Practical Dynamic Information Flow Control

Abhishek Bichhawat

A dissertation submitted towards the degree
Doctor of Engineering
of the Faculty of Mathematics and Computer Science
of Saarland University

by
Abhishek Bichhawat

Saarbrücken, den some very lucky day in the future

Dekan: Dekan

Erstgutachter: first advisor

Zweitgutachter: second reviewer

Drittgutachter: possibly another reviewer

Tag der mündlichen Prüfung: some very lucky day in the future

Abstract

Over the years, computer systems and applications have grown significantly complex while handling a plethora of private and sensitive user information. The complexity of these applications is often assisted by a set of (un)intentional bugs with both malicious and non-malicious intent leading to information leaks. Information flow control has been studied extensively as an approach to mitigate such information leaks. The technique works by enforcing the security property of non-interference using a specified set of security policies. A vast majority of existing work in this area is based on static analyses. However, some of these applications, especially on the Web, are developed using dynamic languages like JavaScript that makes the static analyses techniques stale and ineffective. As a result, there has been a growing interest in recent years to develop dynamic information flow analysis techniques. In spite of that, dynamic information flow analysis has not been at the helm of information flow security in settings like the Web; the prime reason being that the analysis techniques and the security property related to them (non-interference) either over-approximate or are too restrictive in most cases. Concretely, the analysis techniques generate a lot of false positives, do not allow legitimate release of sensitive information, support only static and rigid security policies or are not general enough to be applied to real-world applications.

This thesis focuses on improving the usability of dynamic information flow techniques by presenting mechanisms that can enhance the precision and permissiveness of the analyses. It begins by presenting a sound improvement and enhancement of the permissive-upgrade strategy (a strategy widely used to enforce dynamic information flow control), which improves the strategy's permissiveness and makes it generic in applicability. The thesis, then, presents a sound and precise dynamic information flow analysis for handling complex features like unstructured control flow and exceptions in higher-order languages. Although non-interference is a desired property for enforcing information flow control, there are program instances that require legitimate release of some parts of the secret data to provide the required functionality. Towards this end, this thesis develops a sound approach to quantify information leaks dynamically while allowing information release in accordance to a pre-specified budget. The thesis concludes by applying these techniques to an information flow control-enabled Web browser and explores a policy specification mechanism that allows flexible and useful information flow policies to be specified for Web applications.

Kurzzusammenfassung

short summary in German, one page.

Acknowledgment

General acknowledgment goes here...

Co-authoring

Contents

Part I

Introduction and Background

Chapter 1

Introduction

With the growth in the use of computers and the Internet for almost every application, the amount of sensitive information that programs compute has also increased dramatically. For instance, e-commerce websites have access to the personal details of a user including her address, credit-card details etc. Similarly, a hospital database stores sensitive and private medical data of its patients. While the complexity of such systems has increased manifold in the last few decades, the privacy and confidentiality guarantees still remain questionable [8]. Often leaks occur either due to buggy programs or due to malicious code. Cryptographic techniques protect data by encrypting it but many programs need to operate on confidential data, which requires the data to be available in plaintext form. To this end, *access control* techniques are widely used for preventing leakage of confidential data. However, once the authorized users have access to data, there is no control on how the data can be used. The data might be input to a program that writes to publicly-observable locations or outputs data that is accessible to all users.

Consider the Web, for instance. Web applications rely extensively on third-party JavaScript to provide useful libraries, page analytics, advertisements and many other features [81]. JavaScript works on a mashup model, wherein the hosting page and included scripts share the page's state (called the DOM). Consequently, by design, 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 webpage providers and users. This leaves 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 [63]. The traditional browser security model is based on restricting scripts' access to data, not on tracking how scripts use data. Existing web security standards and web browsers address this problem unsatisfactorily, favoring functionality over privacy. The

same-origin policy implemented in all major browsers restricts a webpage and third-party scripts included in it to communicating with web servers from the including webpage'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. The candidate web standard Content Security Policy [4], also implemented in most browsers, allows a page to white list scripts that may be included, but places no restriction on scripts that have been included, thus not helping with the problem above. Quite a few *fine-grained* access control techniques have also been proposed [2, 6, 41, 48, 72, 77, 99, 108]. However, all these techniques enforce only access policies and cannot control what a script does with data it has been provided in good faith, i.e., if a third-party script was allowed access to some data only for local computations, the policy places no restriction on the script for sending it on the network while performing the computation. More broadly, no mechanism based only on access control can solve the problem of information leakage.

The academic community has proposed solutions based on *information flow control* (IFC), also known as mandatory access control. It 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 [43, 44, 56, 62, 79, 82, 91, 103], dynamic methods that track information flow through program execution at runtime [13, 16–18, 24, 46, 49, 57, 93, 98, 105], and hybrid approaches that combine both static and dynamic analyses to add precision to the analysis [25, 54, 55, 60, 80, 89, 101] for handling information leaks described above.

Dynamic analysis has the drawback of being less precise and introducing significant performance overheads at runtime, though it can be more permissive than static analysis methods in some cases [89]. Although helpful in most scenarios, 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 [86, 87] with features like dynamic typing, `eval`, scope-chains, prototype chains etc. makes sound static analysis difficult. Thus, recent research has focussed on dynamic analysis for enforcing information flow control especially in languages like JavaScript. Even though research in dynamic information flow control has made significant inroads in the last decade, the applicability of these techniques still remains bleak, *permissiveness* being the major challenge to the practicality of dynamic information flow control.

In recent years, researchers have explored methods to quantify the amount of sensitive information leaked by a program as an alternative to the qualitative notion of information flow control. The developer determines an "acceptable" upper-bound on the number of bits of sensitive information a program may leak. Programs that leak less than this "acceptable" amount of information are considered secure. The information released is generally quantified as the knowledge gained by an adversary about the sensitive data as a result of the information flow in the program. Specifically, the adversary has knowledge about the possible set of initial values that can be associated with the sensitive data. A flow in the program could reduce the number of possibilities, thereby increasing the knowledge of the adversary about the specific value associated with the sensitive data in that execution of the program. This change in knowledge can be measured as the number of bits of in-

formation of sensitive data released to the adversary [40]. Statically quantifying the information leaked by a program has been well studied in the literature and various static quantitative information flow measures [10, 37, 75, 96, 97] and techniques [19, 36, 37, 45, 68, 97] have been proposed. However, quantifying information leaks in a purely dynamic setting is still largely an open problem (prior work is limited to [76]) as dynamic analysis is generally limited to the current execution of the program in contrast to the static methods, which analyze the program as a whole.

1.1 Contributions of the Thesis

This thesis focuses on improving the usability of dynamic information flow techniques by presenting mechanisms that enhance the precision and permissiveness of the analyses. The main contributions are as follows:

- **Generalized Permissive-Upgrade.** To improve the permissiveness of dynamic information flow analysis, the thesis presents a sound improvement and enhancement of the permissive-upgrade strategy (a strategy widely used to enforce dynamic information flow control). 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.

- **Precise Dynamic Information Flow Control.** Most of the existing work in dynamic information flow control does not handle complex features like unstructured control flow and exceptions. The proposals that handle these features are too conservative and require additional annotations in the program. This thesis presents a sound and precise dynamic information flow analysis for handling these features without requiring any additional annotations from the programmer.

- **Bounding Information Leaks Dynamically.** Although non-interference is a desired property for enforcing information flow control, there are program instances that require legitimate release of some parts of the secret data to provide the required functionality. Towards this end, this thesis develops a sound approach to quantify information leaks dynamically while allowing information release in accordance with a pre-specified *budget*.

- **Application to a Web Browser.** The thesis concludes by applying these techniques to an information flow control-enabled Web browser. An information flow control system enforces security policies that are a collection of rules for labeling private information sources, generally specified by a policy component in the system. To complement the work on enforcement components in Web browsers, this thesis explores a policy specification mechanism to specify flexible and useful information flow policies for Web applications.

Chapter 2

Background and Overview

2.1 Information Flow Control

Information flow control (IFC) refers to controlling 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 thesis 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 to flow to public outputs, the adversary does not get any information about the private or sensitive data.

The seminal work by Denning [43–45] defined most of the theoretical ideas pertaining to information flow control. In general, information can flow along many channels. However, this thesis considers two of the most important flows — *explicit* and *implicit* — in deterministic programs. Covert channels like timing or resource usage are beyond the scope of this thesis.

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 2.1. 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 [15].

The correctness of the approaches enforcing information flow control is often stated in terms of a well-defined property known as *non-interference* [52], which basically stipulates that high input of a program must not influence its low output. While non-interference is too strong a property in practice, different variants of the definition are proven. One such variant of non-interference usually

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

Listing 2.1: Implicit flow from z to y

established for information flow control techniques is *termination-insensitive non-interference* [103]. Roughly, a program is termination-insensitive non-interferent if any two terminating 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. Termination-insensitive means that one-bit of leak is tolerable when an adversary checks whether or not the program terminated. In particular, this discounted one-bit leak accounts for termination due to the failure of a runtime security check. Askarov et al. [15] show that for programs with intermediate observable outputs, termination-insensitivity may leak more than one bit but also show that this attack is limited to a brute-force attack.

Information flow control approaches either statically analyze the program at compile-time and reject or accept a program based on the security policy, or dynamically analyze the flow of data in a program by either using a modified runtime or with the help of a reference monitor, or employ a combination of both the approaches to shrug off some of the individual limitations of the two approaches. Static approaches normally support restricted policies, and are impermissive in nature. Besides, it becomes extremely complicated to use such methods with dynamically-typed languages and languages that are loaded with dynamic features. Dynamic approaches, on the other hand, add runtime overheads and are less precise as compared to the static approaches. Dynamic information flow control, which is the central theme of this thesis, is described in detail later in the chapter.

2.1.1 Flow-sensitivity

Static analysis techniques are usually flow-insensitive in nature, i.e., they do not account for the ordering of the instructions in the program or the general flow of execution of the program. In other words, all operations should be individually secure to secure the program as a whole. For instance, consider the program snippet:

$$\texttt{l} = \texttt{h}$$

Assume that h is a secret variable labeled H and l is a public variable labeled L. This program gets rejected by a flow-insensitive analysis as the assignment of a secret value to a public variable is considered insecure.

Flow-sensitive analysis improves the permissiveness of static analysis techniques and is mostly used in dynamic analyses. Under a flow-sensitive static analysis, the above program upgrades the labels of l to H indicating the influence of H-labeled h on l. If the program was to output the value of l on a L-observable channel later, the program gets rejected by the analysis. Flow-sensitive analysis also accepts programs with dead-code like the one shown in Listing 2.2, thus improving the

```
1 var x
2 if (not(z))
3   x = true
4 x = false
```

Listing 2.2: Permissiveness of flow-sensitive analysis

permissiveness of the analysis. Some flow-sensitive static analyses have been proposed to enforce information flow control [56, 62] while almost all dynamic approaches are flow-sensitive in nature.

2.2 Information Release

Non-interference prevents all unauthorized flows without regarding the severity of the leaks. However, many practical applications require some fragment of the sensitive data to be released to programs for providing proper functionality or to improve the user-experience. For instance, according to non-interference even a standard login application would be considered insecure because it leaks information about the user's password, which is considered confidential, by either allowing or denying access. As a remedy, researchers have proposed various ways to intentionally release or *declassify* sensitive data through relaxations of non-interference [95], which is generally enforced using annotations like `declassify` in the program. For instance, `declassify(pwd == input)` would treat the result of the password check as public, thus allowing access-information to be released to the user. Most of these approaches require policy specifications by the developer that define what information can be released by the program as in many settings it is difficult to trust third-parties to include correct and appropriate (declassify) annotations for information release in the code. Declassification approaches generally ensure that the adversary is unable to declassify or force declassification of sensitive information as per his/her needs. Sabelfeld and Sands [95] present a survey of many such techniques that allow release of information in a secure manner.

Some of these declassification techniques are described below. Cohen [38, 39] in his work on selective dependency used assertions to eliminate certain information paths thereby allowing programs that satisfy the constraint to be accepted. The idea was that information flows to the adversary should not allow her to deduce more information about secrets than is allowed by the assertions. Volpano and Smith [102] present relative secrecy, a property that instead of providing absolute secrecy allows information to be released through specific *match* queries relative to the size of the secret, i.e., if the adversary cannot guess the secret in polynomial time. Giacobazzi and Mastroeni [51] generalize non-interference by modelling what property the adversary can observe about the secrets. Li and Zdancewic's relaxed non-interference characterizes the information released through downgrading policies [70]. Sabelfeld and Sands [94] proposed the PER model using partial equivalence relations for specifying information flow policies. Sabelfeld and Myers [92] introduced the notion of delimited release, which enables the specification of declassification policy using *escape hatches*. The policy specifies upfront what information can be released through these escape hatches. Localized delimited release [12] requires that an expression is only released after a declassification operation for this expression has appeared in the code. Lux and Mantel [73] propose explicit reference points

to allow flexible specification of what secrets may be declassified at the specific reference points. Robust declassification [106] ensures that only trusted code is allowed to declassify information. Chong and Myers [33] propose a framework for specifying application-specific information release policies. Chong [32] further proposes the concept of required information release, which considers applications that are obligated to release information under certain conditions. Askarov and Sabelfeld introduce the concept of gradual release [11] for allowing release of information at certain release or declassification points. The policy does not allow the adversary knowledge to refine its knowledge of secrets at points other than the release points. Banerjee et al. [20] present a way to specify declassification policies that satisfy conditioned gradual release, which is an extension of the gradual release property. Similarly, another security property, *policy controlled release*, that extends gradual release was presented by Rocha et al. [88] for declassification in untrusted and legacy programs.

2.3 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}.

Flow-sensitive 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 individual labels on `y` and `z`, which is the final label of `x`, i.e., if either of `y` or `z` is labeled confidential or H, then the final label of `x` is also labeled H[1].

Implicit flows in a flow-sensitive 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 2.1, 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 2.1 is a classic counterexample, taken from [16]. 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

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

about z despite correct use of pc.

2.3.1 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 [107]. In the program in Listing 2.1, 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 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 [16].

2.3.2 Permissive-Upgrade and Faceted Execution

To tackle the issue of permissiveness with the no-sensitive-upgrade strategy, Austin and Flanagan proposed the permissive-upgrade strategy [17] and faceted execution [18]; faceted execution being the most permissive of the three. Faceted execution draws inspiration from secure multi-execution and simulates multiple executions simultaneously within a single runtime. They introduce the concept of faceted values that are pairs of values for both low and high observers. With multiple levels, each value in the pair is represented as a pair. When branching on a faceted value, multiple executions are simulated for the different values in the facet. Although the approach incurs lesser overhead when compared to secure multi-execution, the overheads are still quite prohibitive for multiple security levels. This thesis, thus, considers only the permissive-upgrade strategy, which is more permissive than the no-sensitive-upgrade strategy and much less performance-intensive compared to faceted execution. Permissive-upgrade is described in detail in Section 3.2.

2.3.3 Other Approaches for Permissiveness

Secure multi-execution [46] is another approach for enforcing non-interference at runtime. Instead of tracking information flow through the program, the approach executes multiple copies of the program with different values of sensitive data. Conceptually, one executes the same code once for each security level (like low and high) with a few constraints. The private data in the low execution are replaced by default values, i.e., the public copy of the program does not see the actual value of the private data but a pre-determined default value, and outputs on an ℓ-labeled channel are permitted only in the ℓ-level execution of the program, i.e., the high execution of the program outputs on the high channel, if any, and the low execution of the program outputs on the low channel, typically the network. The modification of the semantics forces that private data and the outputs resulting from it are visible to only high-level observers. The public-level observers or the adversary observe inaccurate results for the outputs that depend on the private value as the value is replaced by the default value in that execution. This modification forces even unsafe programs to adhere to non-interference. Additionally, secure multi-execution guarantees precision, i.e., the semantics of a

secure program is not altered. Thus, for a secure program the outputs are all accurate. Secure multi-execution normally guarantees termination-insensitive non-interference as the high execution may not terminate in some cases. FlowFox [42] demonstrates secure multi-execution in the context of web browsers. However, executing a program multiple times can be prohibitive for a security lattice with multiple levels [18]. The runtime overhead incurred can be reduced if the executions are run in parallel (which requires more hardware resources), though the program has to be run for all levels irrespective of whether it uses private data or not. In addition to this, secure multi-execution makes declassification complicated as it requires synchronization between different executions [84].

Birgisson *et al.* [30] 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 [57]. Hritcu *et al.* improve permissiveness by making IFC errors recoverable in the language Breeze [61]. This is accomplished by a combination of two methods: making all labels public (by upgrading them when necessary in a public pc) and by delaying exceptions. 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 in recent work by Buiras *et al.* in the context of a language with a dedicated monad for tracking information flows [31]. 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 thesis confirms the latter and shows how it can be done.

Part II

Generalized Permissive-Upgrade Strategy

Chapter 3

Improved Permissive-Upgrade

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 3.1, 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 $\texttt{y} \mapsto \texttt{true}^L$, 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.)

To improve permissiveness, Austin and Flanagan [17] proposed the permissive-upgrade strategy as a replacement for NSU. However, that development lacks permissiveness in certain cases. This chapter presents the soundness results of the permissive-upgrade strategy with the improvement for further permissiveness in place.

3.1 Overview

This section presents a *formal* description of the no-sensitive-upgrade check. The technical development in this thesis is mostly based on the simple imperative language shown in Figure 3.1. However, the key ideas are orthogonal to the choice of language and generalize to other languages easily. The

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

Listing 3.1: Impermissiveness of the NSU strategy

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

Figure 3.1: Syntax of the Language

EXPRESSIONS:

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

STATEMENTS:

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

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

Figure 3.2: Semantics

use of a simpler language is to simplify non-essential technical details. The parts in this thesis that require a more complex language define the additional features in place. The language's expressions include constants or values (n), 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. The lattice contains different labels $\{L, M, H, \ldots\}$ with a partial ordering between the elements. Join ($\sqcup$) and meet ($\sqcap$) operations are defined as usual on the lattice. The program counter label pc is an element of the lattice.

3.1.1 Basic IFC Semantics

The rules in Figure 3.2 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

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

Figure 3.3: Assignment rule for NSU

form $\mathtt{n}^k$. $\mathtt{b}$ represents a Boolean constant. For now, labels $k ::= \ell$; this is generalized later when the "partially-leaked" taints are introduced in Section 3.2.

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 $\Gamma(\sigma(\mathtt{x})) = k$. $\bot$ denotes the least element of the lattice.

The rule for sequencing $c_1; c_2$ (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 rule for `if-else` (IF-ELSE) 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 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 Figure 3.2 because they depend on the strategy used to control implicit flows. The rule for assignment (ASSN-NSU) corresponding to the NSU check is shown in Figure 3.3. 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. This is exactly the NSU check described in Section 2.3.1.

3.1.2 Formalization of 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 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 $\mathtt{n}_2^m$ are ℓ-equivalent, written $\mathtt{n}_1^k \sim_\ell \mathtt{n}_2^m$, iff either:*

1. *$(k = m) \sqsubseteq \ell$ and $\mathtt{n}_1 = \mathtt{n}_2$ 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 $m \not\sqsubseteq \ell$). The additional constraint $k = m$ in clause (1) is needed to prove non-interference by induction. In the lattice $L \sqsubset H$, 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* $\mathtt{x}$, $\sigma_1(\mathtt{x}) \sim_\ell \sigma_2(\mathtt{x})$.

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 Figure 3.3, 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., [16] □

3.2 Austin and Flanagan's Permissive-Upgrade 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* [17]. 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 2.1 at line 4 when `z` contains $\mathtt{false}^H$, but it allows the program of Listing 3.1 to execute to completion when `y` contains $\mathtt{true}^L$.

In the revised syntax of labels, summarized in Figure 3.4, 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, they extend the order $\sqsubset$ to $L \sqsubset H \sqsubset P$. However, P is not a new "top" member of the lattice because it receives special treatment in the semantic rules.

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

Figure 3.4: Syntax of labels including the partially-leaked label P

The rule for assignment with permissive-upgrade is

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

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 (Figure 3.3). The third condition applies, in particular, when a variable whose initial label is L is assigned with $pc = H$. The NSU 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` (Figure 3.2), 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 $\mathtt{n}_1^k$ and $\mathtt{n}_2^m$ are equivalent to an adversary at level L, written $\mathtt{n}_1^k \sim_L \mathtt{n}_2^m$, iff either:*

1. *$(k = m) = L$ and $\mathtt{n}_1 = \mathtt{n}_2$ 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 $\forall \mathtt{x}.\sigma_1(\mathtt{x}) \sim_L \sigma_2(\mathtt{x})$.*

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 [17]. □

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

Listing 3.2: Example showing the impermissiveness of the original permissive-upgrade strategy

3.3 Improved Permissive-Upgrade Strategy

The original permissive-upgrade strategy, however, lacks permissiveness; it rejects secure programs like the one shown in Listing 3.2. Consider that x is labeled H and w, y are labeled L. With the original permissive-upgrade strategy, the label of z on line 4 would remain P and the execution would be terminated when branching on z on line 5. The improvement

$$H \sqcup P = H$$

allows the analysis to accept such programs while remaining sound. With the improvement, z would be labeled H on line 4, which would allow the execution to branch on line 5, thus, taking the execution to completion. The idea behind the improvement is 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, any operation involving a partially-leaked value and a H-labeled value does not reveal any information to an adversary at level L about the partially leaked value.

The final label k in the assignment rule ASSN-PUS under the improved permissive-upgrade strategy becomes:

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

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 the theorem using the modified notation for the imperative language is that the definition of $\sim$ is not transitive. The same problem arises for the soundness proofs in [17]. There, the authors resolve the issue by defining a special relation called evolution. The need for evolution is averted here using the auxiliary lemmas listed below. Lemma 2 proves the required result substituting evolution.

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* $\sigma_1 \sim_L \sigma_2$*, then* $\mathtt{n}_1^{k_1} \sim_L \mathtt{n}_2^{k_2}$*.*

Proof. By induction on e. □

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

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 derivation rules. □

Theorem 3 (TINI for improved permissive-upgrade with a two-point lattice)**.** *With the assignment rule* ASSN-PUS *and the modified syntax of Figure 3.4, 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. □

The detailed proofs are provided in Appendix A.1. Note that the definitions and proofs presented in this chapter are specific to the two-point lattice and with respect to an adversary at level L.

Chapter 4

Generalized Permissive-Upgrade

Although the permissive-upgrade strategy as described above is useful, its development in literature is incomplete so far: Austin and Flanagan's original paper [17], and the work building on it (Chapter 3), develops 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 chapter shows by construction that a generalization of permissive-upgrade 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 Generalization of the Improved Permissive-Upgrade Strategy

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 in the previous chapter. Specifically, let X be an index set — these indices are called principals in [17]. 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

The content of this chapter is based on the work published as part of the paper, "Generalizing Permissive-Upgrade in Dynamic Information Flow Analysis" [24]

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

Figure 4.1: Labels and label operations

by replacing it with the following rule:

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

where k is defined as follows:

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

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* $\mathtt{n}_1^k$ *and* $\mathtt{n}_2^m$ *are* a*-equivalent, written* $\mathtt{n}_1^k \approx^a \mathtt{n}_2^m$*, iff either:*

1. $k(a) = m(a) = L$ *and* $\mathtt{n}_1 = \mathtt{n}_2$ *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* $\mathtt{x}$*,* $\sigma_1(\mathtt{x}) \approx^a \sigma_2(\mathtt{x})$*.*

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. Outlined above. □

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 Figure 4.3). Such a generalization is developed in the next section.

4.2 Generalized 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:

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

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

Figure 4.2: Assignment rules for the generalized permissive-upgrade

> 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 Figure 4.1. Labels k, l, m may be lattice elements ℓ or $\star$-ed lattice elements $\ell^{\star}$. In examples, suggestive lattice element names L, M, H (low, medium, high) are used. Labels of the form ℓ are called $\star$-free or *pure*. Figure 4.1 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 Figure 4.2. They strictly generalize the rule ASSN-PUS for the two-point lattice, treating $P = L^{\star}$. Rule ASSN-N applies when the existing label of the variable being assigned to is $\ell_{\mathtt{x}}$ or $\ell_{\mathtt{x}}{}^{\star}$ and $pc \sqsubseteq \ell_{\mathtt{x}}$. The key intuition behind the rule is the following: If $pc \sqsubseteq \ell_{\mathtt{x}}$, then it is safe to overwrite the variable, because $\ell_{\mathtt{x}}$ 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 $pc \not\sqsubseteq \ell_{\mathtt{x}}$. 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 $\ell = \ell_{\mathtt{x}}$, the pure part of the original label of x. The final label on x would be $\ell_{\mathtt{x}}{}^{\star}$ 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 3.2 and 3.3. However, for a more general lattice, this intuition is unsound, as illustrated with an example below. The correct label is $((pc \sqcup m) \sqcap \ell_{\mathtt{x}})^{\star}$.

Example The need for the label $k := ((pc \sqcup m) \sqcap \ell_{\mathtt{x}})^{\star}$ instead of $k := \ell_{\mathtt{x}}{}^{\star}$ in rule ASSN-S is illustrated below. Consider the lattice of Figure 4.3 and the program of Listing 4.1. Assume that, initially, the variables z, w, $\mathtt{x}_1$, x′, $\mathtt{x}_2$, $\mathtt{y}_1$ and $\mathtt{y}_2$ 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 $\mathtt{x}_1$ at $\mathtt{true}^{L_1}$, so that the branch on line 5 is always taken and line 6 is always executed. Set $\mathtt{y}_1 \mapsto \mathtt{false}^{M_1}, \mathtt{y}_2 \mapsto \mathtt{true}^{M_2}, \mathtt{w} \mapsto \mathtt{false}^{L_1}$ initially. The initial value of z is irrelevant. Consider two executions of the program starting from two stores σ_1 with

```
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 4.1: Example explaining rule ASSN-S

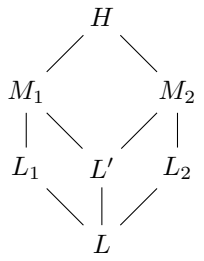

Figure 4.3: Lattice explaining rule ASSN-S

$\mathtt{x}' \mapsto \mathtt{true}^{L'}, \mathtt{x}_2 \mapsto \mathtt{true}^{L_2}$ and σ_2 with $\mathtt{x}' \mapsto \mathtt{false}^{L'}, \mathtt{x}_2 \mapsto \mathtt{false}^{L_2}$. 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 $k := \ell_{\mathtt{x}}{}^{\star}$ 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 third columns of Table 4.1. 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; hence, the program leaks the value of $\mathtt{x}'$ and $\mathtt{x}_2$ to L_1.

With the correct ASSN-S rule in place, this leak is avoided (last column of Table 4.1). 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.

4.2.1 Termination-Insensitive Non-interference

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* $\mathtt{n}_1^k$ *and* $\mathtt{n}_2^m$ *are* ℓ*-equivalent, written* $\mathtt{n}_1^k \sim_\ell \mathtt{n}_2^m$*, iff either*

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

Table 4.1: Execution steps in two runs of the program from Listing 4.1, with two variants of the rule ASSN-S

1. $k = m = \ell' \sqsubseteq \ell$ *and* $\mathtt{n}_1 = \mathtt{n}_2$, *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* $\mathtt{x}$, $\sigma_1(\mathtt{x}) \sim_\ell \sigma_2(\mathtt{x})$.

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. A significant difficulty in proving the theorem is that the definition of $\sim_\ell$ is not transitive unlike the previous definition of $\sim$. Detailed proofs of all the lemmas and the theorems are presented in Appendix A.3.

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

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 derivation rule. □

Corollary 1. *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 5. □

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

Proof. By induction on the derivation rule. □

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. □

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 derivation rule. □

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. □

4.3 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. For brevity, these generalizations are called puP (Section 4.1) and puA (Section 4.2), respectively (P and A stand for powerset and arbitrary, respectively). Since both puP and puA apply to powerset lattices, an obvious question is whether one is more permissive than the other on such lattices. The generalization of puP presented in this chapter can be more permissive than puA on powerset lattices in certain cases as shown by the example below. The reason for this permissiveness is that puP tracks finer taints, i.e., it tracks partial leaks for each principal separately.

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

Listing 4.2: Example where puP is more permissive than puA

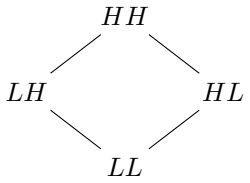

Figure 4.4: A powerset/product lattice

Example The powerset lattice of Figure 4.4 is used for illustration purpose. 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 puP 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 puA, 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 puA applied to this lattice). Consider the program in Listing 4.2. Assume that x, y and z have initial labels LL, HL and LH, respectively and that the initial store contains $\mathtt{y} \mapsto \mathtt{true}^{HL}, \mathtt{z} \mapsto \mathtt{true}^{LH}$, so the branches on lines 1 and 3 are both taken. The initial value in x is irrelevant but its label is important. Under puA, 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 puP, 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, puP is more permissive than puA.

An example for which puA is more permissive than puP in the case of poweset lattices was not found. But the generalization puA presented in Section 4.2 is more general than the product construction puP (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 improved permissive upgrade strategy generalises to arbitrary lattices. By developing this generalization, this work makes permissive-upgrade applicable to arbitrary security lattices like other IFC techniques.

Part III

Handling Unstructured Control Flow

Chapter 5

Dynamic IFC with Unstructured Control Flow and Exceptions

This chapter presents a mechanism to prevent leaks due to implicit flows in the presence of unstructured control constructs like break, return-in-the-middle and exceptions.

Implicit flow corresponds to *control dependence* in program analysis, where a predicate governs which program path is executed and leaks information through the control flow of the program. To avoid overtainting pc labels, an important goal in implicit flow tracking is to determine when the influence of a control construct has ended. For block-structured control flow limited to `if` and `while` commands, this is straightforward: The effect of a control construct ends with its lexical scope, e.g., in

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

`h` influences the control flow at `l = 1` but not at `l = 2`. This leads to a straightforward implementation of a pc upgrading and downgrading strategy: One maintains a *stack* of pc labels [107]; 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 to non-block-structured control flow constructs such as `break`, `continue` and `return`-in-the-middle for functions, all of which occur in 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 block-scope guarded by `h`. Indeed, the pc upgrading and downgrading strategy just described is

The content of this chapter is based partly on the work published as part of the paper, "Information Flow Control in WebKit's JavaScript Bytecode" [25]

ineffective for this program.

Implicit flow in the form of error handling is also a source of information leak as it helps the adversary to learn about the system [50]. For instance, an exception handler might print the stack trace of the error, from which an attacker can determine what sort of attacks the system is vulnerable to.

Tracking information flow in the presence of unstructured control flows is non-trivial as the control breaks out of block structures. Exceptions are much more difficult to handle as they allow for non-local control transfer. Much work on error handling has focussed in the context of static analysis [14, 79] and the work on IFC for dynamic languages has mostly ignored exceptions and other unstructured control flow constructs [16, 17, 35, 47, 53]. Just et al. [64] 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 [57] 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 flows [59]. These labels are more restrictive than needed, e.g., the code indicated by dots in the example above 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.

To solve this issue, this chapter presents a precise dynamic analysis approach using *post-dominator* analysis [45, 64].

5.1 Control Flow Graphs and Post-dominator Analysis

The approach presented in this chapter performs 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 fows. For every branch node, its immediate post-dominator (IPD) is computed [25, 45, 64]. 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 [74, 104], the authors prove 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 are represented by a single *return* node. Theorem 6 shows that the IPD of a node is the most precise node where the context of an operation can be removed. For proving Theorem 6, a few definitions are defined below.

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 end point of* $\mathcal{G}$. *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 $(n_1, n_2, ..., n_m)$ 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 $\texttt{pd}$ 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 $\texttt{pd}$ n and*
2. *$\nexists n_o \in \mathcal{N}.((n_o \texttt{ pd } n) \wedge (n_o < i))$ or*
   $\forall n_o \in \mathcal{N}.((n_o \neq n) \implies (n_o \texttt{ pd } n \implies n_o \texttt{ pd } i))$.

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. The proof of the theorem is described in Appendix B. □

5.2 Exceptions and Synthetic Exit Nodes

Maintaining a precise CFG for dynamic analysis in the presence of exceptions is expensive. The CFG of a function is constructed statically on-the-fly at runtime and 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.

In the design presented in this chapter, every function that may throw an unhandled exception and has an exception handler present earlier in the call-stack (which is known at runtime) 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, is already on top of the pc-stack. 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. The proof is shown in Appendix B.

Theorem 7. *The actual IPD of a node having `SEN` as its IPD is the node on the top of the pc-stack, which lies in a previously called function.*

In summary, these semantics emulate the effect of having cross-function edges. For illustration, consider the following two functions `f` and `g`. 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() = {
 l = 0;
 try { g(); } catch(e) { l = 1; }
 □ return l;
}
```

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

It should be clear that in the absence of instrumentation, 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`. 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 the point of exception, i.e., H. Consequently, permissive-upgrade prevents the implicit information leak in this case.

5.3 Formal Model

This section formally models the semantics of the language with dynamic IFC instrumentation including unstructured control flow and exceptions. The language from Figure 3.1 is extended to include unstructured control flow constructs like `break`, `continue`, `return` and exceptions. Programs are considered as 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 in the formal model. Thus, the program itself is modeled as a huge control flow graph ($\mathcal{G}$). IPDs are computed using the algorithm by Lengauer and Tarjan [69] when the CFG is created. The CFG is statically constructed (only once) as new functions are called or discovered at runtime. 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.

$$e := \mathtt{n} \mid \mathtt{x} \mid e_1 \odot e_2$$
$$\iota := \mathtt{end} \mid \mathtt{x} := e \mid \mathtt{branch}\ e \mid \mathtt{jmp} \mid \mathtt{return} \mid \mathtt{throw}\ e \mid \mathtt{catch}\ \mathtt{x} \mid \mathtt{SEN}$$

Figure 5.1: Language Syntax

The syntax of the language modeling the nodes in a CFG is shown in Figure 5.1. The command `if` e `then` c_1 `else` c_2 is represented as the node `branch` e with $\mathit{Left}(\mathcal{G}, \iota) = c_1$ and $\mathit{Right}(\mathcal{G}, \iota) = c_2$. Similarly, `while` e `do` c is represented as `branch` e with $\mathit{Left}(\iota) = c$ and $\mathit{Right}(\iota)$ being the command following `while` in the program. A `jmp` node in the CFG corresponds to `break` and `continue` with $\mathit{Succ}(\iota)$ 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 normally ends with the `return` node. Multiple return statements in a function are represented using a single `return` node. $\mathit{Succ}(\iota)$ points the node to return to in the previous CFG, while the return value is saved in a global variable that can be accessed by the program later on. When a new CFG is added for a function, the `return` node of that CFG points to the successor node of the function call in the previous CFG. However, the IPDs are computed when the CFGs are intraprocedural. 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. (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

$$\text{ASSN}\ \frac{\iota = (\mathtt{x} := e) \qquad \langle \sigma, e\rangle \Downarrow \mathtt{n}^m \qquad l = \Gamma(\sigma(\mathtt{x})) \qquad l = \ell_{\mathtt{x}} \vee l = \ell_{\mathtt{x}}^{\star} \qquad pc = \Gamma(!\rho) \qquad k = \left\{ \begin{array}{ll} pc \sqcup m, & pc \sqsubseteq \ell_{\mathtt{x}} \\ ((pc \sqcup m) \sqcap \ell_{\mathtt{x}})^{\star}, & pc \not\sqsubseteq \ell_{\mathtt{x}} \end{array} \right\} \qquad \sigma' = \sigma[\mathtt{x} \mapsto \mathtt{n}^k] \qquad \iota' = Succ(\iota) \qquad \rho' = isIPD(\iota', \rho)}{\langle \sigma, \iota, \rho\rangle \rightarrow \langle \sigma', \iota', \rho'\rangle}$$

$$\text{BRANCH}\ \frac{\iota = \mathtt{branch}\ e \qquad \langle \sigma, e\rangle \Downarrow \mathtt{b}^{\ell} \qquad \iota' = \left\{ \begin{array}{ll} Left(\iota), & \textit{if } \mathtt{b} = \mathtt{true} \\ Right(\iota), & \textit{otherwise} \end{array} \right\} \qquad \rho'' = \rho.push(\ell, IPD(\iota)) \qquad \rho' = isIPD(\iota', \rho'')}{\langle \sigma, \iota, \rho\rangle \rightarrow \langle \sigma, \iota', \rho'\rangle}$$

$$\text{JMP, RET, SEN}\ \frac{\iota = \mathtt{jmp}\ \textit{or}\ \mathtt{return}\ \textit{or}\ \mathtt{SEN} \qquad \iota' = Succ(\iota) \qquad \rho' = isIPD(\iota', \rho)}{\langle \sigma, \iota, \rho\rangle \rightarrow \langle \sigma, \iota', \rho'\rangle}$$

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

$$\text{CATCH}\ \frac{\iota = \mathtt{catch}\ \mathtt{x} \qquad excValue = \mathtt{n}^m \qquad l = \Gamma(\sigma(\mathtt{x})) \qquad l = \ell_{\mathtt{x}} \vee l = \ell_{\mathtt{x}}^{\star} \qquad pc = \Gamma(!\rho) \qquad k = \left\{ \begin{array}{ll} pc \sqcup m, & pc \sqsubseteq \ell_{\mathtt{x}} \\ ((pc \sqcup m) \sqcap \ell_{\mathtt{x}})^{\star}, & pc \not\sqsubseteq \ell_{\mathtt{x}} \end{array} \right\} \qquad \sigma' = \sigma[\mathtt{x} \mapsto \mathtt{n}^k] \qquad \iota' = Succ(\iota) \qquad \rho' = isIPD(\iota', \rho)}{\langle \sigma, \iota, \rho\rangle \rightarrow \langle \sigma', \iota', \rho'\rangle}$$

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

Figure 5.2: Semantics

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 formalized in the function $\rho.push(\ell, \iota)$.

The derivation rules in Figure 5.2 define small-step semantics that define the judgment: $\langle \sigma, \iota, \rho\rangle \rightarrow \langle \sigma', \iota', \rho'\rangle$ (as big-step semantics cannot model unstructured control flow). The rules for expressions are the same as in Figure 3.2. The rules are informally explained above. The soundness of the analysis including unstructured control flow and exceptions is proven below.

To state the theorem formally, the equivalence of different data structures with respect to the adversary needs to be defined. Value and memory equivalence is defined as before in Definitions 7 and 8. To prove termination-insensitive non-interference, some additional definitions and auxiliary lemmas are defined below. The detailed proofs can be found in Appendix C.

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 label 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. $\langle \sigma_1, \iota_1, \rho_1 \rangle \rightarrow^* \langle \sigma_1', \texttt{end}, [] \rangle$
3. $\langle \sigma_2, \iota_2, \rho_2 \rangle \rightarrow^* \langle \sigma_2', \texttt{end}, [] \rangle$

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

Part IV

Limited Information Release

Chapter 6

Bounding Information Leaks Dynamically

Information flow control allows tracking of sensitive and private information while preventing leaks to unauthorized public channels. However, in some cases it is required that (some part of) the sensitive data be accessible over certain public channels. This violates the security property of non-interference but is needed for practical reasons. This is generally achieved by specifying special declassification policies, which allow for such release of information. For instance, the declassification policy `declassify(pwd == input)` would treat the result of the equality check (`pwd == input`) as public, thus allowing information to be released to the user. The enforcement generally requires `declassify` annotations to be added to the code to specify which operations need to release the value. Most of the approaches require policy specifications by the developer that define *what* information can be released by the program when the data to be declassified is part of the untrusted code; else the untrusted code can arbitrarily add `declassify` annotations and release information at will.

An alternative approach to that is to quantify the amount of information about a sensitive value that can be released by a program and to determine an acceptable upper bound, which accepts or rejects the program. The information released is generally quantified by computing the difference in the entropy, a measure of uncertainty, of information contained in the sensitive value before and after the release with respect to an adversary. Most analysis methods that have been proposed until now are static in nature and quite a few of these determine an average worst-case bound on the amount of information that a program can release as a whole. However, all of these measures rely on the fact that the probability distribution of the output values is known upfront.

The motivation of this work is to provide an analysis that bounds information leaks dynamically at runtime and to prove that the approach is sound. Previously proposed quantitative information flow analysis methods in general yield leakage bounds over all possible executions of a program. A

```
1  age = getCurrentAge(birthday, birthyear);
2  if (age < 18)
3     preference = "child";
4  else
5     preference = "adult";
```

Listing 6.1: Age-based Advertisement

dynamic analysis, on the other hand, considers only the path being taken in the current execution, which in general consensus seems to be unsound for quantifying information leaks. However, such an analysis would prove useful in scenarios like the Web where the program to be analyzed might only be available at runtime and one needs to only bound the amount of information released by the program about a sensitive value to untrusted sources [28]. Additionally, it seems intuitive to reason about the path taken by the current execution of the program alone. Such leaks can be accounted for and bounded in the dynamic analysis presented in this chapter.

McCamant and Ernst [76] provide a dynamic analysis for quantifying information flows but do not prove the soundness of their approach. They, however, provide a simulation-based proof for quantifying the amount of information released by the program about a secret for a simple system with a two-level lattice. However, it is unclear how their technique scales for multiple security levels or multiple secrets in the system. Moreover, the approach is more suitable for quantification than bounding information leaks; bounding information leaks introduces additional issues as detailed in later sections. Other prior work [23] used self-information to quantify the number of bits of information that could be leaked about a secret value. However, this technique requires the knowledge of the probability distribution or individual probabilities of the outputs that are based on secret values. In cases where it is difficult to determine the probability of the outputs upfront, the use of this measure for analysis does not help much.

This chapter explores an alternative approach, where the developer can specify a *budget* for every sensitive value in the system (defaulting to zero if not specified), which is basically an upper bound on the amount of information allowed to leak about that value. An underlying enforcement mechanism ensures that no more information about that value is leaked than what is specified by the budget.

Bounding information leaks has additional advantages — in scenarios like the Web where the untrusted third-party code changes quite often, it is difficult for the developer to keep up with the changes and modify the *what* policy specifications accordingly. For instance, consider the program snippet in Listing 6.1 where the displayed advertisements for some webpage depend on the viewer's age. The exact age is considered sensitive data, but in order to provide its functionality only an abstraction of the age is required, namely whether `age < 18`. The developer might specify `declassify(age < 18)` as a declassification policy. However, different jurisdictions might entail adaptations to this rule (requiring a different check on `age`), resulting in a cumbersome process where the developer needs to update the policy frequently. If, however, the developer were able to specify that one *comparison* with the value of `age` is permitted, the advertiser could specify the exact limit without requiring any adaptations of the policy. Additionally, there are quite a few real-world programs

(e.g., password checker) that allow a limited number of comparison operations to be performed.

This chapter provides a formal meaning for such operations while establishing important security properties related to them. Thus, in the above example if a budget of 3 is associated with the initial value in the variable age (meaning that only 3 bits of information about age are allowed to leak), this example will be accepted in the envisioned model. If the advertiser changes the criteria to include another check whether the user is a teenager, this comparison can be included without requiring any modifications to the policy. The soundness of the underlying enforcement would ensure that no more information than the specified budget can ever leak.

The challenge, however, is to build an enforcement mechanism that quantifies such information leaks soundly at runtime. As a solution, this chapter proposes *limited information release*, an information release policy that allows the flow of secret information to public channels in a controlled manner by declassifying fragments of the secret. The policy enforces that the information leaked about the secrets by a program is bounded by the specified budgets. The soundness of the approach is proven by coding the outputs generated through information leaks and by showing that the codes are uniquely-decodable. The proof utilizes an important result by [96] that the average length of uniquely-decodable codes upper bounds the Shannon entropy of the outputs.

6.1 Quantifying Information Leaks

The information released, or the *mutual information*, is generally quantified as the difference in the entropy, which is a measure of uncertainty, of the sensitive information before and after the information is released, i.e.,

mutual information = initial uncertainty - final uncertainty

Roughly speaking, this amounts to the gain in knowledge of the adversary about a sensitive value. The adversary computes a probable initial set of values for the sensitive value. By observing the information released, the adversary can refine his/her knowledge set by removing the improbable values.

Various measures [10, 37, 75, 96, 97] have been proposed for quantifying information leaks. Many of these measures determine an average worst-case bound on the amount of information that a program can leak. These measures are based on associating variable-sized bit-codes to different outputs. Other measures consider the vulnerability of a secret, which is directly proportional to its probability distribution. The entropy of the program outputs is the expected value of a random variable formed by the probability distribution of the different outputs or the most probable output.

Intuitively, the measures computes the average number of bits required by an adversary to uniquely encode the different outputs of the program related to the sensitive value. Thus, even a single bit can encode a 32-bit or 64-bit value. Imagine that a sensitive value can take the following values: “N”, “S”, “E”, and “W”. To accurately determine the value, comparisons done twice would suffice. Assuming that only the results of the comparison operations are visible, 2 bits can be used to reveal any of the four values: “N” as {`true`, `true`}, “S” as {`false`, `true`}, “E” as {`true`, `false`} and “W” as {`false`, `false`}. In all the four cases, the two bits reveal all the 32 or 64 bits contained in the value.

```
1 age = getCurrentAge(birthday, birthyear);
2 if (age < 13)
3   preference = "child";
4 else if (age < 20)
5   preference = "teenager";
6 else
7   preference = "adult";
```

Listing 6.2: Age-based Advertisement

However, all the entropy-based measures would compute a maximum of 2-bit leakage (depending on the probabilities the computed leakage can be less than 2 bits).

Thus, it is more intuitive to view the measure of mutual information as being bounded by the number of operations (guesses) allowed to reveal information about the secret. Its average over different executions is shown to be bounded by the Shannon entropy of the different outputs in this chapter using the Shannon source-coding theorem [96].

6.2 Limited Information Release

Limited information release (LIR) is an information release policy that declassifies small fractions of sensitive information. The motivation for LIR is that, in general, certain information flows leak only an insignificant piece of a secret. As an example, the *comparison* of a secret with a constant value is largely considered acceptable and its rejection by standard IFC analyses is too restrictive in practice [21, 65]. Such information leaks are usually acceptable if one can guarantee that an adversary cannot widen the declassification to launder information. The need for such policies is motivated with some examples below.

6.2.1 Motivating Examples

Age-based advertisements A third-party advertising library might want to display ads based on the viewer's age. Consider the program snippet in Listing 6.2, a modified version of the program in Listing 6.1 which determines the preferred advertisements based on the secret value `age`. The third-party ad service does not need to know the exact age of the viewer for that purpose. It is sufficient to know whether the viewer is a child, a teenager or an adult. While leaking very little information this provides focussed functionality and might even be required in some jurisdictions to protect minors from inappropriate ads. As another example, consider a music app that hosts advertisements for music shows and concerts in a town. Based on the age (whether the user is an adult, a teenager or a child) and the preferences of the user, the advertisement might display different categories. Again, there is a tradeoff between a slight amount of private information and better services.

Password checker User authentication is often based on a secret password or PIN. The password checker in Listing 6.3 compares the secret password to the variable `uPwd` containing the password

```
1  dbPwd = getActualPassword(user);
2  uPwd = readUserPassword();
3  login = (dbPwd == uPwd);
```

Listing 6.3: Password Checker

```
1 window.addEventListener("keypress", function(event) {
2       if (event.altKey) {
3          send("ALT key pressed!");
4  }});
```

Listing 6.4: Shortcut Key Usage

entered by the user. The public variable `login` reflects whether these match, indicating whether login was successful. Strict non-interference prohibits assignments to the low variable `login` as the value is derived from the secret password. However, releasing the login status to the user cannot be avoided. Normally, a user enters her password correctly in a single try but the probability of an adversary guessing the correct password in one (or even a few) tries can be assumed to be negligible when the password is strong. In general, a brute-force attack on the secret would be required, and if the secret is from a large domain then such an attack is not a threat [102]. Thus, the assignment to the `login` variable may be permitted.

Analytics many web pages include analytics scripts to track user behavior on the page in order to improve the user experience. Most of these analytics scripts track events on the web page. One such analytics code is shown in Listing 6.4, which tracks whether or not the `alt` modifier is pressed by the user. To prevent precise keylogging, the event properties (the keys pressed) are secret in this case, hence this program snippet does not satisfy non-interference. However, the only information that the script gains in this case is whether or not the `alt` modifier was pressed by the user. Such checks could be allowed in practice unless the script tries to track all the keys pressed by the user.

Sanity check Another common case where information release is acceptable is during sanity checks, e.g., whether input has been provided or not, as shown in Listing 6.5. The `textInput` could be some secret value but in order to use the value it might be necessary to check if the value is non-null. If no input is provided, the error "`no input`" only reveals that the secret `textInput` is equal to `NULL`.

```
1  if (textInput ≠ NULL)
2     useInput ();
3  else
4     error("no input");
```

Listing 6.5: Sanity Check

```
1  pub = 0; i = 1;
2  while (i ≤ 2^31) {
3     if ((sec & i) == i)
4        pub = pub | i;
5     i = i << 1;
6  }
```

Listing 6.6: Laundering attack via implicit flow

6.2.2 Limited Information Release Policy

In practice most implicit flows leak a single bit of information only, which might be required for providing proper functionality. King *et al.* [67] investigate the occurrence of implicit flows in some standard algorithms and discuss both the pros and cons of handling implicit flows. Russo *et al.* [90] also observe that implicit flows cannot be exploited in non-malicious code to leak secrets efficiently. Unfortunately, allowing information release when only an insignificant amount of information is leaked can be widened to leak the complete secret [92, 102]. The LIR policy is, thus, guided by two key tenets:

- **Declassification at comparison operations.** As has been observed by various authors [21, 65] and is exemplified above (Section 7.3), comparison operations often provide a low bandwidth channel for information leak. For instance, $(h \neq l)$ only reveals whether the two values in h and l are equal or not. Similarly, $(h < l)$ and $(h > l)$ only reveal that the value of h is lesser and greater than the value of l, respectively. With LIR, such comparison operations are treated as potential points of information release. This means that under LIR only expressions involving comparison operations can be declassified. As all comparison operations need not be declassified, information is released only for those comparison operations that are annotated with `declassify`.

- **Bounding the information released.** It is well known in the literature that low bandwidth channel (like leaks due to comparisons involving secrets) can be widened to laundering attacks [92, 102]. The example in Listing 6.6 is a classical example of a laundering attack. It implicitly leaks the secret value in `sec` to the variable `pub` without any direct assignments. Every time the check on line 3 is performed, it leaks one bit of `sec`. The whole secret gets implicitly laundered into the variable `pub` as this comparison is performed in a loop of length equal to the size of the secret (in bits).

  To limit the amount of information that can be leaked by such laundering attacks, LIR introduces a notion of *budget* (a non-negative number) associated with a sensitive value, which is an upper bound on the amount of information that is allowed to leak about the sensitive value. A budget is associated with every secret in the program and defaults to zero, if left unspecified.

Intuitively, a program is said to satisfy limited information release, if the information released about the initial value of each secret in the system is limited by its pre-specified budget. If the budgets for all secrets drop to zero, LIR falls back to the standard non-interference policy. It is

important to note that even when staying within the bounds of the budget one can leak the entire secret via comparison operations. For instance, in an equality comparison ($h == l$) involving secret h and a public value l, if the operation returns $true$, the adversary knows that the value of the secret is the same as the value in l. The bounds are only meaningful in an average sense like in many other existing entropy-based notions of information leakage [36, 45, 97]. In other words, LIR only ensures that the average information leak over multiple executions is bounded by the specified budget and it should not be understood as providing leakage bounds over a single execution.

Although declassification of only those expressions that involve comparison operations is allowed by LIR, this can be extended to other expressions as well. In such cases, the leak is accounted for by assuming that the entire secret is leaked (as shown by [36]). Such policies, however, are not of interest as this would effectively mean that the secret value either need not be labeled secret to begin with or can be declassified upfront.

6.3 LIR Enforcement

6.3.1 Language and Syntax

This section describes a runtime enforcement of LIR for the simple imperative language shown in Figure 3.1 extended with the `declassify` operator as shown in Figure 6.1. The comparison and arithmetic operators are separated as $\oplus$ and $\odot$, respectively, for the rules. The language is sufficient for describing the key idea of LIR — appropriate deduction of budgets at declassification of comparison involving secrets.

$$e := \mathtt{n} \mid \mathtt{x} \mid e_1 \odot e_2 \mid e_1 \oplus e_2 \mid \mathtt{declassify}(e_1 \oplus e_2)$$
$$c := \mathtt{skip} \mid \mathtt{x} := e \mid c_1; c_2 \mid \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \mid \mathtt{while}\ e\ \mathtt{do}\ c$$

Figure 6.1: Syntax of the language

Every input (coming in via store variables) to the program has a confidentiality label (part of a security lattice and represented using k, m) associated with it referred to as the *initial label*. An immutable map $\mathbb{L}$ represents the mapping from a variable to its initial label. Along with the initial confidentiality label, every input is also associated with a *budget* and an immutable *budget-label*. The budget on an input represents an upper bound on the amount of information that is allowed to leak about that input. The budget-label serves the purpose of indicating the confidentiality level to which the value can be declassified. Additionally, the budget-label helps in increasing the permissiveness of the approach as described later. An immutable map $\mathbb{B}$ tracks the budget-label associated with different variables. Since the budgets change as the program executes, they themselves can be a channel of information leak. The budgets are represented as $\mathtt{n}^l$, where $\mathtt{n}$ is the budget and l is the budget-label, i.e., if a variable has a budget of 1 and the budget-label L, it is denoted as 1^L.

Instead of directly tracking the label on the values, the variable dependencies (also referred to as provenance) are tracked. As every variable has its initial label, its budget and its budget label

associated with it, tracking dependencies enables access to this metadata for every dependency individually. This assists the enforcement by choosing the right variable to deduct budget from, which is required at the point of declassification. These dependencies are split into two parts and represented as a pair: (l, δ). δ is a set of variables on whose initial values the current value depends. Information can be released only about the variables that are part of the provenance set (δ). The security level l of the label is the join of security levels (upper-bound) of all the provenances for which no more information is allowed to be released. Once the budget of a variable expires (becomes 0), the security level of that provenance is joined with the security level of the value and the variable is removed from the provenance set, i.e., for initial $\mathtt{x} = \mathtt{n}^{(\bot,\{\mathtt{x},\mathtt{y}\})}$ such that $\mathbb{L}(\mathtt{x}) = l$ and $\mathbb{L}(\mathtt{y}) = m$, once the budget of y expires, $\mathtt{x} = \mathtt{n}^{(m,\{\mathtt{x}\})}$. Thus, the security level of a value (l) represents the minimal level at which the value can be observed. The current value in the variable can never be released to a level below l. Initially, every variable depends only on itself. Every initial variable x is labeled $(\bot, \{\mathtt{x}\})$, where $\bot$ represents the least element in the lattice.

The function $\Gamma(\mathtt{n}^{(l,\delta)})$ returns the label of the value, i.e., $\Gamma(\mathtt{n}^{(l,\delta)}) = l \bigsqcup_{\mathtt{x}\in\delta} \mathbb{L}(\mathtt{x}) \bigsqcup_{\mathtt{x}\in\delta} \mathbb{B}(\mathtt{x})$. The function $\Gamma_f(\mathtt{n}^{(l,\delta)})$ returns the security level l of the label. The function $\delta(\mathtt{x})$ returns all the provenances of x. The join operation on the labels returns the join of the security levels and the union of the provenance sets as the final label, i.e.,

$$(l, \delta) \sqcup (l', \delta') = (l \sqcup l', \delta \cup \delta')$$

Similarly, the ordering on the labels is defined as:

$$(l, \delta) \sqsubseteq (l', \delta') = (l \sqsubseteq l', \delta \subseteq \delta')$$

pc denotes the current program-context level, and contains *only* the security level, represented as $(l, \{\})$, i.e., none of the provenances in the pc are allowed to release any information through the variables being assigned within the branch and are hence joined with the security level l of the pc. This decision is justified below with an example. The function $\Gamma(pc)$ returns the security level l of the pc. When the meaning is clear from the context, pc is used instead of $\Gamma(pc)$.

6.3.2 Key Aspects of the Enforcement

6.3.2.1 Provenance Tracking

As the monitor is *flow-sensitive*, information can be released through other variables that are dependent on the initial value of some of the secret variables. Instead of directly tracking abstract labels, which gives a bound on the secrecy but loses the actual input variable dependency, variable dependencies are tracked. These dependencies determine the variables to deduct the budget from. Provenance tracking is required for a correct analysis and to prevent an adversary from laundering secret information as described below. Consider the example in Listing 6.7 with the security lattice $L \sqsubseteq H$. Assume that the initial values in `sec` and `h` are both secret with the initial label H and an initial budget of 1^L; the initial values in `pub` and `i` are both public with an initial label L.

Suppose that at the assignment on line 2 only the label of `sec` is carried over to the label of `h` (like any flow-sensitive IFC monitor would) but the actual variable dependencies are not tracked. Without tracking the dependent variables, the declassification on line 4 would release the value

```
1  pub = 0;
2  h = sec % 2;
3  sec = sec / 2;
4  if (declassify(h == 1))
5     pub = pub | 1;
6  i = sec % 2;
7  if (declassify(i == 1))
8     pub = pub | 2;
```

Listing 6.7: Leak due to dependent variables

```
1  x = (a % 2)
2  if (a == 0)
3    x = b
4  z = declassify(x == 1)
5  y = declassify(b == 0)
```

Listing 6.8: Example to illustrate budget reduction

of h (the last bit of sec) to pub deducting the budget of h but not sec. As a result another bit of information about sec can be leaked later (through declassify on line 6), which should not have been allowed as the budget of sec is just 1.

With dependency tracking, initially h and sec will be $\delta(\texttt{h}) = \texttt{h}$ and $\delta(\texttt{sec}) = \texttt{sec}$. The monitor would update the dependence of h on sec on line 2. As a result, the declassification on line 4 would deduct the budget from sec and not h. On line 6, i is updated with the next bit of sec and so is the dependence of i on sec, i.e., $\delta(\texttt{i}) = \texttt{sec}$. When executing the branch on line 7 as sec has expired its budget, the monitor joins the security level of sec in the new program context making it secret H, hence limiting the amount of information released about the initial value of sec.

6.3.2.2 Budget-label Constraints

When declassifying a value labeled (l, δ), budget is reduced for only those variables in the provenance set δ that have a budget-label at least as high as l, i.e., $\forall \texttt{x} \in \delta.l \sqsubseteq \mathbb{B}(\texttt{x})$ as when the budget-label is below l no useful information can be released. The reduction of budget and release of information is subject to the condition that declassification of a value in a secret context is not allowed, as this could leak information about the context itself. In the current setting, this would mean that if the label of the value being declassified is (l, δ), then $pc \sqsubseteq l$. It turns out that this suffices to prevent leaks via change of budgets, as for declassification it is required that the budget-label is at least as secret as l, i.e., no value is declassified to a level below pc. Without these checks in place, the monitor could leak additional information as shown below.

Consider, for illustration, the example in Listing 6.8. Assume that a and b are two secrets with initial labels $(M, \{\})$ and $(L, \{\texttt{b}\})$, initial labels M and H such that $L \sqsubseteq M \sqsubseteq H$, and initial budgets of 0 and 1^L, respectively. On line 1, x's label becomes $(M, \{\})$ as the budget of a has already expired.

```
1  if (x ≤ 10)
2    x = y
3  z = declassify(x ≤ 5)
```

Listing 6.9: Example illustrating handling of implicit flows

Consider two executions of the program depending on whether a is 0 or not. When a is 0, x is assigned b on line 3 with a label $(M, \{\mathtt{b}\})$. On line 4, x can be declassified to level $M \sqcup L$ (as $\mathbb{B}(\mathtt{b}) = L$). Assume that the value that is declassified is true. Thus, the label of the declassified value in z is $(M, \{\})$ and the budget of b becomes 0. On line 5, as b has expired its budget, y is labeled $(H, \{\})$. The value released in this run is $\mathtt{true}^M$. In the other run when a is not 0, x has a label of $(M, \{\})$ on line 4. Thus, no declassification occurs on line 4. As the budget of b on line 5 is still 1^L, the value on line 5 is declassified to L. Thus, the label of y after the assignment becomes $(L, \{\})$ and the value released for this run is $\mathtt{false}^L$. Thus, in the first run an L-level adversary cannot observe any value while in the second run, the adversary can see the value as it is labeled L. As the adversary knows the budget of b, he/she can conclude that in the first case the branch on line 2 was taken and in the second case it was not taken. This leaks an additional bit of information about a although the budget of a doesn't allow any information release.

To prevent this leak, in the first case, the value of x on line 4 is not declassified as the budget-label of b ($\mathbb{B}(\mathtt{b}) = L$) is lower than the security level of the value (M).

6.3.2.3 Handling implicit flows

Implicit flows are handled as per the permissive-upgrade strategy — assignments to public variables in a secret context (pc) are labeled partially-leaked. However, for simplicity of exposition the no-sensitive-upgrade check is used here and further in the chapter instead of the permissive-upgrade strategy.

Since variable dependencies are used in addition to the security levels, those dependencies also need to be checked for during assignment operations. For an assignment to succeed, it is important that no new dependencies are handed over to the variable as part of the assignment operation. Thus, it is required that $pc \sqsubseteq k$, where k is the security level of the variable being assigned, i.e., the security level of the variable being assigned is at least as high as the security level of the pc. This prevents leaks as the assignments that happen under a pc would carry the dependencies of the pc too while in an alternate run those dependencies might not be present in the label of the assigned variable. The mismatch of dependencies in the two cases can leak information as illustrated using an example in Listing 6.9.

For the program in Listing 6.9, assume that x, y and z have the initial confidentiality labels of H, H and L ($L \sqsubseteq H$), and budgets of 1^L, 0 and 0 assigned to them, respectively. Also assume that their current dependencies are of the form $(L, \{\mathtt{x}\})$, $(H, \{\})$ and $(L, \{\})$. Consider two runs of the program with $\mathtt{x} \leq 10$ in one and $\mathtt{x} > 10$ in the other. Suppose that check for provenances are kept in the pc, i.e., the assignment succeeds if $pc = (l, \delta) \sqsubseteq (k_o, \delta_o)$ where (k_o, δ_o) is the label of the variable x. When $\mathtt{x} \leq 10$, the branch on line 1 is taken and the pc on line 2 would be $(L, \{\mathtt{x}\})$, which would

```
1  if (med ≤ 0)
2    x = declassify(sec == y);
3  if (declassify(y ≠ 100))
4    pub = 1;
```

Listing 6.10: Example to illustrate permissiveness

allow the assignment (as dependency of x is not less sensitive than the dependencies in pc). As a result the label of x would be updated to $(H, \{x\})$ (join of pc and label of y). On line 3, when the value of $x \leq 5$ is declassified, no information is released as the budget-label of x is lower than its security level (H) (as explained above). In the other run when $x > 10$, the branch is not taken and the value of $x \leq 5$ on line 3 is released to the level L as the label of x on line 3 is $(L, \{x\})$ and the budget is 1^L. This leads to different sensitivity of z in the two runs which leads to leaking the value of $x \leq 10$ in the first run without being accounted for in its budget.

With the modification suggested above in place, in the first run — the value of $pc = H$ (because $\mathbb{L}(x) = H$) and $\Gamma_f(x) = L$ and $H \not\sqsubseteq L$. Thus, in the first run of the above example, the assignment is not allowed as per the no-sensitive-upgrade check. In case of the permissive-upgrade strategy, the label would be computed as $((\Gamma(pc \sqcup m)) \sqcap l)^\star$. A partially-leaked value does not carry any provenances as no declassification is allowed on a partially-leaked value.

6.3.2.4 Permissiveness

The overall confidentiality of the underlying value with a label (l, δ) is the join of l with the initial label of all the variables in the dependency set. Thus, all those dependencies whose initial label is below the first part of the label can be removed as keeping those dependencies does not affect the confidentiality of the value. This improves the permissiveness of the technique by not marking the release of information for variables that have a label lesser than or equal to the current security level or the pc because no information can be released below the current security level of the variable or the current pc. For instance, consider a value $n^{(k,\{x\})}$ such that its label's first part is k and the dependency set consists of just one variable x. If $\mathbb{L}(x) \sqsubseteq k$ then x is removed from the dependency set and represent the value as $n^{(k,\{\})}$. As explained earlier, budgets are deducted only when $pc \sqsubseteq k$, thus, the budgets of any variable whose initial label is below the pc are not deducted from.

For illustration, consider the program in Listing 6.10. Assume that, med, x, sec, y, and pub are labeled $(M, \{\})$, $(M, \{\})$, $(M, \{sec\}), (L, \{y\})$, and $(L, \{pub\})$, respectively, $\mathbb{L}(sec) = H, \mathbb{L}(y) = M, \mathbb{L}(pub) = L$ such that $L \sqsubset M \sqsubset H$ and the budgets of sec and y are 1^M and 1^L, respectively. The program context label (pc) on line 2 is $(M, \{\})$. The expression sec == y on line 2 has the label $(M, \{sec, y\})$. Without the check for the current security level, the expression evaluation on line 2 releases the value of sec and y into x. However as the security level is M, the final value of x would have the security level of at least M. Thus, information about y is released to M-level observers, who already had access to the value of y. Subsequently, on line 3 as the budget of y has expired, no more information about y is released and the assignment on line 4 fails. Thus, no useful information about y is released at any point in the program, yet the program is terminated because of the

NSU check. With the check for current security level and the context label, the budget of y is not deducted on line 2, which allows the check on line 3 to release information about y thereby allowing the assignment on line 4 to succeed.

6.3.3 Semantics

To formally define the information released by a program starting from some initial memory containing secret values, big-step semantics is defined for the language shown in Figure 6.1 that releases some information about initial secret values at comparison operations.

Program configurations are extended with ι representing the budget store, which is a map from the variables to the budget of these initial values (inputs to the program). The configurations for expressions (e) and commands (c) are, thus, represented by $\langle \sigma, \iota, e \rangle$ and $\langle \sigma, \iota, c \rangle$, respectively where σ represents the memory store as before. $\langle \sigma, \iota, e \rangle \underset{pc}{\Downarrow} \mathtt{n}^{(k,\delta)}, \iota', \tau$ defines expression evaluation. It means that under some pc, starting with memory σ and budget store ι, the expression e evaluates to a value $\mathtt{n}$ labeled (k, δ) resulting in a budget store ι'. Additionally, the evaluation generates a trace τ, which is a list of values of the form $\mathtt{n}_1^{k_1} :: \mathtt{n}_2^{k_2} :: \ldots :: \mathtt{n}_j^{k_j}$. Every declassified value is recorded on the trace along with the level to which it is declassified. The two immutable maps — initial label map ($\mathbb{L}$) and budget-label map ($\mathbb{B}$) are assumed to be available and omitted from the rules for clarity.

The evaluation rules for expressions shown in Figure 6.2 are explained below:

- CONST: Constant values evaluate to themselves with the label $(\bot, \{\})$ as they do not have any dependencies. There is no change to the budget store and it generates an empty trace (ϵ).
- VAR: The evaluation of a variable returns its value $\mathtt{n}$ along with the label (k_o, δ_o). The function $\mathbb{R}$ returns those variables in δ_o which have a budget 0. The current security level k_o is joined with the initial labels of those variables while removing them from the provenance set δ_o. As no new information is released, ι remains unchanged and no trace is generated.
- AOP and COP: For arithmetic operations and normal comparison operations, the label of the expression is the join of the label of the values that the two sub-expressions evaluate to. The individual sub-expressions can release some information τ_1 and τ_2, thus updating ι to ι'.
- DCOPR: This rule corresponds to the comparison operation with `declassify`, and allows information release in certain cases. The separation of `declassify` from the normal comparison operations is to offer more control over what information is required to be released. The rule applies only when the current context pc is lower than or equal to the label of the value obtained by evaluation ($pc \sqsubseteq k_o$). This is to ensure that the declassification does not happen in a high context. The function $\Delta(\delta, l, \mathbb{L})$ returns the variables in the provenance set δ that have an initial label greater than l. Additionally, if the budget of the variables in the provenance set has expired during the evaluation of the sub-expressions, those variables are removed from the provenance set (using the function $\mathbb{R}$). To prevent the leak described earlier, if the budget label of any of the remaining variables in the provenance set is not at least as high as the original security level of the computed value ($k_o \not\sqsubseteq \mathbb{B}(\mathtt{x})$), then the variable is removed from the dependency set. For all the variables removed from the provenance set, their security level is

$$\textsc{const}\ \frac{}{\langle \sigma, \iota, \mathtt{n} \rangle \underset{pc}{\Downarrow} \mathtt{n}^{(\bot,\{\})}, \iota, \epsilon}$$

$$\textsc{var}\ \frac{\sigma(\mathtt{x}) = \mathtt{n}^{(k_o,\delta_o)} \qquad \delta' = \mathbb{R}(\delta_o, \iota, k_o, \mathbb{B}) \qquad \delta = \delta_o \setminus \delta' \qquad k = k_o \bigsqcup_{\mathtt{x} \in \delta'} \mathbb{L}(\mathtt{x})}{\langle \sigma, \iota, \mathtt{x} \rangle \underset{pc}{\Downarrow} \mathtt{n}^{(k,\delta)}, \iota, \epsilon}$$

$$\textsc{aop}\ \frac{\begin{array}{c}\langle \sigma, \iota, e_1 \rangle \underset{pc}{\Downarrow} \mathtt{n}_1^{(k_1,\delta_1)}, \iota_1, \tau_1 \\ \langle \sigma, \iota_1, e_2 \rangle \underset{pc}{\Downarrow} \mathtt{n}_2^{(k_2,\delta_2)}, \iota', \tau_2 \\ \mathtt{n} = \mathtt{n}_1 \odot \mathtt{n}_2 \qquad k = k_1 \sqcup k_2 \\ \delta = \delta_1 \cup \delta_2 \qquad \tau = \tau_1 :: \tau_2\end{array}}{\langle \sigma, \iota, (e_1 \ \odot \ e_2) \rangle \underset{pc}{\Downarrow} \mathtt{n}^{(k,\delta)}, \iota', \tau}$$

$$\textsc{cop}\ \frac{\begin{array}{c}\langle \sigma, \iota, e_1 \rangle \underset{pc}{\Downarrow} \mathtt{n}_1^{(k_1,\delta_1)}, \iota_1, \tau_1 \\ \langle \sigma, \iota_1, e_2 \rangle \underset{pc}{\Downarrow} \mathtt{n}_2^{(k_2,\delta_2)}, \iota', \tau_2 \\ \mathtt{n} = \mathtt{n}_1 \oplus \mathtt{n}_2 \qquad k = k_1 \sqcup k_2 \\ \delta = \delta_1 \cup \delta_2 \qquad \tau = \tau_1 :: \tau_2\end{array}}{\langle \sigma, \iota, (e_1 \ \oplus \ e_2) \rangle \underset{pc}{\Downarrow} \mathtt{n}^{(k,\delta)}, \iota', \tau}$$

$$\textsc{dcopr}\ \frac{\begin{array}{c}\langle \sigma, \iota, e_1 \rangle \underset{pc}{\Downarrow} \mathtt{n}_1^{(k_1,\delta_1)}, \iota_1, \tau_1 \qquad \langle \sigma, \iota_1, e_2 \rangle \underset{pc}{\Downarrow} \mathtt{n}_2^{(k_2,\delta_2)}, \iota_2, \tau_2 \\ \mathtt{n} = \mathtt{n}_1 \oplus \mathtt{n}_2 \qquad \delta_o = \delta_1 \cup \delta_2 \qquad k_o = k_1 \sqcup k_2 \\ pc \sqsubseteq k_o \qquad \delta' = \Delta(\delta_o, k_o, \mathbb{L}) \qquad \delta'' = \mathbb{R}(\delta', \iota_2, k_o, \mathbb{B}) \\ \delta = \delta' \setminus \delta'' \\ k = k_o \bigsqcup_{\mathtt{x} \in \delta''} \mathbb{L}(\mathtt{x}) \bigsqcup_{\mathtt{y} \in \delta} \mathbb{B}(\mathtt{y}) \qquad \iota' = \mathbb{I}(\iota_2, \delta) \qquad \tau = \left\{ \begin{array}{ll} \tau_1 :: \tau_2 :: \mathtt{n}^k, & \textit{if } (\delta \neq \emptyset) \\ \tau_1 :: \tau_2, & \textit{otherwise} \end{array} \right\}\end{array}}{\langle \sigma, \iota, \mathtt{declassify}(e_1 \oplus e_2) \rangle \underset{pc}{\Downarrow} \mathtt{n}^{(k,\{\})}, \iota', \tau}$$

$$\textsc{dcopn}\ \frac{\begin{array}{c}\langle \sigma, \iota, e_1 \rangle \underset{pc}{\Downarrow} \mathtt{n}_1^{(k_1,\delta_1)}, \iota_1, \tau_1 \qquad \langle \sigma, \iota_1, e_2 \rangle \underset{pc}{\Downarrow} \mathtt{n}_2^{(k_2,\delta_2)}, \iota_2, \tau_2 \\ \mathtt{n} = \mathtt{n}_1 \oplus \mathtt{n}_2 \qquad \delta = \delta_1 \cup \delta_2 \qquad k = k_1 \sqcup k_2 \qquad pc \not\sqsubseteq k \qquad \tau = \tau_1 :: \tau_2\end{array}}{\langle \sigma, \iota, \mathtt{declassify}(e_1 \oplus e_2) \rangle \underset{pc}{\Downarrow} \mathtt{n}^{(k,\delta)}, \iota_2, \tau}$$

Auxiliary functions:

$$\Delta(\delta, k, \mathbb{L}) = \left\{ \mathtt{x} \;\middle|\; (\mathtt{x} \in \delta) \wedge (\mathbb{L}(\mathtt{x}) \not\sqsubseteq k) \right\}$$

$$\mathbb{R}(\delta, \iota, k, \mathbb{B}) = \left\{ \mathtt{x} \;\middle|\; (\mathtt{x} \in \delta) \wedge \left(\iota(\mathtt{x}) = 0 \ \vee \ k \not\sqsubseteq \mathbb{B}(\mathtt{x})\right) \right\}$$

$$\mathbb{I}(\iota, \delta) = \left\{ (\mathtt{x}, \mathtt{n}) \;\middle|\; \left((\mathtt{x}, \mathtt{j}) \in \iota\right) \bigwedge \left(\mathtt{n} = \left\{ \begin{array}{ll} \mathtt{j} - 1, & \textit{if } (\mathtt{x} \in \delta) \\ \mathtt{j}, & \textit{otherwise} \end{array} \right\} \right) \right\}$$

Figure 6.2: LIR - Semantics of expressions

joined with that of the actual label. Depending on whether the provenance set δ is empty or not, either:

- the budget of all remaining variables in the provenance set δ is deducted by 1, corresponding to the 1-bit Boolean value released on the trace. The function $\mathbb{I}(\iota, \delta)$ reduces the budget and returns a new budget store. Their budget label is also joined with the actual security level, which gives the final label of the declassified value *or*
- if the remaining provenance set is empty, no declassification occurs

If declassification occurs, the declassified value is appended to the trace.

- DCOPN: If the current context (pc) is not lower than or equal to the security level of the value obtained by evaluation ($pc \not\sqsubseteq k$), no information release is allowed. The final label is a join of the security levels of all the provenances. The individual sub-expressions can release some information τ_1 and τ_2, thus updating ι to ι_2.

$$\text{SKIP}\ \frac{}{\langle \sigma, \iota, \texttt{skip} \rangle \underset{pc}{\Downarrow} \langle \sigma, \iota, \epsilon \rangle}$$

$$\text{ASSN}\ \frac{pc \sqsubseteq \Gamma_f(\mathtt{x}) \qquad \langle \sigma, \iota, e \rangle \underset{pc}{\Downarrow} \mathtt{n}^{(m,\delta')}, \iota', \tau \qquad k = pc \sqcup m \qquad \delta = \Delta(\delta', k, \mathbb{L})}{\langle \sigma, \iota, \mathtt{x} := e \rangle \underset{pc}{\Downarrow} \langle \sigma[\mathtt{x} \mapsto \mathtt{n}^{(k,\delta)}], \iota', \tau \rangle}$$

$$\text{SEQ}\ \frac{\langle \sigma, \iota, c_1 \rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau_1 \rangle \qquad \langle \sigma', \iota', c_2 \rangle \underset{pc}{\Downarrow} \langle \sigma'', \iota'', \tau_2 \rangle}{\langle \sigma, \iota, c_1; c_2 \rangle \underset{pc}{\Downarrow} \langle \sigma'', \iota'', \tau_1 :: \tau_2 \rangle}$$

$$\text{IF-ELSE}\ \frac{\langle \sigma, \iota, e \rangle \underset{pc}{\Downarrow} \mathtt{n}^{(k_o,\delta)}, \iota', \tau_1 \qquad i = \left\{ \begin{array}{ll} 1, & \textit{if}\ (\mathtt{n} = \mathtt{true}) \\ 2, & \textit{otherwise} \end{array} \right\} \qquad k = k_o \bigsqcup_{\mathtt{x} \in \delta} \mathbb{L}(\mathtt{x}) \qquad \langle \sigma, \iota', c_i \rangle \underset{pc \sqcup k}{\Downarrow} \langle \sigma', \iota'', \tau_2 \rangle}{\langle \sigma, \iota, (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2) \rangle \underset{pc}{\Downarrow} \langle \sigma', \iota'', \tau_1 :: \tau_2 \rangle}$$

$$\text{WHILE-F}\ \frac{\langle \sigma, \iota, e \rangle \underset{pc}{\Downarrow} \mathtt{false}^{(k,\delta)}, \iota', \tau}{\langle \sigma, \iota, \texttt{while}\ e\ \texttt{do}\ c \rangle \underset{pc}{\Downarrow} \langle \sigma, \iota', \tau \rangle}$$

$$\text{WHILE-T}\ \frac{\langle \sigma, \iota, e \rangle \underset{pc}{\Downarrow} \mathtt{true}^{(k_o,\delta)}, \iota_1, \tau_1 \qquad k = k_o \bigsqcup_{\mathtt{x} \in \delta} \mathbb{L}(\mathtt{x}) \qquad \langle \sigma, \iota_1, c; \texttt{while}\ e\ \texttt{do}\ c \rangle \underset{pc \sqcup k}{\Downarrow} \langle \sigma', \iota', \tau_2 \rangle}{\langle \sigma, \iota, \texttt{while}\ e\ \texttt{do}\ c \rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau_1 :: \tau_2 \rangle}$$

Figure 6.3: LIR - Semantics of commands

The judgment for a command execution is given by $\langle \sigma, \iota, c\rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau\rangle$, which means that under a program context pc, the execution of a command c starting with memory σ and budget store ι results in a final memory σ' and budget store ι' while generating a trace τ of released values from expression declassification. The semantics of commands is shown in Figure 6.3. The rules are standard except for some changes in the assignment rule and branching rules. The assignment rule ASSN does the standard NSU check that disallows assignment to a public variable in a secret context — $\Gamma(pc) \sqsubseteq \Gamma_f(\mathtt{x})$. The branching rules IF-ELSE and WHILE-T compute the context label by joining the initial labels ($\mathbb{L}$) of all the provenances in δ.

6.4 Formalization of LIR

The following section formalizes the property of LIR and prove it for the semantics presented above. LIR is essentially a property stating that the total number of bits that can be leaked about a secret (by the program) is upper bounded by its pre-specified budget. The trace generated in the LIR semantics captures every declassified value along with the level to which it is declassified. Thus, if a trace is produced, then its projection to the level of an adversary would return the values that the adversary can observe, defined formally in Definition 16. Similarly, the adversary can view a projection of the budget map as defined in Definition 17.

Definition 16 (Trace projection)**.** *Given a trace, τ, the trace projection w.r.t. an adversary at level ℓ, written $\tau{\uparrow_\ell}$, is defined as:*

$$[]{\uparrow_\ell} = []$$
$$(\mathtt{n}^m :: \tau){\uparrow_\ell} = \begin{cases} \mathtt{n}^m :: \tau{\uparrow_\ell} & \textit{if } m \sqsubseteq \ell, \\ \tau{\uparrow_\ell} & \textit{else.} \end{cases}$$

Definition 17 (Budget map projection)**.** *The projection of a budget map, ι, w.r.t. an adversary at level ℓ, written $\iota{\uparrow_\ell}$, is defined as:*

$$\iota{\uparrow_\ell} = \Pi(\iota, \ell, \mathbb{B}) = \left\{ (\mathtt{x}, \mathtt{n}) \;\middle|\; \big((\mathtt{x}, \mathtt{j}) \in \iota\big) \bigwedge \left(\mathtt{n} = \left\{ \begin{array}{ll} 0, & \textit{if } (\mathbb{B}(x) \not\sqsubseteq \ell) \\ \mathtt{j}, & \textit{otherwise} \end{array} \right\} \right) \right\}$$

Definition 18. *The difference between two budget maps, ι and ι', written $\iota - \iota'$, is defined as:*

$$\iota - \iota' = \sum_{\mathtt{x} \in \iota} \big(\iota(\mathtt{x}) - \iota'(\mathtt{x})\big)$$

As per LIR, the length of the projected trace is bounded by the total budget deducted. The length of a trace τ is represented as $|\tau|$. This intuition of LIR is formalized in Definition 19.

Definition 19 (Limited information release)**.** *A program c is said to satisfy* limited information release *w.r.t. an adversary at level ℓ if for any given memory store σ, ι, and pc, $\langle \sigma, \iota, c\rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau\rangle$ then $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell} - \iota'{\uparrow_\ell}$*

Theorem 10 shows that the monitored semantics presented in Section 6.3.3 satisfy limited information release. For proving the theorem, the observational equivalence of adversary need to be defined for the various data structures in the semantics.

Definition 20. *Two labeled values* $v_1 = \mathtt{n}_1^{(l_1,\delta_1)}$ *and* $v_2 = \mathtt{n}_2^{(l_2,\delta_2)}$ *are observationally equivalent at level* ℓ*, written* $v_1 \sim_\ell v_2$ *iff either:*

1. $\mathtt{n}_1 = \mathtt{n}_2$, $l_1 = l_2 \sqsubseteq \ell$ *and* $\delta_1 = \delta_2$ *(or)*
2. $\Gamma(v_1) \not\sqsubseteq \ell$ *and* $\Gamma(v_2) \not\sqsubseteq \ell$ *such that* $l_1 = l_2 \sqsubseteq \ell$ *and* $\delta_1 = \delta_2$ *(or)*
3. $\Gamma(v_1) \not\sqsubseteq \ell$ *and* $\Gamma(v_2) \not\sqsubseteq \ell$ *such that* $l_1 \not\sqsubseteq \ell \vee \exists \mathtt{x} \in \delta_1.\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell$ *and* $l_2 \not\sqsubseteq \ell \vee \exists \mathtt{x} \in \delta_2.\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell$

Definition 21. *Two memory stores* σ_1 *and* σ_2 *are observationally equivalent at level* ℓ*, written* $\sigma_1 \sim_\ell \sigma_2$ *iff* $\forall \mathtt{x}.\ \sigma_1(\mathtt{x}) \sim_\ell \sigma_2(\mathtt{x})$.

Definition 22. *Two budget maps* ι *and* ι' *are equivalent at level* ℓ*, written* $\iota \sim_\ell \iota'$*, iff* $\forall \mathtt{x}.\mathbb{B}(\mathtt{x}) \sqsubseteq \ell \implies \iota(\mathtt{x}) = \iota'(\mathtt{x})$.

Definition 23. *Two traces* τ *and* τ' *are equivalent at level* ℓ*, written* $\tau \sim_\ell \tau'$*, iff* $\tau{\uparrow_\ell} = \tau'{\uparrow_\ell}$.

The proof of the theorem employs quite a few properties and lemmas that are described below. Lemmas 10 and 11 prove that in a secret context with respect to an adversary at level ℓ, no declassification occurs that is visible to the adversary. Theorem 9 shows that in a store containing only one secret value with respect to ℓ-level adversary, the length of the trace is bounded by the budget reduced, i.e., the number of bits declassified is bounded by the deduction in budget of that secret value. The theorem generalizes this result to all secrets in the system. The proofs are detailed in Appendix D.

Lemma 9 (Trace-projection)**.** $(\tau_1 :: \tau_2){\uparrow_\ell} = \ \tau_1{\uparrow_\ell} :: \tau_2{\uparrow_\ell}$

Lemma 10. *If* $\langle \sigma, \iota, e\rangle \underset{pc}{\Downarrow} \mathtt{n}^{(m,\delta)}, \iota', \tau$*, and* $\Gamma(pc) \not\sqsubseteq \ell$*, then*
$\iota \sim_\ell \iota'$ *and* $\tau{\uparrow_\ell} = \epsilon$

Lemma 11 (Confinement)**.** *If* $\langle \sigma, \iota, c\rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau\rangle$*, and* $\Gamma(pc) \not\sqsubseteq \ell$*, then*
$\sigma \sim_\ell \sigma'$, $\iota \sim_\ell \iota'$ *and* $\tau{\uparrow_\ell} = \epsilon$

Lemma 12. *If*
$\langle \sigma, \iota, e\rangle \underset{pc}{\Downarrow} \mathtt{n}^{(l,\delta)}, \iota', \tau$ *and* $\exists \mathtt{x} \in \sigma.\Big(\Gamma(\sigma(\mathtt{x})) \not\sqsubseteq \ell \wedge \big(\forall \mathtt{y} \in \sigma.\mathtt{y} \neq \mathtt{x} \wedge \Gamma(\sigma(\mathtt{y})) \sqsubseteq \ell\big)\Big)$,
then $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell}(\mathtt{x}) - \iota'{\uparrow_\ell}(\mathtt{x})$.

Theorem 9 (Limited information release for a single secret)**.** *If*
$\langle \sigma, \iota, c\rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau\rangle$*, and* $\exists \mathtt{x} \in \sigma.\Big(\Gamma(\sigma(\mathtt{x})) \not\sqsubseteq \ell \wedge \big(\forall \mathtt{y} \in \sigma.\mathtt{y} \neq \mathtt{x} \wedge \Gamma(\sigma(\mathtt{y})) \sqsubseteq \ell\big)\Big)$,
then $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell}(\mathtt{x}) - \iota'{\uparrow_\ell}(\mathtt{x})$.

Lemma 13. *If* $\langle \sigma, \iota, e\rangle \underset{pc}{\Downarrow} \mathtt{n}^{(l,\delta)}, \iota', \tau$*, then* $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell} - \iota'{\uparrow_\ell}$.

Theorem 10 (Limited information release)**.** *For any memory store* σ*, budget store* ι *and program* c *if* $\langle \sigma, \iota, c\rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau\rangle$*, then* c *satisfies limited information release for* σ *and* ι

6.5 Soundness and Decoding Semantics

To prove the soundness of the approach, it needs to be shown that the bounds obtained (also for just a single secret) above are well-founded in an information theoretic sense. The trace of declassified

values (when projected to an attacker) are all the bits an adversary can observe about the only secret input given to the program. The projected trace can, thus, be regarded as a code for the information leaked about the secret to the attacker. According to Shannon's source coding theorem [96], for uniquely-decodable codes, the information leaked as computed using Shannon entropy is upper bounded by the average length of the codes. Hence, if it can be proven that the trace projected to an adversary (regarded as a code) is uniquely-decodable then its average length over all executions would be an upper bound on the information leaked by the program as computed by the definition of Shannon entropy.

In order to prove that the projected trace corresponds to a uniquely-decodable code, this section presents a decoding semantics that is almost similar to the LIR semantics from Section 6.3.3, except for the following differences:

1. Decoding semantics is specialized to a fixed adversary, represented as ℓ.
2. Decoding semantics operates on an ℓ-projected memory (where the secret inputs are replaced by $\star$, representing an unknown value as shown in Definition 24) and ℓ-projected budget store (where secret budget values are replaced by 0 as in Definition 17).
3. At declassification points, the decoding semantics reads a value from the trace, which is also ℓ-projected.
4. Decoding semantics completely skips the code that is executed under a pc influenced by $\star$.

Definition 24 (Memory store projection)**.** *The projection of a memory store, σ, w.r.t. an adversary at level ℓ, written $\sigma{\uparrow_\ell}$, is defined as:*

$$\sigma{\uparrow_\ell} = \Pi(\sigma, \ell) = \left\{ \mathtt{x} \;\middle|\; \left(\mathtt{x} = \begin{cases} \mathtt{y}, & \textit{if } (\mathtt{y} \in \sigma) \wedge \left(\Gamma(\sigma(\mathtt{y})) \sqsubseteq \ell\right) \\ \star^{(\bot, \{\mathtt{x}\})}, & \textit{otherwise} \end{cases} \right) \right\}$$

Program configurations for expressions and commands are extended with the projected trace and projections of the memory and budget store, given by $\langle \sigma{\uparrow_\ell}, \iota{\uparrow_\ell}, e, \tau{\uparrow_\ell} \rangle$ and $\langle \sigma{\uparrow_\ell}, \iota{\uparrow_\ell}, c, \tau{\uparrow_\ell} \rangle$, respectively. $\sigma{\uparrow_\ell}$, $\iota{\uparrow_\ell}$, and $\tau{\uparrow_\ell}$ represent the ℓ-projected memory, budget-map and trace, respectively.

The semantics of expression evaluation is shown in Figure 6.4 and is similar to the one presented earlier (Figure 6.2) except for the trace τ it takes as input. The rules generate a new trace τ', which is a suffix of the original trace τ. The main difference shows up in the S-DCOPR rule — when declassification occurs, the declassified value is read from the trace (as opposed to the one computed by local evaluation in the sub-expressions).

The semantics of commands shown in Figure 6.5 is also similar to the rules shown in Figure 6.3 (Section 6.3.3) except for the branching rules. Branching or looping on $\star$ values skips the execution as shown in rules S-IF-ELSE-S and S-WHILE-FS.

The proof for the trace being uniquely-decodable is based on the decoding semantics utilizing the trace generated by the LIR semantics. This requires defining equivalence between the memory stores in the two semantics. The memory equivalence (Definition 26) is defined in terms of value equivalence (Definition 25). The main idea underlying the defintion of value equivalence is that either the both values are equal and observable to the attacker (Definition 25.1) or can never become observable to the attacker even after declassification (Definition 25.2).

$$\textsc{s-const}\ \frac{}{\langle \sigma, \iota, \mathtt{n}, \tau\rangle \underset{pc}{\downarrow} \mathtt{n}^{(\bot,\{\})}, \iota, \tau}$$

$$\textsc{s-var}\ \frac{\sigma(\mathtt{x}) = \mathtt{v}^{(k,\delta_o)} \qquad \delta' = \mathbb{R}(\delta_o, \iota, k_o, \mathbb{B}) \qquad \delta = \delta_o \setminus \delta' \qquad k = k_o \bigsqcup_{\mathtt{x}\in\delta'} \mathbb{L}(\mathtt{x})}{\langle \sigma, \iota, \mathtt{x}, \tau\rangle \underset{pc}{\downarrow} \mathtt{v}^{(k,\delta)}, \iota, \tau}$$

$$\textsc{s-aop}\ \frac{\langle \sigma, \iota, e_1, \tau\rangle \underset{pc}{\downarrow} \mathtt{v}_1^{(k_1,\delta_1)}, \iota_1, \tau_1 \qquad \langle \sigma, \iota_1, e_2, \tau_1\rangle \underset{pc}{\downarrow} \mathtt{v}_2^{(k_2,\delta_2)}, \iota', \tau' \qquad \mathtt{v} = \mathtt{v}_1 \odot \mathtt{v}_2 \qquad k = k_1 \sqcup k_2 \qquad \delta = \delta_1 \cup \delta_2}{\langle \sigma, \iota, (e_1 \odot e_2), \tau\rangle \underset{pc}{\downarrow} \mathtt{v}^{(k,\delta)}, \iota', \tau'}$$

$$\textsc{s-cop}\ \frac{\langle \sigma, \iota, e_1, \tau\rangle \underset{pc}{\downarrow} \mathtt{v}_1^{(k_1,\delta_1)}, \iota_1, \tau_1 \qquad \langle \sigma, \iota_1, e_2, \tau_1\rangle \underset{pc}{\downarrow} \mathtt{v}_2^{(k_2,\delta_2)}, \iota', \tau' \qquad \mathtt{v} = \mathtt{v}_1 \oplus \mathtt{v}_2 \qquad k = k_1 \sqcup k_2 \qquad \delta = \delta_1 \cup \delta_2}{\langle \sigma, \iota, (e_1 \oplus e_2), \tau\rangle \underset{pc}{\downarrow} \mathtt{v}^{(k,\delta)}, \iota', \tau'}$$

$$\textsc{s-dcopr}\ \frac{\begin{array}{c}\langle \sigma, \iota, e_1, \tau\rangle \underset{pc}{\downarrow} \mathtt{v}_1^{(k_1,\delta_1)}, \iota_1, \tau_1 \qquad \langle \sigma, \iota_1, e_2, \tau_1\rangle \underset{pc}{\downarrow} \mathtt{v}_2^{(k_2,\delta_2)}, \iota_2, \tau_2 \\ \mathtt{v}_o = \mathtt{v}_1 \oplus \mathtt{v}_2 \qquad \delta_o = \delta_1 \cup \delta_2 \qquad k_o = k_1 \sqcup k_2 \\ pc \sqsubseteq k_o \\ \delta' = \Delta(\delta_o, k_o, \mathbb{L}) \qquad \delta'' = \mathbb{R}(\delta', \iota_2, k_o, \mathbb{B}) \qquad \delta = \delta' \setminus \delta'' \qquad m = k_o \bigsqcup_{\mathtt{x}\in\delta''} \mathbb{L}(\mathtt{x}) \bigsqcup_{\mathtt{y}\in\delta} \mathbb{B}(\mathtt{y}) \\ \iota' = \mathbb{I}(\iota_2, \delta) \qquad (\tau', \mathtt{v}, k) = \left\{ \begin{array}{ll} (\tau'', \mathtt{v}_t, k_t), & \textit{if } (\delta \neq \emptyset) \wedge (\tau_2 = \mathtt{v}_t^{k_t} :: \tau'') \\ (\tau_2, \mathtt{v}_o, m), & \textit{otherwise} \end{array} \right\}\end{array}}{\langle \sigma, \iota, \mathtt{declassify}(e_1 \oplus e_2), \tau\rangle \underset{pc}{\downarrow} \mathtt{v}^{(k,\{\})}, \iota', \tau'}$$

$$\textsc{s-dcopn}\ \frac{\langle \sigma, \iota, e_1, \tau\rangle \underset{pc}{\downarrow} \mathtt{v}_1^{(k_1,\delta_1)}, \iota_1, \tau_1 \qquad \langle \sigma, \iota_1, e_2, \tau_1\rangle \underset{pc}{\downarrow} \mathtt{v}_2^{(k_2,\delta_2)}, \iota_2, \tau_2 \qquad \mathtt{v} = \mathtt{v}_1 \oplus \mathtt{v}_2 \qquad \delta = \delta_1 \cup \delta_2 \qquad k = k_1 \sqcup k_2 \qquad pc \not\sqsubseteq k}{\langle \sigma, \iota, \mathtt{declassify}(e_1 \oplus e_2), \tau\rangle \underset{pc}{\downarrow} \mathtt{v}^{(k,\delta)}, \iota_2, \tau_2}$$

Figure 6.4: Decoding semantics of expressions

$$\textsc{s-skip}\ \frac{}{\langle \sigma, \iota, \texttt{skip}, \tau\rangle \underset{pc}{\downarrow} \langle \sigma, \iota, \tau\rangle}$$

$$\textsc{s-assn}\ \frac{pc \sqsubseteq \Gamma(\sigma(\texttt{x})) \qquad \langle \sigma, \iota, e, \tau\rangle \underset{pc}{\downarrow} \texttt{n}^{(m,\delta')}, \iota', \tau' \qquad k = pc \sqcup m \qquad \delta = \Delta(\delta', k, \mathbb{L})}{\langle \sigma, \iota, (\texttt{x} := e), \tau\rangle \underset{pc}{\downarrow} \langle \sigma[\texttt{x} \mapsto \texttt{n}^{(k,\delta)}], \iota', \tau'\rangle}$$

$$\textsc{s-seq}\ \frac{\langle \sigma, \iota, c_1, \tau\rangle \underset{pc}{\downarrow} \langle \sigma', \iota', \tau_1\rangle \qquad \langle \sigma', \iota', c_2, \tau_1\rangle \underset{pc}{\downarrow} \langle \sigma'', \iota'', \tau'\rangle}{\langle \sigma, \iota, c_1; c_2, \tau\rangle \underset{pc}{\downarrow} \langle \sigma'', \iota'', \tau'\rangle}$$

$$\textsc{s-if-else-n}\ \frac{\langle \sigma, \iota, e, \tau\rangle \underset{pc}{\downarrow} \texttt{v}^{(k_o,\delta)}, \iota', \tau_1 \qquad \texttt{v} \neq \star \qquad i = \left\{ \begin{array}{ll} 1, & \textit{if } (\texttt{v} = \texttt{true}) \\ 2, & \textit{otherwise} \end{array} \right\} \qquad k = k_o \bigsqcup_{\texttt{x} \in \delta} \mathbb{L}(\texttt{x}) \qquad \langle \sigma, \iota', c_i, \tau_1\rangle \underset{pc \sqcup k}{\downarrow} \langle \sigma', \iota'', \tau'\rangle}{\langle \sigma, \iota, (\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2), \tau\rangle \underset{pc}{\downarrow} \langle \sigma', \iota'', \tau'\rangle}$$

$$\textsc{s-if-else-s}\ \frac{\langle \sigma, \iota, e, \tau\rangle \underset{pc}{\downarrow} \star^{(\_,\_)}, \iota', \tau'}{\langle \sigma, \iota, (\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2), \tau\rangle \underset{pc}{\downarrow} \langle \sigma, \iota', \tau'\rangle}$$

$$\textsc{s-while-fs}\ \frac{\langle \sigma, \iota, e, \tau\rangle \underset{pc}{\downarrow} \texttt{v}^{(\_,\_)}, \iota', \tau' \qquad (\texttt{v} = \star) \vee (\texttt{v} = \texttt{false})}{\langle \sigma, \iota, \texttt{while } e \texttt{ do } c, \tau\rangle \underset{pc}{\downarrow} \langle \sigma, \iota', \tau'\rangle}$$

$$\textsc{s-while-t}\ \frac{\langle \sigma, \iota, e, \tau\rangle \underset{pc}{\downarrow} \texttt{true}^{(k_o,\delta)}, \iota_1, \tau_1 \qquad k = k_o \bigsqcup_{x \in \delta} \mathbb{L}(\texttt{x}) \qquad \langle \sigma, \iota_1, c; \texttt{while } e \texttt{ do } c, \tau_1\rangle \underset{pc \sqcup k}{\downarrow} \langle \sigma', \iota', \tau_2\rangle}{\langle \sigma, \iota, \texttt{while } e \texttt{ do } c, \tau\rangle \underset{pc}{\downarrow} \langle \sigma'', \iota'', \tau''\rangle}$$

Figure 6.5: Decoding semantics of commands

Definition 25. *Two values* $\texttt{v}_o$ *and* $\texttt{v}_s$*, where* $\texttt{v}_o$ *is a value in the original memory store and* $\texttt{v}_s$ *is a value in the projected memory store for simulation, are observationally equivalent at level* ℓ*, written* $\texttt{v}_o \simeq_\ell \texttt{v}_s$ *iff*

1. $\texttt{v}_o = \texttt{n}_o^{(k,\delta)}$, $\texttt{v}_s = \texttt{n}_s^{(k',\delta')}$, $\texttt{n}_o = \texttt{n}_s$ *and* $(k = k') \sqsubseteq \ell$ *and* $\delta = \delta'$ *(or)*
2. $\texttt{v}_o = \texttt{n}_o^{(k,\delta)}$, $\Gamma(\texttt{v}_o) \not\sqsubseteq \ell$, $\texttt{v}_s = \star^{(k',\delta')}$ *and either:*
   (a) $k \not\sqsubseteq \ell \ \wedge\ k' \not\sqsubseteq \ell$
   (b) $k \sqsubseteq \ell \ \wedge\ \exists \texttt{x} \in \delta.(\mathbb{B}(\texttt{x}) \not\sqsubseteq \ell) \ \wedge\ k' \not\sqsubseteq \ell$
   (c) $(k = k') \sqsubseteq \ell \ \wedge\ \delta = \delta'$

Definition 26. *Given a memory store σ and a projected memory store $\sigma'{\uparrow_\ell}$, their equivalence $\simeq_\ell$ at level ℓ is defined as:* $\forall \mathtt{x}.\ \sigma(\mathtt{x}) \simeq_\ell \sigma'{\uparrow_\ell}(\mathtt{x})$

Under a given memory σ and budget store ι, if a program executes to completion under the LIR semantics and generates a trace τ, then for any chosen adversary ℓ the same program executing under ℓ-projected memory ($\sigma{\uparrow_\ell}$) and budget store ($\iota{\uparrow_\ell}$) along with the projected trace $\tau{\uparrow_\ell}$ results in an observationally equivalent final memory stores obtained at the end of both the executions with respect to the adversary ℓ (Theorem 11).

Theorem 11 (Simulation Theorem)**.** *If* $\langle \sigma, \iota, c\rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau\rangle$, *then*
$\langle \sigma{\uparrow_\ell}, \iota{\uparrow_\ell}, c, \tau{\uparrow_\ell}\rangle \underset{pc}{\downarrow} \langle \sigma'', \iota'', \tau''\rangle$ *such that* $\sigma' \simeq_\ell \sigma''$, $\iota' \sim_\ell \iota''$ *and* $\tau'' = \epsilon$

The proof of the theorem is detailed in Appendix D. It uses results from various lemmas shown below. Lemmas 14, 15, and 16 prove some trace and store related properties.

Lemma 14. *If* $\langle \sigma, \iota, e, \tau\rangle \underset{pc}{\downarrow} v, \iota', \tau'$, *then* $\langle \sigma, \iota, e, \tau :: \tau^r\rangle \underset{pc}{\downarrow} v, \iota', \tau' :: \tau^r$

Lemma 15. *If* $\langle \sigma, \iota, c, \tau\rangle \underset{pc}{\downarrow} \sigma', \iota', \tau'$, *then* $\langle \sigma, \iota, c, \tau :: \tau^r\rangle \underset{pc}{\downarrow} \sigma', \iota', \tau' :: \tau^r$

Corollary 3 (Trace Reduction)**.** *If* $\langle \sigma, \iota, c, \tau\rangle \underset{pc}{\downarrow} \sigma', \iota', \epsilon$, *then* $\langle \sigma, \iota, c, \tau :: \tau^r\rangle \underset{pc}{\downarrow} \sigma', \iota', \tau^r$

Lemma 16 (Equivalence)**.** *If* $\sigma \sim_\ell \sigma'$ *and* $\sigma \simeq_\ell \sigma''$, *then* $\sigma' \simeq_\ell \sigma''$

Lemma 17 (Expression Simulation)**.** *If* $\langle \sigma, \iota, e\rangle \underset{pc}{\Downarrow} v, \iota', \tau$, *then*
$\langle \sigma{\uparrow_\ell}, \iota{\uparrow_\ell}, e, \tau{\uparrow_\ell}\rangle \underset{pc}{\downarrow} v', \iota'', \tau'$ *such that* $v \simeq_\ell v'$, $\iota'{\uparrow_\ell} = \iota''$, *and* $\tau' = \epsilon$

As a sanity check, it is shown that in the case when the budgets of all the secrets in the store are 0, then the semantics satisfy non-interference (Corollary 4).

Corollary 4 (Non-interference)**.** *If*
$\langle \sigma_1, \iota_1, c\rangle \underset{pc}{\Downarrow} \langle \sigma_1', \iota_1', \tau_1\rangle$, $\langle \sigma_2, \iota_2, c\rangle \underset{pc}{\Downarrow} \langle \sigma_2', \iota_2', \tau_2\rangle$,
$\sigma_1 \sim_\ell \sigma_2$, *and* $\iota_1{\uparrow_\ell} = \iota_2{\uparrow_\ell} = 0$, *then* $\sigma_1' \sim_\ell \sigma_2'$.

Part V

Application to a Web Browser

Chapter 7

Information Flow Policies for Web Browsers

Many approaches [25, 34, 58, 85], have been proposed for enforcing dynamic information flow control in web browsers, but currently lack adequate support for specifying policies conveniently. Flowfox [100] provides a rich policy framework but all websites are subject to the same policy, and the underlying IFC technique, secure multi-execution [46], does not handle shared state soundly. COWL [98] uses coarse-grained isolation, allowing scripts' access to either remote domains or the shared state, but not both. This requires significant code changes when both are needed simultaneously (see Chapter 9 for more details).

This chapter presents WebPol, a policy framework that allows a webpage developer to release data selectively to third-party scripts (to obtain useful functionality), yet control what the scripts can do with the data. WebPol policies label sensitive content (page elements and user-generated events) at source, and selectively declassify them by specifying where (to which domains) the content and its derivatives can flow. Host page developers specify WebPol policies in JavaScript, a language already familiar to them. WebPol is integrated with the instrumentation of the dynamic IFC framework for WebKit presented thus far (WebPol integrates with any taint-based IFC solution to overcome the shortcomings listed above). The expressiveness of WebPol policies is demonstrated through examples and by applying WebPol to two real websites.

7.1 Policy Component for Web Browsers

IFC is a broad term for techniques that control the flow of sensitive information in accordance with pre-defined policies. Sensitive information is information derived from sources that are confidential

The content of this chapter is based on the work published as part of the paper, "WebPol: Fine-grained Information Flow Policies for Web Browsers" [27]

or private. Any IFC system has two components—the *policy component* and the *enforcement component*. The policy component allows labeling of private information sources. The label on a source specifies how private information from that source can be used and where it can flow. The collection of rules for labeling is called the policy. The enforcement component enforces policies. This could, for example, be the dynamic taint tracking approach described earlier. WebPol contributes a policy component to complement existing work on enforcement components in web browsers.

In the context of webpages, data sources are objects generated in response to user events like the content of a password box generated due to key presses or a mouse click on a sensitive button, and data obtained in a network receive event. In WebPol, data sources can be labeled with three kinds of labels, in increasing order of confidentiality: 1) the label `public` represents non-sensitive data, 2) for each domain `domain`, the label `domain` represents data private to the domain; such data's flow should be limited only to the browser and servers belonging to `domain` and its subdomains, and 3) the label `local` represents very confidential data that must never leave the browser. These labels are ordered $\texttt{public} < \texttt{domain}_i < \texttt{local}$. These labels are fairly expressive. For example, labeling a data source with the domain of the hosting page restricts its transfer to only the host, and prevents exfiltration to third-parties. Labeling a data source with the domain of a service provider such as an analytics provider allows transfer to only that service.

Since most data on a webpage is not sensitive, it is reasonable to label data sources `public` by default and only selectively assign a different label. WebPol uses this blacklisting approach. Two nuances of source labeling are noteworthy. The first is its fine granularity. Not all objects generated by the same class of events have the same label. For instance, characters entered in a password field may have the domain label of the hosting page, limiting their flow only to the host, but characters entered in other fields may be accessible to third-party advertising or analytics scripts unrestricted. This leads to the following requirement on the policy component.

Requirement 1: The policy component must allow associating different policies with different elements of the page.

The second is that the label of an object can be dynamic, i.e., history-dependent. Consider a policy that hides from an analytics script how many times a user clicked within an interactive panel, but wants to share whether or not the user clicked at least once. The label of a click event on the panel is `public` the first time the user clicks on it and private afterwards and, hence, it depends on the history of user interaction. This yields the following requirement on the policy component.

Requirement 2: Labels may be determined dynamically. This requirement means that labels must be set by *trusted policy code* that is executed on-the-fly and that has local state.

7.2 WebPol policy model

WebPol works on a browser that has already been augmented with IFC enforcement based on taint tracking. It provides a framework that allows setting labels at fine-granularity, thus expressing and enforcing rich policies. This section explains the design of WebPol. WebPol prevents under-the-hood

exfiltration of sensitive data that has been provided to third-party scripts for legitimate reasons. So, third-party scripts are not trusted but code from the host domain is trusted. WebPol's policies are agnostic to specific channels of leak. However, current IFC enforcements in browsers track only explicit and implicit flows. Consequently, leaks over other channels such as timing and memory-usage are currently out of scope.

7.2.1 Policies as event handlers

The first question in the design of WebPol is who should specify policies. Since the goal here is to prevent exfiltration of data by third-party scripts and it is the developer of the host page who bootstraps the inclusion of scripts and best understands how data on the page should be used, it is natural and pragmatic to have the developer specify policies, possibly as part of the page itself.

The next question is how the developer specifies policies. To answer this, recall the two requirements identified in Section 7.1 — it should be possible to specify different policies on different page elements and policies should be allowed to include code that is executed on-the-fly to generate labels. Considering the fact that sensitive data is usually generated by input events, it is clear that policies should *be* page element-specific (trusted) code that is executed after events have occurred (this code labels event-generated data). Fortunately, web browsers provide exactly this abstraction in the form of event handlers! So, the event-handling logic in web browsers is extended to express WebPol policies. This allows leveraging a lot of the existing browser logic for event handler installation, parsing and event dispatch in interpreting policies. The rest of this section explains how this is done.

In WebPol, policies are special event handlers, specified using a special marker in the HTML source of the hosting page. These special handlers, called *policy handlers*, follow standard JavaScript syntax, can be attached to any page element, listening for any event and, like other handlers, are triggered every time the event is dispatched on the element or any of its descendants. However, unlike other handlers, the sole goal of policy handlers is to assign labels to other sensitive objects, including the event being dispatched. To allow the policy handlers to do this, the browser is slightly modified to afford these handlers two special privileges:

- Policy handlers can execute two new JavaScript API functions that set labels on other objects. No other JavaScript code can execute these two functions. These functions are described later.
- During event dispatch all applicable policy handlers are executed before ordinary handlers. This ensures that labels are set before ordinary handlers (including those of third-party scripts) execute.

To maintain the integrity of the policies, policy handlers must be included in the HTML source of the page directly. They *cannot* be installed dynamically by JavaScript code. Otherwise, third-party scripts could install policy handlers that set very permissive labels. Also, if a DOM element has a policy handler, third-party scripts are disallowed from detaching that element or moving it elsewhere, as that can change the interpretation of the policy. Similarly, changing the attributes of such an element is restricted.

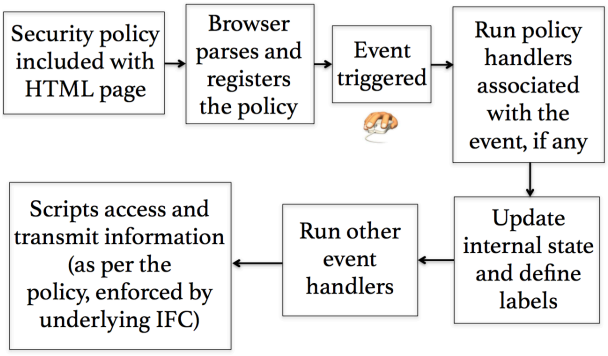

Figure 7.1: Workflow of the WebPol policy model

Since different policy handlers can be associated with different elements, Requirement 1 is satisfied. Moreover, policy handlers are ordinary JavaScript code, so they can also maintain local state in private variables, thus satisfying Requirement 2.

The workflow of policy interpretation in WebPol is shown in Figure 7.1. Briefly, the steps are:

1. The web page developer specifies the policy in the host HTML page in the form of special event handlers.
2. The browser parses the policy and registers its handlers (mostly like usual handlers, but with the two special privileges mentioned above).
3. When an event dispatches, listening policy handlers are executed first.
4. These policy handlers set labels on objects affected by the event, including the event object itself. They may also update any local state they maintain.
5. The remaining event handlers are dispatched as usual. The IFC enforcement in the browser enforces all labels that have been set by the policy handlers (during any prior event's dispatch), thus preventing any data leak in contravention of the labels.

7.2.2 Integration with the web browser

WebPol needs minor modifications to the browser to parse and interpret policies and to expose additional JavaScript API functions to set labels.

HTML and event dispatch changes. WebPol adds an HTML extension to differentiate policy code from other JavaScript code. Concretely, the browser's parser is changed to interpret any script file with the extension `.policy` included directly in the host page as a policy. If such a *policy script* installs a handler, it is treated as a policy handler. Additionally, a policy script can set labels on the page's global variables and DOM elements (like password fields). If a script does this, it should be included in the host page before third-party scripts that use those variables. WebPol also requires a small change to the browser's event dispatch mechanism to execute policy handlers before other handlers.

```
var p = document.getElementById("pwd");
p.addEventListener("keypress", function (e){
  var score = checkPwdStrength(p.value);
  document.getElementById("pwdStrength").innerText = score;
  new Image().src = "http://stealer.com/pwd.jsp?pwd="+p +score;
});
```

Listing 7.1: Password strength checking script that leaks the password

Label-setting APIs. WebPol exposes two new JavaScript API functions to set labels. These functions can be called only by the policy code in `.policy` files and handlers installed by such files (the browser is modified to enforce this).

The function `setLabel(label)` sets the label of the object on which it is called to `label`. As explained earlier, `label` can be `public`, a domain name, or `local` (the default is `public`). Once an object's label is set, it is enforced by the underlying IFC enforcement. The special label `HOST` is a proxy for the domain of the host page.

The function `setContext(label)` can be called only on an event object. It restricts the *visibility* of the event to label `label` and higher. In simple terms, if `label` is a domain, then only that domain can ever learn that this event occurred, whereas if `label` is `local`, then no domain can ever learn that this event occurred. Technically, this is accomplished by setting the pc of the event handlers running during the dispatch to `label`, which ensures that their side-effects (writes to DOM and network communication) are labeled `label` or higher.

As opposed to `setLabel`, which makes individual data objects (like password fields) private, `setContext` makes the *existence* of an event private. This is useful. For instance, clicking on the "politics" section of a news feed might indicate that the user is interested in politics, which may be private information, so the page may want to hide even the existence of click events from third-party scripts. (The distinction between the privacy of event content and event occurrence has been previously described by Rafnsson and Sabelfeld [83].)

7.3 Expressiveness of WebPol

The expressiveness of WebPol policies is illustrated through the following examples.

7.3.1 Example 1: Password strength checker

Many websites deploy *password strength checkers* on pages where users set new passwords. A password strength checker is an event handler from a third-party library that is triggered each time the user enters a character in the new password field. The handler provides visual feedback to the user about the strength of the password entered so far. Strength checkers usually check the length of the password and the diversity of characters used. Consequently, they do not require any network communication. However, standard browser policies cannot enforce this and the password strength

```
1  function currencyConverter() {
2          var toCur = document.getElementById("to").value;
3          var xh = new XMLHttpRequest();
4          xh.onreadystatechange = function() {
5                  if (xh.readyState == 4) {
6                          currencyRate = eval(xhttp.responseText);
7                          var aAmt = document.getElementById("amt").value;
8                          var convAmt = aAmt * currencyRate;
9                          document.getElementById("camt").innerHTML = convAmt;
10                         xh.open("GET","http://currConv.com/amount.jsp?atc=" + aAmt);
11                         xh.send(); }}
12         xh.open("GET","http://currConv.com/conv.jsp?toCur=" + toCur, true);
13         xh.send(); }
```

Listing 7.2: Currency converter script that leaks a private amount

checker can easily leak the password if it wants to. Listing 7.1 shows such a "leaky" password checker. The checker installs a listener for keypresses in the password field (line 2). In response to every keypress, the listener delivers its expected functionality by checking the strength of the password and indicating this to the user (lines 3, 4), but then it leaks out the password to `stealer.com` by requesting an image at a url that includes the password (lines 5, 6).

With WebPol, the developer of the host webpage can prevent any exfiltration of the password by including the policy script:

```
document.getElementById("pwd").setLabel("HOST");
```

This policy sets the label of the password field to the host's own domain using the function `setLabel()`. Subsequently, the IFC enforcement restricts all outgoing communication that depends on the password field to the host.

Conceptually, this example is simple because it does not really leverage the fine-granularity of WebPol policies and fine-grained dynamic IFC. Here, the third-party script does not need any network communication for its intended functionality and, hence, simpler confinement mechanisms that prohibit a third-party script from communicating with remote servers would also suffice. The next example is a scenario where the third-party script legitimately needs remote communication, which leverages the fine-granularity of WebPol policies and fine-grained dynamic IFC.

7.3.2 Example 2: Currency conversion

Consider a webpage from an e-commerce website which displays the cost of an item that the user intends to buy. The amount is listed in the site's native currency, say US dollars (USD), but for the user's convenience, the site also allows the user to see the amount converted to a currency of his/her choice. For this, the user selects a currency from a drop-down list. A third-party JavaScript library reads both the USD amount and the second currency, converts the amount to the second currency and inserts it into the webpage, next to the USD amount. The third-party script fetches the current

```
1 var p = document.getElementbyId("sect_name");
2 p.addEventListener("click",function(event){
3   event.setLabel("HOST"); });
```

Listing 7.3: Policy that allows counting clicks but hides details of the clicks

```
1 clickCount = 0;
2 var p = document.getElementbyId("sect_name");
3 p.addEventListener("click",function clkHdlr(e){ clickCount += 1; });
```

Listing 7.4: Analytics script that counts clicks

conversion rate from its backend service at `currConv.com`. Consequently, it must send the *name* of the second currency to its backend service, but must not send the amount being converted (that is private information). The web browser's same-origin policy has been relaxed (using, say, CORS [3]) to allow the script to talk to its backend service at `currConv.com`. The risk is that the script can now exfiltrate the private amount. Listing 7.2 shows a leaky script that does this. On line 11, the script makes a request to its backend service passing to it the two currencies. The callback handler (lines 4–11) reads the amount from the page element `amt`, converts it and inserts the result into the page (lines 6–9). Later, it leaks out the amount to the backend service on line 10, in contravention of the intended policy.

With WebPol, this leak can be prevented with the following policy that sets the label of the amount to the host only:

```
document.getElementById("amt").setLabel("HOST")
```

This policy will prevent exfiltration of the amount and will not interfere with the requirement to exfiltrate the second currency. Importantly, no modifications are required to a script that does not try to leak data (e.g., the script obtained by dropping the leaky line 10 of Listing 7.2).

7.3.3 Example 3: Web analytics

To better understand how users interact with their websites, web developers often include third-party analytics scripts that track user clicks and keypresses to generate useful artifacts like page heat-maps (which part of the page did the user interact with most?). Although a web developer might be interested in tracking only certain aspects of their users' interaction, the inclusion of the third-party scripts comes with the risk that the scripts will also record and exfiltrate other private user behavior (possibly for monetizing it later). Using WebPol, the web developer can write precise policies on which user events an analytics script can access and when. Several examples of this are shown.

To allow a script to only count the number of occurrences of a class of events (e.g., mouse clicks) on a section of the page, but to hide the details of the individual events (e.g., the coordinates of every individual click), the web developer can add a policy handler on the top-most element of the section

```
1 var alreadyClicked = false;
2 var p = document.getElementById("sect_name");
3 p.addEventListener("click",function(event){
4   if (alreadyClicked = true)
5      event.setContext("HOST");
6   else {
7      alreadyClicked = true;
8      event.setLabel("HOST");
9 }});
```

Listing 7.5: Policy that only tracks whether a click happened or not

```
1 document.body.addEventListener("keypress", function(event){
2     var o = window.getComputedStyle(event.target).getPropertyValue("opacity");
3     if (o < 0.5)
4          event.setLabel("HOST");
5 });
```

Listing 7.6: Example policy to prevent overlay-based stealing of keystrokes

to set the label of the individual event objects to `HOST`. This prevents the analytics script's listening handler from examining the details of individual events, but since the handler is still invoked at each event, it can count their total number. Listings 7.3 and 7.4 show the policy handler and the corresponding analytics script that counts clicks in a page section named `sect_name`.

Next, consider a restriction of this policy, which allows the analytics script to learn only whether or not *at least one* click happened in the page section, completely hiding clicks beyond the first. This policy can be represented in WebPol using a local state variable in the policy to track whether or not a click has happened and the function `setContext()`. Listing 7.5 shows the policy. The policy uses a variable `alreadyClicked` to track whether or not the user has clicked in the section. Upon the user's first click, the policy handler sets the event's label to the host's domain (line 8). This makes the event object private but allows the analytics handler to trigger and record the occurrence of the event. On every subsequent click, the policy handler sets the event's *context* to the host domain using `setContext()` (line 5). This prevents the analytics script from exfiltrating any information about the event, including the fact that it occurred.

Finally, note that a developer can subject different page sections to different policies by attaching different policy handlers to them. The most sensitive sections may have a policy that unconditionally sets the event context to the host's, effectively hiding all user events in those sections. Less sensitive sections may have policies like those of Listings 7.5 and 7.3. Non-sensitive sections may have no policies at all, allowing analytics scripts to see all events in them.

7.3.4 Example 4: Defending against overlay-based attacks

To bypass WebPol policies, an adversarial script may "trick" a user using transparent overlays. For example, suppose a script wants to exfiltrate the contents of a password field that is correctly protected by a WebPol policy. The script can create a transparent overlay on top of the password field. Any password the user enters will go into the overlay, which *isn't* protected by any policy and, hence, the script can leak the password.

Such attacks can be prevented easily in WebPol using a *single* policy, attached to the top element of the page, that labels data entered into all significantly transparent overlays as `HOST`. Listing 7.6 shows such a policy. This particular policy labels all keypress events on elements of opacity below 0.5 as `HOST`, thus preventing their exfiltration. The threshold value 0.5 can be changed, and the policy can be easily extended to other user events like mouse clicks.

7.3.5 Summary of WebPol expressiveness

The security community has extensively studied several aspects of policy labeling, colloquially called the *dimensions of declassification* (see [95] for a survey). Broadly speaking, WebPol policies cover three of these dimensions—the policies specify what data is declassified (dimension: what), to which domains (dimension: to whom) and under what state (dimension: when). "What data is declassified" is specified by selectively attaching policies to elements of the page. "Which domains get access" is determined directly by the labels that the policy sets. Finally, labels generated by policy handlers can depend on state, as illustrated in Listing 7.5.

There are two other common dimensions of labeling—who can label the data (dimension: who) and where in the code can the labels change (dimension: where). These dimensions are fixed in WebPol due to the specifics of the problem: All policies are specified by the host page in statically defined policies.

Chapter 8

Policy Implementation and Evaluation in an IFC-enabled Browser

The ideas presented in the earlier chapters are incorporated into WebKit, the browser engine used in Safari. The instrumentation is built on top of the previous work enforcing dynamic IFC [25, 64, 85], which instruments the three main components of the engine — JavaScript bytecode interpreter, the document object model (DOM) engine, and the event handling mechanism. Their instrumentation is briefly described below.

The authors instrument WebKit's JavaScript bytecode interpreter (JavaScriptCore) to implement the IFC semantics formalized and presented earlier for enforcing dynamic IFC for JavaScript. In WebKit, bytecode is generated by a source-code compiler and organized into code blocks. Each code block is a sequence of bytecodes with line numbers and corresponds to the instructions for a function or an `eval` statement. A code block is generated when a function is created or an `eval` is executed. Their instrumentation performs control flow analysis on a code block when it is created and generates a CFG for it before it starts executing. The IPDs of its nodes are calculated by static analysis of its bytecode; they are computed using an algorithm by Lengauer and Tarjan with CFG as an input to the algorithm [69].

For exceptions, the synthetic exit node is added to the CFG along with the edges as described earlier in Section 5.2. Additional changes are requried to the compiler to make it compliant with the instrumentation. The modification is to emit a slightly different, but functionally equivalent bytecode sequence for `finally` blocks; this is needed for accurate computation of IPDs.

Labels in their instrumentation is a word size bit-set (currently 64 bits); each bit in the bit-set represents label from a distinct domain (like google.com). Join on labels is simply bitwise or. Their instrumentation adds a label to all data structures, including registers, object properties and scope

The content of this chapter is based partly on the work published as part of the paper, "WebPol: Fine-grained Information Flow Policies for Web Browsers" [27]

chain pointers, adds code to propagate explicit and implicit labels and implements the permissive upgrade check. Additionally, all native JavaScript methods in the Array, RegExp, and String objects are instrumented [85]. The formalization of the bytecodes, the semantics of its bytecode interpreter with the instrumentation of dynamic IFC, and the proof of correctness of instrumentation of the bytecodes with the IFC semantics is shown in [26]. Security labels are also attached to every node in the DOM graph and all its properties, including pointer to other nodes. Appropriate IFC checks are added in the native C code implementing all DOM APIs up to Level 3. Input events on webpages like mouse clicks, key presses and network receives trigger JavaScript functions called handlers. The event handling logic of a browser is complex. Each input event can trigger handlers registered not just on the node on which the event occurs (e.g., the button which is clicked), but also on its ancestors in the HTML parse tree. This is called *event dispatch.* They modify the event handling loop, labeling every event and event handler based on their formalization of the event handling loop with the IFC checks [85].

This chapter describes the implementation of the LIR policy (Chapter 6) and WebPol(Chapter 7) on top of the existing IFC-instrumentation.

8.1 Implementation and Evaluation of WebPol

8.1.1 Implemenation of WebPol

WebPol is prototyped in WebKit on top of the prior IFC enforcement in WebKit described above [25, 64, 85]. To implement WebPol, the HTML parser was modified to distinguish policy files (extension `.policy`) from other JavaScript files and to give policy code extra privileges. Two new JavaScript API functions — `setLabel()` and `setContext()` — were added. Finally, the event dispatch logic was modified to trigger policy handlers before other handlers. In all, 25 lines in the code of the parser were modified, 60 lines for the two new API functions were added and 110 lines in the event dispatch logic were modified. Thus, implementing WebPol has low overhead, and can be ported to other browsers easily.

8.1.2 Evaluation of WebPol

The goal of the evaluation is two-fold. The first goal is to measure the overhead of the system with IFC enforcement and WebPol, both in parsing and installing policies during page load and for executing policy handlers later. This is done by running a few benchmarks, by measuring the overhead for the examples presented in Chapter 7.3, and for two real-world websites. Second, to understand whether WebPol can be used easily, WebPol policies are applied to two real-world websites. All the experiments were performed on a 3.2GHz Quad-core Intel Xeon processor with 8GB RAM, running Mac OS X version 10.7.4 using Safari 6.0. The implementation and evaluation is done on WebKit nightly build $\#r122160$. As the existing IFC instrumentation does not handle JIT, JIT support was disabled in all the experiments.

	JavaScript Execution Time			Page Load Time		
Example #	**Base**	**IFC**	**WebPol**	**Base**	**IFC**	**WebPol**
Example 1	2430	2918 (+20.1%)	2989 (+1.9%)	16	17 (+6.3%)	19 (+12.5%)
Example 2	3443	4361 (+26.7%)	5368 (+29.2%)	41	43 (+4.9%)	46 (+7.2%)
Example 3 (count)	1504	1737 (+15.5%)	1911 (+11.6%)	24	25 (+4.2%)	31 (+25.0%)
Example 3 (presence)	1780	2095 (+17.7%)	2414 (+18.9%)	26	28 (+7.7%)	30 (+7.7%)

Table 8.1: Performance of examples from Section 7.3. All time in ms. The numbers in parenthesis are additional overheads relative to **Base**.

8.1.2.1 Performance Overheads on Synthetic Examples

To measure the instrumentation's runtime overhead, four examples from Chapter 7.3 (Examples 1, 2 and the two sub-examples of Example 3) were tested in three different configurations: **Base**—uninstrumented browser, no enforcement; **IFC**—existing instrumented browser with IFC checks, but no policy handlers (everything is labeled public); **WebPol**—instrumented browser running policy handlers.

JavaScript execution time: The overheads of executing policy handler code were measured by interacting with all four programs manually by entering relevant data and performing clicks a fixed number of times. For each of these configurations, the total time spent *only in executing JavaScript* was measured, including scripts and policies loaded initially with the page and the scripts and policies executed in response to events. The difference between **IFC** and **Base** run times is the overhead of dynamic IFC, while the difference between the **WebPol** and **IFC** run times is the overhead of evaluating policy handlers. Since only JavaScript execution time is measured and there are no time-triggered handlers in these examples, variability in the inter-event gap introduced by the human actor does not affect the measurements.

The left half of Table 8.1 shows these observations. All numbers are averages of 5 runs and the standard deviations are all below 7%. Taint-tracking (**IFC**) adds overheads ranging from 15.5% to 26.7% over **Base**. To this, policy handlers (**WebPol**) adds overheads ranging from 1.9% to 29.2%. The **WebPol** overheads are already modest, but this is also a very challenging (conservative) experiment for WebPol. The scripts in both sub-examples of Example 3 do almost nothing. The scripts in Examples 1 and Example 2 are slightly longer, but are still much simpler than real scripts. On real and longer scripts, the relative overheads of evaluating the policy handlers is significantly lower as shown later. Moreover, the baseline in this experiment does not include other browser costs, such as the cost of page parsing and rendering, and network delays. Compared to those, both **IFC** and **WebPol** overheads are negligible.

Page load time: The time taken for loading the initial page (up to the DOMContentLoaded event) was measured separately. The difference between **sys** and **IFC** is the overhead for parsing and load-

```
1 document.getElementById("passwordPwd").setLabel("secret");
2 document.getElementById("passwordTxt").setLabel("secret");
3 var x = document.getElementsByTagName("div");
4 var i = 0;
5 for (i = 0; i < x.length; i++)
6     x[i].setLabel("secret");
```

Listing 8.1: Policy code for password strength checking website

```
1  var x = document.getElementsByClassName("user"); // username
2  var y = document.getElementsByClassName("pwd"); // password
3  for (i = 0; i < x.length; i++) {
4      x[i].addEventListener("keypress", function(event){
5      event.setLabel("HOST");
6      });}
7  for (i = 0; i < y.length; i++) {
8      y[i].addEventListener("keypress", function(event){
9      event.setLabel("HOST");
10     });}
```

Listing 8.2: Policy code for bank login website with an analytics script

ing policies. The right half of Table 8.1 shows these observations. All numbers are the average of 20 runs and standard deviations were below 8%. WebPol overheads due to policy parsing and loading range from 7.2% to 25% (last column). When the overheads due to taint tracking (column **IFC**) are added, the numbers increase to 12.1% to 29.2%. Note that page-load overheads are incurred only once on every page (re-)load.

8.1.2.2 Policies on Real-world Websites.

Further, WebPol was evaluated by writing policies for two real-world applications—a website that deploys a password-strength checker (similar to Example 1) and a bank login page that includes third-party analytics scripts (similar to Example 3). The WebPol policies specified for the password strength checking website and the bank website with an analytics script are shown in Listings 8.1 and 8.2, respectively. The code on lines 3–6 of Listing 8.1 allows the strength-checking script to write back the visual indicator of password strength to the host page's DOM.

Experience writing policies: In both cases, meaningful policies could be specified easily after understanding the code, suggesting that WebPol policies can be (and should be) written by website developers. The policy for the password-strength checker is similar to Listing 7.1 and prevents the password from being leaked to third-parties. Additional policy code of 4 lines was needed to allow the script to write the results of the password strength check (which depends on the password) into the host page. The analytics script on the bank website communicates all user-behavior to its server. The policy specified disallows exfiltration of keypresses on the username and the password text-boxes to third-parties.

	JavaScript Execution Time			Page Load Time		
Website	**Base**	**IFC**	**WebPol**	**Base**	**IFC**	**WebPol**
Password	79.5	115.5 (+45.3%)	126 (+13.2%)	303	429 (+41.6%)	441 (+4.0%)
Analytics	273.4	375.1 (+37.2%)	386.1 (+4.0%)	2151	2422 (+12.6%)	2499 (+3.6%)

Table 8.2: Performance on two real-world websites. All time in ms. The numbers in parenthesis are additional overheads relative to **Base**.

Performance overheads: The performance overheads on the two websites were also measured, in the same configurations as for the synthetic examples. Table 8.2 shows the results. On real-world websites, where actual computation is long, the overheads of WebPol are rather small. The overheads of executing policy handlers, even relative only to **Base**'s JavaScript execution time, are 4.0% and 13.2%, while the overheads of parsing and loading policies are no more than 4.0%. Even the total overhead of **IFC** and **WebPol** does not adversely affect the user experience in any significant way.

8.2 Implementation and Evaluation of LIR

To implement the LIR semantics described in Chapter 6, the security label attached to the JavaScript objects and the DOM nodes was modified to carry a provenance label, representing the provenance set. The provenance label is basically a bitvector where each bit represents a distinct JavaScript object or DOM node. Each bit in the provenance label is mapped to a budget, a budget label and an actual label of the object. The `setLabel()` API is extended to include the budget, and the budget label for a JavaScript object or a DOM node. If the budget is not specified, it is assumed to be 0. Similarly, if a budget label is not specified, it is assumed to be $\bot$. The checks are performed as per the semantics at every comparison operation assuming an implicit declassification at each comparison operation.

The performance overhead added as part of the LIR instrumentation in the IFC-enabled browser is evaluated and compared against the uninstrumented browser and the existing IFC instrumentation without LIR enforcement. The performance evaluation was done for the standard SunSpider 1.0.2 JavaScript benchmark suite on the uninstrumented browser, on the IFC-instrumented browser and on the LIR instrumented browser. The average overhead for LIR instrumentation over the original uninstrumented browser is 170% and adds only 17% to the overhead of the existing IFC-instrumented browser.

Chapter 9

Related Work

Browser security is a very widely-studied topic. Here, only closely related work on information flow control for browsers, browser security policies and policy enforcement techniques is described.

Information flow control and script isolation. With the widespread use of JavaScript, research in dynamic techniques for IFC has regained momentum. Nonetheless, static analyses are not completely futile. Guarnieri *et al.* [53] present a static abstract interpretation for tracking taints in JavaScript. However, the omnipresent `eval` construct is not supported and this approach does not take implicit flows into account. Chugh *et al.* propose a staged information flow approach for JavaScript [35]. They perform static server-side policy checks on statically available code and generate residual policy-checks that are applied to dynamically loaded code. This approach is limited to certain JavaScript constructs excluding dynamic features like dynamic field access or the `with` construct.

Austin and Flanagan [16] propose purely dynamic IFC for dynamically-typed languages like JavaScript. They use the no-sensitive-upgrade (NSU) check [107] to handle implicit flows. Their permissive-upgrade strategy [17] is more permissive than NSU but retains termination-insensitive noninterference. This work builds on the permissive-upgrade strategy. They also present faceted evaluation [18], which is the most permissive technique of the three. However, given the performance considerations, the technique is not suitable for enforcing information flow control in browsers. Just et al. [64] present dynamic IFC for JavaScript bytecode with static analysis to determine implicit flows precisely even in the presence of semi-unstructured control flow like `break` and `continue`. Again, NSU is leveraged to prevent implicit flows. This thesis builds on top of their work to enforce information flow control in browsers.

JSFlow [57, 58] is a stand-alone implementation of a JavaScript interpreter with fine-grained taint tracking. Many seminal ideas for labeling and tracking flows in JavaScript owe their lineage to JSFlow, but since JSFlow is written from scratch it has very high overheads and introduces annotations to deal with semi-structured control flow. It detects security violations due to branches that have not been executed and injects annotations to prevent these in subsequent runs. The approach presented

in this thesis relies on analyzing CFGs and does not require annotations. To improve permissiveness, their subsequent work [30] uses testing.

Kerschbaumer et al. [66] build an implementation of an information flow monitor for WebKit but do not handle all implicit flows. A black-box approach to enforcing non-interference is based on secure multi-execution (SME) [46]. Bielova *et al.* [29] and De Groef *et al.* [42] implement SME for web browsers. These systems do not attach labels to specific fields in the DOM. Instead, labels are attached to individual DOM APIs.

Chudnov and Naumann [34] present another approach to fine-grained IFC for JavaScript. They rewrite source programs to add shadow variables that hold labels and additional code that tracks taints. This approach is inherently more portable than that of JSFlow or the work presented in this thesis, both of which are tied to specific, instrumented browsers. However, it is unclear how this approach could be extended with a policy framework like WebPol that assigns state-dependent labels at runtime.

The work most closely related to WebPol is that of Vanhoef *et al.* [100] on stateful declassification policies in reactive systems, including web browsers. Their policies are similar to the ones presented here, but there are significant differences. First, their policies are attached to the browser and they are managed by the browser user rather than website developers. Second, the policies have coarse-granularity: They apply uniformly to all events of a certain type. Hence, it is impossible to specify a policy that makes keypresses in a password field secret, but makes other keypresses public. Third, the enforcement is based on secure multi-execution [46], which is, so far, not compatible with shared state like the DOM.

COWL [98] enforces mandatory access control at coarse-granularity. In COWL, third-party scripts are sandboxed. Each script gets access to either remote servers or the host's DOM, but not both. Scripts that need both must be re-factored to pass DOM elements over a message-passing API (`postMessage`). This can be both difficult and have high overhead. For scripts that do not need this factorization, COWL is more efficient than solutions based on FGTT.

Mash-IF [71] uses static analysis to enforce IFC policies. Mash-IF's model is different from WebPol's model. Mash-IF policies are attached only to DOM nodes and there is no support for adding policies to new objects or events. Also, in Mash-IF, the browser user (not the website developer) decides what declassifications are allowed. Mash-IF is limited to a JavaScript subset that excludes commonly used features such as `eval` and dynamic property access.

JSand [9] uses server-side changes to the host page to introduce wrappers around sensitive objects, in the style of object capabilities [78]. These wrappers mediate every access by third-party scripts and can enforce rich access policies. Through secure multi-execution, coarse-grained information flow policies are also supported. However, as mentioned earlier, it is unclear how secure multi-execution can be used with scripts that share state with the host page.

WebPol policies are enforced using an underlying IFC component. Although, in principle, any IFC technique such as fine-grained taint tracking [25, 58, 63], coarse-grained taint tracking [98] or secure multi-execution [46] can be used with WebPol, to leverage the full expressiveness of WebPol's finely-granular policies, a fine-grained IFC technique is needed.

Access control. The traditional browser security model is based on restricting scripts' access to data, not on tracking how scripts use data. In the traditional model, it is impossible to allow scripts access to data they need for legitimate purposes and, simultaneously, to prevent them from leaking the data on the side, which is the goal of IFC and WebPol. More broadly, no mechanism based only on access control can solve this problem. Nonetheless, some closely related work on access control in web browsers is discussed below.

All browsers today implement the same-origin policy (SOP) [22], which prevents a page and scripts included in it from making requests to domains other than the page's host. However, for pragmatic reasons, image requests are exempt, which is sufficient to leak information. Consequently, the SOP is not effective against malicious or buggy scripts. Cross-origin resource sharing (CORS) [3] relaxes the SOP further to allow some cross-origin requests. Content security policies (CSPs) [4] allow white- and black-listing scripts. Unlike WebPol, CSPs offer no protection against scripts that have been included by the developer without realizing that they leak information.

Conscript [77] allows the specification of fine-grained access policies on individual scripts, limiting what actions every script can perform. Similarly, AdJail [72] limits the execution of third-party scripts to a shadow page and restricts communication between the script and the host page.

Zhou and Evans [108] take a dual approach, where fine-grained access control rules are attached to DOM elements. The rules specify which scripts can and cannot access individual elements. Along similar lines, Dong *et al.* [48] present a technique to isolate sensitive data using authenticated encryption. Their goal is to reduce the size of the trusted computing base.

ADsafe [41] and FBJS [6] restrict third-party code to subsets of JavaScript, and use static analysis to check for illegitimate access. Caja [2] uses object capabilities to mediate all access by third-party scripts. Webjail [99] supports least privilege integration of third-party scripts by restricting script access based on high-level policies specified by the developer.

All these techniques enforce only access policies and cannot control what a script does with data it has access to.

Part VI

Conclusion and outlook

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 $\langle \sigma_1, e_1 \rangle \Downarrow \mathtt{n}_1'^{k_1'}$, $\langle \sigma_2, e_1 \rangle \Downarrow \mathtt{n}_2'^{k_2'}$, $\sigma_1 \sim \sigma_2$, then $\mathtt{n}_1'^{k_1'} \sim \mathtt{n}_2'^{k_2'}$.
   IH2: If $\langle \sigma_1, e_2 \rangle \Downarrow \mathtt{n}_1''^{k_1''}$, $\langle \sigma_2, e_2 \rangle \Downarrow \mathtt{n}_2''^{k_2''}$, $\sigma_1 \sim \sigma_2$, then $\mathtt{n}_1''^{k_1''} \sim \mathtt{n}_2''^{k_2''}$.
   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, $\mathtt{n}_1'^{k_1'} \sim \mathtt{n}_2'^{k_2'}$ and $\mathtt{n}_1''^{k_1''} \sim \mathtt{n}_2''^{k_2''}$.
   Proof by case analysis on low-equivalence definition (Definition 3) for $\mathtt{n}_1'^{k_1'} \sim \mathtt{n}_2'^{k_2'}$ followed by case analysis on low-equivalence definition for $\mathtt{n}_1''^{k_1''} \sim \mathtt{n}_2''^{k_2''}$.
   - $\mathtt{n}_1' = \mathtt{n}_2'$ 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 Figure 3.4, 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. ℓ represents the labels that are above the level of the attacker.

Table A.1 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 defintion 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 $\mathtt{n}_1^{k_1} \sim_\ell \mathtt{n}_2^{k_2}$.

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 $\langle \sigma_1, e_1 \rangle \Downarrow \mathtt{n}_1'^{k_1'}$, $\langle \sigma_2, e_1 \rangle \Downarrow \mathtt{n}_2'^{k_2'}$, $\sigma_1 \sim_\ell \sigma_2$, then $\mathtt{n}_1'^{k_1'} \sim_\ell \mathtt{n}_2'^{k_2'}$.
   IH2: If $\langle \sigma_1, e_2 \rangle \Downarrow \mathtt{n}_1''^{k_1''}$, $\langle \sigma_2, e_2 \rangle \Downarrow \mathtt{n}_2''^{k_2''}$, $\sigma_1 \sim_\ell \sigma_2$, then $\mathtt{n}_1''^{k_1''} \sim_\ell \mathtt{n}_2''^{k_2''}$.
   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 $k_1 = k_1' \sqcup k_1''$, $k_2 = k_2' \sqcup k_2''$.

	ℓ, ℓ	$\ell_1{}^\star, \ell_2$	$\ell_1, \ell_2{}^\star$	$\ell_1{}^\star, \ell_2{}^\star$	ℓ_1, ℓ_2	$\ell_1{}^\star, \ell_2$	$\ell_1, \ell_2{}^\star$

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*`

ℓ_1, ℓ_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*`

$\ell_1{}^\star, \ell_2$	`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*`

$\ell_1, \ell_2{}^\star$	`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*`
							-

Table A.1: Examples for all possible transitions of low-equivalent to low-equivalent values

As $\sigma_1 \sim_\ell \sigma_2$, from IH1 and IH2, $\mathtt{n}_1'^{k_1'} \sim_\ell \mathtt{n}_2'^{k_2'}$ and $\mathtt{n}_1''^{k_1''} \sim_\ell \mathtt{n}_2''^{k_2''}$.
Proof by case analysis on low-equivalence definition for $\mathtt{n}_1'^{k_1'} \sim_\ell \mathtt{n}_2'^{k_2'}$ followed by case analysis on low-equivalence definition for $\mathtt{n}_1''^{k_1''} \sim_\ell \mathtt{n}_2''^{k_2''}$.

□

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

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 $\mathtt{x}$ in statement c, $\Gamma(\sigma'(\mathtt{x})) = ((pc \sqcup m) \sqcap \ell)^\star = \ell'$. Thus, $\ell' \sqsubseteq \ell$.
   For any other y, $\sigma(y) = \sigma'(y)$. Thus, $\ell' = \ell$.
4. SEQ : IH1 : $\forall \mathtt{x}$.If $\langle \sigma, c\rangle \Downarrow_{pc} \sigma''$, $\Gamma(\sigma(\mathtt{x})) = \ell^\star$ and $pc \not\sqsubseteq \ell$, then $\Gamma(\sigma''(\mathtt{x})) = \ell''^\star \wedge \ell'' \sqsubseteq \ell$
   IH2 : $\forall \mathtt{x}$.If $\langle \sigma'', c\rangle \Downarrow_{pc} \sigma'$, $\Gamma(\sigma''(\mathtt{x})) = \ell''^\star$ and $pc \not\sqsubseteq \ell''$, then $\Gamma(\sigma'(\mathtt{x})) = \ell'^\star \wedge \ell' \sqsubseteq \ell''$
   Thus, from IH1 and IH2, $\Gamma(\sigma'(\mathtt{x})) = \ell'^\star \wedge \ell' \sqsubseteq \ell$.
5. IF-ELSE: Let $k = \ell''$.
   IH: $\forall \mathtt{x}$.If $\langle \sigma, c\rangle \Downarrow_{pc\sqcup\ell''} \sigma'$, $\Gamma(\sigma(\mathtt{x})) = \ell^\star$ and $pc \sqcup \ell'' \not\sqsubseteq \ell$, then $\Gamma(\sigma'(\mathtt{x})) = \ell'^\star \wedge \ell' \sqsubseteq \ell$
   As $pc \not\sqsubseteq \ell$, so $pc \sqcup \ell'' \not\sqsubseteq \ell$.
   Thus from IH, $\Gamma(\sigma'(\mathtt{x})) = \ell'^\star \wedge \ell' \sqsubseteq \ell$
6. WHILE-T: Let $k = \ell_e$.
   IH1: $\forall \mathtt{x}$.If $\langle \sigma, c\rangle \Downarrow_{pc\sqcup\ell_e} \sigma''$, $\Gamma(\sigma(\mathtt{x})) = \ell^\star$ and $pc \sqcup \ell_e \not\sqsubseteq \ell$, then $\Gamma(\sigma''(\mathtt{x})) = \ell''^\star \wedge \ell'' \sqsubseteq \ell$
   IH2: $\forall \mathtt{x}$.If $\langle \sigma'', c\rangle \Downarrow_{pc\sqcup\ell_e} \sigma'$, $\Gamma(\sigma''(\mathtt{x})) = \ell''^\star$ and $pc \sqcup \ell_e \not\sqsubseteq \ell$, then $\Gamma(\sigma'(\mathtt{x})) = \ell'^\star \wedge \ell' \sqsubseteq \ell$
   As $pc \not\sqsubseteq \ell$, so $pc \sqcup \ell_e \not\sqsubseteq \ell$.
   Thus from IH1 and IH2, $\Gamma(\sigma'(\mathtt{x})) = \ell'^\star \wedge \ell' \sqsubseteq \ell$
7. WHILE-F : $\sigma = \sigma'$.

□

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

Proof. Immediate from Lemma 5. □

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

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

- SKIP: $\sigma(\mathtt{x}) = \sigma'(\mathtt{x})$.

- ASSN-N: For $\mathtt{x}$ in the statement c, by premises, $\ell = pc \sqcup \ell_e$. Thus, $pc \sqsubseteq \ell$.
  For any other $\mathtt{y}$ s.t. $\Gamma(\sigma'(\mathtt{y})) = \ell'$, $\sigma(\mathtt{y}) = \sigma'(\mathtt{y})$. For ASSN-S, case does not apply.

- SEQ: IH1: If $\langle \sigma, c_1 \rangle \Downarrow_{pc} \sigma''$, then $\forall \mathtt{x}.\Gamma(\sigma''(\mathtt{x})) = \ell'' \implies (\sigma(\mathtt{x}) = \sigma''(\mathtt{x})) \vee pc \sqsubseteq \ell''$.
  IH2: If $\langle \sigma'', c_2 \rangle \Downarrow_{pc} \sigma'$, then $\forall \mathtt{x}.\Gamma(\sigma'(\mathtt{x})) = \ell \implies (\sigma''(\mathtt{x}) = \sigma'(\mathtt{x})) \vee pc \sqsubseteq \ell$.
  From IH2, if $\sigma''(\mathtt{x}) \neq \sigma'(\mathtt{x})$, then $pc \sqsubseteq \ell$.
  If $\sigma''(\mathtt{x}) = \sigma'(\mathtt{x})$, then from IH1:
  - If $\sigma(\mathtt{x}) = \sigma''(\mathtt{x})$: $\sigma(\mathtt{x}) = \sigma'(\mathtt{x})$.
  - If $\sigma(\mathtt{x}) \neq \sigma''(\mathtt{x})$: $pc \sqsubseteq \ell''$, where $\ell'' = \Gamma(\sigma''(\mathtt{x}))$. As $\sigma''(\mathtt{x}) = \sigma'(\mathtt{x})$, $\ell'' = \Gamma(\sigma'(\mathtt{x})) = \ell$. Thus, $pc \sqsubseteq \ell$.

- IF-ELSE: IH: If $\langle \sigma, c \rangle \Downarrow_{pc \sqcup \ell_e} \sigma'$, then $\forall \mathtt{x}.\Gamma(\sigma'(\mathtt{x})) = \ell \implies (\sigma(\mathtt{x}) = \sigma'(\mathtt{x})) \vee pc \sqcup \ell_e \sqsubseteq \ell$.
  From IH, 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-T: IH1: If $\langle \sigma, c \rangle \Downarrow_{pc \sqcup \ell_e} \sigma''$, then $\forall \mathtt{x}.\Gamma(\sigma''(\mathtt{x})) = \ell'' \implies (\sigma(\mathtt{x}) = \sigma''(\mathtt{x})) \vee pc \sqsubseteq \ell''$.
  IH2: If $\langle \sigma'', c \rangle \Downarrow_{pc \sqcup \ell_e} \sigma'$, then $\forall \mathtt{x}.\Gamma(\sigma'(\mathtt{x})) = \ell \implies (\sigma''(\mathtt{x}) = \sigma'(\mathtt{x})) \vee pc \sqsubseteq \ell$.
  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 $\mathtt{x} \in dom(\sigma)$, respective $\mathtt{x}'' \in dom(\sigma'')$ and respective $\mathtt{x}' \in dom(\sigma')$, $\mathtt{x} \sim_\ell \mathtt{x}''$ 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, $\mathtt{x} \sim_\ell \mathtt{x}'$.

- $k_1 = \ell_1 \wedge k_2 = {\ell_2}^\star$ *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, $\mathtt{x} \sim_\ell \mathtt{x}'$.

  (b) $k_2 = {\ell_2}^\star \wedge k_3 = \ell_3$ *s.t.* $(\ell_3 \not\sqsubseteq \ell)$ (case 4): By definition 7.2, $\mathtt{x} \sim_\ell \mathtt{x}'$.

  (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, $\mathtt{x} \sim_\ell \mathtt{x}'$.

- $k_1 = \ell_1 \wedge k_2 = {\ell_2}^\star$ *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, $\mathtt{x} \sim_\ell \mathtt{x}'$.

  (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 : $\sigma = \sigma'$

□

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 $\sigma_1 \sim_\ell \sigma_2$, $\forall \mathtt{x}.\sigma_1(\mathtt{x}) \sim_\ell \sigma_2(\mathtt{x})$. Let $\sigma_1(\mathtt{x}) = \mathtt{v}_1^{k_1}$, $\sigma_2(\mathtt{x}) = \mathtt{v}_2^{k_2}$ and $\sigma_1'(\mathtt{x}) = \mathtt{v}_1'^{k_1'}$, $\sigma_2'(\mathtt{x}) = \mathtt{v}_2'^{k_2'}$
   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, $\mathtt{x}$, 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, $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
    - 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 7.2 $k_2' = \ell_2' \not\sqsubseteq \ell$. By definition 7.4, $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
    - If $k_1' = \ell_1' \wedge k_2' = \ell_2'^\star$, 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, $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$. If $k_1' = \ell_1'^\star \ \wedge \ k_2' = \ell_2'^\star$, then by definition 7.3, $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
  (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, $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
    - 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 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, $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
    - 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 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, $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.

(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, $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
  - 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 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, $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
  - 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 7.2, $\ell_2' \not\sqsubseteq \ell$. 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 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, $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
- $\ell_1 \sqsubseteq \ell_2 \sqsubseteq \ell$:
  - If $k_1' = \ell_1'{}^\star \ \wedge \ k_2' = \ell_2'{}^\star$, by definition 7.3, $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
  - 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 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, $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
  - If $k_1' = \ell_1'{}^\star \ \wedge \ k_2' = \ell_2'$, then as $\sigma_1 \sim_\ell \sigma_1'$ and $\sigma_2 \sim_\ell \sigma_2'$, $\ell_1' \sqsubseteq (pc \sqcup \ell_1^e) \sqcap \ell_1$ 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, $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
  - 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, $pc \sqcup \ell_1^e \sqsubseteq \ell_1$. 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, $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
  - 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 7.5, $\sigma_1'(\mathtt{x}) \sim_\ell \sigma_2'(\mathtt{x})$.
  - 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 7.2, $\ell_1' \not\sqsubseteq \ell$. 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: $\sigma_1' = \sigma_1 \sim_\ell \sigma_2 = \sigma_2'$

□

B Proofs of Precision for Dynamic IFC with Unstructured Control Flow and Exceptions

Theorem 3 (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' 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' 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 prediate 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 $n = IPD(b) = i$. □

Theorem 4. *The actual IPD of a node having* `SEN` *as its IPD is the node on the top of the pc-stack, which lies in a previously called function.*

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 $b < b' < (i = IPD(b))$, $(i \in \mathcal{N}) \neq$ `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. $\nexists n \in b' \rightarrow_p i \mid (n \text{ pd } b') \wedge (b' < n < i)$.
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 $IPD(G()) = (n'' \in \mathcal{N}'') \leq n$ 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. □

C Proofs for IFC with Unstructured Control Flow

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 induction on the derivation rules and case analysis on the last rule:

- 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* $\forall (0 \leq i \leq n).\Gamma(!\rho_i) \not\sqsubseteq \ell$*, 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 $\forall \mathtt{x}.\Gamma(\sigma(\mathtt{x})) = \ell^\star \wedge (\Gamma(!\rho) \not\sqsubseteq \ell) \implies \Gamma(\sigma'(\mathtt{x})) = \ell'^\star \wedge \ell' \sqsubseteq \ell$

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

- 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* $\Gamma(\sigma(\mathtt{x})) = \ell^\star$ *and* $\Gamma(\sigma'(\mathtt{x})) = \ell'$*, 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* $\forall (0 \leq i \leq n).\Gamma(!\rho_i) \not\sqsubseteq \ell$*, 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, $\forall \mathtt{x}.(\sigma_0(\mathtt{x}) \sim_\ell \sigma_{n-1}(\mathtt{x})$. From Lemma 8, $\sigma_{n-1} \sim_\ell \sigma_n$. Thus, $\forall \mathtt{x}.(\sigma_{n-1}(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$
Assume that $\sigma_0(\mathtt{x}) = \mathtt{v}_0^k$, $\sigma_{n-1}(\mathtt{x}) = \mathtt{v}_{n-1}^{k'}$, and $\sigma_n(\mathtt{x}) = \mathtt{v}_n^{k''}$ 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, $\sigma_0(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$.
  2. $k' = \ell'$ and $k'' = \ell''^\star \wedge \ell'' \sqsubseteq \ell' \sqsubseteq \ell$: By definition 7.5. Thus, $\sigma_0(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$.
- $k = \ell_1 \not\sqsubseteq \ell \wedge k' = \ell_2 \not\sqsubseteq \ell$:
  1. $k' = \ell_2 \not\sqsubseteq \ell \wedge k'' = \ell_3 \not\sqsubseteq \ell$. By definition 7.2, $\sigma_0(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$.
  2. $k' = \ell_2 \not\sqsubseteq \ell \wedge k'' = \ell_3{}^\star$: $\ell_1 \not\sqsubseteq \ell$. Thus, by definition 7.5, $\sigma_0(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$.
- $k = \ell_1{}^\star \wedge k' = \ell_2{}^\star$:
  1. $k' = \ell_2{}^\star \wedge k'' = \ell_3{}^\star$: By definition 7.3, $\sigma_0(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$.
  2. $k' = \ell_2{}^\star \wedge k'' = \ell_3 \wedge (\ell_3 \not\sqsubseteq \ell)$: By definition 7.4, $\sigma_0(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$.
  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, $\sigma_0(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$.
- $k = \ell_1{}^\star \wedge k' = \ell_2 \; s.t. \; (\ell_2 \not\sqsubseteq \ell)$: Either
  - $k' = \ell_2 \not\sqsubseteq \ell \wedge k'' = \ell_3 \not\sqsubseteq \ell$: By definition 7.4, $\sigma_0(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$
  - $k' = \ell_2 \not\sqsubseteq \ell \wedge k'' = \ell_3{}^\star$: By definition 7.3, $\sigma_0(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$
- $k = \ell_1{}^\star \wedge k' = \ell_2 \; s.t. \; (\ell_1 \sqsubseteq \ell_2)$:
  - $k' = k'' = \ell_2$: By definition 7.4, $\sigma_0(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$
  - $k' = \ell_2 \not\sqsubseteq \ell \wedge k'' = \ell_3 \not\sqsubseteq \ell$: By definition 7.4, $\sigma_0(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$
  - $k' = \ell_2 \not\sqsubseteq \ell \wedge k'' = \ell_3{}^\star$: By definition 7.3, $\sigma_0(\mathtt{x}) \sim_\ell \sigma_n(\mathtt{x})$
- $k = \ell_1 \wedge k' = \ell_2{}^\star \; s.t. \; (\ell_1 \not\sqsubseteq \ell)$:
  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 $\mathfrak{n}^m$ 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 $\mathfrak{b}^{\ell_i}$ 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 $\mathcal{P} = s_1 \to s_2 \to \ldots \to s_n$ 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 5 (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* $n \leq m$,
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. $\langle \sigma_1, \iota_1, \rho_1 \rangle \rightarrow^* \langle \sigma'_1, \textit{end}, [] \rangle$
3. $\langle \sigma_2, \iota_2, \rho_2 \rangle \rightarrow^* \langle \sigma'_2, \textit{end}, [] \rangle$

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 5, $n = m = k$. 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, $\sigma'_1 \sim_\ell \sigma'_2$. □

D Proofs for Limited Information Release

Lemma 14 (Trace-projection). $(\tau_1 :: \tau_2)\uparrow_\ell = \ \tau_1\uparrow_\ell :: \tau_2\uparrow_\ell$

Proof. By induction on τ_1.
Base case: If $\tau_1 = []$, then $([] :: \tau_2)\uparrow_\ell = \tau_2\uparrow_\ell = [] :: \tau_2\uparrow_\ell = []\uparrow_\ell :: \tau_2\uparrow_\ell$.
Inductive case: IH: $(\tau_1 :: \tau_2)\uparrow_\ell = \ \tau_1\uparrow_\ell :: \tau_2\uparrow_\ell$
TS: $((\mathtt{n}^m :: \tau_1) :: \tau_2)\uparrow_\ell = \ (\mathtt{n}^m :: \tau_1)\uparrow_\ell :: \tau_2\uparrow_\ell$

$$(\mathtt{n}^m :: \tau_1)\uparrow_\ell = \begin{cases} \mathtt{n}^m :: \tau_1\uparrow_\ell & \textit{if } m \sqsubseteq \ell, \\ \tau_1\uparrow_\ell & \textit{else.} \end{cases} \tag{1}$$

Also by associativity of list concatentation, $((\mathtt{n}^m :: \tau_1) :: \tau_2) = (\mathtt{n}^m :: (\tau_1 :: \tau_2))$. Again,

$$(\mathtt{n}^m :: (\tau_1 :: \tau_2))\uparrow_\ell = \begin{cases} \mathtt{n}^m :: (\tau_1 :: \tau_2)\uparrow_\ell & \textit{if } m \sqsubseteq \ell, \\ (\tau_1 :: \tau_2)\uparrow_\ell & \textit{else.} \end{cases} \tag{2}$$

If $m \sqsubseteq \ell$: from (2), LHS = $\mathtt{n}^m :: (\tau_1 :: \tau_2)\uparrow_\ell$ and from (1), RHS = $\mathtt{n}^m :: \tau_1\uparrow_\ell :: \tau_2\uparrow_\ell$. From IH, LHS = RHS. Else: from (2), LHS = $(\tau_1 :: \tau_2)\uparrow_\ell$ and from (1), RHS = $\tau_1\uparrow_\ell :: \tau_2\uparrow_\ell$. From IH, LHS = RHS. □

Lemma 15 (Confinement of expressions). *If* $\langle \sigma, \iota, e\rangle \underset{pc}{\Downarrow} \mathtt{n}^{(m,\delta)}, \iota', \tau$, *and* $\Gamma(pc) \not\sqsubseteq \ell$, *then* $\iota \sim_\ell \iota'$ *and* $\tau\uparrow_\ell = \epsilon$

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

- CONST and VAR: $\iota = \iota'$ and $\tau = \epsilon$
- AOP and COP:
  IH1: If $\langle \sigma, \iota, e_1\rangle \underset{pc}{\Downarrow} \mathtt{n}_1^{(m_1,\delta_1)}, \iota_1, \tau_1$, and $pc \not\sqsubseteq \ell$, then $\iota \sim_\ell \iota_1$ and $\tau_1\uparrow_\ell = \epsilon$
  IH2: If $\langle \sigma, \iota_1, e_2\rangle \underset{pc}{\Downarrow} \mathtt{n}_2^{(m_2,\delta_2)}, \iota', \tau_2$, and $pc \not\sqsubseteq \ell$, then $\iota_1 \sim_\ell \iota'$ and $\tau_2\uparrow_\ell = \epsilon$
  As $pc \not\sqsubseteq \ell$, $\iota \sim_\ell \iota_1$ and $\tau_1\uparrow_\ell = \epsilon$ and $\iota_1 \sim_\ell \iota'$ and $\tau_2\uparrow_\ell = \epsilon$. Thus, $(\tau_1\uparrow_\ell :: \tau_2\uparrow_\ell) = (\tau_1 :: \tau_2)\uparrow_\ell = \epsilon$.
  As $\iota \sim_\ell \iota_1$, $\forall \mathtt{x} \in \iota.\mathbb{B}(\mathtt{x}) \sqsubseteq \ell \implies \iota(\mathtt{x}) = \iota_1(\mathtt{x})$ and $\iota_1 \sim_\ell \iota'$, $\forall \mathtt{x} \in \iota_1.\mathbb{B}(\mathtt{x}) \sqsubseteq \ell \implies \iota_1(\mathtt{x}) = \iota'(\mathtt{x})$. Thus, $\forall \mathtt{x} \in \iota.\mathbb{B}(\mathtt{x}) \sqsubseteq \ell \implies \iota(\mathtt{x}) = \iota'(\mathtt{x})$.
- DCOPR: From above reasoning, $\tau_1\uparrow_\ell :: \tau_2\uparrow_\ell = (\tau_1 :: \tau_2)\uparrow_\ell = \epsilon$ and $\forall \mathtt{x} \in \iota.\mathbb{B}(\mathtt{x}) \sqsubseteq \ell \implies \iota(\mathtt{x}) = \iota_2(\mathtt{x})$. As $pc \sqsubseteq k_o$ and $pc \not\sqsubseteq \ell$, $k_o \not\sqsubseteq \ell$. ι_2 changes for only those x that have a budget label $\mathbb{B}(\mathtt{x}) \sqsupseteq k_o$. Thus, $\forall \mathtt{x} \in \iota.\mathbb{B}(\mathtt{x}) \sqsubseteq \ell \implies \iota(\mathtt{x}) = \iota'(\mathtt{x})$. As $l \not\sqsubseteq \ell$, $\mathtt{n}^l\uparrow_\ell = \epsilon$. Thus, $\tau\uparrow_\ell = (\tau_1 :: \tau_2 :: \mathtt{n}^l)\uparrow_\ell = \epsilon$
- DCOPN: From above reasoning, $\tau_1\uparrow_\ell :: \tau_2\uparrow_\ell = (\tau_1 :: \tau_2)\uparrow_\ell = \epsilon$ and $\forall \mathtt{x} \in \iota.\mathbb{B}(\mathtt{x}) \sqsubseteq \ell \implies \iota(\mathtt{x}) = \iota_2(\mathtt{x})$. $\tau\uparrow_\ell = \epsilon$.

□

Lemma 16 (Confinement). *If* $\langle \sigma, \iota, c\rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau\rangle$, *and* $\Gamma(pc) \not\sqsubseteq \ell$, *then* $\sigma \sim_\ell \sigma'$, $\iota \sim_\ell \iota'$ *and* $\tau\uparrow_\ell = \epsilon$

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

- SKIP: $\sigma = \sigma'$, $\iota = \iota'$ and $\tau = \epsilon$
- ASSN: As $pc \not\sqsubseteq \ell$ and $pc \sqsubseteq \Gamma_f(\sigma(\mathtt{x}))$, $\Gamma_f(\sigma(\mathtt{x})) \not\sqsubseteq \ell$. Also, $(\Gamma_f(\sigma'(\mathtt{x})) = pc \sqcup m) \not\sqsubseteq \ell$ and all other $\sigma(\mathtt{x})$ remains unchanged. Thus, $\sigma \sim_\ell \sigma'$. From Lemma 15, $\iota \sim_\ell \iota'$ and $\tau{\uparrow_\ell} = \epsilon$.
- SEQ: IH1: If $\langle \sigma, \iota, c_1 \rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau_1 \rangle$, and $pc \not\sqsubseteq \ell$, then $\sigma \sim_\ell \sigma'$, $\iota \sim_\ell \iota'$ and $\tau_1{\uparrow_\ell} = \epsilon$
  IH2: If $\langle \sigma', \iota', c_2 \rangle \underset{pc}{\Downarrow} \langle \sigma'', \iota'', \tau_2 \rangle$, and $pc \not\sqsubseteq \ell$, then $\sigma' \sim_\ell \sigma''$, $\iota' \sim_\ell \iota''$ and $\tau_2{\uparrow_\ell} = \epsilon$
  As $\sim_\ell$ is an equivalence relation, from IH1 and IH2, $\sigma \sim_\ell \sigma''$, $\iota \sim_\ell \iota''$. From Lemma 14, $(\tau_1 :: \tau_2){\uparrow_\ell} = \epsilon$.
- IF-ELSE: IH: If $\langle \sigma, \iota', c_i \rangle \underset{pc \sqcup k}{\Downarrow} \langle \sigma', \iota'', \tau_2 \rangle$, and $pc \sqcup (k, \delta) \not\sqsubseteq \ell$, then $\sigma \sim_\ell \sigma'$, $\iota' \sim_\ell \iota''$ and $\tau_2{\uparrow_\ell} = \epsilon$
  From Lemma 15, $\iota \sim_\ell \iota'$ and $\tau_1{\uparrow_\ell} = \epsilon$. Also, as $pc \not\sqsubseteq \ell$, $pc \sqcup l \not\sqsubseteq \ell$ and from IH, $\sigma \sim_\ell \sigma'$, $\iota' \sim_\ell \iota''$ and $\tau_2{\uparrow_\ell} = \epsilon$. Thus, $\iota \sim_\ell \iota''$ and $(\tau_1 :: \tau_2){\uparrow_\ell} = \epsilon$
- WHILE-T: Similar to IF-ELSE and SEQ
- WHILE-F: $\sigma = \sigma'$. From Lemma 15, $\iota \sim_\ell \iota'$ and $\tau{\uparrow_\ell} = \epsilon$

□

Lemma 17. *If*
$\langle \sigma, \iota, e \rangle \underset{pc}{\Downarrow} \mathtt{n}^{(l,\delta)}, \iota', \tau$ *and* $\exists \mathtt{x} \in \sigma.\Big(\Gamma(\sigma(\mathtt{x})) \not\sqsubseteq \ell \wedge \big(\forall \mathtt{y} \in \sigma.\mathtt{y} \neq \mathtt{x} \wedge \Gamma(\sigma(\mathtt{y})) \sqsubseteq \ell\big)\Big)$,
then $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell}(\mathtt{x}) - \iota'{\uparrow_\ell}(\mathtt{x})$.

Proof. Induction on the derivation for expressions and case analysis on the last rule.

- CONST: $\iota = \iota'$ and $|\tau{\uparrow_\ell}| = 0$.
- VAR: $\iota = \iota'$ and $|\tau{\uparrow_\ell}| = 0$.
- AOP and COP: IH1: If $\langle \sigma, \iota, e_1 \rangle \underset{pc}{\Downarrow} \mathtt{n}_1^{(k_1,\delta_1)}, \iota_1, \tau_1$, and
  $\exists \mathtt{x} \in \sigma.\Big(\Gamma(\sigma(\mathtt{x})) \not\sqsubseteq \ell \wedge \big(\forall \mathtt{y} \in \sigma.\mathtt{y} \neq \mathtt{x} \wedge \Gamma(\sigma(\mathtt{y})) \sqsubseteq \ell\big)\Big)$ then $|\tau_1{\uparrow_\ell}| \leq \iota{\uparrow_\ell}(\mathtt{x}) - \iota_1{\uparrow_\ell}(\mathtt{x})$.
  IH2: If $\langle \sigma, \iota_1, e_2 \rangle \underset{pc}{\Downarrow} \mathtt{n}_2^{(k_2,\delta_2)}, \iota', \tau_2$, and
  $\exists \mathtt{x} \in \sigma.\Big(\Gamma(\sigma(\mathtt{x})) \not\sqsubseteq \ell \wedge \big(\forall \mathtt{y} \in \sigma.\mathtt{y} \neq \mathtt{x} \wedge \Gamma(\sigma(\mathtt{y})) \sqsubseteq \ell\big)\Big)$ then $|\tau_2{\uparrow_\ell}| \leq \iota_1{\uparrow_\ell}(\mathtt{x}) - \iota'{\uparrow_\ell}(\mathtt{x})$.
  From IH1 and IH2, $|\tau_1{\uparrow_\ell}| + |\tau_2{\uparrow_\ell}| = \iota{\uparrow_\ell}(\mathtt{x}) - \iota'{\uparrow_\ell}(\mathtt{x})$. Thus, $|(\tau_1 :: \tau_2){\uparrow_\ell}| \leq \iota{\uparrow_\ell}(\mathtt{x}) - \iota'{\uparrow_\ell}(\mathtt{x})$.
- DCOPR: Similar to COP, $|(\tau_1 :: \tau_2){\uparrow_\ell}| \leq \iota{\uparrow_\ell}(\mathtt{x}) - \iota_2{\uparrow_\ell}(\mathtt{x})$. Either:
  - $k_o \sqsubseteq \ell \wedge \mathtt{x} \in \delta_o.\mathbb{L}(\mathtt{x}) \not\sqsubseteq \ell$: If $x \in \delta.(\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell) \implies l \not\sqsubseteq \ell$. Thus, $|\tau{\uparrow_\ell}| = |\tau_1{\uparrow_\ell}| + |\tau_2{\uparrow_\ell}|$. As $\iota_2 = \iota'$, $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell}(\mathtt{x}) - \iota'{\uparrow_\ell}(\mathtt{x})$.
    If $x \in \delta.(\mathbb{B}(\mathtt{x}) \sqsubseteq \ell) \implies l \sqsubseteq \ell$ and $\iota'(\mathtt{x}) = \iota_2(\mathtt{x}) - 1$. $\tau = \tau_1 :: \tau_2 :: \mathtt{n}^l$. $|\tau{\uparrow_\ell}| = |\tau_1{\uparrow_\ell}| + |\tau_2{\uparrow_\ell}| + 1$. Thus, $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell}(\mathtt{x}) - \iota'{\uparrow_\ell}(\mathtt{x})$.
    If $\delta = \emptyset$ then $\tau{\uparrow_\ell} = \tau_1{\uparrow_\ell} :: \tau_2{\uparrow_\ell}$ and $\iota' = \iota_2$. Thus, $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell}(\mathtt{x}) - \iota'{\uparrow_\ell}(\mathtt{x})$
  - $k_o \not\sqsubseteq \ell$: $l \not\sqsubseteq \ell$. Thus, $|\tau{\uparrow_\ell}| = |\tau_1{\uparrow_\ell}| + |\tau_2{\uparrow_\ell}|$. As $\iota_2 = \iota'$, $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell}(\mathtt{x}) - \iota'{\uparrow_\ell}(\mathtt{x})$.

- DCOPN: Similar to COP, $|(\tau_1 :: \tau_2)\uparrow_\ell| \leq \iota\uparrow_\ell - \iota_2\uparrow_\ell$. Thus, $|\tau\uparrow_\ell| \leq \iota\uparrow_\ell(\mathtt{x}) - \iota'\uparrow_\ell(\mathtt{x})$

□

Theorem 6 (LIR for a single secret). *If* $\langle \sigma, \iota, c\rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau\rangle$, *and*
$\exists \mathtt{x} \in \sigma.\Big(\Gamma(\sigma(\mathtt{x})) \not\sqsubseteq \ell \wedge \big(\forall \mathtt{y} \in \sigma.\mathtt{y} \neq \mathtt{x} \wedge \Gamma(\sigma(\mathtt{y})) \sqsubseteq \ell\big)\Big)$,
then $|\tau\uparrow_\ell| \leq \iota\uparrow_\ell(\mathtt{x}) - \iota'\uparrow_\ell(\mathtt{x})$.

Proof. Induction on the derivation for commands and case analysis on the last rule.

- SKIP: $\iota = \iota'$ and $|\tau\uparrow_\ell| = 0$.
- ASSN: From Lemma 17.
- SEQ: IH1: If $\langle \sigma, \iota, c_1\rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau_1\rangle$, and
  $\exists \mathtt{x} \in \sigma.\Big(\Gamma(\sigma(\mathtt{x})) \not\sqsubseteq \ell \wedge \big(\forall \mathtt{y} \in \sigma.\mathtt{y} \neq \mathtt{x} \wedge \Gamma(\sigma(\mathtt{y})) \sqsubseteq \ell\big)\Big)$ then $|\tau_1\uparrow_\ell| \leq \iota\uparrow_\ell(\mathtt{x}) - \iota'\uparrow_\ell(\mathtt{x})$.
  IH2: If $\langle \sigma', \iota', c_2\rangle \underset{pc}{\Downarrow} \langle \sigma'', \iota'', \tau_2\rangle$, and
  $\exists \mathtt{x} \in \sigma.\Big(\Gamma(\sigma(\mathtt{x})) \not\sqsubseteq \ell \wedge \big(\forall \mathtt{y} \in \sigma.\mathtt{y} \neq \mathtt{x} \wedge \Gamma(\sigma(\mathtt{y})) \sqsubseteq \ell\big)\Big)$ then $|\tau_2\uparrow_\ell| \leq \iota'\uparrow_\ell(\mathtt{x}) - \iota''\uparrow_\ell(\mathtt{x})$.
  $x \in \sigma \implies \mathtt{x} \in \sigma_1$.
  From IH1 and IH2, $|\tau_1\uparrow_\ell| + |\tau_2\uparrow_\ell| \leq \iota\uparrow_\ell(\mathtt{x}) - \iota''\uparrow_\ell(\mathtt{x})$. Thus, $|\tau\uparrow_\ell| \leq \iota\uparrow_\ell(\mathtt{x}) - \iota''\uparrow_\ell(\mathtt{x})$.
- IF-ELSE: From Lemma 17, $|\tau_1\uparrow_\ell| \leq \iota\uparrow_\ell(\mathtt{x}) - \iota'\uparrow_\ell(\mathtt{x})$.
  IH: If $\langle \sigma, \iota', c_i\rangle \underset{pc\sqcup k}{\Downarrow} \langle \sigma', \iota'', \tau_2\rangle$, and
  $\exists \mathtt{x} \in \sigma.\Big(\Gamma(\sigma(\mathtt{x})) \not\sqsubseteq \ell \wedge \big(\forall \mathtt{y} \in \sigma.\mathtt{y} \neq \mathtt{x} \wedge \Gamma(\sigma(\mathtt{y})) \sqsubseteq \ell\big)\Big)$ then $|\tau_2\uparrow_\ell| \leq \iota'\uparrow_\ell(\mathtt{x}) - \iota''\uparrow_\ell(\mathtt{x})$.
  From IH and Lemma 17, $|\tau\uparrow_\ell| \leq \iota\uparrow_\ell(\mathtt{x}) - \iota''\uparrow_\ell(\mathtt{x})$.
- WHILE-T: Similar to IF-ELSE
- WHILE-F: From Lemma 17.

□

Lemma 18. *If* $\langle \sigma, \iota, e\rangle \underset{pc}{\Downarrow} \mathtt{n}^{(l,\delta)}, \iota', \tau$, *then* $|\tau\uparrow_\ell| \leq \iota\uparrow_\ell - \iota'\uparrow_\ell$.

Proof. Induction on the derivation for expressions and case analysis on the last rule.

- CONST: $\iota = \iota'$ and $|\tau\uparrow_\ell| = 0$.
- VAR: $\iota = \iota'$ and $|\tau\uparrow_\ell| = 0$.
- AOP and COP:
  IH1: If $\langle \sigma, \iota, e_1\rangle \underset{pc}{\Downarrow} \mathtt{n}_1^{(k_1,\delta_1)}, \iota_1, \tau_1$, then $|\tau_1\uparrow_\ell| \leq \iota\uparrow_\ell - \iota_1\uparrow_\ell$.
  IH2: If $\langle \sigma, \iota_1, e_2\rangle \underset{pc}{\Downarrow} \mathtt{n}_2^{(k_2,\delta_2)}, \iota', \tau_2$, then $|\tau_2\uparrow_\ell| \leq \iota_1\uparrow_\ell - \iota'\uparrow_\ell$.
  From IH1 and IH2, $|\tau_1\uparrow_\ell| + |\tau_2\uparrow_\ell| = \iota\uparrow_\ell - \iota'\uparrow_\ell$. Thus, $|(\tau_1 :: \tau_2)\uparrow_\ell| \leq \iota\uparrow_\ell - \iota'\uparrow_\ell$.
- DCOPR: Similar to COP, $|(\tau_1 :: \tau_2)\uparrow_\ell| \leq \iota\uparrow_\ell - \iota_2\uparrow_\ell$. Either:

- $k_o \sqsubseteq \ell \wedge \forall \mathtt{x} \in \delta_o.\mathbb{L}(\mathtt{x}) \sqsubseteq \ell$: If $\delta \neq \emptyset$ then $\exists x.\iota_2(\mathtt{x}) > 0$ and $\iota_2(\mathtt{x}) - \iota'(\mathtt{x}) = 1$. For more than one x, $\iota_2{\uparrow_\ell} - \iota'{\uparrow_\ell} \geq 1$. Also, $\tau = \tau_1 :: \tau_2 :: \mathtt{n}^l$. $|\tau{\uparrow_\ell}| = |\tau_1{\uparrow_\ell}| + |\tau_2{\uparrow_\ell}| + 1$. Thus, $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell} - \iota'{\uparrow_\ell}$
- $k_o \sqsubseteq \ell \wedge \exists \mathtt{x} \in \delta_o.\mathbb{L}(\mathtt{x}) \not\sqsubseteq \ell$: If $\exists \mathtt{x} \in \delta.(\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell) \implies l \not\sqsubseteq \ell$. Thus, $|\tau{\uparrow_\ell}| = |\tau_1{\uparrow_\ell}| + |\tau_2{\uparrow_\ell}|$. As $\iota_2 = \iota'$, $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell} - \iota'{\uparrow_\ell}$.
  If $\forall \mathtt{x} \in \delta.(\mathbb{B}(\mathtt{x}) \sqsubseteq \ell) \implies l \sqsubseteq \ell$ and $\iota'(\mathtt{x}) = \iota_2(\mathtt{x}) - 1$. $\tau = \tau_1 :: \tau_2 :: \mathtt{n}^l$. $|\tau{\uparrow_\ell}| = |\tau_1{\uparrow_\ell}| + |\tau_2{\uparrow_\ell}| + 1$. Thus, $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell} - \iota'{\uparrow_\ell}$.
  If $\delta = \emptyset$ then $\tau{\uparrow_\ell} = \tau_1{\uparrow_\ell} :: \tau_2{\uparrow_\ell}$ and $\iota' = \iota_2$. Thus, $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell} - \iota'{\uparrow_\ell}$
- $k_o \not\sqsubseteq \ell$: $l \not\sqsubseteq \ell$. Thus, $|\tau{\uparrow_\ell}| = |\tau_1{\uparrow_\ell}| + |\tau_2{\uparrow_\ell}|$. As $\iota_2 = \iota'$, $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell} - \iota'{\uparrow_\ell}$. If $\delta = \emptyset$ then $\tau{\uparrow_\ell} = \tau_1{\uparrow_\ell} :: \tau_2{\uparrow_\ell}$ and $\iota' = \iota_2$. Thus, $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell} - \iota'{\uparrow_\ell}$

- DCOPN: Similar to COP, $|(\tau_1 :: \tau_2){\uparrow_\ell}| \leq \iota{\uparrow_\ell} - \iota_2{\uparrow_\ell}$. Thus, $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell} - \iota'{\uparrow_\ell}$

□

Theorem 7 (LIR). *If* $\langle \sigma, \iota, c\rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau\rangle$, *then* $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell} - \iota'{\uparrow_\ell}$.

Proof. Induction on the derivation for commands and case analysis on the last rule.

- SKIP: $\iota = \iota'$ and $|\tau{\uparrow_\ell}| = 0$.
- ASSN: From Lemma 18.
- SEQ: IH1: If $\langle \sigma, \iota, c_1\rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau_1\rangle$, then $|\tau_1{\uparrow_\ell}| \leq \iota{\uparrow_\ell} - \iota'{\uparrow_\ell}$.
  IH2: If $\langle \sigma', \iota', c_2\rangle \underset{pc}{\Downarrow} \langle \sigma'', \iota'', \tau_2\rangle$, then $|\tau_2{\uparrow_\ell}| \leq \iota'{\uparrow_\ell} - \iota''{\uparrow_\ell}$.
  From IH1 and IH2, $|\tau_1{\uparrow_\ell}| + |\tau_2{\uparrow_\ell}| \leq \iota{\uparrow_\ell} - \iota''{\uparrow_\ell}$. Thus, $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell} - \iota''{\uparrow_\ell}$.
- IF-ELSE: From Lemma 18, $|\tau_1{\uparrow_\ell}| \leq \iota{\uparrow_\ell} - \iota'{\uparrow_\ell}$.
  IH: If $\langle \sigma, \iota', c_i\rangle \underset{pc \sqcup k}{\Downarrow} \langle \sigma', \iota'', \tau_2\rangle$, then $|\tau_2{\uparrow_\ell}| \leq \iota'{\uparrow_\ell} - \iota''{\uparrow_\ell}$.
  From IH and Lemma 18, $|\tau{\uparrow_\ell}| \leq \iota{\uparrow_\ell} - \iota''{\uparrow_\ell}$.
- WHILE-T: Similar to IF-ELSE
- WHILE-F: From Lemma 18.

□

Lemma 19. *If* $\langle \sigma, \iota, e, \tau\rangle \underset{pc}{\downarrow} \mathtt{v}, \iota', \tau'$, *then* $\langle \sigma, \iota, e, \tau :: \tau^r\rangle \underset{pc}{\downarrow} \mathtt{v}, \iota', \tau' :: \tau^r$

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

- S-CONST and S-VAR: $\tau = \tau' = \epsilon$, thus $\tau :: \tau^r = \tau' :: \tau^r$.
- S-AOP and S-COP:
  IH1: If $\langle \sigma, \iota, e_1, \tau\rangle \underset{pc}{\downarrow} \mathtt{v}, \iota_1, \tau'$, then $\langle \sigma, \iota, e_1, \tau :: \tau^r\rangle \underset{pc}{\downarrow} \mathtt{v}, \iota_1, \tau' :: \tau^r$
  IH2: If $\langle \sigma, \iota, e_2, \tau'\rangle \underset{pc}{\downarrow} \mathtt{v}, \iota_1, \tau''$, then $\langle \sigma, \iota, e, \tau' :: \tau^r\rangle \underset{pc}{\downarrow} \mathtt{v}, \iota', \tau'' :: \tau^r$
  T.S. If $\langle \sigma, \iota, e, \tau\rangle \underset{pc}{\downarrow} \mathtt{v}, \iota', \tau''$, then $\langle \sigma, \iota, e, \tau :: \tau^r\rangle \underset{pc}{\downarrow} \mathtt{v}, \iota', \tau'' :: \tau^r$. From IH1 and IH2.

- S-DCOPR: As above, if $\langle \sigma, \iota, e, \tau \rangle \underset{pc}{\downarrow} \mathtt{v}, \iota', \tau''$, then $\langle \sigma, \iota, e, \tau :: \tau^r \rangle \underset{pc}{\downarrow} \mathtt{v}, \iota', \tau'' :: \tau^r$. As $\mathtt{v}$ is the same, $\tau'' = \mathtt{n}^l :: \tau_1$ and $\tau'' :: \tau^r = \mathtt{n}^l :: \tau_1 :: \tau^r$.
  Thus, if $\langle \sigma, \iota, e, \tau \rangle \underset{pc}{\downarrow} \mathtt{v}, \iota', \tau_f$, then $\langle \sigma, \iota, e, \tau :: \tau^r \rangle \underset{pc}{\downarrow} \mathtt{v}, \iota', \tau_f :: \tau^r$
- S-DCOPN: Similar to above case

□

Lemma 20. *If* $\langle \sigma, \iota, c, \tau \rangle \underset{pc}{\downarrow} \sigma', \iota', \tau'$, *then* $\langle \sigma, \iota, c, \tau :: \tau^r \rangle \underset{pc}{\downarrow} \sigma', \iota', \tau' :: \tau^r$

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

- S-SKIP: $\tau = \tau'$, thus $\tau :: \tau^r = \tau' :: \tau^r$.
- S-ASSN: From Lemma 19.
- S-SEQ: IH1: If $\langle \sigma, \iota, c, \tau \rangle \underset{pc}{\downarrow} \sigma', \iota', \tau_1$, then $\langle \sigma, \iota, c, \tau :: \tau^r \rangle \underset{pc}{\downarrow} \sigma', \iota', \tau_1 :: \tau^r$
  IH2: If $\langle \sigma, \iota, c, \tau_1 \rangle \underset{pc}{\downarrow} \sigma', \iota', \tau'$, then $\langle \sigma, \iota, c, \tau_1 :: \tau^r \rangle \underset{pc}{\downarrow} \sigma', \iota', \tau' :: \tau^r$.
  From IH1 and IH2.
- S-IF-ELSE-N and S-WHILE-T: Similar to above case
- S-IF-ELSE-S and S-WHILE-FS: From Lemma 19.

□

Corollary 5 (Trace Reduction)**.** *If* $\langle \sigma, \iota, c, \tau \rangle \underset{pc}{\downarrow} \sigma', \iota', \epsilon$, *then* $\langle \sigma, \iota, c, \tau :: \tau^r \rangle \underset{pc}{\downarrow} \sigma', \iota', \tau^r$

Proof. From Lemma 20 □

Lemma 21 (Equivalence)**.** *If* $\sigma \sim_\ell \sigma'$ *and* $\sigma \simeq_\ell \sigma''$, *then* $\sigma' \simeq_\ell \sigma''$

Proof. Consider $\mathtt{x} = \mathtt{n}_1^{(l,\delta)}$ in σ and $\mathtt{x} = \mathtt{n}_2^{(l',\delta')}$ in σ and $\mathtt{x} = \mathtt{v}_s$ such that $\sigma(\mathtt{x}) \sim_\ell \sigma'(\mathtt{x})$. Thus, either:

- $\mathtt{n}_1 = \mathtt{n}_2 \wedge l = l' \wedge \delta = \delta'$: As $\sigma \simeq_\ell \sigma''$, $\mathtt{v}_s = \mathtt{n}_s^{(l_s,\delta_s)}$ and $\mathtt{n}_1 = \mathtt{n}_s \wedge l = l_s \wedge \delta = \delta_s$. Thus, $\mathtt{n}_2 = \mathtt{n}_s \wedge l' = l_s \wedge \delta' = \delta_s$. Hence, $\sigma' \simeq_\ell \sigma''$ from Definition 25.1
- $l = l' \wedge \delta = \delta' \wedge \exists \mathtt{y}^m \in \delta.m \not\sqsubseteq \ell$: As $\sigma \simeq_\ell \sigma''$, $\mathtt{v}_s = \star^{(l_s,\delta_s)}$ and $l \sqsubseteq \ell$ and $l_s = \top$ and $\exists \mathtt{y}^m \in \delta.\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell$. Thus, $l' \sqsubseteq \ell$ and $l_s = \top$ and $\exists \mathtt{y}^m \in \delta'.\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell$. Else, $l = l_s \wedge \delta = \delta_s$. Thus, $l' = l_s \wedge \delta' = \delta_s$. Hence, $\sigma' \simeq_\ell \sigma''$ from Definition 25.2(b), 25.2(c)
- $l \not\sqsubseteq \ell \wedge l' \not\sqsubseteq \ell$: As $\sigma \simeq_\ell \sigma''$, $\mathtt{v}_s = \star^{(l_s,\delta_s)}$ and $l_s = \top$. Hence, $\sigma' \simeq_\ell \sigma''$ from Definition 25.2(a).

□

Lemma 22 (Expression Simulation)**.** *If* $\langle \sigma, \iota, e \rangle \underset{pc}{\Downarrow} \mathtt{v}, \iota', \tau$, *then*
$\langle \sigma\uparrow_\ell, \iota\uparrow_\ell, e, \tau\uparrow_\ell \rangle \underset{pc}{\downarrow} v', \iota'', \tau'$ *such that* TTv $\simeq_\ell \mathtt{v}'$, $\iota'\uparrow_\ell = \iota''$, *and* $\tau' = \epsilon$

Proof. Assume $\sigma^s = \sigma{\uparrow_\ell}$. Proof by induction on the derivation for expressions and case analysis on the last rule.

- CONST: $\mathtt{n}^\perp = \mathtt{n}^\perp$ and $\delta = \delta' = \{\}$. Thus, $\mathtt{v} \simeq_\ell \mathtt{v}'$. $\iota = \iota'$ and $\iota{\uparrow_\ell} = \iota''$, thus, $\iota'{\uparrow_\ell} = \iota''$. $\tau = \epsilon$, thus, $\tau{\uparrow_\ell} = \tau' = \epsilon$.

- VAR: As $\sigma \simeq_\ell \sigma^s$, $\sigma(\mathtt{x}) \simeq_\ell \sigma^s(\mathtt{x})$, thus $\mathtt{v} \simeq_\ell \mathtt{v}'$. $\iota = \iota'$ and $\iota{\uparrow_\ell} = \iota''$, thus, $\iota'{\uparrow_\ell} = \iota''$. $\tau = \epsilon$, thus, $\tau{\uparrow_\ell} = \tau' = \epsilon$.

- AOP and COP:
  IH1: If $\langle \sigma, \iota, e_1\rangle \underset{pc}{\Downarrow} \mathtt{v}_1, \iota_1, \tau_1$ and $\sigma \simeq_\ell \sigma^s$, then $\langle \sigma^s, \iota{\uparrow_\ell}, e_1, \tau_1{\uparrow_\ell}\rangle \underset{pc}{\downarrow} \mathtt{v}'_1, \iota'_1, \tau'_1$ such that $\mathtt{v}_1 \simeq_\ell \mathtt{v}'_1$, $\iota_1{\uparrow_\ell} = \iota'_1$ and $\tau'_1 = \epsilon$.
  IH2: If $\langle \sigma, \iota_1, e_2\rangle \underset{pc}{\Downarrow} \mathtt{v}_2, \iota_2, \tau_2$ and $\sigma \simeq_\ell \sigma^s$, then $\langle \sigma^s, \iota_1{\uparrow_\ell}, e_2, \tau_2{\uparrow_\ell}\rangle \underset{pc}{\downarrow} \mathtt{v}'_2, \iota'_2, \tau'_2$ such that $\mathtt{v}_2 \simeq_\ell \mathtt{v}'_2$, $\iota_2{\uparrow_\ell} = \iota'_2$ and $\tau'_2 = \epsilon$.
  As $\sigma \simeq_\ell \sigma^s$, from IH1 and IH2 $\mathtt{v}_1 \simeq_\ell \mathtt{v}'_1$ and $\mathtt{v}_2 \simeq_\ell \mathtt{v}'_2$, where $\mathtt{v}_1 = \mathtt{n}_1^{(k_1,\delta_1)}$, $\mathtt{v}_2 = \mathtt{n}_2^{(k_2,\delta_2)}$, $\mathtt{v}'_1 = (\mathtt{n}_{s_1}^{(k'_1,\delta'_1)} \vee \star^{(k'_1,\delta'_1)})$, $\mathtt{v}'_2 = (\mathtt{n}_{s_2}^{(k'_2,\delta'_2)} \vee \star^{(k'_2,\delta'_2)})$.
  $\iota' = \iota_2$ and $\iota'' = \iota'_2$, thus from IH1 and IH2, $\iota'{\uparrow_\ell} = \iota''$.
  T.S: $\mathtt{v} \simeq_\ell \mathtt{v}'$, i.e, $\mathtt{v}_1 \odot \mathtt{v}_2 \simeq_\ell \mathtt{v}'_1 \odot \mathtt{v}'_2$.
  As $\mathtt{v}_1 \simeq_\ell \mathtt{v}'_1$, either:

  1. $\mathtt{v}_1 = \mathtt{n}_1^{(k_1,\delta_1)}$, $\mathtt{v}'_1 = \mathtt{n}_{s_1}^{(k'_1,\delta'_1)}$, $\mathtt{n}_1 = \mathtt{n}_{s_1}$ and $k_1 = k'_1 \sqsubseteq \ell$ and $\delta_1 = \delta'_1$:
     As $\mathtt{v}_2 \simeq_\ell \mathtt{v}'_2$, either:
     (a) $\mathtt{v}_2 = \mathtt{n}_2^{(k_2,\delta_2)}$, $\mathtt{v}'_2 = \mathtt{n}_{s_2}^{(k'_2,\delta'_2)}$, $\mathtt{n}_2 = \mathtt{n}_{s_2}$ and $k_2 = k'_2 \sqsubseteq \ell$ and $\delta_2 = \delta'_2$:
     $\mathtt{n}_1 \odot \mathtt{n}_2 = \mathtt{n}_{s_1} \odot \mathtt{n}_{s_2}$ and $(k_1 \sqcup k_2 = k'_1 \sqcup k'_2) \sqsubseteq \ell$ and $\delta_1 \cup \delta_2 = \delta'_1 \cup \delta'_2$. From Definition 25.1, $\mathtt{v} \simeq_\ell \mathtt{v}'$.
     (b) $\mathtt{v}_2 = \mathtt{n}_2^{(k_2,\delta_2)}$ and $\Gamma(\mathtt{v}_2) \not\sqsubseteq \ell$ and $\mathtt{v}'_2 = \star^{(k'_2,\delta'_2)}$:
     - $k'_2 = \top$ and $k_2 \not\sqsubseteq \ell$:
       $(l = k_1 \sqcup k_2) \not\sqsubseteq \ell$, thus $\Gamma(\mathtt{v}) \not\sqsubseteq \ell$, $\mathtt{n}_{s_1} \odot \star = \star$ and $l' = k'_1 \sqcup \top = \top$. From Definition 25.2(a), $\mathtt{v} \simeq_\ell \mathtt{v}'$.
     - $k'_2 = \top$, $k_2 \sqsubseteq \ell$ and $\exists \mathtt{x} \in \delta_2.(\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell)$:
       $(l = k_1 \sqcup k_2) \sqsubseteq \ell$, $(\Gamma(\mathtt{v}) = \Gamma(\mathtt{v}_1) \sqcup \Gamma(\mathtt{v}_2)) \not\sqsubseteq \ell$ and $l' = \top$. $(\mathtt{n}_{s_1} \odot \star = \star)$. $\delta = \delta_1 \cup \delta_2$, thus $\mathtt{x} \in \delta_2 \implies \mathtt{x} \in \delta$, i.e., $\exists \mathtt{x} \in \delta.(\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell)$. From Definition 25.2(b), $\mathtt{v} \simeq_\ell \mathtt{v}'$.
     - $(k_2 = k'_2) \sqsubseteq \ell \ \wedge \ \delta_2 = \delta'_2$:
       $(l = k_1 \sqcup k_2) \sqsubseteq \ell$ and $(\Gamma(\mathtt{v}) = \Gamma(\mathtt{v}_1) \sqcup \Gamma(\mathtt{v}_2)) \not\sqsubseteq \ell$. As $\delta_1 = \delta'_1$, $\delta = \delta_1 \cup \delta_2$, and $\delta' = \delta'_1 \cup \delta'_2$. $\delta = \delta'$. $k_1 = k'_1$ and $k_2 = k'_2$, so, $l = l'$. From Definition 25.2(c), $\mathtt{v} \simeq_\ell \mathtt{v}'$

  2. $\mathtt{v}_1 = \mathtt{n}_1^{(k_1,\delta_1)}$, $\Gamma(\mathtt{v}_1) \not\sqsubseteq \ell$, $\mathtt{v}'_1 = \star^{(k'_1,\delta'_1)}$:
     As $\mathtt{v}_2 \simeq_\ell \mathtt{v}'_2$, either:
     - $\mathtt{v}_2 = \mathtt{n}_2^{(k_2,\delta_2)}$, $\mathtt{v}'_2 = \mathtt{n}_{s_2}^{(k'_2,\delta'_2)}$, $\mathtt{n}_2 = \mathtt{n}_{s_2}$, $k_2 = k'_2 \sqsubseteq \ell$ and $\delta_2 = \delta'_2$:
       Similar to the case (1.b) above
     - $\mathtt{v}_2 = \mathtt{n}_2^{(k_2,\delta_2)}$, $\Gamma(\mathtt{v}_2) \not\sqsubseteq \ell$ and $\mathtt{v}'_2 = \star^{(k'_2,\delta'_2)}$:
       * $k'_1 = \top$ and $k_1 \not\sqsubseteq \ell$:
         $l' = k'_1 \sqcup k'_2 = \top$ and $(l = k_1 \sqcup k_2) \not\sqsubseteq \ell$. From Definition 25.2(a) $\mathtt{v} \simeq_\ell \mathtt{v}'$

* $\mathtt{v}'_1 = \star^{(k'_1,\delta'_1)}$, $k'_1 = \top$, $k_1 \sqsubseteq \ell$, and $\exists \mathtt{x} \in \delta_1.(\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell)$:
  $\delta = \delta_1 \cup \delta_2$ and $l' = k'_1 \sqcup k'_2 = \top$. As $l' = \top$ either:
  $(l = k_1 \sqcup k_2) \sqsubseteq \ell$. As $\mathtt{x} \in \delta_1 \implies \mathtt{x} \in \delta$. Thus, $\exists \mathtt{x} \in \delta.(\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell)$. From Definition 25.2(b), $\mathtt{v} \simeq_\ell \mathtt{v}'$ (or)
  $k_2 \not\sqsubseteq \ell$ and $(l = k_1 \sqcup k_2) \not\sqsubseteq \ell$. From Definition 25.2(a), $\mathtt{v} \simeq_\ell \mathtt{v}'$
* $(k_1 = k'_1) \sqsubseteq \ell$ and $\delta_1 = \delta'_1$:
  As $\mathtt{v}_2 \simeq_\ell \mathtt{v}'_2$, either:
  $k'_2 = \top$ and $k_2 \not\sqsubseteq \ell$. $l = k_1 \sqcup k_2$ and $l' = k'_1 \sqcup k'_2$. From Definition 25.2(a), $\mathtt{v} \simeq_\ell \mathtt{v}'$. (or)
  $k_2 \sqsubseteq \ell$, $k'_2 = \top$ and $\exists \mathtt{x} \in \delta_2.(\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell)$. $k_1 \sqcup k_2 \sqsubseteq \ell$, $k'_1 \sqcup k'_2 = \top$, $\delta = \delta_1 \cup \delta_2$ and $\exists \mathtt{x} \in \delta.(\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell)$. From Definition 25.2(b), $\mathtt{v} \simeq_\ell \mathtt{v}'$ (or)
  $(k_2 = k'_2) \sqsubseteq \ell$, $\delta_2 = \delta'_2$: $l = k_1 \sqcup k_2$ and $l' = k'_1 \sqcup k'_2$, so $l = l'$. $\delta = \delta_1 \cup \delta_2$ and $\delta' = \delta'_1 \cup \delta'_2$. Thus, $\delta = \delta'$. From Definition 25.2(c), $\mathtt{v} \simeq_\ell \mathtt{v}'$.

T.S.: If $\langle \sigma, \iota, e\rangle \underset{pc}{\Downarrow} \mathtt{n}^{(l,\delta)}, \iota', \tau$, and $\langle \sigma^s, \iota, e, \tau{\uparrow_\ell}\rangle \underset{pc}{\downarrow} n', \delta', \iota'', \tau'$, and $\sigma \simeq_\ell \sigma^s$, then $\tau' = \epsilon$. $\tau = \tau_1 :: \tau_2$.

$\tau{\uparrow_\ell} = \tau_1{\uparrow_\ell} :: \tau_2{\uparrow_\ell}$. From AOP, COP and IH1, $\langle \sigma^s, \iota, e_1, \tau_1{\uparrow_\ell} :: \tau_2{\uparrow_\ell}\rangle \underset{pc}{\downarrow} \mathtt{n}'_1, \delta'_1, \iota'_1, \tau_2{\uparrow_\ell}$ because $\tau'_1 = \epsilon$. From AOP, COP and IH2, $\tau' = \epsilon$

- DCOPR: IH1: If $\langle \sigma, \iota, e_1\rangle \underset{pc}{\Downarrow} \mathtt{v}_1, \iota_1, \tau_1$ and $\sigma \simeq_\ell \sigma^s$, then $\langle \sigma^s, \iota{\uparrow_\ell}, e_1, \tau_1{\uparrow_\ell}\rangle \underset{pc}{\downarrow} \mathtt{v}'_1, \iota'_1, \tau'_1$ such that $\mathtt{v}_1 \simeq_\ell \mathtt{v}'_1$, $\iota_1{\uparrow_\ell} = \iota'_1$ and $\tau'_1 = \epsilon$.
  IH2: If $\langle \sigma, \iota_1, e_2\rangle \underset{pc}{\Downarrow} \mathtt{v}_2, \iota_2, \tau_2$ and $\sigma \simeq_\ell \sigma^s$, then $\langle \sigma^s, \iota_1{\uparrow_\ell}, e_2, \tau_2{\uparrow_\ell}\rangle \underset{pc}{\downarrow} \mathtt{v}'_2, \iota'_2, \tau'_2$ such that $\mathtt{v}_2 \simeq_\ell \mathtt{v}'_2$, $\iota_2{\uparrow_\ell} = \iota'_2$ and $\tau'_2 = \epsilon$.
  As $\sigma \simeq_\ell \sigma^s$, from IH1 and IH2 $\mathtt{v}_1 \simeq_\ell \mathtt{v}'_1$, $\mathtt{v}_2 \simeq_\ell \mathtt{v}'_2$ and $\tau'_1 = \tau'_2 = \epsilon$.
  Also from IH1 and IH2, $\iota_2{\uparrow_\ell} = \iota'_2$.
  T.S: $\mathtt{v} \simeq_\ell \mathtt{v}'$, $\iota'{\uparrow_\ell} = \iota''$ and $\tau' = \epsilon$.
  $\mathtt{v} = \mathtt{v}_1 \oplus \mathtt{v}_2$ and $\mathtt{v}' = \mathtt{v}'_1 \oplus v'_2$.
  As $\mathtt{v}_1 \simeq_\ell \mathtt{v}'_1$, either:

  1. $\mathtt{v}_1 = \mathtt{n}_1^{(k_1,\delta_1)}$, $\mathtt{v}'_1 = \mathtt{n}_{s_1}^{(k'_1,\delta'_1)}$, $\mathtt{n}_1 = \mathtt{n}_{s_1}$, $k_1 = k'_1 \sqsubseteq \ell$ and $\delta_1 = \delta'_1$:
     As $\mathtt{v}_2 \simeq_\ell \mathtt{v}'_2$, either:
     (a) $\mathtt{v}_2 = \mathtt{n}_2^{(k_2,\delta_2)}$, $\mathtt{v}'_2 = \mathtt{n}_{s_2}^{(k'_2,\delta'_2)}$, $\mathtt{n}_2 = \mathtt{n}_{s_2}$, $k_2 = k'_2 \sqsubseteq \ell$ and $\delta_2 = \delta'_2$:
     Assume $\delta_i = \delta_1 \cup \delta_2$ and $\delta'_i = \delta'_1 \cup \delta'_2$, thus, $\delta_i = \delta'_i$. As $(l = k_1 \sqcup k_2 = k'_1 \sqcup k'_2 = l')$, $\delta_i = \delta'_i$ and $\iota_2{\uparrow_\ell} = \iota'_2$, $\delta_i = \delta'_i$. Either:
     - $\delta_i = \delta'_i = \emptyset$: $n = \mathtt{n}_1 \oplus \mathtt{n}_2 = \mathtt{n}_{s_1} \oplus \mathtt{n}_{s_2} = \mathtt{n}_s$ and $(l = k_1 \sqcup k_2 = k'_1 \sqcup k'_2 = l') \sqsubseteq \ell$, thus, $\mathtt{v} \simeq_\ell \mathtt{v}'$.
       Also, $\iota' = \iota_2$ and $\iota'' = \iota'_2$. Thus, $\iota'{\uparrow_\ell} = \iota''$.
       $\tau = \tau_1 :: \tau_2$. From S-DCOPR and IH1, $\langle \sigma^s, \iota{\uparrow_\ell}, e_1, \tau_1{\uparrow_\ell} :: \tau_2{\uparrow_\ell}\rangle \underset{pc}{\downarrow} \mathtt{v}'_1, \iota'_1, \tau_2{\uparrow_\ell}$ because $\tau'_1 = \epsilon$. From S-DCOPR and IH2, $\tau' = \tau'_2 = \epsilon$.
     - $(\delta_i = \delta'_i) \neq \emptyset$: $\tau = \tau_1 :: \tau_2 :: \mathtt{n}^l$. From S-DCOPR and IH1, $\langle \sigma^s, \iota{\uparrow_\ell}, e_1, \tau_1 :: \tau_2 :: n\rangle \underset{pc}{\downarrow} \mathtt{v}'_1, \iota'_1, \tau_2 :: n$ because $\tau'_1 = \epsilon$. From S-DCOPR and IH2, $\langle \sigma^s, \iota_1{\uparrow_\ell}, e_2, \tau_2 :: n\rangle \underset{pc}{\downarrow} \mathtt{v}'_2, \iota'_2, n$ because $\tau'_2 = \epsilon$. From S-DCOPR, $\tau' = \epsilon$ and $\tau_2 = \mathtt{n}^l$. Thus, $\mathtt{v} \simeq_\ell \mathtt{v}'$.
       As $\delta_i = \delta'_i$ and $\iota_2{\uparrow_\ell} = \iota'_2$, $\iota'{\uparrow_\ell} = \iota''$.

(b) $\mathtt{v}_2 = \mathtt{n}_2^{(k_2,\delta_2)}$, $\Gamma(\mathtt{v}_2) \not\sqsubseteq \ell$, $\mathtt{v}_2' = \star^{(k_2',\delta_2')}$ and either:

- $k_2 \not\sqsubseteq \ell$ and $k_2' = \top$: $(k_o = k_1 \sqcup k_2) \not\sqsubseteq \ell$, thus, $l \not\sqsubseteq \ell$. As $l' = \top$ from S-DCOPR, $\delta_i' = \emptyset$ and $\mathtt{v}' = \star^{(\top,\{\})}$. From Definition 25.2(a), $\mathtt{v} \simeq_\ell \mathtt{v}'$. From S-DCOPR, IH1, and IH2, $\iota_2{\uparrow_\ell} = \iota_2'$ and $\tau_2' = \epsilon$. Thus, $\iota'{\uparrow_\ell} = \iota''$ and $\tau' = \epsilon$.
- $k_2' = \top$ and $k_2 \sqsubseteq \ell$ and $\exists \mathtt{x} \in \delta_2.(\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell)$:
  $\mathtt{v}' = \star^{(\top,\delta')}$. $(k_o = k_1 \sqcup k_2) \sqsubseteq \ell$ and $\delta = \delta_1 \cup \delta_2$. As $\exists \mathtt{x} \in \delta.(\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell)$, $l \not\sqsubseteq \ell$. If $\iota_2(\mathtt{x}) = 0$, then $l \sqsupseteq k_o \sqcup m$, else $l \sqsupseteq k_o \sqcup \mathbb{B}(\mathtt{x})$. In either case, $l \not\sqsubseteq \ell$. Also, $l = k_1' \sqcup k_2' = \top$ so $\delta_i' = \emptyset$. From Definition 25.2(a), $\mathtt{v} \simeq_\ell \mathtt{v}'$. From S-DCOPR, IH1, and IH2, $\iota_2{\uparrow_\ell} = \iota_2'$ and $\tau_2' = \epsilon$. Thus, $\iota'{\uparrow_\ell} = \iota''$ and $\tau' = \epsilon$.
- $k_2 = k_2' \sqsubseteq \ell$ and $\delta_2 = \delta_2'$:
  $\delta = \delta_1 \cup \delta_2$ and $\delta' = \delta_1' \cup \delta_2'$. As $\delta_1 = \delta_1'$, $\delta = \delta'$. From IH1, and IH2, $\iota_2{\uparrow_\ell} = \iota_2'$ and $\tau_2' = \epsilon$. If $\exists x.\iota_2{\uparrow_\ell}(\mathtt{x}) = \iota_2(\mathtt{x}) = 0 \ \wedge \ \mathbb{L}(\mathtt{x}) \not\sqsubseteq \ell$, then $l \not\sqsubseteq \ell$ and $l' = \top$. Else $l = l' \sqsubseteq \ell$. Thus, $\delta_i = \delta_i'$. Thus, if $\delta_i = \delta_i' = \emptyset$, $l' = \top$ and $l \not\sqsubseteq \ell$, $\iota' = \iota_2{\uparrow_\ell} = \iota_2' = \iota''$ and $\tau' = \tau_2' = \epsilon$. Else, $\tau = \tau_1 :: \tau_2 :: \mathtt{n}^l$ and $\tau_2{\uparrow_\ell} = \mathtt{n}^l :: \tau'$ and $\mathtt{v}' = \mathtt{n}^l$. Hence, $\mathtt{v} \simeq_\ell \mathtt{v}'$ and $\tau' = \epsilon$. As $\delta = \delta'$ and $\iota_2{\uparrow_\ell} = \iota_2'$, $\iota'{\uparrow_\ell} = \iota''$

2. $\mathtt{v}_1 = \mathtt{n}_1^{(k_1,\delta_1)}$ and $\Gamma(\mathtt{v}_1) \not\sqsubseteq \ell$ and $\mathtt{v}_1' = \star^{(k_1',\delta_1')}$:
   As $\mathtt{v}_2 \simeq_\ell \mathtt{v}_2'$, either:
   (a) $\mathtt{v}_2 = \mathtt{n}_2^{(k_2,\delta_2)}$, $\mathtt{v}_2' = \mathtt{n}_{s_2}^{(k_2',\delta_2')}$, $\mathtt{n}_2 = \mathtt{n}_{s_2}$, $k_2 = k_2' \sqsubseteq \ell$ and $\delta_2 = \delta_2'$: Similar to case (1.b)
   (b) $\mathtt{v}_2 = \mathtt{n}_2^{(k_2,\delta_2)}$ and $\Gamma(\mathtt{v}_2) \not\sqsubseteq \ell$ and $\mathtt{n}_2' = \star^{(k_2',\delta_2')}$:
   Different cases of Definition 25.2:
   - $k_1' = \top$ and $k_1 \not\sqsubseteq \ell$: From DCOPR, $l \not\sqsubseteq \ell$ and from S-DCOPR $l' = \top$. $\forall \mathtt{x} \in \delta.\mathbb{L}(\mathtt{x}) \not\sqsubseteq \ell$, so $\iota_2{\uparrow_\ell} = \iota'{\uparrow_\ell}$ and $\tau{\uparrow_\ell} = \tau_1{\uparrow_\ell} :: \tau_2{\uparrow_\ell}$. As $\delta = \emptyset$, $\iota_2' = \iota''$ and $\tau' = \tau_2' = \epsilon$. Thus, $\iota'' = \iota'{\uparrow_\ell}$ and from Definition 25.2(a), $\mathtt{v} \simeq_\ell \mathtt{v}'$.
   - $k_1 \sqsubseteq \ell$ and $\exists \mathtt{x} \in \delta_1.(\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell)$ and $k_1' = \top$: $\exists \mathtt{x} \in \delta.(\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell)$ as $\delta = \delta_1 \cup \delta_2$. From DCOPR, $l \not\sqsubseteq \ell$ and $l' = \top$. Similar to above case.
   - $k_1 \sqsubseteq \ell$ and $\delta_1 = \delta_1'$:
     If $k_2' = \top$ and either $k_2 \not\sqsubseteq \ell$ or $k_2 \sqsubseteq \ell$ and $\exists \mathtt{x} \in \delta_2.(\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell)$ - similar to above two cases.
     If $k_2 \sqsubseteq \ell$ and $\delta_2 = \delta_2'$: $\delta = \delta_1 \cup \delta_2$ and $\delta' = \delta_1' \cup \delta_2'$ - similar to last case of 1.b

- DCOPN: Similar to AOP, COP.

□

Theorem 8 (Simulation Theorem)**.** *If* $\langle \sigma, \iota, c\rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau\rangle$, *then* $\langle \sigma{\uparrow_\ell}, \iota{\uparrow_\ell}, c, \tau{\uparrow_\ell}\rangle \underset{pc}{\downarrow} \langle \sigma'', \iota'', \tau''\rangle$ *such that* $\sigma' \simeq_\ell \sigma''$, $\iota'{\uparrow_\ell} = \iota''$ *and* $\tau'' = \epsilon$

Proof. Assume $\sigma{\uparrow_\ell} = \sigma^s$. Thus, $\sigma \simeq_\ell \sigma^s$. As ℓ simulates the run, initial $pc \sqsubseteq \ell$. Proof by induction on the derivation for commands and case analysis on the last rule.

- SKIP: $\sigma' = \sigma \simeq_\ell \sigma^s = \sigma''$. $\tau = \epsilon$. Thus $\tau{\uparrow_\ell} = \tau'' = \epsilon$. $\iota = \iota'$ and $\iota{\uparrow_\ell} = \iota''$. So, $\iota'{\uparrow_\ell} = \iota''$

- ASSN: As $\sigma \simeq_\ell \sigma^s$, $\sigma(\mathtt{x}) \simeq_\ell \sigma^s(\mathtt{x})$. Thus, either $(\Gamma_f(\sigma(\mathtt{x})) = \Gamma_f(\sigma^s(\mathtt{x})))$ (or) $\Gamma_f(\sigma(\mathtt{x})) \not\sqsubseteq \ell$ and $\Gamma_f(\sigma^s(\mathtt{x})) = \top$. In either case, $pc \sqsubseteq \Gamma_f(\sigma(\mathtt{x}))$ and $pc \sqsubseteq \Gamma_f(\sigma^s(\mathtt{x}))$. From Lemma 22, $\mathtt{n}^{(m,\delta')} \simeq_\ell \mathtt{n}_s^{(m',\delta'')}$, $\iota'\uparrow_\ell = \iota''$ and $\tau'' = \epsilon$. From Definition 25, $\mathtt{n}^{(l,\delta)} \simeq_\ell \mathtt{n}_s^{(l_s,\delta_s)}$. Thus, $\sigma' \simeq_\ell \sigma''$, $\iota'\uparrow_\ell = \iota''$ and $\tau'' = \epsilon$.

- SEQ: IH1: If $\langle \sigma, \iota, c_1\rangle \underset{pc}{\Downarrow} \langle \sigma_1, \iota_1, \tau_1\rangle$, then $\langle \sigma\uparrow_\ell, \iota\uparrow_\ell, c_1, \tau_1\uparrow_\ell\rangle \underset{pc}{\downarrow} \langle \sigma_1', \iota_1', \tau_1'\rangle$ such that $\sigma_1 \simeq_\ell \sigma_1'$, $\iota_1\uparrow_\ell = \iota_1'$ and $\tau_1' = \epsilon$
  IH2: If $\langle \sigma_1, \iota_1, c_2\rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau_2\rangle$, then $\langle \sigma_1\uparrow_\ell, \iota_1\uparrow_\ell, c_2, \tau_2\uparrow_\ell\rangle \underset{pc}{\downarrow} \langle \sigma'', \iota'', \tau''\rangle$ such that $\sigma' \simeq_\ell \sigma''$, $\iota'\uparrow_\ell = \iota''$ and $\tau'' = \epsilon$.
  T.S : If $\langle \sigma, \iota, c_1; c_2\rangle \underset{pc}{\Downarrow} \langle \sigma', \iota', \tau_1 :: \tau_2\rangle$, then
  $\langle \sigma\uparrow_\ell, \iota\uparrow_\ell, c_1; c_2, \tau_1\uparrow_\ell :: \tau_2\uparrow_\ell\rangle \underset{pc}{\downarrow} \langle \sigma'', \iota'', \tau''\rangle$ such that $\sigma' \simeq_\ell \sigma''$, $\iota'\uparrow_\ell = \iota''$ and $\tau'' = \epsilon$.
  From IH1, $\sigma_1' = \sigma_1\uparrow_\ell$ and $\iota_1\uparrow_\ell = \iota_1'$, thus, from IH2 $\sigma' \simeq_\ell \sigma''$ and $\iota'\uparrow_\ell = \iota''$.
  From IH1 $\tau_1' = \epsilon$, and Corollary 5, $\langle \sigma\uparrow_\ell, \iota\uparrow_\ell, c_1, \tau_1\uparrow_\ell :: \tau_2\uparrow_\ell\rangle \underset{pc}{\downarrow} \langle \sigma'', \iota'', \tau_2\uparrow_\ell\rangle$. From IH2, $\tau'' = \epsilon$.

- IF-ELSE: From Lemma 22, $\mathtt{v} \simeq_\ell \mathtt{v}'$, $\iota_1\uparrow_\ell = \iota_1'$, and $\tau_1' = \epsilon$.
  IH1: If $\langle \sigma, \iota_1, c_i\rangle \underset{pc\sqcup k}{\Downarrow} \langle \sigma', \iota', \tau_2\rangle$, then $\langle \sigma\uparrow_\ell, \iota_1\uparrow_\ell, c_i, \tau_2\uparrow_\ell\rangle \underset{pc\sqcup k}{\downarrow} \langle \sigma'', \iota'', \tau''\rangle$ such that $\sigma' \simeq_\ell \sigma''$, $\iota'\uparrow_\ell = \iota''$ and $\tau'' = \epsilon$.
  If $\mathtt{v}' = \mathtt{n}_s^{(l',\delta')}$, then from IH1 $\sigma' \simeq_\ell \sigma''$, $\iota'\uparrow_\ell = \iota''$ and $\tau'' = \epsilon$.
  Else from confinement lemma 16 and Lemma 21, $\sigma' \simeq_\ell \sigma''$, $\iota'\uparrow_\ell = \iota''$ and $\tau'' = \epsilon$.

- WHILE-T and WHILE-F: Similar to if and seq above

□

Lemma 23. *If $\sigma_1 \sim_\ell \sigma_2$ then $\sigma_1\uparrow_\ell = \sigma_2\uparrow_\ell$*

Proof. From Definition 21, $\sigma_1 \sim_\ell \sigma_2 \implies \forall \mathtt{x}.\sigma_1(\mathtt{x}) \sim_\ell \sigma_2(\mathtt{x})$. Assume $\sigma_1(\mathtt{x}) = \mathtt{n}_1^{(k_1,\delta_1)}$ and $\sigma_2(\mathtt{x}) = \mathtt{n}_2^{(k_2,\delta_2)}$. From Definition 20, either:

- $\mathtt{n}_1 = \mathtt{n}_2$, $k_1 = k_2 \sqsubseteq \ell$ and $\delta_1 = \delta_2$: By definition of $\sigma\uparrow_\ell$, $\sigma_1\uparrow_\ell(\mathtt{x}) = \mathtt{n}_1^{(k_1,\delta_1)}$ and $\sigma_2\uparrow_\ell(\mathtt{x}) = \mathtt{n}_2^{(k_2,\delta_2)}$. Hence, $\sigma_1\uparrow_\ell(\mathtt{x}) = \sigma_2\uparrow_\ell(\mathtt{x})$
- $\Gamma(\mathtt{v}_1) \not\sqsubseteq \ell$ and $\Gamma(\mathtt{v}_2) \not\sqsubseteq \ell$ such that $k_1 = k_2 \sqsubseteq \ell$ and $\delta_1 = \delta_2$: By definition of $\sigma\uparrow_\ell$, $\sigma_1\uparrow_\ell(\mathtt{x}) = \star^{(k_1,\delta_1)}$ and $\sigma_2\uparrow_\ell(\mathtt{x}) = \star^{(k_2,\delta_2)}$. Hence, $\sigma_1\uparrow_\ell(\mathtt{x}) = \sigma_2\uparrow_\ell(\mathtt{x})$
- $\Gamma(\mathtt{v}_1) \not\sqsubseteq \ell$ and $\Gamma(\mathtt{v}_2) \not\sqsubseteq \ell$ such that $k_1 \not\sqsubseteq \ell$ and $k_2 \not\sqsubseteq \ell$: By definition of $\sigma\uparrow_\ell$, $\sigma_1\uparrow_\ell(\mathtt{x}) = \star^{(\top,\{\})}$ and $\sigma_2\uparrow_\ell(\mathtt{x}) = \star^{(\top,\{\})}$. Hence, $\sigma_1\uparrow_\ell(\mathtt{x}) = \sigma_2\uparrow_\ell(\mathtt{x})$

□

Lemma 24. *If $\iota_1 \sim_\ell \iota_2$ then $\iota_1\uparrow_\ell = \iota_2\uparrow_\ell$*

Proof. From Definition 22, $\iota_1 \sim_\ell \iota_2 \implies \forall \mathtt{x}.\mathbb{B}(\mathtt{x}) \sqsubseteq \ell \implies \iota_1(\mathtt{x}) = \iota_2(\mathtt{x})$. By definition of $\iota\uparrow_\ell$, if $(\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell$ then $\iota_1\uparrow_\ell(\mathtt{x}) = 0$ else $\iota_1\uparrow_\ell(\mathtt{x}) = \iota_1(\mathtt{x}))$ and if $(\mathbb{B}(\mathtt{x}) \not\sqsubseteq \ell)$ then $\iota_2\uparrow_\ell(\mathtt{x}) = 0$ else $\iota_2\uparrow_\ell(\mathtt{x}) = \iota_2(\mathtt{x}))$. Thus, if $\mathbb{B}(\mathtt{x}) \sqsubseteq \ell$, then $\iota_1\uparrow_\ell(\mathtt{x}) = \iota_2\uparrow_\ell(\mathtt{x})$ else $\iota_1\uparrow_\ell(\mathtt{x}) = \iota_2\uparrow_\ell(\mathtt{x}) = 0$. Thus, $\iota_1\uparrow_\ell = \iota_2\uparrow_\ell$. □

Lemma 25. *If $\sigma_1 \simeq_\ell \sigma'$ and $\sigma_2 \simeq_\ell \sigma'$, then $\sigma_1 \sim_\ell \sigma_2$*

Proof. From Definition 26, $\sigma_1 \simeq_\ell \sigma' \implies \forall \mathtt{x}.\sigma_1(\mathtt{x}) \simeq_\ell \sigma'(\mathtt{x})$ and $\sigma_2 \simeq_\ell \sigma' \implies \forall \mathtt{x}.\sigma_2(\mathtt{x}) \simeq_\ell \sigma'(\mathtt{x})$.
Assume $\sigma_1(\mathtt{x}) = \mathtt{n}_1^{(k_1,\delta_1)}$, $\sigma_2(\mathtt{x}) = \mathtt{n}_2^{(k_2,\delta_2)}$ and $\sigma'(\mathtt{x}) = \mathtt{v}^{(l',\delta')}$.
T.S.: $\forall \mathtt{x}.\sigma_1(\mathtt{x}) \sim_\ell \sigma_2(\mathtt{x})$ From Definition 25 for $\sigma_1(\mathtt{x}) \simeq_\ell \sigma'(\mathtt{x})$, either:

- $\mathtt{v} = \mathtt{n}_s$ and $\mathtt{n}_1 = \mathtt{n}_s$ and $(k_1 = l') \sqsubseteq \ell$ and $\delta_1 = \delta'$: As $\sigma_2(\mathtt{x}) \simeq_\ell \sigma'(\mathtt{x})$, by Definition 25.1 $\mathtt{n}_2 = \mathtt{n}_s$ and $(k_1, \delta_1) = (l', \delta')$. Thus, $\sigma_1(\mathtt{x}) = \sigma_2(\mathtt{x})$.
- $\Gamma(\sigma_1(\mathtt{x})) \not\sqsubseteq \ell$, $\mathtt{v} = \star$ and either:
  1. $k_1 \not\sqsubseteq \ell \ \wedge \ l' = \top$: Either $k_2 \not\sqsubseteq \ell$ or $k_2 \sqsubseteq \ell \ \wedge \ \exists \mathtt{y} \in \delta.(\mathbb{B}(\mathtt{y}) \not\sqsubseteq \ell$: From Definition 20.
  2. $k_1 \sqsubseteq \ell \ \wedge \ \exists \mathtt{y} \in \delta.(\mathbb{B}(\mathtt{y}) \not\sqsubseteq \ell) \ \wedge \ l' = \top$: Similar to above case
  3. $(k_1 = l') \sqsubseteq \ell \ \wedge \ \delta_1 = \delta'$: As $l' \sqsubseteq \ell$, $k_2 \sqsubseteq \ell \ \wedge \ \delta_2 = \delta'$. From Definition 20.(2).

□

Lemma 26. *If $\langle \sigma_1, \iota_1, c\rangle \underset{pc}{\Downarrow} \langle \sigma_1', \iota_1', \tau_1\rangle$, $\langle \sigma_2, \iota_2, c\rangle \underset{pc}{\Downarrow} \langle \sigma_2', \iota_2', \tau_2\rangle$, $\sigma_1 \sim_\ell \sigma_2$, $\iota_1 \sim_\ell \iota_2$, and $\tau_1 \sim_\ell \tau_2$, then $\sigma_1' \sim_\ell \sigma_2'$.*

Proof. From Theorem 8, if $\langle \sigma_1, \iota_1, c\rangle \underset{pc}{\Downarrow} \langle \sigma_1', \iota_1', \tau_1\rangle$,
then $\langle \sigma_1{\uparrow_\ell}, \iota_1{\uparrow_\ell}, c, \tau_1{\uparrow_\ell}\rangle \underset{pc}{\downarrow} \langle \sigma_1'', \iota_1'', \tau_1''\rangle$ such that $\sigma_1' \simeq_\ell \sigma_1''$, $\iota_1'{\uparrow_\ell} = \iota_1''$ and $\tau_1'' = \epsilon$ and
if $\langle \sigma_2, \iota_2, c\rangle \underset{pc}{\Downarrow} \langle \sigma_2', \iota_2', \tau_2\rangle$, then
$\langle \sigma_2{\uparrow_\ell}, \iota_2{\uparrow_\ell}, c, \tau_2{\uparrow_\ell}\rangle \underset{pc}{\downarrow} \langle \sigma_2'', \iota_2'', \tau_2''\rangle$ such that $\sigma_2' \simeq_\ell \sigma_2''$, $\iota_2'{\uparrow_\ell} = \iota_2''$ and $\tau_2'' = \epsilon$.
$\sigma_1 \sim_\ell \sigma_2 \implies \sigma_1{\uparrow_\ell} = \sigma_2{\uparrow_\ell}$. $\iota_1 \sim_\ell \iota_2 \implies \iota_1{\uparrow_\ell} = \iota_2{\uparrow_\ell}$.
As $\tau_1 \sim_\ell \tau_2$, from Definition 23 $\tau_1{\uparrow_\ell} = \tau_2{\uparrow_\ell}$.
Thus, $\sigma_1'' = \sigma_2''$, $\iota_1'' = \iota_2''$ and $\tau_1'' = \tau_2''$.
$\sigma_1' \simeq_\ell \sigma_1''$, and $\sigma_2' \simeq_\ell \sigma_2''$, i.e., $\sigma_2' \simeq_\ell \sigma_1''$. From Lemma 25. □

Corollary 6 (Non-interference)**.** *If $\langle \sigma_1, \iota_1, c\rangle \underset{pc}{\Downarrow} \langle \sigma_1', \iota_1', \tau_1\rangle$, $\langle \sigma_2, \iota_2, c\rangle \underset{pc}{\Downarrow} \langle \sigma_2', \iota_2', \tau_2\rangle$, $\sigma_1 \sim_\ell \sigma_2$, and $\iota_1{\uparrow_\ell} = \iota_2{\uparrow_\ell} = 0$, then $\sigma_1' \sim_\ell \sigma_2'$.*

Proof. From Theorem 7 and Lemma 26 □

List of Figures

List of Tables

Bibliography

[1] Alexa top 500 global sites. URL http://www.alexa.com/topsites. [Online; accessed 25-Apr-2017].

[2] Google Caja - A source-to-source translator for securing JavaScript-based web content. https://developers.google.com/caja/. [Online; accessed 25-Apr-2017].

[3] Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/. [Online; accessed 25-Apr-2017].

[4] Content Security Policy Level 3. https://www.w3.org/TR/CSP3/. [Online; accessed 25-Apr-2017].

[5] Dromaeo: JS performance testing. URL http://dromaeo.com/. [Online; accessed 25-Apr-2017].

[6] Facebook. FBJS. https://developers.facebook.com/docs/javascript. [Online; accessed 25-Apr-2017].

[7] Sunspider 1.0.2 javascript benchmark. URL http://www.webkit.org/perf/sunspider/sunspider.html. [Online; accessed 25-Apr-2017].

[8] World's Biggest Data Breaches. http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/, 2017. [Online; accessed 25-Apr-2017].

[9] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications. In *Proc. Annual Computer Security Applications Conference*, pages 1–10, 2012.

[10] M. S. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith. Measuring information leakage using generalized gain functions. In *Proc. 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, pages 265–279, 2012.

[11] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proc. 2007 IEEE Symposium on Security and Privacy (SP '07)*, pages 207–221, 2007.

[12] A. Askarov and A. Sabelfeld. Localized delimited release: Combining the what and where dimensions of information release. In *Proc. 2007 Workshop on Programming Languages and Analysis for Security*, PLAS '07, pages 53–60, 2007.

[13] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, pages 43–59, 2009.

[14] 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*, pages 45–57, 2009.

[15] 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*, pages 333–348, 2008.

[16] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 113–124, 2009.

[17] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 1–12, 2010.

[18] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 165–178, 2012.

[19] M. Backes, B. Kopf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *Proc. 2009 30th IEEE Symposium on Security and Privacy*, pages 141–153, 2009.

[20] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 339–353, 2008.

[21] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu. Strict control dependence and its effect on dynamic information flow analyses. In *Proc. 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 13–24, 2010.

[22] A. Barth. The web origin concept. `http://tools.ietf.org/html/rfc6454`. [Online; accessed 25-Apr-2017].

[23] F. Besson, N. Bielova, and T. Jensen. Hybrid information flow monitoring against web tracking. In *Proc. IEEE 26th Computer Security Foundations Symposium (CSF '13)*, pages 240–254, 2013.

[24] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Generalizing permissive-upgrade in dynamic information flow analysis. In *Proc. 9th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 15:15–15:24, 2014.

[25] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in WebKit's JavaScript bytecode. In *Proc. Principles of Security and Trust*, pages 159–178. Springer Berlin Heidelberg, 2014.

[26] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information Flow Control in WebKit's JavaScript Bytecode. *CoRR*, abs/1401.4339, 2014. URL `http://arxiv.org/abs/1401.4339`.

[27] A. Bichhawat, V. Rajani, J. Jain, D. Garg, and C. Hammer. WebPol: Fine-grained Information Flow Policies for Web Browsers. In *Proc. 22nd European Symposium on Research in Computer Security (ESORICS '17)*, 2017.

[28] N. Bielova. Dynamic leakage - a need for a new quantitative information flow measure. In *Proc. 11th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2016.

[29] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *Proc. International Conference on Network and System Security*, pages 97–104, 2011.

[30] A. Birgisson, D. Hedin, and A. Sabelfeld. Boosting the permissiveness of dynamic information-flow tracking by testing. In *Proc. European Symposium on Research in Computer Security (ESORICS)*, pages 55–72, 2012.

[31] 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, pages 65–79. IEEE Computer Society, 2014.

[32] S. Chong. Required information release. In *Proc. 2010 23rd IEEE Computer Security Foundations Symposium*, pages 215–227, July 2010.

[33] S. Chong and A. C. Myers. Security policies for downgrading. In *Proc. 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 198–209, 2004.

[34] A. Chudnov and D. A. Naumann. Inlined information flow monitoring for javascript. In *Proc. 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 629–643, 2015.

[35] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–62, 2009.

[36] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *J. Comput. Secur.*, 15(3):321–371, 2007.

[37] M. R. Clarkson, A. C. Myers, and F. B. Schneider. Quantifying information flow with beliefs. *J. Comput. Secur.*, 17(5):655–701, Oct. 2009.

[38] E. Cohen. Information transmission in computational systems. In *Proc. Sixth ACM Symposium on Operating Systems Principles*, SOSP '77, pages 133–139, 1977.

[39] E. Cohen. *Information Transmission in Sequential Programs*. Foundations of Secure Computation. Academic Press, 1978.

[40] T. M. Cover and J. A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, 2006.

[41] D. Crockford. ADsafe. `http://adsafe.org/`. [Online; accessed 25-Apr-2017].

[42] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: a web browser with flexible and precise information flow control. In *Proc. ACM Conference on Computer and Communications Security*, pages 748–759, 2012.

[43] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[44] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[45] D. E. R. Denning. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc., 1982.

[46] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. IEEE Symposium on Security and Privacy*, pages 109–124, 2010.

[47] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Proc. Annual Computer Security Applications Conference*, pages 382–391, 2009.

[48] 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*, pages 1311–1324, 2013.

[49] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143, 1974.

[50] O. Foundation. Information leakage. `https://www.owasp.org/index.php/Information_Leakage`, June 2013.

[51] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 186–197, 2004.

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

[53] 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*, pages 177–187, 2011.

[54] G. L. Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proc. IEEE Computer Security Foundations Symposium*, pages 218–232, 2007.

[55] G. L. Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference on Secure Software*, pages 75–89, 2006.

[56] 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):399–422, Dec. 2009.

[57] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *Proc. IEEE Computer Security Foundations Symposium*, pages 3–18, 2012.

[58] 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*, pages 1663–1671, 2014.

[59] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proc. 29th ACM Symposium on Applied Computing*, 2014.

[60] 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, pages 351–365, 2015.

[61] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCException are belong to us. In *Proc. IEEE Symposium on Security and Privacy (Oakland)*, pages 3–17, 2013.

[62] S. Hunt and D. Sands. On flow-sensitive security types. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 79–90, 2006.

[63] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proc. ACM Conference on Computer and Communications Security*, pages 270–283, 2010.

[64] S. Just, A. Cleary, B. Shirley, and C. Hammer. Information flow analysis for JavaScript. In *Proc. ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*, pages 9–18, 2011.

[65] M. G. Kang, S. McCamant, P. Poosankam, , and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *Proc. 18th Network and Distributed System Security Symposium (NDSS '11)*, 2011.

[66] C. Kerschbaumer, E. Hennigan, P. Larsen, S. Brunthaler, and M. Franz. Towards precise and efficient information flow control in web browsers. In *Proc. Trust and Trustworthy Computing*, pages 187–195. 2013.

[67] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In R. Sekar and A. K. Pujari, editors, *Proc. International Conference on Information Systems Security (ICISS)*, volume 5352 of *Lecture Notes in Computer Science*, pages 56–70. Springer, Dec. 2008.

[68] B. Kopf and A. Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 3–14, July 2010.

[69] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, Jan. 1979.

[70] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 158–170, 2005.

[71] Z. Li, K. Zhang, and X. Wang. Mash-if: Practical information-flow control within client-side mashups. In *Proc. 2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 251–260, June 2010.

[72] 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*, pages 24–24, 2010.

[73] A. Lux and H. Mantel. Declassification with explicit reference points. In *Proc. 14th European Conference on Research in Computer Security*, ESORICS'09, pages 69–85, 2009.

[74] W. Masri and A. Podgurski. Algorithms and tool support for dynamic information flow analysis. *Information & Software Technology*, 51(2):385–404, 2009.

[75] J. L. Massey. Guessing and entropy. In *In Proc. 1994 IEEE International Symposium on Information Theory*, page 204, 1994.

[76] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proc. 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 193–205, 2008.

[77] 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*, pages 481–496, 2010.

[78] M. Miller. *Robust composition: Towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, 2006.

[79] A. C. Myers. JFlow: practical mostly-static information flow control. In *Proc. 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 228–241, 1999.

[80] 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.

[81] 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, pages 736–747, 2012.

[82] F. Pottier and V. Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, Jan. 2003.

[83] W. Rafnsson and A. Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *Proc. 2013 IEEE 26th Computer Security Foundations Symposium*, pages 33–48, 2013.

[84] W. Rafnsson and A. Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *Proc. 2013 IEEE 26th Computer Security Foundations Symposium*, CSF '13, pages 33–48, 2013.

[85] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer. Information flow control for event handling and the DOM in web browsers. In *Proc. IEEE 28th Computer Security Foundations Symposium (CSF '15)*, pages 366–379, 2015.

[86] 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 *Proc. European Conference on Object-Oriented Programming*, pages 52–78, 2011.

[87] 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, pages 305–322, 2013.

[88] B. P. S. Rocha, S. Bandhakavi, J. d. Hartog, W. H. Winsborough, and S. Etalle. Towards static flow-based declassification for legacy and untrusted programs. In *Proc. 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 93–108, 2010.

[89] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. 2010 IEEE 23rd Computer Security Foundations Symposium*, pages 186–199, 2010.

[90] A. Russo, A. Sabelfeld, and K. Li. Implicit flows in malicious and nonmalicious code. In *Proc. 2009 Marktoberdorf Summer School*, 2009.

[91] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), Jan 2003.

[92] A. Sabelfeld and A. C. Myers. A model for delimited information release. In K. Futatsugi, F. Mizoguchi, and N. Yonezaki, editors, *Software Security - Theories and Systems*, volume 3233 of *LNCS*, pages 174–191. Springer Berlin Heidelberg, 2004.

[93] 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*, pages 352–365, 2010.

[94] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.

[95] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17 (5):517–548, Oct. 2009.

[96] C. E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, Jan. 2001.

[97] G. Smith. *Foundations of Software Science and Computational Structures: 12th International Conference, FOSSACS 2009*, chapter On the Foundations of Quantitative Information Flow, pages 288–302. Springer, 2009.

[98] 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*, pages 131–146, 2014.

[99] 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*, pages 307–316, 2011.

[100] M. Vanhoef, W. De Groef, D. Devriese, F. Piessens, and T. Rezk. Stateful declassification policies for event-driven programs. In *Proc. 2014 IEEE 27th Computer Security Foundations Symposium*, pages 293–307, 2014.

[101] 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 *Proc. Network and Distributed System Security Symposium*, 2007.

[102] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 268–276, 2000.

[103] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, Jan 1996.

[104] B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. In *Proc. 2007 International Symposium on Software Testing and Analysis*, pages 185–195, 2007.

[105] 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, pages 85–96, 2012.

[106] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. 14th IEEE Workshop on Computer Security Foundations*, CSFW '01, pages 15–23, 2001.

[107] S. A. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, August 2002.

[108] Y. Zhou and D. Evans. Protecting private web content from embedded scripts. In *Proc. 16th European Conference on Research in Computer Security*, pages 60–79, 2011.