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

***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], [68], 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], [65], [70].

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 [65], [70] 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-

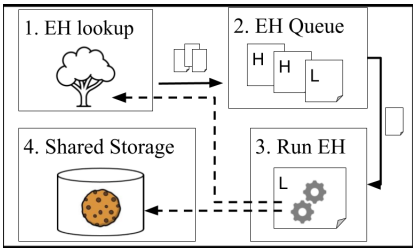


Figure 1: High-level depiction of a reactive system

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 the Appendix.

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:

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

A weaker form of noninterference that allows implicit leaks is *weak secrecy* [70] or *explicit secrecy* [65]. 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:

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

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 [65], [70]. 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

```
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

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.

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* [65], [70], 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:

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

Program syntax:

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

Runtime configurations:

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

Figure 2: Syntax for the compositional framework

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

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

$$\frac{\mathcal{P}(id.Ev(v)) = H \qquad G, \mathcal{P}, \sigma \vdash \mathsf{lookupEHAll}(id.ev(v)) \rightsquigarrow_H ks}{G, \mathcal{P} \vdash \sigma; \cdot \xrightarrow{id.Ev(v)} \sigma; ks}\ \textsc{In-H}$$

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

(a) Simplified input rules

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

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

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

(b) Simplified output rules

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

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. $\mathsf{producer}(\kappa)$ and $\mathsf{consumer}(\kappa)$ 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. $\mathsf{out}_{\mathcal{V}}(...)$ 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. $\mathsf{assign}_G(...)$ 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-

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

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

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

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

Figure 4: Selected command semantics

Shared Storage

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

SME/SMS **Variable Storage**

Single store: $\sigma_{pc} \quad ::= \quad \cdot \mid x \mapsto v$

SME/SMS *Storage:* $\sigma^{\mathsf{SME}}, \sigma_g^{\mathsf{SMS}} \quad ::= \quad \sigma_H, \sigma_L$

MF/FS **Variable Storage**

Faceted value: $v^{\mathsf{MF}}, v^{\mathsf{FS}} \quad ::= \quad v \mid \langle v_H | v_L \rangle \mid \langle \cdot | v \rangle \mid \langle v | \cdot \rangle$

MF *Storage:* $\sigma^{\mathsf{MF}} \quad ::= \quad \cdot \mid \sigma^{\mathsf{MF}}, x \mapsto v^{\mathsf{MF}}$

FS *Storage:* $\sigma_g^{\mathsf{FS}} \quad ::= \quad \cdot \mid \sigma_g^{\mathsf{FS}}, x \mapsto v^{\mathsf{FS}}$

TT/TS **Variable Storage**

Labeled value: $v^{\mathsf{TT}}, v^{\mathsf{TS}} \quad ::= \quad (v, l)$

TT *Storage:* $\sigma^{\mathsf{TT}} \quad ::= \quad \cdot \mid \sigma^{\mathsf{TT}}, x \mapsto v^{\mathsf{TT}}$

TS *Storage:* $\sigma_g^{\mathsf{TS}} \quad ::= \quad \cdot \mid \sigma_g^{\mathsf{TS}}, x \mapsto v^{\mathsf{TS}}$

Figure 5: Storage syntax

EH Storage:

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

Unstructured SMS **DOM:**

Single store: $\sigma_{pc} \quad ::= \quad \cdot \mid id \mapsto (v, M)$

DOM: $\sigma_{EH}^{\mathsf{SMS}} \quad ::= \quad \sigma_H, \sigma_L$

Unstructured FS **DOM:**

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

Unstructured TS **DOM:**

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

DOM addresses:

Location: $loc \quad \in \quad \mathsf{Address}$

Address: $a \quad ::= \quad loc \mid \mathsf{NULL}$

Root address: $a^{\mathsf{rt}} \quad ::= \quad loc$

Address list: $A \quad ::= \quad \cdot \mid A, a$

Tree-structured SMS **DOM:**

Node: $\phi^{\mathsf{SMS}} \quad ::= \quad (id, v, M, a_p, A)$

Single store: $\sigma_{pc} \quad ::= \quad a^{\mathsf{rt}} \mapsto \phi^{\mathsf{SMS}} \mid \sigma_{pc}, loc \mapsto \phi^{\mathsf{SMS}}$

DOM: $\sigma_{EH}^{\mathsf{SMS}} \quad ::= \quad \sigma_H, \sigma_L$

Tree-structured FS **DOM:**

Faceted address: $a^{\mathsf{FS}} \quad ::= \quad a \mid \langle a_H | a_L \rangle \mid \langle \cdot | a \rangle \mid \langle a | \cdot \rangle$

Faceted address list: $A^{\mathsf{FS}} \quad ::= \quad \cdot \mid a^{\mathsf{FS}} :: A^{\mathsf{FS}}$

Node: $\phi^{\mathsf{FS}} \quad ::= \quad (id, v^{\mathsf{FS}}, M, a_p^{\mathsf{FS}}, A^{\mathsf{FS}})$

DOM: $\sigma_{EH}^{\mathsf{FS}} \quad ::= \quad a^{\mathsf{rt}} \mapsto \phi^{\mathsf{FS}} \mid \sigma^{\mathsf{FS}}, loc \mapsto \phi^{\mathsf{FS}}$

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 [48], which is useful for reactive systems like OS processes, as well as a more traditional tree-structure [61], 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 [48] 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 [61] 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}	$\langle \cdot \vert v^{\mathsf{std}} \rangle$	$\langle v^{\mathsf{std}} \vert \cdot \rangle$	(v^{std}, L)	(v^{std}, L)	(v^{std}, H)
	$\langle v_H \vert v_L \rangle$	$\langle v_H \vert v_L \rangle$	v_L	v_H	$\langle v_H \vert v_L \rangle$	v_L	v_H	$\langle v_H \vert v_L \rangle$	(v_L, L)	(v_H, H)
	$\langle v \vert \cdot \rangle$	$\langle v \vert \cdot \rangle$	dv	v	$\langle v \vert \cdot \rangle$	dv	v	$\langle v \vert \cdot \rangle$	(dv, L)	(v, H)
	$\langle \cdot \vert v \rangle$	$\langle \cdot \vert v \rangle$	v	dv	$\langle \cdot \vert v \rangle$	v	dv	$\langle \cdot \vert v \rangle$	(v, L)	(dv, H)
	(v, L)	$-$	v	v	v	v	v	$-$	(v, L)	(v, H)
	(v, H)	$-$	dv	v	$\langle v \vert \mathsf{dv} \rangle$	dv	v	$-$	(v, H)	(v, H)

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

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

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 $T = G, \mathcal{P} \vdash T' \xRightarrow{\alpha_l} K$ 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 42.

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

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

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

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

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

Figure 9: Rules for projecting execution traces to L

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. $(T \overset{\alpha}{\Longrightarrow} K) \in \mathsf{runs}(\sigma_0^G, \mathcal{P})$, then $\mathcal{K}(T \overset{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{P}) \supseteq_\preceq \mathcal{K}_p(T, \sigma_0^G, \mathcal{P})$.*

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 44. 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* [65], [70] 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.

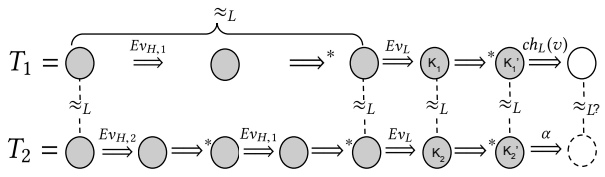
Knowledge-based weak secrecy definition. Since we allow information to leak through control flow decisions, we define another form of knowledge to capture this:

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

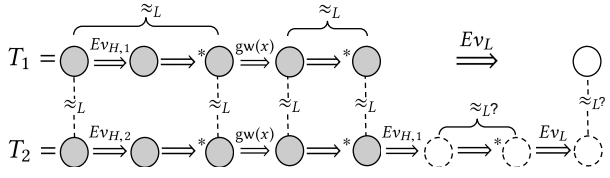
$\mathsf{last}(T)$ returns the last configuration in a trace. Here, $\approx_L$ ensures the implicit leaks *up to this point* were the same and wkTrace ensures *the next* implicit leak is the same. If T is about to output $\mathsf{br}(b)$ or $\mathsf{gw}(x)$, then T' can be extended to produce the same 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 ● $\stackrel{\alpha}{\Longrightarrow}$ ○ 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 ● $\stackrel{Ev_L}{\Longrightarrow}$ ○, 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. $(T \stackrel{\alpha}{\Longrightarrow} K) \in \mathsf{runs}(\sigma_0^G, \mathcal{P})$, the following holds*

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

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 $\mathcal{V} \in \{\mathsf{SME}, \mathsf{MF}, \mathsf{TT}\}$ and the global storage is enforced with $G \in \{\mathsf{SMS}, \mathsf{FS}, \mathsf{TS}\}$, 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 44 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 44. 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* [63] of secret information can be incorporated into our framework without extensive modification. Based on prior work on stateful declassification for SME [25], [48], [69], 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], [74], mobile phone applications [33], [39], [55], 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 [65], [70]. 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 [60], 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], [71] discusses its extension to applications where the policy is specified separately from the code. Ngo et al. [56] generalize MF to multi-level lattices. Stefan et al. [67] 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. [73] 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. [57] 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. [64] 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 [63] in reactive systems is an interesting problem. Various approaches have been proposed for declassifying information in reactive systems that employ SME [25], [48], [59], [69] 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], [54], 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 [70] originally defined weak secrecy as a means to formalize data-dependent flows as opposed to the stronger property of noninterference. Schoepe et al. [65] 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. [66] 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], [68] 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], [60], [61]. 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], [49]. McCullough, further, defined the property of restrictiveness for security of systems [51] based on what a user can infer about sensitive data, which is composable. Zakinthinos and Lee [72] showed important results about the composition of generalized noninterference, which was earlier proven to be not fully compositional [50]. 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], [52], [53], [62] given the complexity that concurrency introduces. Bauereiss et al. [18] verify the security of a distributed social media platform by composition. Rafnsson and Sabelfeld [58] 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, H. Zhang, and L. Jia. Knowledge-based security of dynamic secrets for reactive programs. In *2018 IEEE CSF*, 2018.

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

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

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

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

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

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

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

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

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

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

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

[60] 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.

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

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

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

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

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

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

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

[68] 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.

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

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

[71] 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.

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

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

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

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 $o := 1$; output (L, o); output (H, o)	- 2; •	$\sigma_H[o \mapsto 1]$ •; 1	- 2; •	$\sigma[o \mapsto \langle 1 \vert 2 \rangle]$ •; 1	- 2; •	$\sigma[o \mapsto 1^H]$ •; 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 $o := 1$; output (L, o); output (H, o)	$\sigma_H[o \mapsto 1]$ 2; 1	$\sigma[o \mapsto \langle 1 \vert 2 \rangle]$ 2; 1	$\sigma[o \mapsto 1^H]$ 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.

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

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 $o := 1$; output (L, o); output (H, o)	 2; 2	 2; 2	$\sigma[o \mapsto 1^L]$ 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.

Appendix A.
Syntax

Syntax for the compositional framework is shown in Figure 16. Syntax for processing the event handler queue and running event handlers is shown in Figure 17. Syntax for variable storage is shown in Figure 18. The unstructured event handler storage syntax is in Figure 19 and the tree-structured event handler storage syntax is in Figure 20. Finally, the syntax for miscellaneous runtime constructs is shown in Figure 21. Annotations are not included in the semantics and proofs when they are clear from the context. In α_l, the label of the actions determines whether they are suppressed and is useful for proofs. The label for intuts is determined by $\mathcal{P}$, so the extra label is not necessary.

Standard label set:	$\mathcal{L}_l$	::=	$\{L, H\}$ *with* $L \sqsubseteq H$
Compositional label set:	$\mathcal{L}$	::=	$\{\cdot, L, H\}$ *with* $\cdot \sqsubseteq L \sqsubseteq H$
Program counter:	pc_l	$\in$	$\mathcal{L}_l$
Compositional program counter:	pc	$\in$	$\mathcal{L}$
Security label:	l	$\in$	$\mathcal{L}$
Event handler enforcement:	$\mathcal{V}$	$\in$	$\{\mathsf{SME}, \mathsf{MF}, \mathsf{TT}\}$
Global storage enforcement:	$\mathcal{G}$	$\in$	$\{\mathsf{SMS}, \mathsf{FS}, \mathsf{TS}\}$
Comp. global storage enforcement:	G	::=	$\mathcal{G}_{EH}, \mathcal{G}_g$
Policy context	$\mathcal{P}$	::=	(Γ, m_l)
Initial IDs	Γ	::=	$\cdot \mid \Gamma, id$
Release	$\mathcal{R}$	::=	$(\rho, \mathcal{D})$
Released value	r	::=	$\mathsf{none} \mid \mathsf{some}(\iota, v)$
Release Channel	d	::=	$\cdot \mid d, (\iota, v)$
Event:	Ev	::=	$\mathsf{Click} \mid \ldots$
Event handler:	eh	::=	$\mathsf{on}Ev(x)\{c\}$
Standard actions:	α	::=	$id.Ev(v^{\mathsf{std}}) \mid ch(v^{\mathsf{std}}) \mid \bullet$
Labeled actions:	α_l	::=	$id.Ev(v^{\mathsf{std}}) \mid (ch(v^{\mathsf{std}}), l) \mid (\bullet, l)$
Compositional configuration:	K^G	::=	$\mathcal{R}, d; \sigma^G; ks$
Global state:	$\sigma^{(\mathcal{G}_{EH}, \mathcal{G}_g)}$	::=	$\sigma^{\mathcal{G}_{EH}}_{EH}, \sigma^{\mathcal{G}_g}_g$

Figure 16: Framework syntax

Compositional framework syntax. The syntax for our compositional enforcement framework is managed by the top-level semantics given by the judgement $G, \mathcal{P} \vdash K^G_1 \overset{\alpha_L}{\Longrightarrow} K^G_2$ (complete semantics in Section C.1). The syntax for our compositional framework is shown in Figure 16.

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. The program context label indicates the context under which the event handlers execute and is denoted pc. Sometimes the pc is restricted to a traditional label, in which case we denote pc_l as a pc which cannot be $\cdot$.

The enforcement mechanisms for event handlers is given by $\mathcal{V}$. Here, we consider SME (SME), faceted execution (MF), and taint tracking (TT). Global storage enforcement mechanisms are given by $\mathcal{G}$. This might be SMS (SMS) which stores everything twice, as in secure multi-execution, a faceted store (FS) which stores values influenced by secrets as facets, like faceted execution, or a tainted store (TS), which stores everything with an associated label which reflects the secrecy level of that data, as is done in taint tracking. The global variable store, $\mathcal{G}_g$, and event handler storage, $\mathcal{G}_{EH}$, may be enforced differently. Together, they form the compositional mechanism G.

The policy context $\mathcal{P}$ keeps track of the labels assigned to input events and output channels, and is also responsible for deciding which events handlers run with which enforcement mechanism using m_l. Using a technique from prior work, we track the IDs of the elements in the event handler store using Γ so that we may prevent the dynamic elements, which may have been added by the attacker, from influencing declassification. The release module $\mathcal{R}$ and declassification channel d come from prior work on stateful declassification which maintains some internal state to determine what to declassify and when. d maps locations, ι, to declassified values, while $\mathcal{R}$ determines whether or not an event is declassified. Event handlers read from a location in d to receive declassified values, possibly aggregated over time (e.g. declassify the average location of 10 clicks). Events released by $\mathcal{R}$ may have the same or different parameters (e.g. declassify the approximate GPS location). The value released, r, may be none (nothing is declassified) or a value at a particular location, $\mathsf{some}(\iota, v)$.

Events Ev (including Click, Keypress) trigger relevant event handlers of the form on$Ev(x)\{c\}$. On triggering Ev, command c runs with input x. Actions emitted by the execution, α, include user-generated input events, outputs on channels $ch(v)$, and silent actions, denoted $\bullet$. We use labeled actions α_l in our semantics to make proofs easier.

The compositional configuration K^G is a snapshot of the current state of the system. This includes the declassification module and channel, $\mathcal{R}, d$, shared storage σ^G which stores the event handlers as well as the variables shared between event handlers, and a configuration stack ks which keeps track of which event handlers are about to run. The structure of σ^G and ks depend on which enforcement mechanisms are composed in the framework. Their syntax is discussed in more detail below.

$$
\begin{array}{llll}
\textit{Configuration stack:} & ks & ::= & \cdot \mid (\mathcal{V}; \kappa^{\mathcal{V}}; pc) :: ks \\
\textit{Execution state:} & s & ::= & P \mid C \\
\textit{Events:} & E & ::= & \cdot \mid E, (id.Ev(v^{\mathsf{std}}), l) \\
 & & & \\
\textit{Expression:} & e & ::= & x \mid v \mid \mathsf{uop}\, e \mid e_1\, \mathsf{bop}\, e_2 \mid \mathsf{ehAPIe}(e_1, \cdots, e_n) \\
\textit{Command:} & c & ::= & \mathsf{skip} \mid c_1; c_2 \mid x := e \mid id := e \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \\
 & & \mid & \mathsf{output}\ ch\ e \mid x := \mathsf{declassify}(\iota, e) \mid \mathsf{ehAPIc}(e_1, \cdots, e_n) \\
 & & & \\
\textit{Standard configuration:} & \kappa^{\mathsf{std}} & ::= & \sigma^{\mathsf{std}}, c, s, E \\
\textit{SME configuration:} & \kappa^{\mathsf{SME}} & ::= & \kappa^{\mathsf{std}}_H, \kappa^{\mathsf{std}}_L \\
\textit{MF Configuration:} & \kappa^{\mathsf{MF}} & ::= & \sigma^{\mathsf{MF}}, c, s, E \\
\textit{TT Configuration:} & \kappa^{\mathsf{TT}} & ::= & \sigma^{\mathsf{TT}}, c, s, E
\end{array}
$$

Figure 17: Syntax for event handler queue processing and running event handlers

Event handler syntax. The next 2 levels of our semantics are responsible for processing the event handler queue and running the individual event handlers. The judgement for the first is $G, \mathcal{V}, d \vdash \sigma^G_1, \kappa \xrightarrow{\alpha}_{pc} \sigma^G_2, ks$ (complete semantics in Section C.2) and the second is $G, \mathcal{V}, d \Vdash \sigma^G_1, \sigma^{\mathcal{V}}_1, c^{\mathsf{std}}_1 \xrightarrow{\alpha}_{pc} \sigma^G_2, \sigma^{\mathcal{V}}_2, c_2, E$ (complete semantics in Section C.3). The syntax for both is shown in Figure 17.

The configuration stack ks keeps track of the event handlers which are about to run. The one at the top of the stack runs first. Each event handler on the stack includes $\mathcal{V}$, which determines the enforcement mechanism used to run the event handler, $\kappa^{\mathcal{V}}$, which is a snapshot of the current state of the event handler, and pc, which is the context the event handler is running in. The enforcement mechanism used for a particular event handler is determined by the policy, $\mathcal{P}$, and may be different for different event handlers.

Commands and expressions are mostly standard in our framework. The event handler APIs ehAPIe and ehAPIc interact with the event handler store, and $id := e$ updates the attributes in the event handler store. The declassification command declassifies e at a given location, ι.

A standard configuration κ^{std} contains a local store, $\sigma^{\mathcal{V}}$, 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 any events triggered by that event handler, E. The structure of κ will differ slightly for each enforcement mechanism. The SME configuration κ^{SME} has two standard configurations, one copy for the L context and one for the H context. The MF configuration and TT configurations are similar to the standard configurations, except that the structures of their local stores will differ. The variable storage for each of these configurations is described next.

$$
\begin{array}{llll}
\textit{Integer:} & n & \in & \mathcal{N} \\
\textit{Boolean:} & b & ::= & \mathsf{true} \mid \mathsf{false} \\
\textit{Standard value:} & v^{\mathsf{std}} & ::= & n \mid b \mid \mathsf{dv} \\
\textit{Standard state:} & \sigma^{\mathsf{std}} & ::= & \cdot \mid \sigma^{\mathsf{std}}, x \mapsto v^{\mathsf{std}} \\
\mathsf{SMS}\ \textit{state:} & \sigma^{\mathsf{SMS}}_g & ::= & \sigma^{\mathsf{std}}_H, \sigma^{\mathsf{std}}_L \\
 & & & \\
\textit{Faceted value:} & v^{\mathsf{MF}}, v^{\mathsf{FS}} & ::= & v^{\mathsf{std}} \mid \langle v^{\mathsf{std}}_H | v^{\mathsf{std}}_L \rangle \mid \langle \cdot | v^{\mathsf{std}}_L \rangle \mid \langle v^{\mathsf{std}}_H | \cdot \rangle \\
\mathsf{MF}\ \textit{local state:} & \sigma^{\mathsf{MF}} & ::= & \cdot \mid \sigma^{\mathsf{MF}}, x \mapsto v^{\mathsf{MF}} \\
\mathsf{FS}\ \textit{shared variable state:} & \sigma^{\mathsf{FS}}_g & ::= & \cdot \mid \sigma^{\mathsf{FS}}_g, x \mapsto v^{\mathsf{FS}} \\
 & & & \\
\textit{Tainted value:} & v^{\mathsf{TT}}, v^{\mathsf{TS}} & ::= & (v^{\mathsf{std}}, l) \\
\mathsf{TT}\ \textit{local state:} & \sigma^{\mathsf{TT}} & ::= & \cdot \mid \sigma^{\mathsf{TT}}, x \mapsto v^{\mathsf{TT}} \\
\mathsf{TS}\ \textit{shared variable state:} & \sigma^{\mathsf{TS}}_g & ::= & \cdot \mid \sigma^{\mathsf{TS}}_g, x \mapsto v^{\mathsf{TS}}
\end{array}
$$

Figure 18: Syntax for variable storage

Variable storage syntax. Values include integers (n), booleans (b), and a pre-determined default value dv, which is used to replace the public copy of private data in multi-execution techniques. A different default value can be assigned

to every value type; for simplicity we use a single default value. Values may be formatted differently depending on the enforcement mechanism, "standard" value formats are annotated with std, as in v^{std}. SME (σ^{SME} for temporary variables, local to a single event handler) and SMS (σ^{SMS} for variables which persist, to be accessible to multiple event handlers) variable storage maps variables to standard values. For SME, each copy of the execution has its own (standard) store, σ^{std}. SMS shared variable storage involves multiple copies of the standard store: one for each security level.

Faceted storage (σ^{MF} for temporary variables, local to a single event handler, and σ^{FS} for persistent variables, accessible to multiple event handlers) will involve faceted values, $v^{\mathsf{MF}}, v^{\mathsf{FS}}$, which may be a standard value (if the value does not depend on a secret) or a value with multiple copies (if the value does depend on a secret. A faceted value with two copies $\langle v_H | v_L \rangle$ is v_H to privileged, H observers and v_L to unprivileged, L observers. A value which does not exist in a context has an empty facet (e.g., a variable with value $\langle v | \cdot \rangle$ only exists in the H context). Tainted storage (σ^{TT} for temporary variables, local to a single event handler, and σ^{TS} for persistent variables, accessible to multiple event handlers) involves tainted values, $v^{\mathsf{TT}}, v^{\mathsf{TS}}$, which is a standard value annotated with a security label.

Standard event handler map:	M	::=	$\cdot \mid M, Ev \mapsto \{(eh_1, pc_{l1}), \cdots, (eh_k, pc_{lk})\}$
Std. node:	$\phi^{\mathsf{SMS}}, \phi^{\mathsf{std}}$	::=	$(v^{\mathsf{std}}, M) \mid \mathsf{NULL}$
Standard EH state:	$\sigma^{\mathsf{std}}_{EH}$	::=	$\cdot \mid \sigma^{\mathsf{std}}_{EH}, id \mapsto \phi^{\mathsf{std}}$
SMS EH state:	$\sigma^{\mathsf{SMS}}_{EH}$	::=	$\sigma^{\mathsf{std}}_{EH,H}, \sigma^{\mathsf{std}}_{EH,L}$
Faceted EH map:	M^{FS}	::=	$\cdot \mid M^{\mathsf{FS}}, Ev \mapsto \{(eh_1, pc_1), \cdots (eh_k, pc_k)\}$
FS *node:*	ϕ^{FS}	::=	$(v^{\mathsf{FS}}, M^{\mathsf{FS}}) \mid \mathsf{NULL}$
FS *EH state:*	$\sigma^{\mathsf{FS}}_{EH}$	::=	$\cdot \mid \sigma^{\mathsf{FS}}_{EH}, id \mapsto \phi^{\mathsf{FS}}$
TS *node:*	ϕ^{TS}	::=	$(v^{\mathsf{TS}}, M, l) \mid \mathsf{NULL}$
TS *EH state:*	$\sigma^{\mathsf{TS}}_{EH}$	::=	$\cdot \mid \sigma^{\mathsf{TS}}_{EH}, id \mapsto \phi^{\mathsf{TS}}$

Figure 19: Unstructured event hanlder storage syntax

Event handler storage syntax. In the unstructured event handler store, nodes are identified by a unique identifier (id) and contain both an attribute (whose structure is determined by the type of enforcement) 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) to prevent leaks due to the existence of an event handler. 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.

The unstructured SMS event handler storage has two copies. Privileged, H observers interact with the H copy of the event handler store and likewise for unprivileged, L observers. Attributes are standard values (v). The unstructured FS event handler store 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 ($\langle v_H | v_L \rangle$). A node 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 a node added in only the L context. The TS event handler store will associate labels with both attributes ((v, l)) and nodes ((v^{TS}, M, l)). The label on the node reflects the context the node was created in, while the label on the attribute reflects whether the attribute has been influenced by a secret ($l = H$) or not ($l = L$).

Location:	loc	$\in$	Address
Address:	a	::=	$loc \mid \mathsf{NULL}$
Address of root:	a^{rt}	::=	loc
Value:	v^{std}	::=	$n \mid b \mid \mathsf{dv} \mid a$
Values (actions):	v^{α}	::=	$n \mid b \mid \mathsf{dv}$
Actions:	α	::=	$id.Ev(v^{\alpha}) \mid ch(v^{\alpha}) \mid \bullet$
Address list:	A	::=	$\cdot \mid a :: A$
Faceted address list:	A^{FS}	::=	$\cdot \mid a^{\mathsf{FS}} :: A^{\mathsf{FS}}$
Std. node:	$\phi^{\mathsf{SMS}}, \phi^{\mathsf{std}}$	::=	$(id, v, M, a_p, A) \mid \mathsf{NULL}$
FS *node:*	ϕ^{FS}	::=	$(id, v^{\mathsf{FS}}, M^{\mathsf{FS}}, a_p^{\mathsf{FS}}, A^{\mathsf{FS}}) \mid \mathsf{NULL}$
Standard EH state:	$\sigma^{\mathsf{std}}_{EH}$	::=	$a^{\mathsf{rt}} \mapsto \phi^{\mathsf{std}} \mid \sigma^{\mathsf{std}}_{EH}, a \mapsto \phi^{\mathsf{std}}$
Faceted EH state:	$\sigma^{\mathsf{FS}}_{EH}$	::=	$a^{\mathsf{rt}} \mapsto \phi^{\mathsf{FS}} \mid \sigma^{\mathsf{FS}}_{EH}, a \mapsto \phi^{\mathsf{FS}}$
SMS EH state:	$\sigma^{\mathsf{std}}_{EH}$	::=	$\sigma^{\mathsf{std}}_{EH,H}, \sigma^{\mathsf{std}}_{EH,L}$

Figure 20: Tree-structured event handler storage syntax

In the tree-structured event handler store, the nodes (ϕ) are stored by reference (loc). Nodes have a unique identifier (id), an attribute, and an event handler map, like in the unstructured event handler store. They also contain a pointer to

their parent (a_p), and a list of pointers to their children (A) (if any). The root node 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.

The tree-structured SMS event handler store has two copies and similar to the unstructured SMS event handler store. The tree-structured FS event handler store 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.

Single enforcement mechanism:	i	$::=$	$\mathcal{V} \mid \mathcal{G}$
Term:	t	$::=$	$v \mid \langle v_H \mid v_L \rangle \mid \phi^{\mathsf{std}} \mid \langle \phi_H^{\mathsf{std}} \mid \phi_L^{\mathsf{std}} \rangle$
Runtime SME configuration:	κ^{SME}	$::=$	$\kappa_H^{\mathsf{std}}, \kappa_L^{\mathsf{std}} \mid \kappa_H^{\mathsf{std}} \mid \kappa_L^{\mathsf{std}}$
Runtime SME store:	σ^{SME}	$::=$	$\sigma_H^{\mathsf{std}} \mid \sigma_L^{\mathsf{std}}$
Labeled faceted actions:	α_l^{MF}	$::=$	$\alpha_l \mid \langle \alpha \mid \alpha' \rangle$
Tainted actions:	α^{TT}	$::=$	$\bullet \mid ch((v^{\mathsf{std}}, l))$
Faceted command:	c^{MF}	$::=$	$c^{\mathsf{std}} \mid \langle c_H^{\mathsf{std}} \mid c_L^{\mathsf{std}} \rangle$
EH queue:	$\mathcal{C}$	$::=$	$\cdot \mid \mathcal{C}, (eh, pc)$

Figure 21: Syntax for runtime semantics

Runtime syntax. We use i to refer to any enforcement mechanism (event handler or shared storage) and t to refer to any term including values, faceted values, nodes, or faceted nodes. The runtime SME configuration and store are useful for splitting the SME executions, since only one can run at a time. When an output command tries to output a faceted value, we replace the action with a faceted action which outputs un-faceted values. This way, the channel receives the value it expects (a standard, un-faceted value). The faceted action also lets us replace one of the actions with $\bullet$ for the observer who does not actually receive the output. This is important for proofs. The tainted action is an intermediate action that emitted by mid-level semantics which is ultimately changed to a traditional action α_l if the output is permitted (i.e., the channel has enough privilege to see the value (v^{std}, l)) or $\bullet$ if it is not. MF may produce a faceted command if branching on a faceted value yields different commands. The mid-level event handling semantics split a faceted command in the $\cdot$ context into two event handlers which run each facet in the L and H context. $\mathcal{C}$ is a list of event handlers and the context they should run in, used by the event handler lookup semantics.

Appendix B.
Supporting Definitions

B.1. General Definitions

Value operations. valOf takes a value or node and returns a standard value by removing any attached labels. It is not defined for faceted values or nodes. labOf takes a value or a node and a pc and returns the label on the value, if there is one, or the pc if the value is not labeled.

$$\mathsf{valOf}(v) = \begin{cases} v & v \text{ is a standard value} \\ v_1 & v = (v_1, \ell) \\ \mathsf{undefined} & v = \langle v_1 | v_2 \rangle \end{cases}$$

$$\mathsf{labOf}(v, pc) = \begin{cases} pc & v \text{ is not a labeled value} \\ \ell & v = (v', \ell) \end{cases}$$

$$\mathsf{valOf}(\phi) = \begin{cases} \phi & \phi \text{ is a standard node} \\ \phi_1 & \phi = (\phi_1, \ell) \\ \mathsf{undefined} & v = \langle \phi_1 | \phi_2 \rangle \vee \phi \text{ is a faceted node} \end{cases}$$

$$\mathsf{labOf}(\phi, pc) = \begin{cases} pc & \phi \text{ is a standard, SMS, or FS node} \\ \ell & \phi = (\phi, \ell) \end{cases}$$

State, G projection. $\sigma^G \downarrow_{EH}$ returns the event handler store from the shared storage, while $\sigma^G \downarrow_g$ returns the variable store from the shared storage. $G \downarrow_{EH}$ returns the enforcement mechanism for the event handler storage and $G \downarrow_g$ returns the enforcement mechanism for the shared variable storage.

$$\frac{\sigma^G = \sigma^{\mathcal{G}}_{EH}, \sigma^{\mathcal{G}'}_g}{\sigma^G \downarrow_{EH} = \sigma^{\mathcal{G}}_{EH}} \qquad \frac{\sigma^G = \sigma^{\mathcal{G}}_{EH}, \sigma^{\mathcal{G}'}_g}{\sigma^G \downarrow_g = \sigma^{\mathcal{G}'}_g}$$

$$\frac{G = \mathcal{G}_{EH}, \mathcal{G}_g}{G \downarrow_{EH} = \mathcal{G}_{EH}} \qquad \frac{G = \mathcal{G}_{EH}, \mathcal{G}_g}{G \downarrow_g = \mathcal{G}_g}$$

consumer/producer. The consumer predicate holds for a compositional configuration if there are no event handlers running (i.e., the configuration stack ks is empty). If there is at least one element in ks, the producer predicate holds. The consumer predicate holds for a configuration stack if the configuration on the top of the stack is in consumer state (i.e., the consumer(κ) predicate holds). Similarly, the producer predicate holds for a configuration stack if the configuration on the top of the stack is in producer state (i.e., the producer(κ) predicate holds). For $ks = \cdot$, neither predicate holds for ks.

$$\frac{K = \mathcal{R}, d; \sigma^G; \cdot}{\mathsf{consumer}(K) = \mathsf{true}} \qquad \frac{K = \mathcal{R}, d; \sigma^G; (\mathcal{V}; \kappa; pc) :: ks}{\mathsf{producer}(K) = \mathsf{true}}$$

$$\frac{ks = (\mathcal{V}; \kappa; pc) :: ks' \qquad \mathsf{consumer}(\kappa)}{\mathsf{consumer}(ks) = \mathsf{true}} \qquad \frac{}{\mathsf{consumer}(\cdot) = \mathsf{false}} \qquad \frac{ks = (\mathcal{V}; \kappa; pc) :: ks' \qquad \mathsf{producer}(\kappa)}{\mathsf{producer}(ks) = \mathsf{true}}$$

$$\frac{}{\mathsf{producer}(\cdot) = \mathsf{false}}$$

$$\frac{\kappa = \sigma, c, C, E}{\mathsf{consumer}(\kappa) = \mathsf{true}} \qquad \frac{\kappa = \sigma, c, P, E}{\mathsf{producer}(\kappa) = \mathsf{true}} \qquad \frac{\mathsf{consumer}(\kappa_H) \qquad \mathsf{consumer}(\kappa_L)}{\mathsf{consumer}((\kappa_H; \kappa_L)) = \mathsf{true}} \qquad \frac{\mathsf{producer}(\kappa_H) \vee \mathsf{producer}(\kappa_L)}{\mathsf{producer}((\kappa_H; \kappa_L)) = \mathsf{true}}$$

B.2. Operations on Faceted Values

Updating/accessing facets. Set and get facet are for creating and updating faceted values. Note for getFacet and dv: the type of the default value can be inferred from the other facet. To keep things simple, we use the same default value for all types.

$$\frac{}{\mathsf{createFacet}(v, H) = \langle v|\cdot\rangle} \qquad \frac{}{\mathsf{createFacet}(v, L) = \langle \cdot|v\rangle} \qquad \frac{}{\mathsf{createFacet}(v, \cdot) = v}$$

$$\frac{v_L = v_o \downarrow_L \qquad v_H = \mathsf{getFacet}(v_n, H)}{\mathsf{updateFacet}(v_o, v_n, H) = \langle v_H|v_L\rangle} \qquad \frac{v_L = v_o \downarrow_H \qquad v_H = \mathsf{getFacet}(v_n, L)}{\mathsf{updateFacet}(v_o, v_n, L) = \langle v_H|v_L\rangle} \qquad \frac{}{\mathsf{updateFacet}(v_o, v_n, \cdot) = v_n}$$

$$\frac{v_H \neq v_L}{\mathsf{setFacetV}(v_H, v_L) = \langle v_H|v_L\rangle} \qquad \frac{}{\mathsf{setFacetV}(v, v) = v} \qquad \frac{\phi_H = (v_H, M_H) \qquad \phi_L = (v_L, M_L) \qquad v_H \neq v_L \vee M_H \neq M_L}{\mathsf{setFacetN}(\phi_H, \phi_L) = \langle \phi_H|\phi_L\rangle}$$

$$\frac{\phi_H = (v, M) \qquad \phi_L = (v, M)}{\mathsf{setFacetN}(\phi_H, \phi_L) = \phi_H} \qquad \frac{c_H \neq c_L}{\mathsf{setFacetC}(c_H, c_L) = \langle c_H|c_L\rangle} \qquad \frac{}{\mathsf{setFacetC}(c, c) = c}$$

$$\frac{}{\mathsf{getFacetV}(v, \cdot) = v} \qquad \frac{v \downarrow_{pc_l} = v_l}{\mathsf{getFacetV}(v, pc_l) = v_l} \qquad \frac{v \downarrow_{pc_l} = \cdot}{\mathsf{getFacetV}(v, pc_l) = \mathsf{dv}} \qquad \frac{\alpha_l \downarrow_l = \alpha}{\mathsf{getFacet}\alpha(\alpha_l, l) = \alpha}$$

$$\frac{\alpha_l \downarrow_l = \cdot}{\mathsf{getFacet}\alpha(\alpha_l, l) = \bullet} \qquad \frac{}{\mathsf{getFacetA}(a, \cdot) = a} \qquad \frac{a \downarrow_{pc_l} \neq \cdot}{\mathsf{getFacetA}(a, pc_l) = a \downarrow_{pc_l}} \qquad \frac{a \downarrow_{pc_l} = \cdot}{\mathsf{getFacetA}(a, pc_l) = \mathsf{NULL}}$$

Optimization for faceted events. We merge locally triggered events in MF so that if the same event handler is triggered in both the H and L context, it runs just once in the $\cdot$ context rather than running twice.

$$\frac{}{\mathsf{mergeEvs}((id.Ev(v), H), (id.Ev(v), L)) = (id.Ev(v), \cdot)}\ \textsc{mergeEv-same}$$

$$\frac{id \neq id' \quad \text{or} \quad Ev \neq Ev' \quad \text{or} \quad v \neq v'}{\mathsf{mergeEvs}((id.Ev(v), H), (id'.Ev'(v'), L)) = (id.Ev(v), H), (id'.Ev'(v'), L)}\ \textsc{mergeEv-diff}$$

$$\frac{E_H = \cdot \text{ or } E_L = \cdot}{\mathsf{mergeEvs}(E_H, E_L) = E_H :: E_L}\ \textsc{mergeEv-s1}$$

Faceted value projection. The L projection of a faceted value is the L facet and likewise for H. The projection of standard values is the same value. Note that we use projection rather than getFacet for determining if two stores are equivalent so if the projection produces $\cdot$ it means the value has not been initiated in that context. We return $\cdot$ to hide that value instead of returning dv so that a store with a $\cdot$ facet and a store without the value at all will still be equivalent.

$\boxed{v \downarrow_{pc} = v'}$

$$\frac{}{v^{\mathsf{std}} \downarrow_{pc} = v^{\mathsf{std}}} \qquad \frac{}{\alpha \downarrow_{pc} = \alpha} \qquad \frac{}{\langle \_|v_L\rangle \downarrow_L = v_L} \qquad \frac{}{\langle v_H|\_\rangle \downarrow_H = v_H} \qquad \frac{}{\langle \_|\alpha\rangle \downarrow_L = \alpha} \qquad \frac{}{\langle \alpha|\_\rangle \downarrow_H = \alpha}$$

$$\frac{}{\langle \_|\cdot\rangle \downarrow_L = \cdot} \qquad \frac{}{\langle \cdot|\_\rangle \downarrow_H = \cdot}$$

B.3. Operations on Event Handlers

We define two projection operations for event handlers. These are used by the event handler lookup semantics. $\downarrow_{pc}$ returns all of the event handler with label visible to pc. When the $pc = \cdot$ we return all event handlers. $@_{pc}$ is the same as $\downarrow_{pc}$ for $pc \sqsubseteq L$, but when $pc = H$ it returns only the H and $\cdot$ event handlers.

$\boxed{(eh), l) \downarrow_{pc}}$

$$\frac{}{(id.Ev(v), l) \downarrow_{\cdot} = (id.Ev(v), l)} \qquad \frac{l \not\sqsubseteq pc_l}{(id.Ev(v), l) \downarrow_{pc_l} = \cdot} \qquad \frac{l \sqsubseteq pc_l}{(id.Ev(v), l) \downarrow_{pc_l} = (id.Ev(v), l)}$$

$\boxed{(eh, l)@_{pc}}$

$$\frac{pc \sqsubseteq L \qquad l \sqsubseteq L}{(eh, l)@pc = (eh, l)} \qquad \frac{pc \not\sqsubseteq L \qquad pc = l \vee pc = \cdot}{(eh, l)@pc = (eh, pc)} \qquad \frac{pc \sqsubseteq L \qquad l \not\sqsubseteq L}{(eh, l)@pc = \cdot} \qquad \frac{pc \not\sqsubseteq L \qquad pc = l \vee pc = \cdot}{(eh, l)@pc = (eh, pc)}$$

B.4. Operations on SMS stores

getStore takes an SMS store and a pc and returns the copy of the store indicated by the pc. setStoreVar updates the SMS shared variable storage. It takes a compositional store σ^G, a pc, and an updated copy of the SMS store indicated by the pc. It updates σ^G so that the pc copy of the global variable store is updated to the given store. setStore does something similar for the SMS EH store.

$$\frac{}{\mathsf{getStore}((\sigma_H, \sigma_L), L) = \sigma_L}\ \textsc{getStore-L} \qquad \frac{}{\mathsf{getStore}((\sigma_H, \sigma_L), H) = \sigma_H}\ \textsc{getStore-H}$$

$$\frac{\sigma^G = ((\sigma_H, \sigma_L), \sigma_{EH})}{\mathsf{setStoreVar}(\sigma^G, H, \sigma) = ((\sigma, \sigma_L), \sigma_{EH})}\ \textsc{setStoreVar-H}$$

$$\frac{\sigma^G = ((\sigma_H, \sigma_L), \sigma_{EH})}{\mathsf{setStoreVar}(\sigma^G, L, \sigma) = ((\sigma_H, \sigma), \sigma_{EH})}\ \textsc{setStoreVar-L}$$

$$\frac{\sigma^G = (\sigma_g, (\sigma_H, \sigma_L))}{\mathsf{setStore}(\sigma^G, H, \sigma) = (\sigma_g, (\sigma, \sigma_L))}\ \textsc{setStore-H} \qquad \frac{\sigma^G = (\sigma_g, (\sigma_H, \sigma_L))}{\mathsf{setStore}(\sigma^G, L, \sigma) = (\sigma_g, (\sigma_H, \sigma))}\ \textsc{setStore-L}$$

B.5. Well-formed Initial State

We say that an initial state is well-formed if all of the following are true:

- Every global variable has been initialized. We assume the set of global variables is static; no new variables are created and the ones which exist in the initial state persist for the duration of the execution
- For the SMS event handler store, the event handlers in the H copy of the store only contains event handlers with label H, and likewise for the event handlers in the L copy
- For the TS event handler store, tainted nodes may only contain event handlers with label H. Un-tainted nodes may contain event handlers labeled L or H

Appendix C.
Semantics

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

$$\frac{\mathcal{P}(id.Ev(v)) = H \quad \mathcal{D}(\rho, id.Ev(v)) = (r, \mathsf{emp}, \rho') \quad d' = \mathsf{update}(d, r) \quad G, \mathcal{P}, \sigma \vdash \cdot; \mathsf{lookupEHAll}(id.ev(v)) \rightsquigarrow_H ks}{G, \mathcal{P} \vdash (\rho, \mathcal{D}), d; \sigma; \cdot \overset{id.Ev(v)}{\Longrightarrow} (\rho', \mathcal{D}), d'; \sigma; ks} \text{ I-NR1}$$

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

$$\frac{\mathcal{P}(id.Ev(v)) = H \quad \mathcal{D}(\rho, id.Ev(v)) = (r, v', \rho') \quad v' \neq v \quad d' = \mathsf{update}(d, r) \quad G, \mathcal{P}, \sigma \vdash \cdot; \mathsf{lookupEHAt}(id.ev(v')) \rightsquigarrow_L ks \quad G, \mathcal{P}, \sigma \vdash ks; \mathsf{lookupEHAt}(id.ev(v)) \rightsquigarrow_H ks'}{G, \mathcal{P} \vdash (\rho, \mathcal{D}), d; \sigma; \cdot \overset{id.Ev(v)}{\Longrightarrow} (\rho', \mathcal{D}), d'; \sigma; ks'} \text{ I-R-DIFF}$$

$$\frac{\mathcal{P}(id.Ev(v)) = H \quad \mathcal{D}(\rho, id.Ev(v)) = (r, v, \rho') \quad d' = \mathsf{update}(d, r) \quad G, \mathcal{P}, \sigma \vdash \cdot; \mathsf{lookupEHAll}(id.ev(v)) \rightsquigarrow_\cdot ks}{G, \mathcal{P} \vdash (\rho, \mathcal{D}), d; \sigma; \cdot \overset{id.Ev(v)}{\Longrightarrow} (\rho', \mathcal{D}), d'; \sigma; ks} \text{ I-R-SAME}$$

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

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

$$\frac{\mathsf{producer}(\kappa) \quad G, \mathcal{P}, \mathcal{V}, d \vdash \sigma, \kappa \overset{ch(v)}{\longrightarrow}_{pc} \sigma', ks' \quad \mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v), pc) \quad \alpha_l = \mathsf{output}(\mathcal{P}, ch(v), pc)}{G, \mathcal{P} \vdash \mathcal{R}, d; \sigma; ((\mathcal{V}; \kappa; pc) :: ks) \overset{\alpha_l}{\Longrightarrow} \mathcal{R}, d; \sigma', ks' :: ks} \text{ O}$$

$$\frac{\mathsf{producer}(\kappa) \quad G, \mathcal{P}, \mathcal{V}, d \vdash \sigma, \kappa \overset{ch(v)}{\longrightarrow}_{pc} \sigma', ks' \quad \neg\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v), pc)}{G, \mathcal{P} \vdash \mathcal{R}, d; \sigma; ((\mathcal{V}; \kappa; pc) :: ks) \overset{(\bullet, pc)}{\Longrightarrow} \mathcal{R}, d; \sigma', ks' :: ks} \text{ O-SKIP}$$

$$\frac{\mathsf{producer}(\kappa) \quad G, \mathcal{P}, \mathcal{V}, d \vdash \sigma, \kappa \overset{\alpha}{\longrightarrow}_{pc} \sigma', ks' \quad \alpha \neq ch(v)}{G, \mathcal{P} \vdash \mathcal{R}, d; \sigma; ((\mathcal{V}; \kappa; pc) :: ks) \overset{(\alpha, pc)}{\Longrightarrow} \mathcal{R}, d; \sigma', ks' :: ks} \text{ O-OTHER}$$

$$\frac{\mathsf{consumer}(\kappa)}{G, \mathcal{P} \vdash \mathcal{R}, d; \sigma; ((\mathcal{V}; \kappa; pc) :: ks) \overset{(\bullet, pc)}{\Longrightarrow} \mathcal{R}, d; \sigma, ks} \text{ O-NEXT}$$

C.1. Framework Semantics

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 $\mathcal{P}$.

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 release module is used to tell us whether or not an event is declassified and 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.

I-NR1 handles the case where a secret input ($\mathcal{P}(id.Ev(v)) = H$) is not released to a public event (the release event given by $\mathcal{D}(id.Ev(v))$ is emp). The declassification state (ρ) and channel (d) are updated. Similarly, I-NR2 handles a secret input associated with a dynamic page element, which is never released to a public event, nor does it update the declassification state/channel. In both cases, the appropriate event handlers are looked up using lookupEHAll in the H context.

I-R-DIFF and I-R-SAME handle cases where a secret input is released to a public event. In the first case, the released event is passed a different argument from the original event ($v' \neq v$), in the second case, the arguments are

the same. When the released event is different, we run H event handlers with the original argument and the L event handlers with the released argument, so we use $\mathsf{lookupEHAt}$ with in the H and L context, respectively. When the event is the same, we run all event handlers in whichever context they were registered in, so we use $\mathsf{lookupEHAll}$ in the $\cdot$ context.

Public events ($\mathcal{P}(id.Ev(v)) = L$) are handled by I-L. Public events do not need to be declassified, so the declassification state/channel are not updated. Again, we use $\mathsf{lookupEHAll}$ in the $\cdot$ context to run all of the event handlers in whichever context they were registered in.

Similar to the input rules, the output rules are the same regardless of the enforcement mechanism or event handler storage. $\mathsf{producer}(\kappa)$ and $\mathsf{consumer}(\kappa)$ 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 (O, O-SKIP, or O-OTHER) and when the event handler has finished, the system is in consumer state (O-NEXT) and the current event handler can be popped off ks. $\mathsf{outCondition}_{\mathcal{V}}(...)$ determines if an output should be allowed (O) or suppressed (O-SKIP) and output determines what the final output is. O-OTHER applies when the action is not an output (such as a silent $\bullet$ step).

$\boxed{G, \mathcal{V}, \sigma^G \vdash ks; \mathsf{lookupEhAPI}(...) \rightsquigarrow_{pc} ks'}$

$$\frac{\mathcal{C} = \mathsf{lookupEhAPI}_G(\sigma^G, id.Ev(v), pc) \qquad ks' = \mathsf{createK}(\mathcal{P}, id.Ev(v), \mathcal{C})}{G, \mathcal{P}, \sigma^G \vdash ks; \mathsf{lookupEhAPI}(id.Ev(v)) \rightsquigarrow_{pc} ks :: ks'} \text{ LOOKUPEHAPI}$$

$$\frac{ks = (\mathcal{V}; \kappa; pc) :: \_ \qquad \mathcal{C} = \mathsf{lookupEHs}_{G,\mathcal{V}}(\sigma^G, id.Ev(v), pc \sqcup l) \qquad ks' = \mathsf{createK}(\mathcal{P}, id.Ev(v), \mathcal{C}) \qquad G, \mathcal{P}, \sigma^G \vdash ks :: ks'; \mathsf{lookupEHs}(E) \rightsquigarrow_{pc} ks''}{G, \mathcal{P}, \sigma^G \vdash ks; \mathsf{lookupEHs}((id.Ev(v), l), E) \rightsquigarrow_{pc} ks''} \text{ LOOKUPEHS-R}$$

$$\frac{}{G, \mathcal{P}, \sigma^G \vdash ks; \mathsf{lookupEHs}(\cdot) \rightsquigarrow_{pc} ks} \text{ LOOKUPEHS-S}$$

$$\frac{\mathcal{P}(id.Ev(v), eh, pc) = \mathcal{V}}{\mathsf{createK}(\mathcal{P}, id.Ev(v), ((c, pc), \mathcal{C})) = \mathsf{crtK}_{\mathcal{V}}(eh, v, pc) :: \mathsf{createK}(\mathcal{P}, id.Ev(v), \mathcal{C})} \qquad \frac{}{\mathsf{createK}(\mathcal{P}, id.Ev(v), \cdot) = \cdot}$$

$$\frac{}{\mathsf{crtK}_{\mathsf{SME}}(eh, v, L) = (\mathsf{SME}; ((\cdot, \mathsf{skip}, C, \cdot); (\cdot, eh(v), P, \cdot)); L)} \qquad \frac{}{\mathsf{crtK}_{\mathsf{SME}}(eh, v, H) = (\mathsf{SME}; ((\cdot, eh(v), P, \cdot); (\cdot, \mathsf{skip}, C, \cdot)); H)}$$

$$\frac{}{\mathsf{crtK}_{\mathsf{SME}}(eh, v, \cdot) = (\mathsf{SME}; ((\cdot, eh(v), P, \cdot); (\cdot, eh(v), P, \cdot)); L)} \qquad \frac{}{\mathsf{crtK}_{\mathsf{MF}}(eh, v, l) = (\mathsf{MF}; (\cdot, eh(v), P, \cdot); l)}$$

$$\frac{l \sqsubseteq L}{\mathsf{crtK}_{\mathsf{TT}}(eh, v, l) = (\mathsf{TT}; (\cdot, eh(v), P, \cdot); L)} \qquad \frac{l \not\sqsubseteq L}{\mathsf{crtK}_{\mathsf{TT}}(eh, v, l) = (\mathsf{TT}; (\cdot, eh(v), P, \cdot); H)}$$

Event handler lookup semantics. The event handler lookup semantics use the judgement $G, \mathcal{V}, \sigma^G \vdash ks; \mathsf{lookupEhAPI}(...) \rightsquigarrow_{pc} ks'$. They take the global store and current configuration stack, ks, use a helper function by the same name as the $\mathsf{lookupEhAPI}$ to look up a list of relevant event handlers and the context they should run in ($\mathcal{C}$). To turn this list into a configuration stack, $\mathsf{createK}$ looks up the enforcement mechanism, $\mathcal{V}$, and formats the configuration depending on which mechanism is used ($\mathsf{crtK}_{\mathcal{V}}(...)$). The final result is another configuration stack, ks', which is appended to the old one $ks :: ks'$.

$\mathsf{lookupEHAt}$ runs event handlers labeled with given pc, or labeled with $\cdot$ using the @ operator. Once event handlers are collected, they are run as follows, depending on the enforcement mechanism:

- SME: runs in matching execution only
- MF, TT: runs at given pc

$\mathsf{lookupEHAll}$ runs event handlers labeled *at or below* the given pc using the $\downarrow$ operator. Once event handlers are collected, they are run as follows, depending on the enforcement mechanism:

- SME: run in H execution if label is at or below H, run in L execution if label is at or below L
- MF: run at the label (i.e. H, L, $\cdot$)
- TT: run at the label if the label is H or L, run at L if the label is $\cdot$

$\boxed{\mathsf{lookupEhAPI}_G(\sigma^G, pc, id.Ev(v)) = \mathcal{C}}$

$$\frac{\mathsf{lookup}_{G\downarrow_{EH}}(\sigma, pc_l, id) = \phi \qquad \mathsf{valOf}(\phi) \neq \mathsf{NULL} \qquad \mathsf{labOf}(\phi, pc_l) = l \qquad \mathcal{C} = (\phi.M(Ev)@pc_l) \sqcup pc_l \sqcup l}{\mathsf{lookupEHAt}_G(\sigma, pc_l, id.Ev(v)) = \mathsf{mergeC}(\mathcal{C})}\ \textsc{lookupEHAt}$$

$$\frac{\mathsf{lookup}_{G\downarrow_{EH}}(\sigma, pc_l, id) = \phi \qquad \mathsf{valOf}(\phi) = \mathsf{NULL}}{\mathsf{lookupEHAt}_G(\sigma, pc_l, id.Ev(v)) = \cdot}\ \textsc{lookupEHAt-s}$$

$$\frac{\mathsf{lookupEHAt}_G(\sigma, H, id.Ev(v)) = \mathcal{C}_H \qquad \mathsf{lookupEHAt}_G(\sigma, L, id.Ev(v)) = \mathcal{C}_L \qquad \mathsf{mergeC}(\mathcal{C}_H, \mathcal{C}_L) = \mathcal{C}}{\mathsf{lookupEHAt}_G(\sigma, \cdot, id.Ev(v)) = \mathcal{C}}\ \textsc{lookupEHAt-nc}$$

$$\frac{\mathsf{lookup}_{G\downarrow_{EH}}(\sigma, pc_l, id) = \phi \qquad \mathsf{valOf}(\phi) \neq \mathsf{NULL} \qquad \mathsf{labOf}(\phi, pc_l) = l \qquad \mathcal{C} = (\phi.M(Ev) \downarrow_{pc_l}) \sqcup pc_l \sqcup l}{\mathsf{lookupEHAll}_G(\sigma, pc_l, id.Ev(v)) = \mathsf{mergeC}(\mathcal{C})}\ \textsc{lookupEHAll}$$

$$\frac{\mathsf{lookup}_{G\downarrow_{EH}}(\sigma, pc, id) = \phi \qquad \mathsf{valOf}(\phi) = \mathsf{NULL}}{\mathsf{lookupEHAll}_G(\sigma, pc_l, id.Ev(v)) = \cdot}\ \textsc{lookupEHAll-s}$$

$$\frac{\mathsf{lookupEHAll}_G(\sigma, H, id.Ev(v)) = \mathcal{C}_H \qquad \mathsf{lookupEHAll}_G(\sigma, L, id.Ev(v)) = \mathcal{C}_L}{\mathsf{lookupEHAll}_{G,\mathcal{V}}(\sigma, \cdot, id.Ev(v)) = \mathsf{mergeC}(\mathcal{C}_H, \mathcal{C}_L)}\ \textsc{lookupEHAll-nc-merge}$$

$$\frac{\mathcal{V} \neq \mathsf{TT} \qquad \mathcal{C} = \mathsf{lookupEHAt}_G(\sigma, pc, id.Ev(v))}{\mathsf{lookupEHs}_{G,\mathcal{V}}(\sigma, pc, id.Ev(v)) = \mathcal{C}} \qquad \frac{pc \sqsubseteq L \qquad \mathcal{C} = \mathsf{lookupEHAll}_G(\sigma, \cdot, id.Ev(v))}{\mathsf{lookupEHs}_{G,\mathsf{TT}}(\sigma, pc, id.Ev(v)) = \mathcal{C}}$$

$$\frac{pc \not\sqsubseteq L \qquad \mathcal{C} = \mathsf{lookupEHAll}_G(\sigma, H, id.Ev(v))}{\mathsf{lookupEHs}_{G,\mathsf{TT}}(\sigma, pc, id.Ev(v)) = \mathcal{C}}$$

mergeC merges all of the event handlers so that any duplicates will have label $\cdot$. If the event handler in $\mathcal{C}$ is unique, it keeps its label. If there is another event handler in $\mathcal{C}$, we use the operator $\wedge$ to compute the new label:

$$\frac{}{l \wedge l = l} \qquad \frac{l \neq l'}{l \wedge l' = \cdot}$$

$$\frac{(eh, l') \notin \mathcal{C}}{\mathsf{mergeC}((eh, l), \mathcal{C}) = (eh, l), \mathsf{mergeC}(\mathcal{C})} \qquad \frac{\mathcal{C} = (\mathcal{C}', (eh, l'), \mathcal{C}'')}{\mathsf{mergeC}((eh, l), \mathcal{C}) = \mathsf{mergeC}((eh, l \wedge l'), \mathcal{C}', \mathcal{C}'')}$$

lookupEHAt performs a join because the @ operation returns everything with label $l \in \{pc, \cdot\}$, but we want to run them at pc. This is similar for all enforcement mechanisms, except SME because it has two executions.

lookupEHAll returns all event handlers visible at that pc and runs them at the join of their label with the pc. This differs for every enforcement mechanism when $pc = \cdot$ because SME has multiple executions, and MF, TT only need results merged sometimes.

lookupEHs returns the event handlers for a locally triggered event. Except for TT, this is the event handlers at pc. For TT, this is *all* event handlers, run at $pc \sqcup l$ (where l is the label of the value) joined with the label of the event handler. (Note that the pc passed to lookupEHs is the triggering pc joined with the label of the value.) So, when $pc \sqcup l \sqsubseteq L$, trigger all event handlers and run them at the label of the event handler. Otherwise, trigger all event handlers at H.

mergeC merges any duplicate event handlers and runs them at the correct security level. SME does not require merging. MF maps duplicate events to the no-context security level. TT runs duplicate events at L since L can output to both L and H channels.

$\boxed{\mathsf{output}(\mathcal{P}, ch(v), pc) = \alpha_l}$

$$\frac{}{\mathsf{output}(\mathcal{P}, ch(v^{\mathsf{std}}), pc_l) = (ch(v^{\mathsf{std}}), pc_l)} \qquad \frac{}{\mathsf{output}(\mathcal{P}, ch(v^{\mathsf{std}}), \cdot) = (ch(v^{\mathsf{std}}), \mathcal{P}(ch))}$$

$$\frac{v = \langle \_ | \_ \rangle \qquad \mathcal{P}(ch) = H}{\mathsf{output}(\mathcal{P}, ch(v), \cdot) = \langle ch(\mathsf{getFacet}(v, H)) | \bullet \rangle} \qquad \frac{v = \langle \_ | \_ \rangle \qquad \mathcal{P}(ch) = L}{\mathsf{output}(\mathcal{P}, ch(v), \cdot) = \langle \bullet | ch(\mathsf{getFacet}(v, L)) \rangle}$$

$\boxed{\mathsf{outputCondition}_{\mathcal{V}}(\mathcal{P}, ch(v), pc) = b}$

$$\frac{\mathcal{P}(ch) = pc}{\mathsf{outCondition}_{\mathsf{SME}}(\mathcal{P}, ch(v), pc) = \mathsf{true}} \qquad \frac{\mathcal{P}(ch) \neq pc}{\mathsf{outCondition}_{\mathsf{SME}}(\mathcal{P}, ch(v), pc) = \mathsf{false}}$$

$$\frac{\mathcal{P}(ch) = pc_l}{\mathsf{outCondition}_{\mathsf{MF}}(\mathcal{P}, ch(v), pc_l) = \mathsf{true}} \qquad \frac{\mathcal{P}(ch) \neq pc_l}{\mathsf{outCondition}_{\mathsf{MF}}(\mathcal{P}, ch(v), pc_l) = \mathsf{false}}$$

$$\frac{v \downarrow_{\mathcal{P}(ch)} \neq \cdot}{\mathsf{outCondition}_{\mathsf{MF}}(\mathcal{P}, ch(v), \cdot) = \mathsf{true}} \qquad \frac{v \downarrow_{\mathcal{P}(ch)} = \cdot}{\mathsf{outCondition}_{\mathsf{MF}}(\mathcal{P}, ch(v), \cdot) = \mathsf{false}}$$

$$\frac{pc \sqcup l \sqsubseteq \mathcal{P}(ch)}{\mathsf{outCondition}_{\mathsf{TT}}(\mathcal{P}, ch((v, l)), pc) = \mathsf{true}} \qquad \frac{pc \sqcup l \not\sqsubseteq \mathcal{P}(ch)}{\mathsf{outCondition}_{\mathsf{TT}}(\mathcal{P}, ch((v, l)), pc) = \mathsf{false}}$$

Output and output conditions. Outputting a standard value is straightforward: if the pc is a standard label (i.e., L or H), label the output with the pc. Otherwise (i.e., $pc = \cdot$), label the output with the label on the channel. If the value is faceted, output a faceted action. The facet matching the label of the channel will contain the matching facet of the value. The other facet will contain $\bullet$.

The output condition for SME event handlers is that the pc matches the label on the channel ($\mathcal{P}(ch)$) since SME is only allowed to output to channels matching the execution. Similarly for MF, if the pc is a standard label (i.e., L or H), the label on the channel must match the pc for the output to succeed. If the $pc = \cdot$, then we check that the value being output is visible to the channel (i.e., $v \downarrow_{\mathcal{P}(ch)} \neq \cdot$). If it is, the output succeeds, otherwise, the output fails. Finally, for TT, outputs succeed if the data is visible to the channel. This means that the data itself as well as the context the data was generated in should both be visible to the channel (i.e., $pc \sqcup l \sqsubseteq \mathcal{P}(ch)$).

$\boxed{G, \mathcal{V}, d \vdash \sigma_1^G, \kappa \xrightarrow{\alpha}_{pc} \sigma_2^G, ks}$

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

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

$$\frac{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c_1^{\mathsf{std}} \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2, c_2^{\mathsf{std}}, E_2}{G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \sigma_1, c_1^{\mathsf{std}}, P, E_1 \xrightarrow{\alpha}_{pc} \sigma_2^G, (\mathcal{V}; (\sigma_2, c_2^{\mathsf{std}}, P, (E_1, E_2)); pc)} \text{ P}$$

C.2. EH queue semantics

The mid-level semantics are of the form: $G, \mathcal{P}, \mathcal{V} \vdash \sigma_1^G, \kappa \xrightarrow{\alpha}_{pc} \sigma_2^G, ks$ and run a single event handler κ with the given enforcement mechanism $\mathcal{V}$ and produce some output α.

While the current event handler is running (i.e., the system is in producer state, $s = P$), P allows the event handler to make progress. When the current event handler has finished (i.e., the next command is skip), if there are any locally-simulated events to process (i.e., the event queue is not empty, $E \neq \cdot$), LC switches the current event handler to consumer state (i.e., $s = C$ for the configuration at the top of the stack) and looks up the appropriate event handlers for the local events and adds them to the configuration stack. If the current event handler has finished and there are not any locally-simulated events (i.e., the event queue is empty $E = \cdot$), PToC switches the current event handler to consumer state.

$\boxed{G, \mathcal{V}, d \vdash \sigma_1^G, \kappa^{\mathcal{V}} \xrightarrow{\alpha}_{pc} \sigma_2^G, ks}$

$$\frac{G, \mathcal{P}, \mathsf{SME}, d \vdash \sigma_1^G, \kappa_L \xrightarrow{\alpha}_L \sigma_2^G, (\mathsf{SME}; \kappa_L'; L) :: ks \qquad \neg\mathsf{consumer}(\kappa_L')}{G, \mathcal{P}, \mathsf{SME}, d \vdash \sigma_1^G, \kappa_H; \kappa_L \xrightarrow{\alpha}_L \sigma_2^G, (\mathsf{SME}; (\kappa_H; \kappa_L'); L) :: ks} \text{ SME-L}$$

$$\frac{G, \mathcal{P}, \mathsf{SME}, d \vdash \sigma_1^G, \kappa_L \xrightarrow{\alpha}_L \sigma_2^G, (\mathsf{SME}; \kappa_L'; L) :: ks \qquad \mathsf{consumer}(\kappa_L')}{G, \mathcal{P}, \mathsf{SME}, d \vdash \sigma_1^G, \kappa_H; \kappa_L \xrightarrow{\alpha}_L \sigma_2^G, (\mathsf{SME}; (\kappa_H; \kappa_L'); H) :: ks} \text{ SME-LToH}$$

$$\frac{G, \mathcal{P}, \mathsf{SME}, d \vdash \sigma_1^G, \kappa_H \xrightarrow{\alpha}_H \sigma_2^G, (\mathsf{SME}; \kappa_H'; H) :: ks}{G, \mathcal{P}, \mathsf{SME}, d \vdash \sigma_1^G, \kappa_H; \kappa_L \xrightarrow{\alpha}_H \sigma_2^G, (\mathsf{SME}; (\kappa_H'; \kappa_L); H) :: ks} \text{ SME-H}$$

$$\frac{G, \mathsf{MF}, d \Vdash \sigma_1^G, \sigma_1, c^{\mathsf{std}} \xrightarrow{\alpha}_{\cdot} \sigma_2^G, \sigma_2, \langle c_H | c_L \rangle, E_2}{G, \mathcal{P}, \mathsf{MF}, d \vdash \sigma_1^G, \sigma_1, c^{\mathsf{std}}, P, E_1 \xrightarrow{\alpha}_{\cdot} \sigma_2^G, (\mathsf{MF}; (\sigma_2, c_L, P, (E_1, E_2)); L) :: (\mathsf{MF}; (\sigma_2, c_H, P, (E_1, E_2)); H)} \text{ P-F}$$

There are a few additional rules for processing some enforcement mechanisms. Recall that SME has multiple executions. The pc tells us which execution to run. We run the L execution first. SME-L and SME-LToH run the L execution. If the system is still in producer state after taking a step ($\neg\mathsf{consumer}(\kappa_L')$ as in SME-L), the pc remains L to continue running the L execution. Otherwise, the low execution is in consumer state ($\mathsf{consumer}(\kappa_L')$ as in SME-LToH) and the pc switches to H to run the H execution. SME-H runs the H execution.

P-F handles the case where MF produces a faceted command. In this case, we split the execution to run the L command with $pc = L$ and the H command with $pc = H$. Note about MF semantics: similar to the original faceted execution semantics, we split execution whenever we see faceted values, but unlike the semantics these rules are based on, *we never go back to a joint execution*. Once the execution splits, it remains split until the event handler finishes execution. We do this because the more efficient semantics are more complex and since this paper is not focused on performance (it is focused on security), we opt for simpler semantics over better performance.

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

$$\frac{}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{skip}; c \xrightarrow{\bullet}_{pc} \sigma^G, \sigma^{\mathcal{V}}, c, \cdot}\ \text{SKIP} \qquad \frac{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1 \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2^{\mathcal{V}}, c_1', E}{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1; c_2 \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2^{\mathcal{V}}, c_1'; c_2, E}\ \text{SEQ}$$

$$\frac{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{pc}^{\mathcal{V}} v \qquad \mathsf{valOf}(v) = \mathsf{true}}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \xrightarrow{\bullet}_{pc} \sigma^G, \sigma^{\mathcal{V}}, c_1, \cdot}\ \text{IF-TRUE}$$

$$\frac{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{pc}^{\mathcal{V}} v \qquad \mathsf{valOf}(v) = \mathsf{false}}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \xrightarrow{\bullet}_{pc} \sigma^G, \sigma^{\mathcal{V}}, c_2, \cdot}\ \text{IF-FALSE}$$

$$\frac{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{pc}^{\mathcal{V}} v \qquad \mathsf{valOf}(v) = \mathsf{true}}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{while}\ e\ \mathsf{do}\ c \xrightarrow{\bullet}_{pc} \sigma^G, \sigma^{\mathcal{V}}, c; \mathsf{while}\ e\ \mathsf{do}\ c, \cdot}\ \text{WHILE-TRUE}$$

$$\frac{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{pc}^{\mathcal{V}} v \qquad \mathsf{valOf}(v) = \mathsf{false}}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{while}\ e\ \mathsf{do}\ c \xrightarrow{\bullet}_{pc} \sigma^G, \sigma^{\mathcal{V}}, \mathsf{skip}, \cdot}\ \text{WHILE-FALSE}$$

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

$$\frac{\mathsf{read}(d, \iota) = v}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, x := \mathsf{declassify}(\iota, e) \xrightarrow{\mathsf{declassify}(\iota, v)}_{L} \sigma^G, \sigma^{\mathcal{V}}, x := v, \cdot}\ \text{DECLASSIFY-L}$$

$$\frac{}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, x := \mathsf{declassify}(\iota, e) \xrightarrow{\bullet}_{H} \sigma^G, \sigma^{\mathcal{V}}, x := e, \cdot}\ \text{DECLASSIFY-H}$$

C.3. EH semantics

The lower-level semantic rules for evaluating individual event handlers are triggered by the mid-level semantics in the "producer" state. The judgement for these rules is $G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1^{\mathsf{std}} \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2^{\mathcal{V}}, c_2, E$. These rules are mostly standard and enforcement-independent, except for interactions with the store. Note: these rules are meant to be general enough to apply to any enforcement mechanism, which is why valOf appears in most rules.

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

$$\frac{G, \mathsf{MF}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1 \xrightarrow{\alpha}_{\cdot} \sigma_2^G, \sigma_2^{\mathcal{V}}, \langle c_H | c_L \rangle, E}{G, \mathsf{MF}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1; c_2 \xrightarrow{\alpha}_{\cdot} \sigma_2^G, \sigma_2^{\mathcal{V}}, \mathsf{setFacetC}(c_H; c_2, c_L; c_2), E}\ \text{SEQ-F}$$

$$\frac{\begin{array}{c} G, \mathsf{MF}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{\cdot}^{\mathsf{MF}} v = \langle \_ | \_ \rangle \qquad v_H = \mathsf{getFacetV}(v, H) \qquad v_L = \mathsf{getFacetV}(v, L) \\ c_H = c_1\ \textit{if}\ v_H = \mathsf{true} \qquad c_H = c_2\ \textit{if}\ v_H = \mathsf{false} \\ c_L = c_1\ \textit{if}\ v_L = \mathsf{true} \qquad c_L = c_2\ \textit{if}\ v_L = \mathsf{false} \end{array}}{G, \mathsf{MF}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \xrightarrow{\bullet}_{\cdot} \sigma^G, \sigma^{\mathcal{V}}, \mathsf{setFacetC}(c_H, c_L), \cdot}\ \text{IF-F}$$

$$\frac{\begin{array}{c} G, \mathsf{MF}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{\cdot}^{\mathsf{MF}} v = \langle \_ | \_ \rangle \qquad v_H = \mathsf{getFacetV}(v, H) \qquad v_L = \mathsf{getFacetV}(v, L) \\ c_H = c; \mathsf{while}\ e\ \mathsf{do}\ c\ \textit{if}\ v_H = \mathsf{true} \qquad c_H = \mathsf{skip}\ \textit{if}\ v_H = \mathsf{false} \\ c_L = c; \mathsf{while}\ e\ \mathsf{do}\ c\ \textit{if}\ v_L = \mathsf{true} \qquad c_L = \mathsf{skip}\ \textit{if}\ v_L = \mathsf{false} \end{array}}{G, \mathsf{MF}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{while}\ e\ \mathsf{do}\ c \xrightarrow{\bullet}_{\cdot} \sigma^G, \sigma^{\mathcal{V}}, \mathsf{setFacetC}(c_H, c_L), \cdot}\ \text{WHILE-F}$$

$$\frac{\mathsf{read}(d, \iota) = v}{G, \mathsf{MF}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, x := \mathsf{declassify}(\iota, e) \xrightarrow{\mathsf{declassify}(\iota, v)}_{\cdot} \sigma^G, \sigma^{\mathcal{V}}, \mathsf{setFacetC}(x := e, x := v), \cdot}\ \text{DECLASSIFY-NC}$$

Faceted semantics. There are a few additional rules for dealing with faceted values. SEQ-F handles the case where the first command in a sequence $c_1; c_2$ steps to a faceted command ($\langle c_H | c_L \rangle$). We use setFacetC to create a new faceted command where the H facet is the sequence $c_H; c_2$ and the L facet is the sequence $c_L; c_2$. IF-F handles branching on a faceted value and WHILE-F handles looping on a faceted value. In both cases, we evaluate the conditional for both facets and produce a faceted command where the H facet is the command resulting from evaluating the H facet of the

condition, and the likewise for the L facet. Finally, when evaluating a declassification with $pc = \cdot$, we create a faceted command where the command in the H facet performs the assignment directly, and the L facet assigns the declassified value.

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

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

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

$\boxed{\mathsf{assign}_i(\sigma, pc, x, v) = \sigma'}$

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

$$\frac{\sigma = \mathsf{getStore}(\sigma^G \downarrow_g, pc_l) \qquad x \in \sigma \qquad \sigma' = \sigma[x \mapsto v]}{\mathsf{assign}_{\mathsf{SMS}}(\sigma^G, pc_l, x, v) = \mathsf{setStoreVar}(\sigma^G, pc_l, \sigma')} \text{ SMS-ASSIGN}$$

$$\frac{\sigma = \mathsf{getStore}(\sigma^G \downarrow_g, pc_l) \qquad x \notin \sigma}{\mathsf{assign}_{\mathsf{SMS}}(\sigma^G, pc_l, x, v) = \sigma^G} \text{ SMS-ASSIGN-S}$$

$$\frac{\sigma' = \sigma[x \mapsto v]}{\mathsf{assign}_{\mathsf{MF}}(\sigma, \cdot, x, v) = \sigma'} \text{ MF-ASSIGN} \qquad \frac{v_L = \mathsf{var}_{\mathsf{MF}}(\sigma, L, x) \qquad v' = \mathsf{setFacetV}(\mathsf{getFacetV}(v, H), v_L)}{\mathsf{assign}_{\mathsf{MF}}(\sigma, H, x, v) = \sigma[x \mapsto v']} \text{ MF-ASSIGN-H}$$

$$\frac{v_H = \mathsf{var}_{\mathsf{MF}}(\sigma, H, x) \qquad v' = \mathsf{setFacetV}(v_H, \mathsf{getFacetV}(v, L))}{\mathsf{assign}_{\mathsf{MF}}(\sigma, L, x, v) = \sigma[x \mapsto v']} \text{ MF-ASSIGN-L}$$

$$\frac{x \in \sigma \qquad \sigma' = \sigma[x \mapsto v]}{\mathsf{assign}_{\mathsf{FS}}(\sigma, \cdot, x, v) = \sigma'} \text{ FS-ASSIGN} \qquad \frac{\begin{array}{c} x \in \sigma \qquad v_L = \mathsf{var}_{\mathsf{FS}}(\sigma, L, x) \\ v' = \mathsf{setFacetV}(\mathsf{getFacetV}(v, H), v_L) \end{array}}{\mathsf{assign}_{\mathsf{FS}}(\sigma, H, x, v) = \sigma[x \mapsto v']} \text{ FS-ASSIGN-H}$$

$$\frac{\begin{array}{c} x \in \sigma \qquad v_H = \mathsf{var}_{\mathsf{FS}}(\sigma, H, x) \\ v' = \mathsf{setFacetV}(v_H, \mathsf{getFacetV}(v, L)) \end{array}}{\mathsf{assign}_{\mathsf{FS}}(\sigma, L, x, v) = \sigma[x \mapsto v']} \text{ FS-ASSIGN-L} \qquad \frac{x \notin \sigma}{\mathsf{assign}_{\mathsf{FS}}(\sigma, pc, x, v) = \sigma} \text{ FS-ASSIGN-S}$$

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

$$\frac{x \notin \sigma}{\mathsf{assign}_{\mathsf{TS}}(\sigma, pc, x, (v, l)) = \sigma} \text{ TS-ASSIGN-S}$$

Variable assignment. The rules for variable assignment involve a function assign which performs the actual assignment and whose behavior differs depending on the enforcement mechanism.

For SME, configurations keep track of each copy of the store with the execution, so assignments can be made directly. On the other hand, to update a variable in an SMS store, we first pick the correct copy of the store using getStore, which is the copy that matches the given pc. We make the assignment in this store, then update the SMS store using setStoreVar to include the updated copy.

Assignments for MF and FS are very similar, except that $x \in \sigma$ is a requirement for FS. If the variable does not exist in the store, the assignment is skipped. Assignments when $pc = \cdot$ are straightforward. For a standard pc (i.e., L or H), we first create a faceted value which combines the old value from the store with the value being assigned. If the new value is faceted, getFacetV will get the correct facet and setFacetV will get the correct facet from the old value. The facet matching the pc comes from the new value, and the other facet comes from the old value: this is the value assigned to the store.

Assignments for TT and TS are similar, except that $x \in \sigma$ is a requirement for TS. If the variable does not exist in the store, the assignment is skipped. The value being assigned is labeled $l \sqcup pc$ so that neither the value nor the context it is assigned in cause any leaks.

$\boxed{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow^i_{pc} v}$

$$\frac{\text{if } x \in \sigma^G, \text{then } v = \mathsf{var}_{G\downarrow_g}(\sigma^G, pc, x) \qquad \text{otherwise}, v = \mathsf{var}_{\mathcal{V}}(\sigma^{\mathcal{V}}, pc, x) \qquad \mathsf{toDst}(v, pc, i) = v'}{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash x \Downarrow^i_{pc} v'} \text{ VAR}$$

$$\frac{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e_1 \Downarrow^i_{pc} v_1 \qquad G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e_2 \Downarrow^i_{pc} v_2 \qquad v = v_1 \text{ bop } v_2}{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e_1 \text{ bop } e_2 \Downarrow^i_{pc} v} \text{ BOP}$$

C.4. Expression semantics

Note that the sub-expressions are converted to the same types in each rule. So, for BOP, this means that types will not be mixed (e.g. (v, l) bop $\langle v_H | v_L \rangle$ is not possible). To keep top-level rules as separate as possible from mechanism-specific details (like the structure of the values), we assume that bop well-defined for the various types of values (i.e. standard, labeled, and faceted).

$\boxed{\mathsf{ehAPIe}(\mathcal{G}, \sigma, pc, id, v_1, \cdots, v_n) = v}$

$$\frac{\begin{array}{c}\mathsf{ehAPIe}_{\mathcal{G}}(\sigma, H, id, \mathsf{getFacetV}(v_1, H), \cdots, \mathsf{getFacetV}(v_n, H)) = v_H \\ \mathsf{ehAPIe}_{\mathcal{G}}(\sigma, L, id, \mathsf{getFacetV}(v_1, L), \cdots, \mathsf{getFacetV}(v_n, L)) = v_L\end{array}}{\mathsf{ehAPIe}(\mathcal{G}, \sigma, \cdot, id, v_1, \cdots, v_n) = \mathsf{createFct}(v_H, v_L)} \text{ EHAPI-NC}$$

$$\frac{\mathsf{ehAPIe}_{\mathcal{G}}(\sigma, pc_l, id, v_1, \cdots, v_n) = v}{\mathsf{ehAPIe}(\mathcal{G}, \sigma, pc_l, id, v_1, \cdots, v_n) = v} \text{ EHAPI}$$

Create facet (with conversion):

$$\frac{\mathsf{labOf}(v_L, L) \sqsubseteq L}{\mathsf{createFct}(v_H, v_L) = \mathsf{setFacetV}(\mathsf{valOf}(v_H), \mathsf{valOf}(v_L))} \qquad \frac{\mathsf{labOf}(v_L, L) \not\sqsubseteq L}{\begin{array}{c}\mathsf{createFct}(v_H, v_L) = \\ \mathsf{setFacetV}(\mathsf{valOf}(v_H), \mathsf{dv})\end{array}}$$

Unstructured EH storage. These rules handle any special cases at the interface between different enforcement mechanisms. The rules are similar for unstructured and tree-structure EH storage semantics; we describe the unstructured EH storage rules here and the tree-structured rules next. Since the expression semantics convert the types of the attributes to the format expected by $\mathcal{G}$, we only need to consider the case where the no-context pc may be used. In this case, we might see faceted values even for non-faceted stores (otherwise, we could lose some detail). We use EHAPI-NC to split the context and compute both the H and L cases to make the result as accurate as possible. After computing both the H and L case, we combine the results using createFct. Other than this case, the EH storage functions are called directly.

$\boxed{\mathsf{ehAPIe}(\mathcal{G}, \sigma, pc, \cdots) = v}$

$$\frac{\begin{array}{c}\mathsf{ehAPIe}_{\mathcal{G}}(\sigma, H, \mathsf{getFacetV}(v_1, H), \cdots, \mathsf{getFacetV}(v_n, H)) = v_H \\ \mathsf{ehAPIe}_{\mathcal{G}}(\sigma, L, \mathsf{getFacetV}(v_1, L), \cdots, \mathsf{getFacetV}(v_n, L)) = v_L\end{array}}{\mathsf{ehAPIe}(\mathcal{G}, \sigma, \cdot, v_1, \cdots, v_n) = \mathsf{createFct}(v_H, v_L)} \text{ EHAPI-NC}$$

$$\frac{\mathsf{ehAPIe}_{\mathcal{G}}(\sigma, pc_l, v_1, \cdots, v_n) = v}{\mathsf{ehAPIe}(\mathcal{G}, \sigma, pc_l, v_1, \cdots, v_n) = v} \text{ EHAPI}$$

Create facet (with conversion):

$$\frac{\mathsf{labOf}(v_L, L) \sqsubseteq L}{\mathsf{createFct}(v_H, v_L) = \mathsf{setFacetV}(\mathsf{valOf}(v_H), \mathsf{valOf}(v_L))} \qquad \frac{\mathsf{labOf}(v_L, L) \not\sqsubseteq L}{\begin{array}{c}\mathsf{createFct}(v_H, v_L) = \\ \mathsf{setFacetV}(\mathsf{valOf}(v_H), \mathsf{dv})\end{array}}$$

Tree-structured EH storage. The rules for the tree-structured EH storage are similar except that the EH nodes are passed by reference rather than passing an id.

$$\frac{dst \in \{\mathsf{SME}, \mathsf{SMS}, \mathsf{MF}, \mathsf{FS}\}}{\mathsf{toDst}(v^{\mathsf{std}}, pc, dst) = v^{\mathsf{std}}}\ \text{MS} \qquad \frac{dst \in \{\mathsf{TT}, \mathsf{TS}\} \qquad pc \sqsubseteq L}{\mathsf{toDst}(v^{\mathsf{std}}, pc, dst) = (v^{\mathsf{std}}, L)}\ \text{MSToT-L}$$

$$\frac{dst \in \{\mathsf{TT}, \mathsf{TS}\} \qquad pc \not\sqsubseteq L}{\mathsf{toDst}(v^{\mathsf{std}}, pc, dst) = (v^{\mathsf{std}}, pc)}\ \text{MSToT-H} \qquad \frac{dst \in \{\mathsf{TT}, \mathsf{TS}\}}{\mathsf{toDst}((v^{\mathsf{std}}, l), pc_l, dst) = (v^{\mathsf{std}}, l \sqcup pc_l)}\ \text{T}$$

$$\frac{dst \notin \{\mathsf{TT}, \mathsf{TS}\} \qquad l \sqsubseteq pc_l}{\mathsf{toDst}((v^{\mathsf{std}}, l), pc_l, dst) = v^{\mathsf{std}}}\ \text{ToMS} \qquad \frac{dst \notin \{\mathsf{TT}, \mathsf{TS}\} \qquad l \not\sqsubseteq pc_l}{\mathsf{toDst}((v^{\mathsf{std}}, l), pc_l, dst) = \mathsf{dv}}\ \text{ToMS-DV}$$

$$\frac{}{\mathsf{toDst}((v^{\mathsf{std}}, L), \cdot, dst) = v^{\mathsf{std}}}\ \text{ToMS-NC-L} \qquad \frac{}{\mathsf{toDst}((v^{\mathsf{std}}, H), \cdot, dst) = \langle v^{\mathsf{std}} | \mathsf{dv} \rangle}\ \text{ToMS-NC-H}$$

$$\frac{v = \langle \_ | \_ \rangle}{\mathsf{toDst}(v, \cdot, dst) = v}\ \text{NC}$$

Type conversion rules. Because the shared variable storage, EH storage, and event handlers may all be enforced differently from each other, computations involving different values from different enforcement mechanisms may need to be converted to a different format. We include the destination enforcement mechanism (i) in the expression semantics and convert the final result to whatever format is expected by this enforcement mechanism.

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. This is handled by variable lookup rules, which are described in the next section.

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 (rule ToMS). 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 (rule ToMS-DV). 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 (rule T). The exception to this is when the pc is $\cdot$. If the value is not tainted, we return the (un-labeled) value (rule ToMS-NC-L). If the value is tainted, we create a faceted value with the (un-labeled) value in the H facet and a default value in the L facet (rule ToMS-NC-H).

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. Rule MS converts a standard value to a multi-storage format (which is also a standard value). If the pc is $\cdot$, a faceted value does not need to be converted since $pc = \cdot$ only for MF and FS (rule NC). For taint tracking techniques, the context is also used to determine the final label on the data. Rule MSToT-L adds the label L to a standard value computed in a public context and MSToT-H adds the label H to a standard value computed in a secret context.

Binary operations, variable lookup. Binary operations are straightforward, except when they involve facets or tainted values. Recall from our expression semantics that values in sub-expressions will be converted to the same format (i.e., binary operations won't mix faceted values and labeled values). To perform a binary operation involving a faceted value (rule BOP-FACET), split the faceted value(s) on the H and L facets, separately, then combine the results into a faceted value. To perform a binary operation on two labeled (tainted) values (rule BOP-LABEL), perform the operation on the values and assign the result a new label that is the join of the labels on the original values.

Variable lookup rules for SME are straightforward. Because each copy of the execution maintains its own store, we simply look up the variable in the store (rule SME-VAR). If it is not found, we return a default value (rule SME-VAR-DV). Looking up variables in SMS is similar to SME except that we first have to pick the correct copy of the store which matches the pc context. We do this using getStore (rule SMS-VAR). If the variable is not found in the store, we return a default value (rule SMS-VAR-DV).

Variable lookups for MF and FS stores are the same. If the lookup happens under a "standard" (i.e., L or H) pc, then we use getFacetV to get the appropriate facet from that value in the store (rules MF-VAR and FS-VAR). getFacetV returns the L or H facet of a faceted value (depending on the pc), or the value, unchanged, if it is not faceted (recall that un-faceted values are visible to both L and H observers). Also note that uninitialized L or H facets will be $\cdot$. getFacetV returns dv in this case. If the pc is $\cdot$, we perform the lookup in both the L and H context and create a faceted value with the results using setFacetV (rules MF-VAR-F and FS-VAR-F). We split the execution rather than returning whatever faceted/unfaceted value is in the store for two reasons. (1) this ensures that uninitialized $\cdot$ facets will have dv rather than remaining empty, which is helpful for proofs. (2) it simplifies the semantics, especially if we wanted to replace empty facets with dv without splitting the execution. As before, if the variable is not in the store, we return dv (rules MF-VAR-DV and FS-VAR-DV).

$\boxed{v_1^{\mathsf{MF}} \text{ bop } v_2^{\mathsf{MF}}}$

$$\frac{\begin{array}{c} v_1 = \langle \_ | \_ \rangle \quad \text{or} \quad v_2 = \langle \_ | \_ \rangle \\ v_H = (\mathsf{getFacetV}(v_1, H)) \text{ bop } (\mathsf{getFacetV}(v_2, H)) \qquad v_L = (\mathsf{getFacetV}(v_1, L)) \text{ bop } (\mathsf{getFacetV}(v_2, L)) \end{array}}{v_1 \text{ bop } v_2 = \mathsf{setFacetV}(v_H, v_L)} \text{ BOP-FACET}$$

$\boxed{v_1^{\mathsf{TT}} \text{ bop } v_2^{\mathsf{TT}}}$

$$\frac{}{(v, l) \text{ bop } (v', l') = (v \text{ bop } v', l \sqcup l')} \text{ BOP-LABEL}$$

Figure 22: Binary operations

$\boxed{\mathsf{var}_{\mathcal{V}}(\sigma^{\mathcal{V}}, pc, x) = v}$

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

$$\frac{\sigma_l = \mathsf{getStore}(\sigma, pc_l)}{\mathsf{var}_{\mathsf{SMS}}(\sigma, pc_l, x) = \sigma_l(x)} \text{ SMS-VAR} \qquad \frac{\sigma_l = \mathsf{getStore}(\sigma, pc_l) \qquad x \notin \sigma_l}{\mathsf{var}_{\mathsf{SMS}}(\sigma, pc_l, x) = \mathsf{dv}} \text{ SMS-VAR-DV}$$

$$\frac{v = \mathsf{getFacetV}(\sigma(x), pc_l)}{\mathsf{var}_{\mathsf{MF}}(\sigma, pc_l, x) = v} \text{ MF-VAR} \qquad \frac{\begin{array}{c} x \in \sigma \\ v_H = \mathsf{var}_{\mathsf{MF}}(\sigma, H, x) \qquad v_L = \mathsf{var}_{\mathsf{MF}}(\sigma, L, x) \end{array}}{\mathsf{var}_{\mathsf{MF}}(\sigma, \cdot, x) = \mathsf{setFacetV}(v_H, v_L)} \text{ MF-VAR-F}$$

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

$$\frac{v = \mathsf{getFacetV}(\sigma(x), pc_l)}{\mathsf{var}_{\mathsf{FS}}(\sigma, pc_l, x) = v} \text{ FS-VAR} \qquad \frac{x \in \sigma \qquad v_H = \mathsf{var}_{\mathsf{FS}}(\sigma, H, x) \qquad v_L = \mathsf{var}_{\mathsf{FS}}(\sigma, L, x)}{\mathsf{var}_{\mathsf{FS}}(\sigma, \cdot, x) = \mathsf{setFacetV}(v_H, v_L)} \text{ FS-VAR-F}$$

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

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

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

Figure 23: Variable lookup rules

The rules for TT and TS are mostly straightforward. If the variable is in the store, we simply return the value (rules TT-VAR and TS-VAR). If not, we always return a default value with label H, regardless of the pc (rules TT-VAR-DV and TS-VAR-DV). If we labeled the default value with the pc, we would leak something to the attacker if the variable did exist in the store and was tainted (because (dv, L) is distinguishable from (v, H)). To hide the possibility of the variable holding a secret, we always taint the default value.

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

$$\frac{G, \mathcal{V}, \sigma_1^G, \sigma_1^{\mathcal{V}} \vdash e \Downarrow_{pc}^{G\downarrow_{EH}} v \qquad \mathsf{assign}_{G\downarrow_{EH}}(\sigma_1^G, pc, id, v) = \sigma_2^G}{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, id := e \xrightarrow{\bullet}_{pc} \sigma_2^G, \sigma_1^{\mathcal{V}}, \mathsf{skip}, \cdot} \text{ ASSIGN-D}$$

$$\frac{G, \mathcal{V}, \sigma_1^G, \sigma_1^{\mathcal{V}} \vdash e \Downarrow_{pc}^{G\downarrow_{EH}} v \qquad \sigma_2^G = \mathsf{createElem}_{G\downarrow_{EH}}(\sigma_1^G, pc, id, v)}{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, \mathsf{create}(id, e) \xrightarrow{\bullet}_{pc} \sigma_2^G, \sigma_1^{\mathcal{V}}, \mathsf{skip}, \cdot} \text{ CREATEELEM}$$

$$\frac{\sigma_2^G = \mathsf{registerEH}_{G\downarrow_{EH}}(\sigma_1^G, pc, id, eh)}{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, \mathsf{register}(id, eh) \xrightarrow{\bullet}_{pc} \sigma_2^G, \sigma_1^{\mathcal{V}}, \mathsf{skip}, \cdot} \text{ ADDEH}$$

$$\frac{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{pc}^{G\downarrow_{EH}} v \qquad E = \mathsf{triggerEH}_{G\downarrow_{EH}}(\sigma^G, pc, id, Ev, v)}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{trigger}(id, Ev, e) \xrightarrow{\bullet}_{pc} \sigma^G, \sigma^{\mathcal{V}}, \mathsf{skip}, E} \text{ TRIGGER}$$

Figure 24: Shared Unstructured EH storage command semantics

$$\frac{G, \mathcal{V}, \sigma_1^G, \sigma_1^{\mathcal{V}} \vdash e \Downarrow_{pc}^{G\downarrow_{EH}} v \qquad \mathsf{assign}_{G\downarrow_{EH}}(\sigma_1^G, pc, a, v) = \sigma_2^G}{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, a := e \xrightarrow{\bullet}_{pc} \sigma_2^G, \sigma_1^{\mathcal{V}}, \mathsf{skip}, \cdot} \text{ ASSIGN-D}$$

$$\frac{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow_{pc}^{G\downarrow_{EH}} v \qquad E = \mathsf{triggerEH}_{G\downarrow_{EH}}(\sigma^G, pc, id, Ev, v)}{G, \mathcal{V}, d \Vdash \sigma^G, \sigma^{\mathcal{V}}, \mathsf{trigger}(id, Ev, e) \xrightarrow{\bullet}_{pc} \sigma^G, \sigma^{\mathcal{V}}, \mathsf{skip}, E} \text{ TRIGGER}$$

$$\frac{G, \mathcal{V}, \sigma_1^G, \sigma_1^{\mathcal{V}} \vdash e \Downarrow_{pc}^{G\downarrow_{EH}} v \qquad \sigma_2^G = \mathsf{createChild}_{G\downarrow_{EH}}(\sigma_1^G, pc, id, a_p, v)}{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, \mathsf{createChild}(id, a_p, e) \xrightarrow{\bullet}_{pc} \sigma_2^G, \sigma_1^{\mathcal{V}}, \mathsf{skip}, \cdot} \text{ CREATECHILD}$$

$$\frac{G, \mathcal{V}, \sigma_1^G, \sigma_1^{\mathcal{V}} \vdash e \Downarrow_{pc}^{G\downarrow_{EH}} v \qquad \sigma_2^G = \mathsf{createSibling}_{G\downarrow_{EH}}(\sigma_1^G, pc, id, a_s, v)}{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, \mathsf{createSibling}(id, a_s, e) \xrightarrow{\bullet}_{pc} \sigma_2^G, \sigma_1^{\mathcal{V}}, \mathsf{skip}, \cdot} \text{ CREATESIBLING}$$

$$\frac{\sigma_2^G = \mathsf{registerEH}_{G\downarrow_{EH}}(\sigma_1^G, pc, a, eh)}{G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, \mathsf{register}(a, eh) \xrightarrow{\bullet}_{pc} \sigma_2^G, \sigma_1^{\mathcal{V}}, \mathsf{skip}, \cdot} \text{ ADDEH}$$

Figure 25: Shared Tree-structured EH storage command semantics

C.5. EH Storage semantics

Unstructured EH Storage command semantics. Here we describe the commands for interacting with the event handler storage. Note: these rules are meant to be general enough to apply to any enforcement mechanism, which is why labOf appears in most rules.

ASSIGN-D updates the attribute for a node with id id. CREATEELEM adds an empty node with id id and assigns value determined by expression e. ADDEH registers a new event handler (eh) to a node given by id. TRIGGER triggers the event handlers for the event Ev associated with a node (given by id) with parameter given by e. TRIGGER-T is necessary for taint tracking because taint tracking assumes that the pc determines what security level to run the event handlers at, not which event handlers to run.

Each of these rules uses a helper function with the type of enforcement ($G\downarrow_{EH}$) as one of the parameters. These functions will be described later in this section.

Tree-structured EH Storage command semantics. Note: these rules are meant to be general enough to apply to any enforcement mechanism, which is why labOf appears in most rules.

TRIGGER triggers the event handlers for the event Ev associated with a node (given by id) with parameter given by e. TRIGGER-T is necessary for taint tracking because taint tracking assumes that the pc determines what security level to run the event handlers at, not which event handlers to run. CREATECHILD adds an empty node as the left-most child of the node at location a_p. The new node has id id and value given by expression e. The parent of the new node is located at a_p. If a_p is NULL, the store is unchanged. CREATESIBLING adds an empty node as the right-hand sibling of the node at location a_s. The new node has id id and value given by expression e. The parent of the new node is the

parent of a_s. If a_s or the parent of a_s is NULL, the store is unchanged. ADDEH registers a new event handler (eh) to a node at address a.

Each of these rules uses a helper function with the type of enforcement ($G \downarrow_{EH}$) as one of the parameters. These functions will be described next.

$$\boxed{\mathsf{lookup}_{\mathcal{G}}(\sigma, pc, id) = \phi}$$

$$\frac{\sigma' = \mathsf{getStore}_{\mathsf{SMS}}(\sigma, pc_l) \qquad \sigma'(id) = \phi}{\mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) = \phi}\ \text{SMS-LOOKUP} \qquad \frac{\sigma' = \mathsf{getStore}_{\mathsf{SMS}}(\sigma, pc_l) \qquad id \notin \sigma'}{\mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) = \mathsf{NULL}}\ \text{SMS-LOOKUP-S}$$

$$\frac{\sigma(id) = \phi \qquad \phi.v \downarrow_{pc_l} \neq \cdot}{\mathsf{lookup}_{\mathsf{FS}}(\sigma, pc_l, id) = \phi}\ \text{FS-LOOKUP} \qquad \frac{id \notin \sigma \qquad \text{or} \qquad \sigma(id).v \downarrow_{pc_l} = \cdot}{\mathsf{lookup}_{\mathsf{FS}}(\sigma, pc_l, id) = \mathsf{NULL}}\ \text{FS-LOOKUP-S}$$

$$\frac{\phi_H = \mathsf{lookup}_{\mathsf{FS}}(\sigma, H, id) \qquad \phi_L = \mathsf{lookup}_{\mathsf{FS}}(\sigma, L, id)}{\mathsf{lookup}_{\mathsf{FS}}(\sigma, \cdot, id) = \mathsf{setFacetN}(\phi_H, \phi_L)}\ \text{FS-LOOKUP-F}$$

$$\frac{\sigma(id) = \phi}{\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc, id) = \phi}\ \text{TS-LOOKUP} \qquad \frac{id \notin \sigma}{\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc, id) = (\mathsf{NULL}, H)}\ \text{TS-LOOKUP-S}$$

Figure 26: Rules for looking up a node in the unstructured EH storage

Unstructured EH Storage command semantics (helper functions). The rules in this section connect the framework to the specific enforcement mechanisms protecting the shared EH storage. For each helper function, there is a set of rules for each enforcement mechanism (SMS, FS, TS).

Node lookup. Figure 26 shows the rules for looking up a node in the unstructured EH storage. For the SMS store, this is straightforward. Rule SMS-LOOKUP handles the case where the node is in the store, and rule SMS-LOOKUP-S handles the case where a node with matching id doesn't exist, in which case we return NULL. We use the pc to determine which copy of the EH storage to use for the lookup.

Recall that for the faceted store, nodes may contain faceted values if they depend on a secret. The FS store also has to check if the node is initialized in the context given by the pc. To do this, rule FS-LOOKUP checks if the attribute in the node is initialized (i.e., whether $\phi.v \downarrow_{pc} \neq \cdot$). If a node with matching id does not exist in the store, or if the node has not been initialized in the given context, rule FS-LOOKUP-S returns NULL. If the $pc = \cdot$, we split the execution and perform the lookup in both the L and H contexts. We create a facet with the results using setFacetN.

The rules for looking up a node in the TS store are straightforward. If a node with matching id exists, TS-LOOKUP is used to return the node directly. Otherwise, TS-LOOKUP-S returns a tainted NULL. We always label NULL as secret in this case, regardless of the pc because we want it to be indistinguishable from a secret node (in case one exists in an equivalent store).

Node attribute update. Figure 27 shows the rules for updating the attribute of a node. The SMS rules are similar to the rules for looking up a node. SMS-ASSIGNEH handles the case where a node with matching id, SMS-ASSIGNEH-S handles the case where there isn't. SMS-ASSIGNEH-NC handles the case where the $pc = \cdot$. Here, we split the context and make assignments to the L and H stores separately.

The rules for FS attribute updates directly replace the attribute in the store if the $pc = \cdot$ and a node with matching id is found in the store (rule FS-ASSIGNEH) and do not change the store if no matching node is found or if the node is not initialized in the given context (rule FS-ASSIGNEH-S). For a standard pc (i.e., it is L or H), we update the attribute by using updateFacet (rule FS-ASSIGNEH-UPD). If the node is faceted (i.e., $\mathsf{lookup}_{\mathsf{FS}}$ returns a faceted value), then the execution splits and performs the assignment in both the L and H context (rule FS-ASSIGNEH-NC).

The rules for the TS store are straightforward. If a node with matching id is found, we update the attribute to be the label on the new value joined with the pc and the label on the node (rule TS-ASSIGNEH). If a node with a matching id is not found, the store is not changed (rule TS-ASSIGNEH-S).

Triggering an event handler. Figure 28 contains rules for an event handler to trigger another event handler. For SMS, SMS-TRIGGEREH handles the case where a node with matching id exists in the store. In this case, the event is triggered in the given context (either L or H). If a node with matching id does not exist, SMS-TRIGGEREH-S does not trigger any events. When the $pc = \cdot$, SMS-TRIGGEREH-NC splits the context to look up the event in both copies of the store. If multiple events are generated, they are merged using mergeEvs (Section B.2).

Similarly, the rules for the FS store check if a node with matching id exists. If it does, FS-TRIGGEREH triggers a new event in the given context. If it doesn't, FS-TRIGGEREH-S does not trigger any events. If the node or argument

$\boxed{\mathsf{assign}_{\mathcal{G}}(\sigma, pc, id, v) = \sigma'}$

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

$$\frac{\sigma = \mathsf{getStore}_{\mathsf{SMS}}(\sigma^G \downarrow_{EH}, pc_l) \qquad id \notin \sigma}{\mathsf{assign}_{\mathsf{SMS}}(\sigma^G, pc_l, id, v) = \sigma^G} \text{ SMS-ASSIGNEH-S}$$

$$\frac{\sigma_1^G = \mathsf{assign}_{\mathsf{SMS}}(\sigma^G, H, id, \mathsf{getFacet}(v, H)) \qquad \sigma_2^G = \mathsf{assign}_{\mathsf{SMS}}(\sigma_1^G, L, id, \mathsf{getFacet}(v, L))}{\mathsf{assign}_{\mathsf{SMS}}(\sigma^G, \cdot, id, v) = \sigma_2^G} \text{ SMS-ASSIGNEH-NC}$$

$$\frac{(v', M) = \mathsf{lookup}_{\mathsf{FS}}(\sigma, \cdot, id)}{\mathsf{assign}_{\mathsf{FS}}(\sigma, \cdot, id, v) = \sigma[id \mapsto (v, M)]} \text{ FS-ASSIGNEH} \qquad \frac{\mathsf{NULL} = \mathsf{lookup}_{\mathsf{FS}}(\sigma, pc, id)}{\mathsf{assign}_{\mathsf{FS}}(\sigma, pc, id, v) = \sigma} \text{ FS-ASSIGNEH-S}$$

$$\frac{\begin{array}{c}(v', M) = \mathsf{lookup}_{\mathsf{FS}}(\sigma, pc_l, id) \\ v'' = \mathsf{updateFacet}(v', v, pc_l)\end{array}}{\mathsf{assign}_{\mathsf{FS}}(\sigma, pc_l, id, v) = \sigma[id \mapsto (v'', M)]} \text{ FS-ASSIGNEH-UPD}$$

$$\frac{\begin{array}{c}\phi = \mathsf{lookup}_{\mathsf{FS}}(\sigma, \cdot, id) = \langle \_ | \_ \rangle \\ \sigma' = \mathsf{assign}_{\mathsf{FS}}(\sigma, H, id, \mathsf{getFacet}(v, H)) \qquad \sigma'' = \mathsf{assign}_{\mathsf{FS}}(\sigma', L, id, \mathsf{getFacet}(v, L))\end{array}}{\mathsf{assign}_{\mathsf{FS}}(\sigma, \cdot, id, v) = \sigma''} \text{ FS-ASSIGNEH-NC}$$

$$\frac{(v', M, l') = \sigma(id)}{\mathsf{assign}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = \sigma[id \mapsto (id, (v, l \sqcup pc \sqcup l'), M, l')]} \text{ TS-ASSIGNEH}$$

$$\frac{id \notin \sigma}{\mathsf{assign}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = \sigma} \text{ TS-ASSIGNEH-S}$$

Figure 27: Rules for updating the attribute of a node in the unstructured EH storage

to the triggered event is faceted, then the execution splits in FS-TRIGGEREH-NC and triggers the event in both the H and L context. The resulting events are merged using mergeEvs.

Again, for the TS store, we first check if a node with matching id exists. If it does, TS-TRIGGEREH triggers a new event in the context given by the pc joined with the label on the node joined with the label on the argument to the event. If a node with matching id does not exist, TS-TRIGGEREH-S does not trigger any events. If the $pc = \cdot$, then TS-TRIGGEREH-NC splits the execution to trigger the event in both the L and H context.

Creating a new node. Figure 29 contains rules for creating a new EH node. To create a new node in the SMS store, we first check if a node with the given id exists. If it does not, SMS-CREATE creates a new node with the given attribute and no registered event handlers. The appropriate copy of the store is retrieved with getStore and updated with setStore. If a node with the given id already exists, SMS-CREATE-U updates the node with the given attribute using $\mathsf{assign}_{\mathsf{SMS}}$. If the $pc = \cdot$, SMS-CREATE-NC splits the execution to create the node in both copies of the store.

Creating a new node in the FS store is straightforward. First we check if a node with the given id exists. If it does not, FS-CREATE adds a new node to the store with the given attribute. The attribute is formatted using createFacet to ensure that the attribute only exists in the given context. We also use getFacetV on the given attribute, to make the proofs easier. FS-CREATE-U is used in case a node with the given id does already exist. In that case the attribute on the existing node is updated using $\mathsf{assign}_{\mathsf{FS}}$.

To create a new node in the TS store, we again first check if a node with the given id exists. If it does not, TS-CREATE adds a new node to the store with the given attribute and label given by the context. If a node with the given id does exist, we check the label on the node (given by l'). If the node is visible in the current context (i.e., $l' \sqsubseteq pc$), TS-CREATE-U1 updates the attribute. The new attribute has the label given by joining the label on the node, pc, and the label on the attribute. If the node is not visible in the current context (i.e., $l' \not\sqsubseteq pc$), TS-CREATE-U2 updates the attribute and the label on the node. The new attribute has the label given by joining the pc and the label on the attribute. The new label on the nodes is the pc. We update the label on the node because leaving the old label on the node would mean the node is still not visible in the current context, despite the node being created. This would leak to the attacker that a secret node with the given id existed. Finally, if the $pc = \cdot$ TS-CREATE-NC creates the node in

$\boxed{\mathsf{triggerEH}_{\mathcal{G}}(\sigma, pc, id, Ev, v) = E}$

$$\frac{\mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) = \phi \neq \mathsf{NULL}}{\mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, pc_l id, Ev, v) = (id.Ev(v), pc_l)}\ \textsc{sms-triggerEH}$$

$$\frac{\phi = \mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) = \mathsf{NULL}}{\mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, pc_l, id, Ev, v) = \cdot}\ \textsc{sms-triggerEH-s}$$

$$\frac{\begin{array}{c}E_H = \mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, H, id, Ev, \mathsf{getFacet}(v, H)) \\ E_L = \mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, L, id, Ev, \mathsf{getFacet}(v, L))\end{array}}{\mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, \cdot, id, Ev, v) = \mathsf{mergeEvs}(E_H, E_L)}\ \textsc{sms-triggerEH-nc}$$

$$\frac{\phi = \mathsf{lookup}_{\mathsf{FS}}(\sigma, pc, id) \neq \langle \_ | \_ \rangle \neq \mathsf{NULL}}{\mathsf{triggerEH}_{\mathsf{FS}}(\sigma, pc, id, Ev, v^{\mathsf{std}}) = (id.Ev(v), pc)}\ \textsc{fs-triggerEH}$$

$$\frac{\begin{array}{c}\langle \_ | \_ \rangle = \mathsf{lookup}_{\mathsf{FS}}(\sigma, pc, id) \vee v = \langle \_ | \_ \rangle \\ E_H = \mathsf{triggerEH}_{\mathsf{FS}}(\sigma, H, id, Ev, \mathsf{getFacet}(v, H)) \\ E_L = \mathsf{triggerEH}_{\mathsf{FS}}(\sigma, L, id, Ev, \mathsf{getFacet}(v, L))\end{array}}{\mathsf{triggerEH}_{\mathsf{FS}}(\sigma, \cdot, id, Ev, v) = \mathsf{mergeEvs}(E_H, E_L)}\ \textsc{fs-triggerEH-nc}$$

$$\frac{\mathsf{lookup}_{\mathsf{FS}}(\sigma, pc, id) = \mathsf{NULL}}{\mathsf{triggerEH}_{\mathsf{FS}}(\sigma, pc, id, Ev, v^{\mathsf{std}}) = \cdot}\ \textsc{fs-triggerEH-s}$$

$$\frac{\mathsf{lookup}_{\mathcal{G}}(\sigma, pc_l, id) = (\_, \_, l_\phi) \qquad l = \mathsf{labOf}(v, pc_l)}{\mathsf{triggerEH}_{\mathcal{G}}(\sigma, pc_l, id, Ev, v) = (id.Ev(v), pc_l \sqcup l_\phi \sqcup l)}\ \textsc{ts-triggerEH}$$

$$\frac{\begin{array}{c}E_H = \mathsf{triggerEH}_{\mathcal{G}}(\sigma, H, id, Ev, \mathsf{getFacetV}(v, H)) \\ E_L = \mathsf{triggerEH}_{\mathcal{G}}(\sigma, L, id, Ev, \mathsf{getFacetV}(v, L))\end{array}}{\mathsf{triggerEH}_{\mathcal{G}}(\sigma, \cdot, id, Ev, v) = \mathsf{mergeEvs}(E_H, E_L)}\ \textsc{ts-triggerEH-nc}$$

$$\frac{\phi = \mathsf{lookup}_{\mathcal{G}}(\sigma, pc_l, id) = (\mathsf{NULL}, \_)}{\mathsf{triggerEH}_{\mathcal{G}}(\sigma, pc_l, id, Ev, v) = \cdot}\ \textsc{ts-triggerEH-s}$$

Figure 28: Rules for triggering an event in the unstructured EH storage

the L context. This is different from the other stores (which split the context and create the node twice) because the given attribute is already formatted as a labeled value. If we were to create the node twice, it would create the node, then overwrite the attribute with the same value anyway. Creating the node just once in the L context is more efficient and does not leak anything because the existence of the node is not secret (since the $pc = \cdot$).

Registering a new event handler. Figure 30 contains rules for registering a new event handler. To register a new event handler in the SMS store, we first look up the node with the given id. If the node exists, SMS-REGISTEREH adds the new event handler to the event handler map with the label given by the pc. If a node with the given id does not exist, SMS-REGISTEREH-S does not change the store. If the $pc = \cdot$, SMS-REGISTEREH-NC splits the execution and adds the event handler in both the L and H context.

The rules for adding a new event handler to the FS store are similar. If a node with the given id exists, FS-REGISTEREH updates the event handler map with the label given by the pc. If it does not exist, FS-REIGSTEREH-S does not change the store. If the node with matching id is faceted, REGISTEREH-NC splits the execution to add the event handler to both nodes in the facet.

The rules for adding a new event handler to the TS store are straightforward and are similar to the ones for the SMS and FS stores. TS-REGISTEREH updates the node with matching id with the label given by the pc joined with the label on the node. TS-REGISTEREH-S handles the case where a node with matching id cannot be found and the store is not changed. If the $pc = \cdot$, TS-REGISTEREH-NC splits the execution and adds the event handler in both the L and H context.

$\boxed{\mathsf{createElem}_{\mathcal{G}}(\sigma_1^G, pc, id, v) = \sigma_2^G}$

$$\frac{\mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) = \mathsf{NULL} \qquad \phi = (v^{\mathsf{std}}, \cdot) \qquad \sigma' = \mathsf{getStore}(\sigma, pc_l)}{\mathsf{createElem}_{\mathsf{SMS}}(\sigma, pc_l, id, v) = \mathsf{setStore}(\sigma, pc_l, \sigma'[id \mapsto \phi])} \text{ SMS-CREATE}$$

$$\frac{\mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc, id) = \phi \neq \mathsf{NULL} \qquad \sigma' = \mathsf{assign}_{\mathsf{SMS}}(\sigma, pc, id, v)}{\mathsf{createElem}_{\mathsf{SMS}}(\sigma, pc_l, id, v) = \sigma'} \text{ SMS-CREATE-U}$$

$$\frac{\sigma' = \mathsf{createElem}_{\mathsf{SMS}}(\sigma, H, id, v) \qquad \sigma'' = \mathsf{createElem}_{\mathsf{SMS}}(\sigma', L, id, v)}{\mathsf{createElem}_{\mathsf{SMS}}(\sigma, \cdot, id, v) = \sigma''} \text{ SMS-CREATE-NC}$$

$$\frac{\mathsf{lookup}_{\mathsf{FS}}(\sigma, \cdot, id) = \mathsf{NULL} \qquad v' = \mathsf{getFacetV}(v, pc) \qquad \phi = (\mathsf{createFacet}(v', pc), \cdot)}{\mathsf{createElem}_{\mathsf{FS}}(\sigma, pc, id, v) = \sigma[id \mapsto \phi]} \text{ FS-CREATE}$$

$$\frac{\mathsf{lookup}_{\mathsf{FS}}(\sigma, \cdot, id) = \phi \neq \mathsf{NULL} \qquad \sigma' = \mathsf{assign}_{\mathsf{FS}}(\sigma, pc, id, v)}{\mathsf{createElem}_{\mathsf{FS}}(\sigma, pc, id, v) = \sigma'} \text{ FS-CREATE-U}$$

$$\frac{\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) = (\mathsf{NULL}, \_) \qquad \phi = (v, \cdot, pc_l)}{\mathsf{createElem}_{\mathsf{TS}}(\sigma, pc_l, id, v) = \sigma[id \mapsto \phi]} \text{ TS-CREATE}$$

$$\frac{\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc, id) = (v', M, l') \qquad l' \sqsubseteq pc \qquad \sigma' = \sigma[id \mapsto ((v, l \sqcup pc \sqcup l'), M, l')]}{\mathsf{createElem}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = \sigma'} \text{ TS-CREATE-U1}$$

$$\frac{\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc, id) = (v', M, l') \qquad l' \not\sqsubseteq pc \qquad \sigma' = \sigma[id \mapsto ((v, l \sqcup pc), M, pc)]}{\mathsf{createElem}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = \sigma'} \text{ TS-CREATE-U2}$$

$$\frac{\sigma' = \mathsf{createElem}_{\mathsf{TS}}(\sigma, L, id, v)}{\mathsf{createElem}_{\mathsf{TS}}(\sigma, \cdot, id, v) = \sigma'} \text{ TS-CREATE-NC}$$

Figure 29: Rules for creating a new node in the unstructured EH storage

$\boxed{\mathsf{registerEH}_{\mathcal{G}}(\sigma_1^G, pc, id, eh) = \sigma_2^G}$

$$\frac{(v, M) = \mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) \quad eh = \mathsf{on}Ev(x)\{c\} \quad M' = M[Ev \mapsto M(Ev) \cup \{(eh, pc_l)\}] \quad \sigma' = \mathsf{getStore}(\sigma, pc_l)}{\mathsf{registerEH}_{\mathsf{SMS}}(\sigma, pc_l, id, eh) = \mathsf{setStore}(\sigma, pc_l, \sigma'[id \mapsto (v, M')])} \text{ SMS-REGISTEREH}$$

$$\frac{\mathsf{NULL} = \mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id)}{\mathsf{registerEH}_{\mathsf{SMS}}(\sigma, pc_l, id, eh) = \sigma} \text{ SMS-REGISTEREH-S}$$

$$\frac{\sigma' = \mathsf{registerEH}_{\mathsf{SMS}}(\sigma, H, id, eh) \quad \sigma'' = \mathsf{registerEH}_{\mathsf{SMS}}(\sigma', L, id, eh)}{\mathsf{registerEH}_{\mathsf{SMS}}(\sigma, \cdot, id, eh) = \sigma''} \text{ SMS-REGISTEREH-NC}$$

$$\frac{(v, M) = \mathsf{lookup}_{\mathsf{FS}}(\sigma, pc, id) \quad eh = \mathsf{on}Ev(x)\{c\} \quad M' = M[Ev \mapsto M(Ev) \cup \{(eh, pc)\}]}{\mathsf{registerEH}_{\mathsf{FS}}(\sigma, pc, id, eh) = \sigma[id \mapsto (v, M')]} \text{ FS-REGISTEREH}$$

$$\frac{\langle \phi_H | \phi_L \rangle = \mathsf{lookup}_{\mathsf{FS}}(\sigma, pc, id) \quad \sigma' = \mathsf{registerEH}_{\mathsf{FS}}(\sigma, H, id, eh) \quad \sigma'' = \mathsf{registerEH}_{\mathsf{FS}}(\sigma', L, id, eh)}{\mathsf{registerEH}_{\mathsf{FS}}(\sigma, \cdot, id, eh) = \sigma''} \text{ FS-REGISTEREH-NC}$$

$$\frac{\mathsf{NULL} = \mathsf{lookup}_{\mathsf{FS}}(\sigma, pc, id)}{\mathsf{registerEH}_{\mathsf{FS}}(\sigma, pc, id, eh) = \sigma} \text{ FS-REGISTEREH-S}$$

$$\frac{(v, M, l) = \mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) \quad eh = \mathsf{on}Ev(x)\{c\} \quad M' = M[Ev \mapsto M(Ev) \cup \{(eh, pc_l \sqcup l)\}]}{\mathsf{registerEH}_{\mathsf{TS}}(\sigma, pc_l, id, eh) = \sigma[id \mapsto (v, M', l)]} \text{ TS-REGISTEREH}$$

$$\frac{\sigma' = \mathsf{registerEH}_{\mathsf{TS}}(\sigma, H, id, eh) \quad \sigma'' = \mathsf{registerEH}_{\mathsf{TS}}(\sigma', L, id, eh)}{\mathsf{registerEH}_{\mathsf{TS}}(\sigma, \cdot, id, eh) = \sigma''} \text{ TS-REGISTEREH-NC}$$

$$\frac{\phi = \mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) \quad \mathsf{valOf}(\phi) = \mathsf{NULL}}{\mathsf{registerEH}_{\mathsf{TS}}(\sigma, pc_l, id, eh) = \sigma} \text{ TS-REGISTEREH-S}$$

Figure 30: Rules for registering a new event handler in the unstructured EH storage

$\boxed{\mathsf{lookupA}_{\mathcal{G}}(\sigma^{\mathcal{G}}, pc, id, A) = a}$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \qquad \sigma'(a).id = id}{\mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc_l, id, (a :: A)) = a}\ \textsc{sms-lookupa}$$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \qquad \sigma'(a).id \neq id \qquad \mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc_l, id, (A :: \sigma'(a).A)) = a}{\mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc_l, id, (a :: A)) = a}\ \textsc{sms-lookupa-r}$$

$$\frac{}{\mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc_l, id, \cdot) = \mathsf{NULL}}\ \textsc{sms-lookupa-s}$$

$$\frac{\sigma(a).id = id \qquad \phi.v \downarrow_{pc_l} \neq \cdot}{\mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc_l, id, (a :: A)) = a}\ \textsc{fs-lookupa} \qquad \frac{\sigma(a).id \neq id \qquad \sigma(a).v \downarrow_{pc_l} \neq \cdot \quad \mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc_l, id, (A :: \sigma(a).A \downarrow_{pc_l})) = a'}{\mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc_l, id, (a :: A)) = a'}\ \textsc{fs-lookupa-r1}$$

$$\frac{\sigma(a).v \downarrow_{pc_l} = \cdot \quad \mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc_l, id, A) = a'}{\mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc_l, id, (a :: A)) = a'}\ \textsc{fs-lookupa-r2}$$

$$\frac{a_H = \mathsf{lookupA}_{\mathsf{FS}}(\sigma, H, id, A \downarrow_H) \qquad a_L = \mathsf{lookupA}_{\mathsf{FS}}(\sigma, L, id, A \downarrow_L)}{\mathsf{lookupA}_{\mathsf{FS}}(\sigma, \cdot, id, A) = \mathsf{setFacetA}(a_H, a_L)}\ \textsc{fs-lookupa-f}$$

$$\frac{}{\mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc_l, id, \cdot) = \mathsf{NULL}}\ \textsc{fs-lookupa-s}$$

Figure 31: Rules for looking up the address of a node in the tree-structured EH storage

$\boxed{\mathsf{lookup}_{\mathcal{G}}(\sigma^{\mathcal{G}}, pc_l, id, A) = \phi}$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \qquad \mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc_l, id, a^{\mathsf{rt}}) = a \neq \mathsf{NULL} \qquad \sigma'(a) = \phi}{\mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) = \phi}\ \textsc{sms-lookup}$$

$$\frac{\mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc_l, id, a^{\mathsf{rt}}) = \mathsf{NULL}}{\mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) = \mathsf{NULL}}\ \textsc{sms-lookup-s}$$

$$\frac{\mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc_l, id, a^{\mathsf{rt}}) = a \neq \mathsf{NULL} \qquad \sigma(a) = \phi}{\mathsf{lookup}_{\mathsf{FS}}(\sigma, pc_l, id) = \phi}\ \textsc{fs-lookup}$$

$$\frac{\mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc_l, id, a^{\mathsf{rt}}) = \mathsf{NULL}}{\mathsf{lookup}_{\mathsf{FS}}(\sigma, pc_l, id) = \mathsf{NULL}}\ \textsc{fs-lookup-s}$$

Figure 32: Rules for looking up a node in the tree-structured EH storage

Tree-structured EH Storage command semantics (helper functions). The rules in this section connect the framework to the specific enforcement mechanisms protecting the shared EH storage. For each helper function, there is a set of rules for each enforcement mechanism (SMS, FS). Note that because we prove the unstructured TS store only satisfies weak secrecy, we do not formalize the tree-structured TS store.

Node address lookup. Figure 31 contains rules for looking up the address of a node. lookupA looks up the address of a node by traversing the EH tree store recursively. The biggest difference between SMS and FS is that the structure of the nodes is different.

To traverse the SMS tree, we check if the "current" node matches the given id. We use getStore to check the appropriate copy of the store. SMS-LOOKUPA handles the case where the id matches, so we return the address of the current node. SMS-LOOKUPA-R handles the case where the current node does not match the given id. In this case, we add the current node's children to the list of nodes to check and we check the rest of the nodes in the list. Finally, once we run out of nodes to check, SMS-LOOKUPA-S returns NULL.

Traversing the FS tree is similar. FS-LOOKUPA handles the case where the current node matches the given id. We

$\boxed{\mathsf{assign}_{\mathcal{G}}(\sigma_1^{\mathcal{G}}, pc, a, v) = \sigma_2^{\mathcal{G}}}$

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

$$\frac{}{\mathsf{assign}_{\mathsf{SMS}}(\sigma, pc_l, \mathsf{NULL}, v) = \sigma}\ \textsc{sms-assignEH-s}$$

$$\frac{\begin{array}{l}\sigma' = \mathsf{assign}_{\mathsf{SMS}}(\sigma, H, \mathsf{getFacet}(a, H), \mathsf{getFacet}(v, H)) \\ \sigma'' = \mathsf{assign}_{\mathsf{SMS}}(\sigma', L, \mathsf{getFacet}(a, L), \mathsf{getFacet}(v, L))\end{array}}{\mathsf{assign}_{\mathsf{SMS}}(\sigma, \cdot, a, v) = \sigma''}\ \textsc{sms-assignEH-nc}$$

$$\frac{(id, v', M, a_p, A) = \sigma(a)}{\mathsf{assign}_{\mathsf{FS}}(\sigma, \cdot, a, v) = \sigma[a \mapsto (id, v, M, a_p, A)]}\ \textsc{fs-assignEH} \qquad \frac{}{\mathsf{assign}_{\mathsf{FS}}(\sigma, pc_l, \mathsf{NULL}, v) = \sigma}\ \textsc{fs-assignEH-s}$$

$$\frac{\mathsf{assign}_{\mathsf{FS}}(\sigma, H, a_H, \mathsf{getFacetV}(v, H)) = \sigma' \qquad \mathsf{assign}_{\mathsf{FS}}(\sigma, L, a_L, \mathsf{getFacetV}(v, L)) = \sigma''}{\mathsf{assign}_{\mathsf{FS}}(\sigma, \cdot, \langle a_H | a_L \rangle, v) = \sigma''}\ \textsc{fs-assignEH-nc}$$

$$\frac{(id, v', M, a_p, A) = \sigma(a) \qquad v'' = \mathsf{updateFacet}(v', v, pc_l)}{\mathsf{assign}_{\mathsf{FS}}(\sigma, pc_l, a, v) = \sigma[a \mapsto (id, v'', M, a_p, A)]}\ \textsc{fs-assignEH-upd}$$

Figure 33: Rules for updating the attribute of a node in the tree-structured EH storage

also have to check that the node is visible in the given context (i.e., $\phi.v \downarrow_{pc_l} \neq \cdot$). If the current node does not match the given id, but it is visible in the given context, FS-LOOKUPA-R1 adds the current node's children (that are visible in the given context, $A \downarrow_{pc_l}$) to the recursive call. If the current node does not match the given id, and is not visible in the given context, FS-LOOKUPA-R2 checks the rest of the nodes in the list but does not add the children of the current node. If $pc = \cdot$, FS-LOOKUPA-F splits the execution to look for the address of the node in both the L and H contexts. The result is made into a faceted address using setFacetA. Finally, if a node with matching id does not exist, FS-LOOKUPA-S returns NULL.

Node lookup. Figure 32 contains rules for looking up nodes by id. This function works by looking up the address of the node with matching id and then returning the node stored at that address.

For the SMS store, SMS-LOOKUP uses $\mathsf{lookupA}_{\mathsf{SMS}}$ to get the address of the node. If the address is not NULL, we look up the node using the address in the copy of the store returned by getStore. If the address is NULL, SMS-LOOKUP-S returns NULL.

Similarly, for FS, FS-LOOKUP returns the node given by the address from $\mathsf{lookupA}_{\mathsf{FS}}$. If the address is NULL, FS-LOOKUP-S returns NULL. Note that lookup is only called with a standard label (i.e., L or H) so we don't have a rule for $pc = \cdot$.

Node attribute update. Figure 33 contains rules for updating the attribute of a node. One of the arguments to this function is the address of the node being updated, so the node does not need to be looked up.

To update the attribute of an SMS node, SMS-ASSIGNEH first looks up the appropriate copy of the store using getStore. The attribute in the node is updated, and the store is changed using setStore. If the address of the node to be updated is NULL, SMS-ASSIGNEH-S leaves the store unchanged. If the $pc = \cdot$, we split the execution and update the attribute in both copies of the store.

Note for FS we assume that we only have the address of valid nodes, visible at this context. To update the attribute of an FS node, when $pc = \cdot$ FS-ASSIGNEH updates the attribute directly. If the $pc \neq \cdot$ (i.e., it is L or H), FS-ASSIGNEH-UPD updates the attribute by using updateFacet which only changes the appropriate facet of the attribute. If the address is NULL, FS-ASSIGNEH-S leaves the store unchanged. If the address is faceted, FS-ASSIGNEH-NC splits the execution and performs the assignment to both addresses.

Triggering an event handler. Figure 34 contains rules for triggering an event handler. Unlike some of the other functions, this one takes the id of the node so it does need to be looked up.

To trigger an event handler in the SMS store, we first look up the node with matching id. If one exists (i.e., the address of the node is not NULL), then SMS-TRIGGEREH returns an event triggered in the given context. If a node with matching id does not exist (i.e., the address of the node is NULL), then SMS-TRIGGEREH-S does not trigger any events. If the $pc = \cdot$, then SMS-TRIGGEREH-NC splits the execution and we trigger the event in both the L and H copy of the store. Finally, if more than one event is triggered, we merge them using mergeEvs (Section B.2).

$\boxed{\mathsf{triggerEH}_{\mathcal{G}}(\sigma^{\mathcal{G}}, pc, id, Ev, v) = E}$

$$\frac{\mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc_l, id, a^{\mathsf{rt}}) \neq \mathsf{NULL}}{\mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, pc_l, id, Ev, v) = (id.Ev(v), pc)}\ \textsc{sms-triggereh}$$

$$\frac{\mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc_l, id, a^{\mathsf{rt}}) = \mathsf{NULL}}{\mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, pc_l, id, Ev, v) = \cdot}\ \textsc{sms-triggereh-s}$$

$$\frac{\begin{array}{c}E_H = \mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, H, id, Ev, \mathsf{getFacet}(v, H)) \\ E_L = \mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, L, id, Ev, \mathsf{getFacet}(v, L))\end{array}}{\mathsf{triggerEH}_{\mathsf{SMS}}(\sigma, \cdot, id, Ev, v) = \mathsf{mergeEvs}(E_H, E_L)}\ \textsc{sms-triggereh-nc}$$

$$\frac{a = \mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc, id, a^{\mathsf{rt}}) \neq \langle \_|\_ \rangle \qquad a \neq \mathsf{NULL}}{\mathsf{triggerEH}_{\mathsf{FS}}(\sigma, pc, id, Ev, v^{\mathsf{std}}) = (id.Ev(v), pc)}\ \textsc{fs-triggereh}$$

$$\frac{\begin{array}{c}\langle \_|\_ \rangle = \mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc, id, a^{\mathsf{rt}}) \vee v = \langle \_|\_ \rangle \\ E_H = \mathsf{triggerEH}_{\mathsf{FS}}(\sigma, H, id, Ev, \mathsf{getFacetV}(v, H)) \\ E_L = \mathsf{triggerEH}_{\mathsf{FS}}(\sigma, L, id, Ev, \mathsf{getFacetV}(v, L))\end{array}}{\mathsf{triggerEH}_{\mathsf{FS}}(\sigma, \cdot, id, Ev, v) = \mathsf{mergeEvs}(E_H, E_L)}\ \textsc{fs-triggereh-nc}$$

$$\frac{\mathsf{lookupA}_{\mathsf{FS}}(\sigma, pc, id, a^{\mathsf{rt}}) = \mathsf{NULL}}{\mathsf{triggerEH}_{\mathsf{FS}}(\sigma, pc, id, Ev, v^{\mathsf{std}}) = \cdot}\ \textsc{fs-triggereh-s}$$

Figure 34: Rules for triggering the event handler in the tree-structured EH storage

Triggering an event in the FS store is similar. If a node with matching id exists in the store, FS-TRIGGEREH triggers an event in the given context. If the node lookup produces a faceted value, FS-TRIGGEREH-NC splits the execution to trigger the event in both the L and H context. The resulting events are merged using mergeEvs. If a node with matching id does not exist in the store, FS-TRIGGEREH-S does not produce any event.

Adding a child to a node. Figure 35 contains rules for adding a child to an existing node. One of the arguments to this function is the address of the parent node, so the node does not need to be looked up.

Adding a child to a node in the SMS store is straightforward. SMS-CREATEC looks up the given id to see if the given node already exists. We also look up the parent node. We create a new node with given attribute, add it to the store (at a fresh location), and also add a pointer to the new node to the list of children of the given parent node. If a node with the given id already exists, or if the pointer to the parent node is NULL, SMS-CREATEC-S leaves the store unchanged. If $pc = \cdot$, SMS-CREATEC-NC splits the execution to add the node to both copies of the store.

Adding a node to the FS store is more involved. FS-CREATEC looks up the given id to see if the given node already exists. Note that we use $pc = \cdot$ for the lookup to see if the node exists in any context, not just the given context. If the node does not exist, we also look up the parent node. We create a new node with given (faceted) attribute and (faceted) parent, add it to the store (at a fresh location), and also add a faceted pointer to the new node to the list of children of the given parent node. We create facets using createFacet (Section B.2). The new node will have a faceted attribute and parent because it only exists in the given context, if a node with the same id is added later in a different context, we will update the other facet of the attribute/parent appropriately. If a node with the given id already exists in the given context, the parent node is not visible in the given context, or if the pointer to the parent node is NULL, FS-CREATEC-S1 or FS-CREATEC-S2 (respectively) leaves the store unchanged. If $pc = \cdot$, FS-CREATEC-NC splits the execution to add the node in both contexts.

Recall that we use the $\cdot$ context to see if a node with the given id already exists in the store. We do this to see if the node already exists in the other context so that we can initialize the node in the new context. FS-CREATEC-UL handles the case where the node already exists in the H context and is being added in the L context (and respectively for FS-CREATEC-UH to add the node in the H context). In these cases, we get a faceted node when we look up the node by id and the facet for the context we want to add the node in is NULL (meaning that the node does note exist in that context yet). Instead of adding a new node, we update the existing node. We use updateFacet to update the attribute and pointer to the parent.

Adding a sibling to a node. Figure 36 contains rules for adding a sibling to an existing node. One of the arguments to this function is the address of the sibling node, so the node does not need to be looked up.

Adding a sibling to a node in the SMS store is similar to adding a child. First, SMS-CREATES looks up the given

$\boxed{\mathsf{createChild}_{\mathcal{G}}(\sigma_1^{\mathcal{G}}, pc, id, a_p, v) = \sigma_2^{\mathcal{G}}}$

$$\frac{\begin{array}{c}\mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc_l, id, a^{\mathsf{rt}}) = \mathsf{NULL} \qquad \sigma' = \mathsf{getStore}(\sigma, pc_l) \qquad a \notin \sigma' \\ (id_p, v', M, a'_p, A) = \sigma'(a_p) \\ \sigma'' = \sigma'[a_p \mapsto (id_p, v', M, a'_p, (a :: A))] \qquad \sigma''' = \sigma''[a \mapsto (id, v^{\mathsf{std}}, \cdot, a_p, \cdot)]\end{array}}{\mathsf{createChild}_{\mathsf{SMS}}(\sigma, pc_l, id, a_p, v) = \mathsf{setStore}(\sigma, pc_l, \sigma''')}\ \textsc{sms-createC}$$

$$\frac{\begin{array}{c}\sigma' = \mathsf{createChild}_{\mathsf{SMS}}(\sigma, H, id, \mathsf{getFacetA}(a_p, H), \mathsf{getFacetV}(v, H)) \\ \sigma'' = \mathsf{createChild}_{\mathsf{SMS}}(\sigma', L, id, \mathsf{getFacetA}(a_p, L), \mathsf{getFacetV}(v, L))\end{array}}{\mathsf{createChild}_{\mathsf{SMS}}(\sigma, \cdot, id, a_p, v) = \sigma''}\ \textsc{sms-createC-nc}$$

$$\frac{\mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc_l, id, a^{\mathsf{rt}}) \neq \mathsf{NULL} \vee a_p = \mathsf{NULL}}{\mathsf{createChild}_{\mathsf{SMS}}(\sigma, pc_l, id, a_p, v) = \sigma}\ \textsc{sms-createC-s}$$

$$\frac{\begin{array}{c}\mathsf{lookupA}_{\mathsf{FS}}(\sigma, \cdot, id, a^{\mathsf{rt}}) = \mathsf{NULL} \qquad a \notin \sigma \qquad \sigma(a_p) = (id_p, v_p, M, a'_p, A) \qquad v_p \downarrow_{pc_l} \neq \cdot \\ \sigma' = \sigma[a_p \mapsto (id_p, v_p, M, a'_p, (\mathsf{createFacet}(a, pc_l) :: A))] \\ \sigma'' = \sigma'[a \mapsto (id, \mathsf{createFacet}(v, pc_l), \cdot, \mathsf{createFacet}(a_p, pc_l), \cdot)]\end{array}}{\mathsf{createChild}_{\mathsf{FS}}(\sigma, pc_l, id, a_p, v) = \sigma''}\ \textsc{fs-createC}$$

$$\frac{\begin{array}{c}\mathsf{lookupA}_{\mathsf{FS}}(\sigma, \cdot, id, a^{\mathsf{rt}}) = \langle a | \mathsf{NULL} \rangle \\ \sigma(a) = (id, v', M, a'_p, A) \qquad \sigma(a_p) = (id_p, v_p, M_p, a''_p, A_p) \qquad v_p \downarrow_L \neq \cdot \\ \sigma' = \sigma[a_p \mapsto (id_p, v_p, M_p, a'_p, (\mathsf{createFacet}(a, L) :: A_p))] \\ \sigma'' = \sigma'[a \mapsto (id, \mathsf{updateFacet}(v', v, L), M, \mathsf{updateFacet}(a'_p, a_p, L), A)]\end{array}}{\mathsf{createChild}_{\mathsf{FS}}(\sigma, L, id, a_p, v) = \sigma''}\ \textsc{fs-createC-uL}$$

$$\frac{\begin{array}{c}\mathsf{lookupA}_{\mathsf{FS}}(\sigma, \cdot, id, a^{\mathsf{rt}}) = \langle \mathsf{NULL} | a \rangle \\ \sigma(a) = (id, v', M, a'_p, A) \qquad \sigma(a_p) = (id_p, v_p, M_p, a''_p, A_p) \qquad v_p \downarrow_H \neq \cdot \\ \sigma' = \sigma[a_p \mapsto (id_p, v_p, M_p, a'_p, (\mathsf{createFacet}(a, H) :: A_p))] \\ \sigma'' = \sigma'[a \mapsto (id, \mathsf{updateFacet}(v', v, H), M, \mathsf{updateFacet}(a'_p, a_p, H), A)]\end{array}}{\mathsf{createChild}_{\mathsf{FS}}(\sigma, H, id, a_p, v) = \sigma''}\ \textsc{fs-createC-uH}$$

$$\frac{\begin{array}{c}\sigma' = \mathsf{createChild}_{\mathsf{FS}}(\sigma, H, id, \mathsf{getFacetA}(a_p, H), \mathsf{getFacetV}(v, H)) \\ \sigma'' = \mathsf{createChild}_{\mathsf{FS}}(\sigma, L, id, \mathsf{getFacetA}(a_p, L), \mathsf{getFacetV}(v, L))\end{array}}{\mathsf{createChild}_{\mathsf{FS}}(\sigma, \cdot, id, a_p, v) = \sigma''}\ \textsc{fs-createC-nc}$$

$$\frac{\mathsf{lookupA}_{\mathsf{FS}}(\sigma, \cdot, id, a^{\mathsf{rt}}) = a \qquad a \downarrow_{pc_l} \neq \mathsf{NULL} \vee \sigma(a_p).v \downarrow_{pc_l} = \cdot}{\mathsf{createChild}_{\mathsf{FS}}(\sigma, pc_l, id, a_p, v) = \sigma}\ \textsc{fs-createC-s1}$$

$$\frac{}{\mathsf{createChild}_{\mathsf{FS}}(\sigma, pc_l, id, \mathsf{NULL}, v) = \sigma}\ \textsc{fs-createC-s2}$$

Figure 35: Rules for adding a child to a node in the tree-structured EH storage

id to see if the node already exists, as well as the sibling whose address was given as an argument (a_s) and that node's parent (a_p). A new node is created (at a fresh address) with the given attribute and a_p as its parent. The new node is added to the list of children under the node at a_p. It is placed in the list to the right of a_s. If a node with the given id already exists in the store, or if $a_s = \mathsf{NULL}$, then SMS-CREATES-S1 leaves the store unchanged. Similarly, if $a_p = \mathsf{NULL}$, then SMS-CREATES-S2 leaves the store unchanged. If the $pc = \cdot$, then SMS-CREATES-NC splits the execution to add the node to both the L and H copy of the store.

The rules for adding a sibling to a node in the FS store are similar to the rules for adding a child to a node. FS-CREATES looks up the given id to see if the given node already exists. Like the rules for adding a child to a node, we use $pc = \cdot$ for the lookup to see if the node exists in any context, not just the given context. If the node does not exist, we also look up the sibling node (a_s) and its parent (a_p). We create a new node with given (faceted) attribute and (faceted) parent, add it to the store (at a fresh location), and also add a faceted pointer to the new node to the list of children of the given parent node. The node is added to the right of a_s in the list of children. We create facets using createFacet (Section B.2). The new node will have a faceted attribute and parent because it only exists in the given context, if a node with the same id is added later in a different context, we will update the other facet of the

$\boxed{\mathsf{createSibling}_{\mathcal{G}}(\sigma_1^{\mathcal{G}}, pc, id, a_s, v) = \sigma_2^{\mathcal{G}}}$

$$\frac{\begin{array}{c}\mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc_l, id, a^{\mathsf{rt}}) = \mathsf{NULL} \qquad \sigma' = \mathsf{getStore}(\sigma, pc_l) \qquad a \notin \sigma' \\ \sigma'(a_s).a_p = a_p \qquad (id_p, v', M, a'_p, (A :: a_s :: A')) = \sigma'(a_p) \\ \sigma'' = \sigma'[a_p \mapsto (id_p, v', M, a'_p, (A :: a_s :: a :: A'))] \qquad \sigma''' = \sigma''[a \mapsto (id, v^{\mathsf{std}}, \cdot, a_p, \cdot)]\end{array}}{\mathsf{createSibling}_{\mathsf{SMS}}(\sigma, pc_l, id, a_s, v) = \mathsf{setStore}(\sigma, pc_l, \sigma''')}\ \textsc{sms-createS}$$

$$\frac{\begin{array}{c}\sigma' = \mathsf{createSibling}_{\mathsf{SMS}}(\sigma, H, id, \mathsf{getFacetA}(a_s, H), \mathsf{getFacetV}(v, H)) \\ \sigma'' = \mathsf{createSibling}_{\mathsf{SMS}}(\sigma', L, id, \mathsf{getFacetA}(a_s, L), \mathsf{getFacetV}(v, L))\end{array}}{\mathsf{createSibling}_{\mathsf{SMS}}(\sigma, \cdot, id, a_s, v) = \sigma''}\ \textsc{sms-createS-nc}$$

$$\frac{\mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc, id, a^{\mathsf{rt}}) \neq \mathsf{NULL} \vee a_s = \mathsf{NULL}}{\mathsf{createSibling}_{\mathsf{SMS}}(\sigma, pc_l, id, a_s v) = \sigma}\ \textsc{sms-createS-s1}$$

$$\frac{\begin{array}{c}\mathsf{lookupA}_{\mathsf{SMS}}(\sigma, pc_l, id, a^{\mathsf{rt}}) = \mathsf{NULL} \qquad \sigma' = \mathsf{getStore}(\sigma, pc_l) \\ \sigma'(a_s).a_p = \mathsf{NULL}\end{array}}{\mathsf{createSibling}_{\mathsf{SMS}}(\sigma, pc_l, id, a_s, v) = \sigma}\ \textsc{sms-createS-s2}$$

$$\frac{\begin{array}{c}\mathsf{lookupA}_{\mathsf{FS}}(\sigma, \cdot, id, a^{\mathsf{rt}}) = \mathsf{NULL} \qquad a \notin \sigma \qquad \sigma(a_s).v \downarrow_{pc_l} \neq \cdot \\ \sigma(a_s).a_p \downarrow_{pc_l} = a_p \qquad \sigma(a_p) = (id_p, v', M, a'_p, (A :: a'_s :: A')) \qquad a'_s \downarrow_{pc_l} = a_s \qquad v' \downarrow_{pc_l} \neq \cdot \\ \sigma' = \sigma[a_p \mapsto (id_p, v', M, a'_p, (A :: a'_s :: \mathsf{createFacet}(a, pc_l) :: A'))] \\ \sigma'' = \sigma'[a \mapsto (id, \mathsf{createFacet}(v, pc_l), \cdot, \mathsf{createFacet}(a_p, pc_l), \cdot)]\end{array}}{\mathsf{createSibling}_{\mathsf{FS}}(\sigma, pc_l, id, a_s, v) = \sigma''}\ \textsc{fs-createS}$$

$$\frac{\begin{array}{c}\mathsf{lookupA}_{\mathsf{FS}}(\sigma, \cdot, id, a^{\mathsf{rt}}) = \langle a | \mathsf{NULL} \rangle \\ \sigma(a) = (id, v', M, a'_p, A) \qquad \sigma(a_s).v \downarrow_L \neq \cdot \qquad \sigma(a_s).a_p \downarrow_L = a_p \\ \sigma(a_p) = (id_p, v_p, M_p, a'_p, (A_p :: a'_s :: A'_p)) \qquad a'_s \downarrow_L = a_s \qquad v_p \downarrow_L \neq \cdot \\ \sigma' = \sigma[a_p \mapsto (id_p, v', M, a'_p, (A :: a'_s :: \mathsf{createFacet}(a, L) :: A'))] \\ \sigma'' = \sigma'[a \mapsto (id, \mathsf{updateFacet}(v', v, L), \cdot, \mathsf{updateFacet}(a'_p, a_p, L), \cdot)]\end{array}}{\mathsf{createSibling}_{\mathsf{FS}}(\sigma, L, id, a_s, v) = \sigma''}\ \textsc{fs-createS-uL}$$

$$\frac{\begin{array}{c}\mathsf{lookupA}_{\mathsf{FS}}(\sigma, \cdot, id, a^{\mathsf{rt}}) = \langle \mathsf{NULL} | a \rangle \\ \sigma(a) = (id, v', M, a'_p, A) \qquad \sigma(a_s).v \downarrow_H \neq \cdot \qquad \sigma(a_s).a_p \downarrow_H = a_p \\ \sigma(a_p) = (id_p, v_p, M_p, a'_p, (A_p :: a'_s :: A'_p)) \qquad a'_s \downarrow_H = a_s \qquad v_p \downarrow_H \neq \cdot \\ \sigma' = \sigma[a_p \mapsto (id_p, v', M, a'_p, (A :: a'_s :: \mathsf{createFacet}(a, H) :: A'))] \\ \sigma'' = \sigma'[a \mapsto (id, \mathsf{updateFacet}(v', v, H), \cdot, \mathsf{updateFacet}(a'_p, a_p, H), \cdot)]\end{array}}{\mathsf{createSibling}_{\mathsf{FS}}(\sigma, H, id, a_s, v) = \sigma''}\ \textsc{fs-createS-uH}$$

$$\frac{\begin{array}{c}\sigma' = \mathsf{createSibling}_{\mathsf{FS}}(\sigma, H, id, \mathsf{getFacetA}(a_s, H), \mathsf{getFacetV}(v, H)) \\ \sigma'' = \mathsf{createSibling}_{\mathsf{FS}}(\sigma, L, id, \mathsf{getFacetA}(a_s, L), \mathsf{getFacetV}(v, L))\end{array}}{\mathsf{createSibling}_{\mathsf{FS}}(\sigma, \cdot, id, a_s, v) = \sigma''}\ \textsc{fs-createS-nc}$$

$$\frac{\mathsf{lookupA}_{\mathsf{FS}}(\sigma, \cdot, id, a^{\mathsf{rt}}) = a \qquad a \downarrow_{pc_l} \neq \cdot}{\mathsf{createSibling}_{\mathsf{FS}}(\sigma, pc_l, id, a_s, v) = \sigma}\ \textsc{fs-createS-s1} \qquad \frac{a_s = \mathsf{NULL} \vee \sigma(a_s).a_p \downarrow_{pc_l} = \mathsf{NULL}}{\mathsf{createSibling}_{\mathsf{FS}}(\sigma, pc_l, id, a_s, v) = \sigma}\ \textsc{fs-createS-s2}$$

$$\frac{\sigma(a_s).v \downarrow_{pc_l} = \cdot \vee \sigma(a_s).a_p \downarrow_{pc_l} = \cdot \vee \sigma(\sigma(a_s).a_p \downarrow_{pc_l}).v \downarrow_{pc_l} = \cdot}{\mathsf{createSibling}_{\mathsf{FS}}(\sigma, pc_l, id, a_s, v) = \sigma}\ \textsc{fs-createS-s3}$$

Figure 36: Rules for adding a sibling to a node in the tree-structured EH storage

$\boxed{\mathsf{registerEH}_{\mathcal{G}}(\sigma_1^{\mathcal{G}}, pc, a, eh) = \sigma_2^{\mathcal{G}}}$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \quad \sigma'(a) = (id, v, M, a_p, A) \quad eh = \mathsf{on}Ev(x)\{c\} \quad M' = M[Ev \mapsto M(Ev) \cup \{(eh, pc_l)\}]}{\mathsf{registerEH}_{\mathsf{SMS}}(\sigma, pc_l, a, eh) = \mathsf{setStore}(\sigma, pc_l, \sigma'[a \mapsto (id, v, M', a_p, A)])} \text{ SMS-REGISTEREH}$$

$$\frac{\sigma' = \mathsf{registerEH}_{\mathsf{SMS}}(\sigma, H, \mathsf{getFacet}(a, H), eh) \quad \sigma'' = \mathsf{registerEH}_{\mathsf{SMS}}(\sigma, L, \mathsf{getFacet}(a, L), eh)}{\mathsf{registerEH}_{\mathsf{SMS}}(\sigma, \cdot, a, eh) = \sigma''} \text{ SMS-REGISTEREH-NC}$$

$$\frac{}{\mathsf{registerEH}_{\mathsf{SMS}}(\sigma, pc_l, \mathsf{NULL}, eh) = \sigma} \text{ SMS-REGISTEREH-S}$$

$$\frac{(id, v, M, a_p, A) = \sigma(a) \quad v \downarrow_{pc_l} \neq \cdot \quad eh = \mathsf{on}Ev(x)\{c\} \quad M' = M[Ev \mapsto M(Ev) \cup \{(eh, pc_l)\}]}{\mathsf{registerEH}_{\mathsf{FS}}(\sigma, pc_l, a, eh) = \sigma[id \mapsto (id, v, M', a_p, A)]} \text{ FS-REGISTEREH}$$

$$\frac{\sigma' = \mathsf{registerEH}_{\mathsf{FS}}(\sigma, H, \mathsf{getFacet}(a, H), eh) \quad \sigma'' = \mathsf{registerEH}_{\mathsf{FS}}(\sigma', L, \mathsf{getFacet}(a, L), eh)}{\mathsf{registerEH}_{\mathsf{FS}}(\sigma, \cdot, a, eh) = \sigma''} \text{ FS-REGISTEREH-NC}$$

$$\frac{a = \mathsf{NULL} \vee \sigma(a).v \downarrow_{pc_l} = \cdot}{\mathsf{registerEH}_{\mathsf{FS}}(\sigma, pc_l, a, eh) = \sigma} \text{ FS-REGISTEREH-S}$$

Figure 37: Rules for registering an event handler to a node in the tree-structured EH storage

attribute/parent appropriately. If a node with the given id already exists in the given context, or if the pointer to the parent node is NULL, FS-CREATES-S1 or FS-CREATES-S2 (respectively) leaves the store unchanged. If a_s or a_p are not visible in the given context, FS-CREATES-S3 leaves the store unchanged. If $pc = \cdot$, FS-CREATEC-NC splits the execution to add the node in both contexts.

Recall that we use the $\cdot$ context to see if a node with the given id already exists in the store. We do this to see if the node already exists in the other context so that we can initialize the node in the new context. FS-CREATES-UL handles the case where the node already exists in the H context and is being added in the L context (and respectively for FS-CREATES-UH to add the node in the H context). In these cases, we get a faceted node when we look up the node by id and the facet for the context we want to add the node in is NULL (meaning that the node does note exist in that context yet). Instead of adding a new node, we update the existing node. We use updateFacet to update the attribute and pointer to the parent.

Registering a new event handler. Figure 37 contains rules for registering an event handler to a node. One of the arguments to this function is the address node being updated, so the node does not need to be looked up.

To register a new event handler in the SMS store, SMS-REGISTEREH looks up the node and adds the new event handler to the event handler map with the given context as the label. getStore and setStore are used to ensure that we interact with the correct copy of the SMS store. If the $pc = \cdot$, SMS-REGISTEREH-NC splits the execution and registers the event handler in both copies of the store. If the node's address is NULL, SMS-REGISTEREH-S leaves the store unchanged.

Registering a new event handler in the FS store is similar. FS-REGISTEREH looks up the node and if the node is visible in the given context (i.e., the attribute is not $\cdot$), it adds the new event handler to the event handler map with the given context as the label. If the $pc = \cdot$, FS-REGISTEREH-NC splits the execution and registers the event handler in both copies of the store. If the node's address is NULL or if the node is not visible in the given context (i.e., the attribute is $\cdot$), FS-REGISTEREH-S leaves the store unchanged.

$\boxed{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow^i_{pc} v}$

$$\frac{\forall i \in [1,n], G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e_i \Downarrow^{G\downarrow_{EH}}_{pc} v_i \qquad \mathsf{ehAPIe}(G\downarrow_{EH}, \sigma^G, pc, id, v_1, \cdots, v_n) = v \qquad \mathsf{toDst}(v, pc, i) = v'}{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash \mathsf{ehAPIe}(id, e_1, \cdots, e_n) \Downarrow^i_{pc} v'} \text{ EHAPI}$$

$\boxed{\mathsf{getVal}_{\mathcal{G}}(\sigma, pc, id) = v}$

$$\frac{\mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) = \phi \neq \mathsf{NULL}}{\mathsf{getVal}_{\mathsf{SMS}}(\sigma, pc_l, id) = \phi.v} \text{ SMS-GETVAL} \qquad \frac{\mathsf{lookup}_{\mathsf{SMS}}(\sigma, pc_l, id) = \mathsf{NULL}}{\mathsf{getVal}_{\mathsf{SMS}}(\sigma, pc_l, id) = \mathsf{dv}} \text{ SMS-GETVAL-S}$$

$$\frac{\mathsf{lookup}_{\mathsf{FS}}(\sigma, pc_l, id) = \phi \neq \mathsf{NULL}}{\mathsf{getVal}_{\mathsf{FS}}(\sigma, pc_l, id) = \mathsf{getFacetV}(\phi.v, pc_l)} \text{ FS-GETVAL} \qquad \frac{\mathsf{lookup}_{\mathsf{FS}}(\sigma, pc_l, id) = \mathsf{NULL}}{\mathsf{getVal}_{\mathsf{FS}}(\sigma, pc_l, id) = \mathsf{dv}} \text{ FS-GETVAL-S}$$

$$\frac{\begin{array}{c}\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) = \phi \\ \mathsf{valOf}(\phi) \neq \mathsf{NULL} \qquad l_\phi = \mathsf{labOf}(\phi, pc_l) \qquad l_v = \mathsf{labOf}(\phi.v, pc) \qquad v = \mathsf{valOf}(\phi.v)\end{array}}{\mathsf{getVal}_{\mathsf{TS}}(\sigma, pc_l, id) = (v, l_\phi \sqcup l_v)} \text{ TS-GETVAL}$$

$$\frac{\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) = \phi \qquad \mathsf{valOf}(\phi) = \mathsf{NULL}}{\mathsf{getVal}_{\mathsf{TS}}(\sigma, pc_l, id) = (\mathsf{dv}, H)} \text{ TS-GETVAL-S}$$

Unstructured EH Storage expression semantics. For the unstructured EH store, we only have one expression which is used to look up the attribute of a node: $\mathsf{getVal}_{\mathcal{G}}$. Similar to EH commands, we evaluate EH expressions using a helper function. We first evaluate sub-expressions, then pass those results to the helper function, and finally use toDst to convert the result to the appropriate format given by i.

To look up the attribute of an SMS node, we look up the node with the given id. If it exists, SMS-GETVAL returns the attribute stored in the node. If it does not exist, SMS-GETVAL-S returns the default value.

Looking up the attribute of an FS node is similar. First, we look up the node with given id. If the node exists, FS-GETVAL returns the attribute using getFacetV, which replaces empty facets with the default value, if the node is not visible in the given context. If a node with matching id does not exist, FS-GETVAL-S returns the default value.

To look up the attribute of a TS node, we first look up the node with given id, like for SMS and FS. If the node exists, TS-GETVAL returns the attribute. The return value attaches a new label that is the join of the label on the attribute with the label on the node. If a node with the given id does not exist in the store, TS-GETVAL-S returns a tainted default value. The label in this case is always H, regardless of the context. This is because we don't want the attacker to distinguish between a lookup of a tainted node (which would have label H) and a lookup of a node which doesn't exist.

$\boxed{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e \Downarrow^i_{pc} v}$

$$\frac{\forall i \in [1,n], G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash e_i \Downarrow^{G\downarrow_{EH}}_{pc} v_i \qquad \mathsf{ehAPIe}(G\downarrow_{EH}, \sigma^G, pc, a, v_1, \cdots, v_n) = v \qquad \mathsf{toDst}(v, pc, i) = v'}{G, \mathcal{V}, \sigma^G, \sigma^{\mathcal{V}} \vdash \mathsf{ehAPIe}(a, e_1, \cdots, e_n) \Downarrow^i_{pc} v'} \text{ EHAPI}$$

$\boxed{\mathsf{moveX}_{\mathsf{SMS}}(\sigma^{\mathsf{SMS}}, pc, \cdots) = a}$

$$\frac{}{\mathsf{moveRoot}_{\mathsf{SMS}}(\sigma, pc_l) = a^{\mathsf{rt}}} \text{ SMS-MOVEROOT}$$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l)}{\mathsf{moveUp}_{\mathsf{SMS}}(\sigma, pc_l, a) = \sigma'(a).a_p} \text{ SMS-MOVEU} \qquad \frac{}{\mathsf{moveUp}_{\mathsf{SMS}}(\sigma, pc_l, \mathsf{NULL}) = \mathsf{NULL}} \text{ SMS-MOVEU-S}$$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \quad \sigma'(a).A = a' :: A}{\mathsf{moveDown}_{\mathsf{SMS}}(\sigma, pc_l, a) = a'} \text{ SMS-MOVED} \qquad \frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \quad \sigma'(a).A = \cdot}{\mathsf{moveDown}_{\mathsf{SMS}}(\sigma, pc_l, a) = \mathsf{NULL}} \text{ SMS-MOVED-S1}$$

$$\frac{}{\mathsf{moveDown}_{\mathsf{SMS}}(\sigma, pc_l, \mathsf{NULL}) = \mathsf{NULL}} \text{ SMS-MOVED-S2}$$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \qquad \sigma'(a).a_p = a_p \qquad \sigma'(a_p).A = A :: a :: a' :: A'}{\mathsf{moveRight}_{\mathsf{SMS}}(\sigma, pc_l, a) = a'} \text{ SMS-MOVER}$$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \qquad \sigma'(a).a_p = a_p \qquad \sigma'(a_p).A = A'_p :: a}{\mathsf{moveRight}_{\mathsf{SMS}}(\sigma, pc_l, a) = \mathsf{NULL}} \text{ SMS-MOVER-S1}$$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \qquad \sigma'(a).a_p = \mathsf{NULL}}{\mathsf{moveRight}_{\mathsf{SMS}}(\sigma, pc_l, a) = \mathsf{NULL}} \text{ SMS-MOVER-S2}$$

$$\frac{}{\mathsf{moveRight}_{\mathsf{SMS}}(\sigma, pc_l, \mathsf{NULL}) = \mathsf{NULL}} \text{ SMS-MOVER-S3}$$

Figure 38: Rules for navigating the tree-structured SMS EH storage

Tree-structured EH Storage expression semantics. For the tree-structured EH store, we have several expressions for navigating the tree (moveX), and looking up the attribute (getVal) and children of a node (getChildren). We evaluate EH expressions using a helper function. Using EHAPI, we first evaluate sub-expressions, then pass those results to a helper function, and finally use toDst to ensure the results are appropriately formatted for the mechanism i.

Traversing the tree. We have several APIs for navigating the EH storage trees. moveRoot returns the root node (i.e., the top-most node in the tree). moveUp takes the address of a node and returns the node above it (i.e., its parent). moveDown takes the address of a node and returns the first node below it (i.e., its first child). moveRight takes the address of a node and returns the node to its right (i.e., its righthand sibling).

Figure 38 shows the rules for navigating the SMS tree. SMS-MOVEROOT returns the address of the root node in the tree. Recall that the root node is always given by a^{rt}. SMS-MOVEU takes the address of a node and returns its parent and SMS-MOVEU-S handles the case where the given address is NULL (in which case we return NULL). SMS-MOVED takes the address of a node and returns its first child. SMS-MOVED-S1 and SMS-MOVED-S2 handle the case where the given node does not have any children, or the given node is NULL (respectively). In both cases we return NULL. SMS-MOVER takes the address of a node (a), navigates to its parent (a_p), and returns a_p's child following a. If there is no child following a, SMS-MOVER-S1 returns NULL. If a_p is NULL, SMS-MOVER-S2 returns NULL. If a is NULL, SMS-MOVER-S3 returns NULL.

Figure 39 shows the rules for navigating the FS tree. FS-MOVEROOT returns the address of the root node in the tree. Recall that the root node is always given by a^{rt}. FS-MOVEU takes the address of a node and returns its parent if it is visible in the given context (i.e., $a_p \downarrow_{pc_l} \neq \cdot$). FS-MOVEU-S1 handles the case where the parent of the given node is not visible in the given context (i.e., $a_p \downarrow_{pc_l} = \cdot$), in which case we return NULL. FS-MOVEU-S2 handles the case where the given address is NULL, in which case we return NULL. FS-MOVED takes the address of a node and returns the first child visible in the given context (i.e., the first child of $A \downarrow_{pc_l}$). FS-MOVED-S1 and FS-MOVED-S2 handle the

$\boxed{\mathsf{moveX}_{\mathsf{FS}}(\sigma^{\mathsf{FS}}, pc, \cdots) = a}$

$$\frac{}{\mathsf{moveRoot}_{\mathsf{FS}}(\sigma, pc_l) = a^{\mathsf{rt}}}\ \textsc{fs-moveRoot}$$

$$\frac{\sigma(a).a_p \downarrow_{pc_l} \neq \cdot}{\mathsf{moveUp}_{\mathsf{FS}}(\sigma, pc_l, a) = \sigma(a).a_p \downarrow_{pc_l}}\ \textsc{fs-moveU} \qquad \frac{\sigma(a).a_p \downarrow_{pc_l} = \cdot}{\mathsf{moveUp}_{\mathsf{FS}}(\sigma, pc_l, a) = \mathsf{NULL}}\ \textsc{fs-moveU-s1}$$

$$\frac{}{\mathsf{moveUp}_{\mathsf{FS}}(\sigma, pc_l, \mathsf{NULL}) = \mathsf{NULL}}\ \textsc{fs-moveU-s2}$$

$$\frac{\sigma(a).A \downarrow_{pc_l} = a' :: A'}{\mathsf{moveDown}_{\mathsf{FS}}(\sigma, pc_l, a) = a'}\ \textsc{fs-moveD} \qquad \frac{\sigma(a).A \downarrow_{pc_l} = \cdot}{\mathsf{moveDown}_{\mathsf{FS}}(\sigma, pc_l, a) = \mathsf{NULL}}\ \textsc{fs-moveD-s1}$$

$$\frac{}{\mathsf{moveDown}_{\mathsf{FS}}(\sigma, pc_l, \mathsf{NULL}) = \mathsf{NULL}}\ \textsc{fs-moveD-s2}$$

$$\frac{\sigma(a).a_p \downarrow_{pc_l} = a_p \qquad \sigma(a_p).A \downarrow_{pc_l} = A' :: a :: a' :: A'}{\mathsf{moveRight}_{\mathsf{FS}}(\sigma, pc_l, a) = a'}\ \textsc{fs-moveR}$$

$$\frac{\sigma(a).a_p \downarrow_{pc_l} = a_p \qquad \sigma(a_p).A \downarrow_{pc_l} = A' :: a}{\mathsf{moveRight}_{\mathsf{FS}}(\sigma, pc_l, a) = \mathsf{NULL}}\ \textsc{fs-moveR-s1}$$

$$\frac{\sigma(a).a_p \downarrow_{pc_l} = \mathsf{NULL} \vee \sigma(a).a_p \downarrow_{pc_l} = \cdot}{\mathsf{moveRight}_{\mathsf{FS}}(\sigma, pc_l, a) = \mathsf{NULL}}\ \textsc{fs-moveR-s2} \qquad \frac{}{\mathsf{moveRight}_{\mathsf{FS}}(\sigma, pc_l, \mathsf{NULL}) = \mathsf{NULL}}\ \textsc{fs-moveR-s3}$$

Figure 39: Rules for navigating the tree-structured FS EH storage

$\boxed{\mathsf{getVal}_{\mathcal{G}}(\sigma^{\mathcal{G}}, pc, a) = v}$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \qquad \phi = \sigma'(a)}{\mathsf{getVal}_{\mathsf{SMS}}(\sigma, pc_l, a) = \phi.v}\ \textsc{sms-getVal} \qquad \frac{}{\mathsf{getVal}_{\mathsf{SMS}}(\sigma, pc_l, \mathsf{NULL}) = \mathsf{dv}}\ \textsc{sms-getVal-s}$$

$$\frac{\sigma(a) = \phi}{\mathsf{getVal}_{\mathsf{FS}}(\sigma, pc_l, a) = \mathsf{getFacetV}(\phi.v, pc_l)}\ \textsc{fs-getVal} \qquad \frac{}{\mathsf{getVal}_{\mathsf{FS}}(\sigma, pc_l, \mathsf{NULL}) = \mathsf{dv}}\ \textsc{fs-getVal-s}$$

Figure 40: Rules for accessing the attribute of node in the tree-structured EH storage

$\boxed{\mathsf{getChildren}_{\mathcal{G}}(\sigma^{\mathcal{G}}, pc, a) = v}$

$$\frac{\sigma' = \mathsf{getStore}(\sigma, pc_l) \qquad \phi = \sigma'(a)}{\mathsf{getChildren}_{\mathsf{SMS}}(\sigma, pc_l, a) = \mathsf{len}(\phi.A)}\ \textsc{sms-getChildren}$$

$$\frac{}{\mathsf{getChildren}_{\mathsf{SMS}}(\sigma, pc_l, \mathsf{NULL}) = \mathsf{dv}}\ \textsc{sms-getChildren-s}$$

$$\frac{\sigma(a) = \phi \qquad \phi.v \downarrow_{pc_l} \neq \cdot}{\mathsf{getChildren}_{\mathsf{FS}}(\sigma, pc_l, a) = \mathsf{len}(\phi.A \downarrow_{pc_l})}\ \textsc{fs-getChildren} \qquad \frac{\sigma(a).v \downarrow_{pc_l} = \cdot \vee a = \mathsf{NULL}}{\mathsf{getChildren}_{\mathsf{FS}}(\sigma, pc_l, a) = \mathsf{dv}}\ \textsc{fs-getChildren-s}$$

Figure 41: Rules for returning the number children a node in the tree-structured EH storage has

case where the given node does not have any children visible in the given context (i.e., $A \downarrow_{pc_l} = \cdot$), or the given node is NULL (respectively). In both cases we return NULL. FS-MOVER takes the address of a node (a), navigates to its parent (a_p), and returns a_p's child following a. If there is no child following a, FS-MOVER-S1 returns NULL. If a_p is NULL, or not visible in the given context, FS-MOVER-S2 returns NULL. If a is NULL, FS-MOVER-S3 returns NULL.

Looking up attributes. Figure 40 shows the rules for looking up the attribute of a given node. To look up the attribute of an SMS node, SMS-GETVAL looks up the given address in the relevant copy of the store and returns the node's attribute. If the given address is NULL, SMS-GETVAL-S returns a default value.

To look up the attribute of an FS node, FS-GETVAL looks up the given address and returns the appropriate facet of the node. If the given address is NULL, FS-GETVAL-S returns a default value.

Number of children. Figure 41 shows the rules for looking up the number of children a node has. Note that we return a default value instead of 0 for invalid or NULL nodes. This is to ensure that the attacker cannot tell the difference between a node not existing and a node being hidden from the current context.

For an SMS node, SMS-GETCHILDREN first looks up the node in the relevant copy of the store. Then, it computes the length of the node's list of children. If the given address is NULL, SMS-GETCHILDREN-S returns a default value.

For an FS node, FS-GETCHILDREN looks up the node and, if it is visible in the current context (i.e., $v \downarrow_{pc_l} \neq \cdot$), then it returns the length of the node's list of children. If the given address is NULL or if the node is not visible in the current context (i.e., $v \downarrow_{pc_l} = \cdot$), then FS-GETCHILDREN-S returns the default value.

Appendix D.
Weak Secrecy Semantics

We differentiate weak from standard semantics by using $\Vdash_w$ instead of $\Vdash$, $\vdash_w$ instead of $\vdash$, assignW instead of assign, etc., respectively.

Modifications to command semantics for tracking conditionals and global writes. IF-FALSE, WHILE-TRUE, and WHILE-FALSE are similar to IF-TRUE.

$\boxed{G, \mathsf{TT}, d \Vdash_w \sigma_1^G, \sigma_1^{\mathsf{TT}}, c_1 \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_2^{\mathsf{TT}}, c_2, E}$

$$\frac{G, \mathsf{TT}, \sigma^G, \sigma \vdash e \Downarrow_{pc}^{\mathsf{TT}} (\mathsf{true}, l) \qquad pc \sqsubseteq L \qquad l \not\sqsubseteq L}{G, \mathsf{TT}, d \Vdash_w \sigma^G, \sigma, \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \xrightarrow{\mathsf{br(true)}}_{pc} \sigma^G, \sigma, c_1, \cdot}\ \text{IF-TRUE-BR}$$

$$\frac{G, \mathsf{TT}, \sigma_1^G, \sigma \vdash e \Downarrow_{pc}^{G\downarrow_g} v \qquad x \in \sigma^G \qquad pc \not\sqsubseteq L \qquad \mathsf{assignW}_{G\downarrow_g}(\sigma_1^G, pc, x, v) = (\sigma_2^G, \alpha)}{G, \mathsf{TT}, d \Vdash_w \sigma_1^G, \sigma_1, x := e \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_1, \mathsf{skip}, \cdot}\ \text{ASSIGN-G-H}$$

$$\frac{G, \mathsf{TT}, \sigma_1^G, \sigma \vdash e \Downarrow_{pc}^{G\downarrow_{EH}} v \qquad pc \not\sqsubseteq L \qquad \mathsf{assignW}_{G\downarrow_{EH}}(\sigma_1^G, pc, id, v) = (\sigma_2^G, \alpha)}{G, \mathsf{TT}, d \Vdash_w \sigma_1^G, \sigma_1, id := e \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_1, \mathsf{skip}, \cdot}\ \text{ASSIGN-D-H}$$

$$\frac{G, \mathsf{TT}, \sigma_1^G, \sigma \vdash e \Downarrow_{pc}^{G\downarrow_{EH}} v \qquad (\sigma_2^G, \alpha) = \mathsf{createElemW}_{G\downarrow_{EH}}(\sigma_1^G, pc, id, v)}{G, \mathsf{TT}, d \Vdash_w \sigma_1^G, \sigma_1, \mathsf{create}(id, e) \xrightarrow{\alpha}_{pc} \sigma_2^G, \sigma_1, \mathsf{skip}, \cdot}\ \text{CREATEELEM-H}$$

For $\mathcal{G} \neq \mathsf{TS}$, all assignment functions return $\bullet$ for α. The modifications to TT, TS variable/EH node assignment are shown below.

$$\frac{x \in \sigma \qquad l \sqcup pc \sqsubseteq \mathsf{labOf}(\sigma(x), pc)}{\mathsf{assignW}_{\mathsf{TS}}(\sigma, pc, x, (v, l)) = (\sigma[x \mapsto (v, l \sqcup pc)], \bullet)}\ \text{TS-ASSIGN}$$

$$\frac{x \in \sigma \qquad l \sqcup pc \not\sqsubseteq \mathsf{labOf}(\sigma(x), pc)}{\mathsf{assignW}_{\mathsf{TS}}(\sigma, pc, x, (v, l)) = (\sigma[x \mapsto (v, l \sqcup pc)], \mathsf{gw}(x))}\ \text{TS-ASSIGN-GW}$$

$$\frac{x \notin \sigma}{\mathsf{assignW}_{\mathsf{TS}}(\sigma, pc, x, (v, l)) = (\sigma, \bullet)}\ \text{TS-ASSIGN-S}$$

$$\frac{(id, (v', l''), M, l') = \sigma(id) \qquad l \sqcup pc \sqsubseteq l'' \vee l' \not\sqsubseteq L}{\mathsf{assignW}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = (\sigma[id \mapsto (id, (v, l \sqcup pc \sqcup l'), M, l')], \bullet)}\ \text{TS-ASSIGNEH}$$

$$\frac{(id, (v', l''), M, l') = \sigma(id) \qquad l \sqcup pc \not\sqsubseteq l'' \qquad l' \sqsubseteq L}{\mathsf{assignW}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = (\sigma[id \mapsto (id, (v, l \sqcup pc \sqcup l'), M, l')], \mathsf{gw}(id))}\ \text{TS-ASSIGNEH-GW}$$

$$\frac{id \notin \sigma}{\mathsf{assignW}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = (\sigma, \bullet)}\ \text{TS-ASSIGNEH-S}$$

Modifications to EH APIs. This includes only the rules for TS. All other mechanisms return $\bullet$ for α. $\mathsf{gw}(id)$ is emitted whenever a node is created in the H context or the value of the node is upgraded from L to H.

$$\frac{\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) = (\mathsf{NULL}, \_) \qquad \phi = (id, v, \cdot, pc_l)}{\mathsf{createElemW}_{\mathsf{TS}}(\sigma, pc_l, id, v) = (\sigma[id \mapsto \phi], \bullet)}\ \textsc{ts-create}$$

$$\frac{(\sigma', \alpha) = \mathsf{createElemW}_{\mathsf{TS}}(\sigma, L, id, v)}{\mathsf{createElemW}_{\mathsf{TS}}(\sigma, \cdot, id, v) = (\sigma', \alpha)}\ \textsc{ts-create-nc} \qquad \frac{\begin{array}{c}\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) = (id, (v', l'), M, l'') \\ l'' \sqsubseteq pc_l \qquad l \sqcup pc_l \sqsubseteq l' \vee l'' \not\sqsubseteq L \\ \sigma' = \sigma[id \mapsto (id, (v, l \sqcup pc_l \sqcup l''), M, l'')]\end{array}}{\mathsf{createElemW}_{\mathsf{TS}}(\sigma, pc_l, id, (v, l)) = (\sigma', \bullet)}\ \textsc{ts-create-u1}$$

$$\frac{\begin{array}{c}\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) = (id, (v', l'), M, l'') \\ l'' \sqsubseteq pc_l \qquad l'' \sqsubseteq L \qquad l \sqcup pc_l \not\sqsubseteq l' \\ \sigma' = \sigma[id \mapsto (id, (v, l \sqcup pc_l \sqcup l''), M, l'')]\end{array}}{\mathsf{createElemW}_{\mathsf{TS}}(\sigma, pc_l, id, (v, l)) = (\sigma', \mathsf{gw}(id))}\ \textsc{ts-create-u1-gw}$$

$$\frac{\begin{array}{c}\mathsf{lookup}_{\mathsf{TS}}(\sigma, pc_l, id) = (id, (v', l'), M, l'') \\ l'' \not\sqsubseteq pc_l \\ \sigma' = \sigma[id \mapsto (id, (v, l \sqcup pc_l), M, pc_l)]\end{array}}{\mathsf{createElemW}_{\mathsf{TS}}(\sigma, pc, id, (v, l)) = (\sigma', \bullet)}\ \textsc{ts-create-u2}$$

Appendix E.
Security Definitions

E.1. Knowledge Definitions

Knowledge:

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

Progress-insensitive knowledge:

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

$$\text{where } \mathsf{prog}(T) \text{ iff } T = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K \text{ and } \exists K_C \text{ s.t. } G, \mathcal{P} \vdash K \Longrightarrow^* K_C \text{ and } \mathsf{consumer}(K_C)$$

Progress-insensitive knowledge with release:

$$\mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha) = \{\tau_i \,|\, \exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{P}), T \approx_L T' \wedge \tau_i = \mathsf{in}(T') \wedge \mathsf{prog}(T') \wedge \alpha' = (\mathsf{last}(T) \stackrel{\alpha}{\Longrightarrow} K)) \downarrow_L, \mathsf{releaseT}(T', \alpha'))\}$$

$$\mathsf{releaseT}(T, \alpha) = \begin{cases} T = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K \wedge \exists \alpha', K' s.t., G, \mathcal{P} \vdash K \stackrel{\alpha'}{\Longrightarrow} K' & \\ \quad \wedge (G, \mathcal{P} \vdash K \stackrel{\alpha'}{\Longrightarrow} K') \downarrow_L = \alpha & \alpha = \mathsf{rls}(\_) \\ T = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K \wedge \exists K' s.t., G, \mathcal{P} \vdash K \stackrel{\alpha}{\Longrightarrow} K' & \alpha = \mathsf{declassify}(\iota, v) \end{cases}$$

Progress-insensitive weak secrecy knowledge:

$$\mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha) = \{\tau_i \,|\, \exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{P}), T \approx_L T' \wedge \tau_i = \mathsf{in}(T') \wedge \mathsf{prog}(T') \wedge \alpha' = (G, \mathcal{P} \vdash \mathsf{last}(T) \stackrel{\alpha}{\Longrightarrow} K)) \downarrow_L, \mathsf{wkT}(T', \alpha'))\}$$

$$\mathsf{wkT}(T, \alpha) = \begin{cases} T = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K \wedge \exists K' \text{ s.t. } K \Longrightarrow K' \wedge (G, \mathcal{P} \vdash K \Longrightarrow K') \downarrow_L = \alpha & \alpha = \mathsf{br}(b) \\ T = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K \wedge \exists K', T' \text{ s.t. } T' = G, \mathcal{P} \vdash K \Longrightarrow^* K' \wedge T' \downarrow_L = \cdot & \alpha = \mathsf{gw}(x) \\ \quad \wedge \exists K'' \text{ s.t. } G, \mathcal{P} \vdash K' \Longrightarrow^* K'' \wedge (G, \mathcal{P} \vdash K' \Longrightarrow^* K'') \downarrow_L = \alpha & \\ T = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K & \alpha = id.Ev(v) \\ \quad \wedge \exists T', K' \text{ s.t. } T' = G, \mathcal{P} \vdash K \Longrightarrow^* K_C \wedge \mathsf{consumer}(K_C) \wedge T' \downarrow_L = \cdot & \\ T = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K & \alpha \in \{\bullet, ch(v)\} \\ \quad \wedge \exists T', K' \text{ s.t. } T' = G, \mathcal{P} \vdash K \Longrightarrow^* K_{lp} \wedge \mathsf{lowProducer}(K_{lp}) \wedge T' \downarrow_L = \cdot & \end{cases}$$

$$\frac{\neg\mathsf{consumer}(K) \quad K = \mathcal{R}, d, \sigma^G, (\mathcal{V}; \kappa; pc) :: ks \qquad pc \sqsubseteq L}{\mathsf{lowProducer}(K)} \qquad \frac{\neg\mathsf{consumer}(K) \quad K = \mathcal{R}, d, \sigma^G, (\mathcal{V}; \kappa; pc) :: ks \qquad pc \not\sqsubseteq L}{\mathsf{highProducer}(K)}$$

$$\frac{pc \sqsubseteq L}{\mathsf{lowContext}((\mathcal{V}; \kappa; pc) :: ks) = \mathsf{true}} \qquad \frac{pc \not\sqsubseteq L}{\mathsf{lowContext}((\mathcal{V}; \kappa; pc) :: ks) = \mathsf{false}}$$

$$\frac{pc \not\sqsubseteq L}{\mathsf{highContext}((\mathcal{V}; \kappa; pc) :: ks) = \mathsf{true}} \qquad \frac{pc \sqsubseteq L}{\mathsf{highContext}((\mathcal{V}; \kappa; pc) :: ks) = \mathsf{false}}$$

E.1.1. Trace Equivalence. $\boxed{T \approx_L T'}$

$$T \approx_L T' \text{ iff } T \downarrow_L = T' \downarrow_L$$

$\boxed{\mathcal{P}(\alpha) = l}$

Note that $\mathcal{P}(\alpha) = \cdot$ has a different interpretation than the $\cdot$ we have previously used. Here, it means "it has no label".

$$\frac{}{\mathcal{P}(ch(v)) = \mathcal{P}(ch)}\ \textsc{Lab-O} \qquad \frac{}{\mathcal{P}(\mathsf{declassify}(\iota, v)) = L}\ \textsc{Lab-D} \qquad \frac{}{\mathcal{P}(\mathsf{gw}(x)) = L}\ \textsc{Lab-GW}$$

$$\frac{}{\mathcal{P}(\mathsf{br}(b)) = L}\ \textsc{Lab-BR} \qquad \frac{\alpha \notin \{id.Ev(v), ch(v), \mathsf{declassify}(\iota, v), \mathsf{gw}(x), \mathsf{br}(b)\}}{\mathcal{P}(\alpha) = \cdot}\ \textsc{Lab-S}$$

$\boxed{T \downarrow_L = \tau}$

$$\frac{}{G, \mathcal{P} \vdash K \downarrow_L = \cdot}\ \text{TP-BASE} \qquad \frac{\mathcal{P}(id.Ev(v)) = L}{(G, \mathcal{P} \vdash K \overset{id.Ev(v)}{\Longrightarrow} T') \downarrow_L = id.Ev(v) :: T' \downarrow_L}\ \text{TP-LI}$$

$$\frac{\mathcal{P}(\alpha) = L \quad \text{or} \quad l \sqsubseteq L}{(G, \mathcal{P} \vdash K \overset{(\alpha,l)}{\Longrightarrow} T') \downarrow_L = \alpha :: T' \downarrow_L}\ \text{TP-L} \qquad \frac{\alpha_l = \langle \_ | \_ \rangle}{(G, \mathcal{P} \vdash K \overset{\alpha_l}{\Longrightarrow} T') \downarrow_L = \mathsf{getFacet}(\alpha_l, L) :: T' \downarrow_L}\ \text{TP-F}$$

$$\frac{\mathsf{rel}(K) = \mathcal{R} \qquad \mathcal{P}(id.ev(v)) = H \qquad \mathcal{R}(\mathcal{P}, id.ev(v)) = \alpha \neq \bullet}{(G, \mathcal{P} \vdash K \overset{id.ev(v)}{\Longrightarrow} T') \downarrow_L = \mathsf{rls}(\alpha) :: T' \downarrow_L}\ \text{TP-HI-R}$$

$$\frac{\mathcal{P}(id.ev(v)) = H_\Delta}{(G, \mathcal{P} \vdash K \overset{id.ev(v)}{\Longrightarrow} T') \downarrow_L = T' \downarrow_L}\ \text{TP-HI-NR1}$$

$$\frac{\mathsf{rel}(K) = \mathcal{R} \qquad \mathcal{P}(id.ev(v)) = H \wedge \mathcal{R}(\mathcal{P}, id.ev(v)) = \bullet}{(G, \mathcal{P} \vdash K \overset{id.ev(v)}{\Longrightarrow} T') \downarrow_L = T' \downarrow_L}\ \text{TP-HI-NR2} \qquad \frac{\mathcal{P}(\alpha) = H \quad \text{or} \quad \alpha = \bullet}{(G, \mathcal{P} \vdash K \overset{(\alpha,H)}{\Longrightarrow} T') \downarrow_L = T' \downarrow_L}\ \text{TP-H}$$

Figure 42: Projection of Traces to L Observation

$\boxed{\mathcal{R}(\mathcal{P}, id.ev(v)) = \alpha}$

$$\frac{\mathcal{R} = (\rho, \mathcal{D}) \qquad \mathcal{D}(\rho, id.Ev(v)) = (\rho', r, v', \mathcal{V})}{\mathcal{R}(\mathcal{P}, id.Ev(v)) = id.Ev(v')} \qquad \frac{\mathcal{R} = (\rho, \mathcal{D}) \qquad \mathcal{D}(\rho, id.Ev(v)) = (\rho', r, \bullet, \mathcal{V})}{\mathcal{R}(\mathcal{P}, id.Ev(v)) = \bullet}$$

$$\begin{aligned}
\mathsf{rlsA}(G, \mathcal{P} \vdash K \overset{\alpha}{\Longrightarrow} K') \quad \text{iff} \quad & (G, \mathcal{P} \vdash K \overset{\alpha}{\Longrightarrow} K') \downarrow_L = \mathsf{rls}(\alpha') \text{ or} \\
& (G, \mathcal{P} \vdash K \overset{\alpha}{\Longrightarrow} K') \downarrow_L = \mathsf{declassify}(\iota, v) \\
\mathsf{wkA}(G, \mathcal{P} \vdash K \overset{\alpha}{\Longrightarrow} K') \quad \text{iff} \quad & (G, \mathcal{P} \vdash K \overset{\alpha}{\Longrightarrow} K') \downarrow_{L,w} = \mathsf{br}(b) \text{ or} \\
& (G, \mathcal{P} \vdash K \overset{\alpha}{\Longrightarrow} K') \downarrow_{L,w} = \mathsf{gw}(x) \text{ or} \\
& \alpha \in \mathsf{in}(G, \mathcal{P} \vdash K \overset{\alpha}{\Longrightarrow} K') \wedge \mathcal{P}(\alpha) = L \text{ or} \\
& (G, \mathcal{P} \vdash K \overset{\alpha}{\Longrightarrow} K') \downarrow_L \in \{\bullet, ch(v)\}
\end{aligned}$$

E.2. Progress-Insensitive Security and Weak Secrecy

We say σ_0^G is well-formed if the initial EH store is defined and the global variables are initialized. The event handlers in the EH store should agree on the names of the global variables.

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

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

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

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

E.3. Progress-Insensitive Security and Weak Secrecy Requirements

Trace Requirements	
Equivalent traces produce **(T1)** L-equivalent states	If T_1 and T_2 produce the same L-observable events and begin in L-equivalent states, then they should end in L-equivalent states.
Empty traces produce **(T2)** L-equivalent states	If T_1 produces no L-observable events, then the start and end state of T_1 should be L-equivalent.
H steps produce L-equivalent states and empty traces **(T3)**	If T_1 takes one or more steps in an H context, then the first and last state are L-equivalent and T_1 produces no L-observable events.
Strong one-step **(T4)**	If T_1 produces an L-observable event in one step, and T_1 and T_2 begin in L-equivalent states, T_2 should be able to take some step(s) to produce the same L-observable event.
Weak one-step **(T5)**	If T_1 and T_2 begin in L-equivalent states and produce the same L-observable events in one step, they should end in L-equivalent states.
Expression Requirements	
L-expressions are equivalent **(E1)**	Evaluating L-equivalent expressions under L-observable pc's result in (strong) L-equivalent values.
Variable Requirements	
L-lookups are equivalent **(V1)**	Lookups of the same variable under L-observable pc's in L-equivalent stores, result in (strong) L-equivalent values.
H-assignments are unobservable **(V2)**	Assignments under L-unobservable pc's result in an L-equivalent store.
L-assignments are equivalent **(V3)**	Assignments under L-observable pc's of L-equivalent values to the same variable in L-equivalent stores result in L-equivalent stores.
Event Handler Storage Requirements	
L-lookups are equivalent **(EH1)**	All lookup APIs of the same structure under L-observable pc's in L-equivalent EH stores result in (strong) L-equivalent values.
H EH lookups are unobservable **(EH2)**	Event handler lookups under L-unobservable pc's produce L-unobservable commands.
H-updates are unobservable **(EH3)**	Updates under L-unobservable pc's result in an L-equivalent store.
L-updates are equivalent **(EH4)**	Updates under L-observable pc's of L-equivalent values to the same structures in L-equivalent stores result in L-equivalent stores.

Figure 43: Requirements for Knowledge-Based Security

(Weak) Trace Requirements	
Equivalent traces produce **(WT1)** L-equivalent states	If T_1 and T_2 produce the same L-observable events and begin in L-equivalent states, then they should end in L-equivalent states.
Empty traces produce **(WT2)** L-equivalent states	If T_1 produces no L-observable events, then the start and end state of T_1 should be L-equivalent.
Strong one-step **(WT4)**	If T_1 produces an L-observable event in one step, and T_1 and T_2 begin in L-equivalent states, T_2 should be able to take some step(s) to produce the same L-observable event.
Weak one-step **(WT5)**	If T_1 and T_2 begin in L-equivalent states and produce the same L-observable events in one step, they should end in L-equivalent states.
(Weak) Expression Requirements	
L-expressions are equivalent **(WE1)**	Evaluating L-equivalent expressions under L-observable pc's result in L-equivalent values.
(Weak) Variable Requirements	
L-lookups are equivalent **(WV1)**	Lookups of the same variable under L-observable pc's in L-equivalent stores, result in L-equivalent values.
H-assignments are unobservable **(WV2)**	Assignments under L-unobservable pc's result in an L-equivalent store.
L-assignments are equivalent **(WV3)**	Assignments under L-observable pc's of L-equivalent values to the same variable in L-equivalent stores result in L-equivalent stores.
(Weak) Event Handler Storage Requirements	
L-lookups are equivalent **(WEH1)**	All lookup APIs of the same structure under L-observable pc's in L-equivalent EH stores result in L-equivalent values.
H-updates are unobservable **(WEH3)**	Updates under L-unobservable pc's result in an L-equivalent store.
L-updates are equivalent **(WEH4)**	Updates under L-observable pc's of L-equivalent values to the same structures in L-equivalent stores result in L-equivalent stores.

Figure 44: Requirements for Knowledge-Based Weak Secrecy

E.4. Supporting Equivalence Definitions

E.4.1. Configuration Equivalence. Equivalence for a the compositional configurations, K. Configurations are equivalent if (1) their release modules $\mathcal{R}$ are in the same state, (2) the same values are on their release channels d, (3) their global stores are equivalent σ^G, and (4) their current configuration stacks ks are equivalent.

$\boxed{K \approx_L K'}$

$$\frac{\mathcal{R}_1 = \mathcal{R}_2 \qquad d_1 = d_2 \qquad \sigma_1^G \approx_L \sigma_2^G \qquad ks_1 \approx_L ks_2}{\mathcal{R}_1, d_1; \sigma_1^G; ks_1 \approx_L \mathcal{R}_2, d_2; \sigma_2^G; ks_2}$$

Equivalence for the configuration stack, ks. Equivalence is defined inductively from the top down; the publicly visible ($pc \sqsubseteq L$) configurations should be equivalent and the private configurations ($pc \not\sqsubseteq L$) are ignored.

$\boxed{ks \approx_L ks}$

$$\frac{}{\cdot \approx_L \cdot} \qquad \frac{\begin{array}{c} pc_1 \sqsubseteq L \wedge pc_2 \sqsubseteq L \\ \mathcal{V}_1 = \mathcal{V}_2 \qquad \kappa_1 \approx_L \kappa_2 \qquad ks_1 \approx_L ks_2 \end{array}}{((\mathcal{V}_1; \kappa_1; pc_1) :: ks_1) \approx_L ((\mathcal{V}_2; \kappa_2; pc_2) :: ks_2)} \qquad \frac{\begin{array}{c} pc_1 \not\sqsubseteq L \\ ks_1 \approx_L ((\mathcal{V}_2; \kappa_2; pc_2) :: ks_2) \end{array}}{((\mathcal{V}_1; \kappa_1; pc_1) :: ks_1) \approx_L ((\mathcal{V}_2; \kappa_2; pc_2) :: ks_2)}$$

$$\frac{\begin{array}{c} pc_2 \not\sqsubseteq L \\ ((\mathcal{V}_1; \kappa_1; pc_1) :: ks_1) \approx_L ks_2 \end{array}}{((\mathcal{V}_1; \kappa_1; pc_1) :: ks_1) \approx_L ((\mathcal{V}_2; \kappa_2; pc_2) :: ks_2)}$$

Equivalence for single configurations, κ. Two configurations are low equivalent if (1) they have equivalent stores σ, (2) they are executing the same command c, (3) they are in the same execution state s, and (4) their local events E are low equivalent. (Note: the first rule is to help with induction for SME proofs)

$\boxed{\kappa_1^{\mathcal{V}} \approx_L \kappa_2^{\mathcal{V}}}$

$$\frac{\kappa_L = \kappa'_L}{\kappa_H; \kappa_L \approx_L \kappa'_H; \kappa'_L} \qquad \frac{\begin{array}{c} \kappa_1 = \sigma_1, c_1, s_1, E_1 \qquad \kappa_2 = \sigma_2, c_2, s_2, E_2 \\ \sigma_1 \approx_L \sigma_2 \qquad c_1 = c_2 \qquad s_1 = s_2 \qquad E_1 \approx_L E_2 \end{array}}{\kappa_1 \approx_L \kappa_2}$$

Event queue E equivalence and projection. Local events, E are low equivalent iff their low projections are the same. The low projection keeps only the publicly visible events ($l \sqsubseteq L$); the secret events ($l \not\sqsubseteq L$) are ignored. Note that in our semantics, tainted arguments to local events also taints the event itself. Also, facets never appear as event arguments: instead they are split into separate events. So only the label on the event needs to be considered for the $\downarrow_L$ definition.

$$E \approx_L E' \text{ iff } E \downarrow_L = E' \downarrow_L$$

$\boxed{E \downarrow_L = E'}$

$$\frac{pc \sqsubseteq L}{((id.Ev(v), pc), E) \downarrow_L = id.Ev(v), E \downarrow_L} \qquad \frac{pc \not\sqsubseteq L}{((id.Ev(v), pc), E) \downarrow_L = E \downarrow_L} \qquad \frac{}{\cdot \downarrow_L = \cdot}$$

Command c equivalence and projection. Commands c are low equivalent iff their low projections are the same. The low projection of most commands c is the same command, c. The only exception is when the command is faceted $\langle c_H | c_L \rangle$. In that case, the low projection is the command in the low facet, c_L.

$$c_1 \approx_L c_2 \text{ iff } c_1 \downarrow_L = c_2 \downarrow_L$$

$\boxed{c \downarrow_L = c}$

$$\frac{}{c^{\mathsf{std}} \downarrow_L = c^{\mathsf{std}}} \qquad \frac{}{\langle c_H | c_L \rangle \downarrow_L = c_L}$$

EH queue $\mathcal{C}$ equivalence and projection. The EH queue is a runtime construct for building the configuration stack. EH queues $\mathcal{C}$ are low equivalent if their low projections are the same. The low projection keeps the publicly visible event handlers ($pc \sqsubseteq L$). The secret event handlers ($pc \not\sqsubseteq L$) are ignored.

$$\mathcal{C} \approx_L \mathcal{C}' \text{ iff } \mathcal{C} \downarrow_L = \mathcal{C}' \downarrow_L$$

$\boxed{\mathcal{C} \downarrow_L = \mathcal{C}'}$

$$\frac{pc \sqsubseteq L}{((eh, pc), \mathcal{C}) \downarrow_L = eh, \mathcal{C} \downarrow_L} \qquad \frac{pc \not\sqsubseteq L}{((eh, pc), \mathcal{C}) \downarrow_L = \mathcal{C} \downarrow_L} \qquad \frac{}{\cdot \downarrow_L = \cdot}$$

E.4.2. Store equivalence. Local stores $\sigma^{\mathcal{V}}$ are low equivalent if their low projections are the same.

$$\sigma_1^{\mathcal{V}} \approx_L \sigma_2^{\mathcal{V}} \text{ iff } \sigma_1^{\mathcal{V}} \downarrow_L = \sigma_2^{\mathcal{V}} \downarrow_L$$

The low projection of a local store $\sigma^{\mathcal{V}}$ keeps the publicly visible variables and ignores the secret variables. *Publicly visible* and *secret* depend on the enforcement mechanism, $\mathcal{V}$. For MF, unfaceted v^{std} values are publicly visible, as well as the low facet v_L of faceted values $\langle v_H | v_L \rangle$. If a faceted value does not have a low facet, as in $\langle v | \cdot \rangle$, the value is not publicly visible. For , L-labeled values are publicly visible (i.e. v is visible in (v, L)) and H-labeled values are secret. For SME, the L copy of the store is visible while the H copy is secret.

$\boxed{\sigma^{\mathcal{V}} \downarrow_L = \sigma^{\mathsf{std}}}$

$$\frac{}{\cdot \downarrow_L = \cdot} \qquad \frac{\sigma_1^{\mathcal{V}} = \sigma_2^{\mathcal{V}}, x \mapsto \langle \_ | v \rangle \quad \text{or} \quad \sigma_1^{\mathcal{V}} = \sigma_2^{\mathcal{V}}, x \mapsto (v, L)}{\sigma_1^{\mathcal{V}} \downarrow_L = x \mapsto v, \sigma_2^{\mathcal{V}} \downarrow_L}$$

$$\frac{\sigma_1^{\mathcal{V}} = \sigma_2^{\mathcal{V}}, x \mapsto (v, H) \quad \text{or} \quad \sigma_1^{\mathcal{V}} = \sigma_2^{\mathcal{V}}, x \mapsto \langle v | \cdot \rangle}{\sigma_1^{\mathcal{V}} \downarrow_L = \sigma_2^{\mathcal{V}} \downarrow_L} \qquad \frac{\sigma_1^{\mathsf{MF}} = \sigma_2^{\mathsf{MF}}, x \mapsto v^{\mathsf{std}}}{\sigma_1^{\mathsf{MF}} \downarrow_L = x \mapsto v^{\mathsf{std}}, \sigma_2^{\mathsf{MF}} \downarrow_L} \qquad \frac{\sigma^{\mathsf{SME}} = (\sigma_H, \sigma_L)}{\sigma^{\mathsf{SME}} \downarrow_L = \sigma_L}$$

Global stores (σ_g, σ_{EH}) are low equivalent if both their global variable stores σ_g and event handler stores σ_{EH} are equivalent.

$$(\sigma_{g,1}^{\mathcal{G}}, \sigma_{EH,1}^{\mathcal{G}'}) \approx_L (\sigma_{g,2}^{\mathcal{G}}, \sigma_{EH,2}^{\mathcal{G}'}) \text{ iff } \sigma_{g,1}^{\mathcal{G}} \approx_L \sigma_{g,2}^{\mathcal{G}} \wedge \sigma_{EH,1}^{\mathcal{G}'} \approx_L \sigma_{EH,2}^{\mathcal{G}'}$$

Equivalence for global variable stores is defined the same as for local variable stores.

$$\sigma_{g,1}^{\mathcal{G}} \approx_L \sigma_{g,2}^{\mathcal{G}} \text{ iff } \sigma_{g,1}^{\mathcal{G}} \downarrow_L = \sigma_{g,2}^{\mathcal{G}} \downarrow_L$$

$\boxed{\sigma_g^{\mathcal{G}} \downarrow_L = \sigma^{\mathsf{std}}}$

$$\frac{}{\cdot \downarrow_L = \cdot} \qquad \frac{\sigma_1^{\mathcal{G}} = \sigma_2^{\mathcal{G}}, x \mapsto \langle \_ | v \rangle \quad \text{or} \quad \sigma_1^{\mathcal{G}} = \sigma_2^{\mathcal{G}}, x \mapsto (v, L)}{\sigma_1^{\mathcal{G}} \downarrow_L = x \mapsto v, \sigma_2^{\mathcal{G}} \downarrow_L}$$

$$\frac{\sigma_1^{\mathcal{G}} = \sigma_2^{\mathcal{G}}, x \mapsto (v, H) \quad \text{or} \quad \sigma_1^{\mathcal{G}} = \sigma_2^{\mathcal{G}}, x \mapsto \langle v | \cdot \rangle}{\sigma_1^{\mathcal{G}} \downarrow_L = \sigma_2^{\mathcal{G}} \downarrow_L} \qquad \frac{\sigma_1^{\mathsf{FS}} = \sigma_2^{\mathsf{FS}}, x \mapsto v^{\mathsf{std}}}{\sigma_1^{\mathsf{FS}} \downarrow_L = x \mapsto v^{\mathsf{std}}, \sigma_2^{\mathsf{FS}} \downarrow_L} \qquad \frac{\sigma^{\mathsf{SMS}} = \sigma_H, \sigma_L}{\sigma^{\mathsf{SMS}} \downarrow_L = \sigma_L}$$

Event handler storage σ_{EH} low equivalence depends on the structure of the event handler storage. We consider an unstructured EH storage and tree structured EH storage.

Unstructured EH storage. Two unstructured EH storages σ_{EH} are low equivalent if their low projections are the same.

$$\sigma_{EH,1}^{\mathcal{G}} \approx_L \sigma_{EH,2}^{\mathcal{G}} \text{ iff } \sigma_{EH,1}^{\mathcal{G}} \downarrow_L = \sigma_{EH,2}^{\mathcal{G}} \downarrow_L$$

The low projection of an unstructured EH storage is the publicly observable parts of publicly observable nodes (secret nodes are ignored). The meaning of *publicly observable* and *secret* depends on the enforcement mechanism, $\mathcal{G}$. For SMS, the entire L copy of the EH storage is publicly observable, while the H copy is secret. In FS, a node is visible if the value stored in the node is visible (i.e. $v \downarrow_L \neq \cdot$). If the value is not visible (i.e. $v \downarrow_L = \cdot$), the node is considered secret. For TS, a node labeled L is publicly observable, while a node labeled H is secret. The definitions for *publicly observable parts* of a node are shown in Section E.4.4.

$\boxed{\sigma_{EH}^{\mathcal{G}} \downarrow_L = \sigma_{EH}^{\mathsf{std}}}$

$$\frac{\sigma_1^{\mathsf{FS}} = \sigma_2^{\mathsf{FS}}, id \mapsto \phi \qquad \phi = (v, M) \qquad v \downarrow_L \neq \cdot}{\sigma_1^{\mathsf{FS}} \downarrow_L = id \mapsto \phi \downarrow_L, \sigma_2^{\mathsf{FS}} \downarrow_L} \qquad \frac{\sigma_1^{\mathsf{FS}} = \sigma_2^{\mathsf{FS}}, id \mapsto \phi \qquad \phi = (v, M) \qquad v \downarrow_L = \cdot}{\sigma_1^{\mathsf{FS}} \downarrow_L = \sigma_2^{\mathsf{FS}} \downarrow_L}$$

$$\frac{\sigma_1^{\mathsf{TS}} = \sigma_2^{\mathsf{TS}}, id \mapsto \phi \qquad \phi = (v, M, l) \qquad l \sqsubseteq L}{\sigma_1^{\mathsf{TS}} \downarrow_L = id \mapsto \phi \downarrow_L, \sigma_2^{\mathsf{TS}} \downarrow_L} \qquad \frac{\sigma_1^{\mathsf{TS}} = \sigma_2^{\mathsf{TS}}, id \mapsto (v, M, l) \qquad l \not\sqsubseteq L}{\sigma_1^{\mathsf{TS}} \downarrow_L = \sigma_2^{\mathsf{TS}} \downarrow_L} \qquad \frac{}{(\sigma_H, \sigma_L) \downarrow_L = \sigma_L}$$

$$\frac{}{\cdot \downarrow_L = \cdot}$$

Tree structured EH storage. Two tree structured EH storages σ_{EH} are low equivalent if the public view of the tree is the same in both EH storage. The meaning of the *public view* of a tree depends on the enforcement mechanism, $\mathcal{G}$. For SMS, the entire L copy of the EH storage is the public view, so they are equivalent only if their L copies of the EH storages are the same. For FS, the low copy of the EH storage is defined as the low projection of the tree, starting at the root node, located at a^{rt}. The low projection of the tree is defined below in Section E.4.4.

$\sigma^{\mathcal{G}}_{EH,1} \approx_L \sigma^{\mathcal{G}}_{EH,2}$

$$\frac{\sigma_1^{\mathsf{SMS}} = (\sigma_{H,1}, \sigma_{L,1}) \qquad \sigma_2^{\mathsf{SMS}} = (\sigma_{H,2}, \sigma_{L,2}) \qquad \sigma_{L,1}(a_1^{\mathsf{rt}}) \downarrow_L^{\sigma_{L,1}} = \sigma_{L,2}(a_2^{\mathsf{rt}}) \downarrow_L^{\sigma_{L,2}}}{\sigma_1^{\mathsf{SMS}} \approx_L \sigma_2^{\mathsf{SMS}}} \qquad \frac{\sigma_1^{\mathsf{FS}}(a_1^{\mathsf{rt}}) \downarrow_L^{\sigma_1^{\mathsf{FS}}} = \sigma_2^{\mathsf{FS}}(a_2^{\mathsf{rt}}) \downarrow_L^{\sigma_2^{\mathsf{FS}}}}{\sigma_1^{\mathsf{FS}} \approx_L \sigma_2^{\mathsf{FS}}}$$

E.4.3. Value equivalence and strong equivalence. Two values are low equivalent iff their low projections are the same. We need a different definition for unstructured and tree-structured EH storages because for the tree-structured EH storage values include locations in the EH store, while the unstructured EH storage can use a simpler equivalence definition because it does not include references.

Unstructured EH storage.

$$v \approx_L v' \text{ iff } v \downarrow_L = v' \downarrow_L$$

The low projection of a standard value v^{std} is the same value v^{std}. The low projection of a faceted value $\langle v_H | v_L \rangle$ is the value in the low facet v_L, or nothing (denoted $\cdot$) when the low facet is empty, as in $\langle v | \cdot \rangle$. The low projection of a labeled value (v, l) is the value v when the label is L, or nothing, otherwise.

$v \downarrow_L = v'$

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

Tree-structured EH storage.

$$v \approx_L^{\sigma_1, \sigma_2} v' \text{ iff } v \downarrow_L^{\sigma_1} = v' \downarrow_L^{\sigma_2}$$

The low projection of a standard value v^{std} is the same value v^{std} when the value is not a (non-NULL) location. If the value is a location, the low projection is the public view of the node at that location. The low projection of a faceted value $\langle v_H | v_L \rangle$ is the value in the low facet v_L, if it is not a location, or nothing (denoted $\cdot$) when the low facet is empty, as in $\langle v | \cdot \rangle$. When the value in the low facet is a location, the low projection is the public view of the node at that location. The low projection of a labeled value (v, l) is the value v when the label is L, or nothing, otherwise.

$v \downarrow_L^{\sigma} = v'$

$$\frac{v \in \{n, b, \mathsf{dv}, \mathsf{NULL}\}}{v^{\mathsf{std}} \downarrow_L^{\sigma} = v^{\mathsf{std}}} \qquad \frac{\phi = \sigma(a)}{a \downarrow_L^{\sigma} = \phi \downarrow_L^{\sigma}} \qquad \frac{v_L \in \{n, b, \mathsf{dv}, \mathsf{NULL}\}}{\langle \_ | v_L \rangle \downarrow_L^{\sigma} = v_L} \qquad \frac{\phi = \sigma(a_L)}{\langle \_ | a_L \rangle \downarrow_L^{\sigma} = \phi \downarrow_L^{\sigma}} \qquad \frac{}{\langle \_ | \cdot \rangle \downarrow_L^{\sigma} = \cdot}$$

$$\frac{v \in \{n, b, \mathsf{dv}, \mathsf{NULL}\}}{(v, L) \downarrow_L^{\sigma} = v} \qquad \frac{\phi = \sigma(a)}{(a, L) \downarrow_L^{\sigma} = \phi \downarrow_L^{\sigma}} \qquad \frac{}{(v, H) \downarrow_L^{\sigma} = \cdot}$$

No-context projection is useful for proofs. It returns the same value.

$v \downarrow_{\cdot} = v'$

$$\frac{}{v \downarrow_{\cdot} = v}$$

Value (strong) low-equivalence. Two values are (strong) low-equivalent when they are both low-equivalent and have publicly observable interpretations. Having *publicly observable interpretations* has different meanings, depending on the format of the value. Standard values and faceted values always have public interpretations: standard values are, themselves, public, while faceted values have a public facet (or the default value). Tainted values only have public interpretations if they themselves are publicly observable (i.e. their label is at or below L).

This distinction is important for the proofs, where tainted values might introduce secrets to the public context and lead to implicit leaks, whereas standard and faceted values will not.

Unstructured EH storage

$v_1 \simeq_L v_2$

$$\frac{v_1^{\mathcal{I}} \approx_L v_2^{\mathcal{I}} \qquad \mathcal{I} \in \{\mathsf{std}, \mathsf{MF}, \mathsf{FS}\}}{v_1^{\mathcal{I}} \simeq_L v_2^{\mathcal{I}}} \qquad \frac{(v_1, l_1) \approx_L (v_2, l_2) \qquad l_1 \sqsubseteq L \qquad l_2 \sqsubseteq L}{(v_1, l_1) \simeq_L (v_2, l_2)}$$

Tree-structured EH storage

$v_1 \simeq_L^{\sigma_1, \sigma_2} v_2$

$$\frac{v_1^{\mathcal{I}} \approx_L^{\sigma_1, \sigma_2} v_2^{\mathcal{I}} \qquad \mathcal{I} \in \{\mathsf{std}, \mathsf{MF}, \mathsf{FS}\}}{v_1^{\mathcal{I}} \simeq_L^{\sigma_1, \sigma_2} v_2^{\mathcal{I}}} \qquad \frac{(v_1, l_1) \approx_L^{\sigma_1, \sigma_2} (v_2, l_2) \qquad l_1 \sqsubseteq L \qquad l_2 \sqsubseteq L}{(v_1, l_1) \simeq_L^{\sigma_1, \sigma_2} (v_2, l_2)}$$

E.4.4. Node equivalence. ϕ low-equivalence is defined differently depending on the structure of the EH storage.

Unstructured EH storage. Publicly observable nodes ϕ are low-equivalent in an unstructured EH storage if their low projections (their publicly observable parts) are the same. The low-projection of a secret EH storage node is denoted $\cdot$ since none of it is publicly observable.

$$\phi \approx_L \phi' \text{ iff } \phi \downarrow_L = \phi' \downarrow_L$$

The low projection of a node ϕ in an unstructured EH storage is defined differently for different enforcement mechanisms, $\mathcal{G}$. The publicly observable parts of a standard node ϕ^{std} are its value and publicly observable event handlers $M \downarrow_L$. For a faceted node ϕ^{FS}, the publicly observable parts are the low projection of the value in the node $v \downarrow_L$, and the publicly observable event handlers $M \downarrow_L$. If the value in the node is not publicly observable (i.e. $v \downarrow_L = \cdot$), then the entire node is considered secret. Finally, the public view of a tainted node ϕ^{TS} is the public view of its value $(v, l) \downarrow_L$ and event handler map $M \downarrow_L$. If the label on the node itself is H, the entire node is considered secret. If the value in a public node is considered secret, it is replaced with a default value dv in the public view.

$\boxed{\phi^{\mathcal{G}} \downarrow_L = \phi^{\mathsf{std}}}$

$$\frac{\phi = (v, M)}{\phi^{\mathsf{std}} \downarrow_L = (v, M \downarrow_L)} \qquad \frac{\phi = (v, M) \qquad v \downarrow_L \neq \cdot}{\phi^{\mathsf{FS}} \downarrow_L = (v \downarrow_L, M \downarrow_L)} \qquad \frac{\phi = (v, M) \qquad v \downarrow_L = \cdot}{\phi^{\mathsf{FS}} \downarrow_L = \cdot}$$

$$\frac{\phi = ((v, l), M, l') \qquad l \sqsubseteq L \qquad l' \sqsubseteq L}{\phi^{\mathsf{TS}} \downarrow_L = (v, M \downarrow_L)} \qquad \frac{\phi = ((v, l), M, l') \qquad l \not\sqsubseteq L \qquad l' \sqsubseteq L}{\phi^{\mathsf{TS}} \downarrow_L = (\mathsf{dv}, M \downarrow_L)} \qquad \frac{\phi = (v, M, l) \qquad l \not\sqsubseteq L}{\phi^{\mathsf{TS}} \downarrow_L = \cdot}$$

$$\frac{}{\mathsf{NULL} \downarrow_L = \mathsf{NULL}} \qquad \frac{l \sqsubseteq L}{(\mathsf{NULL}, l) \downarrow_L = \mathsf{NULL}} \qquad \frac{l \not\sqsubseteq L}{(\mathsf{NULL}, l) \downarrow_L = \cdot}$$

Tree structured EH storage. Denote $N = (id, v^{\mathsf{std}}, M, ID, Ns)$ or $\cdot$ where ID is an id or $\cdot$, and Ns is an ordered list of N's.

The public view of a tree-structured node given store σ, $\phi \downarrow_L^\sigma$, is defined inductively over the node's children. For a standard node ϕ^{std}, the public view includes (1) the node's id, (2) the value stored in the node v, (3) the public event handlers $M \downarrow_L$, (4) the id of the parent, and (5) an ordered list of the public view of each of its children $A \downarrow_L^\sigma$. If the node has no parent ($a_p = \mathsf{NULL}$), then ID in the public view is $\cdot$. The public view of a faceted node ϕ^{FS} includes (1) the node's id, (2) the public view of the value stored in the node $v \downarrow_L$, (3) the public event handlers $M \downarrow_L$, (4) the id of the publicly observable parent, and (5) and ordered list of the public view of each of its children $A \downarrow_L^\sigma$. If the publicly observable parent is NULL ($a_p \downarrow_L = \mathsf{NULL}$), then ID in the public view is $\cdot$. If there is no publicly observable value ($v \downarrow_L = \cdot$) or parent ($a_p \downarrow_L = \cdot$), then the node itself is not publicly observable.

$\boxed{\phi^{\mathcal{G}} \downarrow_L^\sigma = N}$

$$\frac{\phi = (id, v, M, a_p, A) \qquad id_p = \sigma(a_p).id}{\phi^{\mathsf{std}} \downarrow_L^\sigma = (id, v \downarrow_L^\sigma, M \downarrow_L, id_p, A \downarrow_L^\sigma)} \qquad \frac{\phi = (id, v, M, \mathsf{NULL}, A)}{\phi^{\mathsf{std}} \downarrow_L^\sigma = (id, v \downarrow_L^\sigma, M \downarrow_L, \cdot, A \downarrow_L^\sigma)}$$

$$\frac{\phi = (id, v, M, a_p, A) \qquad a_p \downarrow_L \neq \cdot \qquad id_p = \sigma(a_p \downarrow_L).id \qquad v \downarrow_L \neq \cdot}{\phi^{\mathsf{FS}} \downarrow_L^\sigma = (id, v \downarrow_L^\sigma, M \downarrow_L, id_p, A \downarrow_L^\sigma)}$$

$$\frac{\phi = (id, v, M, a_p, A) \qquad a_p = \mathsf{NULL} \vee a_p \downarrow_L = \mathsf{NULL} \qquad v \downarrow_L \neq \cdot}{\phi^{\mathsf{FS}} \downarrow_L^\sigma = (id, v \downarrow_L^\sigma, M \downarrow_L, \cdot, A \downarrow_L^\sigma)}$$

$$\frac{\phi = (id, v, M, a_p, A) \qquad v \downarrow_L = \cdot \vee a_p \downarrow_L = \cdot}{\phi^{\mathsf{FS}} \downarrow_L^\sigma = \cdot} \qquad \frac{}{\mathsf{NULL} \downarrow_L^\sigma = \mathsf{NULL}}$$

The public view of a list of nodes $A \downarrow_L^\sigma$ from store σ is defined inductively on the structure of the list and produces an ordered list of publicly observable nodes Ns. If the first element in the list is a standard address a (i.e. not faceted), then the public view of the list is the public view of the node at that address $\phi \downarrow_L^\sigma$, followed by the public view of the rest of the list. If the first address in the list is faceted a^{FS}, then the public view of the list is public view of the node $\phi \downarrow_L^\sigma$ at the public view of the address $a \downarrow_L$ followed by the public view of the rest of the list. If the address is not publicly visible (i.e. $a \downarrow_L = \cdot$), then that address is ignored.

$\boxed{A \downarrow_L^\sigma = Ns}$

$$\frac{\phi = \sigma(a)}{(a :: A) \downarrow_L^\sigma = \phi \downarrow_L^\sigma :: A \downarrow_L^\sigma} \qquad \frac{a \downarrow_L \neq \cdot \qquad \phi = \sigma(a \downarrow_L)}{(a^{\mathsf{FS}} :: A^{\mathsf{FS}}) \downarrow_L^\sigma = \phi^{\mathsf{FS}} \downarrow_L^\sigma :: A^{\mathsf{FS}} \downarrow_L^\sigma} \qquad \frac{a \downarrow_L = \cdot}{(a^{\mathsf{FS}} :: A^{\mathsf{FS}}) \downarrow_L^\sigma = A^{\mathsf{FS}} \downarrow_L^\sigma} \qquad \frac{}{(\cdot) \downarrow_L^\sigma = \cdot}$$

E.4.5. Event handler map projection. The low projection of an event handler map M is defined inductively over the structure of the map. For one event $Ev \mapsto \mathit{EH}$, it is defined as the low projection of the event handler sets EH for each event Ev. Events which do not have publicly observable event handlers (i.e. $\mathit{EH} \downarrow_L = \emptyset$) are ignored.

$$\frac{\mathit{EH} \downarrow_L = \mathit{EH}' \neq \emptyset}{((Ev \mapsto \mathit{EH}), M) \downarrow_L = Ev \mapsto \mathit{EH}', M \downarrow_L} \qquad \frac{\mathit{EH} \downarrow_L = \emptyset}{((Ev \mapsto \mathit{EH}), M) \downarrow_L = M \downarrow_L} \qquad \frac{}{\cdot \downarrow_L = \cdot}$$

The low projection of an event handler set EH is the set of publicly observable event handlers in EH. An event handler is public if it was registered under a public pc (i.e. $pc \sqsubseteq L$). In the projected set, all publicly observable event handlers have $pc = L$. Secret event handlers (i.e. $pc \not\sqsubseteq L$) are ignored.

$$\frac{pc \sqsubseteq L}{(\{eh, pc\} \cup \mathit{EH}) \downarrow_L = \{eh, L\} \cup \mathit{EH} \downarrow_L} \qquad \frac{pc \not\sqsubseteq L}{(\{eh, pc\} \cup \mathit{EH}) \downarrow_L = \mathit{EH} \downarrow_L} \qquad \frac{}{\emptyset \downarrow_L = \emptyset}$$

Appendix F.
Security and Weak Secrecy Proofs

F.1. Progress-Insensitive Security implies Weak Secrecy

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

Proof.
We want to show that the conditions for weak secrecy hold for any trace satisfying standard security.
Let $\mathcal{V}, G$ be progress-insensitive secure and σ_0^G be well-formed.
Let T, α, K be s.t. $(G, \mathcal{P} \vdash T \overset{\alpha}{\Longrightarrow} K) \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{P})$. Then,
(1) If $\mathsf{rlsA}(G, \mathcal{P} \vdash \mathsf{last}(T) \overset{\alpha}{\Longrightarrow} K)$:
$\mathcal{K}(G, \mathcal{P} \vdash T \overset{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_\preceq \mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha)$
(2) Otherwise:
$\mathcal{K}(G, \mathcal{P} \vdash T \overset{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_\preceq \mathcal{K}_{p}(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$

We examine each case of $G, \mathcal{P} \vdash T \overset{\alpha}{\Longrightarrow} K$. For each case, we want to show that the corresponding condition for weak secrecy holds.
Case I: $\mathsf{rlsA}(\mathsf{last}(T) \overset{\alpha}{\Longrightarrow} K)$
This case follows from (1) since the corresponding case for release events is the same for weak secrecy as it is for standard security.

Case II: $\mathsf{wkA}(\mathsf{last}(T) \overset{\alpha}{\Longrightarrow} K)$
From the assumption, we want to show
$\mathcal{K}(G, \mathcal{P} \vdash T \overset{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_\preceq \mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha)$
By assumption and since visible, non-release events fall into the "other" category from (2),
$\mathcal{K}(G, \mathcal{P} \vdash T \overset{\alpha}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_\preceq \mathcal{K}_{p}(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$
Then, it is sufficient to show that:
$\mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P}) \supseteq_\preceq \mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha)$
i.e., for any $\tau \in \mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha), \exists \tau' \in \mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$ s.t. $\tau \preceq \tau'$
Let
(II.1) $\tau \in \mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha)$
From (II.1),
(II.2) $\exists T' \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{P})$ s.t. $T \approx_L T'$ and
(II.3) $\tau = \mathsf{in}(T')$
(II.4) $\mathsf{prog}(T')$
From (II.2)-(II.4),
(II.5) $\tau \in \mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$
Let
(II.6) $\tau' = \tau$
From (II.5) and (II.6),
$\tau' \in \mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P})$
From (II.6) and the definition of $\preceq$,
$\tau \preceq \tau'$

Case III: $\neg\mathsf{rlsA}(\mathsf{last}(T) \overset{\alpha}{\Longrightarrow} K)$ and $\neg\mathsf{wkA}(\mathsf{last}(T) \overset{\alpha}{\Longrightarrow} K)$
This case follows from (2) since the corresponding case for non-release events is the same for standard security as it is for the non-release, non-visible events for weak secrecy.

□

F.2. Lemma dependency graph

Top-level theorems
PINI implies PI weak secrecy: Theorem 2
None!
Progress-insensitive Noninterference: Theorem 2
(T1) Lemma 7 Equivalent trace, equivalent state
(T4) Lemma 17 Strong One-Step
Weak Secrecy: Theorem 4
(WT1) Lemma 8 Equivalent trace, equivalent state (Weak Secrecy)

(WT4) Lemma 23 Strong One-Step (Weak Secrecy)

Trace Requirements

(T1) Lemma 7 Equivalent trace, equivalent state
- **(T5)** Lemma 28 Weak One-Step
- **(T2)** Lemma 9 Empty traces produce L-equivalent states

(WT1) Lemma 8 Equivalent trace, equivalent state (Weak secrecy)
- **(WT5)** Lemma 31 Weak One-Step (Weak secrecy)
- **(WT2)** Lemma 12 Empty traces produce L-equivalent states (Weak secrecy)

(T2) Lemma 9 Empty traces produce L-equivalent states
- **(V2)** Lemma 41 H assignments are unobservable
- **(EH2)** Lemma 66 H EH lookups are unobservable
- **(EH3)** Lemma 73, Lemma 74 H updates are unobservable

(WT2) Lemma 12 Empty traces produce L-equivalent states (Weak secrecy)
- **(WV2)** Lemma 42 H assignments are unobservable (Weak secrecy)
- **(EH2)** Lemma 66 H EH lookups are unobservable
- **(WEH3)** Lemma 76, Lemma 79 H updates are unobservable (Weak secrecy)

(T3) Lemma 15, Lemma 16 H steps produce L equivalent states and empty traces
- **(EH2)** Lemma 66 H EH lookups are unobservable
- **(T2)** Lemma 10 Empty traces produce L-equivalent states (mid-level)

(WT3) *not required!*

(T4) Lemma 17 Strong One-Step
- **(EH1)** Lemma 56, Lemma 63 L lookups are equivalent
- **(T3)** Lemma 15, Lemma 16 H steps produce L-equivalent states and empty traces
- **(EH2)** Lemma 66 H EH lookups are unobservable
- **(E1)** Lemma 36 L Lookups are equivalent
- **(V3)** Lemma 44 L Assignments are equivalent
- **(EH4)** Lemma 80, Lemma 81 L Updates are equivalent

(WT4) Lemma 23 Strong One-Step (Weak Secrecy)
- **(EH1)** Lemma 56, Lemma 63 L lookups are equivalent
- **(WT2)** Lemma 12 Empty traces produce L-equivalent states (Weak Secrecy)
- **(T4)** Lemma 20, Lemma 22, and Lemma 21 Strong One-Step Lemma
- **(EH2)** Lemma 66 H EH lookups are unobservable
- **(WE1)** Lemma 37 L lookups are equivalent (Weak Secrecy)
- **(WV3)** Lemma 45 L assignments are equivalent (Weak Secrecy)
- **(WEH4)** Lemma 82, Lemma 83 L Updates are equivalent (Weak Secrecy)

(T5) Lemma 28 Weak One-Step
- **(EH1)** Lemma 56, Lemma 63 L lookups are equivalent
- **(EH2)** Lemma 66 H EH lookups are unobservable
- **(E1)** Lemma 36 L lookups are equivalent
- **(V3)** Lemma 44 L assignments are equivalent
- **(EH4)** Lemma 80, Lemma 81 L updates are equivalent

(WT5) Lemma 31 Weak One-Step (Weak Secrecy)
- **(EH1)** Lemma 56, Lemma 63 L lookups are equivalent
- **(EH2)** Lemma 66 H EH lookups are unobservable
- **(WV1)** Lemma 37
- **(WV2)** Lemma 43 H assignments are unobservable
- **(WEH3)** Lemma 75, Lemma 77, Lemma 78 H updates are unobservable

Expression Requirements

(E1) Lemma 36 L expressions are equivalent
- **(V1)** Lemma 38, Lemma 39 L lookups are equivalent
- **(EH1)** Lemma 46 L lookups are equivalent

(WE1) Lemma 37 L expressions are equivalent (Weak Secrecy)
- **(WV1)** Lemma 40 L lookups are equivalent (Weak Secrecy)
- **(WEH1)** Lemma 64 L lookups are equivalent (Weak Secrecy)

Variable Store Requirements

(V1) Lemma 38, Lemma 39 L lookups are equivalent
- None!

(WV1) Lemma 40 L lookups are equivalent (Weak Secrecy)
- None!

(V2) Lemma 41 H assignments produce L-equivalent states

None!
(WV2) Lemma 42, Lemma 43 H assignments produce L-equivalent states (Weak Secrecy)
None!
(V3) Lemma 44 L assignments are equivalent
None!
(WV3) Lemma 45 L assignments are equivalent (Weak Secrecy)
None!

EH storage Requirements
(EH1) Lemma 46, Lemma 48, Lemma 49, Lemma 56 L lookups are equivalent
(EH2) Lemma 66, Lemma 67, Lemma 69, Lemma 70, Lemma 71, Lemma 72 H EH lookups are unobservable
(WEH1) Lemma 64 L lookups are equivalent (Weak Secrecy)
(EH1) Lemma 48 (L lookups are equivalent)
(EH2) Lemma 66, Lemma 67, Lemma 69, Lemma 70, Lemma 71, Lemma 72 H EH lookups are unobservable
None!
(WE2) *not required!*
(EH3) Lemma 73, Lemma 74 H updates are unobservable
None!
(WEH3) Lemma 75, Lemma 76, Lemma 77, Lemma 78, Lemma 79 H updates are unobservable (Weak Secrecy)
None!
(EH4) Lemma 80, Lemma 81 L updates are equivalent
(EH1) Lemma 48, Lemma 49 L lookups are equivalent
(EH3) Lemma 73, Lemma 74 H updates are unobservable
(WEH4) Lemma 82, Lemma 83 L updates are equivalent (Weak Secrecy)
(EH1) Lemma 48 L lookups are equivalent

F.3. Top-Level Security and Weak Secrecy

Theorem 2 (Soundness - Security). *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.* $G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{P}, \mathcal{I})$*, and* σ_0^G *is well-formed,*

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

Proof.
The proof is split between two cases depending on the action, shown below. In either case, we want to show that $\exists \tau' \in \mathcal{K}(G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$ s.t. $\tau \preceq \tau'$ for τ defined below

Case I: $\mathsf{rlsA}(G, \mathcal{P} \vdash \mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K)$
Let
(I.1) $\tau \in \mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_l)$
(I.2) $\alpha' = (G, \mathcal{P} \vdash \mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K) \downarrow_L$
$\exists T_1, K_0, K_1$ s.t.
(I.3) $T_1 = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K_1$ and
(I.4) $\tau = \mathsf{in}(T_1)$
From definition $\mathcal{K}_{rp}()$,
(I.5) $T_1 \approx_L T$
(I.6) $\mathsf{prog}(T_1)$
(I.7) $\mathsf{release}(T_1, \alpha')$
From (I.7), $\exists K_1', \alpha_{l,1}$ s.t.
(I.8) $G, \mathcal{P} \vdash T_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1'$ with
(I.9) $(G, \mathcal{P} \vdash K_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1') \downarrow_L = \alpha'$
From (I.5),
(I.10) $T \downarrow_L = T_1 \downarrow_L$
From (I.2), (I.9), (I.10), and the definition of $\approx_L$,
(I.11) $(G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K) \approx_L (G, \mathcal{P} \vdash T_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1')$
From (I.11) and the definition of $\mathcal{K}()$,
$\mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1') \in \mathcal{K}(G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$
Let
(I.12) $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1')$

From (I.12), and (I.4),
$\tau \preceq \tau'$

Case II: $\neg\mathsf{rlsA}(G, \mathcal{P} \vdash \mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K)$
Let
(II.1) $\tau \in \mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_l)$
$\exists T_1, K_0, K_1, K_2$ s.t.
(II.2) $T_1 = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K_1$ and
(II.3) $\tau = \mathsf{in}(T_1)$
(II.4) $T = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K_2$ and
(II.5) $G, \mathcal{P} \vdash K_2 \overset{\alpha_l}{\Longrightarrow} K$
From definition $\mathcal{K}_{rp}()$,
(II.6) $T_1 \approx_L T$
(II.7) $\mathsf{prog}(T_1)$
From (II.6),
(II.8) $T \downarrow_L = T_1 \downarrow_L$

Subcase i: $(G, \mathcal{P} \vdash \mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K) \downarrow_L = \cdot$
By assumption,
(i.1) $T \approx_L (G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K)$
From (II.4) and (i.1),
(i.2) $T_1 \approx_L (G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K)$
Let
(i.3) $\tau' = \mathsf{in}(T_1)$
From (i.2) and (i.3),
$\tau' \in \mathcal{K}(G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$
From (II.3) and (i.3),
$\tau \preceq \tau'$

Subcase ii: $(G, \mathcal{P} \vdash \mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K) \downarrow_L \neq \cdot$
From (II.2), (II.6), (II.4), and Lemma 7 (Requirement **(T1)**),
(ii.1) $K_1 \approx_L K_2$
By assumption, (II.2), (II.4), (II.6), (ii.1), (II.5), (II.7), and Lemma 17 (Requirement **(T4)**),
(ii.2) $\exists K_1', \tau''$ s.t. $G, \mathcal{P} \vdash K_1 \overset{\tau''}{\Longrightarrow}^* K_1'$ and
(ii.3) $(G, \mathcal{P} \vdash K_1 \overset{\tau''}{\Longrightarrow}^* K_1') \approx_L (G, \mathcal{P} \vdash K_2 \overset{\alpha_l}{\Longrightarrow} K)$
By (II.6) and (ii.3),
(ii.4) $(G, \mathcal{P} \vdash T_1 \overset{\tau''}{\Longrightarrow}^* K_1') \approx_L (G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K)$
By (ii.4) and the definition of $\mathcal{K}()$,
(ii.5) $\mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \overset{\tau''}{\Longrightarrow}^* K_1') \in \mathcal{K}(G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$
Let
(ii.6) $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \overset{\tau''}{\Longrightarrow}^* K_1')$
By (ii.5) and (ii.6),
$\tau' \in \mathcal{K}(G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$
By (II.3) and (ii.6),
$\tau \preceq \tau'$

□

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.* $G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K \in \mathsf{runs}(\sigma_0^G, \mathcal{R}, \mathcal{P}, \mathcal{I})$*, and* σ_0^G *is well-formed,*

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

Proof.

The proof is split between two cases depending on the action, shown below. In either case, we want to show that $\exists \tau' \in \mathcal{K}(G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$ s.t. $\tau \preceq \tau'$ for τ defined below

Case I: $\mathsf{rlsA}(G, \mathcal{P} \vdash \mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K)$
Let
(I.1) $\tau \in \mathcal{K}_{rp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_l)$
(I.2) $\alpha' = (G, \mathcal{P} \vdash \mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K) \downarrow_L$
$\exists T_1, K_0, K_1$ s.t.
(I.3) $T_1 = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K_1$ and
(I.4) $\tau = \mathsf{in}(T_1)$
From definition $\mathcal{K}_{rp}()$,
(I.5) $T_1 \approx_L T$
(I.6) $\mathsf{prog}(T_1)$
(I.7) $\mathsf{release}(T_1, \alpha')$
From (I.7), $\exists K_1', \alpha_{l,1}$ s.t.
(I.8) $G, \mathcal{P} \vdash T_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1'$ with
(I.9) $(G, \mathcal{P} \vdash K_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1') \downarrow_L = \alpha'$
From (I.5),
(I.10) $T \downarrow_L = T_1 \downarrow_L$
From (I.2), (I.9), (I.10), and the definition of $\approx_L$,
(I.11) $(G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K) \approx_L (G, \mathcal{P} \vdash T_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1')$
From (I.11) and the definition of $\mathcal{K}()$,
$\mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1') \in \mathcal{K}(G, \mathcal{P}\mathcal{P}T \overset{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$
Let
(I.12) $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1')$
From (I.12), and (I.4),
$\tau \preceq \tau'$

Case II: $\mathsf{wkA}(G, \mathcal{P} \vdash \mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K)$
Let
(II.1) $\tau \in \mathcal{K}_{wp}(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_l)$
(II.2) $\alpha' = (G, \mathcal{P} \vdash \mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K) \downarrow_L$
$\exists T_1, K_0, K_1, K_2$ s.t.
(II.3) $T_1 = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K_1$ and
(II.4) $\tau = \mathsf{in}(T_1)$
(II.5) $T = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K_2$
From definition $\mathcal{K}_{wp}()$,
(II.6) $T_1 \approx_L T$
(II.7) $\mathsf{prog}(T_1)$
(II.8) $\mathsf{wkT}(T_1, \alpha')$

Subcase i: $\alpha' = \mathsf{br}(b)$
By assumption and from (II.8), $\exists K_1'$ s.t.
(i.1) $G, \mathcal{P} \vdash T_1 \Longrightarrow K_1'$ with
(i.2) $(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1') \downarrow_L = \alpha'$
From (II.6)
(i.3) $T \downarrow_L = T_1 \downarrow_L$
From (II.2), (i.2), (i.3), and the definition of $\approx_L$,
(i.4) $(G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K) \approx_L (G, \mathcal{P} \vdash T_1 \Longrightarrow K_1')$
From (i.4) and the definition of $\mathcal{K}()$,
$\mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1') \in \mathcal{K}(G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$
Let
(i.5) $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1')$
From (i.5) and (II.4),
$\tau \preceq \tau'$

Subcase ii: $\alpha' = \mathsf{gw}(x)$
By assumption and from (II.8), $\exists K_1', K_1''$ s.t.
(ii.1) $G, \mathcal{P} \vdash T_1 \Longrightarrow^* K_1'$ with

(ii.2) $(G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1') \downarrow_L = \cdot$
(ii.3) $(G, \mathcal{P} \vdash K_1' \Longrightarrow^* K_1'') \downarrow_L = \alpha'$
From (II.6)
(ii.4) $T \downarrow_L = T_1 \downarrow_L$
From (II.2), (ii.2)-(ii.4), and the definition of $\approx_L$,
(ii.5) $(G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K) \approx_L (G, \mathcal{P} \vdash T_1 \Longrightarrow K_1')$
From (ii.5) and the definition of $\mathcal{K}()$,
$\mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1'') \in \mathcal{K}(G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$
Let
(ii.6) $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \Longrightarrow K_1'')$
From (ii.6) and (II.4),
$\tau \preceq \tau'$

Subcase iii: $\alpha' = id.Ev(v)$ or $\alpha' \in \{\bullet, ch(v)\}$
From (II.3), (II.5), (II.6), and Lemma 8 (Requirement **(WT1)**),
(iii.1) $K_1 \approx_L K_2$
By assumption and from (II.3), (II.5), (iii.1), (II.2), (II.6)-(II.8), and Lemma 23 (Requirement **(WT4)**), $\exists K_1'$ s.t.
(iii.2) $G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'$ with
(iii.3) $(G, \mathcal{P} \vdash K_2 \Longrightarrow K) \approx_L (G, \mathcal{P} \vdash K_1 \Longrightarrow K_1')$
From (II.6) and (iii.3),
(iii.4) $(G, \mathcal{P} \vdash T_1 \Longrightarrow^* K_1') \approx_L (G, \mathcal{P} \vdash T \Longrightarrow K)$
From (ii.4) and the definition of $\mathcal{K}()$,
(iii.5) $\mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \overset{\tau''}{\Longrightarrow}{}^* K_1') \in \mathcal{K}(G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$
Let
(iii.6) $\tau' = \mathsf{in}(T_1) :: \mathsf{in}(G, \mathcal{P} \vdash K_1 \overset{\tau''}{\Longrightarrow}{}^* K_1')$
By (iii.5) and (iii.6),
$\tau' \in \mathcal{K}(G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$
By (II.4) and (iii.6),
$\tau \preceq \tau'$

Case III: $\neg\mathsf{rlsA}(G, \mathcal{P} \vdash \mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K)$ and $\neg\mathsf{wkA}(G, \mathcal{P} \vdash \mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K)$
Let
(III.1) $\tau \in \mathcal{K}_p(T, \sigma_0^G, \mathcal{R}, \mathcal{P}, \alpha_l)$
$\exists T_1, K_0, K_1, K_2$ s.t.
(III.2) $T_1 = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K_1$ and
(III.3) $\tau = \mathsf{in}(T_1)$
(III.4) $T = G, \mathcal{P} \vdash K_0 \Longrightarrow^* K_2$ and
(III.5) $G, \mathcal{P} \vdash K_2 \overset{\alpha_l}{\Longrightarrow} K$
From definition $\mathcal{K}_{rp}()$,
(III.6) $T_1 \approx_L T$
(III.7) $\mathsf{prog}(T_1)$
From (III.6),
(III.8) $T \downarrow_L = T_1 \downarrow_L$

Subcase i: $(G, \mathcal{P} \vdash \mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K) \downarrow_L = \cdot$
By assumption,
(i.1) $T \approx_L (G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K)$
From (III.4) and (i.1),
(i.2) $T_1 \approx_L (G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K)$
Let
(i.3) $\tau' = \mathsf{in}(T_1)$
From (i.2) and (i.3),
$\tau' \in \mathcal{K}(G, \mathcal{P} \vdash T \overset{\alpha_l}{\Longrightarrow} K, \sigma_0^G, \mathcal{R}, \mathcal{P})$
From (III.3) and (i.3),
$\tau \preceq \tau'$

Subcase ii: $(G, \mathcal{P} \vdash \mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K) \downarrow_L \neq \cdot$
By assumption, and from the definition of $\downarrow_L$, α_l is a release event, branch, global write, low input, or low

output. Then, $\mathsf{rlsA}(G, \mathcal{P} \vdash \mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K)$ or $\mathsf{wkA}(G, \mathcal{P} \vdash \mathsf{last}(T) \overset{\alpha_l}{\Longrightarrow} K)$, which violates the assumption that neither condition holds.
Therefore, this case holds vacuously.

□

F.4. Trace Requirements

Requirement (T1) Equivalent traces produce L-equivalent states

Lemma 7 (Equivalent Trace, Equivalent State)**.** *If* $T_1 = G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'$ *and* $T_2 = G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_2'$ *with* $K_1 \approx_L K_2$ *and* $T_1 \approx_L T_2$*, then* $K_1' \approx_L K_2'$

Proof.
By induction on $\mathsf{len}(T_1)$ and $\mathsf{len}(T_2)$
By assumption,
(1) $T_1 = G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'$
(2) $T_2 = G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_2'$
(3) $K_1 \approx_L K_2$
(4) $T_1 \approx_L T_2$

Base Case I: $\mathsf{len}(T_1) = 0$ and $\mathsf{len}(T_2) = n$
By assumption and from (1),
(I.1) $T_1 = K_1$
(I.2) $K_1 = K_1'$
From (I.1),
(I.3) $T_1 \downarrow_L = \cdot$
From (4) and (I.3),
(I.4) $T_2 \downarrow_L = \cdot$
From (I.4) and Lemma 9 (Requirement **(T2)**),
(I.5) $K_2 \approx_L K_2'$
From (3), (i.2), and (I.5),
$K_1' \approx_L K_2'$

Base Case II: $\mathsf{len}(T_1) = n$ and $\mathsf{len}(T_2) = 0$
The proof is similar to **Base Case I**

Inductive Case III: $\mathsf{len}(T_1) = n + 1$ and $\mathsf{len}(T_2) = m + 1$
We assume the conclusion holds for $\mathsf{len}(T_1) \leq n$ and $\mathsf{len}(T_2) \leq m$
By assumption and from (1) and (2),
(III.1) $T_1 = G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'' \Longrightarrow K_1'$ with
(III.2) $\mathsf{len}(G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'') = n$
(III.3) $T_2 = G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_2'' \Longrightarrow K_2'$ with
(III.4) $\mathsf{len}(G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_2'') = m$

Subcase i: $(G, \mathcal{P} \vdash K_1'' \Longrightarrow K_1') \downarrow_L = \cdot$
By assumption and from (III.1),
(i.1) $T_1 \downarrow_L = (G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'') \downarrow_L$
From (i.1),
(i.2) $T_1 \approx_L (G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'')$
From (4) and (i.2),
(i.3) $T_2 \approx_L (G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'')$
From (3) and (i.3),
The IH may be applied on $(G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'')$ and T_2
IH on $(G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'')$ and T_2 gives
(i.4) $K_1'' \approx_L K_2'$
By assumption and from Lemma 9 (Requirement **(T2)**),
(i.5) $K_1'' \approx_L K_1'$
From (i.4) and (i.5),
$K_1' \approx_L K_2'$

Subcase ii: $(G, \mathcal{P} \vdash K_2'' \Longrightarrow K_2') \downarrow_L = \cdot$
The proof is similar to **Subcase i**

Subcase iii: $(G, \mathcal{P} \vdash K_1'' \Longrightarrow K_1') \downarrow_L \neq \cdot$ and $(G, \mathcal{P} \vdash K_2'' \Longrightarrow K_2') \downarrow_L \neq \cdot$

By assumption and from (4),

(iii.1) $(G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'') \approx_L (G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_2'')$ and

(iii.2) $(G, \mathcal{P} \vdash K_1'' \Longrightarrow K_1') \approx_L (G, \mathcal{P} \vdash K_2'' \Longrightarrow K_2')$

From (3) and (iii.1),

The IH may be applied to $(G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'')$ and $(G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_2'')$

IH on $(G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'')$ and $(G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_2'')$ gives,

(iii.3) $K_1'' \approx_L K_2''$

By assumption and from (iii.2), (iii.3), and Lemma 28 (Requirement **(T5)**),

$K_1' \approx_L K_2'$

□

Requirement (WT1) Equivalent traces produce L-equivalent states (Weak Secrecy)

Lemma 8 (Equivalent Trace, Equivalent State, Weak Secrecy)**.** *If* $T_1 = G, \mathcal{P} \vdash_w K_1 \Longrightarrow^* K_1'$ *and* $T_2 = G, \mathcal{P} \vdash_w K_2 \Longrightarrow^* K_2'$ *with* $K_1 \approx_L K_2$ *and* $T_1 \approx_L T_2$*, then* $K_1' \approx_L K_2'$

Proof (sketch): The proof is the same as for Lemma 7, except that it uses Lemma 12 (Req **(WT2)**) and Lemma 31 (Req **(WT5)**) □

Requirement (T2) Empty traces produce L-equivalent states

Lemma 9. *If* $T = G, \mathcal{P} \vdash K \Longrightarrow^* K'$ *and* $T \downarrow_L = \cdot$*, then* $K \approx_L K'$

Proof.
By induction on the length of T.
By assumption,
(1) $T = G, \mathcal{P} \vdash K \Longrightarrow^* K'$
(2) $T \downarrow_L = \cdot$

Base Case I: $\mathsf{len}(T) = 0$
By assumption and from (1),
(I.1) $T = K$ and
(I.2) $K' = K$
From (I.2),
$K \approx_L K'$

Inductive Case II: $\mathsf{len}(T) = n + 1$
By assumption and from (1),
(II.1) $T = G, \mathcal{P} \vdash K \Longrightarrow^* K_1 \Longrightarrow K_2$
Want to show $K \approx_L K_2$
From (2) and (II.1),
(II.2) $(G, \mathcal{P} \vdash K \Longrightarrow^* K_1) \downarrow_L = \cdot$
IH on $(G, \mathcal{P} \vdash K \Longrightarrow^* K_1)$ gives
(II.3) $K \approx_L K_1$
Let $T' = G, \mathcal{P} \vdash K_1 \Longrightarrow K_2$
From (1),
(II.4) $T' \downarrow_L = \cdot$
Therefore, from (II.3), want to show $K_1 \approx_L K_2$

Subcase i: T' ends in I-NR1 or I-NR2
By assumption,
(i.1) $\sigma_1^G = \sigma_2^G$
(i.2) $\kappa_1 = \cdot$
(i.3) $G, \mathcal{P}, \sigma_1^G \vdash ks_1; \mathsf{lookupEHAll}(id.Ev(v)) \rightsquigarrow_H ks_2$
From (i.3) and Lemma 66 (Requirement **(EH2)**),
(i.4) $ks_1 \approx_L ks_2$
From (i.1) and (i.4),
$K_1 \approx_L K_2$

Subcase ii: T' ends in I-R-DIFF, I-R-SAME, or I-L,
By assumption and from the definition of $\downarrow_L$ for execution traces,
(ii.1) $(G, ks \vdash K_1 \Longrightarrow K_2) \downarrow_L \neq \cdot$
But (ii.1) contradicts (1), so this case holds vacuously

Subcase iii: T' ends in O, O-SKIP, or O-OTHER with $pc \sqsubseteq L$
By assumption and from the definition of $\downarrow_L$ for execution traces,
(iii.1) $(G, ks \vdash K_1 \Longrightarrow K_2) \downarrow_L \neq \cdot$
But (iii.1) contradicts (1), so this case holds vacuously

Subcase iv: T' ends in O, O-SKIP, or O-OTHER with $pc \not\sqsubseteq L$
The conclusion follows from our security lattice (i.e., pc must be H) and Lemma 10

□

Lemma 10. *If* $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa^{\mathcal{V}} \xrightarrow{\alpha}_H \sigma_2^G, ks$*, then* $\sigma_1^G \approx_L \sigma_2^G$ *and* $(\mathcal{V}; \kappa^{\mathcal{V}}; H) \approx_L ks$

Proof.
By induction on the structure of $\mathcal{E} :: G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa \xrightarrow{\alpha}_H \sigma_2^G, ks$

Case I: $\mathcal{E}$ ends in LC
By assumption,
(I.1) $ks' = \mathcal{V}; (\sigma^{\mathcal{V}}, \mathsf{skip}, C, \cdot); H$
(I.2) $\sigma_2^G = \sigma_1^G$

(I.3) $G, \mathcal{P}, \sigma^G \vdash ks'; \mathsf{lookupEHs}(E) \rightsquigarrow_H ks$
From Lemma 66 (Requirement **(EH2)**),
(I.4) $ks' \approx_L ks$
From (I.1),
(I.5) $ks' \approx_L (\mathcal{V}; \kappa; H)$
From (I.4) and (I.5),
$(\mathcal{V}; \kappa; H) \approx_L ks$
From (I.2),
$\sigma_2^G \approx_L \sigma_1^G$

Case II: $\mathcal{E}$ ends in PTOC
By assumption,
(II.1) $\sigma_2^G = \sigma_1^G$
(II.2) $ks = \mathcal{V}; (\sigma^{\mathcal{V}}, \mathsf{skip}, C, \cdot); H$
From (II.1),
$\sigma_2^G \approx_L \sigma_1^G$
From (II.2),
$(\mathcal{V}; \kappa; H) \approx_L ks$

Case III: $\mathcal{E}$ ends in P
$\sigma_1^G \approx_L \sigma_2^G$ follows from Lemma 11
By assumption,
$ks = (\mathcal{V}; (\sigma_2^{\mathcal{V}}, c_2, P, (E_1, E_2)); H)$
Therefore,
$(\mathcal{V}; \kappa; H) \approx_L ks$

Case IV: $\mathcal{E}$ ends in SME-L, SME-LTOH, or P-F,
These cases contradict the assumption that $pc = H$, so these cases hold vacuously.

Case V: $\mathcal{E}$ ends in SME-H
By assumption,
(V.1) $\exists \mathcal{E}' :: G, \mathcal{P}, \mathsf{SME}, d \vdash \sigma_1^G, \kappa_H \xrightarrow{\alpha}_H \sigma_2^G, (\mathsf{SME}; \kappa'_H) :: ks'$
(V.2) $ks = (\mathsf{SME}; (\kappa'_H; \kappa_L); H) :: ks'$
IH on (V.1) gives
$\sigma_1^G \approx_L \sigma_2^G$
(V.3) $(\mathsf{SME}; \kappa_H; H) \approx_L ((\mathsf{SME}; \kappa'_H); H) :: ks'$
From (V.2) and (V.3),
$(\mathsf{SME}; \kappa; H) \approx_L ks$

□

Lemma 11. *If* $G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1 \xrightarrow{\alpha}_H \sigma_2^G, \sigma_2^{\mathcal{V}}, c_2, E$, *then* $\sigma_1^G \approx_L \sigma_2^G$

Proof.
By induction on the structure of $\mathcal{E} :: G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1 \xrightarrow{\alpha}_H \sigma_2^G, \sigma_2^{\mathcal{V}}, c_2, E$

The cases where $\sigma_2^G = \sigma_1^G$ and are trivial, the proofs for the other cases are shown below.

Base Case I: $\mathcal{E}$ ends in ASSIGN-G
By assumption and from Lemma 41 (Requirement **(V2)**), $\sigma_1^G \approx_L \sigma_2^G$

Unstructured EH storage:
Base Case U.I: $\mathcal{E}$ ends in ASSIGN-D
By assumption and from Lemma 73.U (Requirement **(EH3)**), $\sigma_1^G \approx_L \sigma_2^G$

Base Case U.II: $\mathcal{E}$ ends in CREATEELEM or ADDEH
By assumption and from Lemma 74.U (Requirement **(EH3)**), $\sigma_1^G \approx_L \sigma_2^G$

Tree-structured EH storage:
Base Case T.I: $\mathcal{E}$ ends in ASSIGN-D
By assumption and from Lemma 73.T (Requirement **(EH3)**), $\sigma_1^G \approx_L \sigma_2^G$

Base Case T.II: $\mathcal{E}$ ends in CREATECHILD, CREATESIBLING, or ADDEH

By assumption and from Lemma 74.T (Requirement **(EH3)**), $\sigma_1^G \approx_L \sigma_2^G$

Inductive Case: $\mathcal{E}$ ends in SEQ
The proof follows from the IH

□

Requirement (WT2) Empty traces produce L-equivalent states (Weak Secrecy)

Lemma 12. *If* $T = G, \mathcal{P} \vdash_w K \Longrightarrow^* K'$ *and* $T \downarrow_L = \cdot$, *then* $K \approx_L K'$

Proof (sketch): The proof is the same as for Lemma 9 (Requirement **(T2)**), except that it uses Lemma 13. The additional assumption that $\alpha \notin \{\mathsf{br}(b), \mathsf{gw}(x)\}$ follows from $T \downarrow_L = \cdot$. □

Lemma 13. *If* $G, \mathcal{P}, \mathcal{V}, d \vdash_w \sigma_1^G, \kappa_1 \xrightarrow{\alpha}_H \sigma_2^G, \kappa_2$ *with* $\alpha \notin \{\mathsf{br}(b), \mathsf{gw}(x)\}$, *then* $\sigma_1^G \approx_L \sigma_2^G$ *and* $\kappa_1 \approx_L \kappa_2$

Proof (sketch): The proof is the same as for Lemma 10 except that is uses Lemma 14, Note that Lemma 66 (Requirement **(EH2)**) is also used. □

Lemma 14. *If* $G, \mathcal{V}, d \Vdash_w \sigma_1^G, \sigma_1^{\mathcal{V}}, c_1 \xrightarrow{\alpha}_H \sigma_2^G, \sigma_2^{\mathcal{V}}, c_2, E$, *with* $\alpha \notin \{\mathsf{br}(b), \mathsf{gw}(x)\}$ *then* $\sigma_1^G \approx_L \sigma_2^G$ *and* $\sigma_1^{\mathcal{V}} \approx_L \sigma_2^{\mathcal{V}}$ *and* $E \approx_L \cdot$

Proof (sketch): The proof is similar to the one for Lemma 11. The cases for ASSIGN-G and ASSIGN-D which involve upgrades hold vacuously due to the assumption that $\alpha \notin \{\mathsf{br}(b), \mathsf{gw}(x)\}$. Otherwise, this proof uses Lemma 42 (Requirement **(WV2)**) instead of Lemma 41 (Requirement **(V2)**) and Lemma 76 (Requirement **(WEH3)**) instead of Lemma 73 (Requirement **(EH3)**), and Lemma 79 (Requirement **(WEH3)**) instead of Lemma 74 (Requirement **(EH3)**). □

Requirement (T3) H steps produce L-equivalent states and empty traces

Lemma 15 (H Step Equivalence). *If* $T = G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_2$ *and* $\forall \alpha \in \tau, \mathsf{output}(\alpha)$ *with* $K_1 = \mathcal{R}, d; \sigma_1; ks_1$ *and* $ks_1 \approx_L \cdot$*, then,* $K_1 \approx_L K_2$ *and* $T \downarrow_L = \cdot$

Proof.
By induction on $\mathsf{len}(T)$
By assumption,
(1) $\forall \alpha \in \tau, \mathsf{output}(\alpha)$ and
(2) $ks_1 \approx_L \cdot$

Base Case 1: $\mathsf{len}(T) = 0$
By assumption,
$T = K_1$ and $K_2 = K_1$
Then, $K_1 \approx_L K_2$ and $T \downarrow_L = \cdot$

Base Case 2: $\mathsf{len}(T) = 1$
By assumption, $T = G, \mathcal{P} \vdash K_1 \overset{\alpha_L}{\Longrightarrow} K_2$

Case I: $\mathcal{E}$ ends in an input rule
This contradicts (1), therefore this case holds vacuously

Case II: $\mathcal{E}$ ends in an output rule with $pc \sqsubseteq L$
This contradicts (2), therefore this case holds vacuously

Case III: $\mathcal{E}$ ends in an output rule with $pc \not\sqsubseteq L$
By assumption and from (2),
(III.1) $pc = H$
(III.2) $\mathcal{E}$ ends in O, O-SKIP, or O-OTHER
From (III.2),
(III.3) $\exists \mathcal{E}' :: G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa \longrightarrow_H \sigma_2^G, ks$
From (III.1), (III.2), and the definitions of outCondition and output,
$T \downarrow_L = \cdot$
From (III.3) and Lemma 10 (Requirement **(T2)**),
(III.4) $\sigma_1^G \approx_L \sigma_2^G$
(III.5) $(\mathcal{V}; \kappa; H) \approx_L ks$
From (III.4) and (III.5),
$K_1 \approx_L K_2$

Inductive Case: $\mathsf{len}(T) = n + 1$
By assumption,
$T = G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1' \Longrightarrow K_2$
IH on $G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1'$ gives
$K_1 \approx_L K_1'$ and $(G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_1') \downarrow_L = \cdot$
From $K_1 \approx_L K_1'$ and (2),
The ks in K_1' is $\approx_L \cdot$
By the same argument as **Base Case 2**,
$K_1' \approx_L K_2$ and $(G, \mathcal{P} \vdash K_1' \Longrightarrow K_2) \downarrow_L = \cdot$
Therefore, the desired conclusion holds

□

Lemma 16 (High Step Equivalence - MF, TT). *If* $T = G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_C$ *with* $\mathsf{consumer}(K_C)$*,* $K_1 = \mathcal{R}, d; \sigma_1; ks_1$ *with* $ks_1 \not\approx_L \cdot$*, and* $\mathsf{highProducer}(K_1)$ *then* $\exists K_2$ *s.t.* $\mathsf{lowProducer}(K_2)$ *and* $T = G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_2 \Longrightarrow^* K_C$ *with* $K_1 \approx_L K_2$ *and* $(G, \mathcal{P} \vdash K_1 \Longrightarrow^* K_2) \downarrow_L = \cdot$

Proof.
By induction on the length of T
By assumption,
(1) $\mathsf{consumer}(K_C)$
(2) $ks_1 \not\approx_L \cdot$
(3) $\mathsf{highProducer}(K_1)$

Base Case: $\mathsf{len} = 0$

By assumption, $T = K_1 = K_C$. From this and (2), $\mathsf{consumer}(K_1)$. But this contradicts (2) and (3), so this case holds vacuously.

Base Case: $\mathsf{len} = 1$
By assumption, $\mathcal{E} :: T = G, \mathcal{P} \vdash K_1 \Longrightarrow K_C$. From this and the structure of the operational semantics, $\mathcal{E}$ must end in O-NEXT with $ks_1 = (\mathcal{V}; \kappa; pc)$ and $\mathsf{consumer}(\kappa)$.
But this contradicts (2), so this case holds vacuously.

Base Case: $\mathsf{len} = 2$
By assumption,
$T = G, \mathcal{P} \vdash K_1 \Longrightarrow K_2 \Longrightarrow K_C$
Denote
$\mathcal{D} :: G, \mathcal{P} \vdash K_1 \Longrightarrow K_2$
$\mathcal{E} :: G, \mathcal{P} \vdash K_2 \Longrightarrow K_C$
Want to show $\mathsf{lowProducer}(K_2)$, $K_1 \approx_L K_2$, and $(G, \mathcal{P} \vdash K_1 \Longrightarrow K_2) \downarrow_L = \cdot$

Case I: $\mathcal{D}$ ends in an input rule
By assumption, $\mathsf{consumer}(K_1)$. But this contradicts (2) and (3), so this case holds vacuously.

Case II: $\mathcal{D}$ ends in O, O-SKIP, or O-OTHER
By assumption and from the structure of the operational semantics,
(II.1) $\exists \mathcal{D}' :: G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa \xrightarrow{\alpha}_{pc} \sigma_2^G, ks$
(II.2) $ks_1 = (\mathcal{V}; \kappa; pc) :: \kappa_1'$
(II.3) $ks_2 = ks :: ks_1'$
From (II.2), (2), and (3),
(II.4) $pc = H$
(II.5) $ks_1' \not\approx_L \cdot$
By assumption and from (II.4) and the definitions of outCondition and output,
$(G, \mathcal{P} \vdash K_1 \Longrightarrow K_2) \downarrow_L = \cdot$

Subcase i: $\mathcal{D}$ ends in LC
By assumption and from (II.4),
(i.1) $\exists \mathcal{D}'' :: G, \mathcal{P}, \mathcal{V}, \sigma_1^G \vdash (\mathcal{V}; (\sigma, \mathsf{skip}, C, \cdot); H); \mathsf{lookupEHs}(E) \rightsquigarrow_H ks$
From (i.1), (II.3), and the rules for lookupEHs, which put $(\mathcal{V}; (\sigma, \mathsf{skip}, C, \cdot); H)$ at the top of the resulting ks,
(i.2) $ks_2 = (\mathcal{V}; (\sigma, \mathsf{skip}, C, \cdot); H) :: ks' :: ks_1'$
From (i.1), (3), and Lemma 66 (Requirement **(EH2)**),
(i.3) $ks_2 \approx_L ks_1$
From (i.2),
(i.4) $\mathsf{consumer}(ks_2)$
From (i.4),
(i.5) $\mathcal{E}$ must end in O-NEXT
From (i.5), (i.2), and (II.5),
(i.6) $ks_C = (ks' :: ks_1') \neq \cdot$
From (i.6),
(i.7) $\neg\mathsf{consumer}(K_C)$
But (i.7) contradicts (1), so this case holds vacuously

Subcase ii: $\mathcal{D}$ ends in PTOC
By a similar argument to **Subcase i**:
$ks_2 = (\mathcal{V}; (\sigma, \mathsf{skip}, C, \cdot); H) :: ks_1'$. Then, $\mathcal{E}$ must end in O-NEXT and $ks_1' \not\approx_L \cdot$ means $ks_C \neq \cdot$ which contradicts (1), so this case holds vacuously.

Subcase iii: $\mathcal{D}$ ends in P
By assumption, the resulting ks will be in H producer state. Then, $\mathcal{E}$ must end in O, O-SKIP, or O-OTHER, which contradicts (1) since the only rule to shrink the ks is O-NEXT.

Subcase iv: $\mathcal{D}$ ends in SME-H
If the resulting κ_H is in consumer state, the rest of the proof is similar to **Subcase i** or **Subcase ii**. Otherwise, the resulting κ_H is in producer state and the rest of the proof is similar to **Subcase iii**.

Subcase v: $\mathcal{D}$ ends in SME-L, SME-LTOH, or P-F
In all of these cases, $pc \sqsubseteq L$, which contradicts (II.4), so these cases hold vacuously.

Case III: $\mathcal{D}$ ends in O-NEXT

By assumption,

(III.1) $ks_1 = (\mathcal{V}; \kappa; pc) :: ks_2$

From (3) and (III.1),

(III.2) $pc = H$

By assumption and from (III.2),

$(G, \mathcal{P} \vdash K_1 \Longrightarrow K_2) \downarrow_L = \cdot$

By assumption and from (III.1) and (III.2),

$K_1 \approx_L K_2$

From (III.1), (III.2), and (2),

(III.3) $ks_2 \not\approx_L \cdot$

From (III.3),

(III.4) $\mathsf{producer}(K_2)$

From (III.4), either

$\mathsf{lowProducer}(K_2)$, in which case the desired conclusion holds.

Otherwise, $\mathsf{highProducer}(K_2)$, in which case the proof proceeds similarly to **Case II.iii**.

Inductive Case: $\mathsf{len}(T) = n + 1$ for $n \geq 2$

By assumption,

$T = G, \mathcal{P} \vdash K_1 \Longrightarrow K \Longrightarrow^* K_C$ with

$\mathsf{len}(K \Longrightarrow^* K_C) = n$

Denote

$\mathcal{D} :: K_1 \Longrightarrow K$

Case I: $\mathcal{D}$ ends in an input rule or O-NEXT

By assumption, $\mathsf{consumer}(K_1)$, but this contradicts (3), so this case holds vacuously

Case II: $\mathcal{D}$ ends in O, O-SKIP, or O-OTHER

By assumption and from the structure of the operational semantics,

(II.1) $\exists \mathcal{D}' :: G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa \xrightarrow{\alpha}_{pc} \sigma^G, ks'$

(II.2) $ks_1 = (\mathcal{V}; \kappa; pc) :: \kappa_1'$

(II.3) $ks = ks' :: ks_1'$

(II.4) $\sigma^G = \sigma_1^G$

From (II.2), (2), and (3),

(II.5) $pc = H$

(II.6) $ks_1' \not\approx_L \cdot$

By assumption and from (II.5) and the definitions of $\mathsf{outCondition}$ and output,

(II.7) $(G, \mathcal{P} \vdash K_1 \Longrightarrow K) \downarrow_L = \cdot$

From (II.2) - (II.4),

(II.8) $K_1 \approx_L K$

From (II.5) and by a similar argument as the subcases for **Base Case II**, above,

(II.9) $\mathsf{highProducer}(K)$

From (II.9), (II.6), (II.3), and Lemma 66 (Requirement **(EH2)**), the IH may be applied on $G, \mathcal{P} \vdash K \Longrightarrow^* K_C$

The conclusion follows from the IH and (II.7) and (II.8)

Case III: $\mathcal{D}$ ends in O-NEXT

By assumption and from the structure of the operational semantics,

(III.1) $ks_1 = (\mathcal{V}; \kappa; pc) :: \kappa_1'$

(III.2) $ks = ks_1'$

(III.3) $\sigma^G = \sigma_1^G$

From (III.1), (2), and (3),

(III.4) $pc = H$

(III.5) $ks_1' \not\approx_L \cdot$

By assumption and from (III.4),

(III.6) $(G, \mathcal{P} \vdash K_1 \Longrightarrow K) \downarrow_L = \cdot$

From (III.1) - (III.3),

(III.7) $K_1 \approx_L K$

From (III.2) and (III.5),

(III.8) $\mathsf{producer}(K)$

Subcase i: $\mathsf{lowProducer}(K)$

Then, let $K_2 = K$ and from (III.6) and (III.7), the desired conclusion holds

Subcase ii: $\mathsf{highProducer}(K)$

Then, the IH may be applied on $G, \mathcal{P} \vdash K \Longrightarrow^* K_C$ and the desired conclusion follows from the IH and (III.6) and (III.7)

□

Requirement (T4) Strong one-step

Lemma 17 (Strong One-step)**.** *If* $K_1 \approx_L K_2$, $T_1 = G, \mathcal{P} \vdash K_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1'$ *with* $T_1 \downarrow_L \neq \cdot$, $\neg\mathsf{rlsA}(T_1)$, *and* $\mathsf{prog}(K_2)$, *then* $\exists K_2', T_2$ *s.t.* $T_2 = G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_2'$ *with* $T_1 \approx_L T_2$ *and* $K_1' \approx_L K_2'$

Proof.

We examine each case of $\mathcal{E} :: T_1 = G, \mathcal{P} \vdash K_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1'$
Denote $K_i = (\mathcal{R}_i, d_i, \sigma_i, ks_i)$ for $i \in \{1, 2\}$
By assumption,
(1) $T_1 \downarrow_L \neq \cdot$
(2) $\neg\mathsf{rlsA}(T_1)$
(3) $K_1 \approx_L K_2$
(4) $\mathsf{prog}(K_2)$
From (3),
(5) $\sigma_1 \approx_L \sigma_2$
(6) $ks_1 \approx_L ks_2$

Case I: $\mathcal{E}$ ends in I-NR1 or I-NR2
In both of these cases $T_1 \downarrow_L = \cdot$.
This contradicts (1), so this case holds vacuously.

Case II: $\mathcal{E}$ ends in I-R-DIFF or I-R-SAME
In both of these cases, $\mathsf{rlsA}(T_1)$
This contradicts (2), so this case holds vacuously.

Case III: $\mathcal{E}$ ends in I-L
By assumption,
(III.1) $\mathcal{P}(\alpha_l) = L$
(III.2) $\sigma_1' = \sigma_1$
(III.3) $G, \mathcal{P}, \sigma_1 \vdash \cdot; \mathsf{lookupEH}(id.Ev(v)) \rightsquigarrow . \ ks_1'$

Subcase i: $\mathsf{consumer}(K_2)$
By assumption and from (III.1),
(i.1) I-L may be applied to K_2 with input $id.Ev(v)$, producing trace $T_2 = G, \mathcal{P} \vdash K_2 \overset{id.Ev(v)}{\Longrightarrow} K_2'$
From (i.1),
(i.2) $G, \mathcal{P}, \sigma_2 \vdash \cdot; \mathsf{lookupEHAll}(id.Ev(v)) \rightsquigarrow . \ ks_2'$
(i.3) $\sigma_2' = \sigma_2$
By assumption and from (III.1), (i.1), and projection for execution traces,
$T_1 \downarrow_L = T_2 \downarrow_L$
From (5), (III.2), and (i.3),
(i.4) $\sigma_1' \approx_L \sigma_2'$
From (III.3), (i.2), (5) and Lemma 56 (Requirement **(EH1)**),
(i.5) $ks_1' \approx_L ks_2'$
From (i.4) and (i.5),
$K_1' \approx_L K_2'$

Subcase ii: $\neg\mathsf{consumer}(K_2)$
From (4),
(ii.1) $\exists T'$ s.t. $T' = G, \mathcal{P} \vdash K_2 \overset{\tau}{\Longrightarrow}^* K_C$ where
(ii.2) $\mathsf{cstate}(K_C)$ and
(ii.3) $\forall \alpha \in \tau, \alpha \in \mathsf{output}(\tau)$
By assumption and from (6),
(ii.4) $ks_2 \approx_L \cdot$
From (ii.1)-(ii.4) and Lemma 15 (Requirement **(T3)**),
(ii.5) $K_C \approx_L K_2$ and
(ii.6) $T' \downarrow_L = \cdot$
The rest of the proof proceeds the same as **Subcase i**

Case IV: $\mathcal{E}$ ends in O
By assumption,
(IV.1) $ks_1 = (\mathcal{V}_1; \kappa_1; pc_1) :: ks_1''$
(IV.2) $\exists \mathcal{E}' :: G, \mathcal{P}, \mathcal{V}_1, d \vdash \sigma_1, \kappa_1 \overset{ch(v_1)}{\longrightarrow}_{pc_1} \sigma_1', ks_1'''$

(IV.3) $ks_1' = ks_1''' :: ks_1''$
(IV.4) $\mathsf{producer}(\kappa_1)$
(IV.5) $\mathsf{outCondition}_{\mathcal{V}_1}(\mathcal{P}, ch(v_1), pc_1) = \mathsf{true}$
(IV.6) $\alpha_{l,1} = \mathsf{output}(\mathcal{P}, ch(v_1), pc_1)$
From (IV.6) and the definition of output: if $pc_1 = H$, then $T_1 \downarrow_L = \cdot$, which contradicts (1). Therefore,
(IV.7) $pc_1 \sqsubseteq L$
By assumption and from (IV.7) and (6),
(IV.8) $ks_2 = (\mathcal{V}_2; \kappa_2; pc_2) :: \kappa_2''$

Subcase i: $pc_2 \sqsubseteq L$
By assumption and from (IV.1), (IV.8), (IV.7), and (6),
(i.1) $\mathcal{V}_1 = \mathcal{V}_2$
(i.2) $\kappa_1 \approx_L \kappa_2$
(i.3) $ks_1'' \approx_L ks_2''$
By assumption and from (i.1), (IV.2), (IV.7), (2), (5), (i.2), and Lemma 18,
(i.4) $\exists \mathcal{D}' :: G, \mathcal{P}, \mathcal{V}_2, d \vdash \sigma_2, \kappa_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_2', ks_2'''$ with
(i.5) $pc_2 \sqsubseteq L$
(i.6) $\sigma_1' \approx_L \sigma_2'$
(i.7) $ks_1''' \approx_L ks_2'''$
(i.8) $\alpha_2 \approx_L ch(v_1)$
From (i.7),
(i.9) $\alpha = ch(v_2)$ with
(i.10) $v_1 \approx_L v_2$
By assumption and from (IV.7), (i.10), (IV.5), (i.1), and Lemma 20,
(i.11) $\mathsf{outCondition}_{\mathcal{V}_2}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{true}$
By assumption and from (i.3) and (i.11),
(i.12) O may be applied to K_2, producing trace $T_2 = G, \mathcal{P} \vdash K_2 \overset{\alpha_{2,l}}{\Longrightarrow} K_2'$
From (i.12), (IV.8), and (i.4),
(i.13) $ks_2' = ks_2''' :: ks_2''$
From (IV.3), (i.13), (i.3), and (i.7),
(i.14) $ks_1' \approx_L ks_2'$
From (i.6), and (i.14),
$K_1' \approx_L K_2'$
From (i.12),
(i.15) $\alpha_{l,2} = \mathsf{output}(\mathcal{P}, ch(v_2), pc_2)$
By assumption and from (i.10), (IV.7), (IV.6), (i.15), and Lemma 22,
(i.16) $\alpha_{l,1} \approx_L \alpha_{l,2}$
From (i.16) and the definition of equivalence for execution traces,
$T_1 \approx_L T_2$

Subcase ii: $pc_2 \not\sqsubseteq L$
From (4), $\exists T', K_C$ s.t.
(ii.1) $T' = G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_C$
(ii.2) $\mathsf{consumer}(K_C)$
By assumption and from (IV.7), (ii.1), (ii.2), and Lemma 16 (Requirement **(T3)**), $\exists K'$ s.t.
(ii.3) $K_2 \Longrightarrow^* K'$ with
(ii.4) $\mathsf{lowProducer}(K')$
(ii.5) $K' \approx_L K_2$ and
(ii.6) $(G, \mathcal{P} \vdash K_2 \Longrightarrow^* K') \downarrow_L = \cdot$
The rest of the proof proceeds the same as **Subcase i**. Trace equivalence uses (ii.6) and state equivalence uses (ii.5).

Case V: $\mathcal{E}$ ends in O-SKIP
By assumption,
(V.1) $ks_1 = (\mathcal{V}_1; \kappa_1; pc_1) :: ks_1''$
(V.2) $\exists \mathcal{E}' :: G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1, \kappa_1 \xrightarrow{ch(v_1)}_{pc_1} \sigma_1', ks_1'''$
(V.3) $\mathsf{producer}(\kappa_1)$
(V.4) $\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v_1), pc_1) = \mathsf{false}$
(V.5) $\alpha_{l,1} = (\bullet, pc_1)$
If $pc_1 = H$, then $T_1 \downarrow_L = \cdot$, which contradicts (1). Therefore,
(V.6) $pc_1 \sqsubseteq L$

By assumption and from (V.6) and (6),
(V.7) $ks_2 = (\mathcal{V}_2; \kappa_2; pc_2) :: \kappa_2''$

Subcase i: $pc_2 \sqsubseteq L$
By assumption and from (V.1), (V.6), (V.7), and (6),
(i.1) $\mathcal{V}_1 = \mathcal{V}_2$
(i.2) $\kappa_1 \approx_L \kappa_2$
(i.3) $ks_1'' \approx_L ks_2''$
By assumption and from (V.2), (V.6), (2), (5), (i.1), (i.2) and Lemma 18,
(i.4) $\exists \mathcal{D}' :: G, \mathcal{P}, \mathcal{V}_2, d \vdash \sigma_2, \kappa_2 \xrightarrow{\alpha}_{pc_2} \sigma_2', ks_2'''$ with
(i.5) $pc_2 \sqsubseteq L$
(i.6) $\sigma_1' \approx_L \sigma_2'$
(i.7) $ks_1' \approx_L ks_2'$
(i.8) $\alpha \approx_L ch(v_1)$
From (i.8),
(i.9) $\alpha = ch(v_2)$ with
(i.10) $v_1 \approx_L v_2$
From (i.10), (V.4), (V.6), (i.5), and Lemma 21,
(i.11) $\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{false}$ or
(i.12) $\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{true}$ and $v_2 = \langle \_ | \_ \rangle$

Subsubcase a: (i.11) is true
By assumption and from (i.4) and (i.11),
(a.1) O-SKIP may be applied to K_2, producing trace $T_2 = G, \mathcal{P} \vdash K_2 \overset{(\bullet, pc_2)}{\Longrightarrow} K_2'$
From (i.6), (i.7), and (a.1),
$K_1' \approx_L K_2'$
From (a.1),
(a.2) $\alpha_{l,2} = (\bullet, pc_2)$
From (V.5), (V.6), (i.5), (a.2), and the definition of equivalence for execution traces,
$T_1 \approx_L T_2$

Subsubcase b: (i.12) is true
By assumption and from (i.4) and (i.12),
(b.1) O may be applied to K_2, producing trace $T_2 = G, \mathcal{P} \vdash K_2 \overset{\alpha_{l,2}}{\Longrightarrow} K_2'$
From (i.6), (i.7), and (b.1),
$K_1' \approx_L K_2'$
From (V.4), (V.6), and the definition of output, either:
(b.2) $pc_1 = L$ and $\mathcal{P}(ch) = H$ or
(b.3) $pc_1 = \cdot$ and $v_1 \downarrow_{\mathcal{P}(ch)} = \cdot$
By assumption and from the structure of the operational semantics,
(b.4) $pc_2 = \cdot$ and $\mathcal{V}_2 = \mathsf{MF}$
If (b.2) is true, then from (b.4), (i.12), and the definition of output,
(b.5) $\mathsf{output}(\mathcal{P}, ch(v_2), pc_2) = \langle ch(\mathsf{getFacet}(v_2, H)) | \bullet \rangle$
If (b.3) is true, then from (b.4) and (i.12),
(b.6) $\mathcal{P}(ch) = H$ because otherwise $\mathsf{outCondition}$ would have been false
From (b.6) and by the same argument as (b.5),
(b.7) $\mathsf{output}(\mathcal{P}, ch(v_2), pc_2) = \langle ch(\mathsf{getFacet}(v_2, H)) | \bullet \rangle$
From (V.5), (V.6), (b.5), (b.7), and the definition of equivalence for execution traces,
$T_1 \approx_L T_2$

Subcase ii: $pc_2 \not\sqsubseteq L$
The proof for this case uses similar logic as **Subcase IV.b** to reach the assumptions for **Subcase i**, at which point the proof proceeds the same as **Subcase i**.

Case VI: $\mathcal{E}$ ends in O-OTHER
The proof for this case is similar to **Case V**. $\alpha_2 = \bullet$ follows from Lemma 18, which tells us that $\alpha_1 \approx_L \alpha_2$ and $\alpha_1 = \bullet$ by assumption.

Case VII: $\mathcal{E}$ ends in O-NEXT
The proof for this case is straightforward. From (6), $ks_1 = (\mathcal{V}_1; \kappa_1; pc_1) :: ks_1''$ and $ks_2 = (\mathcal{V}_2; \kappa_2; pc_2) :: ks_2''$ with $\mathcal{V}_1 = \mathcal{V}_2$ and $\kappa_1 \approx_L \kappa_2$ and $ks_1'' \approx_L ks_2''$ when $pc_2 \sqsubseteq L$. Then, from $\kappa_1 \approx_L \kappa_2$, $\mathsf{consumer}(\kappa_2)$, and O-NEXT can be

applied to K_2, which gives $T_1 \approx_L T_2$. $K_1' \approx_L K_2'$ follows from $ks_1'' \approx_L ks_2''$.
When $pc_2 \not\sqsubseteq L$, the proof is similar to **Case IV.b**.

□

Lemma 18 (Strong One-Step - Single Execution Semantics)**.** *If* $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa_1 \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, ks_1$ *with* $pc_1 \sqsubseteq L$, $\alpha_1 \neq \mathsf{declassify}(\iota, v)$, $\sigma_1^G \approx_L \sigma_2^G$, $\kappa_1 \approx_L \kappa_2$, *and* $pc_2 \sqsubseteq L$, *then* $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_2^G, \kappa_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, ks_2$ *with* $\sigma_3^G \approx_L \sigma_4^G$, $ks_1 \approx_L ks_2$, *and* $\alpha_1 \approx_L \alpha_2$

Proof.
By induction on the structure of $\mathcal{E} :: G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa_1 \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, ks_1$
By assumption:
(1) $pc_1 \sqsubseteq L$
(2) $\alpha_1 \neq \mathsf{declassify}(\iota, v)$
(3) $\sigma_1^G \approx_L \sigma_2^G$
(4) $\kappa_1 \approx_L \kappa_2$
(5) $pc_2 \sqsubseteq L$

Case I: $\mathcal{E}$ ends in LC
By assumption,
(I.1) $\kappa_1 = \sigma_1, \mathsf{skip}, P, E_1$
(I.2) $E_1 \neq \cdot$
(I.3) $\sigma_3^G = \sigma_1^G$
(I.4) $G, \mathcal{P}, \mathcal{V}, \sigma_1^G \vdash (\mathcal{V}; (\sigma_1, \mathsf{skip}, C, \cdot); pc_1); \mathsf{lookupEHs}(E_1) \rightsquigarrow_{pc_1} ks_1$
(I.5) $\alpha_1 = \bullet$
From (4) and (I.1),
(I.6) $\kappa_2 = \sigma_2, \mathsf{skip}, P, E_2$ with
(I.7) $\sigma_1 \approx_L \sigma_2$ and
(I.8) $E_1 \approx_L E_2$

Subcase i: $E_2 = \cdot$
By assumption and from (4),
(i.1) $E_1 \approx_L \cdot$
From (i.1), (I.4), and Lemma 66 (Requirement **(EH2)**),
(i.2) $ks_1 \approx_L (\mathcal{V}; (\sigma_1, \mathsf{skip}, C, \cdot); pc_1)$
By assumption and from (I.6),
(i.3) PTOC may be applied, resulting in trace $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_2^G, \kappa_2 \xrightarrow{\bullet}_{pc_2} \sigma_2^G, (\mathcal{V}; (\sigma_2, \mathsf{skip}, C, \cdot); pc_2)$
From (i.3),
(i.4) $\sigma_4^G = \sigma_2^G$
(i.5) $\alpha_2 = \bullet$
(i.6) $ks_2 = (\mathcal{V}; (\sigma_2, \mathsf{skip}, C, \cdot); pc_2)$
From (3), (I.3), and (i.4),
$\sigma_3^G \approx_L \sigma_4^G$
From (I.5) and (i.5),
$\alpha_1 \approx_L \alpha_2$
From (1), (5), (i.2), (i.6), and (I.7),
$ks_1 \approx_L ks_2$

Subcase ii: $E_2 \neq \cdot$
By assumption,
(ii.1) LC may be applied, resulting in trace $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_2^G, \kappa_2 \xrightarrow{\bullet}_{pc_2} \sigma_2^G, ks_2$
From (ii.1),
(ii.2) $\sigma_4^G = \sigma_2^G$
(ii.3) $\alpha_2 = \bullet$
(ii.4) $G, \mathcal{P}, \mathcal{V}, \sigma_2^G \vdash (\mathcal{V}; (\sigma_2, \mathsf{skip}, C, \cdot); pc_2); \mathsf{lookupEHs}(E_2) \rightsquigarrow_{pc_2} ks_2$
From (3), (I.3), and (ii.2),
$\sigma_3^G \approx_L \sigma_4^G$
From (I.5) and (ii.3),
$\alpha_1 \approx_L \alpha_2$
From (1), (5), and (I.7),
(ii.5) $(\mathcal{V}; (\sigma_1, \mathsf{skip}, C, \cdot); pc_1) \approx_L (\mathcal{V}; (\sigma_2, \mathsf{skip}, C, \cdot); pc_2)$
From (1), (5), (3), (ii.5), (I.8), (I.4), (ii.4), and Lemma 56 (Requirement **(EH1)**),
$ks_1 \approx_L ks_2$

Case II: $\mathcal{E}$ ends in PTOC
The proofs for these cases are similar to **Case I**

Case III: $\mathcal{E}$ ends in P
By assumption,
(III.1) $\kappa_1 = \sigma_1, c, P, E_1$
(III.2) $G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1', E_1'$
(III.3) $ks_1 = (\mathcal{V}; (\sigma_1', c_1', P, (E_1, E_1')); pc_1)$
From (4),$\kappa_2 = \sigma_2, c, P, E_2$ with
(III.4) $\sigma_1 \approx_L \sigma_2$
(III.5) $E_1 \approx_L E_2$
From (1)-(5), (III.4), (III.2), and Lemma 19,
(III.6) $G, \mathcal{V}, d \Vdash \sigma_2^G, \sigma_2, c \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, \sigma_2', c_2', E_2'$ with
$\sigma_3^G \approx_L \sigma_4^G$
$\alpha_1 \approx_L \alpha_2$
(III.7) $\sigma_1' \approx_L \sigma_2'$
(III.8) $c_1' \approx_L c_2'$
(III.9) $E_1' \approx_L E_2'$

Subcase i: $c_2' \neq \langle \_ | \_ \rangle$
By assumption and from (III.6),
(i.1) LC may be applied , resulting in trace $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_2^G, \kappa_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, ks_2$
From (III.6) and (i.1),
(i.2) $ks_2 = (\mathcal{V}; (\sigma_2', c_2', P, (E_2, E_2')); pc_2)$
From (1), (5), (III.3), (i.2), and (III.7)-(III.9),
$ks_1 \approx_L ks_2$

Subcase ii: $c_2' = \langle c_H | c_L \rangle$
By assumption and from (III.6),
(ii.1) P-F may be applied , resulting in trace $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_2^G, \kappa_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, ks_2$
From (III.6) and (ii.1),
(ii.2) $ks_2 = (\mathcal{V}; (\sigma_2', c_L, P, (E_2, E_2')); L) :: (\mathcal{V}; \sigma_2', c_H, P, (E_2, E_2'); H)$
From (1), (III.3), (ii.2), and (III.7)-(III.9),
$ks_1 \approx_L ks_2$

Case IV: $\mathcal{E}$ ends in SME-L or SME-LTOH
The conclusion follows from the IH

Case V: $\mathcal{E}$ ends in SME-H
This contradicts (1), so this case holds vacuously.

Case VI: $\mathcal{E}$ ends in P-L
The proof for this case is similar to **Case III**

□

Lemma 19 (Strong One-Step - Command Semantics)**.** *If* $pc_1, pc_2 \sqsubseteq L$ *and* $G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1, E_1$*, with* $\alpha_1 \neq \mathsf{declassify}(\iota, v)$*, and* $\sigma_1^G \approx_L \sigma_2^G$*, and* $\sigma_1 \approx_L \sigma_2$*, then* $G, \mathcal{V}, d \Vdash \sigma_2^G, \sigma_2, c \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, \sigma_2', c_2, E_2$ *with* $\sigma_3^G \approx_L \sigma_4^G$*,* $\sigma_1' \approx_L \sigma_2'$*,* $\alpha_1 \approx_L \alpha_2$*,* $c_1 \approx_L c_2$*, and* $E_1 \approx_L E_2$*.*

Proof.
By induction on the structure of $\mathcal{E} :: G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c \xrightarrow{\alpha}_{pc_1} \sigma_3^G, \sigma_2, c_1, E_1$

By assumption,
(1) $\alpha_1 \neq \mathsf{declassify}(\iota, v)$
(2) $\sigma_1^G \approx_L \sigma_2^G$
(3) $\sigma_1 \approx_L \sigma_2$
(4) $pc_1, pc_2 \sqsubseteq L$

Case I: $\mathcal{E}$ ends in SKIP
This case is straightforward.

Case II: $\mathcal{E}$ ends in SEQ or SEQ-F
The conclusion follows from the IH. If the result is faceted, SEQ-F is applied. Otherwise, SEQ is applied.

Case III: $\mathcal{E}$ ends in DECLASSIFY-L or DECLASSIFY-NC
By assumption, $\alpha_1 = \mathsf{declassify}(\iota, v)$, which contradicts (3), so this case holds vacuously.

Case IV: $\mathcal{E}$ ends in DECLASSIFY-H
By assumption, $pc_1 = H$, which contradicts (2), so this case holds vacuously.

Unstructured EH storage:
Case U.I: $\mathcal{E}$ ends in ASSIGN-L
By assumption,
(U.I.1) $x \notin \sigma_1^G$
(U.I.2) $\sigma_3^G = \sigma_1^G$
(U.I.3) $c = x := e$
(U.I.4) $c_1 = \mathsf{skip}$
(U.I.5) $E_1 = \cdot$
(U.I.6) $\alpha_1 = \bullet$
(U.I.7) $G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e \Downarrow^{\mathcal{V}}_{pc_1} v_1$
(U.I.8) $\sigma_1' = \mathsf{assign}_{\mathcal{V}}(\sigma_1, pc_1, x, v_1)$
From (3), (4), (U.I.7), and Lemma 36.U (Requirement **(E1)**),
(U.I.9) $G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e \Downarrow^{\mathcal{V}}_{pc_2} v_2$ with
(U.I.10) $v_1 \simeq_L v_2$
From U.I,
(U.I.11) $x \notin \sigma_2^G$
Applying ASSIGN-L on (U.I.7) and (U.I.10) produces trace
(U.I.12) $G, \mathcal{V}, d \Vdash \sigma_2^G, \sigma_2, x := e \xrightarrow{\bullet}_{pc_2} \sigma_4^G, \sigma_2', \mathsf{skip}, \cdot$ where
(U.I.13) $\sigma_2' = \mathsf{assign}_{\mathcal{V}}(\sigma_2, pc_2, x, v_2)$
From (2), (4), (U.I.8), (U.I.13), and Lemma 44.U (Requirement **(V3)**),
$\sigma_1' \approx_L \sigma_2'$
From (U.I.12),
(U.I.14) $\sigma_4^G = \sigma_2^G$
(U.I.15) $\alpha_2 = \bullet$
(U.I.16) $c_2 = \mathsf{skip}$
(U.I.17) $E_2 = \cdot$
From (2), (U.I.2), and (U.I.14),
$\sigma_3^G \approx_L \sigma_4^G$
From (U.I.6) and (U.I.15),
$\alpha_1 \approx_L \alpha_2$
From (U.I.4) and (U.I.16),
$c_1 \approx_L c_2$
From (U.I.5) and (U.I.17),
$E_1 \approx_L E_2$

Case U.II: $\mathcal{E}$ ends in ASSIGN-G
The proof is similar to **Case U.I**

Case U.III: $\mathcal{E}$ ends in IF-TRUE
By assumption,
(U.III.1) $c = \mathsf{if}\ e\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2'$
(U.III.2) $c_1 = c_1'$
(U.III.3) $E_1 = \cdot$
(U.III.4) $\sigma_3^G = \sigma_1^G$
(U.III.5) $\sigma_1' = \sigma_1$
(U.III.6) $\alpha_1 = \bullet$
(U.III.7) $G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e \Downarrow^{\mathcal{V}}_{pc_1} v_1$ with
(U.III.8) $\mathsf{valOf}(v_1) = \mathsf{true}$
From (2)-(4), (U.III.7), and Lemma 36.U (Requirement **(E1)**),
(U.III.9) $G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e \Downarrow_{pc_2} v_2$ with
(U.III.10) $v_2 \simeq_L \mathsf{true}$
From (U.III.8) and (U.III.10),
(U.III.11) $\mathsf{valOf}(v_2) = \mathsf{true}$ or

(U.III.12) $v_2 = \langle \_ | \mathsf{true} \rangle$

Subcase i: (U.III.11) is true
Applying IF-TRUE produces trace
(i.1) $G, \mathcal{V}, d \Vdash \sigma_2^G, \sigma_2, x := e \xrightarrow{\bullet}_{pc_2} \sigma_2^G, \sigma_2, c_1', \cdot$
From (i.1),
(i.2) $\sigma_4^G = \sigma_2^G$
(i.3) $\sigma_2' = \sigma_2$
(i.4) $c_2 = c_1'$
(i.5) $E_2 = \cdot$
(i.6) $\alpha_2 = \bullet$
From (2), (U.III.4), and (i.2),
$\sigma_3^G \approx_L \sigma_4^G$
From (3), (U.III.5), and (i.3),
$\sigma_1' \approx_L \sigma_2'$
From (U.III.6) and (i.6),
$\alpha_1 \approx_L \alpha_2$
From (U.III.2) and (i.4),
$c_1 \approx_L c_2$
From (U.III.3) and (i.5),
$E_1 \approx_L E_2$

Subcase ii: (U.III.12) is true
From (U.III.12) and since facets are only produced when the $pc = \cdot$, it must be the case that $pc_2 = \cdot$
Applying IF-F produces trace $G, \mathcal{V}, d \Vdash \sigma_2^G, \sigma_2, x := e \xrightarrow{\bullet}_{pc_2} \sigma_2^G, \sigma_2, \langle c_H | c_1' \rangle, \cdot$
The rest of the proof proceeds similarly to **Subcase i**.

Case U.IV: $\mathcal{E}$ ends in IF-FALSE, WHILE-TRUE, WHILE-FALSE, IF-F, WHILE-F, IF-F, or WHILE-F
The proof for this case is similar to **Case U.III**

Case U.V: $\mathcal{E}$ ends in OUTPUT
The proof for this case is similar to previous cases. It uses Lemma 36.U (Requirement **(E1)**) to show that the outputs are (strong) equivalent.

Case U.VI: $\mathcal{E}$ ends in ASSIGN-D
The proofs are similar to **Case U.I** except that Lemma 80.U (Requirement **(EH4)**) is used.

Case U.VII: $\mathcal{E}$ ends in CREATEELEM
The proof for this case is similar to previous cases. It uses Lemma 36.U (Requirement **(E1)**) to show that the arguments are (strong) equivalent and Lemma 81.U (Requirement **(EH4)**) to show that $\sigma_3^G \approx_L \sigma_4^G$

Case U.VIII: $\mathcal{E}$ ends in ADDEH
The proof for this case is similar to previous cases. It uses Lemma 81.U (Requirement **(EH4)**) to show that $\sigma_3^G \approx_L \sigma_4^G$

Case U.IX: $\mathcal{E}$ ends in TRIGGER
The proof for this case is similar to previous cases. It uses Lemma 36.U (Requirement **(E1)**) to show that the arguments are (strong) equivalent and Lemma 63 (Requirement **(EH1)**) to show that $E_1 \approx_L E_2$

Tree-structured EH storage:
Case T.I: $\mathcal{E}$ ends in ASSIGN-L
The proof for this case is similar to **Case U.I** except that Lemma 36.T is used to show that the values being assigned are equivalent and Lemma 44.T is used to show that the stores are equivalent after assignment.

Case T.II: $\mathcal{E}$ ends in ASSIGN-G
The proof is similar to **Case T.1**

Case T.III: $\mathcal{E}$ ends in IF-TRUE
The proof for this case is similar to **Case U.III** except that Lemma 36.T is used to show that the branch is equivalent.

Case T.IV: $\mathcal{E}$ ends in IF-FALSE, WHILE-TRUE, WHILE-FALSE, IF-F, WHILE-F, IF-F, or WHILE-F

The proof for this case is similar to **Case T.III**

Case T.V: $\mathcal{E}$ ends in OUTPUT
The proof for this case is similar to previous cases. It uses Lemma 36.T (Requirement **(E1)**) to show that the outputs are (strong) equivalent.

Case T.VI: $\mathcal{E}$ ends in ASSIGN-D
The proofs are similar to **Case T.I** except that Lemma 80.T (Requirement **(EH4)**) is used for ASSIGN-D.

Case T.VII: $\mathcal{E}$ ends in CREATECHILD or CREATESIBLING
The proofs for these cases are similar to previous cases. They use Lemma 36 (Requirement **(E1)**) to show that the arguments are (strong) equivalent and Lemma 81.T (Requirement **(EH4)**) to show that $\sigma_3^G \approx_L \sigma_4^G$

Case T.VIII: $\mathcal{E}$ ends in ADDEH
The proof for this case is similar to previous cases. It uses Lemma 81.T (Requirement **(EH4)**) to show that $\sigma_3^G \approx_L \sigma_4^G$

Case T.IX: $\mathcal{E}$ ends in TRIGGER
The proof for this case is similar to previous cases. It uses Lemma 36 (Requirement **(E1)**) to show that the arguments are (strong) equivalent and Lemma 63 (Requirement **(EH1)**) to show that $E_1 \approx_L E_2$

□

Lemma 20. *If* $v_1 \approx_L v_2$ *and* $pc_1, pc_2 \sqsubseteq L$, *with* $\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v_1), pc_1) = \mathsf{true}$, *then* $\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{true}$

Lemma 21. *If* $v_1 \approx_L v_2$ *and* $pc_1, pc_2 \sqsubseteq L$, *with* $\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v_1), pc_1) = \mathsf{false}$, *then either:*
1) $\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{false}$, *or*
2) $\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{true}$ *and* $v_2 = \langle \_ | \_ \rangle$

Lemma 22. *If* $v_1 \approx_L v_2$ *and* $pc_1, pc_2 \sqsubseteq L$ *with* $\alpha_{l,1} = \mathsf{output}(\mathcal{P}, ch(v_1), pc_1)$ *and* $\alpha_{l,2} = \mathsf{output}(\mathcal{P}, ch(v_2), pc_2)$, *then* $\alpha_{l,1} \approx_L \alpha_{l,2}$

Requirement (WT4) Strong one-step (Weak Secrecy)

Lemma 23 (Strong One-step, Weak Secrecy)**.** *If* $K_1 \approx_L K_2$ *and* $T_1 = G, \mathcal{P} \vdash K_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1'$ *with* $T_1 \downarrow_L = \alpha'$, $\neg\mathsf{rlsA}(T_1)$, *and* $\mathsf{prog}(K_2)$, $\mathsf{wkT}(K_2, \alpha')$, *then* $\exists K_2', T_2, \alpha_{l,2}$ *s.t.* $T_2 = G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_2'$ *with* $T_1 \approx_L T_2$ *and* $K_1' \approx_L K_2'$

Proof.
We examine each case of $\mathcal{E} :: T_1' = G, \mathcal{P} \vdash K_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1'$
By assumption,
(1) $K_1 \approx_L K_2$
(2) $T_1 \downarrow_L = \alpha'$
(3) $\neg\mathsf{rlsA}(T_1)$
(4) $\mathsf{prog}(K_2)$
(5) $\mathsf{wkT}(K_2, \alpha')$
From (1),
(6) $\sigma_1 \approx_L \sigma_2$
(7) $ks_1 \approx_L ks_2$

Case I: $\mathcal{E}$ ends in I-NR1 or I-NR2
In both of these cases $T_1 \downarrow_L = \cdot$.
This contradicts (2), so this case holds vacuously.

Case II: $\mathcal{E}$ ends in I-R-DIFF or I-R-SAME
In both of these cases, $\mathsf{rlsA}(T_1)$
This contradicts (2), so this case holds vacuously.

Case III: $\mathcal{E}$ ends in I-L
By assumption,
(III.1) $\mathcal{P}(\alpha_l) = L$
(III.2) $\sigma_1' = \sigma_1$
(III.3) $G, \mathcal{P}, \sigma_1 \vdash \cdot; \mathsf{lookupEH}(id.Ev(v)) \rightsquigarrow \cdot\ ks_1'$

Subcase i: $\mathsf{consumer}(K_2)$
By assumption and from (III.1),

(i.1) I-L may be applied to K_2 with input $id.Ev(v)$, producing trace $T_2 = G, \mathcal{P} \vdash K_2 \overset{id.Ev(v)}{\Longrightarrow} K_2'$
From (i.1),
(i.2) $G, \mathcal{P}, \sigma_2 \vdash \cdot; \mathsf{lookupEHAll}(id.Ev(v)) \rightsquigarrow \cdot\ ks_2'$
(i.3) $\sigma_2' = \sigma_2$
By assumption and from (III.1), (i.1), and projection for execution traces,
$T_1 \downarrow_L = T_2 \downarrow_L$
From (6), (III.2), and (i.3),
(i.4) $\sigma_1' \approx_L \sigma_2'$
From (III.3), (i.2), (6) and Lemma 56 (Requirement **(EH1)**),
(i.5) $ks_1' \approx_L ks_2'$
From (i.4) and (i.5),
$K_1' \approx_L K_2'$

Subcase ii: $\neg\mathsf{consumer}(K_2)$
From (5) and from $\alpha_{l,1} = id.Ev(v)$,
(ii.1) $\exists T', K_{lp}$ s.t. $T' = G, \mathcal{P} \vdash K_2 \overset{\tau}{\Longrightarrow}{}^* K_C$ where
(ii.2) $\mathsf{cstate}(K_C)$ and
(ii.3) $T' \downarrow_L = \cdot$
From (ii.1), (ii.3) and Lemma 12 (Requirement **(WT2)**),
(ii.4) $K_2 \approx_L K_C$
The rest of the proof proceeds the same as **Subcase i**.

Case IV: $\mathcal{E}$ ends in O
By assumption,
(IV.1) $ks_1 = (\mathcal{V}_1; \kappa_1; pc_1) :: ks_1''$
(IV.2) $\exists \mathcal{E}' :: G, \mathcal{P}, \mathcal{V}_1, d \vdash \sigma_1, \kappa_1 \overset{ch(v_1)}{\longrightarrow}_{pc_1} \sigma_1', ks_1'''$
(IV.3) $ks_1' = ks_1''' :: ks_1''$
(IV.4) $\mathsf{producer}(\kappa_1)$
(IV.5) $\mathsf{outCondition}_{\mathcal{V}_1}(\mathcal{P}, ch(v_1), pc_1) = \mathsf{true}$
(IV.6) $\alpha_{l,1} = \mathsf{output}(\mathcal{P}, ch(v_1), pc_1)$
From (IV.6) and the definition of output: if $pc_1 = H$, then $T_1 \downarrow_L = \cdot$, which contradicts (2). Therefore,
(IV.7) $pc_1 \sqsubseteq L$
By assumption and from (IV.7) and (7),
(IV.8) $ks_2 = (\mathcal{V}_2; \kappa_2; pc_2) :: ks_2''$

Subcase i: $pc_2 \sqsubseteq L$
By assumption and from (IV.1), (IV.8), (IV.7), and (7),
(i.1) $\mathcal{V}_1 = \mathcal{V}_2$
(i.2) $\kappa_1 \approx_L \kappa_2$
(i.3) $ks_1'' \approx_L ks_2''$
By assumption and from (5),
(i.4) $\mathsf{wkK}(\kappa_2, \alpha_1)$
By assumption and from (i.1), (IV.2), (IV.7), (3), (i.4), (6), (i.2), and Lemma 25,
(i.5) $\exists \mathcal{D}' :: G, \mathcal{P}, \mathcal{V}_2, d \vdash \sigma_2, \kappa_2 \overset{\alpha_2}{\longrightarrow}_{pc_2} \sigma_2', ks_2'''$ with
(i.6) $pc_2 \sqsubseteq L$
(i.7) $\sigma_1' \approx_L \sigma_2'$
(i.8) $ks_1''' \approx_L ks_2'''$
(i.9) $\alpha_2 \approx_L ch(v_1)$
From (i.8),
(i.10) $\alpha = ch(v_2)$ with
(i.11) $v_1 \approx_L v_2$
By assumption and from (IV.7), (i.11), (IV.5), (i.1), and Lemma 20 (Requirement **(T4)**),
(i.12) $\mathsf{outCondition}_{\mathcal{V}_2}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{true}$
By assumption and from (i.3) and (i.11),
(i.13) O may be applied to K_2, producing trace $T_2 = G, \mathcal{P} \vdash K_2 \overset{\alpha_{2,l}}{\Longrightarrow} K_2'$
From (i.14), (IV.8), and (i.5),
(i.14) $ks_2' = ks_2''' :: ks_2''$
From (IV.3), (i.14), (i.3), and (i.8),
(i.15) $ks_1' \approx_L ks_2'$
From (i.6), and (i.15),
$K_1' \approx_L K_2'$

From (i.13),
(i.16) $\alpha_{l,2} = \mathsf{output}(\mathcal{P}, ch(v_2), pc_2)$
By assumption and from (i.11), (IV.7), (IV.6), (i.16), and Lemma 22 (Requirement **(T4)**),
(i.17) $\alpha_{l,1} \approx_L \alpha_{l,2}$
From (i.17) and the definition of equivalence for execution traces,
$T_1 \approx_L T_2$

Subcase ii: $pc_2 \not\sqsubseteq L$
From (5) and (IV.6), $\exists T', K_{lp}$ s.t.
(ii.1) $T' = G, \mathcal{P} \vdash K_2 \Longrightarrow^* K_{lp}$
(ii.2) $\mathsf{lowProducer}(K_{lp})$ and
(ii.3) $T' \downarrow_L = \cdot$
From (ii.1), (ii.3) and Lemma 12 (Requirement **(WT2)**),
(ii.4) $K_2 \approx_L K_{lp}$
The rest of the proof proceeds the same as **Subcase i**. Trace equivalence uses (ii.3) and state equivalence uses (ii.4).

Case V: $\mathcal{E}$ ends in O-SKIP
By assumption,
(V.1) $ks_1 = (\mathcal{V}_1; \kappa_1; pc_1) :: ks_1''$
(V.2) $\exists \mathcal{E}' :: G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1, \kappa_1 \xrightarrow{ch(v_1)}_{pc_1} \sigma_1', ks_1'''$
(V.3) $\mathsf{producer}(\kappa)$
(V.4) $\mathsf{outCondition}_{\mathcal{I}}(\mathcal{P}, ch(v_1), pc_1) = \mathsf{false}$
(V.5) $\alpha_{l,1} = (\bullet, pc_1)$
If $pc_1 = H$, then $T_1 \downarrow_L = \cdot$, which contradicts (2). Therefore,
(V.6) $pc_1 \sqsubseteq L$
By assumption and from (V.6) and (7),
(V.7) $ks_2 = (\mathcal{V}_2; \kappa_2; pc_2) :: \kappa_2''$

Subcase i: $pc_2 \sqsubseteq L$
By assumption and from (V.1), (V.6), (V.7), and (7),
(i.1) $\mathcal{V}_1 = \mathcal{V}_2$
(i.2) $\kappa_1 \approx_L \kappa_2$
(i.3) $ks_1'' \approx_L ks_2''$
By assumption and from (V.2), (V.6), (3), (6), (i.1), (i.2) and Lemma 18,
(i.4) $\exists \mathcal{D}' :: G, \mathcal{P}, \mathcal{V}_2, d \vdash \sigma_2, \kappa_2 \xrightarrow{\alpha}_{pc_2} \sigma_2', ks_2'''$ with
(i.5) $pc_2 \sqsubseteq L$
(i.6) $\sigma_1' \approx_L \sigma_2'$
(i.7) $ks_1' \approx_L ks_2'$
(i.8) $\alpha \approx_L ch(v_1)$
From (i.8),
(i.9) $\alpha = ch(v_2)$ with
(i.10) $v_1 \approx_L v_2$
From (i.10), (V.4), (V.6), (i.5), and Lemma 21 (Requirement **(T4)**),
(i.11) $\mathsf{outCondition}_{\mathcal{I}}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{false}$ or
(i.12) $\mathsf{outCondition}_{\mathcal{I}}(\mathcal{P}, ch(v_2), pc_2) = \mathsf{true}$ and $v_2 = \langle \_ | \_ \rangle$

Subsubcase a: (i.11) is true
By assumption and from (i.4) and (i.11),
(a.1) O-SKIP may be applied to K_2, producing trace $T_2 = G, \mathcal{P} \vdash K_2 \overset{(\bullet, pc_2)}{\Longrightarrow} K_2'$
From (i.6), (i.7), and (a.1),
$K_1' \approx_L K_2'$
From (a.1),
(a.2) $\alpha_{l,2} = (\bullet, pc_2)$
From (V.5), (V.6), (i.5), (a.2), and the definition of equivalence for execution traces,
$T_1 \approx_L T_2$

Subsubcase b: (i.12) is true
By assumption and from (i.4) and (i.12),
(b.1) O may be applied to K_2, producing trace $T_2 = G, \mathcal{P} \vdash K_2 \overset{\alpha_{l,2}}{\Longrightarrow} K_2'$
From (i.6), (i.7), and (b.1),

$K_1' \approx_L K_2'$

From (V.4), (V.6), and the definition of output, either:

(b.2) $pc_1 = L$ and $\mathcal{P}(ch) = H$ or

(b.3) $pc_1 = \cdot$ and $v_1 \downarrow_{\mathcal{P}(ch)} = \cdot$

By assumption and from the structure of the operational semantics,

(b.4) $pc_2 = \cdot$ and $\mathcal{V}_2 = \mathsf{MF}$

If (b.2) is true, then from (b.4), (i.12), and the definition of output,

(b.5) $\mathsf{output}(\mathcal{P}, ch(v_2), pc_2) = \langle ch(\mathsf{getFacet}(v_2, H))|\bullet\rangle$

If (b.3) is true, then from (b.4) and (i.12),

(b.6) $\mathcal{P}(ch) = H$ because otherwise outCondition would have been false

From (b.6) and by the same argument as (b.5),

(b.7) $\mathsf{output}(\mathcal{P}, ch(v_2), pc_2) = \langle ch(\mathsf{getFacet}(v_2, H))|\bullet\rangle$

From (V.5), (V.6), (b.5), (b.7), and the definition of equivalence for execution traces,

$T_1 \approx_L T_2$

Subcase ii: $pc_2 \not\sqsubseteq L$

The proof for this case uses similar logic as **Subcase IV.b** to reach the assumptions for **Subcase i**, at which point the proof proceeds the same as **Subcase i**.

Case VI: $\mathcal{E}$ ends in O-OTHER

The proof for this case is similar to **Case V**. $\alpha_2 = \bullet$ follows from Lemma 25, which tells us that $\alpha_1 \approx_L \alpha_2$ and $\alpha_1 = \bullet$ by assumption.

Case VII: $\mathcal{E}$ ends in O-NEXT

The proof for this case is straightforward. From (7), $ks_1 = (\mathcal{V}_1; \kappa_1; pc_1) :: ks_1''$ and $ks_2 = (\mathcal{V}_2; \kappa_2; pc_2) :: ks_2''$ with $\mathcal{V}_1 = \mathcal{V}_2$ and $\kappa_1 \approx_L \kappa_2$ and $ks_1'' \approx_L ks_2''$ when $pc_2 \sqsubseteq L$. Then, from $\kappa_1 \approx_L \kappa_2$, $\mathsf{consumer}(\kappa_2)$, and O-NEXT can be applied to K_2, which gives $T_1 \approx_L T_2$. $K_1' \approx_L K_2'$ follows from $ks_1'' \approx_L ks_2''$.

When $pc_2 \not\sqsubseteq L$, the proof is similar to **Case IV.b**.

□

Definition 24. $\mathsf{wkK}(\kappa, \alpha)$ *for the mid-level semantics is similar to* wkT *for execution traces: it says that when* $\alpha = \mathsf{br}(b)$*, the mid-level semantics can take a step from* κ *producing the same action.*

Lemma 25 (Strong One-Step - Single Execution Semantics, Weak Secrecy)**.** *If* $G, \mathcal{P}, \mathcal{V}, d \vdash_w \sigma_1^G, \kappa_1 \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, ks_1$ *with* $pc_1 \sqsubseteq L$*,* $\alpha_1 \neq \mathsf{declassify}(\iota, v)$*,* $\sigma_1^G \approx_L \sigma_2^G$*,* $\kappa_1 \approx_L \kappa_2$*,* $\mathsf{wkK}(\kappa_2, \alpha_1)$ *and* $pc_2 \sqsubseteq L$*, then* $G, \mathcal{P}, \mathcal{V}, d \vdash_w \sigma_2^G, \kappa_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, ks_2$ *with* $\sigma_3^G \approx_L \sigma_4^G$*,* $ks_1 \approx_L ks_2$*, and* $\alpha_1 \approx_L \alpha_2$

Proof (sketch): The proof is the same as for Lemma 18 (Requirement **(T4)**) except that it uses Lemma 27 instead of Lemma 19. The proof uses Lemma 66 (Requirement **(EH2)**) for the LC case. $\mathsf{wkK}(\kappa_2, \alpha_1)$ is used to show $\mathsf{wkC}(c_2, \sigma_2^G, \sigma_2, \alpha_1)$ for the command semantics. □

Definition 26. $\mathsf{wkC}(c, \sigma^G, \sigma, \alpha)$ *for the command semantics is similar to* wkT *for execution traces and* wkK *for mid-level semantics: it says that when* $\alpha = \mathsf{br}(b)$*, the command* c *can take a step under* σ^G *and* σ*, producing the same action*

Lemma 27 (Strong One-Step - Command Semantics, Weak Secrecy)**.** *If* $pc_1, pc_2 \sqsubseteq L$ *and* $G, \mathcal{V}, d \Vdash_w \sigma_1^G, \sigma_1, c \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1, E_1$*, with* $\alpha_1 \neq \mathsf{declassify}(\iota, v)$*,* $\mathsf{wkC}(c, \sigma_2^G, \sigma_2, \alpha_1)$ *and* $\sigma_1^G \approx_L \sigma_2$*, and* $\sigma_1 \approx_L \sigma_2^G$*, then* $G, \mathcal{V}, d \Vdash_w \sigma_2^G, \sigma_2, c \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, \sigma_2', c_2, E_2$ *with* $\sigma_3^G \approx_L \sigma_4^G$*,* $\sigma_1' \approx_L \sigma_2'$*,* $\alpha_1 \approx_L \alpha_2$*,* $c_1' \approx_L c_2'$*, and* $E_1 \approx_L E_2$*.*

Proof.

By induction on the structure of $\mathcal{E} :: G, \mathcal{V}, d \Vdash_w \sigma_1^G, \sigma_1, c \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1, E_1$,

The proof is similar to the one for Lemma 19 (Requirement **(T4)**). The most noteworthy differences are shown below:

By assumption,

(1) $pc_1, pc_2 \sqsubseteq L$

(2) $\alpha_1 \neq \mathsf{declassify}(\iota, v)$

(3) $\mathsf{wkC}(c, \sigma_2^G, \sigma_2, \alpha_1)$

(4) $\sigma_1^G \approx_L \sigma_2$

(5) $\sigma_1 \approx_L \sigma_2$

Case I: $\mathcal{E}$ ends in IF-TRUE

The cases for IF-FALSE, WHILE-TRUE, and WHILE-FALSE when $\mathcal{V} = \mathsf{TT}$ are similar.

The proofs for the cases where $\mathcal{V} \neq \mathsf{TT}$ are the same as for Lemma 19 since the rules are unchanged in those cases.

By assumption,

(I.1) $c = \mathsf{if}\ e\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2'$

(I.2) $c_1 = c_1'$
(I.3) $E_1 = \cdot$
(I.4) $\sigma_3^G = \sigma_1^G$
(I.5) $\sigma_1' = \sigma_1$
(I.6) $\alpha_1 = \mathsf{brOutput}((\mathsf{true}, l_1))$
(I.7) $G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e \Downarrow_{pc_1}^{\mathsf{TT}} (\mathsf{true}, l_1)$
From (1), (4), (5), (I.7), and Lemma 37 (Requirement **(WE1)**),
(I.8) $G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e \Downarrow_{pc_2}^{\mathsf{TT}} v_2$ with
(I.9) $v_2 \approx_L (\mathsf{true}, l_1)$
From (I.9),
(I.10) $v_2 = (\mathsf{true}, l_2)$ and $l_1 = l_2$ or
(I.11) $v_2 = (\mathsf{false}, H)$ and $l_1 = H$

Subcase i: (I.10) is true
Applying IF-TRUE produces trace
(i.1) $G, \mathcal{V}, d \Vdash_w \sigma_2^G, \sigma_2, x := e \xrightarrow{\alpha_2}_{pc_2} \sigma_2^G, \sigma_2, c_1', \cdot$
From (i.1),
(i.2) $\sigma_4^G = \sigma_2^G$
(i.3) $\sigma_2' = \sigma_2$
(i.4) $c_2 = c_1'$
(i.5) $E_2 = \cdot$
(i.6) $\alpha_2 = \mathsf{brOutput}((\mathsf{true}, l_2))$
From (4), (I.4), and (i.2),
$\sigma_3^G \approx_L \sigma_4^G$
From (5), (I.5), and (i.3),
$\sigma_1' \approx_L \sigma_2'$
From (I.10), (I.6), (i.6), and the definition of brOutput,
(i.7) If $l_1 = l_2 = L$, then $\alpha_1 = \alpha_2 = \bullet$
(i.8) Otherwise, $l_1 = l_2 = H$ and $\alpha_1 = \alpha_2 = \mathsf{br}(\mathsf{true})$
From (i.7) and (i.8),
$\alpha_1 \approx_L \alpha_2$
From (I.2) and (i.4),
$c_1 \approx_L c_2$
From (I.3) and (i.5),
$E_1 \approx_L E_2$

Subcase ii: (I.11) is true
Applying IF-FALSE produces trace
(ii.1) $G, \mathcal{V}, d \Vdash \sigma_2^G, \sigma_2, x := e \xrightarrow{\alpha_2}_{pc_2} \sigma_2^G, \sigma_2, c_2, \cdot$
From (ii.1) and (I.11),
(ii.2) $\alpha_2 = \mathsf{brOutput}((\mathsf{false}, H))$
From (I.11), (ii.2), (I.6), and the definition of brOutput,
(ii.3) $\alpha_1 = \mathsf{br}(\mathsf{true})$
(ii.4) $\alpha_2 = \mathsf{br}(\mathsf{false})$
But (ii.3) and (ii.4) contradicts (3), so this case holds vacuously

Case II: $\mathcal{E}$ ends in ASSIGN-G
The cases for ASSIGN-D and CREATEELEM when $\mathcal{V} = \mathsf{TT}$ are similar to the case from Lemma 19 (Requirement **(T4)**), except that it uses Lemma 45 (Requirement **(WV3)**) instead of Lemma 44 (Requirement **(V3)**) for ASSIGN-G, Lemma 82 (Requirement **(WEH4)**) instead of Lemma 80 (Requirement **(EH4)**) for ASSIGN-D, and Lemma 83 (Requirement **(WEH4)**) instead of Lemma 81 (Requirement **(EH4)**) for CREATEELEM.
The proofs for the cases where $\mathcal{V} \neq \mathsf{TT}$ are the same as for Lemma 19 (Requirement **(T4)**) since the rules always return $\bullet$ in those cases, meaning they are effectively the same rules.

□

Requirement (T5) Weak one-step

Lemma 28 (Weak One-Step)**.** *If* $T_1 = G, \mathcal{P} \vdash K_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1'$ *and* $T_2 = G, \mathcal{P} \vdash K_2 \overset{\alpha_{l,2}}{\Longrightarrow} K_2'$*, with* $T_1 \approx_L T_2$*,* $K_1 \approx_L K_2$*,* $T_1 \downarrow_L \neq \cdot$*, and* $T_2 \downarrow_L \neq \cdot$*, then* $K_1' \approx_L K_2'$

Proof.

We examine each case of $\mathcal{E} :: G, \mathcal{P} \vdash K_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1'$

Denote $\mathcal{D} :: G, \mathcal{P} \vdash K_2 \overset{\alpha_{l,2}}{\Longrightarrow} K_2'$

By assumption,

(1) $K_1 \approx_L K_2$

(2) $T_1 \approx_L T_2$

(3) $T_1 \downarrow_L \neq \cdot$

(4) $T_2 \downarrow_L \neq \cdot$

From (1),

(5) $\sigma_1 \approx_L \sigma_2$

(6) $ks_1 \approx_L ks_2$

Case I: $\mathcal{E}$ ends in I-NR1 or I-NR2

By assumption, $T_1 \downarrow_L = \cdot$, which contradicts (3), so this case holds vacuously.

Case II: $\mathcal{E}$ ends in I-R-DIFF

By assumption,

(II.1) $\mathcal{R}_1 = (\rho_1, D_1)$ with $D_1(\rho_1, id.Ev(v)) = (\rho_1', r, v')$

(II.2) $G, \mathcal{P}, \sigma_1 \vdash \cdot; \mathsf{lookupEHAt}(id.Ev(v')) \rightsquigarrow_L ks_1''$

(II.3) $G, \mathcal{P}, \sigma_1 \vdash ks_1''; \mathsf{lookupEHAt}(id.Ev(v)) \rightsquigarrow_H \kappa_1'$

(II.4) $\sigma_1' = \sigma_1$

From (II.1),

(II.5) $T_1 \downarrow_L = \mathsf{rls}(id.Ev(v'))$

From (2) and (II.5),

(II.6) $\mathcal{D}$ must end in I-R-DIFF or I-R-SAME with input $\alpha_{l,2}$ s.t. $T_2 \downarrow_L = \mathsf{rls}(id.Ev(v'))$

Subcase i: $\mathcal{D}$ ends in I-R-DIFF

By assumption,

(i.1) $G, \mathcal{P}, \sigma_2 \vdash \cdot; \mathsf{lookupEHAt}(id.Ev(v')) \rightsquigarrow_L ks_2''$

(i.2) $G, \mathcal{P}, \sigma_2 \vdash ks_2''; \mathsf{lookupEHAt}(\alpha_{l,2}) \rightsquigarrow_H \kappa_2'$

(i.3) $\sigma_2' = \sigma_2$

From (5), (II.2), (i.1), and Lemma 56 (Requirement **(EH1)**),

(i.4) $ks_1'' \approx_L ks_2''$

From (II.3), (i.2), and Lemma 66 (Requirement **(EH2)**),

(i.5) $ks_1'' \approx_L ks_1'$

(i.6) $ks_2'' \approx_L ks_2'$

From (i.4)-(i.6),

(i.7) $ks_1' \approx_L ks_2'$

From (5), (II.4), and (i.3),

(i.8) $\sigma_1' \approx_L \sigma_2'$

From (i.7) and (i.8),

$K_1' \approx_L K_2'$

Subcase ii: $\mathcal{D}$ ends in I-R-SAME

By assumption,

(ii.1) $G, \mathcal{P}, \sigma_2 \vdash \cdot; \mathsf{lookupEHAll}(id.Ev(v')) \rightsquigarrow . ks_2'$

(ii.2) $\sigma_2' = \sigma_2$

From (5), (ii.1), (II.2), and Lemma 56 (Requirement **(EH1)**),

(ii.3) $ks_1'' \approx_L ks_2'$

From (II.3), and Lemma 66 (Requirement **(EH2)**),

(ii.4) $\kappa_1'' \approx_L \kappa_1'$

From (ii.3) and (ii.4),

(ii.5) $\kappa_1' \approx_L \kappa_2'$

From (5), (II.4), and (ii.2),

(ii.6) $\sigma_1' \approx_L \sigma_2'$

From (ii.5) and (ii.6),

$K_1' \approx_L K_2'$

Case III: $\mathcal{E}$ ends in I-R-SAME or I-L
The proof for these cases are similar to **Case II**

Case IV: $\mathcal{E}$ ends in O
By assumption,
(IV.1) $ks_1 = (\mathcal{V}; \kappa_1; pc_1) :: \kappa_1''$
(IV.2) $\mathsf{producer}(\kappa_1)$
(IV.3) $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1, \kappa_1 \xrightarrow{ch(v_1)}_{pc_1} \sigma_1', ks_1'''$
(IV.4) $\alpha_{l,1} = \mathsf{output}(\mathcal{P}, ch(v_1), pc_1)$
(IV.5) $\mathsf{outCondition}_{\mathcal{V}}(\mathcal{P}, ch(v_1), pc_1) = \mathsf{true}$
(IV.6) $ks_1' = ks_1''' :: ks_1''$
From (3) and the definition of trace projection,
(IV.7) $pc_1 \sqsubseteq L$ or
(IV.8) $\mathcal{P}(ch) = L$ or
(IV.9) $\alpha_{l,1} = \langle \_ | \_ \rangle$ and $\mathsf{getFacet}(\alpha_{l,1}, L) \neq \cdot$
If (IV.8) is true, then from (IV.3), (IV.4), and the definition of output and outCondition: $pc_1 \sqsubseteq L$
If (IV.9) is true, then from (IV.3), (IV.4), and the definition of output and outCondition: $pc_1 = \cdot$
Then from (IV.7)-(IV.9),
(IV.10) $pc_1 \sqsubseteq L$
From (6), (IV.1), and (IV.10),
(IV.11) $ks_2 = (\mathcal{V}_2; \kappa_2; pc_2) :: ks_2''$
From the operational semantics, either
(IV.12) $\mathcal{D}$ ends in O or
(IV.13) $\alpha_{l,2} = (\_, pc_2)$
If (IV.12) is true, then from (2) and by a similar argument to above: $pc_2 \sqsubseteq L$
If (IV.13) is true, then from (4) and the definition of trace projection: $pc_2 \sqsubseteq L$
Then from (IV.12) and (IV.13),
(IV.14) $pc_2 \sqsubseteq L$
From (6), (IV.1), (IV.11), (IV.10), and (IV.14),
(IV.15) $\mathcal{V}_2 = \mathcal{V}$
(IV.16) $\kappa_1 \approx_L \kappa_2$ and
(IV.17) $ks_1'' \approx_L ks_2''$

Subcase i: $\mathcal{D}$ ends in O
By assumption,
(i.1) $\exists \mathcal{D}' :: G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_2, \kappa_2 \xrightarrow{ch(v_2)}_{pc_2} \sigma_2', ks_2'''$
(i.2) $ks_2' = ks_2'' :: ks_2'''$
From (IV.3), (i.1), (5), (IV.16), and Lemma 29,
(i.3) $\sigma_1' \approx_L \sigma_2'$
(i.4) $ks_1''' \approx_L ks_2'''$
From (IV.6), (i.2), (i.4), and (IV.17),
(i.5) $ks_1' \approx_L ks_2'$
From (i.3) and (i.5),
$K_1' \approx_L K_2'$

Subcase ii: $\mathcal{D}$ ends in O-SKIP or O-OTHER
The proofs for these cases are similar to **Subcase i**.

Subcase iii: $\mathcal{D}$ ends in O-NEXT
By assumption, $\mathsf{consumer}(\kappa_2)$
But this contradicts (IV.2) and (IV.16), so this case holds vacuously.

Case V: $\mathcal{E}$ ends in O-SKIP or O-OTHER
The proof for this case is similar to the proof for **Case IV**. The biggest difference is that $pc_1 \sqsubseteq L$ follows from (3) and the definition of trace projection.

Case VI: $\mathcal{E}$ ends in O-NEXT
By assumption,
(VI.1) $ks_1 = (\mathcal{V}; \kappa_1; pc_1) :: ks_1'$
(VI.2) $\mathsf{consumer}(\kappa_1)$

(VI.3) $\sigma_1' = \sigma_1$
(VI.4) $\alpha_{l,1} = (\bullet, pc_1)$
From (3), (VI.4), and the definition of trace projection,
(VI.5) $pc_1 \sqsubseteq L$
From (6), (VI.1), and (VI.5),
(VI.6) $ks_2 = (\mathcal{V}_2; \kappa_2; pc_2) :: ks_2''$
From the operational semantics, either
(VI.7) $\mathcal{D}$ ends in O or
(VI.8) $\alpha_{l,2} = (\_, pc_2)$
If (VI.7) is true, then from (2) and by a similar argument to the one in **Case IV**: $pc_2 \sqsubseteq L$
If (VI.8) is true, then from (4) and the definition of trace projection: $pc_2 \sqsubseteq L$
Then, from (VI.7) and (VI.8),
(VI.9) $pc_2 \sqsubseteq L$
From (6), (VI.1), (VI.6), (VI.5), and (VI.9),
(VI.10) $\mathcal{V}_2 = \mathcal{V}$
(VI.11) $\kappa_1 \approx_L \kappa_2$
(VI.12) $ks_1' \approx_L ks_2''$
From (VI.2) and (VI.11),
(VI.13) $\mathsf{consumer}(\kappa_2)$
From (VI.13),
(VI.14) $\mathcal{D}$ must end in O-NEXT
From (VI.14) and (VI.6),
(VI.15) $\sigma_2' = \sigma_2$
(VI.16) $ks_2' = ks_2''$
From (5), (VI.3), and (VI.15),
(VI.17) $\sigma_1' \approx_L \sigma_2'$
From (6), (VI.12), and (VI.16),
(VI.18) $ks_1' \approx_L ks_2'$
From (VI.17) and (VI.18),
$K_1' \approx_L K_2'$

□

Lemma 29 (Weak One-Step - Single Execution Semantics)**.** *If* $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa_1 \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, ks_1$*, and* $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_2^G, \kappa_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, ks_2$ *with* $pc_1, pc_2 \sqsubseteq L$*,* $\sigma_1^G \approx_L \sigma_2^G$*, and* $\kappa_1 \approx_L \kappa_2$*, then* $\sigma_3^G \approx_L \sigma_4^G$ *and* $ks_1 \approx_L ks_2$

Proof.
By induction on the structure of $\mathcal{E} :: G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa_1, \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, ks_1$
By assumption
(1) $\sigma_1^G \approx_L \sigma_2^G$
(2) $\kappa_1 \approx_L \kappa_2$
(3) $pc_1, pc_2 \sqsubseteq L$
From (2),
(4) $\sigma_1 \approx_L \sigma_2$
(5) $c_1 = c_2$
(6) $s_1 = s_2$
(7) $E_1 \approx_L E_2$
Denote
$\mathcal{D} :: G, \mathcal{P}, \mathcal{V}, d, \vdash \sigma_2^G, \kappa_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, ks_2$

Case I: $\mathcal{E}$ ends in LC
By assumption,
(I.1) $c_1 = \mathsf{skip}$
(I.2) $s_1 = P$
(I.3) $E_1 \neq \cdot$
(I.4) $\sigma_3^G = \sigma_1^G$
(I.5) $\mathcal{I}, \mathcal{P}, \mathcal{V}, \sigma_1^G \vdash (\mathcal{V}; (\sigma_1, \mathsf{skip}, C, \cdot); pc_1); \mathsf{lookupEHs}(E_1) \rightsquigarrow_{pc_1} ks_1$
From (3), (5)-(7) and (I.1)-(I.3),
(I.6) $c_2 = c_1 = \mathsf{skip}$
(I.7) $s_2 = s_1 = P$
(I.8) $E_2 \approx_L E_1$

Subcase i: $E_2 = \cdot$
By assumption,

(i.1) The last rule applied to $\mathcal{D}$ must have been PToC
From (i.1),
(i.2) $\sigma_4^G = \sigma_2^G$
(i.3) $ks_2 = (\mathcal{V}; (\sigma_2; \mathsf{skip}, C, \cdot); pc_2)$
By assumption and from (I.8),
(i.4) $E_1 \approx_L \cdot$
From (i.4), (I.5), and Lemma 66 (Requirement **(EH2)**),
(i.6) $ks_1 \approx_L (\mathcal{V}; (\sigma_1, \mathsf{skip}, C, \cdot); pc_1)$
From (3), (4), (i.3), and (i.6),
$ks_1 \approx_L ks_2$
From (1), (I.4), and (i.2),
$\sigma_3^G \approx_L \sigma_4^G$

Subcase ii: $E_2 \neq \cdot$
By assumption,
(ii.1) The last rule applied to $\mathcal{D}$ must have been LC
From (ii.1),
(ii.2) $\sigma_4^G = \sigma_2^G$
(ii.3) $G, \mathcal{P}, \mathcal{V}, \sigma_2^G \vdash (\mathcal{V}; (\sigma_2, \mathsf{skip}, C, \cdot); pc_2); \mathsf{lookupEHs}(E_2) \rightsquigarrow_{pc_2} ks_2$
From (1), (I.4), and (i.2),
$\sigma_3^G \approx_L \sigma_4^G$
From (1), (I.5), (ii.3), (4), (7), and Lemma 56 (Requirement **(EH1)**),
$ks_1 \approx_L ks_2$

Case II: $\mathcal{E}$ ends in PToC
The proof for this case is similar to **Case I**

Case III: $\mathcal{E}$ ends in P
By assumption and from the structure of the operational semantics,
(III.1) $c_1 \neq \mathsf{skip}$
By assumption,
(III.2) $s_1 = P$
(III.3) $\exists \mathcal{E}' :: G, \mathcal{P}, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c_1 \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1', E_1''$
(III.4) $s_1' = P$
(III.5) $E_1' = (E_1, E_1'')$
(III.6) $ks_1 = (\mathcal{V}; (\sigma_1', c_1', P, E_1'); pc_1)$
From (3), (5), (6), (III.1), and (III.2),
(III.7) $c_2 \neq \mathsf{skip}$
(III.8) $s_2 = P$
From (III.7) and (III.8),
(III.9) $\mathcal{D}$ must end in P or P-F
From (III.9),
(III.10) $\exists \mathcal{D}' :: G, \mathcal{P}, \mathcal{V}, d \Vdash \sigma_2^G, \sigma_2, c_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, \sigma_2', c_2', E_2''$
From (1), (3)-(5), (III.3), (III.10), and Lemma 30,
$\sigma_3^G \approx_L \sigma_4^G$
(III.11) $\sigma_1' \approx_L \sigma_2'$
(III.12) $c_1' \approx_L c_2'$
(III.13) $E_1'' \approx_L E_2''$

Subcase i: $c_2' \neq \langle \_ | \_ \rangle$
By assumption,
(i.1) $\mathcal{D}$ must end in P
From (i.1),
(i.2) $ks_2 = (\mathcal{V}; (\sigma_2', c_2', P, (E_2, E_2'')); pc_2)$
From (III.6), (i.2), (III.11)-(III.13), and (III.5),
$ks_1 \approx_L ks_2$

Subcase ii: $c_2' = \langle c_H | c_L \rangle$
By assumption and from (III.12),
(ii.1) $c_L = c_1'$
By assumption,
(ii.2) $\mathcal{D}$ must end in P-F

From (ii.2),
(ii.3) $ks_2 = (\mathsf{MF}; (\sigma_2', c_L, P, (E_2, E_2'')); L) :: (\mathsf{MF}; (\sigma_2', c_H, P, (E_2, E_2'')); H)$
From (III.6), (ii.3), (III.11), (ii.1), (III.12), and (III.5),
$ks_1 \approx_L ks_2$

Case IV: $\mathcal{E}$ ends in SME-L
By assumption,
(IV.1) $\exists \mathcal{E}' :: G, \mathcal{P}, \mathsf{SME}, d \vdash \sigma_1^G, \kappa_{L,1} \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \kappa_{L,1}'$
(IV.2) $\kappa_1 = \kappa_{H,1}, \kappa_{L,1}$
(IV.3) $ks_1 = (\mathsf{SME}; (\kappa_{H,1}; \kappa_{L,1}'); L)$
(IV.4) $\neg\mathsf{consumer}(\kappa_{L,1}')$
From (2),
(IV.5) $\kappa_2 = \kappa_{H,2}, \kappa_{L,2}$ with
(IV.6) $\kappa_{L,2} = \kappa_{L,1}$
From (IV.6),
(IV.7) $\mathcal{D}$ ends in SME-L or SME-LTOH
From (IV.7),
(IV.8) $\exists \mathcal{D}' :: G, \mathcal{P}, \mathsf{SME}, d \vdash \sigma_2^G, \kappa_{L,2} \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, \kappa_{L,2}'$
IH on $\mathcal{E}'$, $\mathcal{D}'$ gives
$\sigma_3^G \approx_L \sigma_4^G$
(IV.9) $\kappa_{L,1}' \approx_L \kappa_{L,2}'$
From (IV.9),
(IV.10) $\kappa_{L,1}' = \kappa_{L,2}'$
From (IV.10) and (IV.4),
(IV.11) $\neg\mathsf{consumer}(\kappa_{L,2}')$
From (IV.11),
(IV.12) $\mathcal{D}$ must end in SME-L
From (IV.12),
(IV.13) $ks_2 = (\mathsf{SME}; (\kappa_{H,2}; \kappa_{L,2}'); L)$
From (IV.3) and (IV.13),
$ks_1 \approx_L ks_2$

Case V: $\mathcal{E}$ ends in SME-LTOH
The proof for this case is similar to **Case IV**. The IH gives $\mathcal{D}$ must end in SME-LTOH.

Case VI: $\mathcal{E}$ ends in SME-H
By assumption, $pc_1 = H$, which contradicts (3), so this case holds vacuously.

Case VII: $\mathcal{E}$ ends in P-L
The proof for this case is similar to **Case III**

□

Lemma 30 (Weak One-Step - Command Semantics)**.** *If* $G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1, E_1$ *and* $G, \mathcal{V}, d \Vdash \sigma_2^G, \sigma_2, c \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, \sigma_2', c_2, E_2$*, with* $pc_1, pc_2 \sqsubseteq L$*,* $\sigma_1^G \approx_L \sigma_2^G$*, and* $\sigma_1 \approx_L \sigma_2$*, then* $\sigma_3^G \approx_L \sigma_4^G$*,* $\sigma_1' \approx_L \sigma_2'$*,* $c_1 \approx_L c_2$*, and* $E_1 \approx_L E_2$

Proof.
By induction on the structure of $\mathcal{E} :: G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1, E_1$.
Denote $\mathcal{D} :: G, \mathcal{V}, d \Vdash \sigma_2^G, \sigma_2, c \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, \sigma_2', c_2, E_2$
All of the cases are straightforward. We outline the proofs and show where additional lemmas are needed.

Case I: $\mathcal{E}$ ends in SKIP, OUTPUT
The proof for these cases are straightforward. No additional lemmas are used.

Case II: $\mathcal{E}$ ends in SEQ
$\mathcal{D}$ ends in SEQ if the resulting command is un-faceted, and SEQ-F if the command is faceted. The conclusion follows from the IH, either way.

Case III: $\mathcal{E}$ ends in DECLASSIFY-H
By assumption, $pc_1 = H$, which contradicts (2), so this case holds vacuously.

Unstructured EH storage:

Case U.I: $\mathcal{E}$ ends in ASSIGN-L or ASSIGN-G
The proofs for these cases use Lemma 36.U (Requirement **(E1)**) and Lemma 44.U (Requirement **(V3)**) to show that the resulting stores are equivalent.

Case U.II: $\mathcal{E}$ ends in IF-TRUE, IF-FALSE, WHILE-TRUE, WHILE-FALSE, IF-F, or WHILE-F
Lemma 36.U (Requirement **(E1)**) gives that the expressions evaluate to (strong) equivalent values. If the result is not faceted, $\mathcal{D}$ ends in the standard rule (IF-TRUE, IF-FALSE, etc.). Otherwise, if the value is faceted, $\mathcal{D}$ ends in IF-F or WHILE-F.

Case U.III: $\mathcal{E}$ ends in DECLASSIFY-L or DECLASSIFY-NC
The proofs for these cases are similar to **Case U.II**. If $pc_2 = \cdot$, then $\mathcal{D}$ must end in DECLASSIFY-NC. Otherwise, it ends in DECLASSIFY-L. The equivalence of the resulting commands follows from the definition of setFacetC.

Case U.IV $\mathcal{E}$ ends in ASSIGN-D
The proofs for this case uses Lemma 36.U (Requirement **(E1)**) and Lemma 80.U (Requirement **(EH4)**) to show that the resulting stores are equivalent.

Case U.V: $\mathcal{E}$ ends in CREATEELEM
The proof for this case uses Lemma 36.U (Requirement **(E1)**) and Lemma 81.U (Requirement **(EH4)**) to prove that the resulting stores are equivalent.

Case U.VI: $\mathcal{E}$ ends in ADDEH
This case uses Lemma 81.U (Requirement **(EH4)**) to show that the resulting stores are equivalent.

Case U.VII: $\mathcal{E}$ ends in TRIGGER
This case uses Lemma 63 (Requirement **(EH1)**) to show that the event queues are equivalent and Lemma 36.U (Requirement **(E1)**) to show that the arguments passed to the events are (strong) equivalent.

Tree structure EH storage:
Case T.I: $\mathcal{E}$ ends in ASSIGN-L or ASSIGN-G
The proofs for these cases use Lemma 36.T (Requirement **(E1)**) and Lemma 44.T (Requirement **(V3)**) to show that the resulting stores are equivalent.

Case T.II: $\mathcal{E}$ ends in IF-TRUE, IF-FALSE, WHILE-TRUE, WHILE-FALSE, IF-F, or WHILE-F
Lemma 36.T (Requirement **(E1)**) gives that the expressions evaluate to (strong) equivalent values. If the result is not faceted, $\mathcal{D}$ ends in the standard rule (IF-TRUE, IF-FALSE, etc.). Otherwise, if the value is faceted, $\mathcal{D}$ ends in IF-F or WHILE-F.

Case T.III: $\mathcal{E}$ ends in DECLASSIFY-L or DECLASSIFY-NC
The proofs for these cases are similar to **Case T.II**. If $pc_2 = \cdot$, then $\mathcal{D}$ must end in DECLASSIFY-NC. Otherwise, it ends in DECLASSIFY-L. The equivalence of the resulting commands follows from the definition of setFacetC.

Case T.IV $\mathcal{E}$ ends in ASSIGN-D
The proofs for this case uses Lemma 36.T (Requirement **(E1)**) and Lemma 80.T (Requirement **(EH4)**) to show that the resulting stores are equivalent.

Case T.V: $\mathcal{E}$ ends in CREATECHILD or CREATESIBLING
The proofs for these cases use Lemma 36.T (Requirement **(E1)**) and Lemma 81.T (Requirement **(EH4)**) to prove that the resulting stores are equivalent.

Case T.VI: $\mathcal{E}$ ends in ADDEH
This case uses Lemma 81.T (Requirement **(EH4)**) to show that the resulting stores are equivalent.

Case T.VII: $\mathcal{E}$ ends in TRIGGER
This case uses Lemma 63 (Requirement **(EH1)**) to show that the event queues are equivalent and Lemma 36.T (Requirement **(E1)**) to show that the arguments passed to the events are (strong) equivalent.

□

Requirement (WT5) Weak one-step (Weak Secrecy)

Lemma 31 (Weak One-Step, Weak Secrecy)**.** *If* $T_1 = G, \mathcal{P} \vdash_w K_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1'$ *and* $T_2 = G, \mathcal{P} \vdash_w K_2 \overset{\alpha_{l,2}}{\Longrightarrow} K_2'$*, with* $T_1 \approx_L T_2$*,* $K_1 \approx_L K_2$*,* $T_1 \downarrow_L \neq \cdot$*, and* $T_2 \downarrow_L \neq \cdot$*, then* $K_1' \approx_L K_2'$

Proof.

We examine each case of $\mathcal{E} :: G, \mathcal{P} \vdash_w K_1 \overset{\alpha_{l,1}}{\Longrightarrow} K_1'$
Denote $\mathcal{D} :: G, \mathcal{P} \vdash_w K_2 \overset{\alpha_{l,2}}{\Longrightarrow} K_2'$
By assumption,
(1) $K_1 \approx_L K_2$
(2) $T_1 \downarrow_L \neq \cdot$
(3) $T_1 \downarrow_L \neq \cdot$
(4) $T_2 \downarrow_L \neq \cdot$
From (1),
(5) $\sigma_1 \approx_L \sigma_2$
(6) $ks_1 \approx_L ks_2$

Case I: $\mathcal{E}$ ends in an input rule
The proofs for these cases are the same as Lemma 28 (Requirement **(T5)**). They use Lemma 56 (Requirement **(EH1)**) and Lemma 66 (Requirement **(EH2)**).

Case II: $\mathcal{E}$ ends in and output rule and $T_1 \downarrow_L = T_2 \downarrow_L = \mathsf{gw}(\_)$
Note that the only rules to emit $\mathsf{gw}(\_)$ are ASSIGN-G-H, ASSIGN-D-H, and CREATEELEM-H, which only run in the H context. The proof for this case follows from Lemma 32.

Case III: $\mathcal{E}$ ends in an output rule O, O-SKIP, or O-OTHER and $T_1 \downarrow_L = T_2 \downarrow_L \neq \mathsf{gw}(\_)$
The proof for this case is similar to the output cases for Lemma 28 (Requirement **(T5)**). Note that all other visible outputs still happen in the L context. These cases use Lemma 34 instead of Lemma 29.
The biggest difference here is that we need to show $\alpha_1 = \alpha_2$ when $\alpha_1, \alpha_2 \neq ch(\_)$ to apply Lemma 34.

Denote $\mathcal{E}' :: G, \mathcal{P}, \mathcal{V}, d \vdash_w \sigma_1, \kappa_1 \xrightarrow{\alpha_1}_{pc_1} \sigma_1', ks_1$
$\mathcal{D}' :: G, \mathcal{P}, \mathcal{V}, d \vdash_w \sigma_2, \kappa_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_2', ks_2$
To show: $\alpha_1 = \alpha_2$ when $\alpha_1, \alpha_2 \neq ch(\_)$
Assume that $\alpha_1, \alpha_2 \neq ch(\_)$
When $\mathcal{E}$ or $\mathcal{D}$ end in O or O-SKIP,
$\alpha_1 = ch(\_)$ or $\alpha_2 = ch(\_)$
Then, by assumption,
(III.1) $\mathcal{E}$ and $\mathcal{D}$ ends in O-OTHER
From (III.1),
(III.2) $\alpha_{l,1} = (\alpha_1, pc_1)$ and
(III.3) $\alpha_{l,2} = (\alpha_2, pc_2)$
From (III.1), (III.2), and (III.3),
(III.4) $\alpha_{l,1}, \alpha_{l,2} \neq \langle | \rangle$
(III.5) $\alpha_{l,1}, \alpha_{l,2} \neq id.Ev(v)$ and
(III.6) $\alpha_{l,1}, \alpha_{l,2} \neq \mathsf{rls}(\_)$
From (III.4)-(III.6), (3), (4), and the definition of trace projection,
(III.7) $T_1 \downarrow_L = \alpha_1$ and
(III.8) $T_2 \downarrow_L = \alpha_2$
From (2), (III.7), (III.8), and the definition of equivalence for traces,
$\alpha_1 = \alpha_2$

Case IV: $\mathcal{E}$ ends in O-NEXT
The proof for this case is the same as from Lemma 28 (Requirement **(T5)**).

□

Lemma 32 (Weak One-Step - Single Execution Semantics - H context, Weak secrecy)**.** *If* $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_1^G, \kappa_1 \xrightarrow{\alpha_1}_H \sigma_3^G, ks_1$*, and* $G, \mathcal{P}, \mathcal{V}, d \vdash \sigma_2^G, \kappa_2 \xrightarrow{\alpha_2}_H \sigma_4^G, ks_2$ *with* $\alpha_1 = \alpha_2 = \mathsf{gw}(\_)$*,* $\sigma_1^G \approx_L \sigma_2^G$*, then* $\sigma_3^G \approx_L \sigma_4^G$ *and* $ks_1 \approx_L ks_2$

Proof (sketch): By induction on the structure of both traces. The only cases where $\alpha_1 = \alpha_2 = \mathsf{gw}(\_)$ is when the traces end in P or SME-H. The first case uses Lemma 33 and the second uses the IH. Since the resulting ks is still in the H context in both cases, showing $ks_1 \approx_L ks_2$ is trivial. Lemma 33 is used to show that $\sigma_3^G \approx_L \sigma_4^G$ □

Lemma 33 (Weak One-Step - Command Semantics, Weak Secrecy)**.** *If* $G, \mathcal{V}, d \Vdash_w \sigma_1^G, \sigma_1, c_1 \xrightarrow{\alpha_1}_H \sigma_3^G, \sigma_1', c_1, E_1$ *and* $G, \mathcal{V}, d \Vdash_w \sigma_2^G, \sigma_2, c_2 \xrightarrow{\alpha_2}_H \sigma_4^G, \sigma_2', c_2, E_2$*, with* $\alpha_1 = \alpha_2 = \mathsf{gw}(\_)$ *and* $\sigma_1^G \approx_L \sigma_2^G$ *then* $\sigma_3^G \approx_L \sigma_4^G$

Proof.
By induction on the structure of $\mathcal{E} :: G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c_1 \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1, E_1$.
Denote $\mathcal{D} :: G, \mathcal{V}, d \Vdash \sigma_2^G, \sigma_2, c_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, \sigma_2', c_2, E_2$
By assumption,

(1) $\alpha_1 = \alpha_2 = \mathsf{gw}(\_)$
(2) $\sigma_1^G \approx_L \sigma_2^G$
From (1) and the assumption that $\mathcal{E}$ and $\mathcal{D}$ are in the H context,
$\mathcal{E}$ and $\mathcal{D}$ must end in ASSIGN-G-H, ASSIGN-D-H, or CREATEELEM

Case I: $\mathcal{E}$ ends in ASSIGN-G-H
By assumption,
(I.1) $\mathsf{assignW}_{G\downarrow_g}(\sigma_1^G, H, x, v_1) = (\sigma_3^G, \alpha_1)$
By assumption and from (1),
(I.2) $\alpha_1 = \mathsf{gw}(x)$
From (1) and (I.2),
(I.3) $\alpha_2 = \mathsf{gw}(x)$
From (I.3),
(I.4) $\mathcal{D}$ ends in ASSIGN-G-H
From (I.4),
(I.5) $\mathsf{assignW}_{G\downarrow_g}(\sigma_2^G, H, x, v_2) = (\sigma_4^G, \alpha_2)$
From (2), (I.1)-(I.3), (I.5), and Lemma 43 (Requirement **(WV2)**),
$\sigma_3^G \approx_L \sigma_4^G$

Case II: $\mathcal{E}$ ends in ASSIGN-D-H
By assumption,
(II.1) $\mathsf{assignW}_{G\downarrow_{EH}}(\sigma_1^G, H, id, v_1) = (\sigma_3^G, \alpha_1)$
By assumption and from (1),
(II.2) $\alpha_1 = \mathsf{gw}(id)$
From (1) and (II.2),
(II.3) $\alpha_2 = \mathsf{gw}(id)$
From (II.3),
(II.4) $\mathcal{D}$ ends in ASSIGN-D-H or CREATEELEM

Subcase i: $\sigma_1^G(id) = (id, v, M, l)$ with $l \sqsubseteq L$
By assumption and from (2) and the definition of $\approx_L$ for EH stores,
(i.1) $id \in \sigma_2^G(id)$
From (i.1) and (II.4),
(i.2) $\mathcal{D}$ must end in ASSIGN-D-H
From (i.2),
(i.3) $\mathsf{assignW}_{G\downarrow_{EH}}(\sigma_2^G, H, id, v_2) = (\sigma_4^G, \alpha_2)$
From (2), (II.1)-(II.3), (i.3), and Lemma 75 (Requirement **(WEH3)**),
$\sigma_3^G \approx_L \sigma_4^G$

Subcase ii: $\sigma_1^G(id) = (id, v, M, l)$ with $l \not\sqsubseteq L$
By assumption and from (2) and the definition of $\approx_L$ for EH stores, either
(ii.1) $id \in \sigma_2^G$ with $\sigma_2^G(id) = (id, \_, \_, H)$ or
(ii.2) $id \notin \sigma_2^G$
If (ii.1) is true, the proof is the same as **Subcase i**.
Otherwise, (ii.2) is true
From (ii.2),
(ii.3) $\mathcal{D}$ must end in CREATEELEM
From (ii.3),
(ii.4) $\mathsf{assignW}_{G\downarrow_{EH}}(\sigma_2^G, H, id, v) = (\sigma_4^G, \alpha_2)$
From (2), (II.1), (ii.4), (II.2), (II.3), and Lemma 77 (Requirement **(WEH3)**),
$\sigma_3^G \approx_L \sigma_4^G$

Case III: $\mathcal{E}$ ends in CREATEELEM
The proof for this case is similar to **Case II**. It uses Lemma 78 (Requirement **(WEH3)**) instead of Lemma 75.

□

Lemma 34 (Weak One-Step - Single Execution Semantics, Weak Secrecy)**.** *If* $G, \mathcal{P}, \mathcal{V}, d \vdash_w \sigma_1^G, \kappa_1 \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, ks_1$, *and* $G, \mathcal{P}, \mathcal{V}, d \vdash_w \sigma_2^G, \kappa_2 \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, ks_2$ *with* $pc_1, pc_2 \sqsubseteq L$, $\alpha_1 = \alpha_2$ *when* $\alpha_1, \alpha_2 \neq ch(\_)$ $\sigma_1^G \approx_L \sigma_2^G$, *and* $\kappa_1 \approx_L \kappa_2$, *then* $\sigma_3^G \approx_L \sigma_4^G$ *and* $ks_1 \approx_L ks_2$

Proof (sketch): The proof is similar to the one for Lemma 29 (Requirement **(T5)**) except that it uses Lemma 35 instead of Lemma 30. It uses Lemma 66 (Requirement **(EH2)**) and Lemma 56 (Requirement **(EH1)**). □

Lemma 35 (Weak One-Step - Command Semantics, Weak Secrecy)**.** *If* $G, \mathcal{V}, d \Vdash_w \sigma_1^G, \sigma_1, c \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1, E_1$ *and* $G, \mathcal{V}, d \Vdash_w \sigma_2^G, \sigma_2, c \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, \sigma_2', c_2, E_2$*, with* $pc_1, pc_2 \sqsubseteq L$*,* $\alpha_1 = \alpha_2$ *when* $\alpha_1, \alpha_2 \neq ch(\_)$*,* $\sigma_1^G \approx_L \sigma_2^G$*, and* $\sigma_1 \approx_L \sigma_2$*, then* $\sigma_3^G \approx_L \sigma_4^G$*,* $\sigma_1' \approx_L \sigma_2'$*,* $c_1 \approx_L c_2$*, and* $E_1 \approx_L E_2$

Proof.
By induction on the structure of $\mathcal{E} :: G, \mathcal{V}, d \Vdash \sigma_1^G, \sigma_1, c \xrightarrow{\alpha_1}_{pc_1} \sigma_3^G, \sigma_1', c_1, E_1$.
Denote $\mathcal{D} :: G, \mathcal{V}, d \Vdash \sigma_2^G, \sigma_2, c \xrightarrow{\alpha_2}_{pc_2} \sigma_4^G, \sigma_2', c_2, E_2$
By assumption,
(1) $pc_1, pc_2 \sqsubseteq L$
(2) $\alpha_1 = \alpha_2$ when $\alpha_1, \alpha_2 \neq ch(\_)$
(3) $\sigma_1^G \approx_L \sigma_2^G$
(4) $\sigma_1 \approx_L \sigma_2$

For the most part, this proof is the same as the one for Lemma 30 (Requirement **(T5)**) except that Lemma 37 (Requirement **(WE1)**) is used instead of Lemma 36 (Requirement **(E1)**). It uses Lemma 63 (Requirement **(EH1)**) for the TRIGGER case. We show the most interesting cases below.

Case I: $\mathcal{E}$ ends in IF-TRUE-BR
By assumption,
(I.1) $\mathcal{V} = \mathsf{TT}$
(I.2) $c = \mathsf{if}\ e\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2'$
(I.3) $G, \mathsf{TT}, \sigma_1^G, \sigma_1 \vdash e \Downarrow^{\mathsf{TT}}_{pc_1} (\mathsf{true}, l_1)$
(I.4) $l_1 \not\sqsubseteq L$
(I.5) $\alpha_1 = \mathsf{br}(\mathsf{true})$
(I.6) $c_1 = c_1'$
(I.7) $\sigma_3^G = \sigma_1^G$
(I.8) $\sigma_1' = \sigma_1$
(I.9) $E_1 = \cdot$
From (I.1) and (I.2),
(I.10) $\mathcal{D}$ must end in IF-TRUE-BR, IF-FALSE-BR, IF-TRUE, or IF-FALSE
From (I.10),
(I.11) $G, \mathsf{TT}, \sigma_2^G, \sigma_2 \vdash e \Downarrow^{\mathsf{TT}}_{pc_2} (b, l_2)$
From (1), (3), (4), (I.3), (I.11), and Lemma 37 (Requirement **(WV1)**),
(I.12) $(\mathsf{true}, l_1) \approx_L (b, l_2)$
From (I.4) and (I.12),
(I.13) $l_2 \not\sqsubseteq L$
From (I.10) and (I.13),
(I.14) $\mathcal{D}$ must end in IF-TRUE-BR, IF-FALSE-BR
From (2) and (I.5),
(I.15) $\alpha_2 = \mathsf{br}(\mathsf{true})$
From (I.14) and (I.15),
(I.16) $\mathcal{D}$ must end in IF-TRUE-BR
From (I.16),
(I.17) $c_2 = c_1'$
(I.18) $\sigma_4^G = \sigma_2^G$
(I.19) $\sigma_2' = \sigma_2$
(I.20) $E_2 = \cdot$
From (3), (4), (I.6)-(I.9) and (I.17)-(I.20),
$\sigma_3^G \approx_L \sigma_4^G$, $\sigma_1' \approx_L \sigma_2'$, $c_1 \approx_L c_2$, and $E_1 \approx_L E_2$

The proof for IF-FALSE-BR is similar to **Case I**
The proofs for IF-TRUE and IF-FALSE is similar to **Case I**. Lemma 37 is used to show that the labels on the branch condition are $\sqsubseteq L$.

□

F.5. Expression Requirements

Requirement (E1) Equivalent traces produce L-equivalent states

Lemma 36. *If* $\sigma_1^G \approx_L \sigma_2^G$ *and* $\sigma_1 \approx_L \sigma_2$ *with* $pc_1, pc_2 \sqsubseteq L$ *and* $G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e \Downarrow^i_{pc_1} v_1$, *then*
Unstructured EH storage: $G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e \Downarrow^i_{pc_2} v_2$ *with* $v_1 \simeq_L v_2$
Tree-structured EH storage: $G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e \Downarrow^i_{pc_2} v_2$ *with* $v_1 \simeq_L^{\sigma_1, \sigma_2} v_2$

Proof.
By induction on the structure of $\mathcal{E} :: G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e \Downarrow^i_{pc_1} v_1$ By assumption,
(1) $\sigma_1^G \approx_L \sigma_2^G$
(2) $\sigma_1 \approx_L \sigma_2$,
(3) $pc_1, pc_2 \sqsubseteq L$
Denote
$\mathcal{D} :: G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e \Downarrow^i_{pc_2} v_2$

Unstructured EH storage:
Case U.I: $\mathcal{E}$ ends in VAR
By assumption,
(U.I.1) $e = x$
(U.I.2) if $x \in \sigma_1^G$, then $v_1' = \mathsf{var}_{G\downarrow_g}(\sigma_1^G, pc_1, x)$
(U.I.3) otherwise, $v_1' = \mathsf{var}_{\mathcal{V}}(\sigma_1, pc_1, x)$
(U.I.4) $v_1 = \mathsf{toDst}(v_1', pc_1, i)$
From (U.I.1),
(U.I.5) The last rule applied to $\mathcal{D}$ must have been VAR
From (U.I.5),
(U.I.6) if $x \in \sigma_2^G$, then $v_2' = \mathsf{var}_{G\downarrow_g}(\sigma_2^G, pc_2, x)$
(U.I.7) otherwise, $v_2' = \mathsf{var}_{\mathcal{V}}(\sigma_2, pc_2, x)$
(U.I.8) $v_2 = \mathsf{toDst}(v_2', pc_2, i)$
Recall that in our semantics, the set of global variables is static, so (U.I.2) is true iff (U.I.6) is true and (U.I.3) is true iff (U.I.7) is true
From (1), (2), (U.I.2), (U.I.3), (U.I.6), (U.I.7) and Lemma 38.U (Requirement **(V1)**),
(U.I.9) $v_1' \simeq_L v_2'$
From (2), (U.I.9), (U.I.4), (U.I.8), and Lemma 39.U (Requirement **(V1)**),
$v_1 \simeq_L v_2$

Case U.II: $\mathcal{E}$ ends in BOP
By assumption,
(U.II.1) $e = e_1 \ \mathsf{bop} \ e_2$
(U.II.2) $\exists \mathcal{E}_1 :: G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e_1 \Downarrow^i_{pc_1} v_{1,1}$
(U.II.3) $\exists \mathcal{E}_2 :: G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e_2 \Downarrow^i_{pc_1} v_{2,1}$
(U.II.4) $v_1 = v_{1,1} \ \mathsf{bop} \ v_{2,1}$
From (U.II.1),
(U.II.5) The last rule applied to $\mathcal{D}$ must have been BOP
From (U.II.5),
(U.II.6) $\exists \mathcal{D}_1 :: G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e_1 \Downarrow^i_{pc_2} v_{1,2}$
(U.II.7) $\exists \mathcal{D}_2 :: G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e_2 \Downarrow^i_{pc_2} v_{2,2}$
(U.II.8) $v_2 = v_{1,2} \ \mathsf{bop} \ v_{2,2}$
IH on $\mathcal{E}_1, \mathcal{D}_1$ and $\mathcal{E}_2, \mathcal{D}_2$ gives
(U.II.9) $v_{1,1} \simeq_L v_{1,2}$
(U.II.10) $v_{2,1} \simeq_L v_{2,2}$
From (U.II.9), (U.II.10), (U.II.4), and (U.II.8),
$v_1 \simeq_L v_2$

Case U.III: $\mathcal{E}$ ends in EHAPI
By assumption,
(U.III.1) $e = \mathsf{ehAPI}(id, e_1, \cdots, e_n)$
(U.III.2) $\forall i \in [1, n], \exists \mathcal{E}_i :: G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e_i \Downarrow^{G\downarrow_{EH}}_{pc_1} v_{i,1}$
(U.III.3) $v_1' = \mathsf{ehAPI}_{G\downarrow_{EH}}(\sigma_1^G, pc_1, id, v_{1,1}, \cdots, v_{n,1})$
(U.III.4) $v_1 = \mathsf{toDst}(v_1', pc_1, i)$
From (U.III.1),
(U.III.5) The last rule applied to $\mathcal{D}$ must have been EHAPI

From (U.III.5),
(U.III.6) $\forall i \in [1,n], \exists \mathcal{D}_i :: G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e_i \Downarrow_{pc_2}^{\mathcal{I}\downarrow_{EH}} v_{i,2}$
(U.III.7) $v_2' = \mathsf{ehAPI}_{G\downarrow_{EH}}(\sigma_2^G, pc_2, id, v_{1,2}, \cdots, v_{n,2})$
(U.III.8) $v_2 = \mathsf{toDst}(v_2', pc_2, i)$
IH on $\mathcal{E}_i, \mathcal{D}_i, \forall i \in [1,n]$ gives
(U.III.9) $\forall i \in [1,n], v_{i,1} \simeq_L v_{i,2}$
From (1), (2), (U.III.9), (U.III.3), (U.III.7), and Lemma 46.U (Requirement **(EH1)**),
$v_1 \simeq_L v_2$

Tree-structured EH storage:
Case T.I: $\mathcal{E}$ ends in VAR
By assumption,
(T.I.1) $e = x$
(T.I.2) if $x \in \sigma_1^G$, then $v_1' = \mathsf{var}_{G\downarrow_g}(\sigma_1^G, pc_1, x)$
(T.I.3) otherwise, $v_1' = \mathsf{var}_{\mathcal{V}}(\sigma_1, pc_1, x)$
(T.I.4) $v_1 = \mathsf{toDst}(v_1', pc_1, i)$
From (T.I.1),
(T.I.5) The last rule applied to $\mathcal{D}$ must have been VAR
From (T.I.5),
(T.I.6) if $x \in \sigma_2^G$, then $v_2' = \mathsf{var}_{G\downarrow_g}(\sigma_2^G, pc_2, x)$
(T.I.7) otherwise, $v_2' = \mathsf{var}_{\mathcal{V}}(\sigma_2, pc_2, x)$
(T.I.8) $v_2 = \mathsf{toDst}(v_2', pc_2, i)$
Recall that in our semantics, the set of global variables is static, so (T.I.2) is true iff (T.I.6) is true and (T.I.3) is true iff (T.I.7) is true
From (1), (2), (T.I.2), (T.I.3), (T.I.6), (T.I.7) and Lemma 38.T (Requirement **(V1)**),
(T.I.9) $v_1' \simeq_L^{\sigma_1,\sigma_2} v_2'$
From (2), (T.I.9), (T.I.4), (T.I.8), and Lemma 39.T (Requirement **(V1)**),
$v_1 \simeq_L^{\sigma_1,\sigma_2} v_2$

Case U.II: $\mathcal{E}$ ends in BOP
By assumption,
(T.II.1) $e = e_1 \ \mathsf{bop} \ e_2$
(T.II.2) $\exists \mathcal{E}_1 :: G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e_1 \Downarrow_{pc_1}^{i} v_{1,1}$
(T.II.3) $\exists \mathcal{E}_2 :: G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e_2 \Downarrow_{pc_1}^{i} v_{2,1}$
(T.II.4) $v_1 = v_{1,1} \ \mathsf{bop} \ v_{2,1}$
From (T.II.1),
(T.II.5) The last rule applied to $\mathcal{D}$ must have been BOP
From (T.II.5),
(T.II.6) $\exists \mathcal{D}_1 :: G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e_1 \Downarrow_{pc_2}^{i} v_{1,2}$
(T.II.7) $\exists \mathcal{D}_2 :: G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e_2 \Downarrow_{pc_2}^{i} v_{2,2}$
(T.II.8) $v_2 = v_{1,2} \ \mathsf{bop} \ v_{2,2}$
IH on $\mathcal{E}_1, \mathcal{D}_1$ and $\mathcal{E}_2, \mathcal{D}_2$ gives
(T.II.9) $v_{1,1} \simeq_L^{\sigma_1,\sigma_2} v_{1,2}$
(T.II.10) $v_{2,1} \simeq_L^{\sigma_1,\sigma_2} v_{2,2}$
From (T.II.9), (T.II.10), (T.II.4), and (T.II.8),
$v_1 \simeq_L^{\sigma_1,\sigma_2} v_2$

Case T.III: $\mathcal{E}$ ends in EHAPI
By assumption,
(T.III.1) $e = \mathsf{ehAPI}(id, e_1, \cdots, e_n)$
(T.III.2) $\forall i \in [1,n], \exists \mathcal{E}_i :: G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e_i \Downarrow_{pc_1}^{G\downarrow_{EH}} v_{i,1}$
(T.III.3) $v_1' = \mathsf{ehAPI}_{G\downarrow_{EH}}(\sigma_1^G, pc_1, id, v_{1,1}, \cdots, v_{n,1})$
(T.III.4) $v_1 = \mathsf{toDst}(v_1', pc_1, i)$
From (T.III.1),
(T.III.5) The last rule applied to $\mathcal{D}$ must have been EHAPI
From (T.III.5),
(T.III.6) $\forall i \in [1,n], \exists \mathcal{D}_i :: G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e_i \Downarrow_{pc_2}^{\mathcal{I}\downarrow_{EH}} v_{i,2}$
(T.III.7) $v_2' = \mathsf{ehAPI}_{G\downarrow_{EH}}(\sigma_2^G, pc_2, id, v_{1,2}, \cdots, v_{n,2})$
(T.III.8) $v_2 = \mathsf{toDst}(v_2', pc_2, i)$
IH on $\mathcal{E}_i, \mathcal{D}_i, \forall i \in [1,n]$ gives
(T.III.9) $\forall i \in [1,n], v_{i,1} \simeq_L^{\sigma_1,\sigma_2} v_{i,2}$

From (1), (2), (T.III.9), (T.III.3), (T.III.7), and Lemma 46.T (Requirement **(EH1)**),
$v_1 \simeq_L^{\sigma_1,\sigma_2} v_2$

□

Requirement (WE1) Equivalent traces produce L-equivalent states (Weak Secrecy)

Lemma 37. *If* $\sigma_1 \approx_L \sigma_2$*, with* $pc_1, pc_2 \sqsubseteq L$ *and* $G, \mathcal{V}, \sigma_1^G, \sigma_1 \vdash e \Downarrow^i_{pc_1} v_1$*, then* $G, \mathcal{V}, \sigma_2^G, \sigma_2 \vdash e \Downarrow^i_{pc_2} v_2$ *with* $v_1 \approx_L v_2$

Proof (sketch): The proof is similar to the one for Lemma 36 (Requirement **(E1)**) except that it uses Lemma 40 (Requirement **(WV1)**) instead of Lemma 38 (Requirement **(V1)**) and Lemma 64 (Requirement **(WEH1)**) instead of Lemma 46 (Requirement **(EH1)**). □

F.6. Variable Storage Requirements

Requirement (V1) L lookups are equivalent

Lemma 38. *If* $\sigma_1 \approx_L \sigma_2$ *and* $pc_1, pc_2 \sqsubseteq L$*, then for*

Unstructured EH storage:

- $\mathsf{var}_{\mathcal{V}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathcal{V}}(\sigma_2, pc_2, x) = v_2$ *then* $v_1 \simeq_L v_2$
- $\mathsf{var}_{\mathcal{G}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathcal{G}}(\sigma_2, pc_1, x) = v_2$ *then* $v_1 \simeq_L v_2$

Tree-structured EH storage:

- $\mathsf{var}_{\mathcal{V}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathcal{V}}(\sigma_2, pc_2, x) = v_2$ *then* $v_1 \simeq_L^{\sigma_1,\sigma_2} v_2$
- $\mathsf{var}_{\mathcal{G}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathcal{G}}(\sigma_2, pc_1, x) = v_2$ *then* $v_1 \simeq_L^{\sigma_1,\sigma_2} v_2$

Proof.
Only cases for $\mathcal{V} \neq \mathsf{TT}$ and $\mathcal{G} \neq \mathsf{TS}$ are considered. The other cases are proven in the weak secrecy version: Lemma 40 (Requirement **(WV1)**).
By induction on the structure of $\mathcal{E} :: \mathsf{var}_i(\sigma_1, pc_1, x) = v_1$ and $\mathcal{D} :: \mathsf{var}_i(\sigma_2, pc_2, x) = v_2$
By assumption,
(1) $\sigma_1 \approx_L \sigma_2$
(2) $pc_1, pc_2 \sqsubseteq L$

Case I: $\mathcal{V} = \mathsf{SME}$
From (1) and (2),
(I.1) $\sigma_1 = \sigma_2$

Case i: $\mathcal{E}$ ends in VAR
By assumption,
(i.1) $x \in \sigma_1$
(i.2) $\sigma_1(x) = v_1^{\mathsf{std}}$
From (I.1) and (i.1),
(i.3) $\mathcal{D}$ ends in VAR
From (i.3),
(i.4) $\sigma_2(x) = v_2^{\mathsf{std}}$
From (I.1), (i.2), and (i.4)
The desired conclusion holds

Case ii: $\mathcal{D}$ ends in VAR
The proof is similar to **Case i**

Case iii: $\mathcal{E}$ ends in VAR-DV
By assumption and from (2),
(iii.1) $x \notin \sigma_1$
(iii.2) $v_1 = \mathsf{dv}$
From (I.1) and (iii.1),
(iii.3) $\mathcal{D}$ ends in VAR-DV
From (iii.3),
(iii.4) $v_2 = \mathsf{dv}$
From (iii.2) and (iii.4),
The desired conclusion holds

Case iv: $\mathcal{D}$ ends in VAR-DV
The proof is similar to **Case iii**

Case II: $\mathcal{V} = \mathsf{MF}$
Case i: $\mathcal{E}$ ends in VAR
By assumption,
(i.1) $pc_1 = L$
(i.2) $v_1 = \mathsf{getFacetV}(\sigma_1(x), L)$
From (i.2) and the definition of getFacetV,
(i.3) If $\sigma_1(x) \downarrow_L = \cdot$, then $v_1 = \mathsf{dv}$
(i.4) otherwise, $v_1 = \sigma(x) \downarrow_L$

Subcase a: $\mathcal{D}$ ends in VAR
By assumption,

(a.1) $v_2 = \mathsf{getFacetV}(\sigma_2(x), L)$

From (2),

(a.2) If $\sigma_1(x) \downarrow_L = \cdot$, then $\sigma_2(x) \downarrow_L = \cdot$

(a.3) If $\sigma_1(x) \downarrow_L \neq \cdot$, then $\sigma_2(x) \downarrow_L = \sigma_1(x) \downarrow_L \neq \cdot$

If (a.2) is true, then from (i.3), (a.1), and the definition of getFacetV,

(a.4) $v_1 = v_2 = \mathsf{dv}$

If (a.3) is true, then from (i.4), (a.1), and the definition of getFacetV,

(a.5) $v_1 = v_2 = \sigma_1(x) \downarrow_L = \sigma_2(x) \downarrow_L$

From (a.4), and (a.5),

the desired conclusion holds

Subcase b: $\mathcal{D}$ ends in VAR-F

Unstructured EH storage:

By assumption,

(U.b.1) $pc_2 = \cdot$

(U.b.2) $v_2 = \mathsf{setFacetV}(v_H, v_L)$ for

(U.b.3) $v_L = \mathsf{var}_{\mathsf{MF}}(\sigma_2, L, x)$

IH on (U.b.2) gives

(U.b.4) $v_1 \simeq_L v_L$

From (U.b.1) and the definition of setFacetV,

(U.b.5) $v_2 \simeq_L v_L$

From (U.b.4) and (U.b.5),

the desired conclusion holds

Tree-structured EH storage:

By assumption,

(T.b.1) $pc_2 = \cdot$

(T.b.2) $v_2 = \mathsf{setFacetV}(v_H, v_L)$ for

(T.b.3) $v_L = \mathsf{var}_{\mathsf{MF}}(\sigma_2, L, x)$

IH on (T.b.2) gives

(T.b.4) $v_1 \simeq_L^{\sigma_1, \sigma_2} v_L$

From (T.b.1) and the definition of setFacetV,

(T.b.5) $v_2 \simeq_L^{\sigma_2, \sigma_2} v_L$

From (T.b.4) and (T.b.5), and the definition of $\simeq_L^{\sigma_A, \sigma_B}$

(T.b.6) $v_1 \downarrow_L^{\sigma_1} = v_L \downarrow_L^{\sigma_2} = v_2 \downarrow_L^{\sigma_2}$

From (T.b.6),

(T.b.7) $v_1 \approx_L^{\sigma_1, \sigma_2} v_2$

From (T.b.7) and from $\mathcal{V} = \mathsf{MF}$,

the desired conclusion holds

Subcase c: $\mathcal{D}$ ends in VAR-DV

By assumption,

(c.1) $v_2 = \mathsf{dv}$

From (i.3) and (c.1),

the desired conclusion holds

Case ii: $\mathcal{D}$ ends in VAR

The proof for this case is similar to **Case i**

Case iii: $\mathcal{E}$ ends in VAR-F

By assumption,

(iii.1) $pc_1 = \cdot$

(iii.2) $v_{L,1} = \mathsf{var}_{\mathsf{MF}}(\sigma_1, L, x)$

(iii.3) $v_1 = \mathsf{setFacetV}(v_{H,1}, v_{L,1})$

Subcase a: $\mathcal{D}$ ends in VAR-F

By assumption,

(a.1) $pc_2 = \cdot$

(a.2) $v_2 = \mathsf{setFacetV}(v_{H,2}, v_{L,2})$ for

(a.3) $v_{L,2} = \mathsf{var}_{\mathsf{MF}}(\sigma_2, L, x)$

IH on (a.2) and (iii.2) gives

(a.4) $v_{L,1} \approx_L v_{L,2}$ (for the Unstructured EH storage)

(a.5) $v_{L,1} \approx_L^{\sigma_1, \sigma_2} v_{L,2}$ (for the Tree-structured EH storage)

From (a.4), (a.5), (iii.3), (a.2), and the definition of setFacetV,

the desired conclusion holds

Subcase b: $\mathcal{D}$ ends in VAR-DV
By assumption,
(b.1) $x \notin \sigma_2$
(b.2) $v_2 = \mathsf{dv}$
IH on (iii.2) gives
(b.3) $v_L \simeq_L v_2$ (for the Unstructured EH storage)
(b.4) $v_L \simeq_L^{\sigma_1,\sigma_2} v_2$ (for the Tree-structured EH storage)
From (b.2)-(b.4) and the definition of $\mathsf{setFacetV}$,
The desired conclusion holds

Case iv: $\mathcal{D}$ ends in VAR-F
The proof is similar to **Case iii**

Case v: $\mathcal{E}$ and $\mathcal{D}$ end in VAR-DV
By assumption,
(v.1) $v_1 = \mathsf{dv}$
(v.2) $v_2 = \mathsf{dv}$
From (v.1) and (v.2),
the desired conclusion holds

Case III: $\mathcal{G} = \mathsf{SMS}$
The proofs for this case are similar to **Case I**

Case IV: $\mathcal{G} = \mathsf{FS}$
The proofs for this case are similar to **Case II**

□

Lemma 39.
Unstructured EH storage: *If* $v_1 \simeq_L v_2$, *with* $pc_1, pc_2 \sqsubseteq L$ *and* $\mathsf{toDst}(v_1, pc_1, i) = v_1'$, *then* $\mathsf{toDst}(v_2, pc_2, i) = v_2'$ *with* $v_1' \simeq_L v_2'$
Tree-structured EH storage: *If* $v_1 \simeq_L^{\sigma_1,\sigma_2} v_2$, *with* $pc_1, pc_2 \sqsubseteq L$ *and* $\mathsf{toDst}(v_1, pc_1, i) = v_1'$, *then* $\mathsf{toDst}(v_2, pc_2, i) = v_2'$ *with* $v_1' \simeq_L^{\sigma_1,\sigma_2} v_2'$

Proof. This proof is by straightforward case analysis on the structure of v_1 and v_2. □

Requirement (WV1) L lookups are equivalent (Weak Secrecy)

Lemma 40. *If* $\sigma_1 \approx_L \sigma_2$ *and* $pc_1, pc_2 \sqsubseteq L$, *then for*
Unstructured EH storage:
- $\mathsf{var}_{\mathcal{V}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathcal{V}}(\sigma_2, pc_2, x) = v_2$ *then* $v_1 \approx_L v_2$
- $\mathsf{var}_{\mathcal{G}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathcal{G}}(\sigma_2, pc_1, x) = v_2$ *then* $v_1 \approx_L v_2$

Tree-structured EH storage:
- $\mathsf{var}_{\mathcal{V}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathcal{V}}(\sigma_2, pc_2, x) = v_2$ *then* $v_1 \approx_L^{\sigma_1,\sigma_2} v_2$
- $\mathsf{var}_{\mathcal{G}}(\sigma_1, pc_1, x) = v_1$ *and* $\mathsf{var}_{\mathcal{G}}(\sigma_2, pc_1, x) = v_2$ *then* $v_1 \approx_L^{\sigma_1,\sigma_2} v_2$

Proof.
Only the cases for TT and TS are shown, since the other cases follow from Lemma 38 (Requirement **(V1)**).
By induction on the structure of $\mathcal{E} :: \mathsf{var}_i(\sigma_1, pc_1, x) = v_1$ and $\mathcal{D} :: \mathsf{var}_i(\sigma_2, pc_2, x) = v_2$
By assumption,
(1) $\sigma_1 \approx_L \sigma_2$
(2) $pc_1, pc_2 \sqsubseteq L$

Case I: $\mathcal{V} = \mathsf{TT}$

Case i: $\mathcal{E}$ ends in VAR
By assumption,
(i.1) $v_1 = \sigma_1(x)$

Subcase a: $\mathcal{E}$ ends in VAR
By assumption,
(a.1) $v_2 = \sigma_2(x)$
From (1), (i.1), and (a.1),
the desired conclusion holds

Subcase b: $\mathcal{E}$ ends in VAR-DV
By assumption,
(b.1) $x \notin \sigma_2$
(b.2) $v_2 = (\mathsf{dv}, H)$
From (2), (b.1), and (i.1),
(b.3) $v_1 = (v, H)$
From (b.2) and (b.3),
the desired conclusion holds

Case ii: $\mathcal{D}$ ends in VAR
The proof for this case is similar to **Case i**

Case iii: $\mathcal{E}$ and $\mathcal{D}$ end in VAR-DV
By assumption,
(iii.1) $v_1 = (\mathsf{dv}, H)$
(iii.2) $v_2 = (\mathsf{dv}, H)$
From (iii.1) and (iii.2),
the desired conclusion holds

Case II: $\mathcal{G} = \mathsf{TS}$
The proofs for this case are similar to **Case I**

□

Requirement (V2) H assignments are unobservable

Lemma 41. $\mathsf{assign}_{\mathcal{G}}(\sigma, H, x, v) \approx_L \sigma$

Proof.
Only cases for $\mathcal{G} \neq \mathsf{TS}$ are considered. The other cases are proven in the weak secrecy version: Lemma 42 (Requirement **(WV2)**)
By case analysis on $\mathcal{G}$
Denote $\mathsf{assign}_i(\sigma, H, x, v) = \sigma'$
Want to show: $\sigma \approx_L \sigma'$

Case I: $\mathcal{V} = \mathsf{SMS}$
By assumption,
(I.1) $\sigma = \sigma_H, \sigma_L$
From the definition of $\mathsf{getStore}_{\mathsf{SMS}}$,
(I.2) $\mathsf{getStore}(\sigma, H) = \sigma_H$

Case i: $x \in \sigma$
By assumption,
(i.1) ASSIGN was applied
From (i.1),
(i.2) $\sigma'_H = \sigma_H[x \mapsto v]$
(i.3) $\sigma' = \mathsf{setStoreVar}_{\mathsf{SMS}}(\sigma, H, \sigma'_H)$
From (i.3) and the definition of $\mathsf{setStoreVar}_{\mathsf{SMS}}$,
$\sigma \approx_L \sigma'$

Case ii: $x \notin \sigma$
By assumption,
(ii.1) ASSIGN-S was applied
From (ii.1),
(ii.2) $\sigma' = \sigma$
From (ii.2),
$\sigma \approx_L \sigma'$

Case II: $\mathcal{V} = \mathsf{FS}$
Case i: $x \in \sigma$
By assumption,
(i.1) ASSIGN-H was applied
From (i.1),
(i.2) $v_L = \mathsf{var}_{\mathsf{FS}}(\sigma, L, x)$
(i.3) $v' = \mathsf{setFacetV}(\mathsf{getFacet}(v, H), v_L)$
(i.4) $\sigma' = \sigma[x \mapsto v']$
By assumption and from (i.2),
(i.5) $v_L = \mathsf{getFacetV}(\sigma(x), L)$
From (i.5) and the definition of getFacet,
(i.6) $v_L \approx_L \sigma(x)$
From (i.6), (i.3), and the definition of setFacetV,
(i.7) $v' \approx_L \sigma(x)$
From (i.4) and (i.7),
$\sigma \approx_L \sigma'$

Case ii: $x \notin \sigma$
By assumption,
(i.1) ASSIGN-S was applied
From (i.1),
(i.2) $\sigma' = \sigma$
From (i.2),
$\sigma \approx_L \sigma'$

□

Requirement (WV2) H assignments are unobservable (Weak Secrecy)

Lemma 42. *If* $\mathsf{assignW}_{\mathcal{G}}(\sigma, H, x, v) = (\sigma', \bullet)$, *then* $\sigma \approx_L \sigma'$

Proof.

Only cases for $\mathcal{G} = \mathsf{TS}$ are shown, since the other cases follow from Lemma 41 (Requirement **(V2)**).
By assumption,
(1) $\mathsf{assignW}_{\mathcal{G}}(\sigma, H, x, v) = (\sigma', \alpha)$ with $\alpha = \bullet$

Case I: $x \in \sigma$
By assumption and from (1),
(I.1) the last rule applied was ASSIGN
From (I.1),
(I.2) $v = (v', l)$ and
(I.3) $l \sqcup H \sqsubseteq \mathsf{labOf}(\sigma(x), H)$
From (I.3) and our security lattice,
(I.4) $\mathsf{labOf}(\sigma(x), H) = H$
From (I.1) and (I.2),
(I.5) $\sigma' = \sigma[x \mapsto (v', l \sqcup H)]$
From (I.4) and (I.5),
$\sigma \approx_L \sigma'$

Case II: $x \notin \sigma$
By assumption and from (1),
(II.1) the last rule applied was ASSIGN-S
From (II.1),
(II.2) $\sigma' = \sigma$
From (II.2),
$\sigma \approx_L \sigma'$

□

Lemma 43. *If* $\sigma_1 \approx_L \sigma_2$ *and* $\mathsf{assignW}_{\mathcal{G}}(\sigma_1, H, x, v) = (\sigma_1', \mathsf{gw}(x))$, *and* $\mathsf{assignW}_{\mathcal{G}}(\sigma_2, H, x, v) = (\sigma_2', \mathsf{gw}(x))$ *then* $\sigma_1' \approx_L \sigma_2'$

Proof.
Denote $\mathcal{D} :: \mathsf{assignW}_{\mathcal{G}}(\sigma_1, H, x, (v_1, l_1)) = (\sigma_1', \alpha_1)$
$\mathcal{E} :: \mathsf{assignW}_{\mathcal{G}}(\sigma_2, H, x, (v_2, l_2)) = (\sigma_2', \alpha_2)$

By assumption,
(1) $\alpha_1 = \alpha_2 = \mathsf{gw}(x)$
(2) $\sigma_1 \approx_L \sigma_2$
From (1) and since only the TS semantics can produce gw(_),
(3) $\mathcal{D}$ and $\mathcal{E}$ end in ASSIGN-GW
From (3),
(4) $\sigma_1' = \sigma_1[x \mapsto (v_1, l_1 \sqcup H)]$
(5) $\sigma_2' = \sigma_2[x \mapsto (v_2, l_2 \sqcup H)]$
From our security lattice,
(6) $l_1 \sqcup H = H$ and $l_2 \sqcup H = H$
From (2) and (4)-(6)
$\sigma_1' \approx_L \sigma_2'$

□

Requirement (V3) L assignments are equivalent

Lemma 44. *If* $\sigma_1 \approx_L \sigma_2$ *and* $pc_1, pc_2 \in \{L, \cdot\}$*, with*
Unstructured EH storage: $v_1 \approx_L v_2$*, then* $\mathsf{assign}_i(\sigma_1, pc_1, x, v_1) \approx_L \mathsf{assign}_i(\sigma_2, pc_2, x, v_2)$
Tree-structured EH storage: $v_1 \approx_L^{\sigma_1,\sigma_2} v_2$*, then* $\mathsf{assign}_i(\sigma_1, pc_1, x, v_1) \approx_L \mathsf{assign}_i(\sigma_2, pc_2, x, v_2)$

Proof.
By case analysis on i
By assumption,
(1) $\sigma_1 \approx_L \sigma_2$
(2) $pc_1, pc_2 \in \{L, \cdot\}$
(3) $v_1 \approx_L v_2$ (for the Unstructured EH storage)
(4) $v_1 \approx_L^{\sigma_1,\sigma_2} v_2$ (for the Tree-structured EH storage)
Denote
$\mathcal{D} :: \mathsf{assign}_i(\sigma_1, pc_1, x, v_1) = \sigma_1'$
$\mathcal{E} :: \mathsf{assign}_i(\sigma_2, pc_2, x, v_2) = \sigma_2'$
Want to show:
$\sigma_1' \approx_L \sigma_2'$

Case I: $\mathcal{V} = \mathsf{SME}$
By assumption and from (1), (2), and $\approx_L$ for SME stores,
(I.1) $\sigma_1 = \sigma_2$
(I.2) $\sigma_1' = \sigma_1[x \mapsto v_1]$
(I.3) $\sigma_2' = \sigma_2[x \mapsto v_2]$
From (I.1)-(I.3) and (3) (for the Unstructured EH storage), and (4) (for the Tree-Structured EH storage),
(I.4) $\sigma_1' = \sigma_2'$
From (I.4) and $\approx_L$ for SME stores,
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{V} = \mathsf{MF}$
Subcase i: $pc_1 = \cdot$
By assumption,
(i.1) $\mathcal{D}$ ends in MF-ASSIGN
From (i.1),
(i.2) $\sigma_1' = \sigma_1[x \mapsto v_1]$

Subsubcase a: $pc_2 = \cdot$
By assumption,
(a.1) $\mathcal{E}$ ends in MF-ASSIGN
From (a.1),
(a.2) $\sigma_2' = \sigma_2[x \mapsto v_2]$
From (1), (3) (for the Unstructured EH storage) and (4) (for the Tree-structured EH storage), (i.2), and (a.2),
$\sigma_1' \approx_L \sigma_2'$

Subsubcase b: $pc_2 = L$
Unstructured EH storage:
By assumption,
(U.b.1) $\mathcal{E}$ ends in MF-ASSIGN-L
From (U.b.1),
(U.b.2) $v_L = \mathsf{getFacet}(v_2, L)$
(U.b.3) $v_2' = \mathsf{setFacetV}(\_, v_L)$
(U.b.4) $\sigma_2' = \sigma_2[x \mapsto v_2']$
From (3) and (U.b.2) and the definition of getFacet,
(U.b.5) $v_L \approx_L v_1$
From (U.b.3) and (U.b.5) and the definition of setFacet,
(U.b.6) $v_2' \approx_L v_1$
From (1), (U.b.6), (i.2), and (U.b.4),
$\sigma_1' \approx_L \sigma_2'$

Tree-structured EH storage:
By assumption,
(T.b.1) $\mathcal{E}$ ends in MF-ASSIGN-L
From (T.b.1),
(T.b.2) $v_L = \mathsf{getFacet}(v_2, L)$

(T.b.3) $v_2' = \mathsf{setFacetV}(\_, v_L)$
(T.b.4) $\sigma_2' = \sigma_2[x \mapsto v_2']$
From (4) and (T.b.2) and the definition of getFacet,
(T.b.5) $v_1 \approx_L^{\sigma_1,\sigma_2} v_L$
From (T.b.3) and (T.b.5) and the definition of setFacet,
(T.b.6) $v_1 \approx_L^{\sigma_1,\sigma_2} v_2'$
From (1), (T.b.6), (i.2), and (T.b.4),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $pc_1 = L$
By assumption,
(ii.1) $\mathcal{E}$ ends in MF-ASSIGN-L
From (ii.1),
(ii.2) $v_{L,1} = \mathsf{getFacet}(v_1, L)$
(ii.3) $v_1' = \mathsf{setFacetV}(\_, v_{L,1})$
(ii.4) $\sigma_1' = \sigma_1[x \mapsto v_1']$

Subsubcase a: $pc_2 = \cdot$
The proof is similar to **Subsubcase i.b**.

Subsubcase b: $pc_2 = L$
Unstructured EH storage:
By assumption,
(U.b.1) $\mathcal{E}$ ends in MF-ASSIGN-L
From (U.b.1),
(U.b.2) $v_{L,2} = \mathsf{getFacet}(v_2, L)$
(U.b.3) $v_2' = \mathsf{setFacetV}(\_, v_{L,2})$
(U.b.4) $\sigma_2' = \sigma_2[x \mapsto v_2']$
From (3), (ii.2), (U.b.2), and the definition of getFacet,
(U.b.5) $v_{L,1} \approx_L v_{L,2}$
From (U.b.5), (ii.3), (U.b.3), and the definition of setFacet,
(U.b.6) $v_1' \approx_L v_2'$
From (1), (U.b.6), (ii.4), and (U.b.4),
$\sigma_1' \approx_L \sigma_2'$

Tree-structured EH storage:
By assumption,
(T.b.1) $\mathcal{E}$ ends in MF-ASSIGN-L
From (T.b.1),
(T.b.2) $v_{L,2} = \mathsf{getFacet}(v_2, L)$
(T.b.3) $v_2' = \mathsf{setFacetV}(\_, v_{L,2})$
(T.b.4) $\sigma_2' = \sigma_2[x \mapsto v_2']$
From (4), (ii.2), (T.b.2), and the definition of getFacet,
(T.b.5) $v_{L,1} \approx_L^{\sigma_1,\sigma_2} v_{L,2}$
From (T.b.5), (ii.3), (T.b.3), and the definition of setFacet,
(T.b.6) $v_1' \approx_L^{\sigma_1,\sigma_2} v_2'$
From (1), (T.b.6), (ii.4), and (T.b.4),
$\sigma_1' \approx_L \sigma_2'$

Case III: $\mathcal{V} = \mathsf{TT}$
Unstructured EH storage:
By assumption,
(U.III.1) $v_1 = (v_1', l_1)$
(U.III.2) $v_2 = (v_2', l_2)$
(U.III.3) $\sigma_1' = \sigma_1[x \mapsto (v_1', l_1 \sqcup pc_1)]$
(U.III.4) $\sigma_2' = \sigma_2[x \mapsto (v_2', l_2 \sqcup pc_2)]$
From (3), (U.III.1), and (U.III.2),
(U.III.5) $l_1, l_2 \sqsubseteq L$ with $v_1' = v_2'$ or
(U.III.6) $l_1, l_2 \not\sqsubseteq L$
From (U.III.5), (U.III.6), and (2),
(U.III.7) $(v_1', l_1 \sqcup pc_1) \approx_L (v_2', l_2 \sqcup pc_2)$
From (1), (U.III.3), (U.III.4), and (U.III.7),
$\sigma_1' \approx_L \sigma_2'$

Tree-structured EH storage:
By assumption,
(T.III.1) $v_1 = (v_1', l_1)$
(T.III.2) $v_2 = (v_2', l_2)$
(T.III.3) $\sigma_1' = \sigma_1[x \mapsto (v_1', l_1 \sqcup pc_1)]$
(T.III.4) $\sigma_2' = \sigma_2[x \mapsto (v_2', l_2 \sqcup pc_2)]$
From (4), (T.III.1), and (T.III.2),
(T.III.5) $l_1, l_2 \sqsubseteq L$ with $v_1' = v_2'$ or
(T.III.6) $l_1, l_2 \not\sqsubseteq L$
From (T.III.5), (T.III.6), and (2),
(T.III.7) $(v_1', l_1 \sqcup pc_1) \approx_L^{\sigma_1, \sigma_2} (v_2', l_2 \sqcup pc_2)$
From (1), (T.III.3), (T.III.4), and (T.III.7),
$\sigma_1' \approx_L \sigma_2'$

Case IV: $\mathcal{G} = \mathsf{SMS}$
The proof for this case is similar to **Case I** except that there is an extra case for $x \notin \sigma$, which produces the same store, so the resulting stores are equivalent (from (1)).

Case V: $\mathcal{G} = \mathsf{FS}$
The proof for this case is similar to **Case II** except that there is an extra case for $x \notin \sigma$, which produces the same store, so the resulting stores are equivalent (from (1)).

Case VI: $\mathcal{G} = \mathsf{TS}$
The proof for this case is similar to **Case III** except that there is an extra case for $x \notin \sigma$, which produces the same store, so the resulting stores are equivalent (from (1)).

□

Requirement (WV3) L assignments are equivalent (Weak Secrecy)

Lemma 45. *If* $\sigma_1 \approx_L \sigma_2$ *and* $pc_1, pc_2 \in \{L, \cdot\}$*, with* $v_1 \approx_L v_2$*, then for* $(\sigma_1', \alpha_1) = \mathsf{assignW}_i(\sigma_1, pc_1, x, v_1)$ *and* $(\sigma_2', \alpha_2) = \mathsf{assignW}_i(\sigma_2, pc_2, x, v_2)$*,* $\sigma_1' \approx_L \sigma_2'$ *and* $\alpha_1 = \alpha_2$

Proof.
Only the cases for $i \in \{\mathsf{TT}, \mathsf{TS}\}$ are shown, since the other cases always produce $\alpha_1 = \alpha_2 = \bullet$, so their proofs follow from Lemma 44 (Requirement **(V3)**). We show the proofs for $\mathcal{I} = \mathsf{TS}$ since the cases for $\mathcal{I} = \mathsf{TT}$ are similar.
By case analysis on $\mathcal{D} :: (\sigma_1', \alpha_1) = \mathsf{assignW}_i(\sigma_1, pc_1, x, (v_1, l_1))$
By assumption,
(1) $\sigma_1 \approx_L \sigma_2$
(2) $pc_1, pc_2 \in \{L, \cdot\}$
(3) $(v_1, l_1) \approx_L (v_2, l_2)$
Denote $\mathcal{E} :: (\sigma_2', \alpha_2) = \mathsf{assignW}_i(\sigma_2, pc_2, x, (v_2, l_2))$

Case I: $\mathcal{D}$ ends in TS-ASSIGN
By assumption,
(I.1) $l_1 \sqcup H \sqsubseteq \mathsf{labOf}(\sigma_1(x), H)$
(I.2) $\sigma_1' = \sigma_1[x \mapsto (v_1, l_1 \sqcup H)]$
(I.3) $\alpha_1 = \bullet$
From (I.1) and our security lattice,
(I.4) $\mathsf{labOf}(\sigma_1(x), H) = H$
From (1) and (I.4),
(I.5) $x \in \sigma_2$ and $\mathsf{labOf}(\sigma_2(x), H) = H$ or
(I.6) $x \notin \sigma_2$

Subcase i: (I.5) is true
From (I.5) and our security lattice,
(i.1) $l_2 \sqcup H \sqsubseteq \mathsf{labOf}(\sigma_2(x), H)$
From (i.1),
(i.2) $\mathcal{E}$ ends in TS-ASSIGN
From (i.2),
(i.3) $\sigma_2' = \sigma_2[x \mapsto (v_2, l_2 \sqcup H)]$
(i.4) $\alpha_2 = \bullet$
From (I.3) and (i.4),
$\alpha_1 = \alpha_2$

From (1), (I.2) and (i.3),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: (I.6) is true
From (I.6),
(ii.1) $\mathcal{E}$ ends in TS-ASSIGN-S
From (ii.1),
(ii.2) $\sigma_2' = \sigma_2$
(ii.3) $\alpha_2 = \bullet$
From (I.3) and (ii.3),
$\alpha_1 = \alpha_2$
From (I.4) and (I.2),
(ii.4) $\sigma_1' \approx_L \sigma_1$
From (1), (ii.2),a nd (ii.4),
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{D}$ ends in TS-ASSIGN-S
The proof for this case is similar to the one for **Case I**.

Case III: $\mathcal{D}$ ends in TS-ASSIGN-GW
By assumption,
(III.1) $l_1 \sqcup H \not\sqsubseteq \mathsf{labOf}(\sigma_1(x), H)$
(III.2) $\sigma_1' = \sigma_1[x \mapsto (v_1, l_1 \sqcup H)]$
(III.3) $\alpha_1 = \mathsf{gw}(x)$
From (III.1) and our security lattice,
(III.4) $\mathsf{labOf}(\sigma_1(x), H) = L$
From (1) and (III.4),
(III.5) $x \in \sigma_2$ and $\mathsf{labOf}(\sigma_2(x), H) = L$
From (III.5),
(III.6) $\mathcal{E}$ ends in TS-ASSIGN-GW
From (III.6),
(III.7) $\sigma_2' = \sigma_2[x \mapsto (v_2, l_2 \sqcup H)]$
(III.8) $\alpha_2 = \mathsf{gw}(x)$
From (III.3) and (III.8),
$\alpha_1 = \alpha_2$
From (1), (III.2), and (III.7),
$\sigma_1' \approx_L \sigma_2'$

□

F.7. Event Handler Storage Requirements

Requirement (EH1) L lookups are equivalent

Lemma 46. *If* $\sigma_1 \approx_L \sigma_2$ *and* $pc_1, pc_2 \sqsubseteq L$ *and*
Unstructured EH storage: $\forall i \in [1,n], v_{i,1} \simeq_L v_{i,2}$ *with* $v_1 = \mathsf{ehAPIe}(\mathcal{G}, \sigma_1, pc_1, id, v_{1,1}, \cdots, v_{n,1})$ *and* $v_2 = \mathsf{ehAPIe}(\mathcal{G}, \sigma_2, pc_2, id, v_{1,2}, \cdots, v_{n,2})$, *then* $v_1 \simeq_L v_2$
Tree-structured EH storage: $\forall i \in [1,n], v_{i,1} \simeq_L^{\sigma_1,\sigma_2} v_{i,2}$ *with* $v_1 = \mathsf{ehAPIe}(\mathcal{G}, \sigma_1, pc_1, a_1, v_{1,1}, \cdots, v_{n,1})$ *and* $v_2 = \mathsf{ehAPIe}(\mathcal{G}, \sigma_2, pc_2, a_2, v_{1,2}, \cdots, v_{n,2})$, *then* $v_1 \simeq_L^{\sigma_1,\sigma_2} v_2$

Proof.
By induction on the structure of $\mathcal{E} :: v_1 = \mathsf{ehAPIe}(...)$ and $\mathcal{D} :: v_2 = \mathsf{ehAPIe}(...)$.
By assumption,
(1) $\sigma_1 \approx_L \sigma_2$
(2) $pc_1, pc_2 \sqsubseteq L$

Unstructured EH storage:
By assumption,
(U.1) $\forall i \in [1,n], v_{i,1} \simeq_L v_{i,2}$
(U.2) $\mathcal{E} :: v_1 = \mathsf{ehAPIe}(\mathcal{G}, \sigma_1, pc_1, id, v_{1,1}, \cdots, v_{n,1})$
(U.3) $\mathcal{D} :: v_2 = \mathsf{ehAPIe}(\mathcal{G}, \sigma_2, pc_2, id, v_{1,2}, \cdots, v_{n,2})$

Case U.I: $\mathcal{E}$ or $\mathcal{D}$ ends in EHAPI-NC
Without loss of generality, assume $\mathcal{E}$ ends in EHAPI-F. The case for $\mathcal{D}$ and both $\mathcal{E}$ and $\mathcal{D}$ ending in EHAPI-F are similar.
By assumption,
(U.I.1) $pc_1 = \cdot$
(U.I.2) $\exists \mathcal{E}_1 :: v_{H,1} = \mathsf{ehAPIe}(\mathcal{G}, \sigma_1, H, id, \mathsf{getFacetV}(v_{1,1}, H), \cdots, \mathsf{getFacetV}(v_{n,1}, H))$
(U.I.3) $\exists \mathcal{E}_2 :: v_{L,1} = \mathsf{ehAPIe}(\mathcal{G}, \sigma_1, L, id, \mathsf{getFacetV}(v_{1,1}, L), \cdots, \mathsf{getFacetV}(v_{n,1}, L))$
(U.I.4) $v_1 = \mathsf{createFct}(v_{H,1}, v_{L,1})$
From (U.1) and the definition of getFacet,
(U.I.5) $\forall i \in [1,n], \mathsf{getFacet}(v_{i,1}, L) \simeq_L v_{i,2}$
From (U.I.5), (U.I.3), and (U.3),
(U.I.6) the IH may be applied on $\mathcal{E}_2$ and $\mathcal{D}$
From (U.I.6) and applying the IH on $\mathcal{E}_2$ and $\mathcal{D}$,
(U.I.7) $v_{L,1} \simeq_L v_2$
From (U.I.7), (U.I.4), and the definition of setFacet,
$v_1 \simeq_L v_2$

Case U.II: $\mathcal{E}$ and $\mathcal{D}$ end in EHAPI
By assumption,
(U.II.1) $v_1 = \mathsf{ehAPIe}_{\mathcal{G}}(\sigma_1, pc_1, id, v_{1,1}, \cdots, v_{n,1})$
(U.II.2) $v_2 = \mathsf{ehAPIe}_{\mathcal{G}}(\sigma_2, pc_2, id, v_{1,2}, \cdots, v_{n,2})$
(U.II.3) $pc_1 = pc_2 = L$
From (1), (U.II.1)-(U.II.3), and Lemma 47.U,
The desired conclusion holds

Tree-structured EH storage:
By assumption,
(T.1) $\forall i \in [1,n], v_{i,1} \simeq_L^{\sigma_1,\sigma_2} v_{i,2}$
(T.2) $\mathcal{E} :: t_1 = \mathsf{ehAPIe}(\mathcal{G}, \sigma_1, v_{1,1}, \cdots, v_{n,1})$
(T.3) $\mathcal{D} :: t_2 = \mathsf{ehAPIe}(\mathcal{G}, \sigma_2, v_{1,2}, \cdots, v_{n,2})$

Case T.I: $\mathcal{E}$ or $\mathcal{D}$ ends in EHAPI-NC
Without loss of generality, assume $\mathcal{E}$ ends in EHAPI-F. The case for $\mathcal{D}$ and both $\mathcal{E}$ and $\mathcal{D}$ ending in EHAPI-F are similar.
By assumption,
(T.I.1) $pc_1 = \cdot$
(T.I.2) $\exists \mathcal{E}_1 :: v_{H,1} = \mathsf{ehAPIe}(\mathcal{G}, \sigma_1, H, \mathsf{getFacetV}(v_{1,1}, H), \cdots, \mathsf{getFacetV}(v_{n,1}, H))$
(T.I.3) $\exists \mathcal{E}_2 :: v_{L,1} = \mathsf{ehAPIe}(\mathcal{G}, \sigma_1, L, \mathsf{getFacetV}(v_{1,1}, L), \cdots, \mathsf{getFacetV}(v_{n,1}, L))$
(T.I.4) $v_1 = \mathsf{createFct}(v_{H,1}, v_{L,1})$
From (T.1) and the definition of getFacet,
(T.I.5) $\forall i \in [1,n], \mathsf{getFacet}(v_{i,1}, L) \simeq_L^{\sigma_1,\sigma_2} v_{i,2}$
From (T.I.5), (T.I.3), and (T.3),

(T.I.6) the IH may be applied on $\mathcal{E}_2$ and $\mathcal{D}$
From (T.I.6) and applying the IH on $\mathcal{E}_2$ and $\mathcal{D}$,
(T.I.7) $v_{L,1} \simeq_L^{\sigma_1,\sigma_2} v_2$
From (T.I.7), (T.I.4), and the definition of setFacet,
$v_1 \simeq_L^{\sigma_1,\sigma_2} v_2$

Case T.II: $\mathcal{E}$ and $\mathcal{D}$ end in EHAPI
By assumption,
(T.II.1) $v_1 = \mathsf{ehAPIe}_{\mathcal{G}}(\sigma_1, pc_1, v_{1,1}, \cdots, v_{n,1})$
(T.II.2) $v_2 = \mathsf{ehAPIe}_{\mathcal{G}}(\sigma_2, pc_2, v_{1,2}, \cdots, v_{n,2})$
(T.II.3) $pc_1 = pc_2 = L$
From (1), (T.II.1)-(T.II.3), and Lemma 47.T,
The desired conclusion holds

□

Lemma 47. *If* $\sigma_1 \approx_L \sigma_2$ *and* $pc_{l,1}, pc_{l,2} \sqsubseteq L$, *then*
Unstructured EH storage: *if* $\forall i \in [1,n] v_{i,1} \simeq_L v_{i,2}$, *then* $\mathsf{ehAPIe}_{\mathcal{G}}(\sigma_1, pc_1, id, v_{1,1}, \cdots, v_{n,1}) \simeq_L \mathsf{ehAPIe}_{\mathcal{G}}(\sigma_2, pc_2, id, v_{1,2}, \cdots, v_{n,2})$
Tree-structured EH storage: *if* $\forall i \in [1,n] v_{i,1} \simeq_L^{\sigma_1,\sigma_2} v_{i,2}$, *then* $\mathsf{ehAPIe}_{\mathcal{G}}(\sigma_1, pc_1, id, v_{1,1}, \cdots, v_{n,1}) \simeq_L^{\sigma_1,\sigma_2} \mathsf{ehAPIe}_{\mathcal{G}}(\sigma_2, pc_2, id, v_{1,2}, \cdots, v_{n,2})$

Proof (sketch): There are additional lemmas for each event handler API.
Unstructured EH storage: The conclusion follows from Lemma 50.U when ehAPIe is getVal.
Tree-structured EH storage: The conclusion follows from Lemma 50.T when ehAPIe is getVal, Lemma 51 when ehAPIe is getChildren, Lemma 52 when ehAPIe is moveRoot, Lemma 53 when ehAPIe is moveUp, Lemma 54 when ehAPIe is moveDown, and Lemma 55 when ehAPIe is moveRight. □

Lemma 48. *If* $\sigma_1 \approx_L \sigma_2$ *and* $pc_1, pc_2 \sqsubseteq L$ *then*
Unstructured EH storage: $\mathsf{lookup}_{\mathcal{G}}(\sigma_1, pc_1, id) \approx_L \mathsf{lookup}_{\mathcal{G}}(\sigma_2, pc_2, id)$
Tree-structured EH storage: $\mathsf{lookup}_{\mathcal{G}}(\sigma_1, pc_1, id) \approx_L^{\sigma_1,\sigma_2} \mathsf{lookup}_{\mathcal{G}}(\sigma_2, pc_2, id)$

Proof.
By induction on the structure of $\mathcal{E} :: \mathsf{lookup}_{\mathcal{G}}(\sigma_1, pc_1, id)$ and $\mathcal{D} :: \mathsf{lookup}_{\mathcal{G}}(\sigma_2, pc_2, id)$
By assumption,
(1) $\sigma_1 \approx_L \sigma_2$
(2) $pc_1, pc_2 \sqsubseteq L$

$\mathcal{G} = \mathsf{SMS}$
Unstructured EH storage:
Case U.I: $\mathcal{E}$ ends in SMS-LOOKUP
By assumption,
(U.I.1) $\sigma_1' = \mathsf{getStore}_{\mathsf{SMS}}(\sigma_1, pc_1)$
(U.I.2) $\sigma_2' = \mathsf{getStore}_{\mathsf{SMS}}(\sigma_2, pc_2)$
(U.I.3) $pc_1, pc_2 \neq \cdot$
(U.I.4) $\sigma_1'(id) = \phi_1$
From (2) and (U.I.3),
(U.I.5) $pc_1 = pc_2 = L$
From (U.I.5), (U.I.1), (U.I.2), and the definition of $\mathsf{getStore}_{\mathsf{SMS}}$ and $\approx_L$ for EH storage,
(U.I.6) $\sigma_1' = \sigma_2'$
From (U.I.6) and (U.I.4),
(U.I.7) $\sigma_2'(id) = \phi_1$
From (U.I.7),
(U.I.8) $\mathcal{D}$ ends in SMS-LOOKUP
From (U.I.8),
(U.I.9) $\sigma_2'(id) = \phi_2$
From (U.I.7) and (U.I.9),
$\phi_1 \approx_L \phi_2$

Case U.II: $\mathcal{E}$ ends in SMS-LOOKUP-S
By assumption,
(U.II.1) $\sigma_1' = \mathsf{getStore}_{\mathsf{SMS}}(\sigma_1, pc_1)$
(U.II.2) $\sigma_2' = \mathsf{getStore}_{\mathsf{SMS}}(\sigma_2, pc_2$
(U.II.3) $pc_1, pc_2 \neq \cdot$
(U.II.4) $id \notin \sigma_1'$
(U.II.5) $\phi_1 = \mathsf{NULL}$

From (2) and (U.II.3),
(U.II.6) $pc_1 = pc_2 = L$
From (U.II.6), (U.II.1), (U.II.2), and the definition of $\mathsf{getStore}_{\mathsf{SMS}}$ and $\approx_L$ for EH storage,
(U.II.7) $\sigma_1' = \sigma_2'$
From (U.II.4) and (U.II.7),
(U.II.8) $id \notin \sigma_2'$
From (U.II.8),
(U.II.9) $\mathcal{D}$ ends in SMS-LOOKUP-S
From (U.II.9),
(U.II.10) $\phi_2 = \mathsf{NULL}$
From (U.II.5) and (U.II.10),
$\phi_1 \approx_L \phi_2$

Tree-structured EH storage:
Case T.I: $\mathcal{D}$ ends in SMS-LOOKUP
By assumption,
(T.I.1) $\mathsf{lookupA}_{\mathsf{SMS}}(\sigma_1, pc_1, id, a_1^{\mathsf{rt}}) = a_1 \neq \mathsf{NULL}$
(T.I.2) $\sigma_1'(a_1) = \phi_1$
From (1),
(T.I.3) $a_1^{\mathsf{rt}} \approx_L^{\sigma_1,\sigma_2} a_2^{\mathsf{rt}}$
From (1), (2), (T.I.3), (T.I.1), and Lemma 49,
(T.I.4) $\mathsf{lookupA}_{\mathsf{SMS}}(\sigma_2, pc_2, id, a_2^{\mathsf{rt}}) = a_2^{\mathsf{rt}}$ with
(T.I.5) $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$
From (T.I.4), (T.I.5), and (T.I.1),
(T.I.6) $\mathcal{E}$ ends in SMS-LOOKUP
From (T.I.6),
(T.I.7) $\phi_2 = \sigma_2'(a_2)$
From (T.I.5), (T.I.2), and (T.I.7),
$\phi_1 \approx_L^{\sigma_1,\sigma_2} \phi_2$

Case T.II: $\mathcal{D}$ ends in SMS-LOOKUP-S
By assumption,
(T.II.1) $\mathsf{lookupA}_{\mathsf{SMS}}(\sigma_1, pc_1, id, a_1^{\mathsf{rt}}) = \mathsf{NULL}$
(T.II.2) $\phi_1 = \mathsf{NULL}$
From (1),
(T.II.3) $a_1^{\mathsf{rt}} \approx_L^{\sigma_1,\sigma_2} a_2^{\mathsf{rt}}$
From (1), (2), (T.II.3), and Lemma 49,
(T.II.4) $\mathsf{lookupA}_{\mathsf{SMS}}(\sigma_2, pc_2, id, a_2^{\mathsf{rt}}) = a_2$ with
(T.II.5) $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$
From (T.II.4), (T.II.5), and (T.II.1),
(T.II.6) $\mathcal{E}$ ends in SMS-LOOKUP-S
From (T.II.6),
(T.II.7) $\phi_2 = \mathsf{NULL}$
From (T.II.2) and (T.II.7),
$\phi_1 \approx_L \phi_2$

$\mathcal{G} = \mathsf{FS}$
Unstructured EH storage:
Case U.I: $\mathcal{E}$ or $\mathcal{D}$ ends in FS-LOOKUP-F
Without loss of generality, assume that $\mathcal{E}$ ends in LOOKUP-F. The cases for $\mathcal{D}$ or both $\mathcal{E}$ and $\mathcal{D}$ ending in FS-LOOKUP-F are similar.
By assumption,
(U.I.1) $\exists \mathcal{E}' :: \phi_L = \mathsf{lookup}_{\mathsf{FS}}(\sigma_1, L, id)$
(U.I.2) $\langle \_ | \phi_L \rangle = \mathsf{setFacetN}(\_, \phi_L)$
IH on $\mathcal{E}'$ and $\mathcal{D}$ gives
(U.I.3) $\phi_L \approx_L \phi_2$
From (U.I.2) and (U.I.3),
$\langle \_ | \phi_L \rangle \approx_L \phi_2$

Case U.II: $\mathcal{E}$ ends in FS-LOOKUP
By assumption,
(U.II.1) $\sigma_1(id) = \phi_1$
(U.II.2) $\phi_1.v \downarrow_{pc_1} \neq \cdot$

From (1), (U.II.1) and the definition of $\approx_L$ for EH stores,
(U.II.3) $\sigma_2(id) \approx_L \phi_1$
From (2), (U.II.3) and (U.II.2),
(U.II.4) $\sigma_2(id).v \downarrow_{pc_2} \neq \cdot$
If $pc_2 = \cdot$, then **Case U.I** applies. Otherwise, from (2),
(U.II.5) $pc_2 = L$
From (U.II.3) - (U.II.5),
(U.II.6) $\mathcal{D}$ ends in FS-LOOKUP
From (U.II.6),
(U.II.7) $\sigma_2(id) = \phi_2$
From (U.II.3) and (U.II.7),
$\phi_1 \approx_L \phi_2$

Case U.III: $\mathcal{E}$ ends in FS-LOOKUP-S
By assumption,
(U.III.1) $id \notin \sigma_1$ or
(U.III.2) $\sigma_1(id).v \downarrow_{pc_1} = \cdot$
(U.III.3) $\phi_1 = \mathsf{NULL}$
From (1), (2), (U.III.1), (U.III.2), and the definition of $\approx_L$ for EH stores,
(U.III.4) $id \notin \sigma_2$ or
(U.III.5) $\sigma_2(id).v \downarrow_{pc_2} = \cdot$
If $pc_2 = \cdot$, then **Case U.I** applies. Otherwise, from (2),
(U.III.6) $pc_2 = L$
From (U.III.4)-(U.III.6),
(U.III.7) $\mathcal{D}$ ends in FS-LOOKUP-S
From (U.III.7),
(U.III.8) $\phi_2 = \mathsf{NULL}$
From (U.III.3) and (U.III.8),
$\phi_1 \approx_L \phi_2$

Tree-structured EH storage:
Case T.I: $\mathcal{D}$ ends in FS-LOOKUP
By assumption,
(T.I.1) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_1, pc_1, id, a_1^{\mathsf{rt}}) = a_1 \neq \mathsf{NULL}$
(T.I.2) $\sigma_1'(a_1) = \phi_1$
From (1),
(T.I.3) $a_1^{\mathsf{rt}} \approx_L^{\sigma_1,\sigma_2} a_2^{\mathsf{rt}}$
From (1), (2), (T.I.3), (T.I.1), and Lemma 49,
(T.I.4) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_2, pc_2, id, a_2^{\mathsf{rt}}) = a_2$ with
(T.I.5) $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$
From (T.I.4), (T.I.5), and (T.I.1),
(T.I.6) $\mathcal{E}$ ends in FS-LOOKUP
From (T.I.6),
(T.I.7) $\phi_2 = \sigma_2(a_2)$
From (T.I.5), (T.I.2), and (T.I.7),
$\phi_1 \approx_L^{\sigma_1,\sigma_2} \phi_2$

Case T.II: $\mathcal{D}$ ends in FS-LOOKUP-S
By assumption,
(T.II.1) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_1, pc_1, id, a_1^{\mathsf{rt}}) = \mathsf{NULL}$
(T.II.2) $\phi_1 = \mathsf{NULL}$
From (1),
(T.II.3) $a_1^{\mathsf{rt}} \approx_L^{\sigma_1,\sigma_2} a_2^{\mathsf{rt}}$
From (1), (2), (T.II.3), and Lemma 49,
(T.II.4) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_2, pc_2, id, a_2^{\mathsf{rt}}) = a_2$ with
(T.II.5) $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$
From (T.II.4), (T.II.5), and (T.II.1),
(T.II.6) $\mathcal{E}$ ends in FS-LOOKUP-S
From (T.II.6),
(T.II.7) $\phi_2 = \mathsf{NULL}$
From (T.II.2) and (T.II.7),
$\phi_1 \approx_L \phi_2$

$\mathcal{G} = \mathsf{TS}$
Note: for $\mathcal{G} = \mathsf{TS}$ only unstructured EH storage rules exist
Case I: $\mathcal{E}$ ends in TS-LOOKUP
By assumption,
(I.1) $\sigma_1(id) = \phi_1$

Subcase i: $\mathsf{labOf}(\phi_1) \sqsubseteq L$
From (1), (I.1) and the definition of $\approx_L$ for EH stores,
(i.1) $id \in \sigma_2$ and
(i.2) $\sigma_2(id) \approx_L \phi_1$
By assumption and from (i.1) and (i.2),
(i.3) $\mathcal{D}$ ends in TS-LOOKUP
From (i.3),
(i.4) $\sigma_2(id) = \phi_2$
From (i.2) and (i.4),
$\phi_1 \approx_L \phi_2$

Subcase ii: $\mathsf{labOf}(\phi_1) \not\sqsubseteq L$
By assumption and from (I.2), either
(ii.1) $\mathsf{labOf}(\sigma_2(id)) \not\sqsubseteq L$ or
(ii.2) $id \notin \sigma_2$

Subsubcase a: (ii.1) is true
The proof for this case is similar to **Subcase i**.

Subsubcase b: (ii.2) is true
From (ii.2),
(b.1) $\mathcal{D}$ ends in TS-LOOKUP-S
From (b.1),
(b.2) $\phi_2 = (\mathsf{NULL}, H)$
By assumption and from (b.2),
$\phi_1 \approx_L \phi_2$

Case II: $\mathcal{E}$ ends in TS-LOOKUP-S
By assumption,
(II.1) $id \notin \sigma_1$
(II.2) $\phi_1 = (\mathsf{NULL}, H)$
From (1), (II.1) and the definition of $\approx_L$ for EH stores,
(II.3) $id \notin \sigma_2$ or
(II.4) $\sigma_2(id) = \phi$ with $\phi \downarrow_L = \cdot$

Subcase i: (II.3) is true
From (II.3),
(i.1) $\mathcal{D}$ ends in TS-LOOKUP-S
From (i.1),
(i.2) $\phi_2 = (\mathsf{NULL}, H)$
From (II.2) and (i.2),
$\phi_1 \approx_L \phi_2$

Subcase ii: (II.4) is true
From (II.4),
(ii.1) $\mathcal{D}$ ends in TS-LOOKUP
From (ii.1),
(ii.2) $\phi_2 = \sigma_2(id)$
From (II.2), (II.4), and (ii.2),
$\phi_1 \approx_L \phi_2$

□

Lemma 49. *If* $\sigma_1 \approx_L \sigma_2$, $pc_1, pc_2 \sqsubseteq L$ *and* $A_1 \approx_L^{\sigma_1,\sigma_2} A_2$, *then* $\mathsf{lookupA}_{\mathcal{G}}(\sigma_1, pc_1, id, A_1) \approx_L^{\sigma_1,\sigma_2} \mathsf{lookupA}_{\mathcal{G}}(\sigma_2, pc_2, id, A_2)$

Proof.
Note that lookupA is only defined for the tree-structured EH storage.
By induction on the structure of $\mathcal{E} :: \mathsf{lookupA}_{\mathcal{G}}(\sigma_1, pc_1, id, A_1)$ and $\mathcal{D} :: \mathsf{lookupA}_{\mathcal{G}}(\sigma_2, pc_2, id, A_2)$

By assumption,
(1) $\sigma_1 \approx_L \sigma_2$
(2) $pc_1, pc_2 \sqsubseteq L$
(3) $A_1 \approx_L^{\sigma_1,\sigma_2} A_2$

$\mathcal{G} = \mathsf{SMS}$
Denote
$\mathcal{E} :: \mathsf{lookupA}_{\mathcal{G}}(\sigma_1, pc_1, id, A_1) = a_1$ and $\mathcal{D} :: \mathsf{lookupA}_{\mathcal{G}}(\sigma_2, pc_2, id, A_2) = a_2$

Case I: $\mathcal{E}$ ends in SMS-LOOKUPA
By assumption,
(I.1) $A_1 = (a_1' :: A_1')$
(I.2) $\sigma_1'(a_1').id = id$
(I.3) $a_1 = a_1'$
From (I.2),
(I.4) $a_1' \neq \mathsf{NULL}$
From (3), (I.1), and (I.4),
(I.5) $A_2 = (a_2' :: A_2')$ with
(I.6) $a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$ and
(I.7) $a_2' \neq \mathsf{NULL}$
From (I.2), (I.6), and (I.7),
(I.8) $\sigma_2'(a_2').id = id$
From (I.5) and (I.8),
(I.9) $\mathcal{D}$ ends in SMS-LOOKUPA
From (I.9),
(I.10) $a_2 = a_2'$
From (I.3), (I.10), and (I.6),
$a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

Case II: $\mathcal{E}$ ends in SMS-LOOKUPA-R
By assumption,
(II.1) $A_1 = (a_1' :: A_1')$
(II.2) $\sigma_1'(a_1').id = id'$ with
(II.3) $id' \neq id$
(II.4) $\sigma_1'(a_1').A = A_1''$
(II.5) $\mathcal{E}' :: a_1 = \mathsf{lookup}_{\mathsf{SMS}}(\sigma_1, pc_1, id, (A_1' :: A_1''))$
From (II.5),
(II.6) $a_1' \neq \mathsf{NULL}$
From (3), (II.1), and (II.6),
(II.7) $A_2 = (a_2' :: A_2')$ with
(II.8) $A_1' \approx_L^{\sigma_1,\sigma_2} A_2'$
(II.9) $a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$ and
(II.10) $a_2' \neq \mathsf{NULL}$
From (II.2), (II.9), and (II.10),
(II.11) $\sigma_2'(a_2').id = id'$
From (II.3) and (II.11),
(II.12) $\mathcal{D}$ ends in SMS-LOOKUPA-R
From (II.12),
(II.13) $\mathcal{D}' :: a_2 = \mathsf{lookup}_{\mathsf{SMS}}(\sigma_2, pc_2, id, (A_2' :: A_2''))$ for
(II.14) $A_2'' = \sigma_2'(a_2').A$
From (II.9), (II.4), and (II.14),
(II.15) $A_1'' \approx_L^{\sigma_1,\sigma_2} A_2''$
From (II.5), (II.13), (II.8), and (II.15),
IH may be applied on $\mathcal{E}'$ and $\mathcal{D}'$
From IH on $\mathcal{E}'$ and $\mathcal{D}'$,
$a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

Case III: $\mathcal{E}$ ends in SMS-LOOKUPA-S
By assumption,
(III.1) $A_1 = \cdot$
(III.2) $a_1 = \mathsf{NULL}$
From (3), (III.2), and the definition of $\approx_L^{\sigma,\sigma'}$ for A,

(III.3) $A_2 \approx_L^{\sigma_2,\sigma_1} \cdot$
From (III.3) and the definition of $\downarrow_L^{\sigma}$ for ϕ^{std},
(III.4) $A_2 = \cdot$
From (III.4),
(III.5) $\mathcal{D}$ ends in SMS-LOOKUPA-S
From (III.5),
(III.6) $a_2 = \mathsf{NULL}$
From (III.2) and (III.6),
$a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

$\mathcal{G} = \mathsf{FS}$
Case I: $\mathcal{E}$ or $\mathcal{D}$ ends in FS-LOOKUPA-F
Without loss of generality, assume that $\mathcal{E}$ ends in FS-LOOKUPA-F. The cases for $\mathcal{D}$ or both $\mathcal{E}$ and $\mathcal{D}$ ends in FS-LOOKUPA-F are similar.
By assumption,
(I.1) $\exists \mathcal{E}' :: a_L = \mathsf{lookup}_{\mathsf{FS}}(\sigma_1, L, id, A_1 \downarrow_L$
(I.2) $a_1 = \mathsf{setFacet}(\_, a_L)$
IH on $\mathcal{E}'$ and $\mathcal{D}$ gives
(I.3) $a_L \approx_L^{\sigma_1,\sigma_2} a_2$
From (I.2), (I.3), and the definition of setFacet,
$a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

Case II: $\mathcal{E}$ ends in FS-LOOKUPA
By assumption,
(II.1) $A_1 = (a_1' :: A_1')$ and
(II.2) $\sigma_1(a_1').id = id$ with
(II.3) $\sigma_1(a_1').v \downarrow_{pc_1} \neq \cdot$
(II.4) $a_1 = a_1'$
If $pc_2 = \cdot$, then **Case I** applies. Otherwise, from (2),
(II.5) $pc_2 = L$
From (3), (II.1), and the definition of $\approx_L^{\sigma,\sigma'}$ for A,
(II.6) $A_2 = (a_2' :: A_2')$ with either
(II.7) $a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$ or
(II.8) $A_1 \approx_L^{\sigma_1,\sigma_2} A_2'$

Subcase i: (II.7) is true
From (II.7) and (II.2),
(i.1) $\sigma_2(a_2').id = id$
From (II.5) and (i.1),
(i.2) $\mathcal{D}$ ends in FS-LOOKUPA
From (i.2),
(i.3) $a_2 = a_2'$
From (II.4), (i.3), and (II.7),
$a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

Subcase ii: (II.8) is true
From (II.8) and (3),
(ii.1) $\sigma_2(a_2) \downarrow_{pc_2} = \cdot$
From (II.5) and (ii.1),
(ii.2) $\mathcal{D}$ ends in FS-LOOKUPA-R2
From (ii.2),
(ii.3) $\mathcal{D}' :: a_2 = \mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id, A_2')$
From (II.8),
IH may be applied on $\mathcal{E}$ and $\mathcal{D}'$
From IH on $\mathcal{E}$ and $\mathcal{D}'$,
$a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

Case III: $\mathcal{E}$ ends in FS-LOOKUPA-R
By assumption,
(III.1) $A_1 = (a_1' :: A_1')$ and
(III.2) $\sigma_1(a_1').id \neq id$ with
(III.3) $\sigma_1(a_1').v \downarrow_{pc_1} \neq \cdot$

(III.4) $\mathcal{E}' :: a_1 = \mathsf{lookup}_{\mathsf{FS}}(\sigma_1, pc_1, id, (list'_1 :: \sigma_1(a'_1).A \downarrow_{pc_1}))$

If $pc_2 = \cdot$, then **Case I** applies. Otherwise, from (2),

(III.5) $pc_2 = L$

From (3), (III.1), and the definition of $\approx_L^{\sigma,\sigma'}$ for A,

(III.6) $A_2 = (a'_2 :: A'_2)$ with either

(III.7) $a'_1 \approx_L^{\sigma_1,\sigma_2} a'_2$ and

(III.8) $A'_1 \approx_L^{\sigma_1,\sigma_2} A'_2$, or

(III.9) $A_1 \approx_L^{\sigma_1,\sigma_2} A'_2$

Subcase i: (III.7) and (III.8) is true

From (III.7) and (III.2),

(i.1) $\sigma_2(a'_2).id \neq id$

From (III.5) and (i.1),

(i.2) $\mathcal{D}$ ends in FS-LOOKUPA-R

From (i.2),

(i.3) $\mathcal{D}' :: a_2 = \mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id, (A'_2 :: \sigma_2(a'_2).A \downarrow_{pc_2}))$

From (III.7),

(i.4) $\sigma_1(a'_1).A \approx_L^{\sigma_1,\sigma_2} \sigma_2(a'_2).A$

From (2), (i.4) and the definition of $\approx_L^{\sigma_1,\sigma_2}$ for A,

(i.5) $\sigma_1(a'_1).A \downarrow_{pc_1} \approx_L^{\sigma_1,\sigma_2} \sigma_2(a'_2).A \downarrow_{pc_2}$

From (III.4), (i.3), (III.8), and (i.5),

IH may be applied on $\mathcal{E}'$ and $\mathcal{D}'$

From IH on $\mathcal{E}'$ and $\mathcal{D}'$,

$a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

Subcase ii: (III.9) is true

From (3), (III.6), and (III.9),

(ii.1) $\sigma_1(a'_2).v \downarrow_{pc_2} = \cdot$

From (ii.1) and (III.5),

(ii.2) $\mathcal{D}$ ends in FS-LOOKUPA-R2

From (ii.2),

(ii.3) $\mathcal{D}' :: a_2 = \mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id, A'_2)$

From (III.9),

IH may be applied on $\mathcal{E}$ and $\mathcal{D}'$

From IH on $\mathcal{E}$ and $\mathcal{D}'$,

$a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

Case IV: $\mathcal{E}$ ends in FS-LOOKUPA-R2

The proof for this case is similar to **Subcase III.ii**.

Case V: $\mathcal{E}$ ends in FS-LOOKUPA-S

By assumption,

(V.1) $A_1 = \cdot$

(V.2) $a_1 = \mathsf{NULL}$

If $pc_2 = \cdot$, then **Case I** applies. Otherwise, from (2),

(V.3) $pc_2 = L$

From (3), (V.1), and the definition of $\approx_L^{\sigma,\sigma'}$ for A

(V.4) $A_2 \approx_L^{\sigma_2,\sigma_1} \cdot$

From (V.4), either

(V.5) $A_2 = \cdot$ or

(V.6) $A_2 = a'_2 :: A'_2$ with

(V.7) $\sigma_2(a'_2).v \downarrow_{pc_2} = \cdot$ and

(V.8) $A'_2 \approx_L^{\sigma_2,\sigma_1} \cdot$

Subcase i: (V.5) is true

From (V.3) and (V.5),

(i.1) $\mathcal{D}$ ends in FS-LOOKUPA-S

From (i.1),

(i.2) $a_2 = \mathsf{NULL}$

From (V.2) and (i.2),

$a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

Subcase ii: (V.6)-(V.8) are true
From (V.3) and (V.6)-(V.8),
(ii.1) $\mathcal{D}$ ends in FS-LOOKUPA-R2
From (ii.1),
(ii.2) $\mathcal{D}' :: a_2 = \mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id, A'_2)$
From (V.1) and (V.8),
(ii.3) IH may be applied on $\mathcal{E}$ and $\mathcal{D}'$
From IH on $\mathcal{E}$ and $\mathcal{D}'$,
$a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

□

Lemma 50. *If $\sigma_1 \approx_L \sigma_2$ and $pc_1, pc_2 \sqsubseteq L$, then*
Unstructured EH storage: $\mathsf{getVal}_{\mathcal{G}}(\sigma_1, pc_1, id) \simeq_L \mathsf{getVal}_{\mathcal{G}}(\sigma_2, pc_2, id)$
Tree-structured EH storage: *if* $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$ $\mathsf{getVal}_{\mathcal{G}}(\sigma_1, pc_1, a_1) \simeq_L^{\sigma_1,\sigma_2} \mathsf{getVal}_{\mathcal{G}}(\sigma_2, pc_2, a_2)$

Proof.
Only the cases for $\mathcal{G} \neq \mathsf{TS}$ are considered. The other cases are proven in the weak secrecy version: Lemma 65 (Requirement **(WE1)**).
We examine each case of $\mathcal{G}$
Denote $\mathcal{D} :: \mathsf{getVal}_{\mathcal{G}}(\sigma_1, pc_1, id) = v_1$ and
$\mathcal{E} :: \mathsf{getVal}_{\mathcal{G}}(\sigma_2, pc_2, id) = v_2$ for the unstructured EH storage, and
Denote $\mathcal{D} :: \mathsf{getVal}_{\mathcal{G}}(\sigma_1, pc_1, a_1) = v_1$ and
$\mathcal{E} :: \mathsf{getVal}_{\mathcal{G}}(\sigma_2, pc_2, a_2) = v_2$ for the tree-structured EH storage
By assumption,
(1) $pc_1, pc_2 \sqsubseteq L$
(2) $\sigma_1 \approx_L \sigma_2$

$\mathcal{G} = \mathsf{SMS}$
Unstructured EH storage
Case U.I: $\mathcal{D}$ ends in SMS-GETVAL
By assumption,
(U.I.1) $\mathsf{lookup}_{\mathsf{SMS}}(\sigma_1, pc_1, id) = \phi_1 \neq \mathsf{NULL}$
(U.I.2) $v_1 = \phi_1.v$
From (1), (2), (U.I.1), and Lemma 48.U,
(U.I.3) $\mathsf{lookup}_{\mathsf{SMS}}(\sigma_2, pc_2, id) = \phi_2$
(U.I.4) $\phi_1 \approx_L \phi_2$
From (U.I.1) and (U.I.4),
(U.I.5) $\phi_2 = \phi_1 \neq \mathsf{NULL}$
From (U.I.3) and (U.I.5),
(U.I.6) $\mathcal{E}$ ends in SMS-LOOKUP
From (U.I.6),
(U.I.7) $v_2 = \phi_2.v$
From (U.I.2), (U.I.7), and (U.I.5),
The desired conclusion holds

Case U.II: $\mathcal{D}$ ends in SMS-GETVAL-S
By assumption,
(U.II.1) $\mathsf{lookup}_{\mathsf{SMS}}(\sigma_1, pc_1, id) = \mathsf{NULL}$
(U.II.2) $v_1 = \mathsf{dv}$
From (1), (2), (U.I.1), and Lemma 48.U,
(U.II.3) $\mathsf{lookup}_{\mathsf{SMS}}(\sigma_2, pc_2, id) = \phi_2$
(U.II.4) $\phi_1 \approx_L \phi_2$
From (U.II.1) and (U.II.4),
(U.II.5) $\phi_2 = \phi_1 = \mathsf{NULL}$
From (U.II.5),
(U.II.6) $\mathcal{E}$ ends in SMS-GETVAL-S
From (U.II.6),
(U.II.7) $v_2 = \mathsf{dv}$
From (U.II.2) and (U.II.7),
The desired conclusion holds

Tree-structured EH storage
By assumption,
(T.1) $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

Denote
$\sigma_1' = \mathsf{getStore}(\sigma_1, pc_1)$
$\sigma_2' = \mathsf{getStore}(\sigma_2, pc_2)$

Case T.I $\mathcal{D}$ ends in SMS-GETVAL
By assumption,
(T.I.1) $\phi_1 = \sigma_1'(a_1)$
(T.I.2) $v_1 = \phi_1.v$
From (T.1), T (T.I.1),
(T.I.3) $\phi_2 = \sigma_2'(a_2)$ and
(T.I.4) $\phi_1 \approx_L^{\sigma_1,\sigma_2} \phi_2$
From (T.I.3),
(T.I.5) $\mathcal{E}$ ends in SMS-GETVAL
From (T.I.5),
(T.I.6) $v_2 = \phi_2.v$
From (T.I.2), (T.I.6), and (T.I.4),
The desired conclusion holds

Case T.II $\mathcal{D}$ ends in SMS-GETVAL-S
By assumption,
(T.II.1) $a_1 = \mathsf{NULL}$
(T.II.2) $v_1 = \mathsf{dv}$
From (T.1) and (T.II.1),
(T.II.3) $a_2 = \mathsf{NULL}$
From (T.II.3),
(T.II.4) $\mathcal{E}$ ends in SMS-GETVAL-S
From (T.II.4),
(T.II.5) $v_2 = \mathsf{dv}$
From (T.II.2) and (T.II.5),
$v_1 \approx_L^{\sigma_1,\sigma_2} v_2$

$\mathcal{G} = \mathsf{FS}$
Unstructured EH storage
Case U.I: $\mathcal{D}$ ends in FS-GETVAL,
By assumption,
(U.I.1) $\mathsf{lookup}_{\mathsf{FS}}(\sigma_1, pc_1, id) = \phi_1 \neq \mathsf{NULL}$
(U.I.2) $v_1 = \mathsf{getFacetV}(\phi_1.v, pc_1)$
From (1), (2), (U.I.1), and Lemma 48.U,
(U.I.3) $\phi_1 \approx_L \phi_2$
From (U.I.1) and (U.I.3),
(U.I.4) $\phi_2 \neq \mathsf{NULL}$
From (U.I.4),
(U.I.5) $\mathcal{E}$ ends in FS-GETVAL
From (U.I.5),
(U.I.6) $v_2 = \mathsf{getFacetV}(\phi_2.v, pc_2)$
From (U.I.3), (1), and the definition of getFacetV,
The desired conclusion holds

Case U.II: $\mathcal{D}$ ends in FS-GETVAL-S
By assumption,
(U.II.1) $\mathsf{lookup}_{\mathsf{FS}}(\sigma_1, pc_1, id) = \mathsf{NULL}$
(U.II.2) $v_1 = \mathsf{dv}$
From (1), (2), (U.I.1), and Lemma 48.U,
(U.II.3) $\phi_1 \approx_L \phi_2$
From (U.II.1) and (U.II.3),
(U.II.4) $\phi_2 = \mathsf{NULL}$
From (U.II.4),
(U.II.5) $\mathcal{E}$ ends in FS-GETVAL-S
From (U.II.5),
(U.II.6) $v_2 = \mathsf{dv}$
From (U.II.2) and (U.II.6),
The desired conclusion holds

Tree-structured EH storage
By assumption,
(T.1) $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

Case T.I: $\mathcal{D}$ ends in FS-GETVAL
By assumption,
(T.I.1) $\sigma_1(a_1) = \phi_1$
(T.I.2) $v_1 = \mathsf{getFacetV}(\phi_1.v, pc_1)$
From (T.1) and (T.I.1),
(T.I.3) $\sigma_2(a_2) = \phi_2$ with
(T.I.4) $\phi_1 \approx_L^{\sigma_1,\sigma_2} \phi_2$
From (T.I.3),
(T.I.5) $\mathcal{E}$ ends in FS-GETVAL
From (T.I.5),
(T.I.6) $a_2 = \mathsf{getFacetV}(\phi_2.v, pc_2)$
From (T.I.2), (T.I.6), and (T.I.4),
The desired conclusion holds

Case T.II: $\mathcal{D}$ ends in FS-GETVAL-S
By assumption,
(T.II.1) $a_1 = \mathsf{NULL}$
(T.II.2) $v_1 = \mathsf{dv}$
From (T.1) and (T.II.1),
(T.II.3) $a_2 = \mathsf{NULL}$
From (T.II.3),
(T.II.4) $\mathcal{E}$ ends in FS-GETVAL-S
From (T.II.4),
(T.II.5) $v_2 = \mathsf{dv}$
From (T.II.2) and (T.II.5),
The desired conclusion holds

□

Lemma 51. *If* $\sigma_1 \approx_L \sigma_2$, $pc_1, pc_2 \sqsubseteq L$, *and* $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$, *then* $\mathsf{getChildren}_{\mathcal{G}}(\sigma_1, pc_1, a_1) \simeq_L^{\sigma_1,\sigma_2} \mathsf{getChildren}_{\mathcal{G}}(\sigma_2, pc_2, a_2)$

Proof.
Note that getChildren is defined only for the tree-structured EH storage.
We examine each case of $\mathcal{G}$
Denote $\mathcal{D} :: \mathsf{getChildren}_{\mathcal{G}}(\sigma_1, pc_1, a_1) = v_1$ and
$\mathcal{E} :: \mathsf{getChildren}_{\mathcal{G}}(\sigma_2, pc_2, a_2) = v_2$
By assumption,
(1) $pc_1, pc_2 \sqsubseteq L$
(2) $\sigma_1 \approx_L \sigma_2$
(3) $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

$\mathcal{G} = \mathsf{SMS}$
Denote
$\sigma_1' = \mathsf{getStore}(\sigma_1, pc_1)$
$\sigma_2' = \mathsf{getStore}(\sigma_2, pc_2)$

Case I: $\mathcal{D}$ ends in SMS-GETCHILDREN
By assumption,
(I.1) $\phi_1 = \sigma_1'(a_1)$
(I.2) $v_1 = \mathsf{len}(\phi_1.A)$
From (3) and (I.1),
(I.3) $\phi_2 = \sigma_2'(a_2)$ with
(I.4) $\phi_1 \approx_L^{\sigma_1,\sigma_2} \phi_2$
From (I.3),
(I.5) $\mathcal{E}$ ends in SMS-GETCHILDREN
From (I.5),
(I.6) $v_2 = \mathsf{len}(\phi_2.A)$
From (I.4), (I.2), and (I.6),
$v_1 \simeq_L^{\sigma_1,\sigma_2} v_2$

Case II: $\mathcal{D}$ ends in SMS-GETCHILDREN-S
By assumption,
(II.1) $a_1 = \mathsf{NULL}$
(II.2) $v_1 = \mathsf{dv}$
From (3) and (II.1),
(II.3) $a_2 = \mathsf{NULL}$
From (II.3),
(II.4) $\mathcal{E}$ ends in SMS-GETCHILDREN-S
From (I.5),
(II.5) $v_2 = \mathsf{dv}$
From (II.2) and (II.5),
$v_1 \simeq_L^{\sigma_1,\sigma_2} v_2$

$\mathcal{G} = \mathsf{FS}$
Case I: $\mathcal{D}$ ends in FS-GETCHILDREN
By assumption,
(I.1) $\phi_1 = \sigma_1'(a_1)$
(I.2) $\phi_1.v \downarrow_{pc_1} \neq \cdot$
(I.3) $v_1 = \mathsf{len}(\phi_1.A \downarrow_{pc_1})$
From (3) and (I.1),
(I.4) $\phi_2 = \sigma_2'(a_2)$ with
(I.5) $\phi_1 \approx_L^{\sigma_1,\sigma_2} \phi_2$
From (1), (I.2), and (I.5),
(I.6) $\phi_2.v \downarrow_{pc_2} \neq \cdot$
From (I.4) and (I.6),
(I.7) $\mathcal{E}$ ends in FS-GETCHILDREN
From (I.7),
(I.8) $v_2 = \mathsf{len}(\phi_2.A \downarrow_{pc_2})$
From (I.3), (I.8), and (I.5),
$v_1 \simeq_L^{\sigma_1,\sigma_2} v_2$

Case II: $\mathcal{D}$ ends in FS-GETCHILDREN-S
By assumption,
(II.1) $\sigma_1(a_1).v \downarrow_{pc_1} = \cdot$ or
(II.2) $a_1 = \mathsf{NULL}$
(II.3) $v_1 = \mathsf{dv}$
From (3), if (II.1) is true,
(II.4) $\sigma_2(a_2).v \downarrow_{pc_2} = \cdot$ or
From (3), if (II.2) is true,
(II.5) $a_2 = \mathsf{NULL}$
From (II.4) and (II.5),
(II.6) $\mathcal{E}$ ends in FS-GETCHILDREN-S
From (I.6),
(II.7) $v_2 = \mathsf{dv}$
From (II.3) and (II.7),
$v_1 \simeq_L^{\sigma_1,\sigma_2} v_2$

□

Lemma 52. *If* $\sigma_1 \approx_L \sigma_2$ *and* $pc_1, pc_2 \sqsubseteq L$, *then* $\mathsf{moveRoot}_{\mathcal{G}}(\sigma_1, pc_1) \simeq_L^{\sigma_1,\sigma_2} \mathsf{moveRoot}_{\mathcal{G}}(\sigma_2, pc_2)$

Proof.
Note that moveRoot is defined only for the tree-structured EH storage.
We examine each case of $\mathcal{G}$
Denote $\mathcal{D} :: \mathsf{moveRoot}_{\mathcal{G}}(\sigma_1, pc_1) = a_1$ and
$\mathcal{E} :: \mathsf{moveRoot}_{\mathcal{G}}(\sigma_2, pc_2) = a_2$
By assumption,
(1) $pc_1, pc_2 \sqsubseteq L$
(2) $\sigma_1 \approx_L \sigma_2$

$\mathcal{G} = \mathsf{SMS}$
Denote
$\sigma_1' = \mathsf{getStore}(\sigma_1, pc_1)$
$\sigma_2' = \mathsf{getStore}(\sigma_2, pc_2)$

Case I: $\mathcal{D}$ and $\mathcal{E}$ ends in SMS-MOVEROOT
By assumption,
(I.1) $a_1 = a_1^{\mathsf{rt}}$
(I.2) $a_2 = a_2^{\mathsf{rt}}$
From (2),
(I.3) $\sigma_1'(a_1^{\mathsf{rt}}) \approx_L^{\sigma_1,\sigma_2} \sigma_2'(a_2^{\mathsf{rt}})$
From (I.1)-(I.3),
$a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

$\mathcal{G} = \mathsf{FS}$
Case I: $\mathcal{D}$ and $\mathcal{E}$ ends in FS-MOVEROOT
By assumption,
(I.1) $a_1 = a_1^{\mathsf{rt}}$
(I.2) $a_2 = a_2^{\mathsf{rt}}$
From (2),
(I.3) $\sigma_1(a_1^{\mathsf{rt}}) \approx_L^{\sigma_1,\sigma_2} \sigma_2(a_2^{\mathsf{rt}})$
From (I.1)-(I.3),
$a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

□

Lemma 53. *If* $\sigma_1 \approx_L \sigma_2$, $pc_1, pc_2 \sqsubseteq L$, *and* $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$, *then* $\mathsf{moveUp}_{\mathcal{G}}(\sigma_1, pc_1, a_1) \simeq_L^{\sigma_1,\sigma_2} \mathsf{moveUp}_{\mathcal{G}}(\sigma_2, pc_2, a_2)$

Proof.
Note that moveUp is defined only for the tree-structured EH storage.
We examine each case of $\mathcal{G}$
Denote $\mathcal{D} :: \mathsf{moveUp}_{\mathcal{G}}(\sigma_1, pc_1, a_1) = a_1'$ and
$\mathcal{E} :: \mathsf{moveUp}_{\mathcal{G}}(\sigma_2, pc_2, a_2) = a_2'$
By assumption,
(1) $pc_1, pc_2 \sqsubseteq L$
(2) $\sigma_1 \approx_L \sigma_2$
(3) $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

$\mathcal{G} = \mathsf{SMS}$
Denote
$\sigma_1' = \mathsf{getStore}(\sigma_1, pc_1)$
$\sigma_2' = \mathsf{getStore}(\sigma_2, pc_2)$

Case I: $\mathcal{D}$ ends in SMS-MOVEU
By assumption,
(I.1) $a_1' = \sigma_1'(a_1).a_p$
From (I.1) and (3),
(I.2) $a_2 \in \sigma_2'$
From (I.2),
(I.3) $\mathcal{E}$ must end in SMS-MOVEU
From (I.3),
(I.4) $a_2' = \sigma_2'(a_2).a_p$
From (2), (I.1), and (I.4),
(I.5) $\sigma_1'(a_1').id = \sigma_2'(a_2').id$
From (2), (I.5), and since id's are unique,
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

Case II: $\mathcal{D}$ ends in SMS-MOVEU-S
By assumption,
(II.1) $a_1 = \mathsf{NULL}$
(II.2) $a_1' = \mathsf{NULL}$
From (3) and (II.1),
(II.3) $a_2 = \mathsf{NULL}$
From (II.2),
(II.4) $\mathcal{E}$ ends in SMS-MOVEU-S
From (II.4),
(II.5) $a_2' = \mathsf{NULL}$
From (II.2) and (II.5),
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

$\mathcal{G} = \mathsf{FS}$
From the structure of the operational semantics, pc_1, pc_2 is not $\cdot$

Case I: $\mathcal{D}$ ends in FS-MOVEU
By assumption,
(I.1) $a_1' = \sigma_1'(a_1).a_p \downarrow_{pc_1}$
From (I.1) and (3),
(I.2) $a_2 \in \sigma_2'$
From (I.2),
(I.3) $\mathcal{E}$ must end in FS-MOVEU
From (I.3),
(I.4) $a_2' = \sigma_2'(a_2).a_p \downarrow_{pc_2}$
From (2), (I.1), and (I.4),
(I.5) $\sigma_1'(a_1').id = \sigma_2'(a_2').id$
From (2), (I.5), and since id's are unique,
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

Case II: $\mathcal{D}$ ends in FS-MOVEU-S1
By assumption,
(II.1) $\sigma(a_1).a_p \downarrow_{pc_1} = \cdot$
(II.2) $a_1' = \mathsf{NULL}$
From (3) and (II.1),
(II.3) $\sigma(a_2).a_p \downarrow_{pc_2} = \cdot$
From (II.3),
(II.4) $\mathcal{E}$ ends in FS-MOVEU-S1
From (II.4),
(II.5) $a_2' = \mathsf{NULL}$
From (II.2) and (II.5),
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

Case III: $\mathcal{D}$ ends in FS-MOVEU-S2
By assumption,
(III.1) $a_1 = \mathsf{NULL}$
(III.2) $a_1' = \mathsf{NULL}$
From (3) and (III.1),
(III.3) $a_2 = \mathsf{NULL}$
From (III.2),
(III.4) $\mathcal{E}$ ends in FS-MOVEU-S2
From (II.4),
(III.5) $a_2' = \mathsf{NULL}$
From (III.2) and (III.5),
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

□

Lemma 54. *If* $\sigma_1 \approx_L \sigma_2$, $pc_1, pc_2 \sqsubseteq L$, *and* $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$, *then* $\mathsf{moveDown}_{\mathcal{G}}(\sigma_1, pc_1, a_1) \simeq_L^{\sigma_1,\sigma_2} \mathsf{moveDown}_{\mathcal{G}}(\sigma_2, pc_2, a_2)$

Proof.
Note that moveDown is defined only for the tree-structured EH storage.
We examine each case of $\mathcal{G}$
Denote $\mathcal{D} :: \mathsf{moveDown}_{\mathcal{G}}(\sigma_1, pc_1, a_1) = a_1'$ and
$\mathcal{E} :: \mathsf{moveDown}_{\mathcal{G}}(\sigma_2, pc_2, a_2) = a_2'$
By assumption,
(1) $pc_1, pc_2 \sqsubseteq L$
(2) $\sigma_1 \approx_L \sigma_2$
(3) $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

$\mathcal{G} = \mathsf{SMS}$
Denote
$\sigma_1' = \mathsf{getStore}(\sigma_1, pc_1)$
$\sigma_2' = \mathsf{getStore}(\sigma_2, pc_2)$

Case I: $\mathcal{D}$ ends in SMS-MOVED
By assumption,

(I.1) $a_1'' :: A_1 = \sigma_1'(a_1).A$
(I.2) $a_1' = a_1''$
From (I.1) and (3),
(I.3) $a_2'' :: A_2 = \sigma_2'(a_2).A$ with
(I.4) $(a_1'' :: A_1) \approx_L^{\sigma_1,\sigma_2} (a_2'' :: A_2)$
From (I.4),
(I.5) $\mathcal{E}$ must end in SMS-MOVED
From (I.3) and (I.5),
(I.6) $a_2' = a_2''$
From (I.4), (I.2), (I.6), and the definition of $\approx_L^{\sigma,\sigma'}$ for A,
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

Case II: $\mathcal{D}$ ends in SMS-MOVED-S1
By assumption,
(II.1) $\sigma_1'(a_1).A = \cdot$
(II.2) $a_1' = \mathsf{NULL}$
From (I.1) and (3),
(II.3) $\sigma_2'(a_2).A = \cdot$
From (II.3),
(II.4) $\mathcal{E}$ ends in SMS-MOVED-S1
From (II.4),
(II.5) $a_2' = \mathsf{NULL}$
From (II.2) and (II.5),
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

Case III: $\mathcal{D}$ ends in SMS-MOVED-S2
By assumption,
(III.1) $a_1 = \mathsf{NULL}$
(III.2) $a_1' = \mathsf{NULL}$
From (3) and (III.1),
(III.3) $a_2 = \mathsf{NULL}$
From (III.2),
(III.4) $\mathcal{E}$ ends in SMS-MOVED-S2
From (III.4),
(III.5) $a_2' = \mathsf{NULL}$
From (III.2) and (III.5),
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

$\mathcal{G} = \mathsf{FS}$
From the structure of the operational semantics, pc_1, pc_2 is not $\cdot$

Case I: $\mathcal{D}$ ends in FS-MOVED
By assumption,
(I.1) $a_1'' :: A_1 = \sigma_1'(a_1).A \downarrow_{pc_1}$
(I.2) $a_1' = a_1''$
From (I.1) and (3),
(I.3) $a_2'' :: A_2 = \sigma_2'(a_2).A \downarrow_{pc_2}$ with
(I.4) $(a_1'' :: A_1) \approx_L^{\sigma_1,\sigma_2} (a_2'' :: A_2)$
From (I.4),
(I.5) $\mathcal{E}$ must end in FS-MOVED
From (I.3) and (I.5),
(I.6) $a_2' = a_2''$
From (I.4), (I.2), (I.6), and the definition of $\approx_L^{\sigma,\sigma'}$ for A^{FS},
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

Case II: $\mathcal{D}$ ends in FS-MOVED-S1
By assumption,
(II.1) $\sigma_1'(a_1).A \downarrow_{pc_1} = \cdot$
(II.2) $a_1' = \mathsf{NULL}$
From (I.1) and (3),
(II.3) $\sigma_2'(a_2).A \downarrow_{pc_2} = \cdot$
From (II.3),

(II.4) $\mathcal{E}$ ends in MOVED-S1
From (II.4),
(II.5) $a_2' = \mathsf{NULL}$
From (II.2) and (II.5),
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

Case III: $\mathcal{D}$ ends in FS-MOVED-S2
By assumption,
(III.1) $a_1 = \mathsf{NULL}$
(III.2) $a_1' = \mathsf{NULL}$
From (3) and (III.1),
(III.3) $a_2 = \mathsf{NULL}$
From (III.2),
(III.4) $\mathcal{E}$ ends in FS-MOVED-S2
From (III.4),
(III.5) $a_2' = \mathsf{NULL}$
From (III.2) and (III.5),
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

□

Lemma 55. *If* $\sigma_1 \approx_L \sigma_2$, $pc_1, pc_2 \sqsubseteq L$, *and* $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$, *then* $\mathsf{moveRight}_{\mathcal{G}}(\sigma_1, pc_1, a_1) \simeq_L^{\sigma_1,\sigma_2} \mathsf{moveRight}_{\mathcal{G}}(\sigma_2, pc_2, a_2)$

Proof.
Note that moveRight is defined only for the tree-structured EH storage.
We examine each case of $\mathcal{G}$
Denote $\mathcal{D} :: \mathsf{moveRight}_{\mathcal{G}}(\sigma_1, pc_1, a_1) = a_1'$ and
$\mathcal{E} :: \mathsf{moveRight}_{\mathcal{G}}(\sigma_2, pc_2, a_2) = a_2'$
By assumption,
(1) $pc_1, pc_2 \sqsubseteq L$
(2) $\sigma_1 \approx_L \sigma_2$
(3) $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

$\mathcal{G} = \mathsf{SMS}$
Denote
$\sigma_1' = \mathsf{getStore}(\sigma_1, pc_1)$
$\sigma_2' = \mathsf{getStore}(\sigma_2, pc_2)$

Case I: $\mathcal{D}$ ends in SMS-MOVER
By assumption,
(I.1) $a_{p,1} = \sigma_1'(a_1).a_p$
(I.2) $A_1 :: a_1 :: a_1'' :: A_1' = \sigma_1'(a_{p,1}).A$
(I.3) $a_1' = a_1''$
From (I.1) and (3),
(I.4) $a_{p,2} = \sigma_1'(a_2).a_p$ with
(I.5) $a_{p,1}.id = a_{p,2}.id$
From (2), (I.5), and since id's are unique,
(I.6) $a_{p,1} \approx_L^{\sigma_1,\sigma_2} a_{p,2}$
From (I.2) and (I.6),
(I.7) $A_2 :: a_2 :: a_2'' :: A_2' = \sigma_2'(a_{p,2}).A$ with
(I.8) $(A_1 :: a_1 :: a_1'' :: A_1') \approx_L^{\sigma_1,\sigma_2} (A_2 :: a_2 :: a_2'' :: A_2')$
From (I.4) and (I.7),
(I.9) $\mathcal{E}$ must end in SMS-MOVER
From (I.7) and (I.9),
(I.10) $a_2' = a_2''$
From (I.8), (I.3), (I.10), and the definition of $\approx_L^{\sigma,\sigma'}$ for A,
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

Case II: $\mathcal{D}$ ends in SMS-MOVER-S1
By assumption,
(II.1) $a_{p,1} = \sigma_1'(a_1).a_p$
(II.2) $A_1 :: a_1 = \sigma_1'(a_{p,1}).A$
(II.3) $a_1' = \mathsf{NULL}$

From (II.1) and (3),
(II.4) $a_{p,2} = \sigma_1'(a_2).a_p$ with
(II.5) $a_{p,1}.id = a_{p,2}.id$
From (2), (II.5), and since id's are unique,
(II.6) $a_{p,1} \approx_L^{\sigma_1,\sigma_2} a_{p,2}$
From (II.2) and (I.6),
(II.7) $A_2 :: a_2 = \sigma_2'(a_{p,2}).A$ with
From (II.4) and (II.7),
(II.8) $\mathcal{E}$ must end in SMS-MOVER-S1
From (II.8),
(II.9) $a_2' = \mathsf{NULL}$
From (II.3) and (II.9),
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

Case III: $\mathcal{D}$ ends in SMS-MOVER-S2
By assumption,
(III.1) $\mathsf{NULL} = \sigma_1'(a_1).a_p \vee \sigma(a).a_p \downarrow_{pc_l} = \cdot$
(III.2) $a_1' = \mathsf{NULL}$
From (III.1) and (3),
(III.3) $\mathsf{NULL} = \sigma_1'(a_2).a_p \vee \sigma(a).a_p \downarrow_{pc_l} = \cdot$
From (III.3),
(III.4) $\mathcal{E}$ must end in SMS-MOVER-S2
From (II.4),
(III.5) $a_2' = \mathsf{NULL}$
From (III.2) and (III.5),
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

Case IV: $\mathcal{D}$ ends in SMS-MOVER-S3
By assumption,
(IV.1) $a_1 = \mathsf{NULL}$
(IV.2) $a_1' = \mathsf{NULL}$
From (3) and (IV.1),
(IV.3) $a_2 = \mathsf{NULL}$
From (IV.2),
(IV.4) $\mathcal{E}$ ends in SMS-MOVER-S3
From (IV.4),
(IV.5) $a_2' = \mathsf{NULL}$
From (IV.2) and (IV.5),
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

$\mathcal{G} = \mathsf{FS}$
From the structure of the operational semantics, pc_1, pc_2 is not $\cdot$

Case I: $\mathcal{D}$ ends in FS-MOVED
By assumption,
(I.1) $a_{p,1} = \sigma_1'(a_1).a_p \downarrow_{pc_1}$
(I.2) $A_1 :: a_1 :: a_1'' :: A_1' = \sigma_1'(a_{p,1}).A \downarrow_{pc_1}$
(I.3) $a_1' = a_1''$
From (I.1) and (3),
(I.4) $a_{p,2} = \sigma_1'(a_2).a_p \downarrow_{pc_2}$ with
(I.5) $a_{p,1}.id = a_{p,2}.id$
From (2), (I.5), and since id's are unique,
(I.6) $a_{p,1} \approx_L^{\sigma_1,\sigma_2} a_{p,2}$
From (I.2) and (I.6),
(I.7) $A_2 :: a_2 :: a_2'' :: A_2' = \sigma_2'(a_{p,2}).A \downarrow_{pc_2}$ with
(I.8) $(A_1 :: a_1 :: a_1'' :: A_1') \approx_L^{\sigma_1,\sigma_2} (A_2 :: a_2 :: a_2'' :: A_2')$
From (I.4) and (I.7),
(I.9) $\mathcal{E}$ must end in FS-MOVER
From (I.7) and (I.9),
(I.10) $a_2' = a_2''$
From (I.8), (I.3), (I.10), and the definition of $\approx_L^{\sigma,\sigma'}$ for A,
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

Case II: $\mathcal{D}$ ends in FS-MOVER-S1
By assumption,
(II.1) $a_{p,1} = \sigma_1'(a_1).a_p \downarrow_{pc_1}$
(II.2) $A_1 :: a_1 = \sigma_1'(a_{p,1}).A \downarrow_{pc_1}$
(II.3) $a_1' = \mathsf{NULL}$
From (II.1) and (3),
(II.4) $a_{p,2} = \sigma_1'(a_2).a_p \downarrow_{pc_2}$ with
(II.5) $a_{p,1}.id = a_{p,2}.id$
From (2), (II.5), and since id's are unique,
(II.6) $a_{p,1} \approx_L^{\sigma_1,\sigma_2} a_{p,2}$
From (II.2) and (I.6),
(II.7) $A_2 :: a_2 = \sigma_2'(a_{p,2}).A \downarrow_{pc_2}$ with
From (II.4) and (II.7),
(II.8) $\mathcal{E}$ must end in FS-MOVER-S1
From (II.8),
(II.9) $a_2' = \mathsf{NULL}$
From (II.3) and (II.9),
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

Case III: $\mathcal{D}$ ends in FS-MOVER-S2
By assumption,
(III.1) $\mathsf{NULL} = \sigma_1'(a_1).a_p \downarrow_{pc_1}$
(III.2) $a_1' = \mathsf{NULL}$
From (III.1) and (3),
(III.3) $\mathsf{NULL} = \sigma_1'(a_2).a_p \downarrow_{pc_2}$ with
From (III.3),
(III.4) $\mathcal{E}$ must end in FS-MOVER-S2
From (II.4),
(III.5) $a_2' = \mathsf{NULL}$
From (III.2) and (III.5),
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

Case IV: $\mathcal{D}$ ends in FS-MOVER-S3
By assumption,
(IV.1) $a_1 = \mathsf{NULL}$
(IV.2) $a_1' = \mathsf{NULL}$
From (3) and (IV.1),
(IV.3) $a_2 = \mathsf{NULL}$
From (IV.2),
(IV.4) $\mathcal{E}$ ends in FS-MOVER-S3
From (IV.4),
(IV.5) $a_2' = \mathsf{NULL}$
From (IV.2) and (IV.5),
$a_1' \approx_L^{\sigma_1,\sigma_2} a_2'$

□

Lemma 56. *If $\sigma_1 \approx_L \sigma_2$ and $ks_1 \approx_L ks_2$ with $E_1 \approx_L E_2$ and $pc_1, pc_2 \sqsubseteq L$, and one of the following:*
1) When $G, \mathcal{P}, \sigma_1 \vdash ks_1; \mathsf{lookupEHAt}(id.Ev(v)) \rightsquigarrow_{pc_1} ks_1'$, and $G, \mathcal{P}, \sigma_2 \vdash ks_2; \mathsf{lookupEHAt}(id.Ev(v)) \rightsquigarrow_{pc_2} ks_2'$,
2) When $G, \mathcal{P}, \sigma_1 \vdash ks_1; \mathsf{lookupEHAll}(id.Ev(v)) \rightsquigarrow_{pc_1} ks_1'$, and $G, \mathcal{P}, \sigma_2 \vdash ks_2; \mathsf{lookupEHAll}(id.Ev(v)) \rightsquigarrow_{pc_2} ks_2'$,
3) When $G, \mathcal{P}, \sigma_1 \vdash ks_1; \mathsf{lookupEHAt}(id.Ev(v)) \rightsquigarrow_L ks_1'$ and $G, \mathcal{P}, \sigma_2 \vdash ks_2; \mathsf{lookupEHAll}(id.Ev(v)) \rightsquigarrow_\cdot ks_2'$, or
4) When $G, \mathcal{P}, \sigma_1 \vdash ks_1; \mathsf{lookupEHs}(E_1) \rightsquigarrow_{pc_1} ks_1'$, and $G, \mathcal{P}, \sigma_2 \vdash ks_2; \mathsf{lookupEHs}(E_2) \rightsquigarrow_{pc_2} ks_2'$,
then $ks_1' \approx_L ks_2'$

Proof.
By induction on the structure of $\mathcal{D} :: G, \mathcal{P}, \sigma_1 \vdash ks_1; \mathsf{lookupEHAPI}(...) \rightsquigarrow_{pc_1} ks_1'$ and
$\mathcal{E} :: G, \mathcal{P}, \sigma_2 \vdash ks_2; \mathsf{lookupEHAPI}(...) \rightsquigarrow_{pc_2} ks_2'$
By assumption,
(1) $\sigma_1 \approx_L \sigma_2$
(2) $ks_1 \approx_L ks_2$
(3) $pc_1, pc_2 \sqsubseteq L$
(4) $E_1 \approx_L E_2$

Case I: $\mathcal{D}$ and $\mathcal{E}$ end in LOOKUPEHAPI
$\mathcal{C}_1 \approx_L \mathcal{C}_2$ follows from Lemma 59, Lemma 60, or Lemma 61, depending on the APIs.
The conclusion follows from (2) and Lemma 57

Case II: $\mathcal{D}$ ends in LOOKUPEHS-S
By assumption,
(II.1) $E_1 = \cdot$
(II.2) $ks_1' = ks_1$

Subcase i: $E_2 = \cdot$
By assumption,
(i.1) $\mathcal{E}$ ends in LOOKUPEHS-S
From (i.1),
(i.2) $ks_2' = ks_2$
From (2), (II.2), and (i.2),
$ks_1' \approx_L ks_2'$

Subcase ii: $E_2 \neq \cdot$
By assumption,
(ii.1) $\mathcal{E}$ ends in LOOKUPEHS-R
From (4) and (II.1),
(ii.2) $E_2 \approx_L \cdot$
From (ii.1),
(ii.3) $G, \mathcal{P}, \sigma_2 \vdash ks_2; \mathsf{lookupEHs}(E_2) \rightsquigarrow_{pc_2} ks_2'$
From (ii.2), (ii.3), and Lemma 66 (Requirement **(EH2)**),
(ii.4) $ks_2 \approx_L ks_2'$
From (2), (II.2), and (ii.4),
$ks_1' \approx_L ks_2'$

Case III: $\mathcal{E}$ ends in LOOKUPEHS-S
The proof for this case is similar to **Case II**.

Case IV: $\mathcal{D}$ ends in LOOKUPEHS-R
By assumption,
(IV.1) $E_1 = (id.Ev(v), l_1), E_1'$
(IV.2) $ks_1 = (\mathcal{V}; \_; pc_1)$
(IV.3) $\mathcal{C}_1 = \mathsf{lookupEHs}_{G,\mathcal{V}}(\sigma_1, id.Ev(v), pc_1 \sqcup l_1)$
(IV.4) $ks_1'' = \mathsf{createK}(\mathcal{P}, id.Ev(v), \mathcal{C}_1)$
(IV.5) $\mathcal{D}' :: G, \mathcal{P}, \sigma_1 \vdash ks_1 :: ks_1''; \mathsf{lookupEHs}(E_1') \rightsquigarrow_{pc_1} ks_1'$

Subcase i: $l_1 \sqsubseteq L$
By assumption and from (4),
(i.1) $E_2 = (id'.Ev'(v'), l_2), E_2'$
From (i.1),
(i.2) $\mathcal{E}$ ends in LOOKUPEHS-R
From (i.2), (2), and (3),
(i.3) $ks_2 = (\mathcal{V}; \_; pc_2)$
(i.4) $\mathcal{C}_2 = \mathsf{lookupEHs}_{G,\mathcal{V}}(\sigma_2, id'.Ev'(v'), pc_2 \sqcup l_2)$
(i.5) $ks_2'' = \mathsf{createK}(\mathcal{P}, id'.Ev'(v'), \mathcal{C}_2)$
(i.6) $\mathcal{E}' :: G, \mathcal{P}, \sigma_2 \vdash ks_2 :: ks_2''; \mathsf{lookupEHs}(E_2') \rightsquigarrow_{pc_2} ks_2'$

Subsubcase a: $l_2 \sqsubseteq L$
By assumption and from (4),
(a.1) $id'.Ev'(v') = id.Ev(v)$ and
(a.2) $E_1' \approx_L E_2'$
By assumption and from (3),
(a.3) $pc_1 \sqcup l_1 \sqsubseteq L$ and $pc_2 \sqcup l_2 \sqsubseteq L$
From (1), (IV.3), (i.4), (a.1), (a.3), and Lemma 62,
(a.4) $\mathcal{C}_1 \approx_L \mathcal{C}_2$
From (a.4), (IV.4), (i.5), (a.1), and Lemma 57,
(a.5) $ks_1'' \approx_L ks_2''$
From (2) and (a.5),

(a.6) $(ks_1 :: ks_1'') \approx_L (ks_2 :: ks_2'')$
From (1), (3), (a.6), (a.2), (IV.5), and (i.6),
(a.7) IH may be applied on $\mathcal{D}'$ and $\mathcal{E}'$
From (a.7) and the IH on $\mathcal{D}'$ and $\mathcal{E}'$,
$ks_1' \approx_L ks_2'$

Subsubcase b: $l_2 \not\sqsubseteq L$
By assumption and from our security lattice,
(b.1) $pc_2 \sqcup l_2 = H$
(b.2) $E_2 \approx_L E_2'$
From (b.1), (i.4), and Lemma 69 (Requirement **(EH2)**),
(b.3) $\mathcal{C}_2 \approx_L \cdot$
From (b.3), (i.5), and Lemma 70 (Requirement **(EH2)**),
(b.4) $ks_2'' \approx_L \cdot$
From (b.4),
(b.5) $(ks_2 :: ks_2'') \approx_L ks_2$
From (b.2) and (4),
(b.6) $E_1 \approx_L E_2'$
From (b.5) and (2),
(b.7) $ks_1 \approx_L (ks_2 :: ks_2'')$
From (1), (3), (b.7), (b.6), and (i.6),
(b.8) IH may be applied on $\mathcal{D}$ and $\mathcal{E}'$
From (b.8),
$ks_1' \approx_L ks_2'$

Subcase ii: $l_1 \not\sqsubseteq L$
The proof for this case is similar to **Subsubcase i.b**. The conclusion follows from applying the IH on $\mathcal{D}'$ and $\mathcal{E}$.

Case V: $\mathcal{E}$ ends in LOOKUPEHS-R
The proof for this case is similar to the one for **Case IV**.

□

Lemma 57. *If* $\mathcal{C}_1 \approx_L \mathcal{C}_2$ *and* $ks_1 = \mathsf{createK}(\mathcal{P}, id.Ev(v), \mathcal{C}_1)$ *and* $ks_2 = \mathsf{createK}(\mathcal{P}, id.Ev(v), \mathcal{C}_2)$*, then* $ks_1 \approx_L ks_2$

Proof (sketch): The proof is by straightforward induction on the structure of $\mathcal{D} :: \mathsf{createK}(\mathcal{P}, id.Ev(v), \mathcal{C}_1)$ and $\mathcal{E} :: \mathsf{createK}(\mathcal{P}, id.Ev(v), \mathcal{C}_2)$. It uses Lemma 58 for L event handlers and Lemma 71 (Requirement **(EH2)**) for any H event handlers. □

Lemma 58. *If* $pc_1, pc_2 \sqsubseteq L$ *and* $ks_1 = \mathsf{crtK}_{\mathcal{V}}(eh, v, pc_1)$ *and* $ks_2 = \mathsf{crtK}_{\mathcal{V}}(eh, v, pc_2)$*, then* $ks_1 \approx_L ks_2$

Proof (sketch): This proof is straightforward. It uses the definition of $\approx_L$ for κ. □

Lemma 59. $\sigma_1 \approx_L \sigma_2$ *with* $pc_1, pc_2 \sqsubseteq L$*, then* $\mathsf{lookupEHAll}_G(\sigma_1, pc_1, id.Ev(v)) \approx_L \mathsf{lookupEHAll}_G(\sigma_2, pc_2, id.Ev(v))$

Proof.
By induction on the structure of
$\mathcal{E} :: \mathcal{C}_1 = \mathsf{lookupEHAll}_G(\sigma_1, id.Ev(v), pc_1)$ and
$\mathcal{D} :: \mathcal{C}_2 = \mathsf{lookupEHAll}_G(\sigma_2, id.Ev(v), pc_2)$
Want to show $\mathcal{C}_1 \approx_L \mathcal{C}_2$

By assumption,
(1) $\sigma_1 \approx_L \sigma_2$
(2) $pc_1, pc_2 \sqsubseteq L$

Case I: $\mathcal{D}$ ends in LOOKUPEHALL-S
By assumption,
(I.1) $\mathsf{lookup}_{G\downarrow_{EH}}(\sigma_1, pc_1, id) = \phi_1$
(I.2) $\mathsf{valOf}(\phi_1) = \mathsf{NULL}$
(I.3) $\mathcal{C}_1 = \cdot$

Subcase i: $pc_2 = L$
From (1), (2), (I.1), and Lemma 48.U (for the unstructured EH storage),
(i.1) $\mathsf{lookup}_{G\downarrow_{EH}}(\sigma_2, pc_2, id) = \phi_2 \approx_L \phi_1$ with $\mathsf{labOf}(\phi_1, pc_1) = \mathsf{labOf}(\phi_2, pc_2) = L$ or
(i.2) $\mathsf{lookup}_{G\downarrow_{EH}}(\sigma_2, pc_2, id) = \phi_2 \approx_L \phi_1$ with $\mathsf{labOf}(\phi_1, pc_1) = \mathsf{labOf}(\phi_2, pc_2) = H$
From (1), (2), (I.1), and Lemma 48.T (for the tree-structured EH storage),

(i.3) $\mathsf{lookup}_{G\downarrow_{EH}}(\sigma_2, pc_2, id) \approx_L^{\sigma_2,\sigma_1} \phi_1$

Subsubcase a: (i.1) or (i.3) are true
From (i.1) (for the unstructured EH storage) and (i.3) (for the tree-structured EH storage),
(a.1) $\mathcal{E}$ ends in LOOKUPEHALL-S
From (i.3),
(a.2) $\mathcal{C}_2 = \cdot$
From (I.3) and (a.2),
$\mathcal{C}_1 \approx_L \mathcal{C}_2$

Subsubcase b: (i.2) is true
From (i.2),
(b.1) $\mathcal{C}_1 \approx_L \cdot$
(b.2) $\mathcal{E}$ ends in LOOKUPEHALL or
(b.3) $\mathcal{E}$ ends in LOOKUPEHALL-S
If (b.2) is true, then from (i.2),
(b.4) $\mathcal{C}_2 \approx_L \cdot$
If (b.3) is true,
(b.5) $\mathcal{C}_2 = \cdot$
From (b.1), (b.40, and (b.5),
$\mathcal{C}_1 \approx_L \mathcal{C}_2$

Subcase ii: $pc_2 = \cdot$
By assumption,
(ii.1) $\mathcal{E}$ ends in LOOKUPEHALL-NC-MERGE
From (ii.1),
(ii.2) $\exists \mathcal{E}' :: \mathsf{lookupEHAll}_G(\sigma_2, H, id.Ev(v)) = \mathcal{C}_H$
(ii.3) $\exists \mathcal{E}'' :: \mathsf{lookupEHAll}_G(\sigma_2, L, id.Ev(v)) = \mathcal{C}_L$
(ii.4) $\mathcal{C}_2 = \mathsf{mergeC}(\mathcal{C}_H, \mathcal{C}_L)$
From (ii.2) and Lemma 67 (Requirement **(EH2)**),
(ii.5) $\mathcal{C}_H \approx_L \cdot$
IH on $\mathcal{D}$ and $\mathcal{E}''$ gives
(ii.6) $\mathcal{C}_L \approx_L \mathcal{C}_1$
From (ii.4)-(ii.6) and the definition of mergeC,
$\mathcal{C}_1 \approx_L \mathcal{C}_2$

Case II: $\mathcal{E}$ ends in LOOKUPEHALL-S
The proof for this case is similar to **Case I**.

Case III: $\mathcal{E}$ ends in LOOKUPEHALL
By assumption,
(III.1) $\mathsf{lookup}_{G\downarrow_{EH}}(\sigma_1, pc_1, id) = \phi_1$
(III.2) $\mathsf{valOf}(\phi_1) \neq \mathsf{NULL}$
(III.3) $\mathsf{labOf}(\phi_1, pc_1) = l_1$
(III.4) $\mathcal{C}_1 = (\phi_1.M(Ev) \downarrow_{pc_1}) \sqcup pc_1 \sqcup l_1$
(III.5) $pc_1 = L$
From (1), (2), (III.1), and Lemma 48.U (for the unstructured EH storage),
(III.6) $\mathsf{lookup}_{G\downarrow_{EH}}(\sigma_2, pc_2, id) = \phi_2 \approx_L \phi_1$ with $\mathsf{labOf}(\phi_1, pc_1) = \mathsf{labOf}(\phi_1, pc_2) = L$
(III.7) $\mathsf{lookup}_{G\downarrow_{EH}}(\sigma_2, pc_2, id) = \phi_2 \approx_L \phi_1$ with $\mathsf{labOf}(\phi_1, pc_1) = \mathsf{labOf}(\phi_1, pc_2) = H$
From (1), (2), (III.1), and Lemma 48.T (for the tree-structured EH storage),
(III.8) $\mathsf{lookup}_{G\downarrow_{EH}}(\sigma_2, pc_2, id) \approx_L^{\sigma_2,\sigma_1} \phi_1$

Subcase i: $pc_2 = L$ and (III.6) or (III.8) is true
From (III.6) and (III.8),
(i.1) $\mathcal{E}$ ends in LOOKUPEHALL
From (i.1),
(i.2) $\mathsf{labOf}(\phi_2, pc_2) = l_2$
(i.3) $\mathcal{C}_2 = (\phi_2.M(Ev) \downarrow_{pc_2}) \sqcup pc_2 \sqcup l_2$
(i.4) $pc_2 = L$
From (III.7), (III.3), and (i.2),
(i.5) $l_1 = l_2 = L$
From (III.6), (III.8), (III.5), (i.4), (III.4), (i.3), and the definition of $\downarrow_L$ for M (which projects L- and $\cdot$-labeled

event handlers to be labeled with L),
(i.6) $\mathcal{C}_1 = \phi_1.M(Ev)\downarrow_L$
(i.7) $\mathcal{C}_2 = \phi_2.M(Ev)\downarrow_L$
From (III.6), (III.8), (i.6), (i.7), and the definition of $\downarrow_L$ for M,
$\mathcal{C}_1 \approx_L \mathcal{C}_2$

Subcase ii: $pc_2 = \cdot$
From (III.8),
(ii.1) $\mathcal{E}$ ends in LOOKUPEHALL-NC-MERGE
The rest of the proof is similar to **Subcase I.ii**.

Subcase iii: (III.7) is true
The proof for this case is similar to **Subcase I.i.b**.

Case IV: $\mathcal{D}$ ends in LOOKUPEHALL
The proof for this case is similar to the one for **Case III**.

Case V: $\mathcal{E}$ ends in LOOKUPEHALL-NC-MERGE
By assumption,
(V.1) $\mathcal{E}' :: \mathcal{C}_H = \mathsf{lookupEHAll}_G(\sigma_1, H, id.Ev(v))$
(V.2) $\mathcal{E}'' :: \mathcal{C}_L = \mathsf{lookupEHAll}_G(\sigma_1, L, id.Ev(v))$
(V.3) $\mathcal{C}_1 = \mathsf{mergeC}(\mathcal{C}_H, \mathcal{C}_L)$
From (V.1) and Lemma 67 (Requirement **(EH2)**),
(V.4) $\mathcal{C}_H \approx_L \cdot$
IH on $\mathcal{D}$ and $\mathcal{E}''$ gives
(V.5) $\mathcal{C}_L \approx_L \mathcal{C}_1$
From (V.3)-(V.5) and the definition of mergeC,
$\mathcal{C}_1 \approx_L \mathcal{C}_2$

Case VI: $\mathcal{D}$ ends in LOOKUPEHALL-NC-MERGE
The proof is similar to **Case V**

□

Lemma 60. *$\sigma_1 \approx_L \sigma_2$ and $pc_1, pc_2 \sqsubseteq L$, then* $\mathsf{lookupEHAt}_{\mathcal{G}}(\sigma_1, pc_1, id.Ev(v_1)) \approx_L \mathsf{lookupEHAt}_{\mathcal{G}}(\sigma_2, pc_2, id.Ev(v_2))$

Proof (sketch): The proof is similar to the proof for Lemma 59. The biggest difference is that lookupEHAt uses the @ operator instead of the $\downarrow$ operator to build up $\mathcal{C}$. But $(M(Ev)@L) \sqcup L$ returns the same thing as $M(Ev)\downarrow_L$, so this does not matter. □

Lemma 61. *$\sigma_1 \approx_L \sigma_2$, then* $\mathsf{lookupEHAt}_{\mathcal{G}}(\sigma_1, L, id.Ev(v)) \approx_L \mathsf{lookupEHAll}_{\mathcal{G}}(\sigma_2, \cdot, id.Ev(v))$

Proof (sketch): The proof for this case is similar to Lemma 59 and uses the fact that $(M(Ev)@L) \sqcup L$ returns the same thing as $M(Ev)\downarrow_L$ and Lemma 67 (Requirement **(EH2)**) is used to show that $\mathsf{lookupEHAll}_{\mathcal{G}}(\sigma_2, \cdot, id.Ev(v)) \approx_L \mathsf{lookupEHAll}_{\mathcal{G}}(\sigma_2, L, id.Ev(v))$. □

Lemma 62. *$\sigma_1 \approx_L \sigma_2$ and $pc_1, pc_2 \sqsubseteq L$, then* $\mathsf{lookupEHs}_{\mathcal{G}}(\sigma_1, pc_1, id.Ev(v)) \approx_L \mathsf{lookupEHs}_{\mathcal{G}}(\sigma_2, pc_2, id.Ev(v))$

Proof (sketch): Follows from Lemma 59 and Lemma 60. □

Lemma 63. *If $\sigma_1 \approx_L \sigma$, $pc_1, pc_2 \sqsubseteq L$, and $v_1 \approx_L v_2$ then* $\mathsf{triggerEH}_{\mathcal{G}}(\sigma_1, pc_1, id, Ev, v_1) \approx_L \mathsf{triggerEH}_{\mathcal{G}}(\sigma_2, pc_2, id, Ev, v_2)$

Proof.
By induction on the structure of $\mathcal{E} :: \mathsf{triggerEH}_{\mathcal{G}}(\sigma_1, pc_1, id, Ev, v_1)$ and $\mathcal{D} :: \mathsf{triggerEH}_{\mathcal{G}}(\sigma_2, pc_2, id, Ev, v_2)$
Denote
$\mathsf{triggerEH}_{\mathcal{G}}(\sigma_1, pc_1, id, Ev, v_1) = E_1$
$\mathsf{triggerEH}_{\mathcal{G}}(\sigma_2, pc_2, id, Ev, v_2) = E_2$
Want to show $E_1 \approx_L E_2$
By assumption,
(1) $\sigma_1 \approx_L \sigma_2$
(2) $pc_1, pc_2 \sqsubseteq L$,
(3) $v_1 \approx_L v_2$

$\mathcal{G} = \mathsf{SMS}$
Unstructured EH storage:
Case U.I: $\mathcal{E}$ or $\mathcal{D}$ ends in SMS-TRIGGEREH
Without loss of generality, assume $\mathcal{D}$ ends in SMS-TRIGGEREH. The proof for $\mathcal{E}$ ending in SMS-TRIGGEREH is similar.

By assumption,
(U.I.1) $\phi_1 = \mathsf{lookup}_{G\downarrow_{EH}}(\sigma_1, pc_1, id) \neq \mathsf{NULL}$
(U.I.2) $E_1 = (id.Ev(v_1), pc_1)$
From (1), (2), (U.I.1), and Lemma 48.U,
(U.I.3) $\phi_2 = \mathsf{lookup}_{G\downarrow_{EH}}(\sigma_2, pc_2, id)$ with
(U.I.4) $\phi_1 \approx_L \phi_2$
From (U.I.1) and (U.I.4),
(U.I.5) $\phi_2 \neq \mathsf{NULL}$
From (U.I.3) and (U.I.5),
(U.I.6) $\mathcal{E}$ ends in SMS-TRIGGEREH
From (U.I.6),
(U.I.7) $E_2 = (id.Ev(v_2), pc_2)$
From (3), (U.I.2), and (U.I.7),
$E_1 \approx_L E_2$

Case U.II: $\mathcal{E}$ or $\mathcal{D}$ ends in SMS-TRIGGEREH-S
Without loss of generality, assume $\mathcal{D}$ ends in SMS-TRIGGEREH-S. The proof for $\mathcal{E}$ ending in SMS-TRIGGEREH-S is similar.
By assumption,
(U.II.1) $\phi_1 = \mathsf{lookup}_{\mathcal{G}}(\sigma_1, pc_1, id) = \mathsf{NULL}$
(U.II.2) $E_1 = \cdot$
From (1), (2), (U.II.1), and Lemma 48.U,
(U.II.3) $\phi_2 = \mathsf{lookup}_{G\downarrow_{EH}}(\sigma_2, pc_2, id)$ with
(U.II.4) $\phi_1 \approx_L \phi_2$
From (U.II.1) and (U.II.4),
(U.II.5) $\phi_2 = \mathsf{NULL}$
From (U.II.3) and (U.II.5),
(U.II.6) $\mathcal{E}$ ends in SMS-TRIGGEREH-S
From (U.II.6),
(U.II.7) $E_2 = \cdot$
From (U.II.2) and (U.II.7),
$E_1 \approx_L E_2$

Case U.III: $\mathcal{E}$ or $\mathcal{D}$ ends in SMS-TRIGGEREH-NC
Without loss of generality, assume$\mathcal{D}$ ends in SMS-TRIGGEREH-NC. The proof for $\mathcal{E}$ ending in SMS-TRIGGEREH-NC is similar.
By assumption,
(U.III.1) $E_{H,1} = \mathsf{triggerEH}_{\mathsf{SMS}}(\sigma_1, H, id, Ev, \mathsf{getFacet}(v_1, H))$
(U.III.2) $\mathcal{D}' :: E_{L,1} = \mathsf{triggerEH}_{\mathsf{SMS}}(\sigma_1, L, id, Ev, \mathsf{getFacet}(v_1, L))$
(U.III.3) $E_1 = \mathsf{mergeEvs}(E_H, E_L)$
From the definition of getFacet,
The IH may be applied on $\mathcal{D}'$ and $\mathcal{E}$
From IH on $\mathcal{D}'$ and $\mathcal{E}$,
(U.III.4) $E_{L,1} \approx_L E_2$
From (U.III.1) and Lemma 72 (Req **(EH2)**),
(U.III.5) $E_{H,1} \approx_L \cdot$
From (U.III.3)-(U.III.5) and the definition of mergeEvs,
$E_1 \approx_L E_2$

Tree-structured EH storage:
Denote
$\sigma_1 = \sigma_{H,1}, \sigma_{L,1}$ and
$\sigma_2 = \sigma H, 2, \sigma_{L,2}$
From (1),
(T.1) $\sigma_{L,1}(a_1^{\mathsf{rt}}) \approx_L^{\sigma_1,\sigma_2} \sigma_{L,2}(a_2^{\mathsf{rt}})$

Case T.I: $\mathcal{E}$ or $\mathcal{D}$ ends in SMS-TRIGGEREH
Without loss of generality, assume $\mathcal{D}$ ends in SMS-TRIGGEREH. The proof for $\mathcal{E}$ ending in SMS-TRIGGEREH is similar.
By assumption,
(T.I.1) $a_1 = \mathsf{lookupA}_{G\downarrow_{EH}}(\sigma_1, pc_1, id, a_1^{\mathsf{rt}}) \neq \mathsf{NULL}$
(T.I.2) $E_1 = (id.Ev(v_1), pc_1)$
From (1), (2), (T.1), (T.I.1), and Lemma 49,

(T.I.3) $a_2 = \mathsf{lookupA}_{G\downarrow_{EH}}(\sigma_2, pc_2, id, a_2^{\mathsf{rt}})$ with
(T.I.4) $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$
From (T.I.1) and (T.I.4),
(T.I.5) $a_2 \neq \mathsf{NULL}$
From (T.I.3) and (T.I.5),
(T.I.6) $\mathcal{E}$ ends in SMS-TRIGGEREH
From (T.I.6),
(T.I.7) $E_2 = (id.Ev(v_2), pc_2)$
From (3), (T.I.2), and (T.I.7),
$E_1 \approx_L E_2$

Case T.II: $\mathcal{E}$ or $\mathcal{D}$ ends in SMS-TRIGGEREH-S
Without loss of generality, assume $\mathcal{D}$ ends in SMS-TRIGGEREH-S. The proof for $\mathcal{E}$ ending in SMS-TRIGGEREH-S is similar.
By assumption,
(T.II.1) $a_1 = \mathsf{lookupA}_{\mathcal{G}}(\sigma_1, pc_1, id, a_1^{\mathsf{rt}}) = \mathsf{NULL}$
(T.II.2) $E_1 = \cdot$
From (1), (2), (T.1), (T.II.1), and Lemma 49,
(T.II.3) $a_2 = \mathsf{lookupA}_{G\downarrow_{EH}}(\sigma_2, pc_2, id, a_2^{\mathsf{rt}})$ with
(T.II.4) $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$
From (T.II.1) and (T.II.4),
(T.II.5) $a_2 = \mathsf{NULL}$
From (T.II.3) and (T.II.5),
(T.II.6) $\mathcal{E}$ ends in SMS-TRIGGEREH-S
From (T.II.6),
(T.II.7) $E_2 = \cdot$
From (T.II.2) and (T.II.7),
$E_1 \approx_L E_2$

Case T.III: $\mathcal{E}$ or $\mathcal{D}$ ends in SMS-TRIGGEREH-NC
Without loss of generality, assume$\mathcal{D}$ ends in SMS-TRIGGEREH-NC. The proof for $\mathcal{E}$ ending in SMS-TRIGGEREH-NC is similar.
By assumption,
(T.III.1) $E_{H,1} = \mathsf{triggerEH}_{\mathsf{SMS}}(\sigma_1, H, id, Ev, \mathsf{getFacet}(v_1, H))$
(T.III.2) $\mathcal{D}' :: E_{L,1} = \mathsf{triggerEH}_{\mathsf{SMS}}(\sigma_1, L, id, Ev, \mathsf{getFacet}(v_1, L))$
(T.III.3) $E_1 = \mathsf{mergeEvs}(E_H, E_L)$
From the definition of getFacet,
The IH may be applied on $\mathcal{D}'$ and $\mathcal{E}$
From IH on $\mathcal{D}'$ and $\mathcal{E}$,
(T.III.4) $E_{L,1} \approx_L E_2$
From (T.III.1) and Lemma 72 (Req **(EH2)**),
(T.III.5) $E_{H,1} \approx_L \cdot$
From (T.III.3)-(T.III.5) and the definition of mergeEvs,
$E_1 \approx_L E_2$

$\mathcal{G} = \mathsf{FS}$
Unstructured EH storage:
Case U.I: $\mathcal{D}$ ends in FS-TRIGGEREH
By assumption,
(U.I.1) $\phi_1 = \mathsf{lookup}_{\mathsf{FS}}(\sigma_1, pc_1, id) \neq \langle \_|\_ \rangle \neq \mathsf{NULL}$
(U.I.2) $E_1 = (id.Ev(v_1), pc_1)$
From (1), (2), (U.I.1), and Lemma 48.U,
(U.I.3) $\phi_2 = \mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id)$ with
(U.I.4) $\phi_1 \approx_L \phi_2$
From (U.I.1) and (U.I.4), either
(U.I.5) $\phi_2 = \langle \_|\_ \rangle$ or $v_2 = \langle \_|\_ \rangle$
(U.I.6) $\phi_2 \neq \langle \_|\_ \rangle$ and $\phi_2 \neq \mathsf{NULL}$ and $v_2 \neq \langle \_|\_ \rangle$

Subcase i: (U.I.5) is true
From (U.I.5),
(i.1) $\mathcal{E}$ ends in FS-TRIGGEREH-NC
From (i.1),

(i.2) $E_H = \mathsf{triggerEH_{FS}}(\sigma_2, H, id, Ev, \mathsf{getFacetV}(v, H))$
(i.3) $\exists \mathcal{E}' :: E_L = \mathsf{triggerEH_{FS}}(\sigma_2, L, id, Ev, \mathsf{getFacetV}(v, L))$
(i.4) $E_2 = \mathsf{mergeEvs}(E_H, E_L)$
From (1), (2), and the definition of getFacetV,
the IH may be applied to $\mathcal{D}$ and $\mathcal{E}'$
From IH on $\mathcal{D}$ and $\mathcal{E}'$,
(i.5) $E_1 \approx_L E_L$
From (i.2) and Lemma 72 (Req **(EH2)**),
(i.6) $E_H \approx_L \cdot$
From (i.4)-(i.6), and the definition of mergeEvs,
$E_1 \approx_L E_2$

Subcase ii: (U.I.6) is true
From (U.I.6),
(ii.1) $\mathcal{E}$ ends in FS-TRIGGEREH
From (ii.1),
(ii.2) $E_2 = (id.Ev(v_2), pc_2)$
From (3), (U.I.2), and (ii.2),
$E_1 \approx_L E_2$

Case U.II: $\mathcal{D}$ ends in FS-TRIGGEREH-NC
By assumption,
(U.II.1) $E_H = \mathsf{triggerEH_{FS}}(\sigma_1, H, id, Ev, \mathsf{getFacetV}(v, H))$
(U.II.2) $\exists \mathcal{D}' :: E_L = \mathsf{triggerEH_{FS}}(\sigma_1, L, id, Ev, \mathsf{getFacetV}(v, L))$
(U.II.3) $E_1 = \mathsf{mergeEvs}(E_H, E_L)$
From (1), (2), and the definition of getFacetV,
the IH may be applied to $\mathcal{D}'$ and $\mathcal{E}$
From IH on $\mathcal{D}'$ and $\mathcal{E}$,
(U.II.4) $E_L \approx_L E_2$
From (U.II.1) and Lemma 72 (Req **(EH2)**),
(U.II.5) $E_H \approx_L \cdot$
From (U.II.3)-(U.II.5), and the definition of mergeEvs,
$E_1 \approx_L E_2$

Case U.III: $\mathcal{D}$ ends in FS-TRIGGEREH-S
By assumption,
(U.III.1) $\mathsf{lookup_{FS}}(\sigma_1, pc_1, id) = \mathsf{NULL}$
(U.III.2) $E_1 = \cdot$
From (1), (2), (U.III.1), and Lemma 48.U,
(U.III.3) $\phi_2 = \mathsf{lookup_{FS}}(\sigma_2, pc_2, id)$ with
(U.III.4) $\mathsf{NULL}_1 \approx_L \phi_2$
From (U.III.1) and (U.III.4), either
(U.III.5) $\phi_2 = \langle \_ | \_ \rangle$ or $v_2 = \langle \_ | \_ \rangle$
(U.III.6) $\phi_2 = \mathsf{NULL}$ and $v_2 \neq \langle \_ | \_ \rangle$

Subcase i: (U.III.5) is true
From (U.III.5),
(i.1) $\mathcal{E}$ ends in FS-TRIGGEREH-NC
From (i.1),
(i.2) $E_H = \mathsf{triggerEH_{FS}}(\sigma_2, H, id, Ev, \mathsf{getFacetV}(v, H))$
(i.3) $\exists \mathcal{E}' :: E_L = \mathsf{triggerEH_{FS}}(\sigma_2, L, id, Ev, \mathsf{getFacetV}(v, L))$
(i.4) $E_2 = \mathsf{mergeEvs}(E_H, E_L)$
From (1), (2), and the definition of getFacetV,
the IH may be applied to $\mathcal{D}$ and $\mathcal{E}'$
From IH on $\mathcal{D}$ and $\mathcal{E}'$,
(i.5) $E_1 \approx_L E_L \approx_L \cdot$
From (i.2) and Lemma 72 (Req **(EH2)**),
(i.6) $E_H \approx_L \cdot$
From (U.III.2), (i.4)-(i.6), and the definition of mergeEvs,
$E_1 \approx_L E_2$

Subcase ii: (U.III.6) is true
By assumption,

(ii.1) $\mathcal{E}$ ends in FS-TRIGGERWEH-S
From (ii.1),
(ii.2) $E_2 = \cdot$
From (U.III.2) and (ii.2),
$E_1 \approx_L E_2$

Tree-structured EH storage
From (1),
(T.1) $\sigma_{L,1}(a_1^{\mathsf{rt}}) \approx_L^{\sigma_1,\sigma_2} \sigma_{L,2}(a_2^{\mathsf{rt}})$

Case T.I: $\mathcal{D}$ ends in FS-TRIGGERWEH
By assumption,
(T.I.1) $a_1 = \mathsf{lookupA}_{\mathsf{FS}}(\sigma_1, pc_1, id, a_1^{\mathsf{rt}}) \neq \langle \_|\_ \rangle \neq \mathsf{NULL}$
(T.I.2) $E_1 = (id.Ev(v_1), pc_1)$
From (1), (2), (T.1), (T.I.1), and Lemma 49,
(T.I.3) $a_2 = \mathsf{lookupA}_{\mathsf{FS}}(\sigma_2, pc_2, id, a_2^{\mathsf{rt}})$ with
(T.I.4) $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$
From (T.I.1) and (T.I.4), either
(T.I.5) $a_2 = \langle \_|\_ \rangle$ or $v_2 = \langle \_|\_ \rangle$
(T.I.6) $a_2 \neq \langle \_|\_ \rangle$ and $a_2 \neq \mathsf{NULL}$ and $v_2 \neq \langle \_|\_ \rangle$

Subcase i: (T.I.5) is true
From (T.I.5),
(i.1) $\mathcal{E}$ ends in FS-TRIGGERWEH-NC
From (i.1),
(i.2) $E_H = \mathsf{triggerEH}_{\mathsf{FS}}(\sigma_2, H, id, Ev, \mathsf{getFacetV}(v, H))$
(i.3) $\exists \mathcal{E}' :: E_L = \mathsf{triggerEH}_{\mathsf{FS}}(\sigma_2, L, id, Ev, \mathsf{getFacetV}(v, L))$
(i.4) $E_2 = \mathsf{mergeEvs}(E_H, E_L)$
From (1), (2), and the definition of getFacetV,
the IH may be applied to $\mathcal{D}$ and $\mathcal{E}'$
From IH on $\mathcal{D}$ and $\mathcal{E}'$,
(i.5) $E_1 \approx_L E_L$
From (i.2) and Lemma 72 (Req **(EH2)**),
(i.6) $E_H \approx_L \cdot$
From (i.4)-(i.6), and the definition of mergeEvs,
$E_1 \approx_L E_2$

Subcase ii: (T.I.6) is true
From (T.I.6),
(ii.1) $\mathcal{E}$ ends in FS-TRIGGERWEH
From (ii.1),
(ii.2) $E_2 = (id.Ev(v_2), pc_2)$
From (3), (T.I.2), and (ii.2),
$E_1 \approx_L E_2$

Case T.II: $\mathcal{D}$ ends in FS-TRIGGERWEH-NC
By assumption,
(T.II.1) $E_H = \mathsf{triggerEH}_{\mathsf{FS}}(\sigma_1, H, id, Ev, \mathsf{getFacetV}(v, H))$
(T.II.2) $\exists \mathcal{D}' :: E_L = \mathsf{triggerEH}_{\mathsf{FS}}(\sigma_1, L, id, Ev, \mathsf{getFacetV}(v, L))$
(T.II.3) $E_1 = \mathsf{mergeEvs}(E_H, E_L)$
From (1), (2), and the definition of getFacetV,
the IH may be applied to $\mathcal{D}'$ and $\mathcal{E}$
From IH on $\mathcal{D}'$ and $\mathcal{E}$,
(T.II.4) $E_L \approx_L E_2$
From (T.II.1) and Lemma 72 (Req **(EH2)**),
(T.II.5) $E_H \approx_L \cdot$
From (T.II.3)-(T.II.5), and the definition of mergeEvs,
$E_1 \approx_L E_2$

Case T.III: $\mathcal{D}$ ends in FS-TRIGGERWEH-S
By assumption,
(T.III.1) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_1, pc_1, id, a_1^{\mathsf{rt}}) = \mathsf{NULL}$
(T.III.2) $E_1 = \cdot$

From (1), (2), (T.III.1), (T.1), and Lemma 49,
(T.III.3) $a_2 = \mathsf{lookupA}_{\mathsf{FS}}(\sigma_2, pc_2, id, a_2^{\mathsf{rt}})$ with
(T.III.4) $\mathsf{NULL} \approx_L^{\sigma_1,\sigma_2} a_2$
From (T.III.1) and (T.III.4), either
(T.III.5) $a_2 = \langle \_ | \_ \rangle$ or $v_2 = \langle \_ | \_ \rangle$
(T.III.6) $a_2 = \mathsf{NULL}$ and $v_2 \neq \langle \_ | \_ \rangle$

Subcase i: (T.III.5) is true
From (T.III.5),
(i.1) $\mathcal{E}$ ends in FS-TRIGGEREH-NC
From (i.1),
(i.2) $E_H = \mathsf{triggerEH}_{\mathsf{FS}}(\sigma_2, H, id, Ev, \mathsf{getFacetV}(v, H))$
(i.3) $\exists \mathcal{E}' :: E_L = \mathsf{triggerEH}_{\mathsf{FS}}(\sigma_2, L, id, Ev, \mathsf{getFacetV}(v, L))$
(i.4) $E_2 = \mathsf{mergeEvs}(E_H, E_L)$
From (1), (2), and the definition of getFacetV,
the IH may be applied to $\mathcal{D}$ and $\mathcal{E}'$
From IH on $\mathcal{D}$ and $\mathcal{E}'$,
(i.5) $E_1 \approx_L E_L \approx_L \cdot$
From (i.2) and Lemma 72 (Req **(EH2)**),
(i.6) $E_H \approx_L \cdot$
From (T.III.2), (i.4)-(i.6), and the definition of mergeEvs,
$E_1 \approx_L E_2$

Subcase ii: (T.III.6) is true
By assumption,
(ii.1) $\mathcal{E}$ ends in FS-TRIGGEREH-S
From (ii.1),
(ii.2) $E_2 = \cdot$
From (T.III.2) and (ii.2),
$E_1 \approx_L E_2$

$\mathcal{G} = \mathsf{TS}$
Note that only the unstructured EH storage is defined for the tainted store.

Case I: $\mathcal{D}$ ends in TS-TRIGGEREH
By assumption,
(I.1) $(\_, \_, l_{\phi,1}) = \mathsf{lookup}_{\mathsf{TS}}(\sigma_1, pc_1, id)$
(I.2) $l_1 = \mathsf{labOf}(v_1, pc_1)$
(I.3) $E_1 = (id.Ev(v_1), pc_1 \sqcup l_{\phi,1} \sqcup l_1)$
From (2) and our security lattice,
(I.4) $(pc_1 \sqcup l_{\phi_1} \sqcup l_1) = (l_{\phi_1} \sqcup l_1)$
From (1), (2), (I.1), and Lemma 48.U,
(I.5) $\phi_2 = \mathsf{lookup}_{\mathsf{TS}}(\sigma_2, pc_2, id)$ with
(I.6) $\phi_2 \approx_L (\_, \_, l_{\phi,1})$

Subcase i: $l_{\phi_1} \sqsubseteq L$ and $l_1 \sqsubseteq L$ and $pc_2 = L$
By assumption and from (3),
(i.1) $l_2 = \mathsf{labOf}(v_2)$ and $l_2 \sqsubseteq L$
By assumption and from (I.6),
(i.2) $\phi_2 = (\_, \_, l_{\phi,2})$ with
(i.3) $l_{\phi,2} \sqsubseteq L$
By assumption and from (i.1) and (i.2),
(i.4) $\mathcal{E}$ ends in TS-TRIGGEREH
From (i.4),
(i.5) $E_2 = (id.Ev(v_2), pc_2 \sqcup l_{\phi,2} \sqcup l_2)$
By assumption and from (i.1), (i.3), and our security lattice,
(i.6) $l_{\phi_1} \sqcup l_1 = L$ and
(i.7) $pc_2 \sqcup l_{\phi,2} \sqcup l_2 = L$
From (I.3), (i.5), (I.4), (i.6), and (i.7),
$E_1 \approx_L E_2$

Subcase ii: $l_{\phi_1} \sqsubseteq L$ and $l_1 \not\sqsubseteq L$ and $pc_2 = L$
By assumption and from (3),

(ii.1) $l_2 = \mathsf{labOf}(v_2)$ and $l_2 \not\sqsubseteq L$
By assumption and from (I.6),
(ii.2) $\phi_2 = (\_, \_, l_{\phi,2})$ with
(ii.3) $l_{\phi,2} \sqsubseteq L$
By assumption and from (ii.1) and (ii.2),
(ii.4) $\mathcal{E}$ ends in TS-TRIGGEREH
From (ii.4),
(ii.5) $E_2 = (id.Ev(v_2), pc_2 \sqcup l_{\phi,2} \sqcup l_2)$
By assumption and from (ii.1), (ii.3), and our security lattice,
(ii.6) $l_{\phi_1} \sqcup l_1 = H$ and
(ii.7) $pc_2 \sqcup l_{\phi,2} \sqcup l_2 = H$
From (I.3), (ii.5), (I.4), (ii.6), and (ii.7),
$E_1 \approx_L E_2$

Subcase iii: $l_{\phi_1} \not\sqsubseteq L$ and $pc_2 = L$
By assumption and from (I.6),
(iii.1) $\phi_2 = (\_, \_, H)$ or
(iii.2) $\phi_2 = (\mathsf{NULL}, H)$
By assumption and from (I.3),
(iii.3) $E_1 = (id.Ev(v_1), H)$

Subsubcase a: (iii.1) is true
By assumption and from (iii.1),
(a.1) $\mathcal{E}$ ends in TS-TRIGGEREH
From (a.1), (iii.1), and our security lattice,
(a.2) $E_2 = (id.Ev(v_2), H)$
From (a.2) and (iii.3),
$E_1 \approx_L E_2$

Subsubcase b: (iii.2) is true
By assumption and from (iii.2),
(b.1) $\mathcal{E}$ ends in TS-TRIGGEREH-S
From (b.1),
(b.2) $E_2 = \cdot$
From (b.2) and (iii.3),
$E_1 \approx_L E_2$

Subcase iv: $pc_2 = \cdot$
By assumption,
(iv.1) $\mathcal{E}$ ends in TS-TRIGGEREH-NC
From (iv.1),
(iv.2) $E_H = \mathsf{triggerEH}(\sigma_2, H, id, Ev, \mathsf{getFacetV}(v_2, H))$
(iv.2) $\mathcal{E}' :: E_L = \mathsf{triggerEH}(\sigma_2, L, id, Ev, \mathsf{getFacetV}(v_2, L))$
(iv.3) $E_2 = \mathsf{mergeEvs}(E_H, E_L)$
From (3) and the definition of $\mathsf{getFacetV}$,
the IH may be applied on $\mathcal{D}$ and $\mathcal{E}'$
From IH on $\mathcal{D}$ and $\mathcal{E}'$,
(iv.4) $E_1 \approx_L E_L$
From (iv.2) and Lemma 72 (Req **(EH2)**),
(iv.5) $E_H \approx_L \cdot$
From (iv.3)-(iv.5), and the definition of $\mathsf{mergeEvs}$,
$E_1 \approx_L E_2$

Case II: $\mathcal{D}$ ends in TS-TRIGGEREH-NC
The proof for this case is similar to **Subcase I.iv**.

Case III: $\mathcal{D}$ ends in TS-TRIGGEREH-S
By assumption,
(III.1) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_1, pc_1, id) = (\mathsf{NULL}, l_1)$
(III.2) $E_1 = \cdot$
From (1), (2), (III.1), and Lemma 48.U,
(III.3) $\phi_2 = \mathsf{lookup}_{\mathsf{TS}}(\sigma_2, pc_2, id)$ with
(III.4) $\phi_2 \approx_L (\mathsf{NULL}, l_1)$

Subcase i: $l_1 \sqsubseteq L$ and $pc_2 = L$
By assumption and from (III.4),
(i.1) $\phi_2 = (\mathsf{NULL}, l_2)$
From (i.1),
(i.2) $E_2 = \cdot$
From (III.2) and (i.2),
$E_1 \approx_L E_2$

Subcase ii: $l_1 \not\sqsubseteq L$ and $pc_2 = L$
By assumption and from (III.4),
(i.1) $\phi_2 = (\mathsf{NULL}, l_2)$ or
(i.2) $\phi_2 = (\_, \_, l_2)$

Subsubcase a: (i.1) is true
The proof for this case is similar to **Subcase i**.

Subsubcase b: (i.2) is true
The proof for this case is similar to **Subcase I.iii**.

Subcase iii: $pc_2 = \cdot$
The proof for this case is similar to **Subcase I.iv**.

□

Requirement (WEH1) L lookups are equivalent

Lemma 64. *If* $\sigma_1 \approx_L \sigma_2$ *and* $pc_1, pc_2 \sqsubseteq L$ *and* $\forall i \in [1, n], v_{i,1} \approx_L v_{i,2}$ *with* $v_{i,1} \downarrow_{pc_1} = v_{i,1}$, $v_{i,2} \downarrow_{pc_2} = v_{i,2}$, *and* $t_1 = \mathsf{ehAPI}_{\mathcal{I}}(\sigma_1, pc_1, id, v_{1,1}, \cdots, v_{n,1})$ *and* $t_2 = \mathsf{ehAPI}_{\mathcal{I}}(\sigma_2, pc_2, id, v_{1,2}, \cdots, v_{n,2})$, *then* $t_1 \approx_L t_2$

Proof (sketch): The proof is similar to the one for Lemma 46 (Requirement **(EH1)**). It uses Lemma 48 (Requirement **(EH1)**) and Lemma 65 instead of Lemma 50. □

Lemma 65. *If* $pc_1, pc_2 \sqsubseteq L$ *and* $\phi_1 \approx_L \phi_2$, *with* $\mathsf{getValG}_{\mathcal{G}}(pc_1, \phi_1) = v_1$ *and* $\mathsf{getValG}_{\mathcal{G}}(pc_1, \phi_2) = v_2$, *then* $v_1 \approx_L v_2$

Proof.
Only the cases for $\mathcal{G} = \mathsf{TS}$ are considered. The other cases follow from Lemma 50 (Requirement **(EH1)**).
Denote $\mathcal{D} :: \mathsf{getValG}_{\mathcal{G}}(pc_1, \phi_1) = v_1$ and $\mathcal{E} :: \mathsf{getValG}_{\mathcal{G}}(pc_1, \phi_2) = v_2$
We examine each case of $\mathcal{D}$
By assumption,
(1) $pc_1, pc_2 \sqsubseteq L$
(2) $\phi_1 \approx_L \phi_2$

Case I: $\mathcal{D}$ ends in TS-GETVALG
By assumption,
(I.1) $\mathsf{valOf}(\phi_1) \neq \mathsf{NULL}$
(I.2) $l_{\phi,1} = \mathsf{labOf}(\phi_1, pc_1)$
(I.3) $l_{v,1} = \mathsf{labOf}(\phi_1.v, pc_1)$
(I.4) $v_1' = \mathsf{valOf}(\phi_1.v)$
(I.5) $v_1 = (v_1', l_{\phi,1} \sqcup l_{v,1})$
From (2) and (I.1), either
(I.6) $\mathsf{valOf}(\phi_2) \neq \mathsf{NULL}$ and $\mathsf{labOf}(\phi_1, pc_1), \mathsf{labOf}(\phi_2, pc_2) \sqsubseteq L$ or
(I.7) $\mathsf{labOf}(\phi_1, pc_1) = \mathsf{labOf}(\phi_2, pc_2) = H$

Subcase i: (I.6) is true
From (I.6),
(i.1) $\mathcal{E}$ ends in TS-GETVALG
From (i.1),
(i.2) $l_{\phi,2} = \mathsf{labOf}(\phi_2, pc_2)$
(i.3) $l_{v,2} = \mathsf{labOf}(\phi_2.v, pc_2)$
(i.4) $v_2' = \mathsf{valOf}(\phi_2.v)$
(i.5) $v_2 = (v_2', l_{\phi,2} \sqcup l_{v,2})$
From (I.6), (I.2), and (i.2),
(i.6) $l_{\phi_1}, l_{\phi,2} \sqsubseteq L$
From (i.6), (I.5), (i.5), and our security lattice,
(i.7) $v_1 = (v_1', l_{v,1})$ and $v_2 = (v_2', l_{v,2})$

From (i.7), (I.3), (I.4), (i.3), and (i.4),
(i.8) $v_1 = \phi_1.v$ and $v_2 = \phi_2.v$
From (2), (i.8), and the definition of $\approx_L$ for ϕ,
$v_1 \approx_L v_2$

Subcase ii: (I.7) is true and $\mathsf{valOf}(\phi_2) \neq \mathsf{NULL}$
By assumption,
(ii.1) $\mathcal{E}$ ends in TS-GETVALG
From (ii.1),
(ii.2) $l_{\phi,2} = \mathsf{labOf}(\phi_2, pc_2)$
(ii.3) $l_{v,2} = \mathsf{labOf}(\phi_2.v, pc_2)$
(ii.4) $v_2' = \mathsf{valOf}(\phi_2.v)$
(ii.5) $v_2 = (v_2', l_{\phi,2} \sqcup l_{v,2})$
From (I.7), (I.2), and (ii.2),
(ii.6) $l_{\phi_1}, l_{\phi,2} = H$
From (ii.6), (I.5), (ii.5), and our security lattice,
$v_1 \approx_L v_2$

Subcase iii: (I.7) is true and $\mathsf{valOf}(\phi_2 = \mathsf{NULL}$
By assumption,
(iii.1) $\mathcal{E}$ ends in TS-GETVALG-S
From (iii.1),
(iii.2) $v_2 = (\mathsf{dv}, H)$
From (I.7) and (I.2),
(iii.3) $l_{\phi,1} = H$
From (iii.2), (I.5), (iii.3), and our security lattice,
$v_1 \approx_L v_2$

Case II: $\mathcal{D}$ ends in TS-GETVALG-S
By assumption,
(II.1) $\mathsf{valOf}(\phi_1) = \mathsf{NULL}$
(II.2) $v_1 = (\mathsf{dv}, H)$
From (2) and (II.1), either
(II.3) $\mathsf{labOf}(\phi_1, pc_1) = \mathsf{labOf}(\phi_2, pc_2) = H$ or
(II.4) $\mathsf{valOf}(\phi_2 = \mathsf{NULL}$ and $\mathsf{labOf}(\phi_1, pc_1), \mathsf{labOf}(\phi_2, pc_2) \sqsubseteq L$

Subcase i: (II.3) is true and $\mathsf{valOf}(\phi_2) \neq \mathsf{NULL}$
By assumption,
(i.1) $\mathcal{E}$ ends in TS-GETVALG
From (i.1),
(i.2) $l_{\phi,2} = \mathsf{labOf}(\phi_2, pc_2)$
(i.3) $l_{v,2} = \mathsf{labOf}(\phi_2.v, pc_2)$
(i.4) $v_2' = \mathsf{valOf}(\phi_2.v)$
(i.5) $v_2 = (v_2', l_{\phi,2} \sqcup l_{v,2})$
From (II.3), and (i.2),
(i.6) $l_{\phi,2} = H$
From (i.6), (II.2), (i.5), and our security lattice,
$v_1 \approx_L v_2$

Subcase ii: (II.3) is true and $\mathsf{valOf}(\phi_2) = \mathsf{NULL}$
By assumption,
(ii.1) $\mathcal{E}$ ends in TS-GETVALG-S
From (ii.1),
(ii.2) $v_2 = (\mathsf{dv}, H)$
From (II.2) and (ii.2),
$v_1 \approx_L v_2$

Subcase iii: (II.4) is true
The proof for this case is similar to **Subcase ii**.

□

Requirement (EH2) H EH lookups are unobservable

Lemma 66. *If any of the folllowing:*
1) $G, \mathcal{P}, \sigma \vdash ks; \mathsf{lookupEHAll}(id.Ev(v)) \rightsquigarrow_H ks'$ *or*
2) $G, \mathcal{P}, \sigma \vdash ks; \mathsf{lookupEHAt}(id.Ev(v)) \rightsquigarrow_H ks'$ *or*
3) $G, \mathcal{P}, \sigma \vdash ks; \mathsf{lookupEHs}(E) \rightsquigarrow_H ks'$ *or*
4) $G, \mathcal{P}, \sigma \vdash ks; \mathsf{lookupEHs}(E) \rightsquigarrow_{pc} ks'$
then $ks \approx_L ks'$

Proof.
By induction on the structure of $\mathcal{E} :: G, \mathcal{P}, \sigma \vdash ks; \mathsf{lookupEHAPI}(...) \rightsquigarrow_{pc} ks'$

Case I: $\mathcal{E}$ ends in LOOKUPEHAPI with $pc = H$
The proof follows from Lemma 67 (Requirement **(EH2)**), Lemma 68 and Lemma 70.

Case II: $\mathcal{E}$ ends in LOOKUPEHS-R with $pc = H$
Follows from Lemma 69 (Requirement **(EH2)**), Lemma 70, and the IH.

Case III: $\mathcal{E}$ ends in LOOKUPEHS-R with $E \approx_L \cdot$
The assumption that $E \approx_L \cdot$ allows us to apply Lemma 69 (Requirement **(EH2)**). Then, the proof is similar to **Case II**.

Case IV: $\mathcal{E}$ ends in LOOKUPEHS-S
Follows from assumption that $ks' = ks$.

□

Lemma 67. $\mathsf{lookupEHAll}_G(\sigma, id.Ev(v), H) \approx_L \cdot$

Proof (sketch): The case for LOOKEHALL follows from our security lattice (everything is joined with the pc which is H, here). The case for LOOKEHALL-S is straightforward. The $pc = \cdot$ in LOOKUPEHALL-NC-MERGE, so this case holds vacuously. □

Lemma 68. $\mathsf{lookupEHAt}_G(\sigma, id.Ev(v), H) \approx_L \cdot$

Proof (sketch): The case for LOOKEHAT follows from our security lattice (everything is joined with the pc which is H, here). The case for LOOKEHAT-S is straightforward. The $pc = \cdot$ in LOOKUPEHAT-NC-MERGE, so this case holds vacuously. □

Lemma 69. $\mathsf{lookupEHs}_G(\sigma, id.Ev(v), H) \approx_L \cdot$

Proof (sketch): The proof follows from Lemma 67 (Requirement **(EH2)**) and Lemma 68 □

Lemma 70. *If* $\mathcal{C} \approx_L \cdot$ *and* $ks = \mathsf{createK}(\mathcal{P}, id.Ev(v), \mathcal{C})$ *then* $ks \approx_L \cdot$

Proof (sketch): This proof is by straightforward induction on the structure of $\mathcal{D} :: \mathsf{createK}(\mathcal{P}, id.Ev(v), \mathcal{C})$. It uses Lemma 71. □

Lemma 71. *If* $ks = \mathsf{crtK}_{\mathcal{V}}(eh, v, H)$ *then* $ks \approx_L \cdot$

Proof (sketch): This proof is straightforward and follows directly from the assumption that the $pc = H$. □

Lemma 72. $\mathsf{triggerEH}_{\mathcal{G}}(\sigma, H, id, Ev, v) \approx_L \cdot$

Proof.
By induction on the structure of $\mathcal{E} :: \mathsf{triggerEH}_{\mathcal{G}}(\sigma, H, id, Ev, v) = E$
Want to show $E \approx_L \cdot$
Note that the rules are very similar for each enforcement mechanism and EH storage, so we do not consider them separately.

Case I: $\mathcal{E}$ ends in TRIGGEREH
By assumption, and from our security lattice, $E = (id.Ev(v), H)$, therefore $E \approx_L \cdot$

Case II: $\mathcal{E}$ ends in TRIGGEREH-S
By assumption, $E = \cdot$, therefore $E \approx_L \cdot$

Case iii: $\mathcal{E}$ ends in TRIGGEREH-NC
By assumption, $pc = \cdot$, but we only want to consider cases where $pc = H$, so this case holds vacuously.

□

Requirement (EH3) H updates are unobservable

Lemma 73.
Unstructured EH storage: $\mathsf{assign}_{\mathcal{G}}(\sigma, H, id, v) \approx_L \sigma$
Tree structure EH storage: $\mathsf{assign}_{\mathcal{G}}(\sigma, H, a, v) \approx_L \sigma$

Proof (sketch): Only the cases for $\mathcal{G} \neq \mathsf{TS}$ are considered. The other cases are proven in the weak secrecy version: Lemma 75 and Lemma 76 (Requirement **(WEH3)**). The proof is straightforward. We examine each case of $\mathcal{E} :: \mathsf{assign}_{\mathcal{G}}(\sigma, H, id, v)$ (for the unstructured EH storage) and $\mathcal{E} :: \mathsf{assign}_{\mathcal{G}}(\sigma, H, a, v)$ (for the tree-structured EH storage) for each $\mathcal{G}$. In every case, only the H view is changed, so the resulting store is equivalent. □

Lemma 74. *For any $\mathcal{G}$, all of the following hold:*
Unstructured EH storage:

- $\mathsf{createElem}_{\mathcal{G}}(\sigma, H, id, v) \approx_L \sigma$
- $\mathsf{registerEH}_{\mathcal{G}}(\sigma, H, id, eh) \approx_L \sigma$

Tree structure EH storage:

- $\mathsf{createChild}_{\mathcal{G}}(\sigma, H, id, a_p, v) \approx_L \sigma$
- $\mathsf{createSibling}_{\mathcal{G}}(\sigma, H, id, a_s, v) \approx_L \sigma$
- $\mathsf{registerEH}_{\mathcal{G}}(\sigma, H, a, eh) \approx_L \sigma$

Proof.
For createElem, only the cases for $\mathcal{G} \neq \mathsf{TS}$ are considered. The other cases are proven in the weak secrecy version: Lemma 78 and Lemma 79 (Requirement **(WEH3)**).

Unstructured EH storage
We examine each case of $\mathcal{E}$ for $\mathcal{E} :: \mathsf{createElem}_{\mathcal{G}}(\sigma, H, id, v) = \sigma'$
Want to show $\sigma \approx_L \sigma'$

Case I: $\mathcal{G} = \mathsf{SMS}$
Case i: $\mathcal{E}$ ends in SMS-CREATE
By assumption,
(i.1) $\sigma_H = \mathsf{getStore}(\sigma, H)$
(i.2) $\sigma' = \mathsf{setStore}(\sigma, H, \sigma[id \mapsto (id, v, \cdot)])$
From (i.1), (i.2), and the definition of getStore, setStore, and $\approx_L$ for SMS stores,
$\sigma \approx_L \sigma'$

Case ii: $\mathcal{E}$ ends in SMS-CREATE-U
The proof follows from Lemma 73 (Requirement **(EH3)**).

Case iii: $\mathcal{E}$ ends in SMS-CREATE-NC
We only consider cases where $pc = H$, so this case holds vacuously.

Case II: $\mathcal{G} = \mathsf{FS}$
Case i: $\mathcal{E}$ ends in FS-CREATE
By assumption,
(i.1) $\phi = (id, \mathsf{getFacetV}(v, H), \cdot)$
(i.2) $\sigma' = \sigma[id \mapsto \phi]$
From (i.1) and the definition of $\downarrow_L$ for FS nodes,
(i.3) $\phi \downarrow_L = \cdot$
From (i.2) and (i.3),
$\sigma \approx_L \sigma'$

Case ii: $\mathcal{E}$ ends in FS-CREATE-U
The proof follows from Lemma 73 (Requirement **(EH3)**).

Case iii: $\mathcal{E}$ ends in FS-CREATE-NC
We only consider cases where $pc = H$, so this case holds vacuously.

We examine each case of $\mathcal{E}$ for $\mathcal{E} :: \mathsf{registerEH}_{\mathcal{G}}(\sigma, H, id, eh) = \sigma'$
Want to show $\sigma \approx_L \sigma'$

Case I: $\mathcal{G} = \mathsf{SMS}$

Case i: $\mathcal{E}$ ends in SMS-REGISTEREH
By assumption,
(i.1) $\sigma_H = \mathsf{getStore}(\sigma, H)$
(i.2) $\sigma' = \mathsf{setStore}(\sigma, H, \sigma_H[id \mapsto (id, v, M[Ev \mapsto \cdots])])$
From (i.1), (i.2), and the definition of getStore, setStore, and $\approx_L$ for SMS stores,
$\sigma \approx_L \sigma'$

Case ii: $\mathcal{E}$ ends in SMS-REGISTEREH-S
By assumption, $\sigma' = \sigma$, therefore $\sigma \approx_L \sigma'$.

Case iii: $\mathcal{E}$ ends in SMS-REGISTEREH-NC
We only consider cases where $pc = H$, so this case holds vacuously.

Case II: $\mathcal{G} = \mathsf{FS}$
Case i: $\mathcal{E}$ ends in FS-REGISTEREH
By assumption,
(i.1) $\sigma' = \sigma[id \mapsto (id, v, M[Ev \mapsto M(eh) \cup \{(eh, H)\}])]$
From (i.1),
$\sigma \approx_L \sigma'$

Case ii: $\mathcal{E}$ ends in FS-REGISTEREH-S
By assumption $\sigma' = \sigma$, therefore $\sigma \approx_L \sigma'$.

Case iii: $\mathcal{E}$ ends in FS-REGISTEREH-NC
We only consider cases where $pc = H$, so this case holds vacuously.

Case III: $\mathcal{G} = \mathsf{TS}$
Case i: $\mathcal{E}$ ends in TS-REGISTEREH
By assumption,
(i.1) $(id, v, M, l) = \mathsf{lookup}_{\mathsf{TS}}(\sigma, H, id)$
(i.2) $eh = \mathsf{on}Ev(x)\{c\}$
By assumption and from (i.1), (i.2), and our security lattice (which gives $H \sqcup l = H$),
(i.3) $\sigma' = \sigma[id \mapsto (id, v, M[Ev \mapsto M(eh) \cup \{(eh, H)\}])]$
From (i.3),
$\sigma \approx_L \sigma'$

Case ii: $\mathcal{E}$ ends in TS-REGISTEREH-S
By assumption, $\sigma' = \sigma$, therefore $\sigma \approx_L \sigma'$.

Case iii: $\mathcal{E}$ ends in TS-REGISTEREH-NC
We only consider cases where $pc = H$, so this case holds vacuously.

Tree-structured EH storage
We examine each case of $\mathcal{E}$ for $\mathcal{E} :: \mathsf{createChild}_{\mathcal{G}}(\sigma, H, id, a_p, v) = \sigma'$
Want to show $\sigma \approx_L \sigma'$

$\mathcal{G} = \mathsf{SMS}$
Case I: $\mathcal{E}$ ends in SMS-CREATEC
By assumption,
(I.1) $\sigma_H = \mathsf{getStore}(\sigma, H)$
(I.2) $\sigma'_H = \sigma_H[a \mapsto (id, v, \cdot, a_p, \cdot)]$
(I.3) $\sigma_H(a_p) = (id_p, v', M, a'_p, A)$
(I.4) $\sigma''_H = \sigma'_H[a_p \mapsto (id_p, v', M, a'_p, (a :: A))]$
(I.5) $\sigma' = \mathsf{setStore}(\sigma, H, \sigma''_H)$
From (I.1)-(I.5) and the definition of setStore,
$\sigma \approx_L \sigma'$

Case II: $\mathcal{E}$ ends in SMS-CREATEC-S
By assumption, $\sigma' \approx_L \sigma$

Case III: $\mathcal{E}$ ends in SMS-CREATEC-NC
By assumption, $pc = H$, so this case holds vacuously.

$\mathcal{G} = \mathsf{FS}$
Case I: $\mathcal{E}$ ends in FS-CREATEC
By assumption,
(I.1) $a \notin \sigma$
(I.2) $\sigma(a_p) = (id_p, v_p, M, a'_p, A)$
(I.3) $\sigma'' = \sigma[a_p \mapsto (id_p, v_p, M, a'_p, (\mathsf{createFacet}(a, H) :: A))]$
(I.4) $\sigma' = \sigma''[a \mapsto (id, \mathsf{createFacet}(v, H), \cdot, \mathsf{createFacet}(a_p, H), \cdot)]$
From the definition of createFacet,
(I.5) $(\mathsf{createFacet}(a, H) :: A) \approx_L^{\sigma', \sigma} A$
From (I.2), (I.3), (I.5), and the definition of $\downarrow_L$ for nodes,
(I.6) $\sigma \approx_L \sigma''$
From the definition of createFacet,
(I.7) $\mathsf{createFacet}(v, H) \downarrow_L = \cdot$ and
(I.8) $\mathsf{createFacet}(a_p, H) \downarrow_L = \cdot$
From (I.1), (I.4), (I.7), (I.8), and the definition of $\downarrow_L$ for nodes,
(I.9) $\sigma'' \approx_L \sigma'$
From (I.6) and (I.9),
$\sigma \approx_L \sigma'$

Case II: $\mathcal{E}$ ends in FS-CREATEC-UH
By assumption,
(II.1) $\sigma(a) = (id, v', M, a'_p, A)$
(II.2) $\sigma(a_p) = (id_p, v_p, M_p, a''_p, A_p)$
(II.3) $\sigma'' = \sigma[a_p \mapsto (id_p, v_p, M_p, a'_p, (\mathsf{createFacet}(a, H) :: A_p))]$
(II.4) $\sigma' = \sigma''[a \mapsto (id, \mathsf{updateFacet}(v', v, H), \cdot, \mathsf{updateFacet}(a'_p, a_p, H), \cdot)]$
From the definition of createFacet,
(II.5) $(\mathsf{createFacet}(a, H) :: A_p) \approx_L^{\sigma', \sigma} A_p$
From (II.2), (II.3), (II.5), and the definition of $\downarrow_L$ for nodes,
(II.6) $\sigma \approx_L \sigma''$
From the definition of updateFacet,
(II.7) $\mathsf{updateFacet}(v', v, H) \approx_L^{\sigma', \sigma} v'$
(II.8) $\mathsf{updateFacet}(a'_p, a_p, H) \approx_L^{\sigma', \sigma} a'_p$
From (II.1), (II.4), (II.7), (II.8), and the definition of $\downarrow_L$ for nodes,
(II.9) $\sigma'' \approx_L \sigma'$
From (II.6) and (II.9),
$\sigma \approx_L \sigma'$

Case III: $\mathcal{E}$ ends in FS-CREATEC-S1 or CREATEC-S2
By assumption, $\sigma' = \sigma$.

Case IV: $\mathcal{E}$ ends in FS-CREATEC-NC or CREATEC-UL
By assumption, $pc = H$, so this case holds vacuously.

We examine each case of $\mathcal{E}$ for $\mathcal{E} :: \mathsf{createSibling}_{\mathcal{G}}(\sigma, H, id, a_s, v) = \sigma'$
Want to show $\sigma \approx_L \sigma'$

$\mathcal{G} = \mathsf{SMS}$
Case I: $\mathcal{E}$ ends in SMS-CREATES
By assumption,
(I.1) $\sigma_H = \mathsf{getStore}(\sigma, H)$
(I.2) $\sigma'_H = \sigma_H[a \mapsto (id, v, \cdot, a_p, \cdot)]$
(I.3) $\sigma_H(a_p) = (id_p, v', M, a'_p, A :: a_s :: A')$
(I.4) $\sigma''_H = \sigma'_H[a_p \mapsto (id_p, v', M, a'_p, (A :: a_s :: a :: A'))]$
(I.5) $\sigma' = \mathsf{setStore}(\sigma, H, \sigma''_H)$
From (I.1)-(I.5) and the definition of setStore,
$\sigma \approx_L \sigma'$

Case II: $\mathcal{E}$ ends in SMS-CREATES-S1 or SMS-CREATES-S2
By assumption, $\sigma' = \sigma$.

Case III: $\mathcal{E}$ ends in SMS-CREATES-NC
By assumption, $pc = H$, so this case holds vacuously.

$\mathcal{G} = \mathsf{FS}$
Case I: $\mathcal{E}$ ends in FS-CREATES
By assumption,
(I.1) $a \notin \sigma$
(I.2) $\sigma(a_p) = (id_p, v_p, M, a'_p, (A :: a'_s :: A')$ for $a_p = \sigma(a_s).a_p \downarrow_H$ and $a'_s \downarrow_H = a_s$
(I.3) $\sigma'' = \sigma[a_p \mapsto (id_p, v_p, M, a'_p, (A :: a'_s :: \mathsf{createFacet}(a, H) :: A'))]$
(I.4) $\sigma' = \sigma''[a \mapsto (id, \mathsf{createFacet}(v, H), \cdot, \mathsf{createFacet}(a_p, H), \cdot)]$
From the definition of createFacet,
(I.5) $(\mathsf{createFacet}(A :: a'_s :: a, H) :: A) \approx_L^{\sigma', \sigma} (A :: a'_s :: A')$
From (I.2), (I.3), (I.5), and the definition of $\downarrow_L$ for nodes,
(I.6) $\sigma \approx_L \sigma''$
From the definition of createFacet,
(I.7) $\mathsf{createFacet}(v, H) \downarrow_L = \cdot$ and
(I.8) $\mathsf{createFacet}(a_p, H) \downarrow_L = \cdot$
From (I.1), (I.4), (I.7), (I.8), and the definition of $\downarrow_L$ for nodes,
(I.9) $\sigma'' \approx_L \sigma'$
From (I.6) and (I.9),
$\sigma \approx_L \sigma'$

Case II: $\mathcal{E}$ ends in FS-CREATES-UH
By assumption,
(II.1) $\sigma(a) = (id, v', M, a'_p, A)$
(II.2) $\sigma(a_p) = (id_p, v_p, M_p, a''_p, (A_p :: a'_s :: A'_p))$ for $a_p = \sigma(a_s).a_p \downarrow_H$ and $a'_s \downarrow_H = a_s$
(II.3) $\sigma'' = \sigma[a_p \mapsto (id_p, v_p, M_p, a'_p, (A_p :: a'_s :: \mathsf{createFacet}(a, H) :: A'_p))]$
(II.4) $\sigma' = \sigma''[a \mapsto (id, \mathsf{updateFacet}(v', v, H), \cdot, \mathsf{updateFacet}(a'_p, a_p, H), \cdot)]$
From the definition of createFacet,
(II.5) $(A_p :: a'_s :: \mathsf{createFacet}(a, H) :: A'_p) \approx_L^{\sigma', \sigma} (A_p :: a'_s :: A'_p)$
From (II.2), (II.3), (II.5), and the definition of $\downarrow_L$ for nodes,
(II.6) $\sigma \approx_L \sigma''$
From the definition of updateFacet,
(II.7) $\mathsf{updateFacet}(v', v, H) \approx_L^{\sigma', \sigma} v'$
(II.8) $\mathsf{updateFacet}(a'_p, a_p, H) \approx_L^{\sigma', \sigma} a'_p$
From (II.1), (II.4), (II.7), (II.8), and the definition of $\downarrow_L$ for nodes,
(II.9) $\sigma'' \approx_L \sigma'$
From (II.6) and (II.9),
$\sigma \approx_L \sigma'$

Case III: $\mathcal{E}$ ends in FS-CREATES-S1, CREATES-S2, or FS-CREATES-S3
By assumption, $\sigma' = \sigma$.

Case IV:$\mathcal{E}$ ends in FS-CREATES-NC or FS-CREATES-UL
By assumption, $pc = H$, so this case holds vacuously.

We examine each case of $\mathcal{E}$ for $\mathcal{E} :: \mathsf{registerEH}_{\mathcal{G}}(\sigma, H, a, eh) = \sigma'$
Want to show $\sigma \approx_L \sigma'$

$\mathcal{G} = \mathsf{SMS}$
Case I: $\mathcal{E}$ ends in SMS-REGISTEREH
By assumption,
(I.1) $\sigma_H = \mathsf{getStore}(\sigma, H)$
(I.2) $\sigma' = \mathsf{setStore}(\sigma, H, \sigma'_H)$
From (I.1) and (I.2) and the definition of setStore,
$\sigma' \approx_L \sigma$

Case II: $\mathcal{E}$ ends in SMS-REGISTEREH-S
By assumption, $\sigma' = \sigma$.

Case III: $\mathcal{E}$ ends in SMS-REGISTEREH-NC
By assumption, $pc = H$, so this case holds vacuously.

$\mathcal{G} = \mathsf{FS}$
Case I: $\mathcal{E}$ ends in FS-REGISTEREH
By assumption,
(I.1) $\sigma(a) = (id, v, M, a_p, A)$
(I.2) $M' = M[Ev \mapsto M(Ev) \cup \{(eh, H)\}]$ for $eh = \mathsf{on}Ev(x)\{c\}$
(I.3) $\sigma' = (id, v, M', a_p, A)$
From (I.2),
(I.4) $M \approx_L M'$
From (I.1), (I.3), and (I.4),
$\sigma \approx_L \sigma'$

Case II: $\mathcal{E}$ ends in FS-REGISTEREH-S
By assumption, $\sigma' = \sigma$.

Case III: $\mathcal{E}$ ends in FS-REGISGTEREH-NC
By assumption, $pc = H$, so this case holds vacuously.

□

Requirement (WEH3) H updates are unobservable (Weak Secrecy)

Lemma 75. *If* $\sigma_1 \approx_L \sigma_2$ *and* $\mathsf{assignW}_{\mathcal{G}}(\sigma_1, H, id, (v_1, l_1)) = (\sigma_1', \mathsf{gw}(x))$ *and* $\mathsf{assignW}_{\mathcal{G}}(\sigma_2, H, id, (v_2, l_2)) = (\sigma_2', \mathsf{gw}(x))$, *then* $\sigma_1' \approx_L \sigma_2'$

Proof. Only the cases for $\mathcal{G} = \mathsf{TS}$ are considered. The other cases are not considered since TS is the only one which emits gw(_) events.
Denote $\mathcal{D} :: \mathsf{assignW}_{\mathcal{G}}(\sigma_1, H, id, (v_1, l_1))$ and $\mathcal{E} :: \mathsf{assignW}_{\mathcal{G}}(\sigma_2, H, id, (v_2, l_2))$.

From the assumption that $\mathcal{D}$ and $\mathcal{E}$ produce $\mathsf{gw}(x)$, $\mathcal{D}$ and $\mathcal{E}$ must end in TS-ASSIGN-GW
By assumption and from our security lattice, $\sigma_1' = \sigma_1[x \mapsto (v_1, H)]$ and $\sigma_2' = \sigma_2[x \mapsto (v_2, H)]$.
Therefore, from the assumption that $\sigma_1 \approx_L \sigma_2$, $\sigma_1' \approx_L \sigma_2'$. □

Lemma 76. *If* $\mathsf{assignW}_{\mathcal{G}}(\sigma, H, id, (v, l)) = (\sigma', \bullet)$,*then* $\sigma \approx_L \sigma'$

Proof.
Only the cases for $\mathcal{G} = \mathsf{TS}$ are considered. The other cases follow from Lemma 73 (Requirement **(EH3)**).
We examine each case of $\mathcal{E} :: \mathsf{assignW}_{\mathcal{G}}(\sigma, H, id, (v, l))$

Case I: $\mathcal{E}$ ends in TS-ASSIGN
By assumption and from our security lattice,
(I.1) $H \sqsubseteq \mathsf{labOf}(\sigma(x), H)$
(I.2) $\sigma' = \sigma[x \mapsto (v, H)]$
From (I.1) and (I.2) and the definition of $\approx_L$ for values,
$\sigma \approx_L \sigma'$

Case II: $\mathcal{E}$ ends in TS-ASSIGN-S
By assumption, $\sigma' = \sigma$. Therefore, $\sigma \approx_L \sigma'$.

Case III: $\mathcal{E}$ ends in TS-ASSIGN-GW
We only consider cases which emit $\bullet$, therefore this case holds vacuously.

□

Lemma 77. *If* $\sigma_1 \approx_L \sigma_2$, *and* $\mathsf{assignW}_{\mathcal{G}}(\sigma_1, H, id, (v_1, l_1)) = (\sigma_1', \mathsf{gw}(id))$ *and* $\mathsf{createElemW}_{\mathcal{G}}(\sigma_2, H, id, (v_2, l_2)) = (\sigma_2', \mathsf{gw}(id))$ *then* $\sigma_1' \approx_L \sigma_2'$

Proof.
Only the cases for $\mathcal{G} = \mathsf{TS}$ are considered. The other cases are not considered since TS is the only one which emits gw(_) events.
Denote $\mathcal{D} :: \mathsf{assignW}_{\mathcal{G}}(\sigma_1, H, id, (v_1, l_1)) = (\sigma_1', \mathsf{gw}(id))$ and $\mathcal{E} :: \mathsf{createElemW}_{\mathcal{G}}(\sigma_2, H, id, (v_2, l_2)) = (\sigma_2', \mathsf{gw}(id))$
From the assumption that $\mathcal{D}$ and $\mathcal{E}$ produce $\mathsf{gw}(id)$,

(1) $\mathcal{D}$ must end in TS-ASSIGNEH-GW
By assumption,
(2) $\sigma_1 \approx_L \sigma_2$
From (1) and our security lattice,
(3) $\sigma_1(id) = (id, \_, M, l_\phi)$
(4) $\sigma_1' = \sigma_1[id \mapsto (id, (v_1, H), M, l_\phi)]$
We examine each case of $\mathcal{E}$

Case I: $\mathcal{E}$ ends in TS-CREATE-U1-GW
By assumption and from our security lattice,
(I.1) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, H, id) = (id, v', M, l')$
(I.2) $\phi_2 = (id, (v_2, H), M, l')$
(I.3) $\sigma_2' = \sigma_2[id \mapsto \phi_2]$
From (I.1)-(I.3), (2)-(4), and since the node labels do not change in either case,
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{E}$ ends in TS-CREATE-NC
We only consider cases where $pc = H$ so this case holds vacuously.

Case III: $\mathcal{E}$ ends in TS-CREATE, TS-CREATE-U1, or TS-CREATE-U2
We only consider rules which emit $\mathsf{gw}(id)$ so these cases hold vacuously.

□

Lemma 78. *If* $\sigma_1 \approx_L \sigma_2$ *and* $\mathsf{createElemW}_{\mathcal{G}}(\sigma_1, H, id, (v_1, l_1)) = (\sigma_1', \mathsf{gw}(x))$ *and* $\mathsf{createElemW}_{\mathcal{G}}(\sigma_2, H, id, (v_2, l_2)) = (\sigma_2', \mathsf{gw}(x))$*, then* $\sigma_1' \approx_L \sigma_2'$

Proof.
Only the cases for $\mathcal{G} = \mathsf{TS}$ are considered. The other cases are not considered since TS is the only one which emits $\mathsf{gw}(\_)$ events.
Denote $\mathcal{D} :: \mathsf{createElemW}_{\mathcal{G}}(\sigma_1, H, id, (v_1, l_1)) = (\sigma_1', \mathsf{gw}(id))$ and
$\mathcal{E} :: \mathsf{createElemW}_{\mathcal{G}}(\sigma_2, H, id, (v_2, l_2)) = (\sigma_2', \mathsf{gw}(id))$
We examine each case of $\mathcal{D}$
By assumption,
(1) $\sigma_1 \approx_L \sigma_2$

Case I: $\mathcal{D}$ ends in TS-CREATE-U1-GW
By assumption and from our security lattice,
(I.1) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_1, H, id) = (id, \_, M_1, l_{\phi,1})$
(I.2) $\sigma_1' = \sigma_1[id \mapsto (id, (v_1, H), M_1, l_{\phi,1})$
From the assumption that $\mathcal{E}$ emits $\mathsf{gw}(id)$ and runs in the H context,
(I.3) $\mathcal{E}$ ends in TS-CREATE-U1-GW
From (I.3)
(I.4) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, H, id) = (id, \_, M_2, l_{\phi,2})$
(I.5) $\sigma_2' = \sigma_2[id \mapsto (id, (v_2, H), M_2, l_{\phi,2})$
From (1), (II.1), (I.4), and the definition of $\approx_L$ for TS nodes, either
(I.6) $l_{\phi_1} = l_{\phi_2} = H$ or
(I.7) $l_{\phi_1} = l_{\phi_2} = L$ and $M_1 \approx_L M_2$
From (1), (II.2), and (I.5)-(I.7),
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{D}$ ends in TS-CREATE, TS-CREATE-U1, or TS-CREATE-U2
We only consider rules which emit $\mathsf{gw}(id)$ so these cases hold vacuously.

Case III: $\mathcal{D}$ ends in TS-CREATE-NC
We only consider cases where $pc = H$ so this case holds vacuously.

□

Lemma 79. *If* $\mathsf{createElemW}_{\mathcal{G}}(\sigma, H, id, (v, l)) = (\sigma', \bullet)$*, then* $\sigma \approx_L \sigma'$

Proof.
Only the cases for $\mathcal{G} = \mathsf{TS}$ are considered. The other cases follow from Lemma 74 (Requirement **(EH3)**).
We examine each case of $\mathcal{D} :: \mathsf{createElemW}_{\mathcal{G}}(\sigma, H, id, v) = (\sigma, \mathsf{gw}(id))$

Case I: $\mathcal{D}$ ends in TS-CREATE

By assumption,
(I.1) $\mathsf{lookup}_{\mathsf{TS}}(\sigma, H, id) = (\mathsf{NULL}, \_)$
(I.2) $\sigma' = \sigma[id \mapsto (id, (v, l), \cdot, H)]$
From (I.1) and the definition of lookup for TS,
(I.3) $id \notin \sigma$
From (I.2) and (I.3),
$\sigma \approx_L \sigma'$

Case II: $\mathcal{D}$ ends in TS-CREATE-U1
By assumption and from our security lattice,
(II.1) $\mathsf{lookup}_{\mathsf{TS}}(\sigma, H, id) = (id, (v', l'), M, l_\phi)$
(II.2) $\sigma' = \sigma_1[id \mapsto (id, (v, H), M, l_\phi)$
(II.3) $H \sqsubseteq l'$
From (II.1)-(II.3),
$\sigma \approx_L \sigma'$

Case III: $\mathcal{D}$ ends in TS-CREATE-U2
By assumption, $\mathsf{lookup}_{\mathsf{TS}}(\sigma, H, id) = (id, v', M, l')$ with $l' \not\sqsubseteq H$
But from our security lattice, such an l' does not exist. So this case holds vacuously.

Case IV: $\mathcal{D}$ ends in TS-CREATE-U1-GW
We only consider rules which emit $\bullet$ so this case holds vacuously.

Case V: $\mathcal{D}$ ends in TS-CREATE-NC
We only consider cases where $pc = H$ so this case holds vacuously.

□

Requirement (EH4) L updates are equivalent

Lemma 80. *If* $\sigma_1 \approx_L \sigma_2$, $pc_1, pc_2 \sqsubseteq L$, *then*
Unstructured EH storage: *if* $v_1 \approx_L v_2$, *then* $\mathsf{assign}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1) \approx_L \mathsf{assign}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2)$
Tree structure EH storage: *if* $v_1 \approx_L^{\sigma_1,\sigma_2} v_2$, *and* $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$, *then* $\mathsf{assign}_{\mathcal{G}}(\sigma_1, pc_1, a, v_1) \approx_L \mathsf{assign}_{\mathcal{G}}(\sigma_2, pc_2, a, v_2)$

Proof.
Only the cases for $\mathcal{G} \neq \mathsf{TS}$ are considered. The other cases are proven in the weak secrecy version: Lemma 82.
By induction on the structure of $\mathcal{D} :: \mathsf{assign}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1) = \sigma_1'$ and $\mathcal{E} :: \mathsf{assign}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2) = \sigma_2'$
Want to show $\sigma_1' \approx_L \sigma_2'$
By assumption,
(1) $\sigma_1 \approx_L \sigma_2$
(2) $pc_1, pc_2 \sqsubseteq L$

Unstructured EH storage
By assumption,
(U.1) $v_1 \approx_L v_2$

$\mathcal{G} = \mathsf{SMS}$
Case I: $\mathcal{D}$ ends in SMS-ASSIGNEH
By assumption and from (2),
(I.1) $pc_1 = L$
By assumption and from (I.1),
(I.2) $\sigma_{L,1} = \mathsf{getStore}(\sigma_1, L)$
(I.3) $(v', M) = \sigma_{L,1}(id)$
(I.4) $\sigma_1' = \mathsf{setStore}(\sigma_1, L, \sigma_{L,1}[id \mapsto (v_1, M)])$

Subcase i: $pc_2 = L$
Let
(i.1) $\sigma_{L,2} = \mathsf{getStore}(\sigma_2, L)$
From (1), (I.2), (i.1), and the definition of getStore,
(i.2) $\sigma_{L,1} = \sigma_{L,2}$
From (I.3) and (i.2),
(i.3) $(v', M) = \sigma_{L,2}(id)$
By assumption and from (i.3),
(i.4) $\mathcal{E}$ ends in SMS-ASSIGNEH
From (i.3) and (i.4),
(i.5) $\sigma_2' = \mathsf{setStore}(\sigma_2, L, \sigma_{L,2}[id \mapsto (v_2, M)])$
From (U.1), (I.4), (i.5), and (i.2),
(i.6) $\sigma_{L,1}[id \mapsto (v_1, M)] = \sigma_{L,2}[id \mapsto (v_2, M)])$
From (i.6), (I.4), (i.5), and the definition of setStore,
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $pc_2 = \cdot$
By assumption,
(ii.1) $\exists \mathcal{E}' :: \sigma_2'' = \mathsf{assign}_{\mathsf{SMS}}(\sigma_2, H, id, \mathsf{getFacet}(v_2, H))$
(ii.2) $\exists \mathcal{E}'' :: \sigma_2' = \mathsf{assign}_{\mathsf{SMS}}(\sigma_2'', L, id, \mathsf{getFacet}(v_2, L))$
From (ii.1) and Lemma 73.U (Requirement **(EH3)**),
(ii.3) $\sigma_2'' \approx_L \sigma_2$
From (1) and (ii.3),
(ii.4) $\sigma_1 \approx_L \sigma_2''$
From (U.1), (ii.2), (ii.4), and the definition of getFacet,
the IH may be applied on $\mathcal{D}$ and $\mathcal{E}''$
From (ii.2) and the IH on $\mathcal{D}$ and $\mathcal{E}''$,
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{E}$ ends in SMS-ASSIGNEH
The proof for this case is similar to **Case I**.

Case III: $\mathcal{D}$ or $\mathcal{E}$ ends in SMS-ASSIGNEH-S
By assumption and from (1), $\sigma_1' = \sigma_1$ and $\sigma_2' = \sigma_2$ and the conclusion follows from (1).

Case IV: $\mathcal{D}$ or $\mathcal{E}$ ends in SMS-ASSIGN-NC

Without loss of generality, assume $\mathcal{D}$ ends in SMS-ASSIGN-NC. The proof for $\mathcal{E}$ is similar.
By assumption,
(IV.1) $\exists \mathcal{D}' :: \sigma_1'' = \mathsf{assign}_{\mathsf{SMS}}(\sigma_1, H, id, \mathsf{getFacet}(v_1, H))$
(IV.2) $\exists \mathcal{D}'' :: \sigma_1' = \mathsf{assign}_{\mathsf{SMS}}(\sigma_1'', L, id, \mathsf{getFacet}(v_1, L))$
From (IV.1) and Lemma 73.U (Requirement **(EH3)**),
(IV.3) $\sigma_1'' \approx_L \sigma_1$
From (1) and (IV.3),
(IV.4) $\sigma_2 \approx_L \sigma_1''$
From (U.1), (IV.2), (IV.4), and the definition of getFacet,
the IH may be applied on $\mathcal{D}''$ and $\mathcal{E}$
From (IV.2) and the IH on $\mathcal{D}''$ and $\mathcal{E}$,
$\sigma_1' \approx_L \sigma_2'$

$\mathcal{G} = \mathsf{FS}$
Case I: $\mathcal{D}$ ends in FS-ASSIGNEH
By assumption,
(I.1) $pc_1 = \cdot$
(I.2) $(v_1', M_1) = \mathsf{lookup}_{\mathsf{FS}}(\sigma_1, \cdot, id)$
(I.3) $\sigma_1' = \sigma_1[id \mapsto (v_1, M_1)]$
From (1), (2), (I.2), and Lemma 48.U,
(I.4) $(v_1', M_1) \approx_L \mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id) = \phi_2$
From (I.4), and the definition of $\downarrow_L$ for FS nodes,
(I.5) $\phi_2 \downarrow_L \neq \mathsf{NULL}$
From (I.5) and (2),
$\mathcal{E}$ ends in FS-ASSIGNEH-UPD, FS-ASSIGNEH, or FS-ASSIGNEH-NC

Subcase i: $\mathcal{E}$ ends in FS-ASSIGNEH-UPD
By assumption and from (2),
(i.1) $(v_2', M_2) = \mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id)$
(i.2) $v_2'' = \mathsf{updateFacet}(v_2', v_2, L)$
(i.3) $\sigma_2' = \sigma[id \mapsto (v_2'', M_2)]$
From (i.1), (U.1) and the definition of updateFacet,
(i.4) $v_1 \approx_L v_2''$
From (I.4) and (i.1),
(i.5) $M_1 \approx_L M_2$
From (1), (I.3), and (i.3)-(i.5),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $\mathcal{E}$ ends in FS-ASSIGNEH
By assumption,
(ii.1) $\sigma_2' = \sigma[id \mapsto (v_2, M_2)]$
From (I.4) and (ii.1),
(ii.2) $M_1 \approx_L M_2$
From (1), (I.3), (ii.1), (U.1), and (ii.2),
$\sigma_1' \approx_L \sigma_2'$

Subsubcase iii: $\mathcal{E}$ ends in FS-ASSIGNEH-NC
By assumption,
(iii.1) $\sigma_2'' = \mathsf{assign}_{\mathsf{FS}}(\sigma_2, H, id, v_2)$
(iii.2) $\mathcal{E}' :: \sigma_2' = \mathsf{assign}_{\mathsf{FS}}(\sigma_2'', L, id, v_2)$
From (iii.1) and Lemma 73.U (Requirement **(EH3)**),
(iii.3) $\sigma_2 \approx_L \sigma_2''$
From (iii.2), (iii.3), and (1),
(iii.4) IH may be applied on $\mathcal{E}'$ and $\mathcal{D}$
From (iii.4) and IH on $\mathcal{E}'$ and $\mathcal{D}$,
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{E}$ ends in FS-ASSIGNEH
The proof is similar to **Case I**.

Case III: $\mathcal{D}$ ends in FS-ASSIGNEH-UPD
By assumption and from (2),
(III.1) $pc_1 = L$

(III.2) $(v_1', M_1) = \mathsf{lookup}_{\mathsf{FS}}(\sigma_1, L, id)$
(III.3) $v_1'' = \mathsf{updateFacet}(v_1', v_1, L)$
(III.4) $\sigma_1' = \sigma_1[id \mapsto (v_1'', M_1)]$
From (III.3) and the definition of updateFacet,
(III.5) $v_1 \approx_L v_1''$
From (1), (2), (III.2), and Lemma 48.U,
(III.6) $(v_1', M_1) \approx_L \mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id) = \phi_2$
From (III.6), and the definition of $\downarrow_L$ for FS nodes,
(III.7) $\phi_2 \downarrow_L \neq \mathsf{NULL}$
From (III.7) and (2),
$\mathcal{E}$ ends in FS-ASSIGNEH-UPD, FS-ASSIGNEH, or FS-ASSIGNEH-NC

Subcase i: $\mathcal{E}$ ends in FS-ASSIGNEH-UPD
By assumption and from (2),
(i.1) $(v_2', M_2) = \mathsf{lookup}_{\mathsf{FS}}(\sigma_2, L, id)$
(i.2) $v_2'' = \mathsf{updateFacet}(v_2', v_2, L)$
(i.3) $\sigma_2' = \sigma_2[id \mapsto (v_2'', M_2)]$
From (III.6) and (i.1),
(i.4) $(v_1', M_1) \approx_L (v_2', M_2)$
From (i.2) and the definition of updateFacet,
(i.5) $v_2 \approx_L v_2''$
From (1), (III.4), (i.3), (i.4), and (i.5),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $\mathcal{E}$ ends in FS-ASSIGNEH
This proof is covered by **Case II**.

Subcase iii: $\mathcal{E}$ ends in FS-ASSIGNEH-NC
By assumption,
By assumption,
(iii.1) $\sigma_2'' = \mathsf{assign}_{\mathsf{FS}}(\sigma_2, H, id, v_2)$
(iii.2) $\mathcal{E}' :: \sigma_2' = \mathsf{assign}_{\mathsf{FS}}(\sigma_2'', L, id, v_2)$
From (iii.1) and Lemma 73.U (Requirement **(EH3)**),
(iii.3) $\sigma_2 \approx_L \sigma_2''$
From (iii.2), (iii.3), and (1),
(iii.4) IH may be applied on $\mathcal{E}'$ and $\mathcal{D}$
From (iii.4) and IH on $\mathcal{E}'$ and $\mathcal{D}$,
$\sigma_1' \approx_L \sigma_2'$

Case IV: $\mathcal{D}$ ends in FS-ASSIGNEH-NC
By assumption,
(IV.1) $\sigma_1'' = \mathsf{assign}_{\mathsf{FS}}(\sigma_1, H, id, v_1)$
(IV.2) $\exists \mathcal{D}' :: \sigma_1' = \mathsf{assign}_{\mathsf{FS}}(\sigma_1'', L, id, v_1)$
From (IV.1) and Lemma 73.U (Requirement **(EH3)**),
(IV.3) $\sigma_1'' \approx_L \sigma_1$
From (IV.2), (IV.3), and (1),
(IV.4) IH may be applied on $\mathcal{D}'$ and $\mathcal{E}$
From (IV.4) and IH on $\mathcal{D}'$ and $\mathcal{E}$ gives
$\sigma_1' \approx_L \sigma_2'$

Case V: $\mathcal{E}$ ends in FS-ASSIGNEH-NC
The proof for this case is similar to **Case IV**.

Case VI: $\mathcal{D}$ ends in FS-ASSIGNEH-S
By assumption,
(VI.1) $\mathsf{lookup}_{\mathsf{FS}}(\sigma_1, pc_1, id) = \mathsf{NULL}$
(VI.2) $\sigma_1' = \sigma_1$
From (1), (2), (VI.1), and Lemma 48.U,
(VI.3) $\mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id) = \mathsf{NULL}$ or
(VI.4) $\mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id) = \langle \_ | \mathsf{NULL} \rangle$

Subcase i: (VI.3) is true
By assumption,

(i.1) $\mathcal{E}$ ends in FS-ASSIGNEH-S
From (i.1),
(i.2) $\sigma_2' = \sigma_2$
From (1), (VI.2), and (i.2),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: (VI.4) is true
By assumption, $\mathcal{E}$ ends in FS-ASSIGNEH-NC.
This case is covered by **Case V**.

Tree-structured EH storage:
By assumption,
(T.1) $v_1 \approx_L^{\sigma_1,\sigma_2} v_2$
(T.2) $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$

$\mathcal{G} = \mathsf{SMS}$
Case I: $\mathcal{D}$ or $\mathcal{E}$ ends in SMS-ASSIGNEH
Without loss of generality, assume that $\mathcal{D}$ ends in ASSIGNEH. The proof for $\mathcal{E}$ is similar.
By assumption,
(I.1) $\sigma_{L,1} = \mathsf{getStore}(\sigma_1, L)$
(I.2) $\sigma_{L,1}(a_1) = (id, v_1', M_1, a_{p,1}, A_1)$
(I.3) $\sigma_{L,1}' = \sigma_{L,1}[a_1 \mapsto (id, v_1, M_1, a_{p,1}, A_1)]$
(I.4) $\sigma_1' = \mathsf{setStore}(\sigma_1, L, \sigma_{L,1}')$

Subcase i: $pc_2 = \cdot$
By assumption,
(i.1) $\mathcal{E}$ ends in SMS-ASSIGNEH-NC
From (i.1),
(i.2) $\sigma_2'' = \mathsf{assign}_{\mathsf{SMS}}(\sigma_2, H, \mathsf{getFacet}(a_2, H), \mathsf{getFacet}(v_2, H))$
(i.3) $\mathcal{E}' :: \sigma_2' = \mathsf{assign}_{\mathsf{SMS}}(\sigma_2'', L, \mathsf{getFacet}(a_2, L), \mathsf{getFacet}(v_2, L))$
From (i.2) and Lemma 73.U (Requirement **(EH3)**),
(i.4) $\sigma_2'' \approx_L \sigma_2$
From (i.4) and (1),
(i.5) $\sigma_1 \approx_L \sigma_2''$
From (i.5), (i.3), (T.1), (T.2), and the definition of getFacet,
(i.6) IH may be applied on $\mathcal{D}$ and $\mathcal{E}'$
From (i.6) and the IH on $\mathcal{D}$ and $\mathcal{E}'$,
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $pc_2 = L$
Denote
(ii.1) $\sigma_{L,2} = \mathsf{getStore}(\sigma_2, L)$
From (1), (T.2), and (I.2),
(ii.2) $\sigma_{L,2}(a_2) = (v_2', M_2, a_{p,2}, A_2)$ with
(ii.3) $(id, v_1', M_1, a_{p,1}, A_1) \approx_L^{\sigma_1,\sigma_2} (id, v_2', M_2, a_{p,2}, A_2)$
By assumption and from (ii.2),
(ii.4) $\mathcal{E}$ ends in SMS-ASSIGNEH
From (ii.4),
(ii.5) $\sigma_{L,2}' = \sigma_{L,2}[a_2 \mapsto (v_2, M_2, a_{p,2}, A_2)]$
(ii.6) $\sigma_2' = \mathsf{setStore}(\sigma_2, L, \sigma_{L,2}')$
From (ii.3) and (T.1),
(ii.7) $(v_1, M_1, a_{p,1}, A_1) \approx_L^{\sigma_1,\sigma_2} (v_2, M_2, a_{p,2}, A_2)$
From (1), (I.3), (ii.5), (I.4), (ii.6), (ii.7), and the definition of setStore,
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{D}$ ends in SMS-ASSIGNEH-S
Without loss of generality, assume that $\mathcal{D}$ ends in SMS-ASSIGNEH-S. The proof for $\mathcal{E}$ is similar.
By assumption,
(II.1) $a_1 = \mathsf{NULL}$
(II.2) $\sigma_1' = \sigma_1$
From (T.2) and (II.1), either

(II.3) $a_2 = \mathsf{NULL}$ or
(II.4) $pc_2 = \cdot$ and $a_2 = \langle \_ | \mathsf{NULL} \rangle$

Subcase i: (II.3) is true
From (II.3),
(i.1) $\mathcal{E}$ ends in SMS-ASSIGNEH-S
From (i.1),
(i.2) $\sigma_2' = \sigma_2$
From (1), (II.2), and (i.2),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: (II.4) is true
The proof for this case is similar to **Subcase I.i**.

Case III: $\mathcal{D}$ ends in SMS-ASSIGNEH-NC
Without loss of generality, assume that $\mathcal{D}$ ends in SMS-ASSIGNEH-NC. The proof for $\mathcal{E}$ is similar.
By assumption,
(III.1) $\sigma_1'' = \mathsf{assign}_{\mathsf{SMS}}(\sigma_1, H, \mathsf{getFacet}(a_1, H), \mathsf{getFacet}(v_1, H))$
(III.2) $\mathcal{D}' :: \sigma_1' = \mathsf{assign}_{\mathsf{SMS}}(\sigma_1'', L, \mathsf{getFacet}(a_1, L), \mathsf{getFacet}(v_1, L))$
From (III.1) and Lemma 73.U (Requirement **(EH3)**),
(III.3) $\sigma_1'' \approx_L \sigma_1$
From (III.3) and (1),
(III.4) $\sigma_1 \approx_L \sigma_2''$
From (III.4), (III.2), (T.1), (T.2), and the definition of getFacet,
(III.5) IH may be applied on $\mathcal{D}'$ and $\mathcal{E}$
From (III.5) and the IH on $\mathcal{D}'$ and $\mathcal{E}$,
$\sigma_1' \approx_L \sigma_2'$

$\mathcal{G} = \mathsf{FS}$
Case I: $\mathcal{D}$ ends in FS-ASSIGNEH
Without loss of generality, assume that $\mathcal{D}$ ends in FS-ASSIGNEH. The proof for $\mathcal{E}$ is similar.
By assumption,
(I.1) $\sigma_1(a_1) = (id, v_1', M_1, a_{p,1}, A_1)$
(I.2) $\sigma_1' = \sigma_1[a_1 \mapsto (id, v_1, M_1, a_{p,1}, A_1)$
From (1), (T.2), and (I.1), either
(I.3) $\sigma_2(a_2) \approx_L^{\sigma_2, \sigma_1} \sigma_1(a_1)$ or
(I.4) $a_2 = \langle \_ | \_ \rangle$

Subcase i: (I.3) is true and $pc = \cdot$
By assumption and from (I.3),
(i.1) $\mathcal{E}$ must end in FS-ASSIGNEH
From (i.1),
(i.2) $\sigma_2(a_2) = (id, v_2', M_2, a_{p,2}, A_2)$
(i.3) $\sigma_2' = \sigma_2[a_2 \mapsto (id, v_2, M_2, a_{p,2}, A_2)]$
From (I.1)-(I.3), (i.2), (i.3), and (T.1),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: (I.3) is true and $pc = L$
By assumption and from (I.3),
(ii.1) $\mathcal{E}$ must end in FS-ASSIGNEH-UPD
By assumption and from (ii.1),
(ii.2) $\sigma_2(a_2) = (id, v_2', M_2, a_{p,2}, A_2)$
(ii.3) $v_2'' = \mathsf{updateFacet}(v_2', v_2, L)$
(ii.4) $\sigma_2' = \sigma_2[a_2 \mapsto (id, v_2'', M_2, a_{p,2}, A_2)$
From (1), (I.1)-(I.3), (ii.2)-(ii.4), and the definition of updateFacet,
$\sigma_1' \approx_L \sigma_2'$

Subcase iii: (I.4) is true
From (I.4),
(iii.1) $\mathcal{E}$ must end in FS-ASSIGNEH-NC
(iii.2) $a_2 = \langle a_H | a_L \rangle$
From (iii.1),
(iii.3) $\sigma_2'' = \mathsf{assign}_{\mathsf{FS}}(\sigma_2, H, a_H, \mathsf{getFacetV}(v_2, H))$

(iii.4) $\mathcal{E}' :: \sigma_2' = \mathsf{assign}_{\mathsf{FS}}(\sigma_2'', L, a_L, \mathsf{getFacetV}(v_2, L))$
From (iii.3) and Lemma 73.T (Requirement **(EH3)**),
(iii.5) $\sigma_2 \approx_L \sigma_2''$
From (T.2) and (iii.5),
(iii.6) $\sigma_1 \approx_L \sigma_2''$
From (1), (2), (T.1), (iii.6), and the definition of getFacetV,
(iii.7) IH may be applied on $\mathcal{D}$ and $\mathcal{E}'$
From (iii.7) and IH on $\mathcal{D}$ and $\mathcal{E}'$,
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{D}$ ends in FS-ASSIGNEH-NC
Without loss of generality, assume that $\mathcal{D}$ ends in FS-ASSIGNEH-NC. The proof for $\mathcal{E}$ is similar.
By assumption,
(II.1) $\sigma_1'' = \mathsf{assign}_{\mathsf{FS}}(\sigma_1, H, a_H, \mathsf{getFacetV}(v_1, H))$
(II.2) $\mathcal{D}' :: \sigma_1' = \mathsf{assign}_{\mathsf{FS}}(\sigma_1'', L, a_L, \mathsf{getFacetV}(v_1, L))$
From (II.1) and Lemma 73.T (Requirement **(EH3)**),
(II.3) $\sigma_1 \approx_L \sigma_1''$
From (T.2) and (II.3),
(II.4) $\sigma_1'' \approx_L \sigma_2$
From (1), (2), (T.1), (II.4), and the definition of getFacetV,
(II.5) IH may be applied on $\mathcal{D}'$ and $\mathcal{E}$
From (II.5) and IH on $\mathcal{D}'$ and $\mathcal{E}$,
$\sigma_1' \approx_L \sigma_2'$

Case III: $\mathcal{D}$ ends in FS-ASSIGNEH-UPD
Without loss of generality, assume that $\mathcal{D}$ ends in FS-ASSIGNEH-UPD. The proof for $\mathcal{E}$ is similar.
By assumption and from (2),
(III.1) $\sigma_1(a_1) = (id, v_1', M_1, a_{p,1}, A_1)$
(III.2) $v_1'' = \mathsf{updateFacet}(v_1', v_1, L)$
(III.3) $\sigma_1' = \sigma_1[a_1 \mapsto (id, v_1'', M_1, a_{p,1}, A_1)$
From (1), (T.2), and (III.1), either
(III.4) $\sigma_2(a_2) \approx_L^{\sigma_2, \sigma_1} \sigma_1(a_1)$ or
(III.5) $a_2 = \langle \_ | \_ \rangle$

Subcase i: (III.4) is true and $pc = \cdot$
This case is covered by **Case I**

Subcase ii: (III.4) is true and $pc = L$
By assumption and from (III.4),
(ii.1) $\mathcal{E}$ ends in FS-ASSIGNEH-UPD
By assumption and from (ii.1),
(ii.2) $\sigma_2(a_2) = (id, v_2', M_2, a_{p,2}, A_2)$
(ii.3) $v_2'' = \mathsf{updateFacet}(v_2', v_2, L)$
(ii.4) $\sigma_2' = \sigma_2[a_2 \mapsto (id, v_2'', M_2, a_{p,2}, A_2)$
From (T.1), (T.2), (III.1)-(III.3), (ii.2)-(ii.4), and the definition of updateFacet,
$\sigma_1' \approx_L \sigma_2'$

Subcase iii: (III.5) is true
This case is covered by **Case II**

Case IV: $\mathcal{D}$ ends in FS-ASSIGNEH-S
Without loss of generality, assume that $\mathcal{D}$ ends in FS-ASSIGNEH-S. The proof for $\mathcal{E}$ is similar.
By assumption,
(IV.1) $a_1 = \mathsf{NULL}$
(IV.2) $\sigma_1' = \sigma_1$
From (T.2) and (IV.1), either
(IV.3) $a_2 = \mathsf{NULL}$ or
(IV.4) $a_2 = \langle \_ | \mathsf{NULL} \rangle$

Subcase i: (IV.3) is true
By assumption,
(i.1) $\mathcal{E}$ ends in FS-ASSIGNEH-S
From (i.1),

(i.2) $\sigma_2' = \sigma_2$
From (1), (IV.2), and (i.2),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: (IV.4) is true
The proof for this case is covered by **Case II**.

□

Lemma 81. *If $\sigma_1 \approx_L \sigma_2$, $pc_1, pc_2 \sqsubseteq L$, then*
Unstructured EH storage: *If $v_1 \approx_L v_2$, then*
- $\mathsf{createElem}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1) \approx_L \mathsf{createElem}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2)$
- $\mathsf{registerEH}_{\mathcal{G}}(\sigma_1, pc_1, id, eh) \approx_L \mathsf{registerEH}_{\mathcal{G}}(\sigma_2, pc_2, id, eh)$

Tree structure EH storage: *If $v_1 \approx_L^{\sigma_1,\sigma_2} v_2$, $a_{p,1} \approx_L^{\sigma_1,\sigma_2} a_{p,2}$, $a_{s,1} \approx_L^{\sigma_1,\sigma_2} a_{s,2}$, and $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$, then*
- $\mathsf{createChild}_{\mathcal{G}}(\sigma_1, pc_1, id, a_{p,1}, v_1) \approx_L \mathsf{createChild}_{\mathcal{G}}(\sigma_2, pc_2, id, a_{p,1}, v_2)$
- $\mathsf{createSibling}_{\mathcal{G}}(\sigma_1, pc_1, id, a_{s,1}, v_1) \approx_L \mathsf{createSibling}_{\mathcal{G}}(\sigma_2, pc_2, id, a_{s,2}, v_2)$
- $\mathsf{registerEH}_{\mathcal{G}}(\sigma_1, pc_1, a_1, eh) \approx_L \mathsf{registerEH}_{\mathcal{G}}(\sigma_2, pc_2, a_2, eh)$

Proof.
By induction on the structure of $\mathcal{D}$ and $\mathcal{E}$
By assumption,
(1) $\sigma_1 \approx_L \sigma$
(2) $pc_1, pc_2 \sqsubseteq L$

$\mathcal{D} :: \mathsf{createElem}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1)$
$\mathcal{E} :: \mathsf{createElem}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2)$
Denote
$\sigma_1' = \mathsf{createElem}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1)$
$\sigma_2' = \mathsf{createElem}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2)$
Want to show $\sigma_1' \approx_L \sigma_2'$
Only the cases for $\mathcal{G} \neq \mathsf{TS}$ are considered. The other cases are proven in the weak secrecy version: Lemma 83.

Unstructured EH storage:
By assumption,
(U.1) $v_1 \approx_L v_2$

$\mathcal{G} = \mathsf{SMS}$
Case I: $\mathcal{D}$ ends in SMS-CREATE
By assumption,
(I.1) $\mathsf{NULL} = \mathsf{lookup}_{\mathsf{SMS}}(\sigma_1, pc_1, id)$
(I.2) $\sigma_{L,1} = \mathsf{getStore}(\sigma_1, pc_1)$
(I.3) $\sigma_1' = \mathsf{setStore}(\sigma_1, pc_1, \sigma_{L,1}[id \mapsto (v_1, \cdot)])$

Subcase i: $pc_2 = L$
From (1), (2), (I.1), and Lemma 48.U,
(i.1) $\mathsf{NULL} = \mathsf{lookup}_{\mathsf{SMS}}(\sigma_2, pc_2, id)$
By assumption and from (i.1),
(i.2) $\mathcal{E}$ ends in SMS-CREATE
From (i.2),
(i.3) $\sigma_{L,2} = \mathsf{getStore}(\sigma_2, pc_2)$
(i.4) $\sigma_2' = \mathsf{setStore}(\sigma_2, pc_2, \sigma_{L,2}[id \mapsto (v_2, \cdot)])$
From (1), (I.2), (i.3), and the definition of getStore,
(i.5) $\sigma_{L,1} = \sigma_{L,2}$
From (i.5), (I.3), (i.4), and the definition of setStore,
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $pc = \cdot$
By assumption,
(ii.1) $\mathcal{E}$ ends in SMS-CREATE-NC
From (ii.1),
(ii.2) $\sigma_2'' = \mathsf{createElem}_{\mathsf{SMS}}(\sigma_2, H, id, v_2)$
(ii.3) $\mathcal{E}' :: \sigma_2' = \mathsf{createElem}_{\mathsf{SMS}}(\sigma_2'', L, id, v_2)$
From (ii.2) and Lemma 74.U (Requirement **(EH3)**),
(ii.4) $\sigma_2 \approx_L \sigma_2''$

From (1) and (ii.3),
(ii.5) IH may be applied on $\mathcal{D}$ and $\mathcal{E}'$
From (ii.5) and IH on $\mathcal{D}$ and $\mathcal{E}'$,
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{D}$ ends in SMS-CREATE
The proof is similar to **Case I**

Case III: $\mathcal{D}$ ends in SMS-CREATE-U
By assumption,
(III.1) $\mathsf{lookup}_{\mathsf{SMS}}(\sigma_1, pc_1, id) \neq \mathsf{NULL}$
(III.2) $\sigma_1' = \mathsf{assign}_{\mathsf{SMS}}(\sigma_1, pc_1, id, v_1)$

Subcase i: $pc_2 = L$
From (1), (2), (III.1), and Lemma 48.U,
(i.1) $\mathsf{lookup}_{\mathsf{SMS}}(\sigma_2, pc_2, id) \neq \mathsf{NULL}$
From (i.1),
(i.2) $\mathcal{E}$ ends in SMS-CREATE-U
From (i.2),
(i.3) $\sigma_2' = \mathsf{assign}_{\mathsf{SMS}}(\sigma_2, pc_2, id, v_2)$
From (1), (2), (U.1), (III.2), (i.3), and Lemma 80.U (Requirement **(EH4)**),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $pc_2 = \cdot$
The proof for this case is similar to the one for **Subcase I.ii**.

Case IV: $\mathcal{E}$ ends in SMS-CREATE-U
The proof is similar to **Case III**.

Case V: $\mathcal{D}$ ends in SMS-CREATE-NC
By assumption,
(V.1) $\sigma_1'' = \mathsf{createElem}_{\mathsf{SMS}}(\sigma_1, H, id, v_1)$
(V.2) $\mathcal{D}' :: \sigma_1' = \mathsf{createElem}_{\mathsf{SMS}}(\sigma_1'', L, id, v_1)$
From (V.1) and Lemma 74.U (Requirement **(EH3)**),
(V.3) $\sigma_1 \approx_L \sigma_1''$
From (1) and (V.3),
(V.4) IH may be applied on $\mathcal{D}'$ and $\mathcal{E}$
From (V.4) and IH on $\mathcal{D}'$ and $\mathcal{E}$
$\sigma_1' \approx_L \sigma_2'$

Case VI: $\mathcal{E}$ ends in SMS-CREATE-NC
The proof is similar to **Case V**.

$\mathcal{G} = \mathsf{FS}$
Case I: $\mathcal{D}$ ends in FS-CREATE
By assumption,
(I.1) $\mathsf{NULL} = \mathsf{lookup}_{\mathsf{FS}}(\sigma_1, pc_1, id)$
(I.2) $\sigma_1' = \sigma_1[id \mapsto (\mathsf{getFacetV}(v_1, pc_1), \cdot)]$

Subcase i: $pc_2 = L$
From (1), (2), (I.1), and Lemma 48.U (Requirement **(EH1)**),
(i.1) $\mathsf{NULL} = \mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id)$
By assumption and from (i.1),
(i.2) $\mathcal{E}$ ends in FS-CREATE
From (i.2),
(i.3) $\sigma_2' = \sigma_2[id \mapsto (\mathsf{getFacetV}(v_2, pc_2), \cdot)]$
From (1), (U.1), (I.2), (i.3), and the definition of $\mathsf{getFacetV}$,
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $pc = \cdot$
By assumption,
(ii.1) $\mathcal{E}$ ends in FS-CREATE-NC
From (ii.1),

(ii.2) $\sigma_2'' = \mathsf{createElem}_{\mathsf{FS}}(\sigma_2, H, id, v_2)$
(ii.3) $\mathcal{E}' :: \sigma_2' = \mathsf{createElem}_{\mathsf{FS}}(\sigma_2'', L, id, v_2)$
From (ii.2) and Lemma 74.U (Requirement **(EH3)**),
(ii.4) $\sigma_2 \approx_L \sigma_2''$
From (1) and (ii.3),
(ii.5) IH may be applied on $\mathcal{D}$ and $\mathcal{E}'$
From (ii.5) and IH on $\mathcal{D}$ and $\mathcal{E}'$,
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{E}$ ends in FS-CREATE
The proof is similar to **Case I**.

Case III: $\mathcal{D}$ ends in FS-CREATE-U
By assumption,
(III.1) $\mathsf{lookup}_{\mathsf{FS}}(\sigma_1, pc_1, id) = \phi_1 \neq \mathsf{NULL}$
(III.2) $\sigma_1' = \mathsf{assign}_{\mathsf{FS}}(\sigma_1, pc_1, id, v_1)$
From (1), (2), (III.1), and Lemma 48.U (Requirement **(EH1)**),
(III.3) $\mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id) \approx_L \phi_1$

Subcase i: $\mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id) \neq \langle \_ | \_ \rangle$
By assumption and from (III.1) and (III.3),
(i.1) $\mathcal{E}$ ends in FS-CREATE-U
From (i.1),
(i.2) $\sigma_2' = \mathsf{assign}_{\mathsf{FS}}(\sigma_2, pc_2, id, v_2)$
From (1), (2), (U.1), (III.2), (i.2), and Lemma 80.U (Requirement **(EH4)**),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $pc_2 = \cdot$
The proof for this case is similar to the one for **Subcase I.ii**.

Case IV: $\mathcal{E}$ ends in FS-CREATE-U
The proof is similar to **Case III**.

Case V: $\mathcal{D}$ ends in FS-CREATE-NC
By assumption,
(V.1) $\sigma_1'' = \mathsf{createElem}_{\mathsf{FS}}(\sigma_1, H, id, v_1)$
(V.2) $\mathcal{D}' :: \sigma_1' = \mathsf{createElem}_{\mathsf{FS}}(\sigma_1'', L, id, v_1)$
From (V.1) and Lemma 74.U (Requirement **(EH3)**),
(V.3) $\sigma_1 \approx_L \sigma_1''$
From (1) and (V.2),
(V.4) IH may be applied on $\mathcal{D}'$ and $\mathcal{E}$
From (V.4) and IH on $\mathcal{D}'$ and $\mathcal{E}$,
$\sigma_1' \approx_L \sigma_2'$

Case VI: $\mathcal{E}$ ends in FS-CREATE-NC
The proof is similar to **Case V**.

$\mathcal{D} :: \mathsf{registerEH}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1)$
$\mathcal{E} :: \mathsf{registerEH}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2)$
Denote
$\sigma_1' = \mathsf{registerEH}_{\mathcal{G}}(\sigma_1, pc_1, id, eh)$
$\sigma_2' = \mathsf{registerEH}_{\mathcal{G}}(\sigma_2, pc_2, id, eh)$
Want to show $\sigma_1' \approx_L \sigma_2'$

$\mathcal{G} = \mathsf{SMS}$
Case I: $\mathcal{D}$ ends in SMS-REGISTEREH
By assumption,
(I.1) $pc_1 = L$
(I.2) $(v, M) = \mathsf{lookup}_{\mathsf{SMS}}(\sigma_1, L, id)$
(I.3) $eh = \mathsf{on}Ev(x)\{c\}$
(I.4) $M' = M[Ev \mapsto M(Ev) \cup \{(eh, L)\}]$

(I.5) $\sigma_{L,1} = \mathsf{getStore}(\sigma_1, L)$
(I.6) $\sigma_1' = \mathsf{setStore}(\sigma_1, L, \sigma_{L,1}[id, v, M'])$

Subcase i: pc_2 = L
From (1), (2), and (I.2), and Lemma 48.U (Requirement **(EH1)**) and the definition of $\approx_L$ for SMS nodes,
(i.1) $(v, M) = \mathsf{lookup}_{\mathsf{SMS}}(\sigma_2, L, id)$
By assumption and from (i.1),
(i.2) $\mathcal{E}$ ends in SMS-REGISTEREH
From (i.2) and (I.4),
(i.3) $\sigma_{L,2} = \mathsf{getStore}(\sigma_2, L)$
(i.4) $\sigma_2' = \mathsf{setStore}(\sigma_2, L, \sigma_{L,2}[id, v, M'])$
From (1), (I.5), and (i.3),
(i.5) $\sigma_{L,1} = \sigma_{L,2}$
From (i.5), (I.6), (i.4), and the definition of setStore,
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $pc_2 = \cdot$
By assumption,
(ii.1) $\mathcal{E}$ ends in SMS-REGISTEREH-NC
From (ii.1),
(ii.2) $\sigma_2'' = \mathsf{registerEH}_{\mathsf{SMS}}(\sigma_2, H, id, eh)$
(ii.3) $\mathcal{E}' :: \sigma_2' = \mathsf{registerEH}_{\mathsf{SMS}}(\sigma_2'', L, id, eh)$
From (ii.2) and Lemma 74.U (Requirement **(EH3)**),
(ii.4) $\sigma_2 \approx_L \sigma_2''$
From (1) and (ii.4),
(ii.5) IH may be applied to $\mathcal{D}$ and $\mathcal{E}'$
From (ii.5) and IH on $\mathcal{D}$ and $\mathcal{E}'$,
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{E}$ ends in SMS-REGISTEREH
The proof is similar **Case I**.

Case III: $\mathcal{D}$ ends in SMS-REGISTEREH-S
By assumption,
(III.1) $\mathsf{lookup}_{\mathsf{SMS}}(\sigma_1, L, id) = \mathsf{NULL}$
(III.2) $\sigma_1' = \sigma_1$

Subcase i: $pc_2 = L$
From (1), (2), and (III.2), and Lemma 48.U (Requirement **(EH1)**) and the definition of $\approx_L$ for SMS nodes,
(i.1) $\mathsf{NULL} = \mathsf{lookup}_{\mathsf{SMS}}(\sigma_2, L, id)$
By assumption and from (i.1),
(i.2) $\mathcal{E}$ ends in SMS-REGISTEREH-S
From (i.2),
(i.3) $\sigma_2' = \sigma_2$
From (1), (III.2), and (i.3),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $pc_2 = \cdot$
The proof for this case is similar to **Subcase I.ii**.

Case IV: $\mathcal{E}$ ends in SMS-REGISTEREH-S
The proof is similar to **Case III**.

Case V: $\mathcal{D}$ ends in SMS-REGISTEREH-NC
By assumption,
(V.1) $\sigma_1'' = \mathsf{registerEH}_{\mathsf{SMS}}(\sigma_1, H, id, eh)$
(V.2) $\mathcal{D}' :: \sigma_1' = \mathsf{registerEH}_{\mathsf{SMS}}(\sigma_1'', L, id, eh)$
From (V.1) and Lemma 74.U (Requirement **(EH3)**),
(V.3) $\sigma_1 \approx_L \sigma_1''$
From (1) and (V.3),
(V.4) the IH may be applied on $\mathcal{D}'$ and $\mathcal{E}$
From (V.4) and the IH on $\mathcal{D}'$ and $\mathcal{E}$,
$\sigma_1' \approx_L \sigma_2'$

Case VI: $\mathcal{E}$ ends in SMS-EGISTEREH-NC
The proof is similar to **Case V**.

$\mathcal{G} = \mathsf{FS}$
Case I: $\mathcal{D}$ ends in FS-REGISTEREH
By assumption,
(I.1) $(v_1, M_1) = \mathsf{lookup}_{\mathsf{FS}}(\sigma_1, L, id)$
(I.2) $eh = \mathsf{on}Ev(x)\{c\}$
(I.3) $M_1' = M_1[Ev \mapsto M_1(Ev) \cup \{(eh, pc_1)\}]$
(I.4) $\sigma_1' = \sigma_1[id \mapsto (v_1, M_1')]$
From (1), (2), and (I.1), and Lemma 48.U (Requirement **(EH1)**) and the definition of $\approx_L$ for FS nodes,
(I.5) $\phi_2 = \mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id)$ with $\phi_2 \approx_L (v_1, M_1)$ or
(I.6) $\langle \_ | \phi_2 \rangle = \mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id)$ with $\phi_2 \approx_L (v_2, M_1)$
From (I.5) and (I.6), $\mathcal{E}$ ends in FS-REGISTEREH or FS-REGISTEREH-NC

Subcase i: $\mathcal{E}$ ends in FS-REGISTEREH
By assumption and from (I.5),
(i.1) $(v_2, M_2) = \mathsf{lookupEH}_{\mathsf{FS}}(\sigma_2, pc_2, id)$
(i.2) $M_2' = M_2[Ev \mapsto M_2(Ev) \cup \{(eh, pc_2)\}]$
(i.3) $\sigma_2' = \sigma_2[id \mapsto (v_2, M_2')]$
From (I.5) and (i.1),
(i.4) $M_1 \approx_L M_2$
From (2), (i.4), (I.3), and (i.2),
(i.5) $M_1' \approx_L M_2'$
From (I.5), (I.1), and (i.1),
(i.6) $v_1 \approx_L v_2$
From (1), (I.4), (i.3), (i.5), and (i.6),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $\mathcal{E}$ ends in FS-REGISTEREH-NC
By assumption,
(ii.1) $\sigma_2'' = \mathsf{registerEH}_{\mathsf{FS}}(\sigma_2, H, id, Ev)$
(ii.2) $\mathcal{E}' :: \sigma_2' = \mathsf{registerEH}_{\mathsf{FS}}(\sigma_2'', L, id, Ev)$
From (ii.1) and Lemma 74.U (Requirement **(EH3)**),
(ii.3) $\sigma_2 \approx_L \sigma_2''$
From (1) and (ii.3),
(ii.4) IH may be applied on $\mathcal{D}$ and $\mathcal{E}'$
From (ii.4) and IH on $\mathcal{D}$ and $\mathcal{E}'$,
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{E}$ ends in FS-REGISTEREH
The proof is similar **Case I**.

Case III: $\mathcal{D}$ ends in FS-REGISTEREH-S
By assumption,
(III.1) $\mathsf{lookup}_{\mathsf{FS}}(\sigma_1, pc_1, id) = \mathsf{NULL}$
(III.2) $\sigma_1' = \sigma_1$
From (1), (2), and (I.1), and Lemma 48.U (Requirement **(EH1)**) and the definition of $\approx_L$ for FS nodes,
(III.3) $\phi_2 = \mathsf{NULL}$ or
(III.4) $\langle \_ | \mathsf{NULL} \rangle$
From (III.3) and (III.4), $\mathcal{E}$ ends in FS-REGISTEREH-S or FS-REGISTEREH-NC

Subcase i: $\mathcal{E}$ ends in FS-REGISTEREH-S
By assumption,
(i.1) $\sigma_2' = \sigma_2$
From (1), (III.2), and (i.1),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $\mathcal{E}$ ends in FS-REGISTEREH-NC
The proof is similar to **Subcase I.ii**.

Case IV: $\mathcal{E}$ ends in FS-REGISTEREH-S

The proof is similar to **Case III**.

Case V: $\mathcal{D}$ ends in FS-REGISTEREH-NC
By assumption,
(V.1) $\sigma_1'' = \mathsf{registerEH}_{\mathsf{FS}}(\sigma_1, H, id, eh)$
(V.2) $\mathcal{D}' :: \sigma_1' = \mathsf{registerEH}_{\mathsf{FS}}(\sigma_1'', L, id, eh)$
From (V.1) and Lemma 74.U (Requirement **(EH3)**),
(V.3) $\sigma_1 \approx_L \sigma_1''$
From (1) and (V.3),
(V.4) IH may be applied on $\mathcal{D}'$ and $\mathcal{E}$
From (V.4) and IH on $\mathcal{D}'$ and $\mathcal{E}$
$\sigma_1' \approx_L \sigma_2'$

Case VI: $\mathcal{E}$ ends in FS-REGISTEREH-NC
The proof is similar to **Case V**.

$\mathcal{G} = \mathsf{TS}$
Case I: $\mathcal{D}$ ends in TS-REGISTEREH
By assumption and from our security lattice,
(I.1) $(v_1, M_1, l_1) = \mathsf{lookup}_{\mathsf{TS}}(\sigma_1, L, id)$
(I.2) $eh = \mathsf{on}Ev(x)\{c\}$
(I.3) $M_1' = M_1[Ev \mapsto M_1(Ev) \cup \{(eh, l_1)\}]$
(I.4) $\sigma_1' = \sigma_1[id \mapsto (v_1, M_1', l_1)]$
From (1), (2), and (I.1), and Lemma 48.U (Requirement **(EH1)**) and the definition of $\approx_L$ for TS nodes,
(I.5) $(v_2, M_2, l_2) = \mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id)$ with $l_1, l_2 \sqsubseteq L$ and $v_1 \approx_L v_2$, $M_1 \approx_L M_2$ or
(I.6) $\phi_2 = \mathsf{lookup}_{\mathsf{FS}}(\sigma_2, pc_2, id)$ with $l_1, \mathsf{labOf}(\phi_2, \_) \not\sqsubseteq L$
From (2), (I.5), and (I.6), $\mathcal{E}$ may end in TS-REGISTEREH, TS-REGISTEREH-NC, or TS-REGISTEREH-S

Subcase i: $\mathcal{E}$ ends in TS-REGISTEREH
By assumption and from our security lattice,
(i.1) $(v_2, M_2, l_2) = \mathsf{lookup}_{\mathsf{TS}}(\sigma_2, L, id)$
(i.2) $eh = \mathsf{on}Ev(x)\{c\}$
(i.3) $M_2' = M_2[Ev \mapsto M_2(Ev) \cup \{(eh, l_2)\}]$
(i.4) $\sigma_2' = \sigma_2[id \mapsto (v_2, M_2', l_2)]$
From (I.5) and (I.6), either
(i.1) $l_1, l_2 \sqsubseteq L$ and $v_1 \approx_L v_2$, $M_1 \approx_L M_2$ or
(i.2) $l_1, l_2 \not\sqsubseteq L$

Subsubcase a: (i.1) is true
By assumption and from (I.3), (i.3), and our security lattice,
(a.1) $M_1' = M_1[Ev \mapsto M_1(Ev) \cup \{(eh, L)\}]$
(a.2) $M_2' = M_2[Ev \mapsto M_2(Ev) \cup \{(eh, L)\}]$
From (i.1), (a.1), and (a.2),
(a.3) $M_1' \approx_L M_2'$
From (1), (I.4), (i.4), (i.1), and (a.3),
$\sigma_1' \approx_L \sigma_2'$

Subsubcase b: (i.2) is true
By assumption and from (1), (I.4), and (i.4),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $\mathcal{E}$ ends in TS-REGISTEREH-S
By assumption,
(ii.1) $\sigma_2' = \sigma_2$
By assumption and from (I.1), (I.6) must be true
From (I.4) and (I.6),
(ii.2) $\sigma_1' \approx_L \sigma_1$
From (1), (ii.1), and (ii.2),
$\sigma_1' \approx_L \sigma_2'$

Subcase iii: $\mathcal{E}$ ends in TS-REGISTEREH-NC
By assumption,
(iii.1) $\sigma_2'' = \mathsf{registerEH}_{\mathsf{TS}}(\sigma_2, H, id, eh)$

(iii.2) $\mathcal{E}' :: \sigma_2' = \mathsf{registerEH_{TS}}(\sigma_2'', L, id, eh)$
From (iii.1) and Lemma 74.U (Requirement **(EH3)**),
(iii.3) $\sigma_2'' \approx_L \sigma_2$
From (1) and (iii.3),
(iii.4) IH may be applied on $\mathcal{D}$ and $\mathcal{E}'$
From (iii.4) and IH on $\mathcal{D}$ and $\mathcal{E}'$,
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{E}$ ends in TS-REGISTEREH
The proof is similar **Case I**.

Case III: $\mathcal{D}$ ends in TS-REGISTEREH-S
By assumption,
(I.1) $(\mathsf{NULL}, l_1) = \mathsf{lookup_{TS}}(\sigma_1, L, id)$
(I.2) $\sigma_1' = \sigma_1$
From (1), (2), and (I.1), and Lemma 48.U (Requirement **(EH1)**) and the definition of $\approx_L$ for TS nodes,
(I.3) $(\mathsf{NULL}, l_2) = \mathsf{lookup_{FS}}(\sigma_2, pc_2, id)$ with $l_1, l_2 \sqsubseteq L$ or
(I.4) $\phi_2 = \mathsf{lookup_{FS}}(\sigma_2, pc_2, id)$ with $l_1, \mathsf{labOf}(\phi_2, \_) \not\sqsubseteq L$
From (2), (I.3), and (I.4), $\mathcal{E}$ may end in TS-REGISTEREH, TS-REGISTEREH-NC, or TS-REGISTEREH-S

Subcase i: $\mathcal{E}$ ends in TS-REGISTEREH
By assumption,
(i.1)$(v_2, M_2, l_2) = \mathsf{lookup_{TS}}(\sigma_2, L, id)$
(i.2) $eh = \mathsf{on}Ev(x)\{c\}$
(i.3) $M_2' = M_2[Ev \mapsto M_2(Ev) \cup \{(eh, l_2)\}]$
(i.4) $\sigma_2' = \sigma_2[id \mapsto (v_2, M_2', l_2)]$
From (i.1), (I.3), and (I.4), (I.4) must be true
From (I.4) and (i.4),
(i.5) $\sigma_2 \approx_L \sigma_2'$
From (1), (I.2), and (i.5),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $\mathcal{E}$ ends in TS-REGISTEREH-S
By assumption,
(ii.1) $\sigma_2' = \sigma_2$
From (1), (I.2), and (ii.1),
$\sigma_1' \approx_L \sigma_2'$

Subcase iii: $\mathcal{E}$ ends in TS-REGISTEREH-NC
The proof for this case is similar to **Subcase I.iii**.

Case IV: $\mathcal{E}$ ends in TS-REGISTEREH-S
The proof is similar to **Case III**.

Case V: $\mathcal{D}$ ends in TS-REGISTEREH-NC
By assumption,
(V.1) $\sigma_1'' = \mathsf{registerEH_{TS}}(\sigma_1, H, id, eh)$
(V.2) $\mathcal{D}' :: \sigma_1' = \mathsf{registerEH_{TS}}(\sigma_1'', L, id, eh)$
From (V.1) and Lemma 74.U (Requirement **(EH3)**),
(V.3) $\sigma_1 \approx_L \sigma_1''$
From (1) and (V.3),
(V.4) IH can be applied on $\mathcal{D}'$ and $\mathcal{E}$
From (V.4) and IH on $\mathcal{D}'$ and $\mathcal{E}$,
$\sigma_1' \approx_L \sigma_2'$

Case VI: $\mathcal{E}$ ends in TS-REGISTEREH-NC
The proof is similar to **Case V**.

Tree-structured EH storage:
By assumption,
(T.1) $v_1 \approx_L^{\sigma_1,\sigma_2} v_2$
(T.2) $a_{p,1} \approx_L^{\sigma_1,\sigma_2} a_{p,2}$
(T.3) $a_{s,1} \approx_L^{\sigma_1,\sigma_2} a_{s,2}$
(T.4) $a_1 \approx_L^{\sigma_1,\sigma_2} a_2$
From (1),
(T.5) $a_1^{\mathsf{rt}} \approx_L^{\sigma_1,\sigma_2} a_2^{\mathsf{rt}}$

$\mathcal{D} :: \sigma_1' = \mathsf{createChild}_{\mathcal{G}}(\sigma_1, pc_1, id, a_{p,1}, v_1)$
$\mathcal{E} :: \sigma_2' = \mathsf{createChild}_{\mathcal{G}}(\sigma_2, pc_2, id, a_{p,1}, v_2)$
Want to show $\sigma_1' \approx_L \sigma_2'$

$\mathcal{G} = \mathsf{SMS}$
Denote
$\sigma_{L,1} = \mathsf{getStore}(\sigma_1, L)$
$\sigma_{L,2} = \mathsf{getStore}(\sigma_2, L)$

Case I: $\mathcal{D}$ or $\mathcal{E}$ ends in SMS-CREATEC
Without loss of generality, assume that $\mathcal{D}$ ends in SMS-CREATEC. The proof for $\mathcal{E}$ is similar.
By assumption,
(I.1) $\mathsf{lookupA}_{\mathsf{SMS}}(\sigma_1, pc_1, id, a_1^{\mathsf{rt}}) = \mathsf{NULL}$
(I.2) $a_1 \notin \sigma_{L,1}$
(I.3) $\sigma_{L,1}(a_{p,1}) = (id_{p,1}, v_{p,1}, M_1, a'_{p,1}, A_1)$
(I.4) $\sigma'_{L,1} = \sigma_{L,1}[a_{p,1} \mapsto (id_{p,1}, v_{p,1}, M_1, a'_{p,1}, (a_1 :: A_1))$
(I.5) $\sigma''_{L,1} = \sigma'_{L,1}[a_1 \mapsto (id, v_1, \cdot, a_{p,1}, \cdot)$
(I.6) $\sigma_1' = \mathsf{setStore}(\sigma_1, L, \sigma''_{L,1})$

Subcase i: $pc_2 = L$
By assumption and from (1), (2), (I.1), (T.5), and Lemma 49 (Requirement **(EH1)**) and the definition of $\approx_L$ for SMS nodes, either
(i.1) $\mathsf{lookupA}_{\mathsf{SMS}}(\sigma_2, pc_2, id, a_2^{\mathsf{rt}}) = \mathsf{NULL}$
From (T.2) and (I.3),
(i.2) $\sigma_{L,2}(a_{p,2}) \approx_L^{\sigma_2,\sigma_1} \sigma_{L,1}(a_{p,1})$
From (I.3) and (i.2),
(i.3) $\sigma_{L,2}(a_{p,2}) = (id, v_{p,2}, M_2, a'_{p,2}, A_2)$
From (i.1) and (i.3),
(i.4) $\mathcal{E}$ ends in SMS-CREATEC
From (i.4),
(i.5) $a_2 \notin \sigma_{L,2}$
(i.6) $\sigma'_{L,2} = \sigma_{L,2}[a_{p,2} \mapsto (id_{p,2}, v_{p,2}, M_1, a'_{p,2}, (a_2 :: A_2))$
(i.7) $\sigma''_{L,2} = \sigma'_{L,2}[a_2 \mapsto (id, v_2, \cdot, a_{p,2}, \cdot)$
(i.8) $\sigma_2' = \mathsf{setStore}(\sigma_2, L, \sigma''_{L,2})$
From (1), (I.4), and (i.6),
(i.9) $\sigma'_{L,1} \approx_L \sigma'_{L,2}$
From (i.9), (I.5), and (i.7),
(i.10) $\sigma''_{L,1} \approx_L \sigma''_{L,2}$
From (I.6), (i.8), (i.10), and the definition of setStore,
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $pc_2 = \cdot$
By assumption,
(ii.1) $\mathcal{E}$ ends in SMS-CREATEC-NC
From (ii.1),
(ii.2) $\sigma_2'' = \mathsf{createChild}_{\mathsf{SMS}}(\sigma_2, H, id, \mathsf{getFacetA}(a_{p,2}, H), \mathsf{getFacetV}(v_2, H))$
(ii.3) $\mathcal{E}' :: \sigma_2' = \mathsf{createChild}_{\mathsf{SMS}}(\sigma_2'', L, id, \mathsf{getFacetA}(a_{p,2}, L), \mathsf{getFacetV}(v_2, L))$
From (ii.2) and Lemma 74.T (Requirement **(EH3)**),
(ii.4) $\sigma_2 \approx_L \sigma_2''$
From (1) and (ii.4),
(ii.5) $\sigma_1 \approx_L \sigma_2''$
From (ii.5), (2), (T.1), (T.2), and the definition of getFacetA and getFacetV,
(ii.6) the IH may be applied on $\mathcal{D}$ and $\mathcal{E}'$

From (ii.6) and the IH,
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{D}$ or $\mathcal{E}$ ends in SMS-CREATEC-NC
Without loss of generality, assume that $\mathcal{D}$ ends in SMS-CREATEC-NC. The proof for $\mathcal{E}$ is similar.
By assumption,
(II.1) $\sigma_1'' = \mathsf{createChild}_{\mathsf{SMS}}(\sigma_1, H, id, \mathsf{getFacetA}(a_{p,1}, H), \mathsf{getFacetV}(v_1, H))$
(II.2) $\mathcal{E}' :: \sigma_1' = \mathsf{createChild}_{\mathsf{SMS}}(\sigma_1'', L, id, \mathsf{getFacetA}(a_{p,1}, L), \mathsf{getFacetV}(v_1, L))$
From (II.1) and Lemma 74.T (Requirement **(EH3)**),
(II.3) $\sigma_1 \approx_L \sigma_1''$
From (1) and (II.3),
(II.4) $\sigma_1'' \approx_L \sigma_2$
From (II.4), (2), (T.1), (T.2), and the definition of getFacetA and getFacetV,
(II.5) the IH may be applied on $\mathcal{D}'$ and $\mathcal{E}$
From (II.5) and the IH,
$\sigma_1' \approx_L \sigma_2'$

Case III: $\mathcal{D}$ or $\mathcal{E}$ ends in SMS-CREATEC-S
Without loss of generality, assume that $\mathcal{D}$ ends in SMS-CREATEC-S. The proof for $\mathcal{E}$ is similar.
By assumption,
(III.1) $\mathsf{lookupA}_{\mathsf{SMS}}(\sigma_1, pc_1, id, a_1^{\mathsf{rt}}) \neq \mathsf{NULL}$ or
(III.2) $a_{p,1} = \mathsf{NULL}$
(III.3) $\sigma_1' = \sigma_1$

Subcase i: $pc_2 = \cdot$
The proof for this case is covered by **Case II**.

Subcase ii: $pc_2 = L$
If (III.1) is true, then by assumption and from (1), (2), (T.5), (III.1) and Lemma 49 (Requirement **(EH1)**),
(ii.1) $\mathsf{lookupA}_{\mathsf{SMS}}(\sigma_2, pc_2, id, a_2^{\mathsf{rt}}) \neq \mathsf{NULL}$
If (III.2) is true, then from (T.2),
(ii.2) $a_{p,2} = \mathsf{NULL}$
Whether (ii.1) or (ii.2) is true,
(ii.3) $\sigma_2' = \sigma_2$
From (1), (III.3), and (ii.3),
$\sigma_1' \approx_L \sigma_2'$

$\mathcal{G} = \mathsf{FS}$
Case I: $\mathcal{D}$ or $\mathcal{E}$ ends in FS-CREATEC
Without loss of generality, assume that $\mathcal{D}$ ends in FS-CREATEC. The proof for $\mathcal{E}$ is similar.
By assumption,
(I.1) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_1, \cdot, id, a_1^{\mathsf{rt}}) = \mathsf{NULL}$
(I.2) $a_1 \notin \sigma_1$
(I.3) $\sigma_1(a_{p,1}) = (id_p, v_{p,1}, M_1, a'_{p,1}, A_1)$
(I.4) $v_{p,1} \downarrow_{pc_1} \neq \cdot$
(I.5) $\sigma_1'' = \sigma_1[a_{p,1} \mapsto (id_p, v_{p,1}, M_1, a'_{p,1}, (\mathsf{createFacet}(a_1, pc_1) :: A_1))]$
(I.6) $\sigma_1' = \sigma_1''[a_1 \mapsto (id, \mathsf{createFacet}(v_1, pc_1), \cdot, \mathsf{createFacet}(a_{p,1}, pc_1), \cdot)]$
From (1), (I.1), (T.5), and Lemma 49 (Requirement **(EH1)**),
(I.7) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_2, \cdot, id, a_2^{\mathsf{rt}}) = \mathsf{NULL}$ or
(I.8) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_2, \cdot, id, a_2^{\mathsf{rt}}) = \langle a_2 | \mathsf{NULL} \rangle$
From (T.2) and (I.3),
(I.9) $\sigma_2(a_{p,2}) = (id_p, v_{p,2}, M_2, a'_{p,2}, A_2)$ with
(I.10) $(id_p, v_{p,1}, M_1, a'_{p,1}, A_1) \approx_L^{\sigma_1, \sigma_2} (id_p, v_{p,2}, M_2, a'_{p,2}, A_2)$
From (I.4) and (I.10),
(I.11) $v_{p,2} \downarrow_{pc_2} \neq \cdot$

Subcase i: (I.7) is true and $pc_2 = L$
By assumption and from (I.7), (I.9), and (I.11),
(i.1) $\mathcal{E}$ ends in FS-CREATEC
From (i.1),
(i.2) $a_2 \notin \sigma_2$
(i.3) $\sigma_2'' = \sigma_2[a_{p,2} \mapsto (id_p, v_{p,2}, M_2, a'_{p,2}, (\mathsf{createFacet}(a_2, pc_2) :: A_2))]$

(i.4) $\sigma_2' = \sigma_2''[a_2 \mapsto (id, \mathsf{createFacet}(v_2, pc_2), \cdot, \mathsf{createFacet}(a_{p,2}, pc_2), \cdot)]$
From (T.1), (T.2), (I.5), (i.3), and the definition of createFacet,
(i.5) $(id, \mathsf{createFacet}(v_1, pc_1), \cdot, \mathsf{createFacet}(a_{p,1}, pc_1), \cdot) \approx_L^{\sigma_1', \sigma_2'}$
$(id, \mathsf{createFacet}(v_2, pc_2), \cdot, \mathsf{createFacet}(a_{p,2}, pc_2), \cdot)$
From (1), (I.10), (I.5), (i.3), (i.5), and the definition of createFacet,
(i.6) $\sigma_1'' \approx_L \sigma_2''$
From (i.6), (I.2), (i.2), (I.6), (i.4), and (i.5),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: (I.8) is true and $pc_2 = L$
By assumption and from (I.8), (I.9), and (I.11),
(ii.1) $\mathcal{E}$ ends in FS-CREATEC-UL
From (ii.1),
(ii.2) $\sigma_2(a_2) = (id, v_2', M_2', a_{p,2}'', A_2')$
(ii.3) $\sigma_2'' = \sigma_2[a_{p,2} \mapsto (id_p, v_{p,2}, M_2, a_{p,2}', (\mathsf{createFacet}(a_2, pc_2) :: A_2))]$
(ii.4) $\sigma_2' = \sigma_2''[a_2 \mapsto (id, \mathsf{updateFacet}(v_2', v_2, L), M_2', \mathsf{updateFacet}(a_{p,2}'', a_{p,2}, L), A_2')]$
From (T.1), (T.2), (I.5), (ii.3), (ii.4), and the definitions of createFacet and updateFacet,
(ii.5) $(id, \mathsf{createFacet}(v_1, pc_1), \cdot, \mathsf{createFacet}(a_{p,1}, pc_1), \cdot) \approx_L^{\sigma_1', \sigma_2'}$
$(id, \mathsf{updateFacet}(v_2', v_2, L), M_2', \mathsf{updateFacet}(a_{p,2}'', a_{p,2}, L), A_2')$
From (1), (I.10), (I.5), (ii.3), (ii.5), and the definition of createFacet,
(ii.6) $\sigma_1'' \approx_L \sigma_2''$
From (ii.6), (I.2), (ii.2), (I.6), (ii.4), and (ii.5),
$\sigma_1' \approx_L \sigma_2'$

Subcase iii: $pc_2 = \cdot$
By assumption,
(iii.1) $\mathcal{E}$ ends in FS-CREATEC-NC
From (iii.1),
(iii.2) $\sigma_2'' = \mathsf{createChild}_{\mathsf{FS}}(\sigma_2, H, id, \mathsf{getFacetA}(a_{p,2}, H), \mathsf{getFacetV}(v_2, H))$
(iii.3) $\mathcal{E}' :: \sigma_2' = \mathsf{createChild}_{\mathsf{FS}}(\sigma_2'', L, id, \mathsf{getFacetA}(a_{p,2}, L), \mathsf{getFacetV}(v_2, L))$
From (iii.2) and Lemma 74.T (Requirement **(EH3)**),
(iii.4) $\sigma_2 \approx_L \sigma_2''$
From (1) and (iii.4),
(iii.5) $\sigma_1 \approx_L \sigma_2''$
From (iii.5), (2), (T.1), (T.2), (iii.3), and the definition of getFacetA and getFacetV,
(iii.6) IH may be applied on $\mathcal{D}$ and $\mathcal{E}'$
From (iii.6) and the IH,
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{D}$ or $\mathcal{E}$ ends in FS-CREATEC-UL
Without loss of generality, assume that $\mathcal{D}$ ends in FS-CREATEC-UL. The proof for $\mathcal{E}$ is similar.
By assumption,
(II.1) $\mathsf{lookup}_{\mathsf{FS}}(\sigma_1, \cdot, id, a_1^{\mathsf{rt}}) = \langle a_1 | \mathsf{NULL} \rangle$
(II.2) $\sigma_1(a_1) = (id, v_1', M_1, a_{p,1}', A_1)$
(II.3) $\sigma_1(a_{p,1}) = (id_p, v_1'', M_1', a_{p,1}'', A_1')$
(II.4) $v_{p,1} \downarrow_L \neq \cdot$
(II.5) $\sigma_1'' = \sigma_1[a_{p,1} \mapsto (id_p, v_1'', M_1', a_{p,1}'', (\mathsf{createFacet}(a_1, L) :: A_1'))]$
(II.6) $\sigma_1' = \sigma_1''[a_1 \mapsto (id, \mathsf{updateFacet}(v_1'', v_1, L), M_1, \mathsf{updateFacet}(a_{p,1}', a_{p,1}, L), A_1)]$
From (1), (II.1), (T.5), and Lemma 49 (Requirement **(EH1)**),
(II.7) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_2, \cdot, id, a_2^{\mathsf{rt}}) = \mathsf{NULL}$ or
(II.8) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_2, \cdot, id, a_2^{\mathsf{rt}}) = \langle a_2 | \mathsf{NULL} \rangle$
From (T.2) and (II.3),
(II.9) $\sigma_2(a_{p,2}) = (id_p, v_{p,2}, M_2, a_{p,2}', A_2)$ with
(II.10) $(id_p, v_{p,1}, M_1, a_{p,1}', A_1) \approx_L^{\sigma_1, \sigma_2} (id_p, v_{p,2}, M_2, a_{p,2}', A_2)$
From (II.4) and (II.10),
(II.11) $v_{p,2} \downarrow_{pc_2} \neq \cdot$

Subcase i: (II.7) is true and $pc_2 = L$
By assumption and from (II.7), (II.9), and (II.11), $\mathcal{E}$ ends in FS-CREATEC. This case is covered by **Case I**.

Subcase ii: (II.8) is true and $pc_2 = L$

By assumption and from (II.8), (II.9), and (II.11),
(ii.1) $\mathcal{E}$ ends in FS-CREATEC-UL
From (ii.1),
(ii.2) $\sigma_2(a_2) = (id, v_2', M_2', a_{p,2}'', A_2')$
(ii.3) $\sigma_2'' = \sigma_2[a_{p,2} \mapsto (id_p, v_{p,2}, M_2, a_{p,2}', (\mathsf{createFacet}(a_2, pc_2) :: A_2))]$
(ii.4) $\sigma_2' = \sigma_2''[a_2 \mapsto (id, \mathsf{updateFacet}(v_2', v_2, L), M_2', \mathsf{updateFacet}(a_{p,2}'', a_{p,2}, L), A_2')]$
From (T.1), (T.2), (II.5), (ii.3), (ii.4), and the definition of updateFacet,
(ii.5) $(id, \mathsf{updateFacet}(v_1'', v_1, L), M_1, \mathsf{updateFacet}(a_{p,1}', a_{p,1}, L), A_1) \approx_L^{\sigma_1', \sigma_2'}$
$(id, \mathsf{updateFacet}(v_2', v_2, L), M_2', \mathsf{updateFacet}(a_{p,2}'', a_{p,2}, L), A_2')$
From (1), (II.10), (II.5), (ii.3), (ii.5), and the definition of createFacet,
(ii.6) $\sigma_1'' \approx_L \sigma_2''$
From (ii.6), (II.2), (ii.2), (II.6), (ii.4), and (ii.5),
$\sigma_1' \approx_L \sigma_2'$

Subcase iii: $pc_2 = \cdot$
The proof for this case is similar to **Subcase I.iii**

Case III: $\mathcal{D}$ or $\mathcal{E}$ ends in FS-CREATEC-NC
Without loss of generality, assume that $\mathcal{D}$ ends in FS-CREATEC-NC. The proof for $\mathcal{E}$ is similar.
By assumption,
(III.1) $\sigma_1'' = \mathsf{createChild}_{\mathsf{FS}}(\sigma_1, H, id, \mathsf{getFacetA}(a_{p,1}, H), \mathsf{getFacetV}(v_1, H))$
(III.2) $\mathcal{D}' :: \sigma_1' = \mathsf{createChild}_{\mathsf{FS}}(\sigma_1, L, id, \mathsf{getFacetA}(a_{p,1}, L), \mathsf{getFacetV}(v_1, L))$
From (III.1) and Lemma 74.T (Requirement **(EH3)**),
(III.3) $\sigma_1 \approx_L \sigma_1'''$
From (1) and (III.3),
(III.4) $\sigma_1'' \approx_L \sigma_2$
From (III.4), (2), (T.1), (T.2), (III.2), and the definition of getFacetA and getFacetV,
(III.5) IH may be applied on $\mathcal{D}'$ and $\mathcal{E}$
From (III.5) and the IH,
$\sigma_1' \approx_L \sigma_2'$

Case IV: $\mathcal{D}$ or $\mathcal{E}$ ends in FS-CREATEC-S1
Without loss of generality, assume that $\mathcal{D}$ ends in FS-CREATEC-S1. The proof for $\mathcal{E}$ is similar.
By assumption,
(IV.1) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_1, \cdot, id) = a_1$ with
(IV.2) $a_1 \downarrow_{pc_1} \neq \mathsf{NULL}$
(IV.3) $\sigma_1' = \sigma_1$

Subcase i: $pc_2 = \cdot$
The proof for this case is similar to **Subcase I.iii**.

Subcase ii: $pc_2 = L$
From (1), (IV.1), (T.5), and Lemma 49 (Requirement **(EH1)**),
(ii.1) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_2, \cdot, id, a_2^{\mathsf{rt}}) = a_2 \neq \mathsf{NULL}$ or
(ii.2) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_2, \cdot, id, a_2^{\mathsf{rt}}) = a_2 = \langle \mathsf{NULL} | a_2' \rangle$
By assumption and from (ii.1) and (ii.2),
(ii.3) $a_2 \downarrow_{pc_2} \neq \mathsf{NULL}$
From (ii.1)-(ii.3),
(ii.4) $\mathcal{E}$ ends in FS-CREATEC-S1
From (ii.4),
(ii.5) $\sigma_2' = \sigma_2$
From (1), (IV.3), and (ii.5),
$\sigma_1' \approx_L \sigma_2'$

Case V: $\mathcal{D}$ or $\mathcal{E}$ ends in FS-CREATEC-S2
Without loss of generality, assume that $\mathcal{D}$ ends in FS-CREATEC-S2. The proof for $\mathcal{E}$ is similar.
By assumption,
(V.1) $a_{p,1} = \mathsf{NULL}$
(V.2) $\sigma_1' = \sigma_1$

Subcase i: $pc_2 = \cdot$
The proof for this case is similar to **Subcase I.iii**.

Subcase ii: $pc_2 = L$

By assumption and from (V.1) and (T.2),

(ii.1) $a_{p,2} = \mathsf{NULL}$

From (ii.1),

(ii.2) $\mathcal{E}$ ends in FS-CREATEC-S2

From (ii.2),

(ii.3) $\sigma'_2 = \sigma_2$

From (1), (V.2), and (ii.3),

$\sigma'_1 \approx_L \sigma'_2$

Case VI: $\mathcal{D}$ or $\mathcal{E}$ ends in FS-CREATEC-UH

We assume that $pc_1, pc_2 \sqsubseteq L$, so this case holds vacuously.

$\mathcal{D} :: \sigma'_1 = \mathsf{createSibling}_{\mathcal{G}}(\sigma_1, pc_1, id, a_{s,1}, v_1)$
$\mathcal{E} :: \sigma'_2 = \mathsf{createSibling}_{\mathcal{G}}(\sigma_2, pc_2, id, a_{s,2}, v_2)$
Want to show $\sigma'_1 \approx_L \sigma'_2$

$\mathcal{G} = \mathsf{SMS}$
Denote

$\sigma_{L,1} = \mathsf{getStore}(\sigma_1, L)$
$\sigma_{L,2} = \mathsf{getStore}(\sigma_2, L)$

Case I: $\mathcal{D}$ or $\mathcal{E}$ ends in SMS-CREATES

Without loss of generality, assume that $\mathcal{D}$ ends in SMS-CREATES. The proof for $\mathcal{E}$ is similar.

By assumption,

(I.1) $\mathsf{lookupA}_{\mathsf{SMS}}(\sigma_1, pc_1, id, a_1^{\mathsf{rt}}) = \mathsf{NULL}$
(I.2) $a_1 \notin \sigma_{L,1}$
(I.3) $a'_{p,1} = \sigma_{L,1}(a_{s,1}).a_p$
(I.4) $\sigma_{L,1}(a'_{p,1}) = (id_p, v_{p,1}, M_1, a''_{p,1}, (A_1 :: a_{s,1} :: A'_1))$
(I.5) $\sigma'_{L,1} = \sigma_{L,1}[a'_{p,1} \mapsto (id_p, v_{p,1}, M_1, a''_{p,1}, (A_1 ::: a_{s,1} :: a_1 :: A'_1))$
(I.6) $\sigma''_{L,1} = \sigma'_{L,1}[a_1 \mapsto (id, v_1, \cdot, a'_{p,1}, \cdot)$
(I.7) $\sigma'_1 = \mathsf{setStore}(\sigma_1, L, \sigma''_{L,1})$

Subcase i: $pc_2 = L$

By assumption and from (1), (2), (I.1), (T.5), and Lemma 49 (Requirement **(EH1)**) and the definition of $\approx_L$ for SMS nodes,

(i.1) $\mathsf{lookupA}_{\mathsf{SMS}}(\sigma_2, pc_2, id, a_2^{\mathsf{rt}}) = \mathsf{NULL}$

From (T.3) and (I.3),

(i.2) $\sigma_{L,2}(a_{s,2}) \approx_L^{\sigma_2,\sigma_1} \sigma_{L,1}(a_{s,1})$

From (i.2),

(i.3) $\sigma_{L,1}(a_{s,1}).a_p \approx_L^{\sigma_1,\sigma_2} \sigma_{L,2}(a_{s,2}).a_p$

From (1), (i.3), (I.3), and since node id's are unique,

(i.4) $\sigma_{L,1}(a'_{p,2}) = (id_p, v_{p,2}, M_2, a''_{p,2}, (A_2 :: a_{s,2} :: A'_2))$

From (i.1), (i.3), and (i.4),

(i.5) $\mathcal{E}$ ends in SMS-CREATES

From (i.5),

(i.6) $a_2 \notin \sigma_{L,2}$
(i.7) $\sigma'_{L,2} = \sigma_{L,2}[a'_{p,2} \mapsto (id_p, v_{p,2}, M_1, a''_{p,2}, (A_2 :: a_{s,2} :: a_2 :: A'_2))$
(i.8) $\sigma''_{L,2} = \sigma'_{L,2}[a_2 \mapsto (id, v_2, \cdot, a'_{p,2}, \cdot)$
(i.9) $\sigma'_2 = \mathsf{setStore}(\sigma_2, L, \sigma''_{L,2})$

From (1), (I.4), (i.4), (I.5), and (i.7),

(i.10) $\sigma'_{L,1} \approx_L \sigma'_{L,2}$

From (T.1), (i.10), (I.6), (I.5), (i.8), and (i.7),

(i.11) $\sigma''_{L,1} \approx_L \sigma''_{L,2}$

From (I.7), (i.9), (i.11), and the definition of setStore,

$\sigma'_1 \approx_L \sigma'_2$

Subcase ii: $pc_2 = \cdot$

By assumption,
(ii.1) $\mathcal{E}$ ends in SMS-CREATES-NC
From (ii.1),
(ii.2) $\sigma_2'' = \mathsf{createSibling}_{\mathsf{SMS}}(\sigma_2, H, id, \mathsf{getFacetA}(a_{s,2}, H), \mathsf{getFacetV}(v_2, H))$
(ii.3) $\mathcal{E}' :: \sigma_2' = \mathsf{createSibling}_{\mathsf{SMS}}(\sigma_2'', L, id, \mathsf{getFacetA}(a_{s,2}, L), \mathsf{getFacetV}(v_2, L))$
From (ii.2) and Lemma 74.T (Requirement **(EH3)**),
(ii.4) $\sigma_2 \approx_L \sigma_2''$
From (1) and (ii.4),
(ii.5) $\sigma_1 \approx_L \sigma_2''$
From (ii.5), (2), (T.1), (T.3), and the definition of getFacetA and getFacetV,
(ii.6) the IH may be applied on $\mathcal{D}$ and $\mathcal{E}'$
From (ii.6) and the IH,
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{D}$ or $\mathcal{E}$ ends in SMS-CREATES-NC
Without loss of generality, assume that $\mathcal{D}$ ends in SMS-CREATES-NC. The proof for $\mathcal{E}$ is similar.
By assumption,
(II.1) $\sigma_1'' = \mathsf{createSibling}_{\mathsf{SMS}}(\sigma_1, H, id, \mathsf{getFacetA}(a_{s,1}, H), \mathsf{getFacetV}(v_1, H))$
(II.2) $\mathcal{D}' :: \sigma_1' = \mathsf{createSibling}_{\mathsf{SMS}}(\sigma_1'', L, id, \mathsf{getFacetA}(a_{s,1}, L), \mathsf{getFacetV}(v_1, L))$
From (II.1) and Lemma 74.T (Requirement **(EH3)**),
(II.3) $\sigma_1'' \approx_L \sigma_1$
From (1) and (II.3),
(II.4) $\sigma_1'' \approx_L \sigma_2$
From (II.4), (2), (T.1), (T.3), and the definition of getFacetA and getFacetV,
(II.5) the IH may be applied on $\mathcal{D}'$ and $\mathcal{E}$
From (II.5) and the IH,
$\sigma_1' \approx_L \sigma_2'$

Case III: $\mathcal{D}$ or $\mathcal{E}$ ends in SMS-CREATES-S1
Without loss of generality, assume that $\mathcal{D}$ ends in SMS-CREATES-S1. The proof for $\mathcal{E}$ is similar.
By assumption,
(III.1) $\mathsf{lookupA}_{\mathsf{SMS}}(\sigma_1, pc_1, id, a_1^{\mathsf{rt}}) \neq \mathsf{NULL}$ or
(III.2) $a_{s,1} = \mathsf{NULL}$
(III.3) $\sigma_1' = \sigma_1$

Subcase i: $pc_2 = L$
If (III.1) is true, then by assumption and from (1), (T.5), and Lemma 49 (Requirement **(EH1)**) and the definition of $\approx_L$ for SMS nodes,
(i.1) $\mathsf{lookupA}_{\mathsf{SMS}}(\sigma_2, pc_2, id, a_2^{\mathsf{rt}}) \neq \mathsf{NULL}$
If (III.2) is true, then by assumption and from (T.3),
(i.2) $a_{s,2} = \mathsf{NULL}$
By assumption and from (i.1) and (i.2),
(i.3) $\mathcal{E}$ ends in SMS-CREATES-S1
From (i.3),
(i.4) $\sigma_2' = \sigma_2$
From (1), (III.3), and (i.4),
$\sigma_1' \approx_L \sigma_2'$

Subcase ii: $pc_2 = \cdot$
By assumption, $\mathcal{E}$ ends in SMS-CREATES-NC. This case follows from **Case II**

Case IV: $\mathcal{D}$ or $\mathcal{E}$ ends in SMS-CREATES-S2
Without loss of generality, assume that $\mathcal{D}$ ends in SMS-CREATES-S2. The proof for $\mathcal{E}$ is similar.
By assumption,
(IV.1) $\mathsf{lookupA}_{\mathsf{SMS}}(\sigma_1, pc_1, id, a_1^{\mathsf{rt}}) = \mathsf{NULL}$
(IV.2) $\sigma_{L,1}(a_{s,1}).a_p = \mathsf{NULL}$
(IV.3) $\sigma_1' = \sigma_1$

Subcase i: $pc_2 = L$
By assumption and from (1), (IV.1), (T.5), and Lemma 49 (Requirement **(EH1)**) and the definition of $\approx_L$ for SMS nodes,
(i.1) $\mathsf{lookupA}_{\mathsf{SMS}}(\sigma_2, pc_2, id, a_2^{\mathsf{rt}}) = \mathsf{NULL}$
From (IV.2) and (T.3),

(i.2) $\sigma_{L,2}(a_{s,2}).a_p = \mathsf{NULL}$
By assumption and from (i.1) and (i.2),
(i.3) $\mathcal{E}$ ends in SMS-CREATES-S2
From (i.3),
(i.4) $\sigma'_2 = \sigma_2$
From (1), (IV.3), and (i.4),
$\sigma'_1 \approx_L \sigma'_2$

Subcase ii: $pc_2 = \cdot$
By assumption, $\mathcal{E}$ ends in SMS-CREATES-NC. This case follows from **Case II**.

$\mathcal{G} = \mathsf{FS}$
Case I: $\mathcal{D}$ or $\mathcal{E}$ ends in FS-CREATES-NC
Without loss of generality, assume that $\mathcal{D}$ ends in FS-CREATES-NC. The proof for $\mathcal{E}$ is similar.
By assumption,
(I.1) $\sigma''_1 = \mathsf{createSibling}_{\mathsf{FS}}(\sigma_1, H, id, \mathsf{getFacet}(a_{s,1}, H), \mathsf{getFacet}(v_1, H))$
(I.2) $\mathcal{D}' :: \sigma'_1 = \mathsf{createSibling}_{\mathsf{FS}}(\sigma''_1, L, id, \mathsf{getFacet}(a_{s,1}, L), \mathsf{getFacet}(v_1, L))$
From (I.1) and Lemma 74.T (Requirement **(EH3)**),
(I.3) $\sigma''_1 \approx_L \sigma_1$
From (1) and (I.3),
(I.4) $\sigma''_1 \approx_L \sigma_2$
From (I.4), (2), (T.1), (T.3), and the definition of getFacetA and getFacetV,
(I.5) the IH may be applied on $\mathcal{D}'$ and $\mathcal{E}$
From (I.5) and the IH,
$\sigma'_1 \approx_L \sigma'_2$

Case II: $\mathcal{D}$ or $\mathcal{E}$ ends in FS-CREATES
Without loss of generality, assume that $\mathcal{D}$ ends in FS-CREATES. The proof for $\mathcal{E}$ is similar.
By assumption,
(II.1) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_1, \cdot, id, a_1^{\mathsf{rt}}) = \mathsf{NULL}$
(II.2) $a_1 \notin \sigma_1$
(II.3) $\sigma_1(a_{s,1}).v \downarrow_{pc_1} \neq \cdot$
(II.4) $\sigma_1(a_{s,1}).a_p \downarrow_{pc_1} = a'_{p,1}$
(II.5) $\sigma_1(a'_{p,1}) = (id_p, v_{p,1}, M_1, a''_{p,1}, (A_1 :: a'_{s,1} :: A'_1))$ where
(II.6) $a'_{s,1} \downarrow_{pc_1} = a_{s,1}$
(II.7) $v_{p,1} \downarrow_{pc_1} \neq \cdot$
(II.8) $\sigma''_1 = \sigma_1[a_{p,1} \mapsto (id_p, v_{p,1}, M_1, a'_{p,1}, (A_1 :: a'_{s,1} :: \mathsf{createFacet}(a_1, pc_1) :: A'_1))]$
(II.9) $\sigma'_1 = \sigma''_1[a_1 \mapsto (id, \mathsf{createFacet}(v_1, pc_1), \cdot, \mathsf{createFacet}(a_{p,1}, pc_1), \cdot)]$

Subcase i: $pc_2 = \cdot$
The proof for this case is covered by **Case I**.

Subcase ii: $pc_2 = L$
From (1), (II.1), (T.5), and Lemma 49 (Requirement **(EH1)**),
(ii.1) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_2, \cdot, id, a_2^{\mathsf{rt}}) = \mathsf{NULL}$ or
(ii.2) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_2, \cdot, id, a_2^{\mathsf{rt}}) = \langle a_2 | \mathsf{NULL} \rangle$
From (T.3), (II.3), and (II.4),
(ii.3) $\sigma_2(a_{s,2}).v \downarrow_L \neq \cdot$
(ii.4) $\sigma_2(a_{s,2}).a_p \downarrow_L = a'_{p,2}$ with
(ii.5) $\sigma_1(a'_{p,1}).id = \sigma_2(a'_{p,2}).id$
From (1), (ii.5), and since id's are unique,
(ii.6) $\sigma_2(a'_{p,2}) = (id_p, v_{p,2}, M_2, a'_{p,2}, (A_2 :: a'_{s,2} :: A'_2))$ with
(ii.7) $a'_{s,2} \downarrow_L = a_s$ and
(ii.8) $(id_p, v_{p,1}, M_1, a'_{p,1}, (A_1 :: a'_{s,1} :: A'_1)) \approx_L^{\sigma_1, \sigma_2} (id_p, v_{p,2}, M_2, a'_{p,2}, (A_2 :: a'_{s,2} :: A'_2))$
From (II.7) and (ii.8),
(ii.9) $v_{p,2} \downarrow_L \neq \cdot$

Subsubcase a: (ii.1) is true
From (ii.1), (ii.3), (ii.4), (ii.6), and (ii.9),
(a.1) $\mathcal{E}$ ends in FS-CREATES
From (a.1),
(a.2) $a_2 \notin \sigma_2$

(a.3) $\sigma_2'' = \sigma_2[a'_{p,2} \mapsto (id_p, v_{p,2}, M_2, a'_{p,2}, (A_2 :: a'_{s,2} :: \mathsf{createFacet}(a_2, L) :: A'_2))]$
(a.4) $\sigma_2' = \sigma_2''[a_2 \mapsto (id, \mathsf{createFacet}(v_2, L), \cdot, \mathsf{createFacet}(a'_{p,2}, L), \cdot)]$
From (T.1), (II.8), (a.3), and the definition of createFacet,
(a.5) $(id, \mathsf{createFacet}(v_1, pc_1), \cdot, \mathsf{createFacet}(a_{p,1}, pc_1), \cdot) \approx_L^{\sigma_1', \sigma_2'}$
$(id, \mathsf{createFacet}(v_2, L), \cdot, \mathsf{createFacet}(a'_{p,2}, L), \cdot)$
From (ii.8), (a.5), (II.9), (a.4), and the definition of createFacet,
(a.6) $(id_p, v_{p,1}, M_1, a'_{p,1}, (A_1 :: a'_{s,1} :: \mathsf{createFacet}(a_1, pc_1) :: A'_1)) \approx_L^{\sigma_1', \sigma_2'}$
$(id_p, v_{p,2}, M_2, a'_{p,2}, (A_2 :: a'_{s,2} :: \mathsf{createFacet}(a_2, L) :: A'_2))$
From (1), (II.8), (a.3), (a.5), and (a.6),
(a.7) $\sigma_1'' \approx_L \sigma_2''$
From (II.9) and (a.4)-(a.7),
$\sigma_1' \approx_L \sigma_2'$

Subsubcase b: (ii.2) is true
From (ii.2)-(ii.4), (ii.6), and (ii.9),
(b.1) $\mathcal{E}$ ends in FS-CREATES-UL
From (b.1),
(b.2) $\sigma_2(a_2) = (id, v_2', M_2', a_{p,2}'', A_2'')$
(b.3) $\sigma_2'' = \sigma_2[a'_{p,2} \mapsto (id_p, v_{p,2}, M_2, a'_{p,2}, (A_2 :: a'_{s,2} :: \mathsf{createFacet}(a_2, L) :: A'_2))]$
(b.4) $\sigma_2' = \sigma_2''[a_2 \mapsto (id, \mathsf{updateFacet}(v_2', v_2, L), \cdot, \mathsf{updateFacet}(a''_{p,2}, a'_{p,2}, L), \cdot)]$
From (T.1), (II.8), (b.3), and the definitions of createFacet and updateFacet,
(b.5) $(id, \mathsf{createFacet}(v_1, pc_1), \cdot, \mathsf{createFacet}(a_{p,1}, pc_1), \cdot) \approx_L^{\sigma_1', \sigma_2'}$
$(id, \mathsf{updateFacet}(v_2', v_2, L), M_2', \mathsf{updateFacet}(a''_{p,2}, a'_{p,2}, L), A_2'')$
From (ii.8), (b.5), (II.9), (b.4), and the definition of createFacet,
(b.6) $(id_p, v_{p,1}, M_1, a'_{p,1}, (A_1 :: a'_{s,1} :: \mathsf{createFacet}(a_1, pc_1) :: A'_1)) \approx_L^{\sigma_1', \sigma_2'}$
$(id_p, v_{p,2}, M_2, a'_{p,2}, (A_2 :: a'_{s,2} :: \mathsf{createFacet}(a_2, L) :: A'_2))$
From (1), (II.8), (b.3), (b.5), and (b.6),
(b.7) $\sigma_1'' \approx_L \sigma_2''$
From (II.9) and (b.4)-(b.7),
$\sigma_1' \approx_L \sigma_2'$

Case III: $\mathcal{D}$ or $\mathcal{E}$ ends in FS-CREATES-UL
Without loss of generality, assume that $\mathcal{D}$ ends in FS-CREATES-UL. The proof for $\mathcal{E}$ is similar.
By assumption,
(III.1) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_1, \cdot, id, a_1^{\mathsf{rt}}) = \langle a_1 | \mathsf{NULL} \rangle$
(III.2) $\sigma_1(a_1) = (id, v_1'', M_1', a_{p,1}'', A_1')$
(III.3) $\sigma_1(a_{s,1}).v \downarrow_{pc_1} \neq \cdot$
(III.4) $\sigma_1(a_{s,1}).a_p \downarrow_{pc_1} = a'_{p,1}$
(III.5) $\sigma_1(a'_{p,1}) = (id_p, v_{p,1}, M_1, a''_{p,1}, (A_1 :: a'_{s,1} :: A'_1))$ where
(III.6) $a'_{s,1} \downarrow_{pc_1} = a_{s,1}$
(III.7) $v_{p,1} \downarrow_{pc_1} \neq \cdot$
(III.8) $\sigma_1'' = \sigma_1[a_{p,1} \mapsto (id_p, v_{p,1}, M_1, a'_{p,1}, (A_1 :: a'_{s,1} :: \mathsf{createFacet}(a_1, pc_1) :: A'_1))]$
(III.9) $\sigma_1' = \sigma_1''[a_1 \mapsto (id, \mathsf{updateFacet}(v_1', v_1, pc_1), M_1', \mathsf{updateFacet}(a''_{p,1}, a_{p,1}, pc_1), A_1')]$

Subcase i: $pc_2 = \cdot$
The proof for this case is covered by **Case I**.

Subcase ii: $pc_2 = L$
From (1), (II.1), (T.5), and Lemma 49 (Requirement **(EH1)**),
(ii.1) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_2, \cdot, id, a_2^{\mathsf{rt}}) = \mathsf{NULL}$ or
(ii.2) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_2, \cdot, id, a_2^{\mathsf{rt}}) = \langle a_2 | \mathsf{NULL} \rangle$
The proof for when (ii.1) is true is covered by **Case II**, so we consider the case where (ii.2) is true
From (T.3), (II.3), and (II.4),
(ii.3) $\sigma_2(a_{s,2}).v \downarrow_L \neq \cdot$
(ii.4) $\sigma_2(a_{s,2}).a_p \downarrow_L = a'_{p,2}$ with
(ii.5) $\sigma_1(a'_{p,1}).id = \sigma_2(a'_{p,2}).id$
From (1), (ii.5), and since id's are unique,
(ii.6) $\sigma_2(a'_{p,2}) = (id_p, v_{p,2}, M_2, a'_{p,2}, (A_2 :: a'_{s,2} :: A'_2))$ with
(ii.7) $a'_{s,2} \downarrow_L = a_s$ and
(ii.8) $(id_p, v_{p,1}, M_1, a'_{p,1}, (A_1 :: a'_{s,1} :: A'_1)) \approx_L^{\sigma_1, \sigma_2} (id_p, v_{p,2}, M_2, a'_{p,2}, (A_2 :: a'_{s,2} :: A'_2))$

From (II.7) and (ii.8),
(ii.9) $v_{p,2} \downarrow_L \neq \cdot$
From (ii.2)-(ii.4), (ii.6), and (ii.9),
(ii.10) $\mathcal{E}$ ends in FS-CREATES-UL
From (ii.10),
(ii.11) $\sigma_2(a_2) = (id, v_2', M_2', a_{p,2}'', A_2'')$
(ii.12) $\sigma_2'' = \sigma_2[a_{p,2}' \mapsto (id_p, v_{p,2}, M_2, a_{p,2}', (A_2 :: a_{s,2}' :: \mathsf{createFacet}(a_2, L) :: A_2'))]$
(ii.13) $\sigma_2' = \sigma_2''[a_2 \mapsto (id, \mathsf{updateFacet}(v_2', v_2, L), \cdot, \mathsf{updateFacet}(a_{p,2}'', a_{p,2}', L), \cdot)]$
From (T.1), (II.8), (ii.12), and the definitions of updateFacet,
(ii.14) $(id, \mathsf{updateFacet}(v_1', v_1, pc_1), M_1', \mathsf{updateFacet}(a_{p,1}'', a_{p,1}, pc_1), A_1') \approx_L^{\sigma_1', \sigma_2'}$
$(id, \mathsf{updateFacet}(v_2', v_2, L), M_2', \mathsf{updateFacet}(a_{p,2}'', a_{p,2}', L), A_2'')$
From (ii.8), (ii.14), (II.9), (ii.13), and the definition of createFacet,
(ii.15) $(id_p, v_{p,1}, M_1, a_{p,1}', (A_1 :: a_{s,1}' :: \mathsf{createFacet}(a_1, pc_1) :: A_1')) \approx_L^{\sigma_1', \sigma_2'}$
$(id_p, v_{p,2}, M_2, a_{p,2}', (A_2 :: a_{s,2}' :: \mathsf{createFacet}(a_2, L) :: A_2'))$
From (1), (II.8), (ii.12), (ii.14), and (ii.15),
(ii.16) $\sigma_1'' \approx_L \sigma_2''$
From (II.9) and (ii.13)-(ii.16),
$\sigma_1' \approx_L \sigma_2'$

Case IV: $\mathcal{D}$ or $\mathcal{E}$ ends in FS-CREATES-S1
Without loss of generality, assume that $\mathcal{D}$ ends in FS-CREATES-S1. The proof for $\mathcal{E}$ is similar.
By assumption,
(IV.1) $\mathsf{lookup}_{\mathsf{FS}}(\sigma_1, \cdot, id, a_1^{\mathsf{rt}}) = a_1$
(IV.2) $a_1 \downarrow_{pc_1} \neq \mathsf{NULL}$
(IV.3) $\sigma_1' = \sigma_1$

Subcase i: $pc_2 = \cdot$
The proof for this case is covered by **Case I**.

Subcase ii: $pc_2 = L$
From (1), (IV.1), (IV.2), (T.5), and Lemma 49 (Requirement **(EH1)**),
(ii.1) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_2, \cdot, id, a_2^{\mathsf{rt}}) = a_2$ or
(ii.2) $\mathsf{lookupA}_{\mathsf{FS}}(\sigma_2, \cdot, id, a_2^{\mathsf{rt}}) = \langle \_ | a_2 \rangle$ with
(ii.3) $a_2 \neq \mathsf{NULL}$
From (ii.1)-(ii.3),
(ii.4) $\mathcal{E}$ ends in FS-CREATES-S1
From (ii.4),
(ii.5) $\sigma_2' = \sigma_2$
From (1), (IV.3), and (ii.5),
$\sigma_1' \approx_L \sigma_2'$

Case V: $\mathcal{D}$ or $\mathcal{E}$ ends in FS-CREATES-S2
Without loss of generality, assume that $\mathcal{D}$ ends in FS-CREATES-S2. The proof for $\mathcal{E}$ is similar.
By assumption,
(V.1) $a_{s,1} = \mathsf{NULL}$ or
(V.2) $\sigma_1(a_{s,1}).a_p \downarrow_{pc_1} = \mathsf{NULL}$
(V.3) $\sigma_1' = \sigma_1$

Subcase i: $pc_2 = \cdot$
The proof for this case is covered by **Case I**.

Subcase ii: $pc_2 = L$
From (T.3), (V.1), and (V.2), either
(ii.1) $a_{s,2} = \mathsf{NULL}$ or
(ii.2) $\sigma_2(a_{s,2}).a_p \downarrow_L = \mathsf{NULL}$
From (ii.1) and (ii.2),
(ii.3) $\mathcal{E}$ ends in FS-CREATES-S2
From (ii.3),
(ii.4) $\sigma_2' = \sigma_2$
From (1), (V.3), and (ii.4),
$\sigma_1' \approx_L \sigma_2'$

Case VI: $\mathcal{D}$ or $\mathcal{E}$ ends in FS-CREATES-S3

Without loss of generality, assume that $\mathcal{D}$ ends in FS-CREATES-S3. The proof for $\mathcal{E}$ is similar.

By assumption,

(V.1) $\sigma_1(a_{s,1}).v \downarrow_{pc_1} = \cdot$ or

(V.2) $\sigma_1(a_{s,1}).a_p \downarrow_{pc_1} = \cdot$ or

(V.3) $\sigma_1(\sigma_1(a_{s,1}).a_p \downarrow_{pc_1}).v \downarrow_{pc_1} = \cdot$

(V.4) $\sigma_1' = \sigma_1$

Subcase i: $pc_2 = \cdot$

The proof for this case is covered by **Case I**.

Subcase ii: $pc_2 = L$

From (T.3) and (V.1)-(V.3), either

(ii.1) $\sigma_2(a_{s,2}).v \downarrow_L = \cdot$ or

(ii.2) $\sigma_2(a_{s,2}).a_p \downarrow_L = \cdot$ or

(ii.3) $\sigma_2(\sigma_2(a_{s,2}).a_p \downarrow_L).v \downarrow_L = \cdot$

From (ii.1)-(ii.3),

(ii.4) $\mathcal{E}$ ends in FS-CREATES-S3

From (ii.4),

(ii.5) $\sigma_2' = \sigma_2$

From (1), (V.4), and (ii.5),

$\sigma_1' \approx_L \sigma_2'$

Case VII: $\mathcal{D}$ or $\mathcal{E}$ ends in FS-CREATES-UH

We assume that $pc_1, pc_2 \sqsubseteq L$, so this case holds vacuously.

$\mathcal{D} :: \sigma_1' = \mathsf{registerEH}_{\mathcal{G}}(\sigma_1, pc_1, a_1, eh)$

$\mathcal{E} :: \sigma_2' = \mathsf{registerEH}_{\mathcal{G}}(\sigma_2, pc_2, a_2, eh)$

Want to show that $\sigma_1' \approx_L \sigma_2'$

$\mathcal{G} = \mathsf{SMS}$

Denote

$\sigma_{L,1} = \mathsf{getStore}(\sigma_1, L)$

$\sigma_{L,2} = \mathsf{getStore}(\sigma_2, L)$

Case I: $\mathcal{D}$ or $\mathcal{E}$ ends in SMS-REGISTEREH-NC

Without loss of generality, assume that $\mathcal{D}$ ends in SMS-REGISTEREH-NC. The proof for $\mathcal{E}$ is similar.

By assumption,

(I.1) $\sigma_1'' = \mathsf{registerEH}_{\mathsf{SMS}}(\sigma_1, H, \mathsf{getFacet}(a_1, H), eh)$

(I.2) $\mathcal{D}' :: \sigma_1' = \mathsf{registerEH}_{\mathsf{SMS}}(\sigma_1'', L, \mathsf{getFacet}(a_1, L), eh)$

From (I.1) and Lemma 74.T (Requirement **(EH3)**),

(I.3) $\sigma_1'' \approx_L \sigma_1$

From (1) and (I.3),

(I.4) $\sigma_1'' \approx_L \sigma_2$

From (I.4), (2), (T.4), and the definition of getFacet,

(I.5) the IH may be applied on $\mathcal{D}'$ and $\mathcal{E}$

From (I.5) and the IH,

$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{D}$ or $\mathcal{E}$ ends in SMS-REGISTEREH

Without loss of generality, assume that $\mathcal{D}$ ends in SMS-REGISTEREH. The proof for $\mathcal{E}$ is similar.

By assumption,

(II.1) $\sigma_{L,1}(a_1) = (id, v_1, M_1, a_{p,1}, A_1)$

(II.2) $M_1' = M[Ev \mapsto M(Ev) \cup \{(eh, pc_1)\}]$ for

(II.3) $eh = \mathsf{on}Ev(x)\{c\}$

(II.4) $\sigma_{L,1}' = \sigma_{L,1}[a_1 \mapsto (id, v_1, M_1', a_{p,1}, A_1)$

(II.5) $\sigma_1' = \mathsf{setStore}(\sigma_1, L, \sigma_{L,1}'$

Subcase i: $pc_2 = \cdot$

This case is covered by **Case I**.

Subcase ii: $pc_2 = L$
From (T.4) and (II.1),
(ii.1) $\sigma_{L,2}(a_2) = (id, v_2, M_2, a_{p,2}, A_2)$ with
(ii.2) $v_1 \approx_L^{\sigma_1,\sigma_2} v_2$
(ii.3) $M_1 \approx_L M_2$
(ii.4) $A_2 \approx_L^{\sigma_1,\sigma_2} A_2$
From (ii.1),
(ii.5) $\mathcal{E}$ ends in SMS-REGISTEREH
From (ii.5),
(ii.6) $M_2' = M[Ev \mapsto M(Ev) \cup \{(eh, L)\}]$
(ii.7) $\sigma_{L,2}' = \sigma_{L,2}[a_2 \mapsto (id, v_2, M_2', a_{p,2}, A_2)$
(ii.8) $\sigma_2' = \mathsf{setStore}(\sigma_2, L, \sigma_{L,2}'$
From (II.2), (II.4), (II.5), (ii.2)-(ii.4), (ii.6)-(ii.8), and the definition of setStore,
$\sigma_1' \approx_L \sigma_2'$

Case III: $\mathcal{D}$ or $\mathcal{E}$ ends in SMS-REGISTEREH-S
Without loss of generality, assume that $\mathcal{D}$ ends in SMS-REGISTEREH-S. The proof for $\mathcal{E}$ is similar.
By assumption,
(III.1) $a_1 = \mathsf{NULL}$
(III.2) $\sigma_1' = \sigma_1$

Subcase i: $pc_2 = \cdot$
This case is covered by **Case I**.

Subcase ii: $pc_2 = L$
From (T.4) and (III.1),
(ii.1) $a_2 = \mathsf{NULL}$
From (ii.1),
(ii.2) $\mathcal{E}$ ends in SMS-REGISTEREH-S
From (ii.2),
(ii.3) $\sigma_2' = \sigma_2$
From (1), (III.2), and (ii.3),
$\sigma_1' \approx_L \sigma_2'$

$\mathcal{G} = \mathsf{FS}$
Case I: $\mathcal{D}$ or $\mathcal{E}$ ends in FS-REGISTEREH-NC
Without loss of generality, assume that $\mathcal{D}$ ends in FS-REGISTEREH-NC. The proof for $\mathcal{E}$ is similar.
By assumption,
(I.1) $\sigma_1'' = \mathsf{registerEH}_{\mathsf{FS}}(\sigma_1, H, \mathsf{getFacet}(a_1, H), eh)$
(I.2) $\mathcal{D}' :: \sigma_1' = \mathsf{registerEH}_{\mathsf{FS}}(\sigma_1'', L, \mathsf{getFacet}(a_1, L), eh)$
From (I.1) and Lemma 74.T (Requirement **(EH3)**),
(I.3) $\sigma_1'' \approx_L \sigma_1$
From (1) and (I.3),
(I.4) $\sigma_1'' \approx_L \sigma_2$
From (I.4), (2), (T.4), and the definition of getFacet,
(I.5) the IH may be applied on $\mathcal{D}'$ and $\mathcal{E}$
From (I.5) and the IH,
$\sigma_1' \approx_L \sigma_2'$

Case II: $\mathcal{D}$ or $\mathcal{E}$ ends in FS-REGISTEREH
Without loss of generality, assume that $\mathcal{D}$ ends in FS-REGISTEREH. The proof for $\mathcal{E}$ is similar.
By assumption,
(II.1) $\sigma_1(a_1) = (id, v_1, M_1, a_{p,1}, A_1)$
(II.2) $v_1 \downarrow_{pc_1} \neq \cdot$
(II.3) $M_1' = M[Ev \mapsto M(Ev) \cup \{(eh, pc_1)\}]$ for
(II.4) $eh = \mathsf{on}Ev(x)\{c\}$
(II.5) $\sigma_1' = \sigma_1[a_1 \mapsto (id, v_1, M_1', a_{p,1}, A_1)$

Subcase i: $pc_2 = \cdot$
This case is covered by **Case I**.

Subcase ii: $pc_2 = L$
From (T.4) and (II.1),
(ii.1) $\sigma_2(a_2) = (id, v_2, M_2, a_{p,2}, A_2)$ with
(ii.2) $v_2 \downarrow_{pc_2} \neq \cdot$
(ii.3) $v_1 \approx_L^{\sigma_1,\sigma_2} v_2$
(ii.4) $M_1 \approx_L M_2$
(ii.5) $A_2 \approx_L^{\sigma_1,\sigma_2} A_2$
From (ii.1) and (ii.2),
(ii.6) $\mathcal{E}$ ends in FS-REGISTEREH
From (ii.6),
(ii.7) $M_2' = M[Ev \mapsto M(Ev) \cup \{(eh, L)\}]$
(ii.8) $\sigma_2' = \sigma_2[a_2 \mapsto (id, v_2, M_2', a_{p,2}, A_2)$
From (II.2), (II.5), (ii.3)-(ii.5), (ii.7), (ii.8), and the definition of setStore,
$\sigma_1' \approx_L \sigma_2'$

Case III: $\mathcal{D}$ or $\mathcal{E}$ ends in FS-REGISTEREH-S
Without loss of generality, assume that $\mathcal{D}$ ends in FS-REGISTEREH-S. The proof for $\mathcal{E}$ is similar.
By assumption,
(III.1) $a_1 = \mathsf{NULL}$ or
(III.2) $\sigma_1(a_1).v \downarrow_{pc_1} = \cdot$
(III.3) $\sigma_1' = \sigma_1$

Subcase i: $pc_2 = \cdot$
This case is covered by **Case I**.

Subcase ii: $pc_2 = L$
From (T.4), (III.1), and (III.2), either
(ii.1) $a_2 = \mathsf{NULL}$ or
(ii.2) $\sigma_2(a_2).v \downarrow_L = \cdot$
From (ii.1) and (ii.2),
(ii.3) $\mathcal{E}$ ends in FS-REGISTEREH-S
From (ii.3),
(ii.4) $\sigma_2' = \sigma_2$
From (1), (III.3), and (ii.4),
$\sigma_1' \approx_L \sigma_2'$

□

Requirement (WEH4) L updates are equivalent (Weak Secrecy)

Lemma 82. *If* $\sigma_1 \approx_L \sigma_2$, $pc_1, pc_2 \sqsubseteq L$, $v_1 \approx_L v_2$, *and* $\alpha_1 = \alpha_2$, *then for* $\mathsf{assign}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1) = (\sigma_1', \alpha_1)$ *and* $\mathsf{assign}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2) = (\sigma_2', \alpha_2)$, $\sigma_1' \approx_L \sigma_2'$

Proof.
Only the cases for $\mathcal{G} = \mathsf{TS}$ are considered. The other cases are not considered since they follow from Lemma 80
We examine each case of $\mathcal{D} :: \mathsf{assign}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1) = (\sigma_1', \alpha_1)$
Denote $\mathcal{E} :: \mathsf{assign}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2) = (\sigma_2', \alpha_2)$
By assumption,
(1) $\sigma_1 \approx_L \sigma_2$
(2) $pc_1, pc_2 \sqsubseteq L$
(3) $v_1 \approx_L v_2$
From $\mathcal{G} = \mathsf{TS}$ and (3),
(4) $v_1 = (v_1', l_1)$ and $v_2 = (v_2', l_2)$ with
(5) $(v_1', l_1) \approx_L (v_2', l_2)$

Case I: $\mathcal{D}$ ends in TS-ASSIGNEH
By assumption,
(I.1) $\sigma_1(id) = (id, (v_1'', l_1''), M_1, l_1')$
(I.2) $\sigma_1' = \sigma_1[id \mapsto (id, (v_1', l_1 \sqcup pc_1 \sqcup l_1'), M_1, l_1')]$
(I.3) $\alpha_1 = \bullet$
(I.4) $l_1 \sqcup pc_1 \sqcup l_1' \sqsubseteq l_1''$
From (I.4) and (2) our security lattice, and since TS stores do not contain $\cdot$,
(I.5) $l_1 \sqsubseteq l_1''$

Subcase i: $l_1' \sqsubseteq L$

By assumption and from (1),

(i.1) $\sigma_2(id) = (id, (v_2'', l_2''), M_2, l_2')$ with

(i.2) $l_2' \sqsubseteq L$

(i.3) $(v_1'', l_1'') \approx_L (v_2'', l_2'')$

(i.4) $M_1 \approx_L M_2$

From (i.1),

(i.5) $\mathcal{E}$ ends in TS-ASSIGNEH

From (i.5), (I.5), (4), (5), and (i.2),

(i.6) $l_2 \sqsubseteq l_2''$ (because otherwise, $l_2 = H$ and $l_2'' = L$, so either (i.1) is not true, or (i.4) or (5) are not true)

From (2), (i.3), (i.7), and our security lattice,

(i.7) $l_2 \sqcup pc_2 \sqsubseteq l_2''$

From (i.8) and (i.6),

(i.8) $\mathcal{E}$ ends in TS-ASSIGNEH

From (i.8),

(i.9) $\sigma_2' = \sigma_2[id \mapsto (id, (v_2', l_2 \sqcup pc_2 \sqcup l_2'), M_2)]$

(i.10) $\alpha_2 = \bullet$

From (1), (2), (5), (i.4), (I.2), and (i.9),

$\sigma_1' \approx_L \sigma_2'$

From (I.3) and (i.10),

$\alpha_1 = \alpha_2$

Subcase ii: $l_1' \not\sqsubseteq L$

By assumption and from (I.2), (I.5) and our security lattice,

(ii.1) $l_1'' = H$

(ii.2) $\sigma_1' \approx_L \sigma_2'$

If $id \notin \sigma_2$, then from TS-ASSIGNEH-S,

(ii.3) $\sigma_2' = \sigma_2$

(ii.4) $\alpha_2 = \bullet$

Then from (1), (ii.2), and (ii.3),

$\sigma_1' \approx_L \sigma_2'$

And from (I.3) and (ii.4),

$\alpha_1 = \alpha_2$

Otherwise, $id \in \sigma_2$, and by assumption and from (1),

(ii.5) $\sigma_2(id) = (id, (v_2'', l_2''), M_2, l_2')$ with

(ii.6) $l_2' \not\sqsubseteq L$

From (ii.5) and (ii.6),

(ii.7) $\mathcal{E}$ ends in TS-ASSIGNEH

From (ii.7),

(ii.8) $\sigma_2' = \sigma_2[id \mapsto (id, (v_2', l_2 \sqcup pc_2 \sqcup l_2'), M_2, l_2')]$

(ii.9) $\alpha_2 = \bullet$

By assumption and from (ii.6), (I.2), and (ii.8),

$\sigma_1' \approx_L \sigma_2'$

From (I.3) and (ii.9),

$\alpha_1 = \alpha_2$

Case II: $\mathcal{D}$ ends in TS-ASSIGNEH-S

By assumption,

(II.1) $id \notin \sigma_1$

(II.2) $\sigma_1' = \sigma_1$

(II.3) $\alpha_1 = \bullet$

From (1) and (II.1), either

(II.4) $id \notin \sigma_2$ or

(II.5) $\mathsf{labOf}(\sigma_2(id), \_) = H$

If (II.4) is true, then $\mathcal{E}$ ends in TS-ASSIGNEH-S and

(II.6) $\sigma_2' = \sigma_2$

(II.7) $\alpha_2 = \bullet$

From (1), (II.2), and (II.6),

$\sigma_1' \approx_L \sigma_2'$

From (II.3) and (II.7),

$\alpha_1 = \alpha_2$

Otherwise, (II.5) is true

From (II.5), $\mathcal{E}$ ends in TS-ASSIGNEH and

(II.8) $\sigma_2' = \sigma_2[id \mapsto (id, (v, l_2 \sqcup pc_2 \sqcup H), \sigma_2(id).M, H)]$
(II.9) $\alpha_2 = \bullet$
From (II.5) and (II.8),
(II.10) $\sigma_2 \approx_L \sigma_2'$
From (1), (II.2), and (II.10),
$\sigma_1' \approx_L \sigma_2'$
From (II.3) and (II.9),
$\alpha_1 = \alpha_2$

Case III: $\mathcal{D}$ ends in TS-ASSIGNEH-GW
By assumption,
(III.1) $\sigma_1(id) = (id, (v_1'', l_1''), M_1, l_1')$ with
(III.2) $l_1' \sqsubseteq L$ and
(III.3) $l_1 \sqcup pc_1 \not\sqsubseteq l_1''$
(III.4) $\sigma_1' = \sigma_1[id \mapsto (id, (v_1', l_1 \sqcup pc_1 \sqcup l_1'), M_1, l_1')]$
(III.5) $\alpha_1 = \mathsf{gw}(id)$
From (1) and (III.1),
(III.6) $\sigma_2(id) = (id, (v_2'', l_2''), M_2, l_2')$ with
(III.7) $l_2' \sqsubseteq L$ and
(III.8) $M_1 \approx_L M_2$
(III.9) $(v_1'', l_1'') \approx_L (v_2'', l_2'')$
From (III.3), (2), our security lattice, and since TS never uses $\cdot$ as a label,
(III.10) $l_1 = H$ and $l_1'' = L$
From (5) and (III.10),
(III.11) $l_2 = H$
From (III.9) and (III.10),
(III.12) $l_2'' = L$
From (2), (III.6), (III.7), (III.11), (III.12), and our security lattice,
(III.13) $\mathcal{E}$ must end in TS-ASSIGNEH-GW
From (III.13),
(III.14) $\sigma_2' = \sigma_2[id \mapsto (id, (v_2', l_2 \sqcup pc_2 \sqcup l_2'), M_2, l_2')]$
(III.15) $\alpha_2 = \mathsf{gw}(id)$
From (III.10), (III.11), and our security lattice,
(III.16) $(v_1', l_1 \sqcup pc_1 \sqcup l_1') \approx_L (v_2', l_2 \sqcup pc_2 \sqcup l_2')$
From (III.4), (III.14), (III.16), (III.8), (III.2), and (III.7),
$\sigma_1' \approx_L \sigma_2'$
From (III.5) and (III.15),
$\alpha_1 = \alpha_2$

□

Lemma 83. *If $\sigma_1 \approx_L \sigma$, and $pc_1, pc_2 \sqsubseteq L$ and $v_1 \approx_L v_2$, then for $(\sigma_1', \alpha_1) = \mathsf{createElem}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1)$ and $(\sigma_2', \alpha_2) = \mathsf{createElem}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2)$, $\sigma_1' \approx_L \sigma_2'$ and $\alpha_1 = \alpha_2$*

Proof.
Only the cases for $\mathcal{G} = \mathsf{TS}$ are considered. The other cases are not considered since they follow from Lemma 81
By induction on the structure of $\mathcal{D} :: \mathsf{createElem}_{\mathcal{G}}(\sigma_1, pc_1, id, v_1) = (\sigma_1', \alpha_1)$ and
$\mathcal{E} :: \mathsf{createElem}_{\mathcal{G}}(\sigma_2, pc_2, id, v_2) = (\sigma_2', \alpha_2)$
By assumption,
(1) $\sigma_1 \approx_L \sigma_2$
(2) $pc_1, pc_2 \sqsubseteq L$
(3) $v_1 \approx_L v_2$
From $\mathcal{G} = \mathsf{TS}$ and (3),
(4) $v_1 = (v_1', l_1)$ and $v_2 = (v_2', l_2)$ with
(5) $(v_1', l_1) \approx_L (v_2', l_2)$

Case I: $\mathcal{D}$ ends in TS-CREATE
By assumption,
(I.1) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_1, L, id) = (\mathsf{NULL}, \_)$
(I.2) $\sigma_1' = \sigma_1[id \mapsto (id, (v_1', l_1), \cdot, L)]$
(I.3) $\alpha_1 = \bullet$
From (1), (2), (I.1) and Lemma 48 (Requirement **(EH1)**),
(I.4) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, L, id) \approx_L (\mathsf{NULL}, \_)$
From (I.4), either
(I.5) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, L, id) = (\mathsf{NULL}, \_)$ or

(I.6) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, L, id) = (id, (v_2'', l_2''), M, H)$

Subcase i: (I.5) is true
From (I.5),
(i.1) $\mathcal{E}$ must end in TS-CREATE
From (i.1),
(i.2) $\sigma_2' = \sigma_2[id \mapsto (id, (v_2', l_2), \cdot, L)]$
(i.3) $\alpha_2 = \bullet$
From (1), (5), (I.2), and (i.2),
$\sigma_1' \approx_L \sigma_2'$
From (I.3) and (i.3),
$\alpha_1 = \alpha_2$

Subcase ii: (I.6) is true
From (I.6) and (2),
(ii.1) $\mathcal{E}$ ends in TS-CREATE-U2
From (ii.1) and (I.6),
(ii.2) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, L, id) = (id, (v_2'', l_2''), M_2, H)$
(ii.3) $\sigma_2' = \sigma_2[id \mapsto (id, (v_2', l_2), M_2, L)$
(ii.4) $\alpha_2 = \bullet$
From Lemma 84,
(ii.5) $M_2 \approx_L \cdot$
From (1), (I.2), (ii.3), (5), and (ii.5),
$\sigma_1' \approx_L \sigma_2'$
From (I.3) and (ii.4),
$\alpha_1 = \alpha_2$

Subcase iii: $pc_2 = \cdot$
The proof for this case follows from applying the IH on $\mathcal{D}$ and the premise of $\mathcal{E}$.

Case II: $\mathcal{E}$ ends in TS-CREATE
The proof is similar to **Case I**.

Case III: $\mathcal{D}$ ends in TS-CREATE-U1
By assumption,
(III.1) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_1, L, id) = (id, (v_1'', l_1''), M_1, l_1')$
(III.2) $l_1' \sqsubseteq L$
(III.3) $l_1 \sqcup L \sqsubseteq l_1''$ or $l_1' \not\sqsubseteq L$
(IIII.4) $\sigma_1' = \sigma_1[id \mapsto (id, (v_1', l_1 \sqcup L \sqcup l_1'), M_1, l_1')]$
(III.5) $\alpha_1 = \bullet$
From (III.2) and since TS never contains $\cdot$,
(III.6) $l_1' = L$
From (III.6), (III.3), and from our security lattice,
(III.7) $l_1 \sqsubseteq l_1''$
From (1), (III.1), (III.2), and Lemma 48 (Requirement **(EH1)**),
(III.8) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, L, id) = (id, (v_2'', l_2''), M_2, l_2')$ with $l_2' \sqsubseteq L$, $(v_1'', l_1'') \approx_L (v_2'', l_2'')$ and $M_1 \approx_L M_2$

Subcase i: $pc_2 = L$
By assumption and from (III.8),
(i.1) $\mathcal{E}$ ends in TS-CREATE-U1 or TS-CREATE-U1-GW
From (5) and (III.8),
(i.2) $l_1'' = l_2''$ and $l_1 = l_2$
From (i.2) and (III.7),
(i.3) $l_2 \sqsubseteq l_2''$
From (i.1) and (i.3),
(i.4) $\mathcal{E}$ ends in TS-CREATE-U1
From (i.4),
(i.5) $\sigma_2' = \sigma_2[id \mapsto (id, (v_2', l_2 \sqcup L \sqcup l_2'), M_2, l_2')]$
(i.6) $\alpha_2 = \bullet$
From (1), (III.4), (i.5), (5), (III.2), and (III.8),
$\sigma_1' \approx_L \sigma_2'$
From (III.5) and (i.6),
$\alpha_1 = \alpha_2$

Subcase ii: $pc_2 = \cdot$
The proof for this case follows from applying the IH on $\mathcal{D}$ and the premise of $\mathcal{E}$.

Case IV: $\mathcal{E}$ ends in TS-CREATE-U1
The proof is similar to **Case III**.

Case V: $\mathcal{D}$ ends in TS-CREATE-U1-GW
By assumption,
(V.1) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_1, L, id) = (id, (v_1'', l_1''), M_1, l_1')$
(V.2) $l_1' \sqsubseteq L$
(V.3) $l_1 \sqcup L \not\sqsubseteq l_1''$
(V.4) $\sigma_1' = \sigma_1[id \mapsto (id, (v_1', l_1 \sqcup L \sqcup l_1'), M_1, l_1')]$
(V.5) $\alpha_1 = \mathsf{gw}(id)$
From (1), (V.1), (V.2), and Lemma 48 (Requirement **(EH1)**),
(V.6) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, L, id) = (id, (v_2'', l_2''), M_2, l_2')$ with $l_2' \sqsubseteq L$ $(v_1'', l_1'') \approx_L (v_2'', l_2'')$ and $M_1 \approx_L M_2$
From (V.3) and our security lattice and since TS never contains $\cdot$,
(V.7) $l_1 = H$ and $l_1'' = L$

Subcase i: $pc_2 = L$
By assumption and from (V.6),
(i.1) $\mathcal{E}$ ends in TS-CREATE-U1 or TS-CREATE-U1-GW
From (V.7), (5), and (V.6),
(i.2) $l_2 = H$ and $l_2'' = L$
From (i.1) and (i.2),
(i.3) $\mathcal{E}$ ends in TS-CREATE-U1-GW
From (i.3),
(i.4) $\sigma_2' = \sigma_2[id \mapsto (id, (v_2', l_2 \sqcup L \sqcup l_2'), M_2, l_2')]$
(i.5) $\alpha_2 = \mathsf{gw}(id)$
From (1), (V.4), (i.4), (5), (V.6), and our security lattice,
(i.6) $\sigma_1' \approx_L \sigma_2'$
From (V.5) and (i.5),
$\alpha_1 = \alpha_2$

Subcase ii: $pc_2 = \cdot$
The proof for this case follows from applying the IH on $\mathcal{D}$ and the premise of $\mathcal{E}$.

Case VI: $\mathcal{E}$ ends in TS-CREATE-U1-GW
The proof is similar to **Case V**.

Case VII: $\mathcal{D}$ ends in TS-CREATE-U2
By assumption,
(VII.1) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_1, L, id) = (id, (v_1'', l_1''), M_1, l_1')$
(VII.2) $l_1' \not\sqsubseteq L$
(VII.3) $\sigma_1' = \sigma_1[id \mapsto (id, (v_1', l_1 \sqcup L), M_1, L)]$
(VII.4) $\alpha_1 = \bullet$
From (VII.2) and our security lattice,
(VII.5) $l_1' = H$
From (1), (VII.1), (VII.5), and Lemma 48 (Requirement **(EH1)**), either
(VII.6) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, L, id) = (id, (v_2'', l_2''), M_2, l_2')$ with $l_2' = H$ or
(VII.7) $\mathsf{lookup}_{\mathsf{TS}}(\sigma_2, L, id) = (\mathsf{NULL}, H)$

Subcase i: (VII.6) is true
From (VII.6),
(i.1) $\mathcal{E}$ ends in TS-CREATE-U2
From (VII.5), (VII.6), and Lemma 84,
(i.2) $M_1 \approx_L M_2 \approx_L \cdot$
From (i.1),
(i.3) $\sigma_2' = \sigma_2[id \mapsto (id, (v_2', l_2 \sqcup L), M_2, L)]$
(i.4) $\alpha_2 = \bullet$
From (1), (VII.3), (i.3), (5), and (i.2),
$\sigma_1' \approx_L \sigma_2'$
From (VII.4) and (i.4),

$\alpha_1 = \alpha_2$

Subcase ii: (VII.7) is true
From (VII.7),
(ii.1) $\mathcal{E}$ ends in TS-CREATE
From (ii.1),
(ii.2) $\sigma_2' = \sigma_2[id \mapsto (id, (v_2', l_2), \cdot, L)]$
(ii.3) $\alpha_2 = \bullet$
From (VII.5) and Lemma 84,
(ii.4) $M_1 \approx_L \cdot$
From (1), (VII.3), (ii.2), (5), and (ii.4),
$\sigma_1' \approx_L \sigma_2'$
From (VII.4) and (ii.3),
$\alpha_1 = \alpha_2$

Subcase iii: $pc_2 = \cdot$
The proof for this case follows from applying the IH on $\mathcal{D}$ and the premise of $\mathcal{E}$.

Case VIII: $\mathcal{E}$ ends in TS-CREATE-U2
The proof is similar to **Case VII**.

Case IX: $\mathcal{D}$ ends in TS-CREATE-NC
The proof for this case follows from applying the IH on the premise of $\mathcal{D}$ and $\mathcal{E}$.

Case X: $\mathcal{E}$ ends in TS-CREATE-NC
The proof is similar to **Case XI**.

□

Lemma 84. *If $\phi = (id, v, M, H)$ were produced by our operational semantics starting in a well-formed state, then $M \approx_L \cdot$*

Proof (sketch): Starting in a well-formed state, the invariant holds, since all event handlers registered to H nodes are labeled H. New nodes created with label H also satisfy the invariant since they are created with an empty event handler map.
The only way to register new event handlers is through one of the registerEH rules. TS-REGISTEREH adds the event handler to the event handler map and performs a join with the label of the node; thus, any new event handler will have label H if the label of the node is H.
The only rule which changes the label of a node is TS-CREATE-U2. This rule changes the label of the node to match the pc only when the label is *above* the pc; i.e. it changes the label from H to L. Therefore, the invariant that tainted nodes never have publicly visible event handlers is maintained by our semantics. □

Appendix G.
Securing TT

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.*

Proof (sketch): The proof follows the same format as the proof for Theorem 2. We consider two cases: one where the last event was a declassification, and another where it wasnt. In the case that the last event was a declassification, the proof follows from the definition of $\mathcal{K}_{rp}()$. When the last event was not a declassification, the proof follows from Lemma 7 (Requirement **(T1)**) and Lemma 17 (Requirement **(T4)**). □

Recall that we strucutre our requirements to be extensible and easily updated. Here, we outline all of the supporting lemmas for proving Theorem 6 and highlight the ones which need updates to prove compositions with TT secure.

Trace Requirements. The proof for Requirement **(T1)** does not need to be changed to prove PINI for TT. It follows from Lemma 9 (Requirement **(T2)**) and Lemma 28 (Requirement **(T5)**).

The proof for Requirement **(T2)** does not need to be changed to prove PINI for TT. It follows from Lemma 66 (Requirement **(EH2)**), Lemma 41 (Requirement **(V2)**), Lemma 73 (Requirement **(EH3)**), and Lemma 74 (Requirement **(EH3)**).

The proof for Requirement **(T3)** does not need to be changed to prove PINI for TT. It follows from Lemma 10 (Requirement **(T2)**) and Lemma 66 (Requirement **(EH2)**).

The proof for Requirement **(T4)** does not need to be changed to prove PINI for TT. It follows from Lemma 56 and Lemma 63 (Requirement **(EH1)**), Lemma 15 and Lemma 16 (Requirement **(T3)**), Lemma 66 (Requirement **(EH2)**), Lemma 36 (Requirement **(E1)**), Lemma 44 (Requirement **(V3)**), and Lemma 80 and Lemma 81 (Requirement **(EH4)**).

The proof for Requirement **(T5)** does not need to be changed to prove PINI for TT. It follows from Lemma 56 and Lemma 63 (Requirement **(EH1)**), Lemma 66 (Requirement **(EH2)**), Lemma 36 (Requirement **(E1)**), Lemma 44 (Requirement **(V3)**), and Lemma 80 and Lemma 81 (Requirement **(EH4)**).

Expression Requirements. The proof for Requirement **(E1)** does not need to be changed to prove PINI for TT. It follows from Lemma 38 and Lemma 39 (Requirement **(V1)**) and Lemma 46 (Requirement **(EH1)**).

Variable Store Requirements. The proofs for Requirement **(V1)** need to be updated to prove TT is secure, but the proofs for Requirements **(V2)** and **(V3)** (global variable storage and variable assignment, respectively) do not need to be changed, nor do they depend on any other requirements. We outline the changes for Requirement **(V1)** below.

Note that we also don't need to add/change any proofs to say that assignments do not leak anything. This is because assignments in the H context can only leak through the global store, which is already proven secure by Requirement **(V2)** for multi-storage techniques. Intuitively, assignments from TT would either change the H copy of the store (which does not leak anything), or would be replaced with a default value in the L copy of the store (which also does not leak anything).

EH storage Requirements. None of the proofs for the event handler storage requirements need to be changed to prove that TT satisfies PINI security, nor do they depend on any other requirements that need to be changed.

Note: When we compose TT with a multi-storage technique, the TT event handlers in the L context no longer receive secrets (i.e., none of the values become tainted). So, there is also no reason to taint default values. We change TT-VAR-DV so that it returns dv labeled with the pc. This also helps maintain the invariant that the local TT store does not have any tainted values when the $pc = L$ (Lemma 86).

Lemma 85. *If $\sigma_1 \approx_L \sigma_2$ and $pc_1, pc_2 \sqsubseteq L$, then for $\mathsf{var}_{\mathsf{TT}}(\sigma_1, pc_1, x) = v_1$ and $\mathsf{var}_{\mathsf{TT}}(\sigma_2, pc_2, x) = v_2$ then $v_1 \simeq_L v_2$*

Proof. By induction on the structure of $\mathcal{E} :: \mathsf{var}_{\mathsf{TT}}(\sigma_1, pc_1, x) = v_1$ and $\mathcal{D} :: \mathsf{var}_{\mathsf{TT}}(\sigma_2, pc_2, x) = v_2$

By assumption,
- (1) $\sigma_1 \approx_L \sigma_2$
- (2) $pc_1, pc_2 \sqsubseteq L$

From Lemma 86,
- (3) $\forall x \in \sigma_1^{\mathsf{TT}}, \mathsf{labOf}(x, \_) = L$
- (4) $\forall x \in \sigma_2^{\mathsf{TT}}, \mathsf{labOf}(x, \_) = L$

From (1) - (4),
- (5) $\sigma_1 = \sigma_2$

Case I: $\mathcal{E}$ ends in TT-VAR

By assumption and from (3),
- (I.1) $x \in \sigma_1$
- (I.2) $\sigma_1(x) = (v_1, L)$

From (I.1) and (5),
- (I.3) $\mathcal{D}$ ends in TT-VAR

From (I.3) and (5),
- (I.4) $\sigma_2(x) = (v_2, L)$

From (5), (I.2), and (I.4)
The desired conclusion holds

Case II: $\mathcal{D}$ ends in TT-VAR
The proof is similar to **Case I**

Case III: $\mathcal{E}$ ends in TT-VAR-DV
By assumption and from (2),
(III.1) $x \notin \sigma_1$
(III.2) $v_1 = (\mathsf{dv}, L)$
From (5) and (III.1),
(III.3) $x \notin \sigma_2$
From (III.3),
(III.4) $\mathcal{D}$ ends in TT-VAR-DV
From (2) and (III.4),
(III.5) $v_2 = (\mathsf{dv}, L)$
From (III.2) and (III.5),
The desired conclusion holds

Case iv: $\mathcal{D}$ ends in TT-VAR-DV
The proof is similar to **Case III**

□

Lemma 86. *Whenever a public event handler is running,* $\forall x \in \sigma^{\mathsf{TT}}, \mathsf{labOf}(x, \_) = L$

Proof (sketch): To prove that the local store only contains public values while public event handlers are running, we need to show that the condition holds when the event handler begins, and is maintained until the event handler finishes.

When the event handler begins running, the local store is empty, so the condition holds trivially.

As the event handler runs, the store is changed by ASSIGN-L. The value assigned is given by the expression semantics. From the assumption that the local store only contains public values, and from Lemma 36 (for expressions involving shared variables), the value being assigned is also public. Therefore, the condition is maintained throughout the event handler execution. □