Journal of Computer Security 29 (2021) 361–401
DOI 10.3233/JCS-211385
IOS Press

Permissive runtime information flow control in the presence of exceptions

Abhishek Bichhawat [a,*,**], Vineet Rajani [b,***], Deepak Garg [c] and Christian Hammer [d]
[a] *IIT Gandhinagar, India*
E-mail: abhishek.b@iitgn.ac.in
[b] *Max Planck Institute for Security and Privacy, Germany*
E-mail: vineet.rajani@csp.mpg.de
[c] *Max Planck Institute for Software Systems, Saarland Informatics Campus, Germany*
E-mail: dg@mpi-sws.org
[d] *University of Potsdam, Germany*
E-mail: hammer@cs.uni-potsdam.de

Abstract. Information flow control (IFC) has been extensively studied as an approach to mitigate information leaks in applications. A vast majority of existing work in this area is based on static analysis. However, some applications, especially on the Web, are developed using dynamic languages like JavaScript where static analyses for IFC do not scale well. As a result, there has been a growing interest in recent years to develop dynamic or runtime information flow analysis techniques. In spite of the advances in the field, runtime information flow analysis has not been at the helm of information flow security, one of the reasons being that the analysis techniques and the security property related to them (non-interference) over-approximate information flows (particularly implicit flows), generating many false positives.

In this paper, we present a sound and precise approach for handling implicit leaks at runtime. In particular, we present an improvement and enhancement of the so-called permissive-upgrade strategy, which is widely used to tackle implicit leaks in dynamic information flow control. We improve the strategy's permissiveness and generalize it. Building on top of it, we present an approach to handle implicit leaks when dealing with complex features like unstructured control flow and exceptions in higher-order languages. We explain how we address the challenge of handling unstructured control flow using immediate post-dominator analysis. We prove that our approach is sound and precise.

Keywords: Runtime information flow control, permissive-upgrade, control-flow graphs, immediate post-dominator analysis, exceptions

1. Introduction

Web applications rely extensively on third-party JavaScript to provide useful libraries, page analytics, advertisements and many other features [50]. In such a *mashup* model, wherein the hosting page and the included scripts share the page's state (called the DOM), all included third-party scripts run with the same access privileges as the hosting page. While some third-party scripts are developed by large, well-known, trustworthy vendors, many other scripts are developed by small, domain-specific vendors whose commercial motives do not always align with those of the web-page providers and users. This leaves

*Corresponding author. E-mail: abhishek.b@iitgn.ac.in.
**Work done while the author was a PhD student at Saarland University.
***Work done while the author was a PhD student at the Max Planck Institute for Software Systems.

sensitive information such as passwords, credit card numbers, email addresses, click histories, cookies and location information vulnerable to inadvertent bugs and deliberate exfiltration by third-party scripts. In many cases, developers are fully aware that a third-party script accesses sensitive data to provide useful functionality, but they are unaware that the script also leaks that data on the side. In fact, this is a widespread problem [41].

The traditional browser security model is based on restricting scripts' *access* to data, not on tracking *how* scripts use data. Existing web security standards like the same-origin policy [11] address this problem unsatisfactorily, favoring functionality over privacy. The same-origin policy (implemented in all major browsers) restricts a web-page and third-party scripts included in it to communicating with web servers from the including web-page's domain only. However, broad exceptions are allowed. For instance, there is no restriction on request parameters in URLs that fetch images and, unsurprisingly, third-party scripts leak information by encoding it in image URLs. Content Security Policy [1], also implemented in most browsers, allows a page to white list scripts that may be included, but places no further restriction on scripts that have been white listed, thus not helping with the problem above.

Quite a few *fine-grained* access control techniques have also been proposed [18,26,27,32,45,47,63,69]. However, all these techniques enforce only access policies and cannot control what a script does with data it has been provided in good faith. That is, if a third-party script was allowed access to some data only for local computations, these techniques do not prevent the script from sending the data on the network. In fact, no mechanism based only on access control can solve the problem of information leakage in this setting.

The academic community has proposed solutions based on *information flow control* (IFC), which ensures the security of confidential information even in the presence of untrusted and buggy code. The idea is to track the flow of information through the program and prevent any undesired flows based on a security policy. Research has considered static methods such as type checking and program analysis, which verify the security policy at compile time [20,21,35,40,48,51,56,65], dynamic methods that track information flow at runtime [5,7–9,13,23,30,38,57,59,61,67], and gradual and hybrid approaches that combine both static and dynamic analyses to add precision to the analysis [12,14,19,25,28,29,34,36,43,49,55,62,64].

While runtime analysis has the drawback of introducing significant performance overheads, it can be more permissive than static analysis methods in certain cases [55]. Moreover, static analyses are mostly ineffective when working with dynamic languages like JavaScript, which is an indispensable part of the modern Web. The dynamic nature of JavaScript [53,54] with features like dynamic typing, runtime code generation (`eval`), scope-chains and prototype chains makes sound static analysis difficult. Thus, recent research has focused on dynamic analysis for enforcing information flow control especially in languages like JavaScript [7–9,14,23,38,59,61]. Even though research in runtime information flow control has made significant inroads in the last decade, the applicability of these techniques still remains bleak. The major challenge to the practicality of runtime information flow control is the conservative handling of implicit leaks, which affects the *permissiveness* of the approaches.

This paper proposes methods to improve the permissiveness of runtime information flow techniques by presenting mechanisms that enhance the precision of the analyses when handling implicit leaks. The two main contributions of the paper are as follows:

- To improve the permissiveness of runtime information flow analysis, this paper presents the *generalized permissive-upgrade strategy*, a sound improvement and enhancement of the permissive-upgrade strategy (which is widely used to tackle implicit leaks when enforcing runtime information

flow control [14,59]). The development improves the original strategy's permissiveness and applicability by generalizing the approach to an arbitrary security lattice, in place of the two-point lattice considered in prior work.

- Most of the existing work in runtime information flow control does not consider implicit leaks due to complex features like unstructured control flow and exceptions. The proposals that handle these features are too conservative and require additional annotations in the program. We present a sound and precise dynamic control scope analysis for handling these features building on top of the generalized permissive upgrade strategy and without requiring any additional annotations from the developer.

This paper extends a part of our conference paper [14] and our workshop paper [13]. In particular, we improve the permissiveness of the existing permissive-upgrade strategy (Section 3), and the formalism presented in the earlier workshop paper [13] to design a more permissive technique for enforcing runtime IFC with arbitrary lattices (Section 4.2). We use the generalized permissive-upgrade strategy to formalize runtime information flow control for a generic system where programs are represented using control-flow graphs and prove it sound (Section 6), and show the precision of our analysis technique (Section 5.3). Several proofs and details are covered in the appendix.

2. Background and overview

2.1. Information flow control

Information flow control (IFC) approaches control the flow of (confidential) information through a program based on a given security policy. Typically, pieces of information are classified into security *labels* and the policy is a lattice over labels. Information is only allowed to flow up the lattice. For illustration purposes often the smallest non-trivial lattice $L \sqsubset H$ is used, which specifies that public (low, L) data must not be influenced by confidential (high, H) data. Information flow control can be used to provide confidentiality (or integrity) of secret (trusted) information; the work in this paper is, however, limited to confidentiality guarantees. Roughly, the idea behind information flow control is that an adversary can view all the public outputs of a program. By preventing private or sensitive data from flowing to public outputs, we prevent the adversary from obtaining any information about the private or sensitive data.

In general, information can flow along many channels. However, this paper considers two of the most important flows – *explicit* and *implicit* – in deterministic programs [20–22]. Covert channels like timing or resource usage are beyond the scope of this paper.

An *explicit flow* occurs as a result of direct assignment, e.g., the statement `public = secret + 1` causes an explicit flow from `secret` to `public`. An *implicit flow* occurs due to the control structure of the program. For instance, consider the program in Listing 1. The final value of `y` is equal to the value of `z` even though there is no direct assignment from `z` to `y`. Leaking a bit like this can be magnified into leaking a bigger secret bit-by-bit [4].

The soundness of approaches enforcing information flow control is often stated in terms of a well-defined security property known as *non-interference* [31], which basically stipulates that high or secret inputs of a program must not influence its low or publicly observable outputs. While non-interference is too strong a property in practice, different variants of the definition are proven. One such variant of non-interference usually established for information flow control techniques is *termination-insensitive non-interference* [65]. Roughly, a program is termination-insensitive non-interferent if any two terminating

```
1 x = false, y = false
2 if (not(z))
3   x = true
4 if (not(x))
5   y = true
```

Listing 1. Implicit flow from z to y

runs of the program starting from low-equivalent heaps (i.e., heaps that look equivalent to the adversary assuming that an adversary can observe some part of the heap) end in low-equivalent heaps. In particular, this means that the parts of the initial heap that the adversary cannot see have no influence on the parts of the final heap that it can see.

2.2. Dynamic information flow control

Dynamic IFC usually works by tracking taints or labels on individual program values in the language runtime. A label represents a mandatory access policy on the value. A value v labeled ℓ is written v^{ℓ}.

Dynamic IFC analysis *propagates* labels as data flows during program execution. *Explicit* flows are generally handled by carrying over the label of the computed value to the variable being assigned. For example, in the statement `x = y + z`, the result of computing `y + z` will have the label that is a join of the labels on `y` and `z`. This will also be the final label of `x`. So, if either of `y` or `z` is labeled H (confidential), then the final label of `x` is also H.[1]

Implicit flows in a runtime IFC analysis are tracked by maintaining an additional taint, usually called the program counter taint or program context taint or *pc*, which is an upper bound on the label of all the control dependencies that lead to the current instruction being executed. For example, in the program of Listing 1, the value in variable `x` at the end of line 3 depends on the value in `z`. If `z` is labeled H, then at line 3, $pc = H$ because of the branch in line 2 that depends on `z`. Thus, by tracking *pc*, dynamic IFC can enforce that `x` has label H at the end of line 3, thus taking into account the control dependency.

However, simply tracking control flow dependencies via *pc* is not enough to guarantee absence of information flows when labels are flow-sensitive, i.e., when the same variable may hold values with different labels depending on what program paths are executed. The program in Listing 1 is a classic counterexample, taken from [7]. Assume that `z` is labeled H and `x` and `y` are labeled L initially. The final value in `y` is computed as a function of the value in `z`. If `z` contains $\texttt{true}^H$, then `y` ends with $\texttt{true}^L$: The branch on line 2 is not taken, so `x` remains $\texttt{false}^L$ at line 4. Hence, the branch on line 4 is taken, but $pc = L$ at line 5 and `y` ends with $\texttt{true}^L$. If `z` contains $\texttt{false}^H$, then similar reasoning shows that `y` ends with $\texttt{false}^L$. Consequently, in both cases `y` ends with label L and its value is exactly equal to the value in `z`. Hence, an adversary can deduce the value of `z` by observing `y` at the end (which is allowed because `y` ends with label L). So, this program leaks information about `z` despite correct use of *pc*.

2.3. Basic IFC semantics

Most of the technical development in this paper is based on the simple imperative language shown in Fig. 1. However, the key ideas are orthogonal to the choice of language and generalize to other languages

[1] "`x` is labeled H" actually means "the value in `x` is labeled H". This language convention is used consistently in the paper.

$$
\begin{aligned}
e &= \mathtt{n} \mid \mathtt{x} \mid e_1 \odot e_2 \\
c &= \mathtt{skip} \mid \mathtt{x} := e \mid c_1; c_2 \mid \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \mid \mathtt{while}\ e\ \mathtt{do}\ c \\
\ell &= L \mid M \mid H \mid \ldots \\
k, l, m, pc &= \ell
\end{aligned}
$$

Fig. 1. Syntax of the language.

EXPRESSIONS:

$$\text{const}\ \frac{}{\langle \sigma, \mathtt{n} \rangle \Downarrow \mathtt{n}^{\perp}} \qquad \text{var}\ \frac{\mathtt{n}^{k} := \sigma(\mathtt{x})}{\langle \sigma, \mathtt{x} \rangle \Downarrow \mathtt{n}^{k}} \qquad \text{oper}\ \frac{\langle \sigma, e' \rangle \Downarrow \mathtt{n}'^{k'} \quad \langle \sigma, e'' \rangle \Downarrow \mathtt{n}''^{k''} \quad \mathtt{n} := \mathtt{n}' \odot \mathtt{n}'' \quad k := k' \sqcup k''}{\langle \sigma, e' \odot e'' \rangle \Downarrow \mathtt{n}^{k}}$$

STATEMENTS:

$$\text{skip}\ \frac{}{\langle \sigma, \mathtt{skip} \rangle \underset{pc}{\Downarrow} \sigma} \qquad \text{seq}\ \frac{\langle \sigma, c_1 \rangle \underset{pc}{\Downarrow} \sigma'' \quad \langle \sigma'', c_2 \rangle \underset{pc}{\Downarrow} \sigma'}{\langle \sigma, c_1; c_2 \rangle \underset{pc}{\Downarrow} \sigma'} \qquad \text{while-f}\ \frac{\langle \sigma, e \rangle \Downarrow \mathtt{false}^{\ell}}{\langle \sigma, \mathtt{while}\ e\ \mathtt{do}\ c \rangle \underset{pc}{\Downarrow} \sigma}$$

$$\text{if-else}\ \frac{\langle \sigma, e \rangle \Downarrow \mathtt{b}^{\ell} \quad i = \left\{ \begin{array}{l} 1, \textit{if}\ \mathtt{b} = \mathtt{true} \\ 2, \textit{otherwise} \end{array} \right\} \quad \langle \sigma, c_i \rangle \underset{pc \sqcup \ell}{\Downarrow} \sigma'}{\langle \sigma, \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \rangle \underset{pc}{\Downarrow} \sigma'} \qquad \text{while-t}\ \frac{\langle \sigma, e \rangle \Downarrow \mathtt{true}^{\ell} \quad \langle \sigma, c \rangle \underset{pc \sqcup \ell}{\Downarrow} \sigma'' \quad \langle \sigma'', \mathtt{while}\ e\ \mathtt{do}\ c \rangle \underset{pc \sqcup \ell}{\Downarrow} \sigma'}{\langle \sigma, \mathtt{while}\ e\ \mathtt{do}\ c \rangle \underset{pc}{\Downarrow} \sigma'}$$

Fig. 2. Semantics.

easily. The use of a simpler language is to simplify non-essential technical details. Parts of the paper that require additional language features define those features in place. The language's expressions include constants or values (n, b), variables (x) and unspecified binary operators ($\odot$) to combine them. The set of variables is fixed upfront. Labels (ℓ) are drawn from a fixed security lattice with a partial order $\sqsubseteq$, join $\sqcup$, meet $\sqcap$, a least element $\perp$ and a top element $\top$. Lattice elements are written L, M, H, ... The meta-variables ℓ and *pc* (program counter label) also ranges over lattice elements.

The rules in Fig. 2 define the big-step semantics of the language, including standard taint propagation for IFC: the evaluation relation $\langle \sigma, e \rangle \Downarrow \mathtt{n}^{k}$ for expressions, and the evaluation relation $\langle \sigma, c \rangle \underset{pc}{\Downarrow} \sigma'$ for commands. Here, σ denotes a store, a map from variables to labeled values of the form $\mathtt{n}^{k}$. b represents

a Boolean constant. For now, labels $k ::= \ell$; this is generalized later when "partially-leaked" taints are introduced in Section 2.5.

The evaluation relation for expressions evaluates an expression e and returns its value n and label k. The label k is the join of labels of all variables occurring in e (according to σ). The relation for commands executes a command c in the context of a store σ, and the current program counter label pc, and yields a new store σ'. The function $\Gamma(\sigma(\mathtt{x}))$ returns the label associated with the value in $\mathtt{x}$ in store σ: If $\sigma(\mathtt{x}) = \mathtt{n}^k$, then $\Gamma(\sigma(\mathtt{x})) = k$.

The sequencing rule (SEQ) evaluates the command c_1 under store σ and the current pc label. This yields a new store σ''. It then evaluates the command c_2 under store σ'' and the same pc label, which yields the final store σ'. The IF-ELSE rule evaluates the branch condition e to a Boolean value $\mathtt{b}$ with label ℓ. Based on the value of $\mathtt{b}$, one of the branches c_1 and c_2 is executed under a pc obtained by joining the current pc and the label ℓ of $\mathtt{b}$. Similarly, the rules for `while` (WHILE-T and WHILE-F) evaluate the loop condition e and execute the loop command c_1 while e evaluates to `true`. The pc for the loop's body is obtained by joining the current pc and the label ℓ of the result of evaluating e.

The rule for assignment statements are conspicuously missing from Fig. 2 because they depend on the strategy used to control implicit flows. We discuss different approaches to handle implicit leaks presented in prior work [7,8] next.

2.4. Dynamically handling implicit leaks using no-sensitive-upgrade check

Preventing leaks due to implicit flow in dynamic IFC requires coarse approximation because a dynamic monitor only sees program branches that are executed and does not know what assignments may happen in alternate branches in other executions. One such coarse approximation is the *no-sensitive-upgrade* (NSU) check proposed by Zdancewic [68]. In the program in Listing 1, $\mathtt{x}$'s label is upgraded from L to H at line 3 in one of the two executions above, but not the other. Subsequently, information leaks in the other execution (where $\mathtt{x}$'s label remains L) via the branch on line 4. The NSU check stops the leak by preventing the assignment on line 3. More generally, it stops a program whenever a public variable's label is upgraded due to a high pc. This check suffices to provide termination-insensitive non-interference, as shown by Austin and Flanagan [7].

The rule for assignment (ASSN-NSU) corresponding to the NSU check is shown in Fig. 3. The rule checks that the label l of the assigned variable $\mathtt{x}$ in the initial store σ is at least as high as pc (premise $pc \sqsubseteq l$). If this condition is not true, the program gets stuck.

2.4.1. Soundness the no-sensitive-upgrade check

For establishing and proving the security property of termination-insensitive non-interference (TINI), the observational power of the adversary needs to be defined. An adversary at level ℓ in the lattice is allowed to view all values that have a label less than or equal to ℓ. To prove the security property

$$\text{assn-nsu}\ \frac{l = \Gamma(\sigma(\mathtt{x})) \qquad pc \sqsubseteq l \qquad \langle \sigma, e\rangle \Downarrow \mathtt{n}^m}{\langle \sigma, \mathtt{x} := e\rangle \underset{pc}{\Downarrow} \sigma[\mathtt{x} \mapsto \mathtt{n}^{(pc \sqcup m)}]}$$

Fig. 3. Assignment rule for NSU.

of non-interference, it is enough to show that when executing a program beginning with two different memory stores that are *observationally equivalent* to an adversary, the final memory stores are also *observationally equivalent* to the adversary. For this, the observational equivalence of two memory stores with respect to an adversary needs to be defined. Store equivalence is formalized as a relation $\sim_\ell$, indexed by lattice elements ℓ, representing the adversary.

Definition 1 (Value equivalence)**.** Two labeled values $\mathtt{n}_1^k$ and n_2^m are ℓ-equivalent, written $n_1^k \sim_\ell n_2^m$, iff either:

(1) $(k = m) \sqsubseteq \ell$ and $n_1 = n_2$ or
(2) $k \not\sqsubseteq \ell$ and $m \not\sqsubseteq \ell$

This definition states that for an adversary at security level ℓ, two labeled values $\mathtt{n}_1^k$ and $\mathtt{n}_2^m$ are equivalent iff either ℓ can access both values, and $\mathtt{n}_1$ and $\mathtt{n}_2$ are equal, or it cannot access either value ($k \not\sqsubseteq \ell$ and $m \not\sqsubseteq \ell$). Note that if $L \sqsubset H$, then any two values labeled L and H are distinguishable for the L-adversary.

Definition 2 (Store equivalence)**.** Two stores σ_1 and σ_2 are ℓ-equivalent, written $\sigma_1 \sim_\ell \sigma_2$, iff for every variable x, $\sigma_1(x) \sim_\ell \sigma_2(x)$.

The following theorem states TINI for the NSU check. The theorem has been proved for various languages in the past.

Theorem 1 (TINI for NSU)**.** *With the assignment rule* ASSN-NSU *from Fig. 3, if* $\sigma_1 \sim_\ell \sigma_2$ *and* $\langle \sigma_1, c\rangle \underset{pc}{\Downarrow} \sigma_1'$ *and* $\langle \sigma_2, c\rangle \underset{pc}{\Downarrow} \sigma_2'$*, then* $\sigma_1' \sim_\ell \sigma_2'$.

Proof. Standard, see e.g., [7]. □

2.5. Permissive-upgrade strategy for handling implicit leaks

The no-sensitive-upgrade (NSU) check described earlier provides the basic foundations for sound dynamic IFC. However, terminating a program preemptively because of the NSU check is quite restrictive in practice. For example, consider the program of Listing 2, where `z` is labeled H and `y` is labeled L. This program potentially upgrades variable `x` at line 3 under a high *pc*, and then executes function `f` when `y` is `true` and executes function `g` otherwise. Suppose that `f` does not read `x`. Then, for $\mathtt{y} \mapsto \mathtt{true}^L$, this program leaks no information, but the NSU check would terminate this program prematurely at line 3. (Note: `g` may read `x`, so `x` is not a dead variable at line 3.)

```
1 x = false
2 if (not(z))
3   x = true
4 if (y) f() else g()
5 x = false
```

Listing 2. Impermissiveness of the NSU strategy

To allow a dynamic IFC analysis to accept safe executions of programs with variable upgrades due to high *pc*, Austin and Flanagan proposed a less restrictive strategy called the *permissive-upgrade strategy* [8]. They study this strategy for a two-point lattice $L \sqsubset H$ and their strategy does not immediately generalize to arbitrary security lattices. Whereas NSU stops a program when a variable's label is upgraded due to assignment in a high *pc*, permissive-upgrade allows the assignment, but labels the variable as *partially-leaked* or P. The exact intuition behind the partially-leaked label P is the following:

> A variable with a value labeled P may have been implicitly influenced by H-labeled values in this execution, but in other executions (obtainable by changing H-labeled values in the initial store), this implicit influence may not exist and, hence, the variable may be labeled L.

The program must be stopped later if it tries to use or case-analyze the variable (in particular, branching on a partially-leaked Boolean variable is stopped). Permissive-upgrade also ensures termination-insensitive non-interference, but is strictly more permissive than NSU. For example, permissive-upgrade stops the leaky program of Listing 1 at line 4 when `z` contains $\texttt{false}^H$, but it allows the program of Listing 2 to execute to completion when `y` contains $\texttt{true}^L$.

In the revised syntax of labels, summarized in Fig. 4, the labels k, l, m on values can be either elements of the lattice (L, H) or P. The *pc* can only be one of L, H because branching on partially-leaked values is prohibited. The join operation $\sqcup$ is lifted to labels including P. Joining any label with P results in P. For brevity in definitions, the order is extended to $L \sqsubset H \sqsubset P$. However, P is not a new "top" member of the lattice because it receives special treatment in the operational semantics.

The rule for assignment with permissive-upgrade is

$$\text{assn-pus}\ \frac{l := \Gamma(\sigma(\mathtt{x})) \qquad \langle \sigma, e\rangle \Downarrow \mathtt{n}^m}{\langle \sigma, \mathtt{x} := e\rangle \underset{pc}{\Downarrow} \sigma[\mathtt{x} \mapsto \mathtt{n}^k]}$$

where k is defined as follows:

$$k = \begin{cases} m & \text{if } pc = L \\ m \sqcup H & \text{if } pc = H \text{ and } l = H \\ P & \text{otherwise} \end{cases}$$

The first two conditions in the definition of k correspond to the NSU rule (Fig. 3). The third condition applies, in particular, when a variable whose initial label is L is assigned where $pc = H$. The NSU

$$\begin{aligned} \ell &= L \mid H \\ pc &= \ell \\ k, l, m &= \ell \mid P \end{aligned} \qquad\qquad \begin{aligned} k \sqcup k &= k \\ L \sqcup H &= H \\ L \sqcup P &= P \\ H \sqcup P &= P \end{aligned}$$

Fig. 4. Syntax of labels including the partially-leaked label P.

check would stop this assignment. With permissive-upgrade, however, the updated variable is labeled P, consistent with the intuitive meaning of P. This allows more permissiveness by allowing the assignment to proceed in all cases. To compensate, any program (in particular, an adversarial program) is disallowed from case analyzing any value labeled P. Consequently, in the rules for `if-then` and `while` (Fig. 2), the label of the branch condition is of the form ℓ, which does not include P. Thus, assignments under high *pc* succeed under the permissive-upgrade check but branching or case-analyzing a partially-leaked value is not permitted as that can also leak information.

The noninterference result obtained for NSU earlier can be extended to permissive-upgrade by changing the definition of store equivalence. Because no program can case-analyze a P-labeled value, such a value is equivalent to any other labeled value.

Definition 3. Two labeled values n_1^k and n_2^m are equivalent to an adversary at level L, written $n_1^k \sim_L n_2^m$, iff either:

(1) $(k = m) = L$ and $n_1 = n_2$ or
(2) $k = H$ and $m = H$ or
(3) $k = P$ or $m = P$

Definition 4. Two stores σ_1 and σ_2 are L-equivalent, written $\sigma_1 \sim_L \sigma_2$, iff $\forall x.\sigma_1(x) \sim_L \sigma_2(x)$.

Theorem 2 (TINI for permissive-upgrade with a two-point lattice)**.** *With the assignment rule* assn-PUS*, if* $\sigma_1 \sim_L \sigma_2$ *and* $\langle c, \sigma_1\rangle \Downarrow_{pc} \sigma_1'$ *and* $\langle c, \sigma_2\rangle \Downarrow_{pc} \sigma_2'$*, then* $\sigma_1' \sim_L \sigma_2'$*.*

Proof. See [8]. □

3. Improved permissive-upgrade strategy

The original permissive-upgrade strategy as described above lacks permissiveness in some cases. For example, it rejects the program of Listing 3, which is actually secure. Consider that `x` is labeled H and `w`, `y` are labeled L. With the original permissive-upgrade strategy, the label of `z` on line 3 would remain P and the execution would be terminated when branching on `z` on line 4.

Our improvement makes a small change to the definition of $\sqcup$. We define:

$$H \sqcup P = H$$

Revisiting the example with this change, `z` would be labeled H on line 3, which would allow the execution to branch on line 4, thus taking the execution to completion. The idea behind the improvement is

```
1 if (not(x))
2   y = true
3 z = y || x
4 if (not(z))
5   w = true
```

Listing 3. Impermissiveness of the permissive-upgrade strategy

that an H-labeled value is never observable at L-level. Similarly, the result of any operation involving an H-labeled value is also never observable at L-level. Thus, the result of combining P-labeled value with a H-labeled value can safely be labeled H.

Following this change, we also redefine the label k in the assignment rule ASSN-PUS.

$$k = \begin{cases} m & \text{if } pc = L \\ H & \text{if } pc = H \text{ and } l = H \\ P & \text{otherwise} \end{cases}$$

The soundness results of the original permissive-upgrade strategy can be extended to show the soundness of the improved permissive-upgrade strategy. However, a significant difficulty in proving soundness with P is that the definition of $\sim$ is no longer transitive. In [8], the authors resolve this issue by defining a special relation called evolution. We follow a related but non-identical proof strategy for showing the improved permissive-upgrade strategy sound.

Lemma 1 (Expression evaluation)**.** *If* $\langle \sigma_1, e\rangle \Downarrow \mathrm{n}_1^{k_1}$ *and* $\langle \sigma_2, e\rangle \Downarrow \mathrm{n}_2^{k_2}$ *and* $\sigma_1 \sim_L \sigma_2$*, then* $\mathrm{n}_1^{k_1} \sim_L \mathrm{n}_2^{k_2}$.

Proof. By induction on e. □

Lemma 2 (Evolution)**.** *If* $pc = H$ *and* $\langle \sigma, c\rangle \underset{pc}{\Downarrow} \sigma'$*, then* $\forall \mathrm{x}.\ \Gamma(\sigma(\mathrm{x})) = P \implies \Gamma(\sigma'(\mathrm{x})) = P$.

Proof. By induction on the derivation rules and case analysis on the last rule. □

Lemma 3 (Confinement for improved permissive-upgrade with a two-point lattice)**.** *If* $pc = H$ *and* $\langle \sigma, c\rangle \underset{pc}{\Downarrow} \sigma'$*, then* $\sigma \sim_L \sigma'$.

Proof. By induction on the given derivation. □

Theorem 3 (TINI for improved permissive-upgrade with a two-point lattice)**.** *With the assignment rule* ASSN-PUS *and the modifications above, if* $\sigma_1 \sim_L \sigma_2$ *and* $\langle \sigma_1, c\rangle \underset{pc}{\Downarrow} \sigma_1'$ *and* $\langle \sigma_2, c\rangle \underset{pc}{\Downarrow} \sigma_2'$*, then* $\sigma_1' \sim_L \sigma_2'$.

Proof. By induction on c and case analysis on the last step. □

Detailed proofs of the above claims are in Appendix A.1. Note that the definitions and proofs presented in this section are specific to the two-point lattice and with respect to an adversary at level L.

4. Generalized permissive-upgrade strategy

Although the permissive-upgrade strategy as described above is useful, its development is incomplete so far: Austin and Flanagan's original paper [8], and our improvement (Section 3), both develop permissive-upgrade for *only* a two-point security lattice, containing levels L and H with $L \sqsubset H$, and the new label P. A generalization to a pointwise product of such two-point lattices (and, hence, a powerset lattice) was suggested by Austin and Flanagan in the original paper, but not fully developed. As

explained later in Section 4.1, this generalization works for the improved permissive-upgrade strategy and can be proven sound.

However, that still leaves open the question of generalizing permissive-upgrade to *arbitrary lattices*. It is not even clear hitherto that this generalization exists. This section shows by construction that a generalization of permissive-upgrade (with our improvement of Section 3) to arbitrary lattices does indeed exist and that it is, in fact, non-obvious. Specifically, the rule for adding partially-leaked labels and the definition of store (memory) equivalence needed to prove non-interference are reasonably involved.

4.1. Generalized improved permissive-upgrade strategy on powerset lattices

Austin and Flanagan point out that permissive-upgrade on a two-point lattice can be generalized to a pointwise product of such lattices. This generalization can also be extended to the improved permissive-upgrade strategy presented above. Specifically, let X be an index set – these indices are called principals (like Alice, Bob etc.) in [8]. Let a label l be a map of type $X \to \{L, H, P\}$ and let the subclass of pure labels contain maps ℓ, pc of type $X \to \{L, H\}$. The order $\sqsubset$ and the join operation $\sqcup$ can be generalized pointwise to these labels. Finally, the rule ASSN-PUS can be generalized pointwise by replacing it with the following rule:

$$\text{assn-gpus}\ \frac{l := \Gamma(\sigma(\mathrm{x})) \qquad \langle \sigma, e\rangle \Downarrow \mathrm{n}^m}{\langle \sigma, \mathrm{x} := e\rangle \underset{pc}{\Downarrow} \sigma[\mathrm{x} \mapsto \mathrm{n}^k]}$$

where k is defined as follows:

$$k(a) = \begin{cases} m(a) & \text{if } pc(a) = L \\ H & \text{if } pc(a) = H \text{ and } l(a) = H \\ P & \text{otherwise} \end{cases}$$

In the definition above, a represents a principal like Alice from the set X. It can be shown that for any semantic derivation in this generalized system, projecting all labels to a given principal yields a valid semantic derivation in the system with a two-point lattice. This immediately implies non-interference for the generalized system, where observations are limited to individual principals.

Definition 5. Two labeled values n_1^k and n_2^m are a-equivalent, written $n_1^k \approx^a n_2^m$, iff either:

(1) $k(a) = m(a) = L$ and $n_1 = n_2$ or
(2) $k(a) = m(a) = H$ or
(3) $k(a) = P$ or $m(a) = P$

Definition 6 (Store equivalence)**.** Two stores σ_1 and σ_2 are ℓ-equivalent, written $\sigma_1 \approx^a \sigma_2$, iff for every variable x, $\sigma_1(x) \approx^a \sigma_2(x)$.

Theorem 4 (TINI for permissive-upgrade with a product lattice)**.** *With the assignment rule* ASSN-GPUS*, if* $\sigma_1 \approx^a \sigma_2$ *and* $\langle \sigma_1, c\rangle \underset{pc}{\Downarrow} \sigma_1'$ *and* $\langle \sigma_2, c\rangle \underset{pc}{\Downarrow} \sigma_2'$*, then* $\sigma_1' \approx^a \sigma_2'$.

Proof. The proof follows from Theorem 3 for every principal a. □

This generalization of the two-point lattice to a *product* of such lattices is interesting because a *powerset* lattice can be simulated using such a product. However, this still leaves open the question of constructing a generalization of permissive-upgrade to an *arbitrary* lattice (for instance, lattices like the one shown in Fig. 7). Such a generalization is developed next.

4.2. Generalized improved permissive-upgrade on arbitrary lattices

This section shows by construction the generalization of the permissive-upgrade strategy to arbitrary security lattices. For every element ℓ of the lattice, a new label $\ell^\star$ is introduced which means "partially-leaked ℓ", with the following intuition:

> A variable labeled $\ell^\star$ may contain partially-leaked data, where ℓ is a *lower-bound* on the $\star$-free labels the variable may have in alternate executions.

The syntax of labels is listed in Fig. 5. Labels k, l, m may be lattice elements ℓ or $\star$-ed lattice elements $\ell^\star$. In examples, we continue to use suggestive lattice element names L, M, H (low, medium, high). Labels of the form ℓ are called $\star$-free or *pure*. Figure 5 also defines the join operation $\sqcup$ on labels. This definition is based on the intuition above. When the two operands of $\odot$ are labeled ℓ_1 and $\ell_2^\star$, $\ell_1 \sqcup \ell_2$ is a lower bound on the pure label of the resulting value in any execution (because ℓ_2 is a lower bound on the pure label of $\ell_2^\star$ in any run). Hence, $\ell_1 \sqcup \ell_2^\star = (\ell_1 \sqcup \ell_2)^\star$. The reason for the definition $\ell_1^\star \sqcup \ell_2^\star = (\ell_1 \sqcup \ell_2)^\star$ is similar.

The rules for assignment are shown in Fig. 6. They strictly generalize the rule ASSN-PUS for the two-point lattice (where $P = L^\star$). Rule ASSN-N applies when the existing label of the variable being

$$\begin{aligned} \ell &= L \mid M \mid H \mid \ldots \\ pc &= \ell \\ k, l, m &= \ell \mid \ell^\star \end{aligned} \qquad \begin{aligned} \ell_1 \sqcup \ell_2^\star &= (\ell_1 \sqcup \ell_2)^\star \\ \ell_1^\star \sqcup \ell_2^\star &= (\ell_1 \sqcup \ell_2)^\star \end{aligned}$$

Fig. 5. Labels and label operations.

$$\text{assn-n}\ \frac{\langle \sigma, e\rangle \Downarrow \mathrm{n}^m \qquad l = \Gamma(\sigma(\mathrm{x})) \qquad l = \ell_\mathrm{x} \vee l = \ell_\mathrm{x}^\star \qquad pc \sqsubseteq \ell_\mathrm{x} \qquad k = pc \sqcup m}{\langle \sigma, \mathrm{x} := e\rangle \underset{pc}{\Downarrow} \sigma[\mathrm{x} \mapsto \mathrm{n}^k]}$$

$$\text{assn-s}\ \frac{\langle \sigma, e\rangle \Downarrow \mathrm{n}^m \qquad l = \Gamma(\sigma(\mathrm{x})) \qquad l = \ell_\mathrm{x} \vee l = \ell_\mathrm{x}^\star \qquad pc \not\sqsubseteq \ell_\mathrm{x} \qquad k = ((pc \sqcup m) \sqcap \ell_\mathrm{x})^\star}{\langle \sigma, \mathrm{x} := e\rangle \underset{pc}{\Downarrow} \sigma[\mathrm{x} \mapsto \mathrm{n}^k]}$$

Fig. 6. Assignment rules for the generalized permissive-upgrade.

assigned to is ℓ_x or $\ell_x^\star$ and $pc \sqsubseteq \ell_x$. The key intuition behind the rule is the following: If $pc \sqsubseteq \ell_x$, then it is safe to overwrite the variable, because ℓ_x is necessarily a lower bound on the (pure) label of x in this and any alternate execution (see the framebox above). Hence, overwriting the variable cannot cause an implicit flow. As expected, the label of the overwritten variable is $pc \sqcup m$, where m is the label of the value assigned to x.

Rule ASSN-S applies in the remaining case – when $pc \not\sqsubseteq \ell_x$. In this case, there may be an implicit flow, so the final label on x must have the form $\ell^\star$ for some ℓ. The question is which ℓ. Intuitively, it may seem that one could choose $\ell = \ell_x$, the pure part of the original label of x. The final label on x would be $\ell_x^\star$ and this would satisfy the intuitive meaning of $\star$ written in the framebox above. Indeed, this intuition suffices for the two-point lattice of Section 2.5 and 3. However, for a more general lattice, this intuition is unsound, as illustrated with an example below. The correct label is $((pc \sqcup m) \sqcap \ell_x)^\star$.

Example. We illustrate the need for the label $k := ((pc \sqcup m) \sqcap \ell_x)^\star$ instead of $k := \ell_x^\star$ in the rule ASSN-S. Consider the lattice of Fig. 7 and the program of Listing 4. Assume that, initially, the variables z, w, x_1, x', x_2, y_1 and y_2 have labels H, L_1, L_1, L', L_2, M_1 and M_2, respectively. Fix the attacker at level L_1. Fix the value of x_1 at $\texttt{true}^{L_1}$, so that the branch on line 5 is always taken and line 6 is always executed. Set $y_1 \mapsto \texttt{false}^{M_1}$, $y_2 \mapsto \texttt{true}^{M_2}$, $w \mapsto \texttt{false}^{L_1}$ initially. The initial value of z is irrelevant. Consider two executions of the program starting from two stores σ_1 with $x' \mapsto \texttt{true}^{L'}$, $x_2 \mapsto \texttt{true}^{L_2}$ and σ_2 with $x' \mapsto \texttt{false}^{L'}$, $x_2 \mapsto \texttt{false}^{L_2}$. Note that as L' and L_2 are incomparable to L_1 in the lattice, σ_1 and σ_2 are equivalent for L_1.

Requiring $k := \ell_x^\star$ in rule ASSN-S causes an implicit flow that is observable for L_1. The intermediate values and labels of the variables for executions starting from σ_1 and σ_2 are shown in the second and

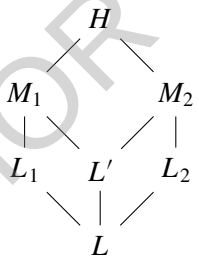

Fig. 7. Lattice explaining rule ASSN-S.

```
1  if (x')
2    z = y1
3  else
4    z = y2
5  if (x1)
6    z = x1
7  if (not(x2))
8    z = x2
9  if (z)
10   w = z
```

Listing 4. Example explaining rule ASSN-S

Table 1
Execution steps in two runs of the program from Listing 4, with two variants of the rule ASSN-S

	$\mathtt{w} = \mathtt{false}^{L_1}, \mathtt{x}_1 = \mathtt{true}^{L_1}, \mathtt{y}_1 = \mathtt{false}^{M_1}, \mathtt{y}_2 = \mathtt{true}^{M_2}$		
	$\mathtt{x}' = \mathtt{true}^{L'}$ $\mathtt{x}_2 = \mathtt{true}^{L_2}$	$\mathtt{x}' = \mathtt{false}^{L'}$ $\mathtt{x}_2 = \mathtt{false}^{L_2}$	
		$k := \ell_{\mathtt{x}}^\star$	$k := ((pc \sqcup m) \sqcap \ell_{\mathtt{x}})^\star$
`if (x')`	$pc = L'$		
`z = y1`	$\mathtt{z} = \mathtt{false}^{M_1}$		
`else`		$pc = L'$	$pc = L'$
`z = y2`		$\mathtt{z} = \mathtt{true}^{M_2}$	$\mathtt{z} = \mathtt{true}^{M_2}$
`if (x1)`	$pc = L_1$	$pc = L_1$	$pc = L_1$
`z = x1`	$\mathtt{z} = \mathtt{true}^{L_1}$	$\mathtt{z} = \mathtt{true}^{M_2^\star}$	$\mathtt{z} = \mathtt{true}^{L^\star}$
`if (not(x2))`	branch not taken	$pc = L_2$	$pc = L_2$
`z = x2`		$\mathtt{z} = \mathtt{false}^{L_2}$	$\mathtt{z} = \mathtt{false}^{L^\star}$
`if (z)`	$pc = L_1$	branch not taken	execution halted
`w = z`	$\mathtt{w} = \mathtt{true}^{L_1}$		
Result	$\mathtt{w} = \mathtt{true}^{L_1}$	$\mathtt{w} = \mathtt{false}^{L_1}$ (leak)	no leak

third columns of Table 1. Starting with σ_1, line 2 is executed, but line 4 is not, so z ends with $\mathtt{false}^{M_1}$ at line 5 (rule ASSN-N applies at line 2). At line 6, z contains $\mathtt{true}^{L_1}$ (again by rule ASSN-N) and line 8 is not executed. Thus, the branch on line 9 is taken and w ends with $\mathtt{true}^{L_1}$ at line 10. Starting with σ_2, line 2 is not executed, but line 4 is, so z becomes $\mathtt{true}^{M_2}$ at line 5 (rule ASSN-N applies at line 4). At line 6, rule ASSN-S applies, but because $k := \ell_{\mathtt{x}}^\star$ is assumed in that rule, z now contains the value $\mathtt{true}^{M_2^\star}$. As the branch on line 7 is taken, at line 8, z becomes $\mathtt{false}^{L_2}$ by rule ASSN-N because $L_2 \sqsubseteq M_2$. Thus, the branch on line 9 is not taken and w ends with $\mathtt{false}^{L_1}$ in this execution. Hence, w ends with $\mathtt{true}^{L_1}$ and $\mathtt{false}^{L_1}$ in the two executions, respectively. The attacker at level L_1 can distinguish these two results and, hence, the program leaks the value of x′ and $\mathtt{x}_2$ to L_1.

With the correct ASSN-S rule in place, this leak is avoided (last column of Table 1). In that case, after the assignment on line 6 in the second execution, z has label $((L_1 \sqcup L_1) \sqcap M_2)^\star = L^\star$. Subsequently, after line 8, z gets the label $L^\star$. As case analysis on a $\star$-ed value is not allowed, the execution is halted on line 9. This guarantees termination-insensitive non-interference with respect to the attacker at level L_1.

4.3. Termination-insensitive non-interference (TINI)

To prove non-interference for the generalized permissive-upgrade, equivalence of labeled values relative to an adversary at arbitrary lattice level ℓ needs to be defined. The definition is shown below (Definition 7). Note that clauses (3)–(5) here refine clause (3) of Definition 5 for the two-point lattice. The obvious generalization of clause (3) of Definition 5 – $\mathtt{n}_1^k \sim_\ell \mathtt{n}_2^m$ whenever either k or m is $\star$-ed – is too coarse to prove non-interference inductively. For the degenerate case of the two-point lattice, this definition also degenerates to Definition 5 (there, ℓ is fixed at L, $P = L^\star$ and only L may be $\star$-ed).

Definition 7. Two values n_1^k and n_2^m are ℓ-equivalent, written $n_1^k \sim_\ell n_2^m$, iff either

(1) $k = m = \ell' \sqsubseteq \ell$ and $n_1 = n_2$, or
(2) $k = \ell' \not\sqsubseteq \ell$ and $m = \ell'' \not\sqsubseteq \ell$, or
(3) $k = \ell_1^\star$ and $m = \ell_2^\star$, or
(4) $k = \ell_1^\star$ and $m = \ell_2$ and ($\ell_2 \not\sqsubseteq \ell$ or $\ell_1 \sqsubseteq \ell_2$), or

(5) $k = \ell_1$ and $m = \ell_2^\star$ and ($\ell_1 \not\sqsubseteq \ell$ or $\ell_2 \sqsubseteq \ell_1$)

Definition 8. Two stores σ_1 and σ_2 are ℓ-equivalent, written $\sigma_1 \sim_\ell \sigma_2$, iff for every variable x, $\sigma_1(x) \sim_\ell \sigma_2(x)$.

This definition is obtained by constructing (through examples) an extensive transition graph of pairs of labels that may be assigned to a single variable at corresponding program points in two executions of the same program. The starting point is label-pairs of the form (ℓ, ℓ). This characterization of equivalence is both sufficient and necessary. It is sufficient in the sense that it allows us to prove TINI inductively. It is necessary in the sense that example programs can be constructed that end in states exercising every possible clause of this definition. Appendix A.2 lists these examples.

Using the above definition of equivalence of labeled values, TINI can be proven for the generalized permissive-upgrade strategy presented above. As before, a significant difficulty in proving the theorem is that the definition of $\sim_\ell$ is not transitive. Detailed proofs of all the lemmas and the theorems are presented in Appendix A.3.

Lemma 4 (Expression evaluation)**.** *If* $\langle \sigma_1, e\rangle \Downarrow \mathrm{n}_1^{k_1}$ *and* $\langle \sigma_2, e\rangle \Downarrow \mathrm{n}_2^{k_2}$ *and* $\sigma_1 \sim_\ell \sigma_2$*, then* $\mathrm{n}_1^{k_1} \sim_\ell \mathrm{n}_2^{k_2}$.

Proof. By induction on e. □

Lemma 5 ($\star$-preservation)**.** *If* $\langle \sigma, c\rangle \underset{pc}{\Downarrow} \sigma'$ *and* $\Gamma(\sigma(x)) = \ell^\star$ *and* $pc \not\sqsubseteq \ell$*, then* $\Gamma(\sigma'(x)) = \ell'^\star$ *and* $\ell' \sqsubseteq \ell$.

Proof. By induction on the given derivation. □

Corollary 1. *If* $\langle \sigma, c\rangle \underset{pc}{\Downarrow} \sigma'$ *and* $\Gamma(\sigma(\mathrm{x})) = \ell^\star$ *and* $\Gamma(\sigma'(\mathrm{x})) = \ell'$*, then* $pc \sqsubseteq \ell$.

Proof. Immediate from Lemma 5. □

Lemma 6 (*pc*-lemma)**.** *If* $\langle \sigma, c\rangle \underset{pc}{\Downarrow} \sigma'$ *and* $\Gamma(\sigma'(\mathrm{x})) = \ell$*, then* $\sigma(\mathrm{x}) = \sigma'(\mathrm{x})$ *or* $pc \sqsubseteq \ell$.

Proof. By induction on the given derivation. □

Corollary 2. *If* $\langle \sigma, c\rangle \underset{pc}{\Downarrow} \sigma'$ *and* $\Gamma(\sigma(\mathrm{x})) = \ell^\star$ *and* $\Gamma(\sigma'(\mathrm{x})) = \ell'$*, then* $pc \sqsubseteq \ell'$.

Proof. Immediate from Lemma 6. □

Using these lemmas, the standard confinement lemma and non-interference can be proven.

Lemma 7 (Confinement lemma)**.** *If* $pc \not\sqsubseteq \ell$ *and* $\langle \sigma, c\rangle \underset{pc}{\Downarrow} \sigma'$*, then* $\sigma \sim_\ell \sigma'$.

Proof. By induction on the given derivation. □

Theorem 5 (TINI for generalized permissive-upgrade for arbitrary lattices)**.** *If* $\sigma_1 \sim_\ell \sigma_2$ *and* $\langle \sigma_1, c\rangle \underset{pc}{\Downarrow} \sigma_1'$ *and* $\langle \sigma_2, c\rangle \underset{pc}{\Downarrow} \sigma_2'$*, then* $\sigma_1' \sim_\ell \sigma_2'$.

Proof. By induction on c. □

4.4. Comparison of the generalization of Section 4.2 with the generalization of Section 4.1

Two distinct and sound generalizations of the permissive-upgrade strategy for the two-point lattice have now been described: The generalization of the improved permissive-upgrade to pointwise products of two-point lattices or, equivalently, to powerset lattices as described in Section 4.1, and the generalization to arbitrary lattices described in Section 4.2. Since both the generalizations apply to powerset lattices, an obvious question is whether one is more permissive than the other on such lattices. The generalization of permissive-upgrade to pointwise lattices described in Section 4.1 can be more permissive than the generalization described in Section 4.2 for powerset lattices in certain cases as shown by the example below. The reason for this permissiveness is that the generalization of permissive-upgrade to pointwise lattices tracks finer taints, i.e., it tracks partial leaks for each principal separately.

Example. We use the powerset lattice of Fig. 8. This lattice is the pointwise lifting of the order $L \sqsubset H$ to the set $S = \{L, H\} \times \{L, H\}$. For brevity, this lattice's elements are written as *LL*, *LH*, etc. When generalization from Section 4.1 is applied to this lattice, labels are drawn from the set $\{L, H, P\} \times \{L, H, P\}$. These labels are concisely written as *LP*, *HL*, etc. For the generalization from Section 4.2 to arbitrary lattices, labels are drawn from the set $S \cup S^\star$. These labels are written *LH*, $LH^\star$, etc. Note that $LH^\star$ parses as $(LH)^\star$, not $L(H^\star)$ (the latter is not a valid label in the generalization applied to this lattice). Consider the program in Listing 5. Assume that `x`, `y` and `z` have initial labels *LL*, *HL* and *LH*, respectively and that the initial store contains $\texttt{y} \mapsto \texttt{true}^{HL}$, $\texttt{z} \mapsto \texttt{true}^{LH}$, so the branches on lines 1 and 3 are both taken. The initial value in `x` is irrelevant but its label is important. Under the generalization

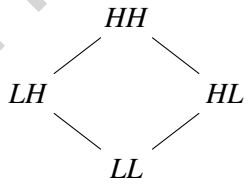

Fig. 8. A powerset/product lattice.

```
1  if (y)
2    x = z
3  if (z)
4    x = z
5  if (x)
6    z = x
```

Listing 5. Example where generalization of permissive-upgrade to pointwise lattices is more permissive than the generalization to arbitrary lattices

from Section 4.1, x obtains label $(((HL) \sqcup (LH)) \sqcap (LL))^\star = LL^\star$ at line 2 by rule ASSN-S. At line 4, the same rule applies but the label of x remains $LL^\star$. When the program tries to branch on x at line 5, it is stopped. In contrast, under generalization of permissive-upgrade to pointwise lattices, this program executes to completion. At line 2, the label of x changes to PH by rule ASSN-GPUS. At line 4, the label changes to LH because *pc* and the label of z are both LH. Since this new label has no P, line 5 executes without halting. Hence, for this example, generalization of permissive-upgrade to pointwise lattices is more permissive than the generalization to arbitrary lattices presented in Section 4.2.

We could not find an example for which the generalization of permissive-upgrade to arbitrary lattices is more permissive than generalization to pointwise lattices in the case of powerset lattices. But the generalization presented in Section 4.2 is more general than the product construction (Section 4.1) when applied to arbitrary lattices (and hence, applicable to a broader set of lattices) as it is unclear whether or how the results of Section 4.1 apply to arbitrary lattices.

5. Handling implicit leaks with complex features

Implicit flows correspond to control dependence in program analysis, where branch conditions govern which program path is executed and leak information through the control flow of the program. For sound analysis and to avoid over-approximation in certain cases, the generalized permissive upgrade strategy presented earlier works by determining when the influence of a control construct has *ended*. This is necessary to lower the *pc* label for subsequent program statements, thus preventing overtainting.

For block-structured control flow limited to `if` and `while` commands, it is straightforward to determine where the influence of a control construct ends as this is obvious *syntactically*. For example, in the program

```
if (h) {l = 1;} l = 2
```

h influences the control flow at `l = 1` but not at `l = 2`. In a big-step operational semantics, lowering the *pc* after the influence of a control construct ends is very straightforward since the *pc* is raised *exactly* for the syntactic scope of the control construct. In a small-step semantics, a bit more is required, but the overall setup is still straightforward: One maintains a *stack* of *pc* labels [68]; the effective *pc* is the top one. When entering a control flow construct like `if` or `while`, a new *pc* label, equal to the join of labels of all values on which the construct's guard depends with the previous effective *pc*, is pushed. When exiting the construct, the label is popped.

Unfortunately, it is unclear how to extend this simple strategy (with both big-step and small-step semantics) to non-block-structured control flow constructs such as `break`, `continue` and `return`-in-the-middle for functions, all of which occur in imperative high-level languages. For example, consider the program

```
l = 1; while(1){...if(h){break; }; l = 0; break; }
```

with h labeled H. This program leaks the value of h into l, but no assignment to l appears in a *syntactic* block-scope guarded by h. Indeed, the lexical scoping strategy just described is ineffective for this program. Tracking information flow in the presence of such unstructured control flows is non-trivial as the control breaks out of block structures.

Exceptions are even more difficult to handle as they allow for non-local control transfer. Consider, for instance, the program snippet shown below:

```
function f() = {
  l = 0;
  try { g(); } catch(e) { l = 1; }
  return l;
}
```

```
function g() = {
  if (h) {throw 9;}
  return 7;
}
```

The function `f()` calls the function `g()`, which throws an exception that is caught by the exception handler in `f()` if the value of `h` is not `false`. The exception handler modifies the value of the public variable `l`, which the function returns. When `f` is invoked with $pc = L$, the two functions together leak the value of `h`, which is assumed to have a label H, into the return value of `f`.

To solve this issue of handling implicit leaks with complex features, we present a precise dynamic analysis approach building on top of the generalized permissive upgrade strategy using *post-dominator* analysis [22,42].

5.1. Control flow graphs and post-dominator analysis

Our approach is based on an on-the-fly post-dominator analysis at runtime to handle implicit flows. A control flow graph (CFG), which is a directed graph, is constructed for every new function before it is executed with every instruction being represented as a node and whose edges represent the possible control flows. For every branch node, its immediate post-dominator (IPD) is computed [14,22,42]. A stack of *pc* labels is maintained. When executing a branch node, a new *pc* label is pushed on the stack *along with* the node's IPD. When the IPD is actually reached, the *pc* label along with the IPD is popped from the stack. In prior work [46,66], it is proved that the IPD marks the end of the scope of an operation and hence the security context of the operation, so our strategy is sound. The IPD-based solution works for all forms of unstructured control flow like break, continue, return-in-the-middle, and exceptions. Multiple return statements in a function can be represented as a single return node.

Maintaining a precise CFG for post-dominator analysis at runtime in the presence of exceptions is expensive. As the CFG of a function is constructed on-the-fly when compiling the function at runtime, an exception-throwing node in a function that does not catch that exception should have an outgoing control flow edge to the next related exception handler in the runtime call-stack. This means that the CFG is, in general, inter-procedural, and edges going out of a function depend on its calling context, so IPDs of nodes in the function must be computed *every time the function is called* (the IPDs change based on the earlier functions in the call-stack that called the particular function where the exception occurs). Moreover, in the case of recursive functions, the nodes must be replicated for every call. This is rather expensive. Ideally, the function's CFG should be built only once when *the function is compiled* and work intra-procedurally.

5.2. Synthetic exit nodes

In our approach, every function that may throw an unhandled exception and has an exception handler present earlier in the call-stack (which is assumed to be known at runtime through additional data structures in the system) has a special *synthetic exit node* (`SEN`), which is placed after the regular return node of the function in the CFG. Every exception-throwing node, whose exception will not be caught

within the function, has an outgoing edge to the SEN. In essence, the SEN is treated as the IPD for nodes whose actual IPDs lie outside of the function. By doing this, all cross-function edges are eliminated and the CFGs become intra-procedural. This allows the computation of the CFGs just once as compared to the inter-procedural case. For every exception-throwing instruction that has an associated handler, its context is maintained during dynamic information flow analysis until the handler is reached.

Thus, function calls and all potential exception-throwing instructions are represented as nodes with multiple edges (branches) and push a node on the *pc*-stack. However, a new node is *not* pushed on the *pc*-stack if the IPD of the current node is the same as the IPD on the top of the *pc*-stack or if the IPD of the current node is the SEN, as in this case the *real* IPD, which is outside of this method, should already be on the *pc*-stack.

In summary, these semantics emulate the effect of having cross-function edges. For illustration, consider the two functions f and g from before. The $\diamond$ at the end of g denotes its SEN. Note that there is an edge from `throw 9` to $\diamond$ because `throw 9` is not handled within g. $\square$ denotes the IPD of the function call `g()` and handler `catch(e)`.

```
function f() = {                          function g() = {
  l = 0;                                    if (h) {throw 9;}
  try { g(); } catch(e) { l = 1; }          return 7;
  □return l;                              }◇
}
```

When calling g, the current *pc* and IPD $(L, \square)$ are pushed on the *pc*-stack. When executing the condition `if (h)` a new node is not pushed again, but the top element is merely updated to $(H, \square)$ as its IPD is the SEN $\diamond$. If h is `false`, control reaches the `return` statement but with $pc = H$. At $\square$, *pc* is lowered to L, so f ends with the return value 0 and public label L. If h is `true`, control reaches the handler, which is in f and invokes it with the same *pc* as at the point of exception, i.e., H. Consequently, the generalized permissive-upgrade strategy marks the assignment in the catch block as partially-leaked and prevents the implicit information leak in this case.

5.3. Precision proofs

We show that the approach presented above where we pop the *pc*-stack at the IPD of a node is precise. In other words, the IPD of a node is the most precise node where the influence of *pc* ends, and the top of the *pc*-stack can be safely popped. Theorem 6 shows this result. For proving Theorem 6, we formally define a control flow graph, paths on the control flow graph, branch-points and post-dominators.

Definition 9 (Control flow graph)**.** A *control flow graph* is a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, n_s, n_e, \mathcal{L})$. $\mathcal{N}$ is the set of nodes. $\mathcal{E}$ is the set of control flow edges $(n_1, n_2) \in \mathcal{E}$, where $n_i \in \mathcal{N}$. (n_1, n_2) represents n_2 may immediately execute after n_1. The nodes $n_s, n_e \in \mathcal{N}$ are special nodes representing the start and the end point of $\mathcal{G}$, respectively. The function $\mathcal{L}$ maps the edges in $\mathcal{E}$ to labels.

Definition 10 (Path)**.** A *path* in a CFG $\mathcal{G}$ is a sequence of nodes $(n_1, n_2, \ldots, n_m)$ such that $(n_i, n_{i+1}) \in \mathcal{E}$, written as $n_1 \rightarrow_p n_m$. A node n that lies on the path $n_1 \rightarrow_p n_m$ is written as $n \in n_1 \rightarrow_p n_m$. The notation $n_1 < n_2$ with respect to two nodes n_1 and n_2 in a CFG $\mathcal{G}$ indicates that n_2 lies on a path $n_1 \rightarrow_p n_e$.

Definition 11 (Branch-point)**.** A *branch-point* b is a node in a CFG $\mathcal{G}$ that has more than one successor, i.e., outdegree$(b) > 1$.

Definition 12 (Post-dominator)**.** In a CFG $\mathcal{G}$, a node n_d is said to be the *post-dominator* of a node n if all paths from n to the end-node pass through n_d, i.e., $\forall p.n \rightarrow_p n_e \implies n_d \in p$. The notation $n_d \ \mathtt{pd}\ n$ indicates that n_d is a post-dominator of n.

Definition 13 (Immediate post-dominator)**.** A node i is the immediate post-dominator of the node n, denoted as $IPD(n)$, iff:

(1) $i \ \mathtt{pd}\ n$ and
(2) $\nexists n_o \in \mathcal{N}.((n_o \ \mathtt{pd}\ n) \wedge (n_o < i))$ or
$\forall n_o \in \mathcal{N}.((n_o \neq n) \implies (n_o \ \mathtt{pd}\ n \implies n_o \ \mathtt{pd}\ i))$.

Theorem 6 (Precision)**.** *Choosing any node other than the IPD to lower the pc-label will either give unsound results or be less precise.*

Proof. Consider a branch-point $b \in \mathcal{N}$ with IPD $IPD(b) = i \in \mathcal{N}$.

Assume that $(n \in \mathcal{N}) \neq i$ is the node where the context of the predicate expression in b is removed. Thus, either:

- $b < n < i$: Then, $\exists p.n \notin b \rightarrow_p n_e$. Thus, if n performs an action that should not have been performed in the context of the predicate expression in b, it might leak information about the predicate expression in b.
- $b < i < n$: Then, for any $n' \in \mathcal{N}$ such that $i < n' < n$ performing an operation that should not be performed in the context of b would be reported illicit as n' would be executed in the context of b.
  - If $n' \ \mathtt{pd}\ b$, then $\forall p.n' \in b \rightarrow_p n_e$. Hence, the statement n' executes irrespective of whether the branch at b is taken or not and hence, does not depend on the predicate expression in b, i.e., there is no implicit flow from the predicate expression in b to n', but still the program might be rejected.
  - If n' is a statement executing under the context of another branch-point b', such that $b' \ \mathtt{pd}\ b$, then as b' does not have any implicit flow from the predicate expression in b, any statement executing under the context of the predicate expression in b' should not be influenced by the context of the predicate expression in b. Hence, the program might be rejected even though there is no information leak.
- $i <> n$: $\forall p.n \notin i \rightarrow_p n_e$ or $\forall p.n \notin b \rightarrow_p n_e$, n will never be reached. Thus, the context of b shall not be removed until n_e such that $b < i < n_e$. Similar reasoning as in the second case with $n = n_e$.

Hence, the most precise node where one can safely remove the context of b is $n = IPD(b) = i$. □

We also show that our design decision to not push the SEN on the *pc*-stack if the IPD of a node is SEN is correct. In fact, the *actual* IPD of a node having SEN as its IPD is the node that is currently on the top of the stack. This result is shown in Theorem 7.

Theorem 7. *The actual IPD of a node having* SEN *as its IPD is the node that is currently on the top of the pc-stack.*

Proof. Assume two functions F and G given by the CFGs $\mathcal{G} = (\mathcal{N}, \mathcal{E}, n_s, n_e, \mathcal{L})$ and $\mathcal{G}' = (\mathcal{N}', \mathcal{E}', n'_s, n'_e, \mathcal{L}')$, respectively. The program's start and exit node are given by N_s and N_e, respectively. Consider a branch-point $b' \in \mathcal{N}'$ having SEN as its IPD. Assume a branch-point $b \in \mathcal{N}$ such that

$b < b' < (i = IPD(b))$, $(i \in \mathcal{N}) \neq \texttt{SEN}$, b is the last executed branch-point and top of the pc-stack contains i.

$\forall p.i \in b \rightarrow_p n_e$. Thus, i pd b' such that $b < b' < i$.

T.S. $\nexists n \in b' \rightarrow_p i \mid (n \texttt{ pd } b') \wedge (b' < n < i)$.

Proof by contradiction: Assume $\exists n.n$ pd $b' \mid b' < n < i$. Then the node n either lies in the function F or G or in another function H given by the CFG $\mathcal{G}'' = (\mathcal{N}'', \mathcal{E}'', n_s'', n_e'', \mathcal{L}'')$, such that F calls H and H calls G.

- $n \in \mathcal{N}'$: As $IPD(b') = \texttt{SEN}$ and SEN is the last node in a function($\mathcal{G}'$), $\exists p.n \notin b \rightarrow_p n_e$.
- $n \in \mathcal{N}''$: As $\forall p.n \in b' \rightarrow_p N_e$ and $G() < b'$, thus, $\forall p.n \in G() \rightarrow_p N_e$, which means $IPD(G()) \neq \texttt{SEN}$. Hence, the top of the pc-stack then would have $IPD(G()) = (n'' \in \mathcal{N}'') \leqslant n$ and not i.
- $n \in \mathcal{N}$: When the call to G or any other function H is made, it would push i, IPD of the branch-point on the top of the pc-stack.

Thus, $\nexists n \in b' \rightarrow_p i \mid (n \texttt{ pd } b') \wedge (b' < n < i)$. Hence, the top of the *pc*-stack, i is the actual IPD of any node b' having SEN as its intra-procedural IPD. □

6. Formal model

We consider a language with unstructured control flow and exceptions, and give it an operational semantics with IFC using the generalized permissive-upgrade check. The language is an extension of that in Fig. 1 with unstructured control flow constructs like `break`, `continue`, `return` and exceptions. A program is a collection of functions (without parameter-passing). The control flow analysis is performed on a function before it is executed and is abstractly represented as a CFG. This involves a translation from the language of Fig. 1 extended with unstructured control flow and exceptions into a formal language of CFGs, the syntax of whose nodes is shown in Fig. 9.

A program in the CFG language is modeled as a huge control flow graph ($\mathcal{G}$). IPDs are computed using the algorithm by Lengauer and Tarjan [44] when the CFG is created. For new functions added at runtime, the CFG is constructed only once. For a non-branching node $\iota \in \mathcal{G}$, $Succ(\iota)$ denotes ι's unique successor. For a conditional branching node ι, $Left(\iota)$ and $Right(\iota)$ denote successors when the condition is `true` and `false`, respectively.

The command `if` e `then` c_1 `else` c_2 is represented as the node `branch` e with $Left(\mathcal{G}, \iota) = c_1$ and $Right(\mathcal{G}, \iota) = c_2$. Similarly, `while` e `do` c is represented as `branch` e with $Left(\iota) = c$ and $Right(\iota)$ being the command following `while` in the program. A `jmp` node is added as $Succ(c)$ with the successor $Succ(\iota)$ set to the branch instruction to jump to the predicate. A `jmp` node in the CFG also corresponds to `break` and `continue` with $Succ(\iota)$ pointing to the next node in the CFG according to the operation. It is also assumed that a function always ends with a `return` statement and thus a CFG

$$\iota := \texttt{end} \mid \texttt{x} := e \mid \texttt{branch}\ e \mid \texttt{jmp} \mid \texttt{return} \mid \texttt{throw}\ e \mid \texttt{catch}\ \texttt{x} \mid \texttt{SEN}$$

$$\rho := [] \mid (pc, \iota) \rhd \rho$$

Fig. 9. Language syntax (extends Fig. 1).

normally ends with the `return` node. (Multiple return statements in a function are represented using a single `return` node.) When ι is a `return` node, $Succ(\iota)$ is the node to return to in the previous CFG. Here "previous" refers to the previous function in the call stack (the caller). The return value is saved in a global variable that can be accessed by the program later on. Every node in the program's CFG is uniquely identifiable.

In general, every function has an associated exception table that maps each potentially exception-throwing instruction in the function to the appropriate exception handler within the function. This is represented by adding a *Right* edge in the CFG from the instruction's node to the handler's node. `throw` has only one outgoing edge. It is conservatively assumed that any unknown code may throw an exception, so function call is exception-throwing for this purpose. If a function contains unhandled exceptions, the corresponding edges in the CFG point to the `SEN` of the CFG. The `SEN` is only created if one of the previous functions in the call-stack has an appropriate exception handler for the unhandled exceptions in the current function. When an `SEN` node is created, an edge is added from the `SEN` of the CFG to a node in the previous CFG, which is either the `catch` node or the `SEN` of that CFG. *Succ* denotes one of these edges. If there are no appropriate handlers in the call-stack, the exception-throwing nodes have an edge to the *end* node of the program CFG. For simplicity of exposition, it is assumed here that all exceptions belong to a single class – for different types of exceptions, the exception class would also be matched for determining the appropriate exception handler.

Program configurations for commands (nodes) are represented as $\langle \sigma, \iota, \rho \rangle$, where σ represents the memory store as before, ι represents the currently executing node, and ρ is the *pc*-stack. The configuration for expressions is the same as before: $\langle \sigma, e \rangle$.

Each entry of the *pc*-stack ρ is a pair (ℓ, ι), where ℓ is a security label and ι is a node in the CFG. When a new control context is entered, the new *pc*-label, which is a join of the current context label and the existing *pc*-label (the label on the top of the stack), is pushed together with the IPD ι of the entry point of the control context. This IPD ι uniquely identifies where the control of the context ends. In the semantics, the meta-function *isIPD* pops the stack. It takes the current instruction and the current *pc*-stack, and returns a new *pc*-stack. $!\rho$ returns the top frame of the *pc*-stack. $\Gamma(!\rho)$ returns the current context label, also represented as *pc* in the semantics.

$$isIPD(\iota, \rho) := \begin{cases} \rho.pop() & \text{if } !\rho = (\_, \iota) \\ \rho & \text{otherwise} \end{cases}$$

As explained in Section 5.2, a new node (ℓ, ι) is pushed onto ρ only when ι (the IPD) differs from the corresponding entry on the top of the stack or it is not `SEN` (Theorem 7). Otherwise, ℓ is joined with the label on the top of the stack. This is formally represented using the function $\rho.push(\ell, \iota)$.

The rules in Fig. 10 define small-step semantics of CFGs. The rules for expressions are the same as in Fig. 2. The rules are informally explained above. The soundness of the analysis including unstructured control flow and exceptions is proved below.

To state the theorem formally, the equivalence of different data structures with respect to the adversary needs to be defined. We reuse value and memory equivalences from Definitions 7 and 8. We prove termination-insensitive non-interference using the generalized permissive-upgrade strategy, for which we formalize some additional definitions and define auxiliary lemmas. The detailed proofs, definitions and other lemmas can be found in Appendix B.

$$\text{assn}\ \frac{\iota = (\mathtt{x} := e) \quad \langle \sigma, e\rangle \Downarrow \mathtt{n}^m \quad l = \Gamma(\sigma(\mathtt{x})) \quad l = \ell_{\mathtt{x}} \vee l = \ell_{\mathtt{x}}^{\star} \quad pc = \Gamma(!\rho) \quad k = \{^{pc \sqcup m,\ pc \sqsubseteq \ell_{\mathtt{x}}}_{((pc \sqcup m) \sqcap \ell_{\mathtt{x}})^{\star},\ pc \not\sqsubseteq \ell_{\mathtt{x}}}\} \quad \sigma' = \sigma[\mathtt{x} \mapsto \mathtt{n}^k] \quad \iota' = Succ(\iota) \quad \rho' = isIPD(\iota', \rho)}{\langle \sigma, \iota, \rho\rangle \rightarrow \langle \sigma', \iota', \rho'\rangle}$$

$$\text{branch}\ \frac{\iota = \mathtt{branch}\ e \quad \langle \sigma, e\rangle \Downarrow \mathtt{b}^{\ell} \quad \iota' = \{^{Left(\iota),\ \ if\ \mathtt{b} = \mathtt{true}}_{Right(\iota),\ otherwise}\} \quad \rho'' = \rho.push(\ell, IPD(\iota)) \quad \rho' = isIPD(\iota', \rho'')}{\langle \sigma, \iota, \rho\rangle \rightarrow \langle \sigma, \iota', \rho'\rangle}$$

$$\text{jmp, ret, sen}\ \frac{\iota = \mathtt{jmp}\ or\ \mathtt{return}\ or\ \mathtt{SEN} \quad \iota' = Succ(\iota) \quad \rho' = isIPD(\iota', \rho)}{\langle \sigma, \iota, \rho\rangle \rightarrow \langle \sigma, \iota', \rho'\rangle}$$

$$\text{throw}\ \frac{\iota = \mathtt{throw}\ e \quad \langle \sigma, e\rangle \Downarrow \mathtt{n}^k \quad pc = \Gamma(!\rho) \quad excValue = \mathtt{n}^{(k \sqcup pc)} \quad \iota' = Succ(\iota) \quad \rho' = isIPD(\iota', \rho)}{\langle \sigma, \iota, \rho\rangle \rightarrow \langle \sigma, \iota', \rho'\rangle}$$

$$\text{catch}\ \frac{\iota = \mathtt{catch}\ \mathtt{x} \quad excValue = \mathtt{n}^m \quad l = \Gamma(\sigma(\mathtt{x})) \quad l = \ell_{\mathtt{x}} \vee l = \ell_{\mathtt{x}}^{\star} \quad pc = \Gamma(!\rho) \quad k = \{^{pc \sqcup m,\ pc \sqsubseteq \ell_{\mathtt{x}}}_{((pc \sqcup m) \sqcap \ell_{\mathtt{x}})^{\star},\ pc \not\sqsubseteq \ell_{\mathtt{x}}}\} \quad \sigma' = \sigma[\mathtt{x} \mapsto \mathtt{n}^k] \quad \iota' = Succ(\iota) \quad \rho' = isIPD(\iota', \rho)}{\langle \sigma, \iota, \rho\rangle \rightarrow \langle \sigma', \iota', \rho'\rangle}$$

$$\text{end}\ \frac{\iota = \mathtt{end}}{\langle \sigma, \iota, \rho\rangle \rightarrow \_}$$

Fig. 10. Semantics.

Definition 14 (*pc*-stack equivalence)**.** For two *pc*-stacks ρ_1, ρ_2, $\rho_1 \sim_\ell \rho_2$ iff the corresponding nodes of ρ_1 and ρ_2 having labels less than or equal to ℓ are equal.

Definition 15 (State equivalence)**.** Two states $s_1 = \langle \sigma_1, \iota_1, \rho_1\rangle$ and $s_2 = \langle \sigma_2, \iota_2, \rho_2\rangle$ are equivalent, written as $s_1 \sim_\ell s_2$, iff $\sigma_1 \sim_\ell \sigma_2$, $\iota_1 = \iota_2$, and $\rho_1 \sim_\ell \rho_2$.

Lemma 8 (Confinement lemma)**.** *If* $\langle \sigma, \iota, \rho\rangle \rightarrow \langle \sigma', \iota', \rho'\rangle$ *and* $\Gamma(!\rho) \not\sqsubseteq \ell$*, then* $\sigma \sim_\ell \sigma'$*, and* $\rho \sim_\ell \rho'$.

Theorem 8. *Suppose:*

(1) $\langle \sigma_1, \iota_1, \rho_1\rangle \sim_\ell \langle \sigma_2, \iota_2, \rho_2\rangle$
(2) $\langle \sigma_1, \iota_1, \rho_1\rangle \rightarrow^* \langle \sigma_1', \mathit{end}, []\rangle$
(3) $\langle \sigma_2, \iota_2, \rho_2\rangle \rightarrow^* \langle \sigma_2', \mathit{end}, []\rangle$

Then, $\sigma_1' \sim_\ell \sigma_2'$.

7. Related work

7.1. Permissive IFC

Secure multi-execution [23] is another approach for enforcing non-interference at runtime, where one executes multiple copies of the program with different values of sensitive data. Conceptually, the same code is executed once for each security level (like low and high) with a few constraints. High inputs in the low execution are replaced by default values, i.e., the low execution of the program does not see the actual value of the high data but a pre-determined default value. Additionally, outputs on an ℓ-labeled channel are permitted only in the ℓ-level execution of the program. This ensures noninterference and precision (the semantics of a secure program are not altered). However, executing a program multiple times can be prohibitive for a security lattice with many levels [2,9]. The runtime overhead incurred can be reduced if the executions are done in parallel but this requires more hardware resources. In addition to this, secure multi-execution makes declassification complicated as it requires synchronization between different executions [52].

To tackle the issue of permissiveness with the no-sensitive-upgrade strategy and the permissive-upgrade strategy, Austin and Flanagan proposed the idea of faceted execution [9], which is the most permissive of the three. Faceted execution simulates multiple executions simultaneously within a single runtime. They introduce the concept of faceted values that are pairs of values, one each for low and high observers. When branching on a faceted value, multiple executions are simulated for the two values in the facet. To achieve better precision and security guarantees, researchers have proposed techniques that combine faceted execution and secure multi-execution on an as-needed basis [3,58]. However, the runtime overheads of faceted execution is also prohibitive for multiple security levels [2]. This paper, thus, considers only the permissive-upgrade strategy, which is more permissive than the no-sensitive-upgrade strategy and much less performance-intensive than faceted execution.

Birgisson *et al.* [15] describe a testing-based approach that adds variable upgrade annotations to avoid halting on the NSU check in an implementation of dynamic IFC for JavaScript [38]. A different way of handling the problem of implicit flows through flow-sensitive labels is to assign a (fixed) label to each label; this approach has been examined by Buiras *et al.* in the context of a language with a dedicated monad for tracking information flows [16]. The precise connection between that approach and permissive-upgrade remains unclear, although Buiras *et al.* sketch a technique related to permissive-upgrade in their system, while also noting that generalizing permissive-upgrade to arbitrary lattices is non-obvious. This paper confirms the latter and shows how it can be done.

7.2. IFC with error handling

Much work on error handling for IFC has been in the context of static analysis [6,48]. Work on IFC for dynamic languages has mostly ignored exceptions and other unstructured control flow constructs [7,8,17,24,33]. Just et al. [42] present dynamic IFC for JavaScript bytecode with static analysis to determine implicit flows precisely but ignore implicit flows due to exceptions. Hedin and Sabelfeld propose a dynamic IFC approach for a language modeling the core features of JavaScript [38] but ignore unstructured control flow constructs like break, continue and return-in-the-middle for functions. For handling exceptions, they introduce annotations and an additional class of labels. An extension introduces similar annotations to deal with unstructured control flow [37]. These labels are more restrictive than needed, e.g., the code indicated by dots in the snippet: `l = 1; while(1){...if(h){break; }; l = 0; break; },`

is executed irrespective of the condition h in the first iteration, and thus there is no need to raise the *pc* before checking that condition.

Stefan et al. [60] use special constructs to catch exceptions and convert them into normal values for enforcing IFC in Haskell. Hritcu et al. [39] propose the propagation of exception values as NaNs, and delay normal and IFC exceptions, changing the semantics of exception handling. Austin et al. [10] extend faceted execution to work with exceptions by including additional constructs to raise and catch exceptions. They use big-step semantics to define exception handling while our formalism uses small-step semantics. Our formalism also supports unstructured control flow, which is not considered in their semantics.

8. Conclusion

We have presented the design of a sound improvement and generalization of the permissive-upgrade strategy for dynamic information flow analysis. The development improves the original strategy's permissiveness by relaxing the rules for handling partially-leaked data while retaining soundness. Additionally, the original strategy's enforcement was limited to a two-point security lattice; we generalize that to an arbitrary lattice.

We also present a sound approach for improving the precision of runtime information flow analysis when handling unstructured control flow and exceptions. Existing approaches to handle these constructs are more conservative and often require additional annotations by the developer. In contrast, the methodology presented here performs a sound and precise runtime information flow analysis using post-dominator analysis to handle these features without requiring any additional annotations from the developer.

Appendix A. Proofs for improved and generalized permissive upgrade

A.1. Proofs for improved permissive upgrade strategy

Lemma 1 (Expression evaluation). *If* $\langle \sigma_1, e\rangle \Downarrow \mathtt{n}_1^{k_1}$ *and* $\langle \sigma_2, e\rangle \Downarrow \mathtt{n}_2^{k_2}$ *and* $\sigma_1 \sim \sigma_2$, *then* $\mathtt{n}_1^{k_1} \sim \mathtt{n}_2^{k_2}$.

Proof. Induction on the derivation and case analysis on the last expression rule.

(1) CONST: $\mathtt{n}_1 = \mathtt{n}_2 = \mathtt{n}$ and $k_1 = k_2 = \bot$.

(2) VAR: As $\sigma_1 \sim \sigma_2$, $\forall \mathtt{x}.\sigma_1(\mathtt{x}) = \mathtt{n}_1^{k_1} \sim \sigma_2(\mathtt{x}) = \mathtt{n}_2^{k_2}$.

(3) OPER: IH1: If $\langle \sigma_1, e_1\rangle \Downarrow \mathtt{n}_1^{\prime k_1'}$, $\langle \sigma_2, e_1\rangle \Downarrow \mathtt{n}_2^{\prime k_2'}$, $\sigma_1 \sim \sigma_2$, then $\mathtt{n}_1^{\prime k_1'} \sim \mathtt{n}_2^{\prime k_2'}$.
IH2: If $\langle \sigma_1, e_2\rangle \Downarrow \mathtt{n}_1^{\prime\prime k_1''}$, $\langle \sigma_2, e_2\rangle \Downarrow \mathtt{n}_2^{\prime\prime k_2''}$, $\sigma_1 \sim \sigma_2$, then $\mathtt{n}_1^{\prime\prime k_1''} \sim \mathtt{n}_2^{\prime\prime k_2''}$.
T.S. $\mathtt{n}_1^{k_1} \sim \mathtt{n}_2^{k_2}$, where $\mathtt{n}_1 = \mathtt{n}_1' \odot \mathtt{n}_1''$, $\mathtt{n}_2 = \mathtt{n}_2' \odot \mathtt{n}_2''$ and $k_1 = k_1' \sqcup k_1''$, $k_2 = k_2' \sqcup k_2''$.
As $\sigma_1 \sim \sigma_2$, from IH1 and IH2, $\mathtt{n}_1^{\prime k_1'} \sim \mathtt{n}_2^{\prime k_2'}$ and $\mathtt{n}_1^{\prime\prime k_1''} \sim \mathtt{n}_2^{\prime\prime k_2''}$.
Proof by case analysis on low-equivalence definition (Definition 3) for $\mathtt{n}_1^{\prime k_1'} \sim \mathtt{n}_2^{\prime k_2'}$ followed by case analysis on low-equivalence definition for $\mathtt{n}_1^{\prime\prime k_1''} \sim \mathtt{n}_2^{\prime\prime k_2''}$.

- $\mathtt{n}_1' = \mathtt{n}_2'$ and $k_1' = k_2' = L$:
  - $\mathtt{n}_1'' = \mathtt{n}_2''$ and $k_1'' = k_2'' = L$: $\mathtt{n}_1 = \mathtt{n}_2$ and $k_1 = k_2 = L$

  - $k_1'' = k_2'' = H$: $k_1 = k_2 = H$
  - $k_1'' = P$ or $k_2'' = P$: $k_1 = P$ or $k_2 = P$
- $k_1' = k_2' = H$:
  - $\mathtt{n}_1'' = \mathtt{n}_2''$ and $k_1'' = k_2'' = L$: $k_1 = k_2 = H$
  - $k_1'' = k_2'' = H$: $k_1 = k_2 = H$
  - $k_1'' = P$ or $k_2'' = P$: $k_1 = H$ and $k_2 = H$
- $k_1' = P$ or $k_2' = P$:
  - $\mathtt{n}_1'' = \mathtt{n}_2''$ and $k_1'' = k_2'' = L$: $k_1 = P$ or $k_2 = P$
  - $k_1'' = k_2'' = H$: $k_1 = k_2 = H$
  - $k_1'' = P$ or $k_2'' = P$: $k_1 = P$ and/or $k_2 = P$ □

Lemma 2 (Evolution)**.** *If* $pc = H$ *and* $\langle \sigma, c \rangle \Downarrow_{pc} \sigma'$*, then* $\forall \mathtt{x}.\Gamma(\sigma(\mathtt{x})) = P \implies \Gamma(\sigma'(\mathtt{x})) = P$.

Proof. Proof by induction on the derivation rules and case analysis on the last rule.

- SKIP,WHILE-F: $\sigma = \sigma'$
- ASSN-PUS: If $pc = H$ and $l = P$, then $k = P$. All other $\sigma(\mathtt{x})$ remain unchanged.
- SEQ:
  IH1: If $pc = H$ and $\langle \sigma, c_1 \rangle \Downarrow_{pc} \sigma''$, then $\forall \mathtt{x}.\Gamma(\sigma(\mathtt{x})) = P \implies \Gamma(\sigma''(\mathtt{x})) = P$.
  IH2: If $pc = H$ and $\langle \sigma'', c_2 \rangle \Downarrow_{pc} \sigma'$, then $\forall \mathtt{x}.\Gamma(\sigma''(\mathtt{x})) = P \implies \Gamma(\sigma'(\mathtt{x})) = P$.
  From IH1 and IH2, if $pc = H$ and $\langle \sigma, c_1; c_2 \rangle \Downarrow_{pc} \sigma'$, then $\forall \mathtt{x}.\Gamma(\sigma(\mathtt{x})) = P \implies \Gamma(\sigma'(\mathtt{x})) = P$.
- IF-ELSE:
  IH: If $pc = H$ and $\langle \sigma, c_i \rangle \Downarrow_{pc \sqcup \ell} \sigma'$, then $\forall \mathtt{x}.\Gamma(\sigma(\mathtt{x})) = P \implies \Gamma(\sigma'(\mathtt{x})) = P$.
  As $H \sqcup \ell = H$, from IH.
- WHILE-T: Similar to SEQ and IF-ELSE □

Lemma 3 (Confinement for improved permissive-upgrade with a two-point lattice)**.** *If* $pc = H$ *and* $\langle \sigma, c \rangle \Downarrow_{pc} \sigma'$*, then* $\sigma \sim \sigma'$.

Proof. Proof by induction on the derivation rules and case analysis on the last step.

- SKIP,WHILE-F: $\sigma = \sigma'$
- ASSN-PUS: If $l = L$, then $k = P$ else if $l = H$, then $k = H$, else if $l = P$, then $k = P$. Thus, $\sigma \sim \sigma'$
- SEQ: IH1: If $pc = H$ and $\langle \sigma, c_1 \rangle \Downarrow_{pc} \sigma''$, then $\sigma \sim \sigma''$ and
  IH2: if $pc = H$ and $\langle \sigma'', c_2 \rangle \Downarrow_{pc} \sigma'$, then $\sigma'' \sim \sigma'$.
  From Lemma 2, $\forall \mathtt{x}.\Gamma(\sigma(\mathtt{x})) = P \implies \Gamma(\sigma''(\mathtt{x})) = P$ and $\forall \mathtt{x}.\Gamma(\sigma''(\mathtt{x})) = P \implies \Gamma(\sigma'(\mathtt{x})) = P$.
  From definition, $\forall \mathtt{x}$ either:
  $\sigma(\mathtt{x}) = \sigma''(\mathtt{x})$ and $\Gamma(\sigma(\mathtt{x})) = \Gamma(\sigma''(\mathtt{x})) = L$: From IH2, either $\sigma''(\mathtt{x}) = \sigma'(\mathtt{x})$ and $\Gamma(\sigma''(\mathtt{x})) = \Gamma(\sigma'(\mathtt{x})) = L$ or $\Gamma(\sigma'(\mathtt{x})) = P$
  or $\Gamma(\sigma(\mathtt{x})) = \Gamma(\sigma''(\mathtt{x})) = H$: From IH2, $\Gamma(\sigma''(\mathtt{x})) = \Gamma(\sigma'(\mathtt{x})) = H$
  or either $\Gamma(\sigma(\mathtt{x})) = P$ or $\Gamma(\sigma''(\mathtt{x})) = P$: If $\Gamma(\sigma(\mathtt{x}) = P$, then from Lemma 2, $\Gamma(\sigma''(\mathtt{x}) = P$.
  Hence, $\Gamma(\sigma'(\mathtt{x})) = P$.

- IF-ELSE: IH: If $pc = H$ and $\langle \sigma, c_i \rangle \Downarrow_{pc \sqcup \ell} \sigma'$, then $\sigma \sim \sigma'$. If $pc = H$, then $H \sqcup \ell = H$. Thus, from IH.
- WHILE-T: Similar to IF-ELSE and SEQ. □

Theorem 1 (TINI for improved permissive-upgrade with a two-point lattice)**.** *With the assignment rule* ASSN-PUS *and the modified syntax of Fig. 4, if* $\sigma_1 \sim \sigma_2$ *and* $\langle \sigma_1, c \rangle \Downarrow_{pc} \sigma_1'$ *and* $\langle \sigma_2, c \rangle \Downarrow_{pc} \sigma_2'$*, then* $\sigma_1' \sim \sigma_2'$.

Proof. Proof by induction on the derivation rules and case analysis on the last step.

- SKIP,WHILE-F: $\sigma_1' = \sigma_1 \sim \sigma_2 = \sigma_2'$
- ASSN-PUS: From Lemma 1, $\mathtt{n}_1^{m_1} \sim \mathtt{n}_2^{m_2}$. If $pc = L$, then $k = m$. If $pc = H$ and $l = H$, then $k_1 = k_2 = H$. If $pc = H$ and $l = L$, then $k_1 = k_2 = P$. Hence, $\sigma_1' \sim \sigma_2'$.
- SEQ: IH1: If $\sigma_1 \sim \sigma_2$ and $\langle \sigma_1, c_1 \rangle \Downarrow_{pc} \sigma_1'$, and $\langle \sigma_2, c_1 \rangle \Downarrow_{pc} \sigma_2'$, then $\sigma_1' \sim \sigma_2'$ and
  IH2: If $\sigma_1' \sim \sigma_2'$ and $\langle \sigma_1', c_2 \rangle \Downarrow_{pc} \sigma_1''$, and $\langle \sigma_2', c_2 \rangle \Downarrow_{pc} \sigma_2''$, then $\sigma_1'' \sim \sigma_2''$.
  From IH1 and IH2, $\sigma_1'' \sim \sigma_2''$
- IF-ELSE: IH: If $\sigma_1 \sim \sigma_2$ and $\langle \sigma_1, c_i \rangle \Downarrow_{pc \sqcup \ell_1} \sigma_1'$, and $\langle \sigma_2, c_j \rangle \Downarrow_{pc \sqcup \ell_2} \sigma_2'$, and $\ell_1 = \ell_2$, and $c_i = c_j$ then $\sigma_1' \sim \sigma_2'$. From Lemma 1, $\mathtt{n}_1^{l_1} \sim \mathtt{n}_2^{l_2}$. Thus, either $\ell_1 = \ell_2 = L$ or $\ell_1 = \ell_2 = H$. If $\ell_1 = \ell_2 = L$, then $\mathtt{n}_1 = \mathtt{n}_2$. Thus, $c_i = c_j$ and hence, from IH $\sigma_1' \sim \sigma_2'$.
  If $\ell_1 = \ell_2 = H$, then $pc \sqcup H = H$. From Lemma 3, $\sigma_1 \sim \sigma_1'$ and $\sigma_2 \sim \sigma_2'$, and $\sigma_1 \sim \sigma_2$.
  T.S. $\sigma_1' \sim \sigma_2'$, i.e., $\forall \mathtt{x}. \sigma_1'(\mathtt{x}) \sim \sigma_2'(\mathtt{x})$.
  Let $\sigma_1(\mathtt{x}) = \mathtt{n}_1^{k_1}$ and $\sigma_2(\mathtt{x}) = \mathtt{n}_2^{k_2}$ and $\sigma_1'(\mathtt{x}) = \mathtt{n}_1'^{k_1'}$ and $\sigma_2'(\mathtt{x}) = \mathtt{n}_2'^{k_2'}$. Case analysis on the definition of equivalence:
  - $\mathtt{n}_1 = \mathtt{n}_2$ and $k_1 = k_2 = L$: Either $\mathtt{n}_1' = \mathtt{n}_1$ and $k_1' = k_1 = L$ and $\mathtt{n}_2' = \mathtt{n}_2$ and $k_2' = k_2 = L$ or $k_1' = P$ or $k_2' = P$
  - $k_1 = k_2 = H$: $k_1' = k_1 = H$ and $k_2' = k_2 = H$
  - $k_1 = P$ or $k_2 = P$: From Lemma 2, $k_1' = P$ or $k_2' = P$
- WHILE-T: Similar to IF-ELSE and SEQ. □

A.2. Examples for equivalence definition

Consider the following notations for the examples:
$l, m, h, l^\star$ represent any variable with label $L, M, H, L^\star$, respectively, such that $L \sqsubseteq M \sqsubseteq H$.
An ℓ-level adversary is assumed. $\boldsymbol{\ell}$ represents the labels that are above the level of the attacker.
Table 2 shows example programs for the transition from low-equivalent values to low-equivalent values. First column and first row of the table represents all the possible ways in which two values can be low-equivalent (from Definition 7).

A.3. Proofs and results for generalized permissive upgrade for arbitrary lattices

Lemma 4 (Expression evaluation lemma)**.**
If $\sigma_1 \sim_\ell \sigma_2$,
$\langle \sigma_1, e \rangle \Downarrow \mathtt{n}_1^{k_1}$,
$\langle \sigma_2, e \rangle \Downarrow \mathtt{n}_2^{k_2}$,
then $\mathtt{n}_1^{k_1} \sim_\ell \mathtt{n}_2^{k_2}$.

Table 2
Examples for all possible transitions of low-equivalent to low-equivalent values

	ℓ, ℓ	$\ell_1^\star, \ell_2$	$\ell_1, \ell_2^\star$	$\ell_1^\star, \ell_2^\star$	$\boldsymbol{\ell_1, \ell_2}$	$\ell_1^\star, \boldsymbol{\ell_2}$	$\boldsymbol{\ell_1}, \ell_2^\star$

ℓ, ℓ	–	`if(h)`
		`x1 = l`
			`if(h)`
			`x1 = l`
				`if(h)`
				`x1 = l`
				`else`
				`x1 = l`
					`x1 = h`	`x1 = m`
						`if(h)`
						`x1 = 4`
						`if(m)`
						`x1 = l⋆`
							`x1 = m`
							`if(h)`
							`x1 = 4`
							`if(m)`
							`x1 = l⋆`

$\ell_1^\star, \ell_2$	`x1 = l`	–	`x1 = l`
			`if(h)`
			`x1 = l`
				`if(h)`
				`x1 = l`
					`x1 = h`	`x1 = m`
						`if (h)`
						`x1 = l`
						`if(m)`
						`x1 = l⋆`
							`x1 = m`
							`if (h)`
							`x1 = l`
							`if(m)`
							`x1 = l⋆`

$\ell_1, \ell_2^\star$	`x1 = l`	`x1 = l`
		`if(h)`
		`x1 = l`
			–	`if(h)`
				`x1 = l`
					`x1 = h`	`x1 = m`
						`if (h)`
						`x1 = l`
						`if(m)`
						`x1 = l⋆`
							`x1 = m`
							`if (h)`
							`x1 = l`
							`if(m)`
							`x1 = l⋆`

$\ell_1^\star, \ell_2^\star$	`x1 = l`	`x1 = l`
		`if (h)`
		`x1 = l`
			`x1 = l`
			`if (h)`
			`x1 = l`
				–	`x1 = h`	`x1 = m`
						`if (h)`
						`x1 = l`
						`if (m)`
						`x1 = l⋆`
							`x1 = m`
							`if (h)`
							`x1 = l`
							`if (m)`
							`x1 = l⋆`

$\boldsymbol{\ell_1, \ell_2}$	`x1 = l`	`x1 = l`
		`if (h)`
		`x1 = l`
			`x1 = l`
			`if (h)`
			`x1 = l`
				`x1 = l`
				`if (h)`
				`x1 = l`
				`else`
				`x1 = l`
					–	`x1 = m`
						`if (h)`
						`x1 = l`
						`if(m)`
						`x1 = l⋆`
							`x1 = m`
							`if (h)`
							`x1 = l`
							`if (m)`
							`x1 = l⋆`

$\ell_1^\star, \boldsymbol{\ell_2}$	`x1 = l`	`x1 = l`
		`if (h)`
		`x1 = l`
			`x1 = l`
			`if (h)`
			`x1 = l`
				`x1 = l`
				`if (h)`
				`x1 = l`
				`else`
				`x1 = l`
					`x1 = h`	–	`x1 = m`
							`if (h)`
							`x1 = l`
							`if (m)`
							`x1 = l⋆`

$\boldsymbol{\ell_1}, \ell_2^\star$	`x1 = l`	`x1 = l`
		`if (h)`
		`x1 = l`
			`x1 = l`
			`if (h)`
			`x1 = l`
				`x1 = l`
				`if (h)`
				`x1 = l`
				`else`
				`x1 = l`
					`x1 = h`	`x1 = m`
						`if (h)`
						`x1 = l`
						`if(m)`
						`x1 = l⋆`
							–

Proof. Proof by induction on the derivation and case analysis on the last expression rule.

(1) CONST: $\mathtt{n}_1 = \mathtt{n}_2 = \mathtt{n}$ and $k_1 = k_2 = \bot$.
(2) VAR: As $\sigma_1 \sim_\ell \sigma_2$, $\forall \mathtt{x}.\sigma_1(\mathtt{x}) = \mathtt{n}_1^{k_1} \sim_\ell \sigma_2(\mathtt{x}) = \mathtt{n}_2^{k_2}$.
(3) OPER: IH1: If $\langle \sigma_1, e_1 \rangle \Downarrow \mathtt{n}_1^{\prime k_1'}$, $\langle \sigma_2, e_1 \rangle \Downarrow \mathtt{n}_2^{\prime k_2'}$, $\sigma_1 \sim_\ell \sigma_2$, then $\mathtt{n}_1^{\prime k_1'} \sim_\ell \mathtt{n}_2^{\prime k_2'}$.
IH2: If $\langle \sigma_1, e_2 \rangle \Downarrow \mathtt{n}_1^{\prime\prime k_1''}$, $\langle \sigma_2, e_2 \rangle \Downarrow \mathtt{n}_2^{\prime\prime k_2''}$, $\sigma_1 \sim_\ell \sigma_2$, then $\mathtt{n}_1^{\prime\prime k_1''} \sim_\ell \mathtt{n}_2^{\prime\prime k_2''}$.
T.S. $\mathtt{n}_1^{k_1} \sim_\ell \mathtt{n}_2^{k_2}$, where $\mathtt{n}_1 = \mathtt{n}_1' \odot \mathtt{n}_1''$, $\mathtt{n}_2 = \mathtt{n}_2' \odot \mathtt{n}_2''$ and $k_1 = k_1' \sqcup k_1''$, $k_2 = k_2' \sqcup k_2''$.
As $\sigma_1 \sim_\ell \sigma_2$, from IH1 and IH2, $\mathtt{n}_1^{\prime k_1'} \sim_\ell \mathtt{n}_2^{\prime k_2'}$ and $\mathtt{n}_1^{\prime\prime k_1''} \sim_\ell \mathtt{n}_2^{\prime\prime k_2''}$.
Proof by case analysis on low-equivalence definition for $\mathtt{n}_1^{\prime k_1'} \sim_\ell \mathtt{n}_2^{\prime k_2'}$ followed by case analysis on low-equivalence definition for $\mathtt{n}_1^{\prime\prime k_1''} \sim_\ell \mathtt{n}_2^{\prime\prime k_2''}$. □

Lemma 5 ($\star$-preservation lemma)**.** $\forall x.$*If* $\langle \sigma, c\rangle \Downarrow_{pc} \sigma'$, $\Gamma(\sigma(x)) = \ell^\star$ *and* $pc \not\sqsubseteq \ell$, *then* $\Gamma(\sigma'(x)) = \ell'^\star \wedge \ell' \sqsubseteq \ell$

Proof. Proof by induction on the derivation and case analysis on the last rule.

(1) SKIP : $\sigma = \sigma'$.
(2) ASSN-N: As $pc \not\sqsubseteq \ell$, these cases do not apply.
(3) ASSN-S: From the premises, for x in statement c, $\Gamma(\sigma'(x)) = ((pc \sqcup m) \sqcap \ell)^\star = \ell'$. Thus, $\ell' \sqsubseteq \ell$.
For any other y, $\sigma(y) = \sigma'(y)$. Thus, $\ell' = \ell$.
(4) SEQ : IH1 : $\forall x.$If $\langle \sigma, c\rangle \Downarrow_{pc} \sigma''$, $\Gamma(\sigma(x)) = \ell^\star$ and $pc \not\sqsubseteq \ell$, then $\Gamma(\sigma''(x)) = \ell''^\star \wedge \ell'' \sqsubseteq \ell$
IH2 : $\forall x.$If $\langle \sigma'', c\rangle \Downarrow_{pc} \sigma'$, $\Gamma(\sigma''(x)) = \ell''^\star$ and $pc \not\sqsubseteq \ell''$, then $\Gamma(\sigma'(x)) = \ell'^\star \wedge \ell' \sqsubseteq \ell''$
Thus, from IH1 and IH2, $\Gamma(\sigma'(x)) = \ell'^\star \wedge \ell' \sqsubseteq \ell$.
(5) IF-ELSE: Let $k = \ell''$.
IH: $\forall x.$If $\langle \sigma, c\rangle \Downarrow_{pc\sqcup\ell''} \sigma'$, $\Gamma(\sigma(x)) = \ell^\star$ and $pc \sqcup \ell'' \not\sqsubseteq \ell$, then $\Gamma(\sigma'(x)) = \ell'^\star \wedge \ell' \sqsubseteq \ell$
As $pc \not\sqsubseteq \ell$, so $pc \sqcup \ell'' \not\sqsubseteq \ell$.
Thus from IH, $\Gamma(\sigma'(x)) = \ell'^\star \wedge \ell' \sqsubseteq \ell$
(6) WHILE-T: Let $k = \ell_e$.
IH1: $\forall x.$If $\langle \sigma, c\rangle \Downarrow_{pc\sqcup\ell_e} \sigma''$, $\Gamma(\sigma(x)) = \ell^\star$ and $pc \sqcup \ell_e \not\sqsubseteq \ell$, then $\Gamma(\sigma''(x)) = \ell''^\star \wedge \ell'' \sqsubseteq \ell$
IH2: $\forall x.$If $\langle \sigma'', c\rangle \Downarrow_{pc\sqcup\ell_e} \sigma'$, $\Gamma(\sigma''(x)) = \ell''^\star$ and $pc \sqcup \ell_e \not\sqsubseteq \ell$, then $\Gamma(\sigma'(x)) = \ell'^\star \wedge \ell' \sqsubseteq \ell$
As $pc \not\sqsubseteq \ell$, so $pc \sqcup \ell_e \not\sqsubseteq \ell$.
Thus from IH1 and IH2, $\Gamma(\sigma'(x)) = \ell'^\star \wedge \ell' \sqsubseteq \ell$
(7) WHILE-F : $\sigma = \sigma'$. □

Corollary 1. *If* $\langle \sigma, c\rangle \underset{pc}{\Downarrow} \sigma'$ *and* $\Gamma(\sigma(x)) = \ell^\star$ *and* $\Gamma(\sigma'(x)) = \ell'$, *then* $pc \sqsubseteq \ell$.

Proof. Immediate from Lemma 5. □

Lemma 6 (*pc* lemma)**.** *If* $\langle \sigma, c\rangle \Downarrow_{pc} \sigma'$, *then* $\forall x.\Gamma(\sigma'(x)) = \ell \implies (\sigma(x) = \sigma'(x)) \vee pc \sqsubseteq \ell$.

Proof. Proof by induction on the derivation and case analysis on the last rule.

- SKIP: $\sigma(x) = \sigma'(x)$.
- ASSN-N: For x in the statement c, by premises, $\ell = pc \sqcup \ell_e$. Thus, $pc \sqsubseteq \ell$.
For any other y s.t. $\Gamma(\sigma'(y)) = \ell'$, $\sigma(y) = \sigma'(y)$. For ASSN-S, case does not apply.
- SEQ: IH1: If $\langle \sigma, c_1\rangle \Downarrow_{pc} \sigma''$, then $\forall x.\Gamma(\sigma''(x)) = \ell'' \implies (\sigma(x) = \sigma''(x)) \vee pc \sqsubseteq \ell''$.
IH2: If $\langle \sigma'', c_2\rangle \Downarrow_{pc} \sigma'$, then $\forall x.\Gamma(\sigma'(x)) = \ell \implies (\sigma''(x) = \sigma'(x)) \vee pc \sqsubseteq \ell$.
From IH2, if $\sigma''(x) \neq \sigma'(x)$, then $pc \sqsubseteq \ell$.
If $\sigma''(x) = \sigma'(x)$, then from IH1:
  - If $\sigma(x) = \sigma''(x)$: $\sigma(x) = \sigma'(x)$.
  - If $\sigma(x) \neq \sigma''(x)$: $pc \sqsubseteq \ell''$, where $\ell'' = \Gamma(\sigma''(x))$. As $\sigma''(x) = \sigma'(x)$, $\ell'' = \Gamma(\sigma'(x)) = \ell$. Thus, $pc \sqsubseteq \ell$.
- IF-ELSE: IH: If $\langle \sigma, c\rangle \Downarrow_{pc\sqcup\ell_e} \sigma'$, then $\forall x.\Gamma(\sigma'(x)) = \ell \implies (\sigma(x) = \sigma'(x)) \vee pc \sqcup \ell_e \sqsubseteq \ell$.
From IH, either $(\sigma(x) = \sigma'(x))$ or $pc \sqcup \ell_e \sqsubseteq \ell$. Thus, $(\sigma(x) = \sigma'(x)) \vee pc \sqsubseteq \ell$.
- WHILE-T: IH1: If $\langle \sigma, c\rangle \Downarrow_{pc\sqcup\ell_e} \sigma''$, then $\forall x.\Gamma(\sigma''(x)) = \ell'' \implies (\sigma(x) = \sigma''(x)) \vee pc \sqsubseteq \ell''$.
IH2: If $\langle \sigma'', c\rangle \Downarrow_{pc\sqcup\ell_e} \sigma'$, then $\forall x.\Gamma(\sigma'(x)) = \ell \implies (\sigma''(x) = \sigma'(x)) \vee pc \sqsubseteq \ell$.

From similar reasoning as in "SEQ", either $\sigma(\mathtt{x}) = \sigma'(\mathtt{x})$ or $pc \sqcup \ell_e \sqsubseteq \ell$. Thus, $\sigma(\mathtt{x}) = \sigma'(\mathtt{x}) \vee pc \sqsubseteq \ell$.

- WHILE-F: $\sigma(\mathtt{x}) = \sigma'(\mathtt{x})$. □

Corollary 2. *If* $\langle \sigma, c\rangle \underset{pc}{\Downarrow} \sigma'$ *and* $\Gamma(\sigma(\mathtt{x})) = \ell^\star$ *and* $\Gamma(\sigma'(\mathtt{x})) = \ell'$, *then* $pc \sqsubseteq \ell'$.

Proof. Immediate from Lemma 6. □

Lemma 7 (Confinement lemma)**.** *If* $pc \not\sqsubseteq \ell$, $\langle \sigma, c\rangle \Downarrow_{pc} \sigma'$, *then* $\sigma \sim_\ell \sigma'$.

Proof. Proof by induction on the derivation and case analysis on the last rule.

(1) SKIP : $\sigma = \sigma'$.

(2) ASSN-N: Let $\mathtt{x}_i = \mathtt{v}_i^{k_i}$ and $\mathtt{x}_f = \mathtt{v}_f^{k_f}$, s.t $k_i = \ell_i \vee k_i = \ell_i^\star$ and $pc \sqsubseteq \ell_i$: As $pc \not\sqsubseteq \ell$, $\ell_i \not\sqsubseteq \ell$. By premises of ASSN-N, $k_f = \ell_f \vee k_f = \ell_f^\star$, where $\ell_f = pc \sqcup \ell_e$. As $pc \not\sqsubseteq \ell$, $\ell_f \not\sqsubseteq \ell$. Thus, by definition 7.2, 7.3, 7.4 or 7.5, $\mathtt{x}_i \sim_\ell \mathtt{x}_f$.

(3) ASSN-S: Let $\mathtt{x}_i = \mathtt{v}_i^{k_i}$ and $\mathtt{x}_f = \mathtt{v}_f^{k_f}$, s.t $k_i = \ell_i \vee k_i = \ell_i^\star$ and $pc \not\sqsubseteq \ell_i$: By premise, $k_f = ((pc \sqcup m) \sqcap \ell_i)^\star$. Thus, $\ell_f \sqsubseteq \ell_i$ and by definition 7.3 or 7.5 $\mathtt{x}_i \sim_\ell \mathtt{x}_f$.

(4) SEQ : IH1: $\sigma \sim_\ell \sigma''$ and IH2: $\sigma'' \sim_\ell \sigma'$. T.S : $\sigma \sim_\ell \sigma'$.
For all $\mathtt{x} \in \mathrm{dom}(\sigma)$, respective $\mathtt{x}'' \in \mathrm{dom}(\sigma'')$ and respective $\mathtt{x}' \in \mathrm{dom}(\sigma')$, $\mathtt{x} \sim_\ell \mathtt{x}''$ and $\mathtt{x}'' \sim_\ell \mathtt{x}'$.
To show: $\mathtt{x} \sim_\ell \mathtt{x}'$.
Let $\mathtt{x} = \mathtt{v}_1^{k_1}, \mathtt{x}'' = \mathtt{v}_2^{k_2}, \mathtt{x}' = \mathtt{v}_3^{k_3}$, where $k_1 = \ell_1 \vee k_1 = \ell_1^\star$, $k_2 = \ell_2 \vee k_2 = \ell_2^\star$ and $k_3 = \ell_3 \vee k_3 = \ell_3^\star$.
Case-analysis on Definition 7 for IH1.

- $(k_1 = k_2) = \ell' \sqsubseteq \ell \wedge \mathtt{v}_1 = \mathtt{v}_2$: By IH2 and Definition 7,
  - (a) $(k_2 = k_3) = \ell' \sqsubseteq \ell \wedge \mathtt{v}_2 = \mathtt{v}_3$ (case 1): Transitivity of equality, $(k_1 = k_3) = \ell' \sqsubseteq \ell \wedge \mathtt{v}_1 = \mathtt{v}_3$. Thus, $\mathtt{x} \sim_\ell \mathtt{x}'$.
  - (b) $k_2 = \ell'$ and $k_3 = \ell_3^\star \wedge \ell_3 \sqsubseteq \ell' \sqsubseteq \ell$ (case 5): By Definition 7(5) $\mathtt{x} \sim_\ell \mathtt{x}'$.
- $k_1 = \ell_1 \not\sqsubseteq \ell \wedge k_2 = \ell_2 \not\sqsubseteq \ell$: By IH2, either
  - (a) $k_2 = \ell_2 \not\sqsubseteq \ell \wedge k_3 = \ell_3 \not\sqsubseteq \ell$. By Definition 7(2), $\mathtt{x} \sim_\ell \mathtt{x}'$.
  - (b) $k_2 = \ell_2 \not\sqsubseteq \ell \wedge k_3 = \ell_3^\star$: $\ell_1 \not\sqsubseteq \ell$. Thus, by Definition 7(5), $\mathtt{x} \sim_\ell \mathtt{x}'$.
- $k_1 = \ell_1^\star \wedge k_2 = \ell_2^\star$: By IH2,
  - (a) $k_2 = \ell_2^\star \wedge k_3 = \ell_3^\star$ (case 3): By Definition 7(3), $\mathtt{x} \sim_\ell \mathtt{x}'$.
  - (b) $k_2 = \ell_2^\star \wedge k_3 = \ell_3 \wedge (\ell_3 \not\sqsubseteq \ell)$ (case 4): By Definition 7(4), $\mathtt{x} \sim_\ell \mathtt{x}'$.
  - (c) $k_2 = \ell_2^\star \wedge k_3 = \ell_3 \wedge (\ell_2 \sqsubseteq \ell_3)$ (case 4): By Corollary 1, $pc \sqsubseteq \ell_2$. As $pc \not\sqsubseteq \ell$ and $\ell_2 \sqsubseteq \ell_3$, so $\ell_3 \not\sqsubseteq \ell$. By Definition 7(4), $\mathtt{x} \sim_\ell \mathtt{x}'$..
- $k_1 = \ell_1^\star \wedge k_2 = \ell_2$ *s.t.* $(\ell_2 \not\sqsubseteq \ell)$ (case 4): Either
  - $k_2 = \ell_2 \not\sqsubseteq \ell \wedge k_3 = \ell_3 \not\sqsubseteq \ell$: By Definition 7(4), $\mathtt{x} \sim_\ell \mathtt{x}'$.
  - $k_2 = \ell_2 \not\sqsubseteq \ell \wedge k_3 = \ell_3^\star$: By Definition 7(3), $\mathtt{x} \sim_\ell \mathtt{x}'$.
- $k_1 = \ell_1^\star \wedge k_2 = \ell_2$ *s.t.* $(\ell_1 \sqsubseteq \ell_2)$ (case 4):
  - $k_2 = k_3 = \ell_2$: By Definition 7(4), $\mathtt{x} \sim_\ell \mathtt{x}'$.
  - $k_2 = \ell_2 \not\sqsubseteq \ell \wedge k_3 = \ell_3 \not\sqsubseteq \ell$: By Definition 7(4), $\mathtt{x} \sim_\ell \mathtt{x}'$.

$*$ $k_2 = \ell_2 \not\sqsubseteq \ell \wedge k_3 = \ell_3^\star$: By Definition 7(3), $\mathrm{x} \sim_\ell \mathrm{x}'$.

- $k_1 = \ell_1 \wedge k_2 = \ell_2^\star \; s.t. \; (\ell_1 \not\sqsubseteq \ell)$: By IH2,
  (a) $k_2 = \ell_2^\star \wedge k_3 = \ell_3^\star$ (case 3): By Definition 7(5), $\mathrm{x} \sim_\ell \mathrm{x}'$.
  (b) $k_2 = \ell_2^\star \wedge k_3 = \ell_3 \; s.t. \; (\ell_3 \not\sqsubseteq \ell)$ (case 4): By Definition 7(2), $\mathrm{x} \sim_\ell \mathrm{x}'$.
  (c) $k_2 = \ell_2^\star \wedge k_3 = \ell_3 \; s.t. \; (\ell_2 \sqsubseteq \ell_3)$ (case 4): By Corollary 1, $pc \sqsubseteq \ell_2$. As $pc \not\sqsubseteq \ell$ and $\ell_2 \sqsubseteq \ell_3$, so $\ell_3 \not\sqsubseteq \ell$. By Definition 7(2), $\mathrm{x} \sim_\ell \mathrm{x}'$.

- $k_1 = \ell_1 \wedge k_2 = \ell_2^\star \; s.t. \; (\ell_2 \sqsubseteq \ell_1)$: Also, $(\ell_2 \sqsubseteq \ell_1 \sqsubseteq \ell)$. By IH2,
  (a) $k_2 = \ell_2^\star \wedge k_3 = \ell_3^\star$ (case 3): As $\ell_2 \sqsubseteq \ell$ and $pc \not\sqsubseteq \ell$, $pc \not\sqsubseteq \ell_2$. By lemma 5, $\ell_3 \sqsubseteq \ell_2$. Thus, $\ell_3 \sqsubseteq \ell_2 \sqsubseteq \ell_1$. By Definition 7(5), $\mathrm{x} \sim_\ell \mathrm{x}'$.
  (b) $k_2 = \ell_2^\star \wedge k_3 = \ell_3$ (case 4): As $\ell_2 \sqsubseteq \ell$ and $pc \not\sqsubseteq \ell$, $pc \not\sqsubseteq \ell_2$. But, by Corollary 1, $pc \sqsubseteq \ell_2$. By contradiction, this case does not hold.

(5) IF-ELSE : IH : $k = \ell'$. If $(pc \sqcup \ell') \not\sqsubseteq \ell$, then $\sigma \sim_\ell \sigma'$.
As $pc \not\sqsubseteq \ell$, $pc \sqcup \ell' \not\sqsubseteq \ell$. Thus, by IH, $\sigma \sim_\ell \sigma'$.
(6) WHILE-T: IH1 : $k = \ell'$. If $(pc \sqcup \ell') \not\sqsubseteq \ell$, then $\sigma \sim_\ell \sigma'$.
As $pc \not\sqsubseteq \ell$, $pc \sqcup \ell' \not\sqsubseteq \ell$. Thus, by IH1, $\sigma \sim_\ell \sigma''$.
IH2 : $k = \ell'$. If $(pc \sqcup \ell') \not\sqsubseteq \ell$, then $\sigma' \sim_\ell \sigma''$.
As $pc \not\sqsubseteq \ell$, $pc \sqcup \ell' \not\sqsubseteq \ell$. Thus, by IH, $\sigma'' \sim_\ell \sigma'$.
Therefore, $\sigma \sim_\ell \sigma''$ and $\sigma'' \sim_\ell \sigma'$.
(Reasoning similar to SEQ.)
(7) WHILE-F : $\sigma = \sigma'$ $\square$

Theorem 2. Termination-insensitive non-interference
If $\sigma_1 \sim_\ell \sigma_2$, $\langle \sigma_1, c \rangle \Downarrow_{pc} \sigma_1'$, $\langle \sigma_2, c \rangle \Downarrow_{pc} \sigma_2'$, *then* $\sigma_1' \sim_\ell \sigma_2'$.

Proof. By induction on the derivation and case analysis on the last step

(1) SKIP: $\sigma_1' = \sigma_1 \sim_\ell \sigma_2 = \sigma_2'$
(2) ASSN-N and ASSN-S: As $\sigma_1 \sim_\ell \sigma_2$, $\forall \mathrm{x}.\sigma_1(\mathrm{x}) \sim_\ell \sigma_2(\mathrm{x})$. Let $\sigma_1(\mathrm{x}) = \mathrm{v}_1^{k_1}$, $\sigma_2(\mathrm{x}) = \mathrm{v}_2^{k_2}$ and $\sigma_1'(\mathrm{x}) = \mathrm{v}_1'^{k_1'}$, $\sigma_2'(\mathrm{x}) = \mathrm{v}_2'^{k_2'}$
s. t. $k_i = \ell_i \vee k_i = \ell_i^\star$ and $k_i' = \ell_i' \vee k_i = \ell_i'^\star$ for $i = 1, 2$.
Let $\langle e_1, \sigma_1 \rangle \Downarrow w_1^{k_1^e} \wedge \langle e_2, \sigma_2 \rangle \Downarrow w_2^{k_2^e}$
s. t. $k_i^e = \ell_i^e \vee k_i^e = \ell_i^{e\star}$ for $i = 1, 2$. For low-equivalence of e_1 and e_2, the following cases arise:
  (a) $k_i^e = \ell_i^e, s.t. \; (\ell_1^e = \ell_2^e) = \ell^e \sqsubseteq \ell \wedge w_1 = w_2$:
    i. $pc \not\sqsubseteq \ell_1 \wedge pc \not\sqsubseteq \ell_2$: By premise of ASSN-S rules, $k_i' = ((pc \sqcup \ell^e) \sqcap \ell_i)^\star$. By Definition 7(3), $\sigma_1' \sim_\ell \sigma_2'$.
    ii. $pc \not\sqsubseteq \ell_1 \wedge pc \sqsubseteq \ell_2$: $k_1' = ((pc \sqcup \ell^e) \sqcap \ell_1)^\star$ and $k_2' = pc \sqcup \ell^e$. As $\ell_1' \sqsubseteq \ell_2'$, by Definition 7(4), $\sigma_1' \sim_\ell \sigma_2'$.
    iii. $pc \sqsubseteq \ell_1 \wedge pc \not\sqsubseteq \ell_2$: $k_2' = ((pc \sqcup \ell^e) \sqcap \ell_2)^\star$ and $k_1' = pc \sqcup \ell^e$. As $\ell_2' \sqsubseteq \ell_1'$, by Definition 7(5), $\sigma_1' \sim_\ell \sigma_2'$.
    iv. $pc \sqsubseteq \ell_1 \wedge pc \sqsubseteq \ell_2$: $k_1' = pc \sqcup \ell^e$ and $k_2' = pc \sqcup \ell^e$. If $pc \sqsubseteq \ell$ and $\ell^e \sqsubseteq \ell$ and $w_1 = w_2$, by Definition 7(1), $\sigma_1' \sim_\ell \sigma_2'$. If $pc \not\sqsubseteq \ell$, $pc \sqcup \ell^e \not\sqsubseteq \ell$. By Definition 7(2), $\sigma_1' \sim_\ell \sigma_2'$.

(b) $\ell_1^e \not\sqsubseteq \ell \wedge \ell_2^e \not\sqsubseteq \ell$: From premise of assignment rules, $k_1' = pc \sqcup \ell_1^e \vee k_1' = (pc \sqcup \ell_1^e)^\star \vee k_1' = ((pc \sqcup \ell_1^e) \sqcap \ell_1)^\star$. Similarly, $k_2' = pc \sqcup \ell_2^e \vee k_2' = (pc \sqcup \ell_2^e)^\star \vee k_2' = ((pc \sqcup \ell_2^e) \sqcap \ell_2)^\star$. Since $\ell_1^e \not\sqsubseteq \ell$ and $\ell_2^e \not\sqsubseteq \ell$, $pc \sqcup \ell_1^e \not\sqsubseteq \ell$ and $pc \sqcup \ell_1^e \not\sqsubseteq \ell$. Therefore, from Definition 7(2), 7(3), 7(4) or 7(5) $\sigma_1' \sim_\ell \sigma_2'$.

(c) $k_i^e = \ell_i^{e\star}$: By premise of ASSN-S rules, $k_i' = ((pc \sqcup \ell_i^e) \sqcap \ell_i)^\star$ or $k_i' = (pc \sqcup \ell_i^e)^\star$. By Definition 7(3), $\sigma_1' \sim_\ell \sigma_2'$.

(d) $k_1^e = \ell_1^{e\star} \wedge k_2^e = \ell_2^e$:

i. $pc \not\sqsubseteq \ell_1 \wedge pc \not\sqsubseteq \ell_2$: By premise of ASSN-S rules, $k_i' = ((pc \sqcup \ell_i^e) \sqcap \ell_i)^\star$. By Definition 7(3), $\sigma_1' \sim_\ell \sigma_2'$.

ii. $pc \not\sqsubseteq \ell_1 \wedge pc \sqsubseteq \ell_2$: $k_1' = ((pc \sqcup \ell_1^e) \sqcap \ell_1)^\star$ and $k_2' = pc \sqcup \ell_2^e$. From Definition 7(4), $\ell_1^e \sqsubseteq \ell_2^e$, so $(pc \sqcup \ell_i^e) \sqcap \ell_1 \sqsubseteq pc \sqcup \ell_2^e$. By Definition 7(4), $\sigma_1' \sim_\ell \sigma_2'$.

iii. $pc \sqsubseteq \ell_1 \wedge pc \not\sqsubseteq \ell_2$: $k_2' = ((pc \sqcup \ell_2^e) \sqcap \ell_2)^\star$ and $k_1' = (pc \sqcup \ell_1^e)^\star$. By Definition 7(3), $\sigma_1' \sim_\ell \sigma_2'$.

iv. $pc \sqsubseteq \ell_1 \wedge pc \sqsubseteq \ell_2$: $k_1' = (pc \sqcup \ell_1^e)^\star$ and $k_2' = pc \sqcup \ell_2^e$. If $\ell_2^e \not\sqsubseteq \ell$, so $pc \sqcup \ell_2^e \not\sqsubseteq \ell$. Else if $\ell_1^e \sqsubseteq \ell_2^e$, then $pc \sqcup \ell_1^e \sqsubseteq pc \sqcup \ell_2^e$. By Definition 7(4), $\sigma_1' \sim_\ell \sigma_2'$.

(e) $k_1^e = \ell_1^e \wedge k_2^e = \ell_2^{e\star}$:

i. $pc \not\sqsubseteq \ell_1 \wedge pc \not\sqsubseteq \ell_2$: By premise of ASSN-S rules, $k_i' = ((pc \sqcup \ell_i^e) \sqcap \ell_i)^\star$. By Definition 7(3), $\sigma_1' \sim_\ell \sigma_2'$.

ii. $pc \not\sqsubseteq \ell_1 \wedge pc \sqsubseteq \ell_2$: $k_1' = ((pc \sqcup \ell_1^e) \sqcap \ell_1)^\star$ and $k_2' = (pc \sqcup \ell_2^e)^\star$. By Definition 7(3), $\sigma_1' \sim_\ell \sigma_2'$.

iii. $pc \sqsubseteq \ell_1 \wedge pc \not\sqsubseteq \ell_2$: $k_1' = pc \sqcup \ell_1^e$ and $k_2' = ((pc \sqcup \ell_2^e) \sqcap \ell_2)^\star$. $(pc \sqcup \ell_2^e) \sqcap \ell_2 \sqsubseteq pc \sqcup \ell_1^e$. By Definition 7(5), $\sigma_1' \sim_\ell \sigma_2'$.

iv. $pc \sqsubseteq \ell_1 \wedge pc \sqsubseteq \ell_2$: $k_1' = (pc \sqcup \ell_1^e)^\star$ and $k_2' = pc \sqcup \ell_2^e$. If $\ell_1^e \not\sqsubseteq \ell$, so $pc \sqcup \ell_1^e \not\sqsubseteq \ell$. Else if $\ell_2^e \sqsubseteq \ell_1^e$, then $pc \sqcup \ell_2^e \sqsubseteq pc \sqcup \ell_1^e$. By Definition 7(5), $\sigma_1' \sim_\ell \sigma_2'$.

(3) SEQ: IH1: If $\sigma_1 \sim_\ell \sigma_2$ then $\sigma_1'' \sim_\ell \sigma_2''$
IH2: If $\sigma_1'' \sim_\ell \sigma_2''$ then $\sigma_1' \sim_\ell \sigma_2'$
Since $\sigma_1 \sim_\ell \sigma_2$, therefore, from IH1 and IH2 $\sigma_1' \sim_\ell \sigma_2'$.

(4) IF-ELSE: IH: If $\sigma_1 \sim_\ell \sigma_2$, $\langle \sigma_1, c\rangle \Downarrow_{pc\sqcup\ell_1^e} \sigma_1'$, $\langle \sigma_2, c\rangle \Downarrow_{pc\sqcup\ell_2^e} \sigma_2'$ and $pc \sqcup \ell_1^e = pc \sqcup \ell_2^e$ then $\sigma_1' \sim_\ell \sigma_2'$.

- If $\ell_1^e \sqsubseteq \ell$, $\ell_1^e = \ell_2^e$ and $\mathtt{n}_1 = \mathtt{n}_2$. By IH, $\sigma_1' \sim_\ell \sigma_2'$.
- If $\ell_1^e \not\sqsubseteq \ell$, then $\ell_2^e \not\sqsubseteq \ell$, $pc \sqcup \ell_i^e \not\sqsubseteq \ell$ for $i = 1, 2$. By Lemma 7, $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$. T.S. $\sigma_1' \sim_\ell \sigma_2'$, i.e., $(\forall \mathtt{x}.\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x}))$
  Case analysis on the definition of low-equivalence of values, x, in σ_1 and σ_2. Let $\sigma_1(\mathtt{x}) = \mathtt{v}_1^{k_1}$ and $\sigma_2(\mathtt{x}) = \mathtt{v}_2^{k_2}$ and $\sigma_1'(\mathtt{x}) = \mathtt{v}_1'^{k_1'}$ and $\sigma_2'(\mathtt{x}) = \mathtt{v}_2'^{k_2'}$

  (a) $(k_1 = k_2) = \ell' \sqsubseteq \ell \wedge \mathtt{v}_1 = \mathtt{v}_2 = \mathtt{v}$:

  * If $k_1' = \ell_1' \wedge k_2' = \ell_2'$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Definition 7(1), $\ell' = \ell_1' \wedge \mathtt{v} = \mathtt{v}_1'$ and $\ell' = \ell_2' \wedge \mathtt{v} = \mathtt{v}_2'$. Thus, $\ell_1' = \ell_2' \wedge \mathtt{v}_1' = \mathtt{v}_2'$, so $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
  * If $k_1' = \ell_1'^\star \wedge k_2' = \ell_2'$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Definition 7(5) $\ell_1' \sqsubseteq \ell_1 = \ell'$ and by Definition 7(1) $k_2' = \ell_2' = \ell_2 = \ell'$. So, $\ell_1' \sqsubseteq \ell_2'$. By Definition 7(4), $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
  * If $k_1' = \ell_1' \wedge k_2' = \ell_2'^\star$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Definition 7(1) $k_1' = \ell_1' = \ell_1 = \ell'$ and by Definition 7(5) $\ell_2' \sqsubseteq \ell_2 = \ell'$. So, $\ell_2' \sqsubseteq \ell_1'$. By Definition 7(5), $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
  * If $k_1' = \ell_1'^\star \wedge k_2' = \ell_2'^\star$, then by Definition 7(3), $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.

  (b) $(k_1 = \ell_1 \not\sqsubseteq \ell) \wedge (k_2 = \ell_2 \not\sqsubseteq \ell)$:

* If $k_1' = \ell_1' \wedge k_2' = \ell_2'$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Definition 7(2), $(k_1' = \ell_1' \not\sqsubseteq \ell) \wedge (k_2' = \ell_2' \not\sqsubseteq \ell)$. So, $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$.
* If $k_1' = \ell_1'^\star \wedge k_2' = \ell_2'$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Definition 7(2) $k_2' = \ell_2' \not\sqsubseteq \ell$. By Definition 7(4), $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$.
* If $k_1' = \ell_1' \wedge k_2' = \ell_2'^\star$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Definition 7(2) $k_1' = \ell_1' \not\sqsubseteq \ell$. By Definition 7(5), $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$. If $k_1' = \ell_1'^\star \wedge k_2' = \ell_2'^\star$, then by Definition 7(3), $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$.

(c) $(k_1 = \ell_1^\star \wedge k_2 = \ell_2^\star)$:

* If $k_1' = \ell_1'^\star \wedge k_2' = \ell_2'^\star$, by Definition 7(3), $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$.
* If $k_1' = \ell_1' \wedge k_2' = \ell_2'^\star$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Corollary 2, $pc \sqcup \ell_1^e \sqsubseteq \ell_1'$. As $pc \sqcup \ell_1^e \not\sqsubseteq \ell$ and by Definition 7(2), $\ell_1' \not\sqsubseteq \ell$. By Definition 7(5), $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$.
* If $k_1' = \ell_1'^\star \wedge k_2' = \ell_2'$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Corollary 2, $pc \sqcup \ell_2^e \sqsubseteq \ell_2'$. As $pc \sqcup \ell_2^e \not\sqsubseteq \ell$ and by Definition 7(2), $\ell_2' \not\sqsubseteq \ell$. By Definition 7(4), $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$.
* If $k_1' = \ell_1' \wedge k_2' = \ell_2'$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Corollary 2, $pc \sqcup \ell_1^e \sqsubseteq \ell_1'$ and $pc \sqcup \ell_2^e \sqsubseteq \ell_2'$. As $pc \sqcup \ell_i^e \not\sqsubseteq \ell$ and by Definition 7(2), $\ell_1' \not\sqsubseteq \ell$ and $\ell_2' \not\sqsubseteq \ell$. By Definition 7(2), $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$.

(d) $(k_1 = \ell_1^\star \wedge k_2 = \ell_2)$:

* $\ell_2 \not\sqsubseteq \ell$:
  * If $k_1' = \ell_1'^\star \wedge k_2' = \ell_2'^\star$, by Definition 7(3), $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$.
  * If $k_1' = \ell_1' \wedge k_2' = \ell_2'^\star$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Corollary 2, $pc \sqcup \ell_1^e \sqsubseteq \ell_1'$. As $pc \sqcup \ell_1^e \not\sqsubseteq \ell$ and by Definition 7(2), $\ell_1' \not\sqsubseteq \ell$. By Definition 7(5), $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$.
  * If $k_1' = \ell_1'^\star \wedge k_2' = \ell_2'$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Definition 7(2), $\ell_2' \not\sqsubseteq \ell$. By Definition 7(4), $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$.
  * If $k_1' = \ell_1' \wedge k_2' = \ell_2'$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Corollary 2, $pc \sqcup \ell_1^e \sqsubseteq \ell_1'$. As $pc \sqcup \ell_1^e \not\sqsubseteq \ell$ and by Definition 7(2), $\ell_1' \not\sqsubseteq \ell$. By Definition 7(2), $\ell_2' \not\sqsubseteq \ell$. By Definition 7(2), $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$.
* $\ell_1 \sqsubseteq \ell_2 \sqsubseteq \ell$:
  * If $k_1' = \ell_1'^\star \wedge k_2' = \ell_2'^\star$, by Definition 7(3), $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$.
  * If $k_1' = \ell_1' \wedge k_2' = \ell_2'^\star$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Corollary 2, $pc \sqcup \ell_1^e \sqsubseteq \ell_1'$. As $pc \sqcup \ell_1^e \not\sqsubseteq \ell$, and by Definition 7(2), $\ell_1' \not\sqsubseteq \ell$. By Definition 7(5), $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$.
  * If $k_1' = \ell_1'^\star \wedge k_2' = \ell_2'$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, $\ell_1' \sqsubseteq (pc \sqcup \ell_1^e) \sqcap \ell_1$ as $pc \sqcup \ell_1^e \not\sqsubseteq \ell_1$ and $\ell_2' = \ell_2$ by Corollary 1 and Definition 7(1). Thus, $\ell_1' \sqsubseteq \ell_2'$. By Definition 7(4), $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$.
  * If $k_1' = \ell_1' \wedge k_2' = \ell_2'$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Corollary 1, $pc \sqcup \ell_1^e \sqsubseteq \ell_1$. As $pc \sqcup \ell_1^e \not\sqsubseteq \ell$, by contradiction the case does not hold.

(e) $(k_1 = \ell_1 \wedge k_2 = \ell_2^\star)$:

* $\ell_1 \not\sqsubseteq \ell$:
  * If $k_1' = \ell_1'^\star \wedge k_2' = \ell_2'^\star$, by Definition 7(3), $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$.
  * If $k_1' = \ell_1'^\star \wedge k_2' = \ell_2'$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Corollary 2, $pc \sqcup \ell_2^e \sqsubseteq \ell_2'$. As $pc \sqcup \ell_2^e \not\sqsubseteq \ell$ and by Definition 7(2), $\ell_2' \not\sqsubseteq \ell$. By Definition 7(5), $\sigma_1'(\mathrm{x}) \sim_\ell \sigma_2'(\mathrm{x})$.

* If $k_1' = \ell_1' \wedge k_2' = \ell_2'^{\star}$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Definition 7(2), $\ell_1' \not\sqsubseteq \ell$. By Definition 7(4), $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
* If $k_1' = \ell_1' \wedge k_2' = \ell_2'$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Corollary 2, $pc \sqcup \ell_2^e \sqsubseteq \ell_2'$. As $pc \sqcup \ell_2^e \not\sqsubseteq \ell$ and by Definition 7(2), $\ell_2' \not\sqsubseteq \ell$. By Definition 7(2), $\ell_1' \not\sqsubseteq \ell$. By Definition 7(2), $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.

* $\ell_2 \sqsubseteq \ell_1$:
  * If $k_1' = \ell_1'^{\star} \wedge k_2' = \ell_2'^{\star}$, by Definition 7(3), $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
  * If $k_1' = \ell_1' \wedge k_2' = \ell_2'^{\star}$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, $\ell_2' \sqsubseteq (pc \sqcup \ell_2^e) \sqcap \ell_2$ as $pc \sqcup \ell_2^e \not\sqsubseteq \ell_2$ and $\ell_1' = \ell_1$ by Corollary 1 and Definition 7(1). Thus, $\ell_2' \sqsubseteq \ell_1'$. By Definition 7(5), $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
  * If $k_1' = \ell_1'^{\star} \wedge k_2' = \ell_2'$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Corollary 2, $pc \sqcup \ell_2^e \sqsubseteq \ell_2'$. As $pc \sqcup \ell_2^e \not\sqsubseteq \ell$, and by Definition 7(2), $\ell_2' \not\sqsubseteq \ell$. By Definition 7(4), $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
  * If $k_1' = \ell_1' \wedge k_2' = \ell_2'$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, by Corollary 1, $pc \sqcup \ell_2^e \sqsubseteq \ell_2$. As $pc \sqcup \ell_2^e \not\sqsubseteq \ell$, by contradiction the case does not hold.

(5) WHILE-T: IH1: If $\sigma_1 \sim_\ell \sigma_2$, $\langle \sigma_1, c\rangle \Downarrow_{pc\sqcup\ell_1^e} \sigma_1''$, $\langle \sigma_2, c\rangle \Downarrow_{pc\sqcup\ell_2^e} \sigma_2''$ and $pc \sqcup \ell_1^e = pc \sqcup \ell_2^e$ then $\sigma_1'' \sim_\ell \sigma_2''$.
IH2: If $\sigma_1'' \sim_\ell \sigma_2''$, $\langle \sigma_1'', c\rangle \Downarrow_{pc\sqcup\ell_1^e} \sigma_1'$, $\langle \sigma_2'', c\rangle \Downarrow_{pc\sqcup\ell_2^e} \sigma_2'$ and $pc \sqcup \ell_1^e = pc \sqcup \ell_2^e$ then $\sigma_1' \sim_\ell \sigma_2'$.

- If $\ell_1^e \sqsubseteq \ell$, $\ell_1^e = \ell_2^e$ and $\mathtt{n}_1 = \mathtt{n}_2$. By IH1 and IH2, $\sigma_1' \sim_\ell \sigma_2'$.
- If $\ell_1^e \not\sqsubseteq \ell$, then $\ell_2^e \not\sqsubseteq \ell$, $pc \sqcup \ell_i^e \not\sqsubseteq \ell$ for $i = 1, 2$. By Lemma 7, $\sigma_1 \sim_\ell \sigma_1''$ and $\sigma_2 \sim_\ell \sigma_2''$.
  T.S. $\sigma_1'' \sim_\ell \sigma_2''$: By similar reasoning as IF-ELSE.
  As $\sigma_1'' \sim_\ell \sigma_2''$, and by Lemma 7, $\sigma_1'' \sim_\ell \sigma_1'$ and $\sigma_2'' \sim_\ell \sigma_2'$.
  T.S. $\sigma_1' \sim_\ell \sigma_2'$: By similar reasoning as IF-ELSE.

(6) WHILE-F: $\sigma_1' = \sigma_1 \sim_\ell \sigma_2 = \sigma_2'$ □

Appendix B. Proofs for IFC with unstructured control flow and exceptions

Lemma 8 (Confinement lemma). *If* $\langle \sigma, \iota, \rho\rangle \rightarrow \langle \sigma', \iota', \rho'\rangle$ *and* $\Gamma(!\rho) \not\sqsubseteq \ell$, *then* $\sigma \sim_\ell \sigma'$, *and* $\rho \sim_\ell \rho'$.

Proof. $\Gamma(!\rho) = pc$ in the proof that follows.

As $pc \not\sqsubseteq \ell$, the nodes in the pc-stack that have label less than or equal to ℓ will remain unchanged. Branching instructions pushing a new node would have label of at least pc due to monotonicity of pc-stack. Even if ι' is the IPD corresponding to the $!\rho.ipd$, it would only pop the top node. Thus, all the nodes that have label less than or equal to ℓ will remain unchanged. Hence, $\rho \sim_\ell \rho'$.

To show: $\sigma \sim_\ell \sigma'$.

By case analysis on the instruction type:

- ASSN, CATCH: Similar to cases ASSN-N and ASSN-S of Lemma 7.
- BRANCH, JMP, RET, SEN, THROW: $\sigma = \sigma'$ □

Lemma 9. *If* $\langle \sigma_0, \iota_0, \rho_0\rangle \rightarrow^n \langle \sigma_n, \iota_n, \rho_n\rangle$ *and* $\forall (0 \leqslant i \leqslant n).\Gamma(!\rho_i) \not\sqsubseteq \ell$, *then* $\rho_0 \sim_\ell \rho_n$

Proof. Proof by induction on n.

Basis: $\rho_0 \sim_\ell \rho_0$

IH : $\rho_0 \sim_\ell \rho_{n-1}$

From Definition 14, all nodes labeled less than or equal to ℓ of ρ_0 and ρ_{n-1} are equal. From Lemma 8, $\rho_{n-1} \sim \rho_n$ so, all nodes labeled less than or equal to ℓ of ρ_{n-1} and ρ_n are equal. Thus, all nodes labeled less than or equal to ℓ of ρ_0 and ρ_n are equal and by Definition 14, $\rho_0 \sim \rho_n$. □

Lemma 10 ($\star$-preservation lemma). *If* $\langle \sigma, \iota, \rho \rangle \rightarrow \langle \sigma', \iota', \rho' \rangle$, *then* $\forall \mathrm{x}.\Gamma(\sigma(\mathrm{x})) = \ell^\star \wedge (\Gamma(!\rho) \not\sqsubseteq \ell) \implies \Gamma(\sigma'(\mathrm{x})) = \ell'^\star \wedge \ell' \sqsubseteq \ell$

Proof. Proof by case analysis on the instruction type:

- ASSN, CATCH: from the premise
- BRANCH, JMP, RET, SEN, THROW: $\sigma = \sigma'$ □

Corollary 3. *If* $\langle \sigma, \iota, \rho \rangle \rightarrow \langle \sigma', \iota', \rho' \rangle$, *and* $\Gamma(\sigma(\mathrm{x})) = \ell^\star$ *and* $\Gamma(\sigma'(\mathrm{x})) = \ell'$, *then* $\Gamma(!\rho) \sqsubseteq \ell$.

Proof. Immediate from Lemma 10. □

Lemma 11. *If* $\langle \sigma_0, \iota_0, \rho_0 \rangle \rightarrow^\star \langle \sigma_n, \iota_n, \rho_n \rangle$ *and* $\forall (0 \leqslant i \leqslant n).\Gamma(!\rho_i) \not\sqsubseteq \ell$, *then* $\sigma_0 \sim_\ell \sigma_n$

Proof. By induction on n.
Basis: $\sigma_0 \sim_\ell \sigma_0$ by Definition 8.
IH: $\sigma_0 \sim_\ell \sigma_{n-1}$.
From IH and Definition 8, $\forall \mathrm{x}.(\sigma_0(\mathrm{x}) \sim_\ell \sigma_{n-1}(\mathrm{x})$. From Lemma 8, $\sigma_{n-1} \sim_\ell \sigma_n$. Thus, $\forall \mathrm{x}.(\sigma_{n-1}(\mathrm{x}) \sim_\ell \sigma_n(\mathrm{x})$
Assume that $\sigma_0(\mathrm{x}) = \mathrm{v}_0^k$, $\sigma_{n-1}(\mathrm{x}) = \mathrm{v}_{n-1}^{k'}$, and $\sigma_n(\mathrm{x}) = \mathrm{v}_n^{k''}$ either:

- $(k = k') = \ell' \sqsubseteq \ell \wedge v_0 = v_{n-1}$:
  (1) $(k' = k'') = \ell' \sqsubseteq \ell \wedge v_{n-1} = v_n$: $(k = k'') = \ell' \sqsubseteq \ell \wedge v_0 = v_n$. Thus, $\sigma_0(\mathrm{x}) \sim_\ell \sigma_n(\mathrm{x})$.
  (2) $k' = \ell'$ and $k'' = \ell'^\star \wedge \ell'' \sqsubseteq \ell' \sqsubseteq \ell$: By Definition 7(5). Thus, $\sigma_0(\mathrm{x}) \sim_\ell \sigma_n(\mathrm{x})$.
- $k = \ell_1 \not\sqsubseteq \ell \wedge k' = \ell_2 \not\sqsubseteq \ell$:
  (1) $k' = \ell_2 \not\sqsubseteq \ell \wedge k'' = \ell_3 \not\sqsubseteq \ell$. By Definition 7(2), $\sigma_0(\mathrm{x}) \sim_\ell \sigma_n(\mathrm{x})$.
  (2) $k' = \ell_2 \not\sqsubseteq \ell \wedge k'' = \ell_3^\star$: $\ell_1 \not\sqsubseteq \ell$. Thus, by Definition 7(5), $\sigma_0(\mathrm{x}) \sim_\ell \sigma_n(\mathrm{x})$.
- $k = \ell_1^\star \wedge k' = \ell_2^\star$:
  (1) $k' = \ell_2^\star \wedge k'' = \ell_3^\star$: By Definition 7(3), $\sigma_0(\mathrm{x}) \sim_\ell \sigma_n(\mathrm{x})$.
  (2) $k' = \ell_2^\star \wedge k'' = \ell_3 \wedge (\ell_3 \not\sqsubseteq \ell)$: By Definition 7(4), $\sigma_0(\mathrm{x}) \sim_\ell \sigma_n(\mathrm{x})$.
  (3) $k' = \ell_2^\star \wedge k'' = \ell_3 \wedge (\ell_2 \sqsubseteq \ell_3)$: By Corollary 3, $\Gamma(!\rho_{n-1}) \sqsubseteq \ell_2$. As $\Gamma(!\rho_{n-1}) \not\sqsubseteq \ell$ and $\ell_2 \sqsubseteq \ell_3$, so $\ell_3 \not\sqsubseteq \ell$. By Definition 7(4), $\sigma_0(\mathrm{x}) \sim_\ell \sigma_n(\mathrm{x})$.
- $k = \ell_1^\star \wedge k' = \ell_2$ *s.t.* $(\ell_2 \not\sqsubseteq \ell)$: Either
  * $k' = \ell_2 \not\sqsubseteq \ell \wedge k'' = \ell_3 \not\sqsubseteq \ell$: By Definition 7(4), $\sigma_0(\mathrm{x}) \sim_\ell \sigma_n(\mathrm{x})$
  * $k' = \ell_2 \not\sqsubseteq \ell \wedge k'' = \ell_3^\star$: By Definition 7(3), $\sigma_0(\mathrm{x}) \sim_\ell \sigma_n(\mathrm{x})$
- $k = \ell_1^\star \wedge k' = \ell_2$ *s.t.* $(\ell_1 \sqsubseteq \ell_2)$:
  * $k' = k'' = \ell_2$: By Definition 7(4), $\sigma_0(\mathrm{x}) \sim_\ell \sigma_n(\mathrm{x})$
  * $k' = \ell_2 \not\sqsubseteq \ell \wedge k'' = \ell_3 \not\sqsubseteq \ell$: By Definition 7(4), $\sigma_0(\mathrm{x}) \sim_\ell \sigma_n(\mathrm{x})$
  * $k' = \ell_2 \not\sqsubseteq \ell \wedge k'' = \ell_3^\star$: By Definition 7(3), $\sigma_0(\mathrm{x}) \sim_\ell \sigma_n(\mathrm{x})$

- $k = \ell_1 \wedge k' = \ell_2^\star \; s.t. \; (\ell_1 \not\sqsubseteq \ell)$:

  (1) $k' = \ell_2^\star \wedge k'' = \ell_3^\star$: By Definition 7(5), $\sigma_0(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$
  (2) $k' = \ell_2^\star \wedge k'' = \ell_3 \; s.t. \; (\ell_3 \not\sqsubseteq \ell)$: By Definition 7(2), $\sigma_0(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$
  (3) $k' = \ell_2^\star \wedge k'' = \ell_3 \; s.t. \; (\ell_2 \sqsubseteq \ell_3)$: By Corollary 3, $\Gamma(!\rho_n) \sqsubseteq \ell_2$. As $\Gamma(!\rho_n) \not\sqsubseteq \ell$ and $\ell_2 \sqsubseteq \ell_3$, so $\ell_3 \not\sqsubseteq \ell$. By Definition 7(2), $\sigma_0(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$.

- $k = \ell_1 \wedge k' = \ell_2^\star \; s.t. \; (\ell_2 \sqsubseteq \ell_1)$: Also, $(\ell_2 \sqsubseteq \ell_1 \sqsubseteq \ell)$.

  (1) $k' = \ell_2^\star \wedge k'' = \ell_3^\star$: As $\ell_2 \sqsubseteq \ell$ and $\Gamma(!\rho_n) \not\sqsubseteq \ell$, $\Gamma(!\rho_n) \not\sqsubseteq \ell_2$. By lemma 10, $\ell_3 \sqsubseteq \ell_2$. Thus, $\ell_3 \sqsubseteq \ell_2 \sqsubseteq \ell_1$. By Definition 7(5), $\sigma_0(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$
  (2) $k' = \ell_2^\star \wedge k'' = \ell_3$: As $\ell_2 \sqsubseteq \ell$ and $\Gamma(!\rho_n) \not\sqsubseteq \ell$, $\Gamma(!\rho_n) \not\sqsubseteq \ell_2$. But, by Lemma 9, $\Gamma(!\rho_n) \sqsubseteq \ell_2$. By contradiction, this case does not hold. □

Lemma 12. *Suppose* $\langle \sigma_1, \iota, \rho_1 \rangle \rightarrow \langle \sigma_1', \iota_1', \rho_1' \rangle$, $\langle \sigma_2, \iota, \rho_2 \rangle \rightarrow \langle \sigma_2', \iota_2', \rho_2' \rangle$, $\sigma_1 \sim_\ell \sigma_2$, $\rho_1 \sim_\ell \rho_2$, $\Gamma(!\rho_1) = \Gamma(!\rho_2) \sqsubseteq \ell$, *and either* $\Gamma(!\rho_1') = \Gamma(!\rho_2') \sqsubseteq \ell$) *or* $\Gamma(!\rho_1') \not\sqsubseteq \ell \wedge \Gamma(!\rho_2') \not\sqsubseteq \ell$) *then* $\sigma_1' \sim_\ell \sigma_2'$ *and* $\rho_1' \sim_\ell \rho_2'$.

Proof. Every instruction executes *isIPD* at the end of the operation. If ι_i' is the IPD corresponding to the $!\rho_i.ipd$, then it pops the first node on the pc-stack. As $\rho_1 \sim \rho_2$ and $\Gamma(!\rho_1) = \Gamma(!\rho_2)$, ι_i' would either pop in both the runs or in none. Thus, $\rho_1' sim \rho_2'$ (BRANCH rule is explained below).

Assume $\sigma_1(x) = v_1^{k_1}$, $\sigma_2(x) = v_2^{k_2}$, $\sigma_1'(x) = v_{1'}^{k_1'}$ and $\sigma_2'(x) = v_{2'}^{k_2'}$.
Proof by case analysis on the instruction type:

- ASSN, CATCH: $\Gamma(!\rho_1) = \Gamma(!\rho_2) = pc \sqsubseteq \ell$

  * $pc \sqsubseteq \ell_1 \wedge pc \sqsubseteq \ell_2$: As $\mathtt{n}^m$ is equivalent, $\sigma_1'(x) \sim_\ell \sigma_2'(x)$.
  * $pc \not\sqsubseteq \ell_1 \wedge pc \not\sqsubseteq \ell_2$: By Definition 7(3), $\sigma_1'(x) \sim_\ell \sigma_2'(x)$.
  * $pc \sqsubseteq \ell_1 \wedge pc \not\sqsubseteq \ell_2$: $k_2' \sqsubseteq pc$ and $pc \sqsubseteq k_1'$. By Definition 7(3) and 7.4, $\sigma_1'(x) \sim_\ell \sigma_2'(x)$. Similarly for the analogous case.

- BRANCH: As $\mathtt{b}^{\ell_i}$ is equivalent in the two runs, either $\ell_1 = \ell_2 \sqsubseteq \ell$ or $\ell_1 \not\sqsubseteq \ell \wedge \ell_2 \not\sqsubseteq \ell$ (ℓ_i does not have *). The IPD of ι would be the same in both the cases. If the IPD is SEN, then the label of $!\rho_i$ is joined with the label obtained above, which is either less than or equal to ℓ and same in both the runs (or) not less than or equal to ℓ in both the runs. Thus, either $\Gamma(!\rho_1') = \Gamma(!\rho_2')$ or $\Gamma(!\rho_1') \not\sqsubseteq \ell \wedge \Gamma(!\rho_2') \not\sqsubseteq \ell$. Because $\rho_1 \sim \rho_2$, $\rho_1' \sim_\ell \rho_2'$.
  If the IPD is not SEN, then it is some other node, which makes the *ipd* field the same. Thus, the pushed node is the same in both the cases or has label not less than or equal to ℓ and hence, $\rho_1' \sim_\ell \rho_2'$
  $\sigma_1' = \sigma_1 \sim_\ell \sigma_2 = \sigma_2'$.
- Other rules: $\sigma_1' = \sigma_1 \sim_\ell \sigma_2 = \sigma_2'$. □

Lemma 13. *Suppose*

(1) $\langle \sigma_0', \iota_0, \rho_0' \rangle \rightarrow \langle \sigma_1', \iota_1', \rho_1' \rangle \rightarrow^{n-1} \langle \sigma_n', \iota_n', \rho_n' \rangle$,
(2) $\langle \sigma_0'', \iota_0, \rho_0'' \rangle \rightarrow \langle \sigma_1'', \iota_1'', \rho_1'' \rangle \rightarrow^{m-1} \langle \sigma_m'', \iota_m'', \rho_m'' \rangle$,
(3) $(\rho_0' \sim_\ell \rho_0'')$, $(\sigma_0' \sim_\ell \sigma_0'')$,
(4) $(\Gamma(!\rho_0') = \Gamma(!\rho_0'') \sqsubseteq \ell)$, $(\Gamma(!\rho_n') = \Gamma(!\rho_m'') \sqsubseteq \ell)$,
(5) $\forall (0 < i < n).(\Gamma(!\rho_i') \not\sqsubseteq \ell) \wedge \forall (0 < j < m).(\Gamma(!\rho_j'') \not\sqsubseteq \ell)$,

then $(\iota_n' = \iota_m'')$, $(\rho_n' \sim_\ell \rho_m'')$, *and* $(\sigma_n' \sim_\ell \sigma_m'')$.

Proof. Starting with the same instruction and high context in both the runs can result in two different instructions, ι'_1 and ι''_1. This is only possible if ι was some branching instruction in the first place and this divergence happened in a high context.

(1) To prove $\iota'_n = \iota''_m$:
From the property of the IPDs, if ι_0 pushes a node with label $\not\sqsubseteq \ell$ on top of *pc*-stack which was originally $\sqsubseteq \ell$, $IPD(\iota_0)$ pops that node. Since the runs start from the same instruction ι_0, $\iota'_n = \iota''_m = IPD(\iota)$, where $\Gamma(!\rho) \sqsubseteq \ell$.

(2) To prove $\rho'_n \sim_\ell \rho''_m$:

- $n > 1$ and $m > 1$: $\Gamma(!\rho'_1) \not\sqsubseteq \ell \wedge \Gamma(!\rho''_1) \not\sqsubseteq \ell$, because ι_0 has the same IPD and ι'_1, ι''_1 are not the IPDs. As $\rho'_0 \sim \rho''_0$ and $\Gamma(!\rho'_1) \not\sqsubseteq \ell \wedge \Gamma(!\rho''_1) \not\sqsubseteq \ell$, from Lemma 12, $\rho'_1 \sim \rho''_1$ and $!\rho'_1.ipd = !\rho''_1.ipd = IPD(\iota_0)$, if $\iota'_1 \neq IPD(\iota_0)$ and $\iota''_1 \neq IPD(\iota_0)$. As $\iota'_n = \iota''_m = IPD(\iota_0)$, it pops the $!\rho'_1$ and $!\rho''_1$, which correspond to ρ'_n and ρ''_m in the nth and mth step. Because $\rho'_1 \sim \rho''_1$ and from Lemma 9, $\rho'_n \sim \rho''_m$.
- $n = 1$ and $m > 1$: If $\iota'_1 = IPD(\iota_0)$, and $\Gamma(!\rho'_1) \sqsubseteq \ell$. It pops the node pushed by ι_0, i.e., $\Gamma(!\rho'_n) \sqsubseteq \ell$. In the other run as $\Gamma(!\rho''_1) \not\sqsubseteq \ell$ and $\Gamma(!\rho''_m) \sqsubseteq \ell$, by the property of IPD $\iota''_m = IPD(\iota_0)$, which would pop from the *pc*-stack $!\rho''_m$, the first frame labelled $\not\sqsubseteq \ell$ on the *pc*-stack. Thus, $\rho'_n \sim \rho''_m$.
- $n > 1$ and $m = 1$: Similar to the above case.

(3) To prove $\sigma'_n \sim_\ell \sigma''_m$:

(a) $n > 1$ and $m > 1$: From Lemma 12, $\sigma'_1 \sim_\ell \sigma''_1$. From Lemma 11, $\sigma'_1 \sim_\ell \sigma'_{n-1}$ and $\sigma''_1 \sim_\ell \sigma''_{m-1}$. And from Lemma 8 $\sigma'_{n-1} \sim_\ell \sigma'_n$ and $\sigma''_{m-1} \sim_\ell \sigma''_m$. Similar case analysis as above for different cases of equivalence.

(b) $n = 1$ and $m > 1$: In case of BRANCH, $\sigma'_0 = \sigma'_1$ and $\sigma''_0 = \sigma''_1$. Thus, $\sigma'_1 \sim_\ell \sigma''_1$. From the above case, if $\sigma'_1 \sim_\ell \sigma''_1$, then $\sigma'_n \sim_\ell \sigma''_m$.

(c) $n > 1$ and $m = 1$: Symmetric case of the above. □

Definition 1 (Trace)**.** A trace is defined as a sequence of configurations or states resulting from a program evaluation, i.e., for a program evaluation $\mathcal{P} = s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n$ where $s_i = \langle \sigma_i, \iota_i, \rho_i \rangle$, the corresponding trace is given as $\mathcal{T}(\mathcal{P}) := s_1 :: s_2 :: \ldots :: s_n$.

Definition 2 (Epoch-trace)**.** An epoch-trace for an adversary at level ℓ, ($\mathcal{E}_\ell$) over a trace $\mathcal{T} = s_1 :: s_2 :: \ldots :: s_n$ where $s_i = \langle \sigma_i, \iota_i, \rho_i \rangle$ is defined inductively as:

$$\mathcal{E}_\ell(nil) := nil$$

$$\mathcal{E}_\ell(s_i :: \mathcal{T}) := \begin{cases} s_i :: \mathcal{E}(\mathcal{T}) & \text{if } \Gamma(!\rho_i) \sqsubseteq \ell, \\ \mathcal{E}(\mathcal{T}) & \text{else if } \Gamma(!\rho_i) \not\sqsubseteq \ell. \end{cases}$$

Theorem 3 (Termination-Insensitive Non-interference)**.** *Suppose $\mathcal{P}$ and $\mathcal{P}'$ are two program evaluations.*

Then for their respective epoch-traces with respect to an adversary at level ℓ given by:

$\mathcal{E}_\ell(\mathcal{T}(\mathcal{P})) = s_1 :: s_2 :: \ldots :: s_n$,

$\mathcal{E}_\ell(\mathcal{T}(\mathcal{P}')) = s'_1 :: s'_2 :: \ldots :: s'_m$,

if $s_1 \sim_\ell s'_1$ and $n \leqslant m$,

then

$s_n \sim_\ell s'_n$

Proof. Proof by induction on n.
Basis: $s_1 \sim_\ell s'_1$, by assumption.
IH: $s_k \sim_\ell s'_k$
To prove: $s_{k+1} \sim_\ell s'_{k+1}$.
Let $s_k \rightarrow_i s_{k+1}$ and $s_k \rightarrow_{i'} s'_{k+1}$, then:

- $i = i' = 1$: From Lemma 12, $s_{k+1} \sim_\ell s'_{k+1}$.
- $i > 1$ or $i' > 1$: From Lemma 13, $s_{k+1} \sim_\ell s'_{k+1}$. □

Corollary 4. *Suppose:*

(1) $\langle \sigma_1, \iota_1, \rho_1 \rangle \sim_\ell \langle \sigma_2, \iota_2, \rho_2 \rangle$
(2) $\langle \sigma_1, \iota_1, \rho_1 \rangle \rightarrow^* \langle \sigma'_1, \texttt{end}, [] \rangle$
(3) $\langle \sigma_2, \iota_2, \rho_2 \rangle \rightarrow^* \langle \sigma'_2, \texttt{end}, [] \rangle$

Then, $\sigma'_1 \sim_\ell \sigma'_2$.

Proof. σ_1, σ_2 and ρ_1, ρ_2 are empty at the end of $*$ steps. From the semantics, in context $\sqsubseteq \ell$ both runs would push and pop the same number of nodes. Thus, both take same number of steps in the epoch-trace. Assume it to be k. Then in Theorem 3, $n = m = k$. Thus, $s_k \sim_\ell s'_k$, where $s_k = \langle \sigma'_1, \texttt{end}, [] \rangle$ and $s'_k = \langle \sigma'_2, \texttt{end}, [] \rangle$. By Definition 15, $\sigma'_1 \sim_\ell \sigma'_2$. □

References

[1] Content Security Policy 1.0, http://www.w3.org/TR/CSP/.
[2] M. Algehed and C. Flanagan, Transparent IFC enforcement: Possibility and (in)efficiency results, in: *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, 2020, pp. 65–78. doi:10.1109/CSF49147.2020.00013.
[3] M. Algehed, A. Russo and C. Flanagan, Optimising faceted secure multi-execution, in: *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, 2019, pp. 1–16.
[4] A. Askarov, S. Hunt, A. Sabelfeld and D. Sands, Termination-insensitive noninterference leaks more than just a bit, in: *Proc. European Symposium on Research in Computer Security*, 2008, pp. 333–348.
[5] A. Askarov and A. Sabelfeld, Tight enforcement of information-release policies for dynamic languages, in: *Proc. IEEE Computer Security Foundations Symposium*, 2009, pp. 43–59.
[6] A. Askarov and A. Sabelfeld, Catch me if you can: Permissive yet secure error handling, in: *Proc. ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, 2009, pp. 45–57. ISBN 978-1-60558-645-8. doi:10.1145/1554339.1554346.
[7] T.H. Austin and C. Flanagan, Efficient purely-dynamic information flow analysis, in: *Proc. ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, 2009, pp. 113–124. ISBN 978-1-60558-645-8. doi:10.1145/1554339.1554353.
[8] T.H. Austin and C. Flanagan, Permissive dynamic information flow analysis, in: *Proc. 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2010, pp. 3:1–3:12. ISBN 978-1-60558-827-8. doi:10.1145/1814217.1814220.
[9] T.H. Austin and C. Flanagan, Multiple facets for dynamic information flow, in: *Proc. 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012, pp. 165–178. ISBN 978-1-4503-1083-3. doi:10.1145/2103656.2103677.
[10] T.H. Austin, T. Schmitz and C. Flanagan, Multiple facets for dynamic information flow with exceptions, *ACM Trans. Program. Lang. Syst.* **39**(3) (2017). doi:10.1145/3024086.
[11] A. Barth, The web origin concept, http://tools.ietf.org/html/rfc6454.

[12] A. Bedford, S. Chong, J. Desharnais, E. Kozyri and N. Tawbi, A progress-sensitive flow-sensitive inlined information-flow control monitor, *Computers & Security* **71** (2017), 114–131. doi:10.1016/j.cose.2017.04.001.
[13] A. Bichhawat, V. Rajani, D. Garg and C. Hammer, Generalizing permissive-upgrade in dynamic information flow analysis, in: *Proc. Workshop on Programming Languages and Analysis for Security*, 2014, pp. 15–24. doi:10.1145/2637113.2637116.
[14] A. Bichhawat, V. Rajani, D. Garg and C. Hammer, Information flow control in WebKit's JavaScript bytecode, in: *Proc. Principles of Security and Trust*, 2014, pp. 159–178. doi:10.1007/978-3-642-54792-8_9.
[15] A. Birgisson, D. Hedin and A. Sabelfeld, Boosting the permissiveness of dynamic information-flow tracking by testing, in: *Computer Security – ESORICS 2012*, LNCS, Vol. 7459, Springer, Berlin Heidelberg, 2012, pp. 55–72. ISBN 978-3-642-33166-4. doi:10.1007/978-3-642-33167-1_4.
[16] P. Buiras, D. Stefan and A. Russo, On dynamic flow-sensitive floating-label systems, in: *Proc. 2014 IEEE 27th Computer Security Foundations Symposium, CSF'14*, IEEE Computer Society, 2014, pp. 65–79. doi:10.1109/CSF.2014.13.
[17] R. Chugh, J.A. Meister, R. Jhala and S. Lerner, Staged information flow for JavaScript, in: *Proc. 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 50–62. doi:10.1145/1542476.1542483.
[18] D. Crockford, ADSafe, http://adsafe.org/.
[19] A.A. de Amorim, M. Fredrikson and L. Jia, Reconciling noninterference and gradual typing, in: *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS'20*, Association for Computing Machinery, New York, NY, USA, 2020, pp. 116–129. ISBN 9781450371049. doi:10.1145/3373718.3394778.
[20] D.E. Denning, A lattice model of secure information flow, *Commun. ACM* **19**(5) (1976), 236–243. doi:10.1145/360051.360056.
[21] D.E. Denning and P.J. Denning, Certification of programs for secure information flow, *Commun. ACM* **20**(7) (1977), 504–513. doi:10.1145/359636.359712.
[22] D.E.R. Denning, *Cryptography and Data Security*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982. ISBN 0-201-10150-5.
[23] D. Devriese and F. Piessens, Noninterference through secure multi-execution, in: *Proc. 2010 IEEE Symposium on Security and Privacy*, 2010, pp. 109–124. ISBN 978-0-7695-4035-1. doi:10.1109/SP.2010.15.
[24] M. Dhawan and V. Ganapathy, Analyzing information flow in JavaScript-based browser extensions, in: *Proc. 2009 Annual Computer Security Applications Conference, ACSAC'09*, 2009, pp. 382–391. ISBN 978-0-7695-3919-5. doi:10.1109/ACSAC.2009.43.
[25] T. Disney and C. Flanagan, Gradual information flow typing, in: *Proceedings of the 2nd International Workshop on Scripts to Programs Evolution, STOP'11*, 2011.
[26] X. Dong, Z. Chen, H. Siadati, S. Tople, P. Saxena and Z. Liang, Protecting sensitive web content from client-side vulnerabilities with CRYPTONS, in: *Proc. 2013 ACM SIGSAC Conference on Computer and Communications Security*, 2013, pp. 1311–1324.
[27] Facebook. FBJS, https://developers.facebook.com/docs/javascript.
[28] L. Fennell and P. Thiemann, Gradual security typing with references, in: *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium, CSF'13*, IEEE Computer Society, Washington, DC, USA, 2013, pp. 224–239. ISBN 978-0-7695-5031-2. doi:10.1109/CSF.2013.22.
[29] L. Fennell and P. Thiemann, LJGS: Gradual security types for object-oriented languages, in: *30th European Conference on Object-Oriented Programming, ECOOP*, 2016, 2016, pp. 9:1–9:26.
[30] J.S. Fenton, Memoryless subsystems, *The Computer Journal* **17**(2) (1974), 143. doi:10.1093/comjnl/17.2.143.
[31] J.A. Goguen and J. Meseguer, Security policies and security models, in: *Proc. 1982 IEEE Symposium on Security and Privacy*, 1982, pp. 11–20. doi:10.1109/SP.1982.10014.
[32] Google Caja – A source-to-source translator for securing JavaScript-based web content, Online; accessed 25-Apr-2017.
[33] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet and R. Berg, Saving the world wide web from vulnerable JavaScript, in: *Proc. 2011 International Symposium on Software Testing and Analysis, ISSTA'11*, 2011, pp. 177–187. ISBN 978-1-4503-0562-4. doi:10.1145/2001420.2001442.
[34] G.L. Guernic, A. Banerjee, T. Jensen and D.A. Schmidt, Automata-based confidentiality monitoring, in: *Proc. Asian Computing Science Conference on Secure Software*, 2006, pp. 75–89.
[35] C. Hammer and G. Snelting, Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs, *International Journal of Information Security* **8**(6) (2009), 399–422. doi:10.1007/s10207-009-0086-1.
[36] D. Hedin, L. Bello and A. Sabelfeld, Value-sensitive hybrid information flow control for a JavaScript-like language, in: *Proc. 2015 IEEE 28th Computer Security Foundations Symposium, CSF'15*, 2015, pp. 351–365. doi:10.1109/CSF.2015.31.
[37] D. Hedin, A. Birgisson, L. Bello and A. Sabelfeld, JSFlow: Tracking information flow in JavaScript and its APIs, in: *Proc. ACM Symposium on Applied Computing*, 2014, pp. 1663–1671. doi:10.1145/2554850.2554909.

[38] D. Hedin and A. Sabelfeld, Information-flow security for a core of JavaScript, in: *Proc. 25th IEEE Computer Security Foundations Symposium*, 2012, pp. 3–18. ISBN 978-0-7695-4718-3. doi:10.1109/CSF.2012.19.
[39] C. Hritcu, M. Greenberg, B. Karel, B.C. Pierce and G. Morrisett, All your IFCException are belong to us, in: *Proceedings of the 2013 IEEE Symposium on Security and Privacy, SP'13*, IEEE Computer Society, USA, 2013, pp. 3–17. ISBN 9780769549774. doi:10.1109/SP.2013.10.
[40] S. Hunt and D. Sands, On flow-sensitive security types, in: *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006, pp. 79–90.
[41] D. Jang, R. Jhala, S. Lerner and H. Shacham, An empirical study of privacy-violating information flows in JavaScript web applications, in: *Proc. 17th ACM Conference on Computer and Communications Security*, 2010, pp. 270–283.
[42] S. Just, A. Cleary, B. Shirley and C. Hammer, Information flow analysis for JavaScript, in: *Proc. 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*, 2011, pp. 9–18. ISBN 978-1-4503-1171-7. doi:10.1145/2093328.2093331.
[43] G. Le Guernic, Automaton-based confidentiality monitoring of concurrent programs, in: *Proc. IEEE Computer Security Foundations Symposium*, 2007, pp. 218–232.
[44] T. Lengauer and R.E. Tarjan, A fast algorithm for finding dominators in a flowgraph, *ACM Trans. Program. Lang. Syst.* **1**(1) (1979), 121–141. doi:10.1145/357062.357071.
[45] M.T. Louw, K.T. Ganesh and V.N. Venkatakrishnan, AdJail: Practical enforcement of confidentiality and integrity policies on web advertisements, in: *Proc. 19th USENIX Conference on Security*, 2010, pp. 24–24.
[46] W. Masri and A. Podgurski, Algorithms and tool support for dynamic information flow analysis, *Information & Software Technology* **51**(2) (2009), 385–404. doi:10.1016/j.infsof.2008.05.008.
[47] L.A. Meyerovich and B. Livshits, ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser, in: *Proc. 2010 IEEE Symposium on Security and Privacy*, 2010, pp. 481–496. doi:10.1109/SP.2010.36.
[48] A.C. Myers, JFlow: Practical mostly-static information flow control, in: *Proc. 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'99*, 1999, pp. 228–241.
[49] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel and G. Vigna, Cross-site scripting prevention with dynamic data tainting and static analysis, in: *Proc. Network and Distributed System Security Symposium*, 2007.
[50] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens and G. Vigna, You are what you include: Large-scale evaluation of remote Javascript inclusions, in: *Proc. 2012 ACM Conference on Computer and Communications Security, CCS'12*, 2012, pp. 736–747.
[51] F. Pottier and V. Simonet, Information flow inference for ML, *ACM Trans. Program. Lang. Syst.* **25**(1) (2003), 117–158. doi:10.1145/596980.596983.
[52] W. Rafnsson and A. Sabelfeld, Secure multi-execution: Fine-grained, declassification-aware, and transparent, in: *Proc. 2013 IEEE 26th Computer Security Foundations Symposium*, 2013, pp. 33–48. doi:10.1109/CSF.2013.10.
[53] G. Richards, C. Hammer, B. Burg and J. Vitek, The eval that men do – a large-scale study of the use of eval in JavaScript applications, in: *ECOOP'11*, M. Mezzini, ed., LNCS, Vol. 6813, 2011, pp. 52–78. ISBN 978-3-642-22654-0.
[54] G. Richards, C. Hammer, F. Zappa Nardelli, S. Jagannathan and J. Vitek, Flexible access control for Javascript, in: *Proc. 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'13*, 2013, pp. 305–322. ISBN 978-1-4503-2374-1. doi:10.1145/2509136.2509542.
[55] A. Russo and A. Sabelfeld, Dynamic vs. static flow-sensitive security analysis, in: *Proc. 2010 IEEE 23rd Computer Security Foundations Symposium*, 2010, pp. 186–199. doi:10.1109/CSF.2010.20.
[56] A. Sabelfeld and A.C. Myers, Language-based information-flow security, *IEEE Journal on Selected Areas in Communications* **21** (2003), 5–19. doi:10.1109/JSAC.2002.806121.
[57] A. Sabelfeld and A. Russo, From dynamic to static and back: Riding the roller coaster of information-flow control research, in: *Proc. Perspectives of Systems Informatics*, 2010, pp. 352–365. doi:10.1007/978-3-642-11486-1_30.
[58] T. Schmitz, M. Algehed, C. Flanagan and A. Russo, Faceted secure multi execution, in: *ACM CCS*, 2018, pp. 1617–1634. ISBN 9781450356930. doi:10.1145/3243734.3243806.
[59] A.L. Scull Pupo, L. Christophe, J. Nicolay, C. de Roover and E. Gonzalez, *Boix, Practical Information Flow Control for Web Applications*, R. Verification, C. Colombo and M. Leucker, eds, Springer International Publishing, Cham, 2018, pp. 372–388. ISBN 978-3-030-03769-7.
[60] D. Stefan, D. Mazières, J.C. Mitchell and A. Russo, Flexible dynamic information flow control in the presence of exceptions, *J. Funct. Program.* **27** (2017), e5. doi:10.1017/S0956796816000241.
[61] D. Stefan, E.Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp and D. Mazières, Protecting users by confining JavaScript with COWL, in: *Proc. USENIX Symposium on Operating Systems Design and Implementation*, 2014, pp. 131–146.
[62] M. Toro, R. Garcia and E. Tanter, Type-driven gradual security with references, *ACM Trans. Program. Lang. Syst.* **40**(4) (2018), 16:1–16:55. doi:10.1145/3229061.
[63] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens and W. Joosen, WebJail: Least-privilege integration of third-party components in web mashups, in: *Proc. 27th Annual Computer Security Applications Conference*, 2011, pp. 307–316.

[64] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel and G. Vigna, Cross site scripting prevention with dynamic data tainting and static analysis, in: *Proceeding of the Network and Distributed System Security Symposium*, 2007, https://www.isoc.org/isoc/conferences/ndss/07/papers/cross-site-scripting_prevention.pdf.
[65] D. Volpano, C. Irvine and G. Smith, A sound type system for secure flow analysis, *J. Comput. Secur.* **4**(2–3) (1996), 167–187. doi:10.3233/JCS-1996-42-304.
[66] B. Xin and X. Zhang, Efficient online detection of dynamic control dependence, in: *Proc. 2007 International Symposium on Software Testing and Analysis*, 2007, pp. 185–195. ISBN 978-1-59593-734-6. doi:10.1145/1273463.1273489.
[67] J. Yang, K. Yessenov and A. Solar-Lezama, A language for automatically enforcing privacy policies, in: *Proc. 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'12*, 2012, pp. 85–96.
[68] S.A. Zdancewic, Programming Languages for Information Security, PhD thesis, Cornell University, 2002.
[69] Y. Zhou and D. Evans, Protecting private web content from embedded scripts, in: *Proc. 16th European Conference on Research in Computer Security*, 2011, pp. 60–79.