $\cdot$ (i.e. it must be L or H). In favor of simplicity, we show only the rules for SME, SMS, TT, and TS.

$$\frac{i \in \{\mathsf{SME}, \mathsf{SMS}, \mathsf{MF}, \mathsf{FS}\}}{\mathsf{toDst}(v^{\mathsf{std}}, pc, i) = v^{\mathsf{std}}} \qquad \frac{i \in \{\mathsf{TT}, \mathsf{TS}\} \qquad pc \sqsubseteq L}{\mathsf{toDst}(v^{\mathsf{std}}, pc, i) = (v^{\mathsf{std}}, L)} \qquad \frac{i \in \{\mathsf{TT}, \mathsf{TS}\} \qquad pc \not\sqsubseteq L}{\mathsf{toDst}(v^{\mathsf{std}}, pc, i) = (v^{\mathsf{std}}, pc)}$$

$$\frac{i \in \{\mathsf{TT}, \mathsf{TS}\}}{\mathsf{toDst}((v^{\mathsf{std}}, l), pc_l, i) = (v^{\mathsf{std}}, l \sqcup pc_l)} \qquad \frac{i \notin \{\mathsf{TT}, \mathsf{TS}\} \qquad l \sqsubseteq pc_l}{\mathsf{toDst}((v^{\mathsf{std}}, l), pc_l, i) = v^{\mathsf{std}}} \qquad \frac{i \notin \{\mathsf{TT}, \mathsf{TS}\} \qquad l \not\sqsubseteq pc_l}{\mathsf{toDst}((v^{\mathsf{std}}, l), pc_l, i) = \mathsf{dv}}$$

$$\frac{}{\mathsf{toDst}((v^{\mathsf{std}}, L), \cdot, i) = v^{\mathsf{std}}} \qquad \frac{}{\mathsf{toDst}((v^{\mathsf{std}}, H), \cdot, i) = \langle v^{\mathsf{std}} | \mathsf{dv} \rangle} \qquad \frac{v = \langle \_ | \_ \rangle}{\mathsf{toDst}(v, \cdot, i) = v}$$

Figure 18: Conversion rules

Variable lookup and assignment. The rules for SME, SMS variable lookup and assignment are shown below. Note that for SME, the appropriate copy of the store is passed to these functions, so accesses and updates can be done directly. For SMS, a helper $\mathsf{getStore}(\sigma, pc_l)$ function would need to be used to retrieve the appropriate copy of the store from σ (see an example below).

$$\frac{}{\mathsf{var}_{\mathsf{SME}}(\sigma^{\mathsf{std}}, pc_l, x) = \sigma^{\mathsf{std}}(x)} \text{ VAR}$$

$$\frac{x \notin \sigma^{\mathsf{std}}}{\mathsf{var}_{\mathsf{SME}}(\sigma^{\mathsf{std}}, pc_l, x) = \mathsf{dv}} \text{ VAR-DV}$$

$$\frac{}{\mathsf{assign}_{\mathsf{SME}}(\sigma^{\mathsf{std}}, H, x, v) = \sigma^{\mathsf{std}}[x \mapsto v]} \text{ ASSIGN-H}$$

$$\frac{}{\mathsf{assign}_{\mathsf{SME}}(\sigma^{\mathsf{std}}, L, x, v) = \sigma^{\mathsf{std}}[x \mapsto v]} \text{ ASSIGN-L}$$

The rules for TT, TS variable lookup and assignment are shown below. If the variable is not in the store as in VAR-DV, a tainted default value is returned. This is important since an attacker would not know the difference between $x \notin \sigma$ and $\sigma(x) = (v, H)$. To satisfy noninterference, we should return a tainted default value even when $pc = L$. ASSIGN joins the label of the value with the pc to reflect the context in which the assignment was performed.

$$\frac{}{\mathsf{var}_{\mathsf{TT}}(\sigma, pc_l, x) = \sigma(x)} \text{ VAR}$$

$$\frac{x \notin \sigma}{\mathsf{var}_{\mathsf{TT}}(\sigma, pc_l, x) = (\mathsf{dv}, H)} \text{ VAR-DV}$$

$$\frac{}{\mathsf{assign}_{\mathsf{TT}}(\sigma, pc, x, (v, l)) = \sigma[x \mapsto (v, l \sqcup pc)]} \text{ ASSIGN}$$

Unstructured DOM. The enforcement-specific functions for unstructured event handler storage includes:

- DOM attribute lookup and assignment
- DOM node lookup
- Simulating (i.e. script-triggered) event
- Creating a DOM node
- Registering a new event handler

The rules for DOM attribute lookups and assignments are shown in Figure 19.

$\mathsf{getStore}(\sigma, pc_l)$ is used for SMS to return from σ the copy of the store with label matching pc_l. Similarly, $\mathsf{setStoreVar}(\sigma^G, pc_l, \sigma)$ updates the pc_l copy of the global variable store in σ^G to be σ and $\mathsf{setStore}(\sigma^G, pc_l, \sigma)$ does the same for the event handler store in σ^G.

Similar to the TT store, looking up uninitialized variables from the TS store with VAR-DV and looking up the attribute of a node which doesn't exist with GETVALG-S returns a tainted dv. The global variable assignment ASSIGN joins the label of the value with the pc to reflect the context in which the value was assigned. The attribute assignment ASSIGNDOM joins the label of the value with the pc and label of the node to ensure that if the attribute is looked up, the result does not leak anything about the existence of the node (which may be secret).

The rules for the other DOM functionality and the tree-structured DOM may be found in the TR [48].

Appendix F. Proof Sketches for Security Theorems

We include the proof sketches for the top-level theorems here to give some intuition for how the requirements from Section 5 map to the proofs. Complete proofs and supporting definitions (like well-formedness and equivalence definitions) may be found in the TR [48].

Theorem 2 (Soundness) *If* $\forall id.Ev(v), eh, pc.$ $\mathcal{P}(id.Ev(v), eh, pc) \in \{\mathsf{SME}, \mathsf{MF}\}$ *and* $\mathcal{G}_g, \mathcal{G}_{EH} \in \{\mathsf{SMS}, \mathsf{FS}\}$ *and* $G = (\mathcal{G}_g, \mathcal{G}_{EH})$, *then* $\forall \mathcal{R}, \mathcal{P}, \sigma_0, T, K, \alpha_l$ *s.t.* $T \xRightarrow{\alpha_l} K \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{P}, \mathcal{I})$, σ_0 *is well-formed,*

- *If* $\mathsf{rlsAction}(\mathsf{last}(T) \xRightarrow{\alpha_l} K)$*:* $\mathcal{K}(T \xRightarrow{\alpha_l} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_l)$
- *Otherwise:* $\mathcal{K}(T \xRightarrow{\alpha_l} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$

Proof (sketch): The proof is split between two cases, depending on whether the most recent action was a release event or not. In either case, we want to show that $\exists \tau' \in \mathcal{K}(T \xRightarrow{\alpha_l} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$ s.t. $\tau \preceq \tau'$ for τ in $\mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_l)$ (when α_l is a release event) or $\mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$ (when α_l is not a release event). The second case relies on Requirements **T1** and **T4** The first case follows from the definitions of $\mathcal{K}_{rp}()$ and $\mathcal{K}()$. □

In general, the structure of the weak secrecy proofs is similar to the proofs for PI Security, except that it uses the "weak" versions of the requirements.

Theorem 4 (Soundness-Weak Secrecy) *If* $\forall id.Ev(v), eh, pc.$ $\mathcal{P}(id.Ev(v), eh, pc) \in \{\mathsf{SME}, \mathsf{MF}, \mathsf{TT}\}$ *and* $\mathcal{G}_g, \mathcal{G}_{EH} \in \{\mathsf{SMS}, \mathsf{FS}, \mathsf{TS}\}$ *and* $G = (\mathcal{G}_g, \mathcal{G}_{EH})$, *then* $\forall \mathcal{R}, \mathcal{P}, \sigma_0, T, K, \alpha_l$ *s.t.* $T \xRightarrow{\alpha_l} K \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{P}, \mathcal{I})$, σ_0 *is well-formed,*

$$\frac{\sigma' = \mathsf{getStore}(\sigma\downarrow_{EH}, pc_l) \qquad \sigma'(id) = \phi}{\mathsf{getVal}_{\mathsf{SMS}}(\sigma, pc_l, id) = \phi.v}\ \textsc{GetVal} \qquad \frac{\sigma' = \mathsf{getStore}(\sigma\downarrow_{EH}, pc_l) \qquad id \notin \sigma'}{\mathsf{getVal}_{\mathsf{SMS}}(\sigma, pc_l, id) = \mathsf{dv}}\ \textsc{GetVal-s}$$

$$\frac{\sigma' = \mathsf{getStore}(\sigma\downarrow_{EH}, pc_l) \qquad (v', M) = \sigma'(id) \qquad \sigma'' = \sigma'[id \mapsto (v, M)]}{\mathsf{assign}_{\mathsf{SMS}}(\sigma, pc_l, id, v) = \mathsf{setStore}(\sigma, pc_l, \sigma'')}\ \textsc{assign-DOM}$$

$$\frac{\sigma' = \mathsf{getStore}(\sigma\downarrow_{EH}, pc_l) \qquad id \notin \sigma'}{\mathsf{assign}_{\mathsf{SMS}}(\sigma, pc_l, id, v) = \sigma}\ \textsc{assign-DOM-s}$$

(a) Unstructured SMS DOM attribute lookups and assignments

$$\frac{\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) = \phi \qquad \mathsf{valOf}(\phi) \neq \mathsf{NULL} \qquad l_\phi = \mathsf{labOf}(\phi, pc_l) \qquad (v, l_v) = \phi.v}{\mathsf{getValG}_{\mathsf{TS}}(\sigma, pc_l, id) = (v, l_\phi \sqcup l_v)}\ \textsc{GetValG}$$

$$\frac{\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) = \phi \qquad \mathsf{valOf}(\phi) = \mathsf{NULL}}{\mathsf{getValG}_{\mathsf{TS}}(\sigma, pc_l, id) = (\mathsf{dv}, H)}\ \textsc{GetValG-s}$$

$$\frac{\sigma(id) = (v', M, l')}{\mathsf{assign}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = \sigma[id \mapsto ((v, l \sqcup pc \sqcup l'), M, l')]}\ \textsc{assignDOM} \qquad \frac{id \notin \sigma}{\mathsf{assign}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = \sigma}\ \textsc{assignDOM-s}$$

(b) Unstructured TS DOM attribute lookups and assignments

Figure 19: Enforcement-specific EH lookups and assignments for the unstructured DOM

<table>
<tr><td>Attacker Knowledge</td><td>$\mathcal{K}(T, \sigma_0^G, \mathcal{R}, \mathcal{S})$</td><td>$\{\tau_i \mid \exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{S}), T \approx_L^{\mathcal{S}} T' \wedge \tau_i = \mathsf{in}(T')\}$</td></tr>
<tr><td>Progress-insensitive Knowledge</td><td>$\mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{S})$</td><td>$\{\tau_i \mid \exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{S}), T \approx_L^{\mathcal{S}} T' \wedge \tau_i = \mathsf{in}(T') \wedge \mathsf{prog}(T', \mathcal{S})\}$</td></tr>
<tr><td>Progress-insensitive Knowledge with Release</td><td>$\mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{S}, \alpha)$</td><td>$\{\tau_i \mid \exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{S}), T \approx_L^{\mathcal{S}} T' \wedge \tau_i = \mathsf{in}(T') \wedge \mathsf{prog}(T', \mathcal{S})$
$\wedge \alpha' = (\mathsf{last}(T) \overset{\alpha}{\Longrightarrow} K)) \downarrow_L^{\mathcal{S}}, \mathsf{rlsTrace}(T', \alpha'))\}$</td></tr>
<tr><td>Weak Progress-insensitive Knowledge</td><td>$\mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{S}, \alpha)$</td><td>$\{\tau_i \mid \exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{S}), T \approx_L^{\mathcal{S}} T' \wedge \tau_i = \mathsf{in}(T') \wedge \mathsf{prog}(T', \mathcal{S})$
$\wedge \alpha' = (\mathsf{last}(T) \overset{\alpha}{\Longrightarrow} K)) \downarrow_L^{\mathcal{S}}, \mathsf{wkTrace}(T', \alpha'))\}$</td></tr>
<tr><td colspan="3">$\mathsf{prog}(T, \mathcal{S})$ iff $T = G, \mathcal{S} \vdash K_0 \Longrightarrow^* K$ and $\exists K_C$ s.t. $G, \mathcal{S} \vdash K \Longrightarrow^* K_C$ and $\mathsf{consumer}(K_C)$</td></tr>
<tr><td colspan="3">$\mathsf{rlsAction}(G, \mathcal{S} \vdash K \overset{\alpha}{\Longrightarrow} K')$ iff $(G, \mathcal{S} \vdash K \overset{\alpha}{\Longrightarrow} K') \downarrow_L^{\mathcal{S}} = \mathsf{rls}(\alpha')$ or $(G, \mathcal{S} \vdash K \overset{\alpha}{\Longrightarrow} K') \downarrow_L^{\mathcal{S}} = \mathsf{declassify}(\iota, v)$</td></tr>
<tr><td colspan="3">$\mathsf{rlsTrace}(T, \alpha) = \begin{cases} T = G, \mathcal{S} \vdash K_0 \Longrightarrow^* K \wedge \exists \alpha', K' s.t., K \overset{\alpha'}{\Longrightarrow} K' \wedge (K \overset{\alpha'}{\Longrightarrow} K') \downarrow_L^{\mathcal{S}} = \alpha & \alpha = \mathsf{rls}(\_) \\ T = G, \mathcal{S} \vdash K_0 \Longrightarrow^* K \wedge \exists K' s.t., K \overset{\alpha}{\Longrightarrow} K' & \alpha = \mathsf{declassify}(\iota, v) \end{cases}$</td></tr>
<tr><td colspan="3">$\mathsf{wkAction}(G, \mathcal{S} \vdash K \overset{\alpha}{\Longrightarrow} K')$ iff $(G, \mathcal{S} \vdash K \overset{\alpha}{\Longrightarrow} K') \downarrow_{L,w}^{\mathcal{S}} = \mathsf{br}(b)$ or $(G, \mathcal{S} \vdash K \overset{\alpha}{\Longrightarrow} K') \downarrow_{L,w}^{\mathcal{S}} = \mathsf{gw}(x)$ or
$\alpha \in \mathsf{in}(G, \mathcal{S} \vdash K \overset{\alpha}{\Longrightarrow} K') \wedge \mathcal{S}(\alpha) = L$ or $(G, \mathcal{S} \vdash K \overset{\alpha}{\Longrightarrow} K') \downarrow_L^{\mathcal{S}} \in \{\bullet, ch(v)\}$</td></tr>
<tr><td colspan="3">$\mathsf{wkTrace}(T, \alpha) = \begin{cases} T = G, \mathcal{S} \vdash K_0 \Longrightarrow^* K \wedge \exists K' \text{ s.t. } K \Longrightarrow K' \wedge (K \Longrightarrow K') \downarrow_L^{\mathcal{S}} = \alpha & \alpha = \mathsf{br}(b) \\ T = G, \mathcal{S} \vdash K_0 \Longrightarrow^* K \wedge \exists K', T' \text{ s.t. } T' = K \Longrightarrow^* K' \wedge T' \downarrow_L^{\mathcal{S}} = \cdot \wedge \exists K'' \text{ s.t. } (K' \Longrightarrow^* K'') \downarrow_L^{\mathcal{S}} = \alpha & \alpha = \mathsf{gw}(x) \\ T = G, \mathcal{S} \vdash K_0 \Longrightarrow^* K \wedge \exists T', K' \text{ s.t. } T' = K \Longrightarrow^* K_C \wedge T' \downarrow_L^{\mathcal{S}} = \cdot \wedge \mathsf{consumer}(K_C) & \alpha = id.Ev(v) \\ T = G, \mathcal{S} \vdash K_0 \Longrightarrow^* K \wedge \exists T', K' \text{ s.t. } T' = K \Longrightarrow^* K_{lp} \wedge T' \downarrow_L^{\mathcal{S}} = \cdot \wedge \mathsf{lowProducer}(K_{lp}) & \alpha \in \{\bullet, ch(v)\} \end{cases}$</td></tr>
</table>

Figure 20: Definitions for attacker knowledge and its invariants

- *If* $\mathsf{rlsAction}(\mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K)$*:*
  $\mathcal{K}(T \overset{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_l)$
- *If* $\mathsf{wkAction}(\mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K)$*:*
  $\mathcal{K}(T \overset{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_l)$
- *Otherwise:*
  $\mathcal{K}(T \overset{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_{\preceq} \mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$

Proof (sketch): The proof is split between three cases, depending on whether the most recent action was a release event, weak action, or neither. In either case, we want to show that $\mathcal{K}(T \overset{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$ s.t. $\tau \preceq \tau'$ for τ in $\mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_l)$ (when α_l is a release event) $\mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_l)$ (when α_l is a weak action) or $\mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$ (otherwise). The first case follows from the definitions of $\mathcal{K}_{rp}()$ and $\mathcal{K}()$. The second case relies on the definitions of $\mathcal{K}_{wp}()$, $\mathcal{K}()$, and Requirements **WT1** and **WT4** The third case follows from the definitions of $\mathcal{K}_p()$. □

Theorem 5 (PI Security implies PI Weak Secrecy) *If script enforcement $\mathcal{V}$ and global storage enforcement G are progress-insensitive secure for some well-formed initial global store, σ_0^G, then $\mathcal{V}$ and G are also weakly progress-insensitive secure.* *Proof (sketch):* We want to show that the conditions for weak secrecy hold for any trace from a system that is progress-insensitive secure. Considering a trace from a system that satisfies progress-insensitive security, this proof looks at three cases: one where the last step was a release, one where it was a "weak action", and one where it was neither. If the last step was a release, then the trace trivially satisfies progress-insensitive weak secrecy since the condition for release events is the same for both types of security. If the last step was a "weak action", then the conclusion follows from the definitions of $\mathcal{K}_p()$ and $\mathcal{K}_{wp}()$. Finally, if the action was neither a release nor a "weak action", then the proof is straightforward since the "other" condition for both types of security is the same. □