ENHANCING EXPRESSIVENESS OF INFORMATION FLOW LABELS: RECLASSIFICATION AND PERMISSIVENESS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Elisavet Kozyri

December 2018

ENHANCING EXPRESSIVENESS OF INFORMATION FLOW LABELS:
RECLASSIFICATION AND PERMISSIVENESS

Elisavet Kozyri, Ph.D.
Cornell University 2018

Increasing the expressiveness of information flow labels can improve the permissiveness of an enforcement mechanism. This thesis studies two formulations of expressive information flow labels: *RIF labels* and *label chains*. Restrictions that a *reactive information flow* (RIF) label imposes on a value depend on the sequence of operations used to derive that value. This allows declassication, endorsement, and other forms of reclassication to be supported in a uniform way. *Piecewise noninterference* (PWNI) is introduced as the appropriate security policy. A type system is given for static enforcement of PWNI in programs that associate *checkable classes* of RIF labels with variables. Two checkable classes of RIF labels are described: general-purpose *RIF automata* and κ*-labels* for programs that use cryptographic operations. But labels themselves can encode information, and thus, certain restrictions should be imposed on their use, too. A new family of dynamic enforcement mechanisms is derived to leverage arbitrarily long label chains, where each label in the chain defines restrictions for its predecessor. These enforcers satisfy *Block-safe Noninterference* (BNI), which proscribes leaks from observing variables, label chains, and blocked executions. Theorems characterize where longer label chains improve permissiveness of dynamic enforcement mechanisms that satisfy BNI. These theorems depend on semantic attributes of such mechanisms as well as on initialization, threat model, and size of lattice of labels.

BIOGRAPHICAL SKETCH

Elisavet Kozyri received a Diploma in Electrical and Computer Engineering from the National Technical University of Athens in 2010. She also received the M.S. in Computer Science from Cornell University in 2015. As part of her doctoral studies at Cornell, she completed a graduate minor in Mathematics.

To Pagona.

ACKNOWLEDGEMENTS

I thank my advisor Fred B. Schneider for believing in me, for teaching me to think, write, and present like a scientist, and for never stopping improving the clarity of this thesis. He was the catalyst for my academic and personal evolution. I thank Emin Gün Sirer for giving me the opportunity to teach my first class in Operating Systems, Richard A. Shore for teaching me logic, and Andrew C. Myers for sharing his expertise in the field of information flow control.

I am thankful to my collaborators in research and teaching. To Josée Desharnais, Nadia Tawbi, Owen Arden, and Andrew Bedford for their patience, constructive criticism, and personal support. To Michael R. Clarkson for giving me the opportunity to design and give lectures on information flow control and for setting the best possible example as a lecturer and as an older academic brother.

My doctoral studies would have been colorless without my extremely talented fellow students. I am fortunate to have you as my friends: Deniz Altinbüken, Konstantinos Mamouras, Bishan Ding, Stavros Nikolaou, Thodoris Lykouris, Efe Gencer, and Moontae Lee.

I am always grateful to my family. To my parents, Zoi and Georgios, who let me free to follow my dream and who are always by my side. To my sister, Alexandra, who makes the ocean that separates us seem like a small pond. To my uncle, Giannis, who has always been a third parent to me. I thank my husband, Thodoris, for building together a dream team whose main player is our sweet daughter Zoe—my life.

The research in this dissertation was supported in part by AFOSR grant F9550-16-0250, NSF grant 1642120, and grants from Microsoft. The views and conclusions contained herein are those of the authors and should not be inter-

preted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A *security policy* (henceforth "policy") on data captures requirements about how this data can be used. An *enforcement mechanism* ensures executions of a system satisfy these policies. Any enforcement mechanism has some specification language for formally specifying policies and uses an abstract model of a given system. Based on the formal specifications of policies and the abstract system model, the enforcement mechanism then decides which system executions are allowed.

All system executions allowed by a *sound* enforcement mechanism will satisfy policies associated with all data in that system. So, a sound enforcement mechanism prevents unsafe executions. However, an enforcement mechanism might prevent safe executions, too.

To combat unnecessary conservatism, a sound enforcement mechanism is expected to be *permissive*, and thus not prevent too many safe executions. In some cases, it is impossible to obtain a sound enforcement mechanism that also allows all safe executions (we give an example later in this chapter). One way to improve permissiveness of an enforcement mechanism is by incorporating more information about the system into the abstract system model. For example, permissiveness of a sound enforcement mechanism can be improved by using information available at run time or by modeling smaller units (e.g., words versus files) and smaller computational steps (e.g., commands versus procedures). The resulting mechanism is then characterized as being more *fine-grained*.

An inexpressive specification language potentially could harm soundness

and/or permissiveness. Consider a specification language that does not allow a desired policy to be expressed. The available specification language thus might be able to express either (i) stricter or (ii) weaker policies than the desired one. Option (i) harms permissiveness (with respect to the desired policy). Option (ii) harms soundness (with respect to the desired policy). Therefore, increased expressiveness of a specification language ought to maintain soundness and improve permissiveness. This thesis studies techniques for increasing the expressiveness of specification languages employed by *information flow control* (IFC) mechanisms.

1.1 From Access Control to Information Flow Control

Access control is widely employed for enforcing security policies on data. Data is stored in *objects* (e.g., files, variables). An *access control policy* associated with an object prescribes allowed *accesses*: which *principals* (e.g., user, computer program) are allowed to perform what operations on the object. For *confidentiality*, an access control policy specifies which principals are allowed to read the content of an object; for *integrity*, it specifies which principals are allowed to write the content of an object. The enforcement mechanism for an access control policy often is a *reference monitor*, which intercepts all accesses to the corresponding object and allows only those prescribed by that policy.

With *discretionary access control* (DAC) [72], a policy on an object is selected by the *owner* of that object. An operating system, for instance, will often support DAC, so users can specify policies for files. DAC can be implemented using *access control lists* [33] or *capabilities* [37].

But deciding whether an access is allowed (i.e., whether a principal is allowed to perform an operation on an object) might depend on whether another access is allowed. For example, to prevent leaking confidential information, a principal allowed to read confidential reports might not be allowed to write to public reports. Also, to address conflicts of interest, a principal might not be allowed to read a company's documents if that principal can also read documents of a competitor [21]. DAC cannot express and enforce such policies.

This lack of expressiveness led to a family of enforcement mechanisms known as *mandatory access control* (MAC) [71]. In MAC, the policy on an object is decided by the institution (i.e., company, government) that maintains the object—not by the owner of the object. *Chinese wall policies* is an example of MAC that address conflicts of interests. Another example of MAC is *role-based access control* [82], which categorizes principals based on a hierarchy of *roles* and associates access rights to principals with respect to that hierarchy.

Multi-level security (MLS) [15] was among the first instantiations of MAC; it allows accesses based on information flow. MLS for confidentiality associates objects with *labels* that represent a *confidentiality level* (e.g., *top secret* (TS), *secret* (S), *confidential* (C), *unclassified* (U)). Each principal also is assigned a label representing the highest confidential level this principal is trusted to read. With MLS, principals may not read objects associated with a higher label than their own. MLS also ensures that principals do not write to objects associated with a lower label than their own. This restriction on writes prevents information from flowing to objects with lower confidentiality level through the actions (i.e., read and write) of principals. Dual requirements [17] can be placed by MLS to prevent low integrity information from flowing to objects of high integrity. So,

MLS restricts the flow of information between objects by restricting read and write actions that principals can perform to these objects.

Restricting information flow based only on principals' actions harms permissiveness. For example, MLS for confidentiality prohibits a principal assigned label C from writing some data to an object labeled U, even if the origin of the data is actually labeled U. So no confidential information would be leaked by writing this data. A more fine-grained approach could model the computation used to derive the data, deduce that there is no dependency on confidential information, and thus, allow the write. Information flow control is such an approach.

1.2 Information Flow Control

An *information flow policy* specifies *restrictions* on the use of data and other information derived from that data. One way that an information flow policy is different from a DAC policy is that DAC restricts how data can be accessed but imposes no restriction on accesses to derived information. In fact, an information flow policy can be considered as a synthesis of DAC policies for initial and derived data. An information flow policy can be set either by the owner of the associated data or the institution that maintains that data. Thus, an information flow policy is not necessarily a MAC policy, either.

1.2.1 Labels

Labels are employed to express information flow policies. Traditionally [36], labels in a set L form a lattice $\langle L, \sqsubseteq \rangle$ with join operation $\sqcup$. Partial order $\sqsubseteq$ on labels defines allowed flow of information between tagged pieces of data. If label ℓ' is *at least as restrictive as* ℓ, meaning $\ell \sqsubseteq \ell'$ holds, then information is allowed to flow from data tagged with label ℓ to data tagged with ℓ'. For example, $\mathsf{L} \sqsubseteq \mathsf{H}$ defines that data tagged with low confidentiality L is allowed to flow to data tagged with high confidentiality H. Join $\ell \sqcup \ell'$ is the label for the combination of two pieces of data tagged with ℓ and ℓ', correspondingly.

A system satisfies *noninterference* (NI) [44] for a given lattice of labels, if executions of that system cause only allowed flows of information (with respect to that lattice). In particular, a system satisfies NI, if changing initial values tagged with $\ell \in L$ does not cause changes in computed values tagged with $\ell' \in L$ when $\ell \not\sqsubseteq \ell'$ holds.

1.2.2 Enforcement Mechanisms

IFC has been applied on abstract system models of different granularities. It has been applied to cyberphysical systems modeled as *timed automata* (e.g., [90]). IFC has been applied to operating systems (OS) (e.g., [40], [99]) to control the flow of information that processes cause between OS objects (e.g., files, sockets, locks). It has also been applied to databases (e.g., [97]), where fields or tables are tagged with labels and the enforcement mechanism controls the information flow caused by queries. This thesis studies IFC at the level of programs.

Enforcement of information flow policies within programs usually builds on existing methods for program analysis (e.g., data and control flow analysis, type checking). Previous research has applied IFC to low-level programming languages (e.g., [10]) and high-level programming languages [78]: imperative, object-oriented, and functional. This thesis mainly considers a simple **while**-language (i.e., imperative). We build on an approach pioneered by Volpano et al. [93], where labels are types associated with variables. Labels associated with variables are either fixed during analysis or they may change, in which case they are called *flow-sensitive* [52]. An enforcement mechanism that supports flow-sensitive labels is itself called flow-sensitive.

Each command in a program is analyzed to ensure that its executions cause only allowed flows with respect to a lattice of labels. An assignment "$x := e$" causes an *explicit* flow from expression e to variable x, because changing the value of e leads to changes in the value of x. For such a flow to be allowed, the label of x should be at least as restrictive as the label on the value of e.

Command "$\mathbf{if}\ y > 0\ \mathbf{then}\ x := 1\ \mathbf{else}\ x := 2\ \mathbf{end}$" causes an *implicit* flow from y to x. This is because the value of x depends on the value of y, even though there is no explicit assignment of y to x in that command. For this implicit flow to be allowed, the label of x should be at least as restrictive as the label of y.

IFC is an undecidable problem [78]—it is impossible to build an enforcement mechanism that accepts exactly those program executions that cause only allowed flows. This is because the halting problem, which is undecidable, can be reduced to IFC. Consider the following command:

$$l := 0;\ \mathbf{if}\ h > 0\ \mathbf{then}\ C; l := 2\ \mathbf{else\ skip\ end}$$

where variable l is tagged with L, variable h is tagged with H and $\mathsf{L} \sqsubset \mathsf{H}$ holds

(i.e., $\mathsf{L} \sqsubseteq \mathsf{H}$ and $\mathsf{H} \not\sqsubseteq \mathsf{L}$). If an enforcement mechanism could precisely decide whether this command satisfies NI, then this mechanism could decide whether command C terminates (if the command satisfies NI, then C does not terminate, but if the command does not satisfy NI, then C terminates), which is impossible.

Static Enforcement

An enforcement mechanism for IFC is called *static* when its actions (e.g., checking fixed labels, deducing flow-sensitive labels) do not depend on run-time information (e.g., values that variables store during program execution). So the program can be analyzed before execution. If the program successfully passes the analysis, then all executions of this program are guaranteed to cause only allowed flows (with respect to a given lattice of labels).

A static enforcement mechanism does not add run-time overhead but it might reject programs whose executions cause only allowed flows. Consider the following command:

$$\textbf{if } f(l) \neq g(l) \textbf{ then } l := h \textbf{ else } l := 0 \textbf{ end} \tag{1.1}$$

where f and g are procedures implementing the exact same function, constant 0 is tagged with L, variable l is tagged with L, variable h is tagged with H and $\mathsf{L} \sqsubset \mathsf{H}$ holds. Command (1.1) causes only allowed flows, because $f(l) \neq g(l)$ is always *false* and assignment $l := 0$ causes an allowed flow. However, a static enforcement mechanism cannot in general decide whether f and g are equivalent. So, it will analyze both branches of (1.1), as if both branches could be executed. Because assignment $l := h$ causes an illicit flow, the entire command (1.1) will be rejected by the analysis.

A static enforcement mechanism also might reject an entire program, even though only a subset of program executions cause illicit flows. Consider command

$$\textbf{if } l \leq 2 \textbf{ then } l := h \textbf{ else } l := 0 \textbf{ end}. \tag{1.2}$$

Executions where $l \leq 2$ holds do cause illicit flows, due to assignment $l := h$. But executions where $l > 2$ holds cause only allowed flows. Similar to (1.1), command (1.2) will be rejected. Thus, no execution of (1.2) will be allowed, including those where $l > 2$ holds.

Dynamic Enforcement

Dynamic enforcement improves permissiveness over static enforcement through the use of run-time information. Actions (e.g., check labels, deduce flow-sensitive labels) of a dynamic enforcement mechanism are performed (at least partially) during program execution, introducing run-time overhead.

We give examples of how a dynamic mechanism could enhance permissiveness. For command (1.1), a dynamic mechanism would deduce that there is no illicit flow, since $l := h$ is never reached (because $f(l) \neq g(l)$ is always *false*). So, all executions will be allowed, because the only assignment that actually occurs is $l := 0$. For command (1.2), if execution reaches $l := h$, then a dynamic enforcement mechanism might *block* that execution before performing assignment $l := h$ but allow all other executions to complete. So, for examples (1.1) and (1.2) a dynamic mechanism can be more permissive than a static mechanism.

Flow-sensitive dynamic mechanisms can enhance permissiveness even fur-

ther. Consider command

$$x := h;\ x := 0;\ l := x \tag{1.3}$$

whose executions cause only allowed flows. A dynamic mechanism that tags x with fixed label H, allows the execution of the first two assignments in (1.3), but blocks assignment $l := x$ because $\mathsf{H} \not\sqsubseteq \mathsf{L}$ holds. Instead, a flow-sensitive dynamic mechanism would tag x with flow-sensitive label H after $x := h$, but then tag x with flow-sensitive label L after $x := 0$. Now, assignment $l := x$ could be allowed to execute. Thus, using flow sensitive labels enhanced permissiveness.

A *purely dynamic* flow-sensitive mechanism analyzes only code that is actually executed. Russo and Sabelfeld [77] showed that if a purely dynamic flow-sensitive mechanism is at least as permissive as the Hunt and Sands classical flow-sensitive static mechanism [52], then that dynamic mechanism is not sound.[1] Here is an example where a purely dynamic flow-sensitive mechanism D is not sound.

$$x := 0;\ \textbf{if } h > 0 \textbf{ then } x := 2 \textbf{ else skip end} \tag{1.4}$$

where constants 0 and 2 are tagged with L, variable h is tagged with label H and x with a flow-sensitive label. When $x := 0$ executes, D tags x with flow-sensitive label L. If $h > 0$ holds, then $x := 2$ executes and D tags x with flow-sensitive label H, thereby capturing the implicit flow from h to x. If $h \not> 0$ holds, no assignment to x is executed, and thus neither the value nor the label of x changes. So, x stores 0 and is tagged with L at termination if $h \not> 0$ holds. However, the value of h has flowed to the final value of x, because knowing

[1] This result does not apply to *secure multi-execution* (SME) [38] , which is a purely dynamic flow-sensitive enforcement mechanism that enforces information flow labels by simultaneously executing the same program as many times as the number of labels.

that $x = 0$ at terminations implies that $h \not> 0$ holds. So, x should always have been tagged with H at termination.

Purely dynamic mechanisms (e.g., [7, 98]) proposed in the past maintain soundness at the expense of permissiveness. For example, such a mechanism would decide to block assignment $x := 2$ in (1.4) when $h > 0$ holds. In this way, the final value of x will always be 0 (independent of h) and its final label will always be L, which is sound.

Allowing a dynamic mechanism to analyze unexecuted code can maintain soundness and permissiveness. Consider, again, (1.4). If $h \not> 0$ holds, then such a mechanism could analyze untaken branch $x := 2$ and tag x with flow-sensitive label H (to capture implicit flow from h to x). If $h > 0$ holds, then x would again be tagged with flow-sensitive label H after assignment $x := 2$ executes. So, such a mechanism always tags x with H, which would not block any of the executions. However, compared to purely dynamic mechanisms, such a mechanism cannot handle programs involving dynamically loaded code, and it introduces additional run-time overhead.

In this thesis, we explore both static and dynamic enforcement mechanisms that use fixed and flow-sensitive expressive labels. The enforcement mechanisms we propose are extensions of mechanisms proposed in the past.

1.3 RIF Labels

Specifying allowed flows of information based only on restrictiveness relation $\sqsubseteq$ on labels and ignoring the semantics of operations that cause these flows can

be inadequate. A flow from ℓ to ℓ' (i.e., from data tagged with ℓ to data tagged with ℓ') caused by an operation may be acceptable, even if $\ell \sqsubseteq \ell'$ does not hold; a flow from ℓ to ℓ' caused by an operation may not be acceptable, even if $\ell \sqsubseteq \ell'$ holds. Some examples illustrate.

Consider first an information flow policy on salaries $h_1, \ldots, h_n$: salaries are highly confidential, but the result of taking their average has low confidentiality. Given lattice $\langle \{\mathsf{L}, \mathsf{H}\}, \sqsubseteq \rangle$ with $\mathsf{L} \sqsubset \mathsf{H}$, salaries should be tagged with H, otherwise the policy is violated. Consider now command

$$l := avg(h_1, \ldots, h_n) \tag{1.5}$$

that computes the average of salaries $h_1, \ldots, h_n$. According to the above policy on salaries, the average result l can be tagged with L. But now the flow of information caused by (1.5) is not allowed by restrictiveness relation $\sqsubseteq$, because $\mathsf{H} \sqsubseteq \mathsf{L}$ does not hold. So, a flow allowed by the policy on salaries is not allowed by $\sqsubseteq$, and thus permissiveness is harmed. In this example, operation avg is said to trigger a *declassification* [81], because it causes a desired flow towards lower confidentiality.

Soundness might also be harmed when flows are allowed based only on $\sqsubseteq$. As an example, consider an information flow policy on the guest list $guests$ of a hotel: $guests$ has low confidentiality, but due to privacy concerns, the final assignment of guests to rooms is deemed highly confidential. Also, the room list $rooms$ of that hotel has low confidentiality. Given again lattice $\langle \{\mathsf{L}, \mathsf{H}\}, \sqsubseteq \rangle$, $guests$ and $rooms$ can then be tagged with L, otherwise the labels would be less permissive than the policy. Consider command

$$agn := assign(guests, rooms) \tag{1.6}$$

that assigns guests to rooms. If agn is tagged with L, then the flow caused by (1.6) is allowed by $\sqsubseteq$, but violates the desired policy on $guests$. In this example, operation $assign$ is said to trigger a *classification* because it forces a flow towards strictly higher confidentiality.

The difficulties illustrated by the above two examples can be addressed by increasing the expressiveness of labels to more completely describe a desired information flow policies. Such expressive labels would be able to specify how restrictions on derived values depend on operations. So, such labels could specify arbitrary *reclassifications*, since restrictions on derived values may be more, fewer, or incomparable to restrictions on inputs. Labels proposed by previous approaches [23, 22, 69, 68, 57, 76, 88, 50, 63, 29, 74, 75, 81] do not explicitly specify arbitrary reclassifications caused by operations.

In Chapter 2, we propose *reactive information flow* (RIF) labels that can specify how restrictions on derived values depend on the applied operations.[2] We also propose *piecewise noninterference* (PWNI) as an extension to NI to prescribe allowed flows under arbitrary reclassifications. A type system is given for static enforcement of PWNI in programs that associate *checkable* classes of RIF labels with variables. Two checkable classes of RIF labels are described. RIF automata (Chapter 3) are general-purpose and based on finite-state automata; κ-labels (Chapter 4) concern confidentiality in programs that use cryptographic operations and assume the Dolev-Yao model of attacks. We actually implemented an enforcement mechanism for RIF automata in Java.[3]

[2]This is joint work with Fred B. Schneider.
[3]This is joint work with Owen Arden, Andrew C. Myers, and Fred B. Schneider.

1.4 Label Chains

Labels themselves are values that could encode sensitive information.[4] Consider a dynamic enforcement mechanism that tags variables with flow-sensitive labels. Sensitive input information might influence which assignments are executed and, consequently, determine how and when the flow-sensitive label that is tagging a variable changes.

That means flow-sensitive labels can depend on sensitive information. Inspecting or directly observing flow-sensitive labels then might leak sensitive information [51]. Consider a program that mixes flow-sensitive labels and fixed labels:

$$\textbf{if } m > 0 \textbf{ then } w := h \textbf{ else } w := l \textbf{ end} \tag{1.7}$$

Suppose w is tagged with a flow-sensitive label, but the other variables are tagged with fixed labels: l is tagged with fixed label L (i.e., low), m with M (i.e., medium), and h with H (i.e., high), where $\mathsf{L} \sqsubset \mathsf{M} \sqsubset \mathsf{H}$ holds.

(i) If $m > 0$ holds, then when (1.7) terminates, w should be tagged with flow-sensitive label H, because H is at least as restrictive as the label H that tags h and the label M that tags m.

(ii) If $m \not> 0$ holds, then when (1.7) terminates w should be tagged with flow-sensitive label M, because M is at least as restrictive as the labels that tag l and m.

So, when (1.7) terminates, the flow-sensitive label tagging w depends on whether $m > 0$ holds. That sensitive information about m leaks to observers

[4] Here, *sensitivity* refers to *confidentiality level*.

that can learn that label.

Blocking (e.g., [2, 31, 67, 83]) an execution based on flow-sensitive labels might leak sensitive information, too. Consider (1.7) extended with two assignments:

$$\textbf{if } m > 0 \textbf{ then } w := h \textbf{ else } w := l \textbf{ end}; \; m := w; \; l := 1 \tag{1.8}$$

(i) If $m > 0$ holds, then $m := w$ should be blocked to prevent information tagged H and stored in w from flowing to m; assignment $l := 1$ is not reached.

(ii) If $m \not> 0$ holds, then $m := w$ does not need to be blocked (because w stores information tagged M). Assignment $l := 1$ could execute.

Depending on whether $m := w$ is blocked, principals monitoring variable l (which is tagged L) either do or do not observe value 1 being assigned to l. The decision to block $m := w$ depends on the flow-sensitive label of w, which depends on sensitive information $m > 0$. So $m > 0$ is leaked[5] to principals monitoring variable l.

In order to prevent such leaks, *metalabels* (e.g., [12]) can be introduced to represent the sensitivity of information encoded in labels. For example, the metalabel for w in (1.8) would be M, corresponding to the sensitivity of information $m > 0$ encoded in the flow-sensitive label tagging w. Only principals authorized to read information allowed by the metalabel (i.e., M) would be allowed to observe the label of w. The metalabel that tags w would also capture the sensitivity of the decision to execute $m := w$ and reach $l := 1$. To prevent the implicit flow

[5] In fact, an arbitrary number of bits can be leaked through blocking executions [5].

of that information (which is tagged with H) to variable l (tagged L), assignment $l := 1$ must never be executed.

Since metalabels are flow-sensitive, they too could encode sensitive information that can be leaked to observers. It is tempting to employ meta-meta labels to prevent those leaks. However, flow-sensitive meta-meta labels might then leak. We seem to need a label chain associated with each variable: a label ℓ_1, metalabel ℓ_2, meta-meta label ℓ_3, etc.

In Chapter 5, a new family of *enforcers*—k-*Enf*, for $2 \leq k \leq \infty$—is derived to leverage arbitrarily long *label chains*, where each label defines the sensitivity of its predecessor.[6] These enforcers satisfy *Block-safe Noninterference* (BNI), which proscribes leaks from observing variables, label chains, and blocked executions. In Chapter 6, theorems characterize where longer label chains improve permissiveness of dynamic enforcement mechanisms that satisfy BNI. These theorems depend on semantic attributes—k-*precise*, k-*varying*, and k-*dependent*—of such mechanisms as well as on initialization, threat model, and size of lattice. Previous approaches [100, 64, 27] can express label chains, but they have not studied the relation of label-chain length with permissiveness. Finally, Chapter 7 concludes this thesis.

[6]This is joint work with Fred B. Schneider, Josée Desharnais, Nadia Tawbi, and Andrew Bedford.

CHAPTER 2

RIF LABELS

The most general form of *flow-derived restrictions* would assign restrictions to the output of an operation $op(x_1, x_2, \ldots, x_n)$ according to operator op, its inputs $x_1, x_2, \ldots, x_n$, and the restrictions associated with those inputs. Information flow control is a well known example of flow-derived restrictions. In Denning's initial work [36] and in much that has followed—both for confidentiality and integrity—the set of restrictions assigned to the output of an operation is the union of the restrictions associated with its inputs. But in ignoring the operator and the values of its inputs, that approach can be too conservative.

The output of operation $op(x_1, x_2, \ldots, x_n)$ might warrant fewer restrictions, additional restrictions, or an incomparable set of restrictions than are being associated with its inputs.

- With an operation that computes the winner of an election, the inputs are votes and the output is the majority. Each input is secret to the principal casting that vote, whereas the output ought to be readable by any principal. So the output is being associated with fewer restrictions than the inputs.
- A conference-management system identifies reviewers for each submission. The inputs—a list of reviewers and a list submissions—can be read by the entire program committee. Conflicts of interest, however, dictate that only a subset of the program committee learn which reviewers are assigned to any given paper. So outputs are associated with more stringent restrictions than inputs.

- A value produced by symmetric-key encryption $Enc(\mathcal{E}, K)$ involving key K may be read by any principal except one that both can read K and is restricted from reading $\mathcal{E}$. So $Enc(\mathcal{E}, K)$ would typically have fewer restrictions than inputs $\mathcal{E}$ and K. Decryption $Dec(Enc(\mathcal{E}, K), K)$ produces a value that has the same restrictions as $\mathcal{E}$ and, thus, could have increased restrictions over those assigned to its inputs $Enc(\mathcal{E}, K)$ and K. The restrictions assigned to $Dec(Enc(\mathcal{E}, K), K')$ when $K \neq K'$ holds depend on whether $Enc(\cdot, K)$ and $Dec(\cdot, K)$ are inverses of each other and on whether any principal knows both K and K'.

Previous work on information flow control—information flow locks [23, 22], expressions for declassification (for confidentiality) and endorsement (for integrity) [69, 68], and capability-based mechanisms for downgrading security policies [57, 76, 88]—has not provided means for arbitrary changes in restrictions to be linked with specific operations. Other approaches (e.g., [50, 63, 29, 74, 75, 81]) allow changes but only between two kinds of restrictions.

Reactive information flow (RIF) labels, which we introduce in §2.1, seek to address these limitations. Piecewise noninterference (PWNI) in §2.2 then extends classical noninterference in a way that handles changes to restrictions that RIF labels support. A type system is given in §2.3 to support static enforcement of PWNI for certain classes of RIF labels.

2.1 Formalization of RIF labels

Restrictions. Restrictions specified by the RIF label for a value are taken from a partially ordered set $\langle R, \sqsubseteq_R \rangle$. For confidentiality, an element of R might identify which principals are allowed to read some value, either by enumerating that set of principals or giving the name (e.g., public, secret, etc.) for such a set; for integrity, it might identify the set of principals allowed to write that value. Other kinds of restrictions might be specified using elements in R that are a set of programs, the name for a task, or the name of a set of tasks.

Partial order $r \sqsubseteq_R r'$ characterizes whether satisfying restrictions $r' \in R$ implies that restrictions $r \in R$ are satisfied too, so r' is *at least as strong* as r or, equivalently, r is *at least as weak* as r'. When elements of R denote sets of principals, then for confidentiality of some value v, we would define $r \sqsubseteq_R r'$ to hold if and only if $r \supseteq r'$ holds—if r' allows a principal to read v then so does r, but r might also allow other principals to read v, too. And, for integrity, we would define $r \sqsubseteq_R r'$ to hold if and only if $r \subseteq r'$ holds, because r' requires putting trust in writes by more principals than r does, and trust can be justified only when behavior is restricted.

Reclassifiers and RIF Labels. A set $\mathcal{F}$ of *reclassifiers* is used to abstract how operations change restrictions on inputs.[1] Each element of $\mathcal{F}$ might correspond to a single operation or to a set of operations. A RIF label maps sequences of reclassifiers (which abstractly describe the provenence of a value) to elements of some given partially ordered set $\langle R, \sqsubseteq_R \rangle$ of restrictions. This is formalized for a set Λ of RIF labels by giving a set $\mathcal{F}$ of reclassifiers and two functions $\mathcal{R}$

[1]The term *reclassify* was used in Denning's thesis [36] to name an operation that changes the restrictions imposed on objects.

	$\mathcal{R}(\lambda) \subset \mathcal{R}(\mathcal{T}(\lambda, \mathsf{f}))$	$\mathcal{R}(\lambda) \supset \mathcal{R}(\mathcal{T}(\lambda, \mathsf{f}))$
Confidentiality	*declassification*	*classification*
Integrity	*deprecation*	*endorsement*

Figure 2.1: Terminology for reclassifications

and $\mathcal{T}$.

– $\mathcal{R}$ maps $\lambda \in \Lambda$ to the restriction $r \in R$ that λ imposes:

$$\mathcal{R}\colon \Lambda \to R \tag{2.1}$$

– $\mathcal{T}$ maps $\lambda \in \Lambda$ and any reclassifier $\mathsf{f} \in \mathcal{F}$ to a RIF label that should be associated with the value produced by an operation abstracted by f:

$$\mathcal{T}\colon \Lambda \times \mathcal{F} \to \Lambda \tag{2.2}$$

$\mathcal{T}$ is extended to a sequence F of reclassifiers in the usual way, with empty sequence ϵ of reclassifiers assumed to be an element of every set $\mathcal{F}$ of reclassifiers (where it serves as an identity reclassifier).

$$\begin{aligned} \mathcal{T}(\lambda, \epsilon) &\triangleq \lambda \\ \mathcal{T}(\lambda, \mathsf{F}\mathsf{f}) &\triangleq \mathcal{T}(\mathcal{T}(\lambda, \mathsf{F}), \mathsf{f}) \end{aligned}$$

If $\lambda \neq \mathcal{T}(\lambda, \mathsf{f})$ holds, then f *triggers* a reclassification. Figure 2.1 gives some specialized terminology when $\mathcal{R}(\lambda)$ gives a set of principals that must be trusted not to divulge the value (for confidentiality) or not to have corrupted the value (for integrity).

Reclassifying Expressions. In order to associate RIF label transitions with operations, we extend the set of *ordinary expressions*: variables and terms

$op(\mathcal{E}_1, \ldots, \mathcal{E}_n)$, where op is an operator and $\mathcal{E}_i$ is an ordinary expression. For ordinary expressions $\mathcal{E}_1$, ..., $\mathcal{E}_n$,

$$[op(\mathcal{E}_1, \ldots, \mathcal{E}_n)]_{\mathsf{f}_1,\mathsf{f}_2,\ldots,\mathsf{f}_n} \tag{2.3}$$

defines a *reclassifying expression*. It specifies that reclassifier f_i identifies how the restrictions associated with the value of ordinary expression $\mathcal{E}_i$ should be changed in order to obtain restrictions associated with the value produced by $op(\mathcal{E}_1, \ldots, \mathcal{E}_n)$. The change in restrictions is effected by transition $\mathcal{T}(\lambda_i, \mathsf{f}_i)$ to RIF label λ_i being associated with the value of expression $\mathcal{E}_i$. Notation $[op(\mathcal{E}_1, \ldots, \mathcal{E}_n)]_{\mathsf{f}}$ is used as a shorthand for $[op(\mathcal{E}_1, \ldots, \mathcal{E}_n)]_{\mathsf{f},\mathsf{f},\ldots,\mathsf{f}}$, and we sometimes abbreviate $[op(\mathcal{E}_1, \ldots, \mathcal{E}_n)]_{\mathsf{f}_1,\mathsf{f}_2,\ldots,\mathsf{f}_n}$ by $[op(\mathcal{E}_1, \ldots, \mathcal{E}_n)]_{\bar{\mathsf{f}}}$ or simply $[\mathcal{E}]_{\bar{\mathsf{f}}}$ when the elided specifics are irrelevant.

When reclassifying expressions are used to compute values, then a sequence of reclassifiers offers an abstract description for the series of operations that have been applied to some value as program execution proceeds. This sequence of reclassifiers then provides a basis for determining the set of restrictions associated with computed values. For example, consider the following program.

$$w_1 := [div(x_1, x_2)]_{\mathsf{f}_1};\ y_2 := [mod(w_1, w_2)]_{\mathsf{f}_2};\ z_3 := [add(y_1, y_2, y_3)]_{\mathsf{f}_3}$$

The restrictions on the value stored in z_3 are informed (in part) by sequence $\mathsf{f}_1\mathsf{f}_2\mathsf{f}_3$ of reclassifiers applied to the restrictions on the value in x_1, because the value in x_1 *flows to* z_3 *through* $\mathsf{f}_1\mathsf{f}_2\mathsf{f}_3$: x_1 first flows to w_1 through f_1, then w_1 flows to y_2 through f_2, and finally y_2 flows to z_3 through f_3. The restrictions on the value stored in z_3 also are informed by sequence $\mathsf{f}_1\mathsf{f}_2\mathsf{f}_3$ applied to the restrictions on the value stored in x_2, sequence $\mathsf{f}_2\mathsf{f}_3$ applied to the restrictions on the value stored in w_2, and sequence f_3 applied to the values stored in y_1 and y_3.

Classes of RIF Labels. We require that $\langle \Lambda, \sqcup, \sqsubseteq \rangle$ form a lattice (which might be infinite), with *join* operator $\sqcup$ used for combining RIF labels; *restrictiveness* relation $\sqsubseteq$ then specifies whether one RIF label is *at least as restrictive as* another. Since, by definition, $\lambda \sqsubseteq \lambda \sqcup \lambda'$ and $\lambda' \sqsubseteq \lambda \sqcup \lambda'$ holds in a lattice, a combination of RIF labels is at least as restrictive as any of its constituents. We posit that Λ includes elements $\bot$ and $\top$ such that for all $\lambda \in \Lambda$: $\bot \sqsubseteq \lambda$ and $\lambda \sqsubseteq \top$ hold.

The specific definition of $\sqcup$ depends on the elements of Λ. The definition of restrictivenesss relation $\sqsubseteq$ uses $\mathcal{F}^*$ to denote the set of finite sequences of reclassifiers in $\mathcal{F}$:

$$\lambda \sqsubseteq \lambda' \quad \triangleq \quad (\forall \mathsf{F} \in \mathcal{F}^*\colon\ \mathcal{R}(\mathcal{T}(\lambda, \mathsf{F})) \sqsubseteq_R \mathcal{R}(\mathcal{T}(\lambda', \mathsf{F}))) \tag{2.4}$$

So if $\lambda \sqsubseteq \lambda'$ holds, then

- current restrictions $\mathcal{R}(\lambda')$ specified by λ' are at least as strong as what λ imposes because, by definition, $\epsilon \in \mathcal{F}^*$, $\mathcal{R}(\mathcal{T}(\lambda, \epsilon)) = \mathcal{R}(\lambda)$ and $\mathcal{R}(\mathcal{T}(\lambda', \epsilon)) = \mathcal{R}(\lambda')$ hold, so (2.4) implies $\mathcal{R}(\lambda) \sqsubseteq_R \mathcal{R}(\lambda')$, and
- restrictions λ' imposes for any derived value are at least as strong as what λ imposes—if v flows to w then there is a sequence $\mathsf{F} \in \mathcal{F}*$ that v flows through, and (2.4) requires that $\mathcal{R}(\mathcal{T}(\lambda, \mathsf{F})) \sqsubseteq_R \mathcal{R}(\mathcal{T}(\lambda', \mathsf{F}))$ hold.

The quantification over all sequences $\mathsf{F} \in \mathcal{F}^*$ in (2.4) means that this definition for $\lambda \sqsubseteq \lambda'$ is conservative, since it could be imposing conditions on sequences F of reclassifiers that never arise in a program execution.

A *class* of RIF labels is formed by putting the pieces together:

$$\langle \langle R, \sqsubseteq_R \rangle, \langle \Lambda, \sqcup, \sqsubseteq \rangle, \mathcal{F}, \mathcal{R}, \mathcal{T} \rangle$$

The definition of such a class is silent about the existence of algorithms for computing $\sqcup$ or for deciding $\sqsubseteq$. We say that a class of RIF labels is *checkable* if and only if:

(i) There exists an algorithm for computing $\lambda \sqcup \lambda'$, for all $\lambda, \lambda' \in \Lambda$.

(ii) There exists a sound (no false positives) if not complete (false negatives possible) decision procedure for determining whether $\lambda \sqsubseteq \lambda'$ holds, for all $\lambda, \lambda' \in \Lambda$.

Programs that use the type system we give in §2.3, which has RIF labels as types, can be statically checked if the RIF labels are from a checkable class.

2.2 A Security Policy for RIF Labels

We assume that every command is deterministic and changes the values of variables in memory. Therefore, execution of a command $\mathcal{C}_1$ that is started in a memory $\mathcal{M}_1$ and terminates in a memory $\mathcal{M}_N$ can be described by giving a finite *trace*

$$\langle \mathcal{C}_1, \mathcal{M}_1 \rangle \rightarrow \langle \mathcal{C}_2, \mathcal{M}_2 \rangle \rightarrow \cdots \rightarrow \langle \bullet, \mathcal{M}_N \rangle \tag{2.5}$$

where each *state* $\langle \mathcal{C}_i, \mathcal{M}_i \rangle$ gives a command $\mathcal{C}_i$ and a memory $\mathcal{M}_i$. Although $\bullet$ is not considered a command, we use $\bullet$ in the final state of a trace to signify that further execution is not possible. Thus the trace for a command $\mathcal{C}$ that terminates will include at least two states: one having $\mathcal{C}$ as the command and one having $\bullet$.

Memories are represented by functions that map all variables x appearing in any command to values $\mathcal{M}(x)$. These functions are then extended as usual for

mapping ordinary expressions and reclassifying expressions.

$$\begin{aligned} \mathcal{M}(op(\mathcal{E}_1, \ldots, \mathcal{E}_n)) &\triangleq op(\mathcal{M}(\mathcal{E}_1), \ldots, \mathcal{M}(\mathcal{E}_n)) \\ \mathcal{M}([\mathcal{E}]_{\mathsf{f}_1,\mathsf{f}_2,\ldots,\mathsf{f}_n}) &\triangleq \mathcal{M}(\mathcal{E}) \end{aligned}$$

An operational semantics for command execution thus defines a partial function Υ, where $\Upsilon(\mathcal{C}, \mathcal{M})$ equals the finite trace corresponding to an execution of $\mathcal{C}$ that terminates when started in memory $\mathcal{M}$, and $\Upsilon(\mathcal{C}, \mathcal{M})$ is undefined if that execution does not terminate.

Observations as a Threat Model. Given is a fixed mapping Γ that associates a label $\Gamma(x) \in \Lambda$ with each variable x. Γ is extended to handle ordinary expressions in the usual way

$$\Gamma(op(\mathcal{E}_1, \ldots, \mathcal{E}_n)) \triangleq \Gamma(\mathcal{E}_1) \sqcup \ldots \sqcup \Gamma(\mathcal{E}_n) \tag{2.6}$$

so we have $\Gamma(op(x_1, \ldots, x_n)) = \Gamma(x_1) \sqcup \ldots \sqcup \Gamma(x_n)$, as expected. The RIF label associated with a reclassifying expression is expected to combine RIF labels obtained after the indicated transitions have been performed:

$$\Gamma([op(\mathcal{E}_1, \ldots, \mathcal{E}_n)]_{\mathsf{f}_1,\ldots,\mathsf{f}_n}) \triangleq \mathcal{T}(\Gamma(\mathcal{E}_1), \mathsf{f}_1) \sqcup \ldots \sqcup \mathcal{T}(\Gamma(\mathcal{E}_n), \mathsf{f}_n) \tag{2.7}$$

Γ induces equivalence classes comprising memories that are indistinguishable to a principal assigned a label $\lambda \in \Lambda$:

$$\mathcal{M} =_\lambda \mathcal{M}' \triangleq (\forall x\colon\ \Gamma(x) \sqsubseteq \lambda \Rightarrow \mathcal{M}(x) = \mathcal{M}'(x))$$

So when a *transition* $\langle \mathcal{C}_i, \mathcal{M}_i \rangle \rightarrow \langle \mathcal{C}_{i+1}, \mathcal{M}_{i+1} \rangle$ occurs during execution, only certain memory changes are visible to a principal assigned a label $\lambda \in \Lambda$. These changes are described by an *observation*

$$\mathcal{M}_{i+1} \ominus_\lambda \mathcal{M}_i \triangleq \{\langle x, \mathcal{M}_{i+1}(x) \rangle \mid \mathcal{M}_i(x) \neq \mathcal{M}_{i+1}(x) \wedge \Gamma(x) \sqsubseteq \lambda\}$$

which is a set giving the new value for each variable that was (i) changed by the transition and (ii) has a label at most λ. It is easy to show:

$$\mathcal{M} =_\lambda \mathcal{M}' \;\Rightarrow\; (\mathcal{M}' \ominus_\lambda \mathcal{M} = \emptyset)$$
$$\lambda \sqsubseteq \lambda' \Rightarrow\; (\mathcal{M}' \ominus_\lambda \mathcal{M}) \subseteq (\mathcal{M}' \ominus_{\lambda'} \mathcal{M})$$

For principals assigned label λ, program execution described by a trace τ results in a sequence $\tau|_\lambda$

$$\theta_0 \rightharpoonup \theta_1 \rightharpoonup \ldots \rightharpoonup \theta_n$$

of non-empty observations $\mathcal{M}'_{i+1} \ominus_\lambda \mathcal{M}_i$ that are derived from the successive transitions $\langle \mathcal{C}_i, \mathcal{M}_i \rangle \rightarrow \langle \mathcal{C}_{i+1}, \mathcal{M}_{i+1} \rangle$ in τ.[2] Traces τ and τ' are indistinguishable to principals with label λ if $\tau|_\lambda = \tau'|_\lambda$ holds, which induces an equivalence relation on traces:

$$\tau =_\lambda \tau' \;\triangleq\; \tau|_\lambda = \tau'|_\lambda$$

Sequences $\tau|_\lambda$ of observations are the basis for our[3] *threat model*. It stipulates that a principal p with label λ is notified of every change in the value of any variable x satisfying $\Gamma(x) \sqsubseteq \lambda$. Such a threat model is well suited for analyzing systems in which principals are co-resident and able to detect changes (subject to restrictions defined by labels) being made to shared memory or other resources.

[2] A formal definition of $\tau|_\lambda$ is thus given by the following.

$$\tau|_\lambda \;\triangleq\; \begin{cases} \epsilon & \text{if } \tau = \epsilon \text{ or } \tau = \langle \mathcal{C}, \mathcal{M} \rangle \\ \epsilon & \text{if } \tau = \langle \mathcal{C}, \mathcal{M} \rangle \rightarrow \langle \mathcal{C}', \mathcal{M}' \rangle \;\wedge\; \mathcal{M}' \ominus_\lambda \mathcal{M} = \emptyset \\ \mathcal{M}' \ominus_\lambda \mathcal{M} & \text{if } \tau = \langle \mathcal{C}, \mathcal{M} \rangle \rightarrow \langle \mathcal{C}', \mathcal{M}' \rangle \;\wedge\; \mathcal{M}' \ominus_\lambda \mathcal{M} \neq \emptyset \\ (\langle \mathcal{C}', \mathcal{M}' \rangle \rightarrow \tau')|_\lambda & \text{if } \tau = \langle \mathcal{C}, \mathcal{M} \rangle \rightarrow \langle \mathcal{C}', \mathcal{M}' \rangle \rightarrow \tau' \wedge\; \mathcal{M}' \ominus_\lambda \mathcal{M} = \emptyset \\ \mathcal{M}' \ominus_\lambda \mathcal{M} \rightharpoonup (\langle \mathcal{C}', \mathcal{M}' \rangle \rightarrow \tau')|_\lambda & \text{otherwise} \end{cases}$$

[3] A similar threat model is found in Askarov and Sabelfeld [3].

Reclassification. Each transition $\langle \mathcal{C}_i, \mathcal{M}_i \rangle \rightarrow \langle \mathcal{C}_{i+1}, \mathcal{M}_{i+1} \rangle$ in a trace involves evaluating some (possibly empty) set of expressions. The operational semantics of commands determines what those expressions are and how their values are computed. We therefore assume that the operational semantics for command includes a function $\Delta(\mathcal{C}_i)$ that gives the set of reclassifying expressions that a command $\mathcal{C}_i$ evaluates in making its transition.

For example, with a simple imperative programming language and ordinary expressions $\mathcal{E}$ and $\mathcal{E}'$ we might expect to have

$$\Delta(x := \mathcal{E}) = \emptyset \tag{2.8}$$

$$\Delta(x := [\mathcal{E}]_{\bar{\mathsf{f}}}) = \{[\mathcal{E}]_{\bar{\mathsf{f}}}\} \tag{2.9}$$

$$\Delta(\textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } x := [\mathcal{E}']_{\bar{\mathsf{f}}}) = \{[\mathcal{E}]_{\bar{\mathsf{f}}}\} \tag{2.10}$$

if we are assuming that assignments are executed as a single transition but that an **if** statement involves a first transition to evaluate its Boolean guard followed by other transitions for the **then** (or **else**) parts.

Piecewise Noninterference. In the absence of reclassifications, an *illicit λ-flow* is present if executing a command in states having different values of a variable with label λ' satisfying $\lambda' \not\sqsubseteq \lambda$ will result in differences in updates to any variable with label λ or less restrictive. Noninterference [44] is a widely studied security policy that prohibits illicit λ-flows for all labels λ. It is the basis for piecewise noninterference.

To formalize an extension to noninterference that accommodates reclassification under our threat model, we introduce some notation. Given τ a non-empty finite trace or subtrace,

$$\langle \mathcal{C}_1, \mathcal{M}_1 \rangle \rightarrow \langle \mathcal{C}_2, \mathcal{M}_2 \rangle \rightarrow \cdots \rightarrow \langle \mathcal{C}_N, \mathcal{M}_N \rangle$$

and indices i and j satisfying $1 \leq i \leq j \leq N$, define

$$\begin{aligned}
\tau[i..] &\triangleq \langle \mathcal{C}_i, \mathcal{M}_i \rangle \rightarrow \cdots \\
\tau[i..j] &\triangleq \langle \mathcal{C}_i, \mathcal{M}_i \rangle \rightarrow \cdots \rightarrow \langle \mathcal{C}_j, \mathcal{M}_j \rangle \\
\tau[i] &\triangleq \langle \mathcal{C}_i, \mathcal{M}_i \rangle \\
\tau[i].\mathcal{C} &\triangleq \mathcal{C}_i \\
\tau[i].\mathcal{M} &\triangleq \mathcal{M}_i \\
|\tau| &\triangleq N
\end{aligned}$$

although we write $\tau.\mathcal{C}$ as an abbreviation for $\tau[1].\mathcal{C}$ and write $\tau.\mathcal{M}$ for $\tau[1].\mathcal{M}$. If $1 \leq i \leq j \leq N$ does not hold then it is convenient to define $\tau[i..]$, $\tau[i..j]$, $\tau[i]$, $\tau[i].\mathcal{C}$, and $\tau[i].\mathcal{M}$ as being be equivalent to ϵ.

We start by formalizing prohibition of illicit λ-flows for commands that do not contain reclassifying expressions. By definition, an illicit λ-flow is not present if executing such a command $\widehat{\mathcal{C}}$ in states $\mathcal{M}$ and $\mathcal{M}'$ satisfying $\mathcal{M} =_\lambda \mathcal{M}'$ does not result in differences in updates to a variable that has a label λ or less restrictive.

$$(\forall \mathcal{M}, \mathcal{M}', \tau, \tau' \colon \quad \tau = \Upsilon(\widehat{\mathcal{C}}, \mathcal{M}) \ \wedge \ \tau' = \Upsilon(\widehat{\mathcal{C}}, \mathcal{M}') \ \ \Rightarrow \ \ NI(\lambda, \tau, \tau'))$$

where

$$NI(\lambda, \tau, \tau') \quad \triangleq \quad \tau.\mathcal{M} =_\lambda \tau'.\mathcal{M} \ \ \Rightarrow \ \ \tau =_\lambda \tau'$$

So we characterize the absence of illicit λ-flows for all labels λ by:

$$(\forall \lambda, \mathcal{M}, \mathcal{M}', \tau, \tau' \colon \ \tau = \Upsilon(\widehat{\mathcal{C}}, \mathcal{M}) \ \wedge \ \tau' = \Upsilon(\widehat{\mathcal{C}}, \mathcal{M}') \ \ \Rightarrow \ \ NI(\lambda, \tau, \tau')) \qquad (2.11)$$

Notice, if $NI(\lambda, \tau, \tau')$ is false then τ and τ' are evidence of an illicit λ-flow.

Handling Downgrades. A reclassifying expression $[\mathcal{E}]_{\bar{\mathsf{f}}}$ is considered to perform a λ-*downgrade* if $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq \lambda$ and $\Gamma(\mathcal{E}) \not\sqsubseteq \lambda$ hold. By definition, assigning the value of $[\mathcal{E}]_{\bar{\mathsf{f}}}$ to a variable having label λ does not constitute an illicit λ-flow (whereas assigning $\mathcal{E}$ would). So λ-downgrades eliminate certain illicit λ-flows by fiat.

A programming language that supports RIF labels will provide syntax that allows programmers to specify λ-downgrades by using reclassifying expressions. To define our security policy for such programs, we stiplulate that the language semantics include a function[4] $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ that satisfies:

$$\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i) \;\subseteq\; \{\mathcal{E} \mid [\mathcal{E}]_{\bar{\mathsf{f}}} \in \Delta(\mathcal{C}_i) \;\wedge\; \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq \lambda \;\wedge\; \Gamma(\mathcal{E}) \not\sqsubseteq \lambda\}$$

So $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ contains some of the expressions that are evaluated when $\mathcal{C}_i$ executes and that satisfy the definition of a λ-downgrade.

A simple generalization of (2.11) accomodates λ-downgrades in $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$. $NI(\lambda, \tau, \tau')$ was defined in terms of the indistinguishability of updates to any variable x satisfying $\Gamma(x) \sqsubseteq \lambda$. But differences in updates to x that arise when expressions in $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ have different values are, by definition, not illict λ-flows. Therefore, initial memory pairs $\mathcal{M}$ and $\mathcal{M}'$ that cause expressions in $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \widehat{\mathcal{C}})$ to have different values should be ignored in checking for indistinguishable observations.

$$(\forall \lambda, \mathcal{M}, \mathcal{M}', \tau, \tau' \colon \quad \tau = \Upsilon(\widehat{\mathcal{C}}, \mathcal{M}) \;\wedge\; \tau' = \Upsilon(\widehat{\mathcal{C}}, \mathcal{M}') \;\Rightarrow\; dNI(\lambda, \tau, \tau')) \qquad (2.12)$$

where

$$dNI(\lambda, \tau, \tau') \;\triangleq\; (\forall \mathcal{E} \in \Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \tau.\mathcal{C}) \colon\; \tau.\mathcal{M}(\mathcal{E}) = \tau'.\mathcal{M}(\mathcal{E})) \;\Rightarrow\; NI(\lambda, \tau, \tau')$$

[4] The $\widehat{\mathcal{C}}$ subscript in $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ enables this function to depend on enclosing program $\widehat{\mathcal{C}}$ and/or the position of $\mathcal{C}_i$ within $\widehat{\mathcal{C}}$. For example, a language designer might elect to omit $\mathcal{E}$ from $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, x := [\mathcal{E}]_{\bar{\mathsf{f}}})$ if that assignment appears in the scope of an **if** having a Boolean guard $\mathcal{E}'$, where $\Gamma(\mathcal{E}') \not\sqsubseteq \Gamma(x)$ holds.

By construction $\tau.\mathcal{C} = \widehat{\mathcal{C}}$ and $\tau'.\mathcal{C} = \widehat{\mathcal{C}}$ hold, therefore $\tau.\mathcal{C} = \tau'.\mathcal{C}$ and

$$\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \tau.\mathcal{C}) = \Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \tau'.\mathcal{C}) \tag{2.13}$$

hold too. Thus, in ignoring downgraded expressions in the first transition of trace τ (i.e., elements of $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \tau.\mathcal{C})$), $dNI(\lambda, \tau, \tau')$ is also ignoring downgraded expressions in the first transition of trace τ' (i.e., elements of $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \tau'.\mathcal{C})$).

To handle traces that perform downgrades after the first transition in $\widehat{\mathcal{C}}$, it suffices to note that every illicit λ-flow is evidenced in a pair of λ-*pieces*, where a λ-piece of a trace τ is a maximal length subtrace of τ that contains either no downgrades or all of its λ-downgrades in its first transition. The problematic pair of λ-pieces describe executing the same command in different memories $\mathcal{M}$ and $\mathcal{M}'$, where that execution leads to (i) different commands in the last states of the pair of λ-pieces and/or (ii) different updates to a variable x in the pair of λ-pieces, where $\Gamma(x) \sqsubseteq \lambda$ holds.

Given a trace τ and a label λ, we introduce operators $\overset{\rightarrow\lambda}{\tau}$ and $\overset{\lambda\rightarrow}{\tau}$ to identify the subtace $\overset{\rightarrow\lambda}{\tau}$ that is the initial λ-piece of τ and the subtrace $\overset{\lambda\rightarrow}{\tau}$ that is the rest of τ. Since the defining characteristic of a λ-piece is having a λ-downgrade as its starting transition, the the last state of $\overset{\rightarrow\lambda}{\tau}$ is also first state of $\overset{\lambda\rightarrow}{\tau}$ (if $\overset{\lambda\rightarrow}{\tau}$ is non-empty). Thus, $\overset{\rightarrow\lambda}{\tau}$ and $\overset{\lambda\rightarrow}{\tau}$ satisfy the following, for traces and subtraces τ satisfying $|\tau| \geq 2$:

$$\begin{aligned}&(\overset{\rightarrow\lambda}{\tau} = \tau \ \ \wedge \ \ \overset{\lambda\rightarrow}{\tau} = \epsilon) \\ &\quad \vee \ \ (\exists 1 < i < |\tau|: \ \overset{\rightarrow\lambda}{\tau} = \tau[1..i] \ \wedge \ \overset{\lambda\rightarrow}{\tau} = \tau[i..] \ \wedge \ \Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \tau[i].\mathcal{C}) \neq \emptyset)\end{aligned} \tag{2.14}$$

$$(\forall 1 < i < |\overset{\rightarrow\lambda}{\tau}|: \ \Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \overset{\rightarrow\lambda}{\tau}[i].\mathcal{C}) = \emptyset) \tag{2.15}$$

And two λ-pieces $\overset{\rightarrow\lambda}{\tau}$ and $\overset{\rightarrow\lambda}{\tau'}$ are evidence of an illicit λ-flow if the following is

false.

$$\overset{\rightarrow\lambda}{\tau}.\mathcal{C} = \overset{\rightarrow\lambda}{\tau'}.\mathcal{C} \;\Rightarrow\; (\tau[\,|\overset{\rightarrow\lambda}{\tau}|\,].\mathcal{C} = \tau[\,|\overset{\rightarrow\lambda}{\tau'}|\,].\mathcal{C} \;\wedge\; dNI(\lambda, \overset{\rightarrow\lambda}{\tau}, \overset{\rightarrow\lambda}{\tau'})). \tag{2.16}$$

By iterating (recursively) through corresponding λ-pieces in τ and τ', the approach embodied by (2.16) gives a way to identify traces τ and τ' that evidence an illicit λ-flow.

$$\begin{aligned} dpNI(\lambda, \tau, \tau') \;\triangleq\; & \tau \neq \epsilon \;\wedge\; \tau' \neq \epsilon \;\wedge\; \overset{\rightarrow\lambda}{\tau}.\mathcal{C} = \overset{\rightarrow\lambda}{\tau'}.\mathcal{C} \\ & \wedge\; (\forall \mathcal{E} \in \Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \tau.\mathcal{C}){:}\;\; \tau.\mathcal{M}(\mathcal{E}) = \tau'.\mathcal{M}(\mathcal{E})) \\ & \wedge\; \tau.\mathcal{M} =_{\lambda} \tau'.\mathcal{M} \\ & \Rightarrow\; \tau[\,|\overset{\rightarrow\lambda}{\tau}|\,].\mathcal{C} = \tau[\,|\overset{\rightarrow\lambda}{\tau'}|\,].\mathcal{C} \;\wedge\; \overset{\rightarrow\lambda}{\tau} =_{\lambda} \overset{\rightarrow\lambda}{\tau'} \\ & \quad \wedge\; dpNI(\lambda, \overset{\lambda\rightarrow}{\tau}, \overset{\lambda\rightarrow}{\tau'}) \end{aligned}$$

Notice, for traces τ and τ' that each are a single λ-piece (i.e., $\tau = \overset{\rightarrow\lambda}{\tau}$ and $\tau' = \overset{\rightarrow\lambda}{\tau'}$ for all λ) then $dpNI(\lambda, \tau, \tau')$ is equivalent to (2.16), since the the final state of both τ and τ' will have $\bullet$ and, therefore, $\tau[\,|\overset{\rightarrow\lambda}{\tau}|\,].\mathcal{C} = \tau[\,|\overset{\rightarrow\lambda}{\tau'}|\,].\mathcal{C}$ is satisfied.

A characterization like (2.12) now handles downgrades that appear throughout a trace.

$$(\forall \lambda, \mathcal{M}, \mathcal{M}', \tau, \tau'{:}\;\; \tau = \Upsilon(\mathcal{C}, \mathcal{M}) \;\wedge\; \tau' = \Upsilon(\mathcal{C}, \mathcal{M}') \;\Rightarrow\; dpNI(\lambda, \tau, \tau')) \tag{2.17}$$

Some example programs illustrate nuances of (2.17). These programs use RIF labels $\Lambda_{\mathsf{LH}} \triangleq \{\mathsf{L}, \mathsf{H}\}$ with $\mathsf{L} \sqsubset \mathsf{H}$ and reclassifiers $\mathcal{F} \triangleq \{\downarrow, \uparrow\}$ satisfying the following: $\mathcal{T}(\mathsf{H}, \downarrow) \triangleq \mathsf{L}$ and $\mathcal{T}(\mathsf{L}, \uparrow) \triangleq \mathsf{H}$. Assume that $\Gamma(low) = \Gamma(low') = \mathsf{L}$ and $\Gamma(high) = \Gamma(high') = \mathsf{H}$.

The first program assigns (in $\mathcal{C}_3$) a value with label H to a variable with label L without use of a reclassifying expression and, thus, would seem to exhibit an

illicit L-flow.

$$\begin{aligned} \mathcal{C}_1\text{:}&\quad low := [high]_{\downarrow};\\ \mathcal{C}_2\text{:}&\quad low' := [high']_{\downarrow};\\ \mathcal{C}_3\text{:}&\quad low := high; \end{aligned} \tag{2.18}$$

Traces for program (2.18) comprise two L-pieces. One L-piece starts with command $\mathcal{C}_1$ and the other L-piece starts with command $\mathcal{C}_2$ (and includes $\mathcal{C}_3$), due to the following.

$$\Delta^{-}_{(2.18)}(\mathsf{L}, \mathcal{C}_1) = \{high\} \qquad \Delta^{-}_{(2.18)}(\mathsf{L}, \mathcal{C}_2) = \{high'\} \qquad \Delta^{-}_{(2.18)}(\mathsf{L}, \mathcal{C}_3) = \emptyset.$$

Checking, we find (2.17) is satisfied despite our earlier premonition about $\mathcal{C}_3$. A close look shows why the flow to low in assignment $\mathcal{C}_3$ actually ought to be allowed, as characterization (2.17) does: $\mathcal{C}_3$ is assigning an L-downgraded value, since the value of $high$ in the right hand side of $\mathcal{C}_3$ was an L-downgraded value in $\mathcal{C}_1$ and has not been changed since.

A second program changes (in $\mathcal{C}_2$) the value in $high$ after the L-downgrade in $\mathcal{C}_1$.

$$\begin{aligned} \mathcal{C}_1\text{:}&\quad low := [high]_{\downarrow}\\ \mathcal{C}_2\text{:}&\quad high := high'\\ \mathcal{C}_3\text{:}&\quad low := high \end{aligned} \tag{2.19}$$

Traces for (2.19) comprise a single L-piece that starts with command $\mathcal{C}_1$ because:

$$\Delta^{-}_{(2.19)}(\mathsf{L}, \mathcal{C}_1) = \{high\} \qquad \Delta^{-}_{(2.19)}(\mathsf{L}, \mathcal{C}_2) = \emptyset \qquad \Delta^{-}_{(2.19)}(\mathsf{L}, \mathcal{C}_3) = \emptyset$$

Program (2.19) does not satisfy (2.17) since traces τ and τ' exist that generate observations that are not (but should be) indistinguishable. This is because there exist memories $\mathcal{M}$ and $\mathcal{M}'$ satisfying $\mathcal{M} =_{\mathsf{L}} \mathcal{M}'$ and $\mathcal{M}(high') \neq \mathcal{M}'(high')$.

When alternative executions of $\mathcal{C}_1$ are started in these two memories, $\mathcal{C}_3$ generates different updates to low. But having (2.17) not satisfied for this program is what we should desire—the value in $high$ when $\mathcal{C}_3$ executes is not the value that was L-downgraded, so in program (2.19) a value with label H that has not been L-downgraded is being used to update a variable with label L.

A final program illustrates the role of conjunct $\tau[\,|\overset{\rightarrow\lambda}{\tau}|\,].\mathcal{C} = \tau[\,|\overset{\rightarrow\lambda}{\tau'}|\,].\mathcal{C}$ in the consequent of $dpNI(\lambda, \tau, \tau')$.

$$\begin{aligned} \mathcal{C}_1\text{: } \textbf{if } high > 0 \textbf{ then } & \ \mathcal{C}_2\text{: } low := [high']_{\downarrow} \\ \textbf{else } & \ \mathcal{C}_3\text{: } low := [high'']_{\downarrow} \end{aligned} \tag{2.20}$$

Consider traces $\tau = \Upsilon(\mathcal{C}_1, \mathcal{M})$ and $\tau' = \Upsilon(\mathcal{C}_1, \mathcal{M}')$, where the following hold: $\mathcal{M} =_{\mathsf{L}} \mathcal{M}'$, $\mathcal{M}(high > 0) = true$, but $\mathcal{M}'(high > 0) = false$. τ comprises a first L-piece that starts with $\mathcal{C}_1$ and a second L-piece starts with $\mathcal{C}_2$; τ' has the same first L-piece but a second L-piece that starts with $\mathcal{C}_3$.

Program (2.20) does not satisfy (2.17) since $\tau[\,|\overset{\rightarrow\lambda}{\tau}|\,].\mathcal{C} = \mathcal{C}_2$ and $\tau[\,|\overset{\rightarrow\lambda}{\tau'}|\,].\mathcal{C} = \mathcal{C}_3$, so conjunct $\tau[|\overset{\rightarrow\lambda}{\tau}|].\mathcal{C} = \tau[|\overset{\rightarrow\lambda}{\tau'}|].\mathcal{C}$ in the consequent of $dpNI(\mathsf{L}, \tau, \tau')$ does not hold. Having (2.17) not be satisfied is what we should desire, though. $\mathcal{C}_1$ is leaking information about the value of $high$ to the program counter, which determines whether $\mathcal{C}_2$ or $\mathcal{C}_3$ executes next. $\mathcal{C}_2$ and $\mathcal{C}_3$ cause different updates to low.

The test we are using for illicit λ-flows involving the program counter, however, can be too conservative. Consider the program obtained when $\mathcal{C}_3$ in (2.20) is replaced by assignment $low := [high' + 0]_{\downarrow}$, which, by design, causes the same sequence of observations to be produced whether the program executes $\mathcal{C}_2$ or $\mathcal{C}_3$. Conjunct $\tau[\,|\overset{\rightarrow\lambda}{\tau}|\,].\mathcal{C} = \tau[\,|\overset{\rightarrow\lambda}{\tau'}|\,].\mathcal{C}$ in the consequent of $dpNI(\mathsf{L}, \tau, \tau')$ still does not hold because $\mathcal{C}_2$ and $\mathcal{C}_3$ are different statements in the program. Yet the updates produced with the replaced $\mathcal{C}_3$ are now are the same as $\mathcal{C}_2$ produces, so

there is not actually an illicit L-flow. The way to obtain a less conservative test would be to replace conjunct $\tau[|\overrightarrow{\tau}^{\lambda}|].\mathcal{C} = \tau[|\overrightarrow{\tau'}^{\lambda}|].\mathcal{C}$ by a predicate for equivalence of the subsequent executions—but doing that requires a predicate to determine whether two programs are equivalent.

Handling Upgrades. A reclassifying expression $[\mathcal{E}]_{\bar{\mathsf{f}}}$ is considered to perform a λ-*upgrade* if $\Gamma(\mathcal{E}) \sqsubseteq \lambda$ and $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda$ hold. Characterization (2.17) is blind to illicit λ-flows caused by λ-upgrades. Consider, for example, a program comprising assignment

$$low := [low']_{\uparrow} \tag{2.21}$$

that uses RIF labels from Λ_{LH}. Since we have assumed $\Gamma(low) = \mathsf{L}$ and $\Gamma(low') = \mathsf{L}$ hold then this program satisfies (2.17). Yet the program exhibits an illicit L-flow: differences in the initial value of an L-upgraded expression ($[low']_{\uparrow}$) with label H result in differences in updates to a variable (low) with label L where $\Gamma([low']_{\uparrow}) \not\sqsubseteq \Gamma(low)$ holds.

In checking for evidence of illicit λ-flows, (2.17) compares traces τ and τ' that differ in initial values of variables whose labels λ' satisfy $\lambda' \not\sqsubseteq \lambda$. That set of comparisons, however, ignores some of the other expressions whose labels λ' satisfy $\lambda' \not\sqsubseteq \lambda$—those expressions that perform λ-upgrades.

A programming language that supports RIF labels will provide syntax that allows programmers to specify λ-upgrades by using reclassifying expressions. We therefore stiplulate that the language semantics include a function $\Delta^{+}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ that satisfies:

$$\Delta^{+}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i) \;\; \supseteq \;\; \{\mathcal{E} \mid [\mathcal{E}]_{\bar{\mathsf{f}}} \in \Delta(\mathcal{C}_i) \;\wedge\; \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda \;\wedge\; \Gamma(\mathcal{E}) \sqsubseteq \lambda\}$$

So $\Delta^{+}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ contains all expressions that are evaluated when $\mathcal{C}_i$ executes and that satisfy the definition of a λ-upgrade.

Since (2.17) correctly handles variables whose labels are not less restrictive than λ, we consider a transformation from λ-upgraded expressions to such variables. Define *translated command* $\mathbb{T}(\lambda, \widehat{\mathcal{C}})$ to be the command that results from substituting $h_{\mathcal{E}}$ for every expression $\mathcal{E} \in \Delta^{+}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$, where $\mathcal{C}_i$ is a subcommand of $\widehat{\mathcal{C}}$. If a program $\widehat{\mathcal{C}}$ exhibits an illict λ-leak from a λ-upgraded expression $\mathcal{E}$ then, by construction, $\mathbb{T}(\lambda, \widehat{\mathcal{C}})$ exhibits an illicit λ-leak from $h_{\mathcal{E}}$. Moreover, because $\mathbb{T}(\lambda, \widehat{\mathcal{C}})$ contains no λ-upgraded expressions, it exhibits no illicit λ-flows from upgraded λ-expressions *per se*. That means we can use (2.17) to check for illicit λ-leaks in $\widehat{\mathcal{C}}$ by checking translated program $\mathbb{T}(\lambda, \widehat{\mathcal{C}})$ for illicit λ-leaks.

The result is the following characterization of *piecewise noninterference* for a program $\widehat{\mathcal{C}}$; it holds if there are no illicit λ-flows in $\widehat{\mathcal{C}}$.

$$
\begin{aligned}
&PWNI(\widehat{\mathcal{C}}) \;\triangleq\; \\
&\quad (\forall \lambda, \mathcal{M}, \mathcal{M}', \tau, \tau' \colon \quad \tau = \Upsilon(\,\mathbb{T}(\lambda, \widehat{\mathcal{C}}),\, \mathcal{M}) \;\wedge\; \tau' = \Upsilon(\,\mathbb{T}(\lambda, \widehat{\mathcal{C}}),\, \mathcal{M}') \\
&\qquad\qquad \Rightarrow \quad dpNI(\lambda, \tau, \tau'))
\end{aligned}
\tag{2.22}
$$

Note, though, that (2.22) can be unnecessarily conservative. $PWNI(\widehat{\mathcal{C}})$ might not be satisfied even though $\widehat{\mathcal{C}}$ does not exhibit illicit λ-leaks. Here is an example, where labels are from Λ_{LH}, and $\Gamma(low) = \mathsf{L}$ holds.

$$
\mathcal{C}_1\colon\ \textbf{if}\ [low = low]_{\uparrow}\ \textbf{then}\ \mathcal{C}_2\colon low := 1\ \textbf{else}\ \mathcal{C}_3\colon low := 2 \tag{2.23}
$$

To evaluate $PWNI$ involves constructing and checking translated program $\mathbb{T}(\lambda, \mathcal{C}_1)$,

$$
\mathcal{C}_1\colon\ \textbf{if}\ h_{[low=low]_{\uparrow}}\ \textbf{then}\ \mathcal{C}_2\colon low := 1\ \textbf{else}\ \mathcal{C}_3\colon low := 2 \tag{2.24}
$$

where $\Gamma(h_{[low=low]_\uparrow}) = \mathsf{H}$ holds. Boolean guard $[low = low]_\uparrow$ in (2.23) is always $true$, so execution of this program produces no traces in which $\mathcal{C}_3$ executes. In translated program (2.24), Boolean guard $h_{[low=low]_\uparrow}$ ranges over all values, so there are traces involving $\mathcal{C}_3$ as well as traces involving $\mathcal{C}_2$. Thus, $PWNI(\mathcal{C}_1)$ is not satisfied for (2.24) due to the illicit H-flow from upgraded expression $h_{[low=low]_\uparrow}$ to low; in program (2.23) the corresponding illicit H-flow from upgraded expression $[low = low]_\uparrow$ does not actually exist.

The above specious detection of an illicit H-flow could be avoided if we realized that $[low = low]_\uparrow$ in (2.23) is a constant, despite occurences of variables in this expression. In the general case, however, the required semantic analysis quickly becomes intractable for determining new variables and expressions to mirror the relationships among the upgraded expressions being replaced. Consider:

$$\begin{array}{l} \mathcal{C}_1\text{: } \textbf{if } [low > 0]_\uparrow \textbf{ then } high := 1 \textbf{ else } high := 2 \\ \mathcal{C}_2\text{: } \textbf{if } [low + 1 > 1]_\uparrow \textbf{ then } high' := 1 \textbf{ else } high' := 2 \\ \mathcal{C}_3\text{: } \textbf{if } high \neq high' \textbf{ then } low' := 1 \textbf{ else } low' := 2 \end{array} \tag{2.25}$$

The Boolean guards on $\mathcal{C}_1$ and $\mathcal{C}_2$ are equivalent, so command $\mathcal{C}_3$ always executes $low' := 2$. Thus, there is no illicit H-flow in $\mathcal{C}_3$ to low' from $[low > 0]_\uparrow$ in $\mathcal{C}_1$ or from $[low + 1 > 1]_\uparrow$ in $\mathcal{C}_2$. But to reach that conclusion requires knowledge about the equivalence of these expressions so they are not replaced by different variables to form the translated program.

$$
\begin{array}{rcl}
\mathcal{E} & ::= & \nu \mid x \mid op(\mathcal{E}_1, \ldots, \mathcal{E}_n) \\
\mathcal{C} & ::= & \mathbf{skip} \\
 & \mid & x := [\mathcal{E}]_{\bar{\mathsf{f}}} \\
 & \mid & \mathbf{if}\ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \mathbf{then}\ \mathcal{C}_t\ \mathbf{else}\ \mathcal{C}_e\ \mathbf{end} \\
 & \mid & \mathbf{while}\ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \mathbf{do}\ \mathcal{C}_t\ \mathbf{end} \\
 & \mid & \mathcal{C}_1; \mathcal{C}_2
\end{array}
$$

Figure 2.2: Syntax of simple imperative language

2.3 Static Enforcement of PWNI

Type-checking allows static enforcement of PWNI when a chackable class of RIF labels are the basis of the type system. A simple imperative language provides a vehicle for demonstration.

Language and Semantics. Figure 2.2 gives a syntax of expressions and a simple programming language for defining commands. There, ν ranges over constants, x ranges over program variables, $\mathcal{E},\ \mathcal{E}_1,\ \mathcal{E}_2, \ldots$ range over ordinary expressions, and $\mathcal{C}_t,\ \mathcal{C}_e,\ \mathcal{C}_1,\ \mathcal{C}_2,\ \ldots$ range over commands. Note, allowing only reclassifyng expressions (rather than ordinary expressions) in the language syntax for commands is not a limitation—identity reclassifier ϵ must be handled by every RIF label, and reclassifying expression $[\mathcal{E}]_\epsilon$ has the same value and label as ordinary expression $\mathcal{E}$.

Figure 2.3 gives an operational semantics for the programming language of Figure 2.2. We write $\mathcal{M}[x \mapsto \nu]$ there to denote a mapping that is identical to $\mathcal{M}$ except $\mathcal{M}(x) = \nu$. The rules in Figures 2.3 define partial function $\Upsilon(\mathcal{C}, \mathcal{M})$.

The final part of the semantics for this programming langage are definitions for sets $\Delta(\mathcal{C}_i)$, $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$, and $\Delta^{+}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$. These definitions are given in Figure 2.4

$$\textsc{Skip:}\ \frac{}{\langle \mathbf{skip}, \mathcal{M}\rangle \to \langle \bullet, \mathcal{M}\rangle}$$

$$\textsc{Asgn:}\ \frac{\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \nu}{\langle x := [\mathcal{E}]_{\bar{\mathsf{f}}}\,, \mathcal{M}\rangle \to \langle \bullet, \mathcal{M}[x \mapsto \nu]\rangle}$$

$$\textsc{Brch1:}\ \frac{\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathit{true}}{\langle \mathbf{if}\ \ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \mathbf{then}\ \mathcal{C}_t\ \mathbf{else}\ \mathcal{C}_e\ \mathbf{end}, \mathcal{M}\rangle \to \langle \mathcal{C}_t, \mathcal{M}\rangle}$$

$$\textsc{Brch2:}\ \frac{\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathit{false}}{\langle \mathbf{if}\ \ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \mathbf{then}\ \mathcal{C}_t\ \mathbf{else}\ \mathcal{C}_e\ \mathbf{end}, \mathcal{M}\rangle \to \langle \mathcal{C}_e, \mathcal{M}\rangle}$$

$$\textsc{Loop1:}\ \frac{\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathit{true}}{\langle \mathbf{while}\ \ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \mathbf{do}\ \mathcal{C}_t\ \mathbf{end}, \mathcal{M}\rangle \to \langle \mathcal{C}_t; \mathbf{while}\ \ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \mathbf{do}\ \mathcal{C}_t\ \mathbf{end}, \mathcal{M}\rangle}$$

$$\textsc{Loop2:}\ \frac{\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathit{false}}{\langle \mathbf{while}\ \ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \mathbf{do}\ \mathcal{C}_t\ \mathbf{end}, \mathcal{M}\rangle \to \langle \bullet, \mathcal{M}\rangle}$$

$$\textsc{Seq1:}\ \frac{\langle \mathcal{C}_1, \mathcal{M}\rangle \to \langle \bullet, \mathcal{M}_1\rangle}{\langle \mathcal{C}_1; \mathcal{C}_2, \mathcal{M}\rangle \to \langle \mathcal{C}_2, \mathcal{M}_1\rangle}$$

$$\textsc{Seq2:}\ \frac{\langle \mathcal{C}_1, \mathcal{M}\rangle \to \langle \mathcal{C}, \mathcal{M}_1\rangle \quad \mathcal{C} \neq \bullet}{\langle \mathcal{C}_1; \mathcal{C}_2, \mathcal{M}\rangle \to \langle \mathcal{C}; \mathcal{C}_2, \mathcal{M}_1\rangle}$$

Figure 2.3: Operational Semantics

through Figure 2.6. In defining $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$, we write $\mathit{lhs}(\mathcal{C}_i)$ to denote the set of target variables in assignments appearing in a command $\mathcal{C}_i$. This allows us to limit $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ so that it does not include λ-downgrades that cannot influence the value being assigned to a variable x where $\Gamma(x) \sqsubseteq \lambda'$ holds. No illicit λ-flow is possible absent such an assignment.

Typing Rules. Figure 2.7 gives rules to associate a type with each expression. The rules for ordinary expressions are standard. Expr-T intantiates (2.6); AnnExpr-T is based on (2.7).

$$
\begin{array}{l}
\Delta(\mathbf{skip}) \;\triangleq\; \emptyset \\
\Delta(x := [\mathcal{E}]_{\bar{\mathsf{f}}}) \;\triangleq\; \{[\mathcal{E}]_{\bar{\mathsf{f}}}\} \\
\Delta(\mathbf{if}\,[\mathcal{E}]_{\bar{\mathsf{f}}}\,\mathbf{then}\,\mathcal{C}_t\,\mathbf{else}\,\mathcal{C}_e) \;\triangleq\; \{[\mathcal{E}]_{\bar{\mathsf{f}}}\} \\
\Delta(\mathbf{while}\,[\mathcal{E}]_{\bar{\mathsf{f}}}\;\mathbf{do}\,\mathcal{C}_w\;) \;\triangleq\; \{[\mathcal{E}]_{\bar{\mathsf{f}}}\} \\
\Delta(\mathcal{C}_1;\mathcal{C}_2) \;\triangleq\; \Delta(\mathcal{C}_1)
\end{array}
$$

Figure 2.4: Definition of $\Delta(\mathcal{C}_i)$

$$
\begin{array}{l}
\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathbf{skip}) \;\triangleq\; \emptyset \\
\Delta^-_{\widehat{\mathcal{C}}}(\lambda, x := [\mathcal{E}]_{\bar{\mathsf{f}}}) \;\triangleq\; \begin{cases} \{\mathcal{E}\} & \text{if } \Gamma(\mathcal{E}) \not\sqsubseteq \lambda \;\wedge\; \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq \lambda \;\wedge\; \Gamma(x) \sqsubseteq \lambda \\ \emptyset & \text{otherwise} \end{cases} \\
\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathbf{if}\,[\mathcal{E}]_{\bar{\mathsf{f}}}\,\mathbf{then}\,\mathcal{C}_t\,\mathbf{else}\,\mathcal{C}_e) \;\triangleq\; \begin{cases} \{\mathcal{E}\} & \text{if } \Gamma(\mathcal{E}) \not\sqsubseteq \lambda \;\wedge\; \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq \lambda \\ & \quad \wedge\, (\exists x \in \mathit{lhs}(\mathcal{C}_t) \cup \mathit{lhs}(\mathcal{C}_e)\colon \Gamma(x) \sqsubseteq \lambda) \\ \emptyset & \text{otherwise} \end{cases} \\
\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathbf{while}\,[\mathcal{E}]_{\bar{\mathsf{f}}}\;\mathbf{do}\,\mathcal{C}_w) \;\triangleq\; \begin{cases} \{\mathcal{E}\} & \text{if } \Gamma(\mathcal{E}) \not\sqsubseteq \lambda \;\wedge\; \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq \lambda \\ & \quad \wedge\, (\exists x \in \mathit{lhs}(\mathcal{C}_w)\colon \Gamma(x) \sqsubseteq \lambda) \\ \emptyset & \text{otherwise} \end{cases} \\
\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_1;\mathcal{C}_2) \;\triangleq\; \Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_1)
\end{array}
$$

Figure 2.5: Definition of $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C})$

$$
\begin{array}{l}
\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathbf{skip}) \;\triangleq\; \emptyset \\
\Delta^+_{\widehat{\mathcal{C}}}(\lambda, x := [\mathcal{E}]_{\bar{\mathsf{f}}}) \;\triangleq\; \begin{cases} \{\mathcal{E}\} & \text{if } \Gamma(\mathcal{E}) \sqsubseteq \lambda \;\wedge\; \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda \\ \emptyset & \text{otherwise} \end{cases} \\
\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathbf{if}\,[\mathcal{E}]_{\bar{\mathsf{f}}}\,\mathbf{then}\,\mathcal{C}_t\,\mathbf{else}\,\mathcal{C}_e) \;\triangleq\; \begin{cases} \{\mathcal{E}\} & \text{if } \Gamma(\mathcal{E}) \sqsubseteq \lambda \;\wedge\; \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda \\ \emptyset & \text{otherwise} \end{cases} \\
\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathbf{while}\,[\mathcal{E}]_{\bar{\mathsf{f}}}\;\mathbf{do}\,\mathcal{C}_w) \;\triangleq\; \begin{cases} \{\mathcal{E}\} & \text{if } \Gamma(\mathcal{E}) \sqsubseteq \lambda \;\wedge\; \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda \\ \emptyset & \text{otherwise} \end{cases} \\
\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_1;\mathcal{C}_2) \;\triangleq\; \Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_1)
\end{array}
$$

Figure 2.6: Definition of $\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C})$

$$\textsc{Val-T}: \frac{}{\Gamma \vdash \nu{:}\bot} \qquad \textsc{Var-T}: \frac{\Gamma(x) = \lambda}{\Gamma \vdash x{:}\lambda}$$

$$\textsc{Expr-T}: \frac{\Gamma \vdash \mathcal{E}_1{:}\lambda_1 \quad \ldots \quad \Gamma \vdash \mathcal{E}_n{:}\lambda_n}{\Gamma \vdash \ op(\mathcal{E}_1, \ldots, \mathcal{E}_n) : \lambda_1 \sqcup \cdots \sqcup \lambda_n}$$

$$\textsc{AnnExpr-T}: \frac{\Gamma \vdash \mathcal{E}_1{:}\lambda_1 \quad \ldots \quad \Gamma \vdash \mathcal{E}_n{:}\lambda_n}{\Gamma \vdash \ [op(\mathcal{E}_1, \ldots, \mathcal{E}_n)]_{\mathsf{f}_1,\ldots,\mathsf{f}_n} : \mathcal{T}(\lambda_1, \mathsf{f}_1) \sqcup \cdots \sqcup \mathcal{T}(\lambda_n, \mathsf{f}_n)}$$

Figure 2.7: Typing rules for expressions

$$\textsc{Skip-T}: \frac{}{\Gamma, \lambda_\kappa \vdash \mathbf{skip}}$$

$$\textsc{Asgn-T}: \frac{\Gamma \vdash [\mathcal{E}]_{\bar{\mathsf{f}}} : \lambda_g \quad \Gamma \vdash x{:}\lambda_x \quad \lambda_\kappa \sqcup \lambda_g \sqsubseteq \lambda_x}{\Gamma, \lambda_\kappa \vdash x := [\mathcal{E}]_{\bar{\mathsf{f}}}}$$

$$\textsc{Brch-T}: \frac{\Gamma \vdash [\mathcal{E}]_{\bar{\mathsf{f}}} : \lambda_g \quad \Gamma, \lambda_\kappa \sqcup \lambda_g \vdash \mathcal{C}_t \quad \Gamma, \lambda_\kappa \sqcup \lambda_g \vdash \mathcal{C}_e}{\Gamma, \lambda_\kappa \vdash \mathbf{if} \ [\mathcal{E}]_{\bar{\mathsf{f}}} \ \mathbf{then}\ \mathcal{C}_t \ \mathbf{else}\ \mathcal{C}_e \ \mathbf{end}}$$

$$\textsc{Loop-T}: \frac{\Gamma \vdash [\mathcal{E}]_{\bar{\mathsf{f}}} : \lambda_g \quad \Gamma, \lambda_\kappa \sqcup \lambda_g \vdash \mathcal{C}_t}{\Gamma, \lambda_\kappa \vdash \mathbf{while} \ [\mathcal{E}]_{\bar{\mathsf{f}}} \ \mathbf{do}\ \mathcal{C}_t \ \mathbf{end}} \qquad \textsc{Seq-T}: \frac{\Gamma, \lambda_\kappa \vdash \mathcal{C}_1 \quad \Gamma, \lambda_\kappa \vdash \mathcal{C}_2}{\Gamma, \lambda_\kappa \vdash \mathcal{C}_1; \mathcal{C}_2}$$

Figure 2.8: Typing rules for commands

Figure 2.8 shows the familiar rules to deduce whether a command is type-correct. Judgment $\Gamma, \lambda_\kappa \vdash \mathcal{C}$ signifies that a command $\mathcal{C}$ is type correct. Parameter λ_κ in these rules is called *context-type*. It is used in checking for implicit flows. Commands in the body of a *conditional command* ("**if** $[\mathcal{E}]_{\bar{\mathsf{f}}}$ **then** ..." or "**while** $[\mathcal{E}]_{\bar{\mathsf{f}}}$ **do** ...") are considered to be in the *context* of the corresponding *guard*, Boolean expresssion $[\mathcal{E}]_{\bar{\mathsf{f}}}$; for nested conditional commands, the context is defined to be the conjunction of the guards for all of the enclosing conditional commands. Context-type λ_κ associates a type with the context, and λ_κ is combined with the type of the right hand side of an assignment statement.

The following theorem states that if a program is type correct, then this pro-

gram satisfies PWNI.

Theorem 1. *If* $\Gamma, \lambda_\kappa \vdash \mathcal{C}$*, then* $PWNI(\mathcal{C})$ *holds.*

Proof. See Appendix A.1. □

2.4 Discussion

2.4.1 Defining flow

Semantically defining flow from initial values of variables to subsequently computed values (i.e. not just final values of variables) is challenging. The literature of information flow has not provided such definitions, and this thesis has not, either. The difficulty to give a semantic definition for flow between values arises from the use of conditional commands. The following intuitive semantic definition of flow does not apply for values computed during execution of a conditional command: changing the initial value in variable x, causes computed value in variable y to change. For example, changing the initial value in variable x, may cause another branch of an **if** command to be executed, where y is not even a target variable. Is this considered a change of y's value? And if yes, when exactly did this change take place? As another example, changing the initial value in variable x, may just delay the assignment of the same value to y (e.g., the assignment to y may follow an **if** command whose branches have different number of commands). Is this phenomenon considered a flow from x to y? These questions could be answered if we had a semantic definition of flow from initial values of variables to subsequently computed values. Such a defini-

tion would lead to more precise information flow specifications and properties to be enforced.

2.4.2 Reclassifications

In this thesis, we assumed that the authors of RIF labels have already examined the reclassifiers that annotate operations in programs and decided which operations may cause what kind of reclassifications to values. In fact, the decision about which operations cause what reclassifications is relative to the decision of what restrictions are initially associated with input values (e.g., which value is initially considered secret and which public), which depends on assumptions about the environment.

An obvious next step is to create a logic for reasoning about when reclassification is allowed. For instance, for confidentiality, we might consider the following approach:

- Consider a particular initial value ν.
- Create a mapping $\mathcal{K}$ from amounts of information conveyed about ν (this amount may be expressed using information theory) to sets of principals,
- Use $\mathcal{K}$ to create RIF labels for ν and all other initial values. If applying F to an initial value ν' conveys X amount of information about ν, and $\mathcal{K}$ maps X to set S of principals, then the RIF label associated with ν' should map F to S.

With this approach, a reclassification that an operation may cause is attributed to mapping $\mathcal{K}$.

The existence of such a logic would help understand when it is sensible for an operation to cause a reclassification. For example, when is it reasonable to increase the confidentiality level of the output of an operation that gets public inputs, from public to secret? Perhaps when the operation is nondeterministic?

Of course, the decision to change the confidentiality of a value may not be always based on a mathematical logic. For example, a user may decide that she no longer wants her age to be publicly available, and thus, she decides, at some point in her life, to increase the confidentiality level of her age. An enforcement mechanism should be flexible enough to incorporate such arbitrary information flow policies as axioms.

2.5 Related Work

RIF labels specify restrictions based on the history of applied operations, which can be enforced using a static type system. This, then, extends an approach starting with Volpano et al. [93], which gives a type system based on Denning's lattice model [36, 34, 35] and enforces noninterference [44]. PWNI extends noninterference for handling arbitrary reclassifications.

History-based Policies We are not the first to use history of applied operations in types. *Typestates* [89] describe types with structures similar to automata that record how execution history affects a value. Hartson and Hsiao [47] seem to be the first to use access history in access control policies. *Stack inspection* [96] and *history-based access control* [1] are more recent access control polices that use history for defining authorization.

Specifications and Enforcement The work of Chong et al. [29] is perhaps closest to our RIF labels. Information flow policies for confidentiality in [29] use predicates on the execution state as a basis for deciding when a variable may be declassified or should be *erased*, which is a form of classification. In [29], when a reclassification (i.e., declassification or erasure) is specified by the policy of a variable, the value of that variable and all other values derived from that variable are reclassified. Instead, when a reclassification is specified by the RIF label of a variable, only the result of applying the corresponding operation to that variable is reclassified. So, RIF labels specify changes of restrictions on values at a more fine-grained level comparing to policies in [29]. *ClickRelease* [66] advances the expressiveness of declassification policies presented in [29] by using linear-time temporal logic (instead of propositional logic) to express formulae over events.

Paralocks [24] are execution state predicates used to specify reclassifications. These predicates involve boolean variables that can be manipulated during program execution. A security policy is written as $\Sigma \Rightarrow \alpha$, where Σ is a set of predicates and α is a principal. A value associated with such a policy may flow to α only if all predicates in Σ are true. Both Paralocks and policies presented in [29] are enforced using a static type system.

In SHAI [41], policies associated with data containers specify confidentiality and integrity restrictions, giving (i) the set of principals allowed to read data in a container, (ii) a superset of principals allowed to read derived data (i.e., causing a declassification) depending on the execution state, (iii) the set of principals allowed to write in the container, and (iv) the type of data allowed to be written in the container. SHAI uses static and dynamic analysis to enforce these policies.

Jeeves [9] employs *faceted values* [8] to specify declassification between two levels of confidentiality (i.e., *high* and *low*). A faceted value of a variable is a pair of a real and a dummy value guarded by a state predicate. If that state predicate (which can be a faceted value) holds, then the real, possibly confidential, value is allowed to flow to low outputs. Otherwise, the dummy value will flow to low outputs.

The policies mentioned above (i.e., [29, 24, 41, 9]) tie reclassifications to state predicates. RIF labels tie reclassifications to sequences of operations applied to individual values. Execution state could be used to track the sequences of operations applied to individual values. So, RIF labels and the above policies all can be viewed as policies that specify functions from sequences of predicates to restrictions.

The specification language used for writing policies defines the set of functions (from sequences of predicates to restrictions) that can be specified. The larger the set of functions that can be specified, the more expressive these policies are. Our theory of RIF labels is agnostic to the expressive power of the specification language—any function from sequences of reclassifiers to restrictions can be defined using primitive functions $\mathcal{R}$ and $\mathcal{T}$. It is not until chapter 3 and 4 that we give two families of RIF labels with particular expressive power (i.e., RIF automata and κ-labels). In fact, these RIF labels are expressive enough to accommodate the examples under consideration, and at the same time, not too expressive to make the decision of restrictiveness relation $\sqsubseteq$ overly conservative.

Each of the approaches mentioned above (i.e., [29, 24, 41, 9]) proposes one particular specification language for expressing policies. So, they explore poli-

cies with particular expressive power. Possible future work could propose a general theory that handles any function from sequences of predicates to restrictions (including RIF labels). Such a work could then compare the expressive power of previously proposed specification languages, give semantics to the restrictiveness relation between these policies, and show whether previously proposed decision algorithms (employed by the corresponding enforcing mechanisms) are sound and complete with respect to this semantics.

We are not the first to annotate operations with identifiers that specify declassification. FJifP[50] enables a principal to declare *trusted methods* to declassify input values. These trusted methods are similar to *trusted subjects*, first introduced by Bell and LaPadula [14] to handle declassifications. But a trusted method declared by p must always perform a declassification of inputs provided by p. RIF labels are thus more expressive, because an annotated operation might trigger different kinds of reclassifications, depending on the RIF labels that tag these input values.

Rocha et al. [74] employ *policy graphs* to specify declassifications between two security levels. In a policy graph, nodes represent variables and edges represent operation identifiers (similar to our reclassifiers). The tail node of an edge is an input of the corresponding operation and the head node of that edge is an output of that operation. Some of the nodes in a policy graph are defined to be *final*, and they represent values in variables that can be declassified: values in variables of non-final nodes are considered secret, while values in variables that correspond to final nodes are considered public. Policy graphs thus can express only declassifications from secret to public (as compared to RIF labels, which can specify arbitrary reclassifications). Data and control flow analysis is used to

check whether a program satisfies the policy graph. Later, Rocha et al. [75] introduced a specification language for defining policies that might also depend on number of times a function is applied to a given value. This specification language seems more expressive than Rocha et al. [74], although the classes of properties being enforced has not been formally characterized. Similar techniques to those presented in [74] for expressing and enforcing declassification were later employed by Hammer et al. [46] and Johnson et al. [53]. The restrictions of a policy graph can be described using a set of RIF automata.

Li and Zdancewic [62] use lambda terms (i.e. functions) to specify downgrading (i.e. declassification and endorsement) between two security levels. When one of these lambda terms is applied to some value, the resulting value is downgraded. So, an information flow label is a set of lambda terms. A type system is given to enforce these policies. Whenever a lambda term is applied to a value, the type system first checks if that lambda term equals one of the lambda terms in the label of that value, before allowing the resulting value to be downgraded. In general, using functions (e.g., lambda terms) to specify downgrades leads to labels that precisely characterize what function of the input values is allowed to be downgraded. Also, this characterization is independent from the program code. Of course, at the same time, enforcement is more challenging, because it involves deciding equivalence of functions, which is in general undecidable. Also, functions alone may not be intuitive for understanding why a specified reclassification is reasonable. The ultimate goal should be to specify reclassifications based on properties of functions. For example, a declassification could be specified based on the number of bits that the output of the corresponding function may reveal about the inputs.

Variants of Noninterference PWNI is the first to handle reclassification due to applied operations in full generality, but not the first to handle declassification. Classical noninterference [44] is violated when a declassification occurs. This led researchers to propose more expressive formulations. For example, *conditional noninterference* [44, 43] proscribes secret information from flowing to public information, unless the sequence of operations involved in this flow (including the principals that invoke these operations) satisfy some given predicate. So, conditional noninterference defines a function from sequences of operations applied to some secret values, and principals that invoke these operations, to whether the resulting value may be considered public. By also considering the principals that invoke operations, conditional noninterference is more expressive than PWNI, when focusing only on declassifications. However, conditional noninterference cannot express classifications.

Gradual release (GR) [3] dictates that designated declassifications (downgrades for confidentiality) should be the only points of execution where attacker's knowledge about initial high variables may increase. Other work (*delimited release* [79] and *relaxed noninterference* [62]) specifies what expressions of high values in the initial state could be declassified, but it does not restrict where in the program such an expression is allowed to be declassified. *Conditional gradual release* [11], similar to our approach, can specify which particular expression in the program is allowed to be declassified, but then it is not always possible to characterize what initial high information becomes available to an observer at every declassification (if it is important to do so, then we could adopt the restriction on programs employed by [11] and [79], namely declassified variables may not be updated prior to declassification). However, conditional gradual release [11] handles only a two-level lattice and it allows declassifications to de-

pend on high context, implying that a declassification may disclose more information than intended (our approach does not allow declassifications to depend on high context by requiring λ-pieces to end at the same command).

Categorization of Policies Sabelfeld et al. [81] surveys papers that specify and enforce expressive declassification policies. They introduce a four-dimension categorization: *what* information is declassified, *who* declassifies information, *where* in the system information is declassified, and *when* information can be declassified. RIF label focus on *what*, *where*, and *when*. An annotated expression that causes a reclassification indicates *what* will be reclassified, *where* in the program code, and *when* during the program execution. RIF labels do not specify *who* is allowed to trigger a reclassification, but they could be extended to specify this.

The *decentralized label model* (DLM) [70, 69] can express *who* is allowed to declassify information. According to DLM, a value may be declassified only if the declassification command is executed on behalf of the value's *owner* or on behalf of a principal that *acts-for* that owner. We could adopt the approach of DLM and have RIF labels specify principals trusted to execute specific operations. $\mathcal{T}$ would map a RIF label, a reclassifier, and a principal trusted to execute an operation that corresponds to that reclassifier to the desired RIF label. So, a transition would be triggered only if the specified principal executes the corresponding operation.

Another way to characterize information flow policies is based on the three-level *hierarchy of control* proposed by Broberg et al. [25]. *Level 0 control* is a set of possible *flow relations* between information *sources* (e.g., input variables) and

sinks (e.g., output channels). A flow relation indicates that information from the source is allowed to flow to the sink. *Level 1 control* is a determining function that selects which flow relation is allowed. *Level 2 control* is a meta policy controlling the way in which the current flow relation may be changed. Our function $\mathcal{T}$ is an instantiation of *Level 1* and *Level 2* controls, which specify how and when *flow relations* between the information sources and sinks may change.

2.6 Summary

This chapter defined RIF labels, which couple arbitrary changes in restrictions to specified operations. A simple programming language served as a vehicle for exploring the use RIF labels. For these programs, PWNI extends classical noninterference to handle arbitrary changes in restrictions, and a type system implements static enforcement of PWNI for checkable RIF labels.

CHAPTER 3

RIF AUTOMATA

Finite state automata provide the basis for a checkable class of RIF labels called *RIF automata*. This class of labels is supported in the JRIF programming language [55], which we built to gain practical experience with using RIF automata for specifying security policies and to understand the compiler modifications needed for replacing traditional security label types by RIF automata. The *privacy automata* used in the Avanance language [20] for specifying use-based privacy also are instances of RIF automata.

3.1 Formalization of RIF Automata

A finite state automaton can serve as a RIF label λ_α by (i) having the set of reclassifiers be the automaton's input alphabet, and (ii) associating restrictions with each automaton state. Restrictions imposed by λ_α are those associated with the current state of the automaton. Reclassifiers, which by construction can change current state, thus cause a (potentially) different set of restrictions to be imposed.

Formally, a set Λ_{RA} comprising RIF automata is defined relative to some given lattice of restrictions $\langle R,\ \sqsubseteq_R,\ \sqcup_R \rangle$. Each RIF automaton $\lambda_\alpha \in \Lambda_{\mathrm{RA}}$ is

described by a 5-tuple $\langle Q_\alpha, \mathcal{F}, \delta_\alpha, q_\alpha, r_\alpha \rangle$ where[1]

Q_α is a finite set of automaton states,

$\mathcal{F}$ is a finite set of reclassifiers,

δ_α: $Q_\alpha \times \mathcal{F} \to Q_\alpha$ is a (deterministic) next-state transition function,

$q_\alpha \in Q_\alpha$ is the current state of the RIF automaton, and

r_α: $Q_\alpha \to R$ gives the restrictions associated with each state.

By requiring transition function δ to be total, any sequence of reclassifiers from $\mathcal{F}$ causes a sequence of transitions.

$\mathcal{R}_{\mathrm{RA}}$ and $\mathcal{T}_{\mathrm{RA}}$ are defined as expected for RIF automata $\lambda_\alpha \in \Lambda_{\mathrm{RA}}$ and reclassifiers $\mathsf{f} \in \mathcal{F}$:

$$\begin{aligned} \mathcal{R}_{\mathrm{RA}}(\lambda_\alpha) &\triangleq r_\alpha(q_\alpha) \\ \mathcal{T}_{\mathrm{RA}}(\lambda_\alpha, \mathsf{f}) &\triangleq \langle Q_\alpha, \mathcal{F}, \delta_\alpha, \delta_\alpha(q_\alpha, \mathsf{f}), r_\alpha \rangle \end{aligned}$$

Instantiating definition (2.4) of $\sqsubseteq$ using these definitions, we get

$$\lambda_\alpha \sqsubseteq_{\mathrm{RA}} \lambda_{\alpha'} \triangleq (\forall \boldsymbol{F} \in \mathcal{F}^*\colon\ r_\alpha(\delta^*_\alpha(q_\alpha, \boldsymbol{F})) \sqsubseteq_R r_{\alpha'}(\delta^*_{\alpha'}(q_{\alpha'}, \boldsymbol{F}))) \tag{3.1}$$

A sound and complete algorithm to compute this relation is given in Figure 3.1; the proof appears in Appendix A.2.

The properties of $\sqcup$ for RIF automata are satisfied[2] by a product automaton defined in the usual way:

$$\begin{aligned} \lambda_\alpha \sqcup_{\mathrm{RA}} \lambda_{\alpha'} &\triangleq \langle Q_\alpha \times Q_{\alpha'},\ \mathcal{F},\ \delta_{\alpha\times\alpha'},\ \langle q_\alpha, q_{\alpha'} \rangle,\ r_{\alpha\times\alpha'} \rangle \\ \text{where}\quad \delta_{\alpha\times\alpha'}(\langle q_\alpha, q_{\alpha'} \rangle, \mathsf{f}) &\triangleq \langle \delta_\alpha(q_\alpha, \mathsf{f}), \delta_{\alpha'}(q_{\alpha'}, \mathsf{f}) \rangle \\ r_{\alpha\times\alpha'}(\langle q_\alpha, q_{\alpha'} \rangle) &\triangleq r_{\alpha'}(q_\alpha) \sqcup_R r_{\alpha'}(q_{\alpha'}) \end{aligned}$$

[1] Closure δ^*_α: $Q_\alpha \times \mathcal{F}^* \to Q_\alpha$ is derived from δ_α in the usual way: $\delta^*_\alpha(q, \epsilon) \triangleq q$ and $\delta^*_\alpha(q, \mathsf{F}\mathsf{f}) \triangleq \delta_\alpha(\delta^*_\alpha(q, \mathsf{F}), \mathsf{f})$.

[2] See Appendix A.3 for a proof.

$rstr(\lambda_1,\ \lambda_2,\ visited)\ \triangleq$
Let $\lambda_1 = \langle Q_1, \mathcal{F}, \delta_1, q_1, r_1\rangle$
Let $\lambda_2 = \langle Q_2, \mathcal{F}, \delta_2, q_2, r_2\rangle$
if $\langle q_1, q_2\rangle \in visited$ **then return**$(true)$
if $r_1(q_1) \not\sqsubseteq_R r_2(q_2)$ **then return**$(false)$
return$(\forall \mathsf{f} \in \mathcal{F}$: $rstr(\ \mathcal{T}_{\mathrm{RA}}(\lambda_1, \mathsf{f}),\ \mathcal{T}_{\mathrm{RA}}(\lambda_2, \mathsf{f}),\ visited \cup \langle q_1, q_2\rangle\ ))$

Figure 3.1: $rstr(\lambda_1,\ \lambda_2,\ \emptyset)$ computes $\lambda_1 \sqsubseteq_{\mathrm{RA}} \lambda_2$

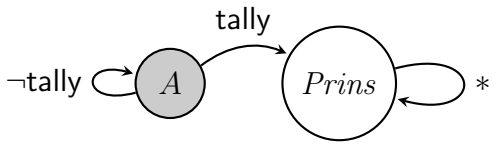


Figure 3.2: RIF automaton $\alpha_{voter(A)}$ for secret ballots

$\langle\langle R, \sqsubseteq_R\rangle, \langle \Lambda_{\mathrm{RA}}, \sqcup_{\mathrm{RA}}, \sqsubseteq_{\mathrm{RA}}\rangle, \mathcal{F}, \mathcal{R}_\alpha, \mathcal{T}_\alpha\rangle$ now forms a class of RIF labels provided that $\lambda_\alpha \sqcup_{\mathrm{RA}} \lambda_{\alpha'} \in \Lambda_{\mathrm{RA}}$ holds for all $\lambda_\alpha, \lambda_{\alpha'} \in \Lambda_{\mathrm{RA}}$. And we will write $\lambda_\alpha(q)$ to specify a RIF automaton λ_α whose current state is q and thus imposes restrictions $r_\alpha(q)$.

Examples of RIF Automata. Though the value of an individual ballot cast in an election might be secret, the value of the majority is not. This is a security policy associated with the ballot that each participant A casts: (i) only A may read the ballot's value and (ii) anyone may read the majority value derived from all of the ballots cast. We can formalize this security policy as a RIF automation $\lambda_{voter(A)}$, where $Prins$ is the set of all principals eligible to learn the election outcome and set $\mathcal{F}$ of reclassifiers includes tally, which will be associated with calculating the election outcome.

Figure 3.2 gives a graphic depiction[3] of $\lambda_{voter(A)}$; the formal definition is:

$$\lambda_{voter(A)} \triangleq \langle \{q_1, q_2\}, \mathcal{F}, \delta_{voter}, q_{voter(A)}, r_{voter(A)} \rangle$$

where

$$\delta_{voter}(q, \mathsf{f}) \triangleq \begin{cases} q_2 & \text{if } q = q_1 \wedge \mathsf{f} = \mathsf{tally} \\ q & \text{othewise} \end{cases}$$

$$r_{voter(A)}(q) \triangleq \begin{cases} \{A\} & \text{if } q = q_1 \\ Prins & \text{if } q = q_2 \end{cases}$$

Restriction $r_{voter(A)}$ is the set of principals who can read the value being labeled by $\lambda_{voter(A)}$:

- the value is secret to A if $q_{voter(A)} = q_1$ holds because $r_{voter(A)}(q_1) = \{A\}$, and
- any value derived by using a tally operation becomes public because $q_{voter(A)} = q_2$ and $r_{voter(A)}(q_2) = Prins$ hold.

Thus, according to the terminology of Figure 2.1, tally causes declassification.

A programmer writes reclassifying expression $[maj(v_A, v_B, \ldots, v_Z)]_{\mathsf{tally}}$ to assert that computing the majority of votes v_A, v_B, ..., v_Z implements the intended effect of a tally operation. That derived value can be stored in a variable whose RIF label imposes no restriction on readers. And assignment

$$winner := [maj(v_A, v_B, \ldots, v_Z)]_{\mathsf{tally}} \tag{3.2}$$

[3] A conventional graphical representation for finite-state automata is used. Circles denote states of the automaton. Arrows between states are labeled with sets of reclassifiers and define allowed transitions, where * abbreviates the list of all reclassifiers. The label inside each state q indicates associated restrictions $r_\alpha(q)$, and the grey-filled state indicates the current automaton state.

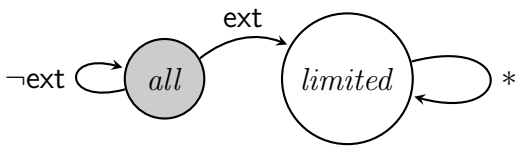


Figure 3.3: RIF automaton of document excerpting.

has exactly that effect if the RIF automaton associated with each variable v_A is $\lambda_{voter(A)}(q_1)$ and the RIF automaton associated with variable $winner$ is at least as restrictive as

$$\lambda_{voter(A)}(q_2) \sqcup_{\text{RA}} \lambda_{voter(B)}(q_2) \sqcup_{\text{RA}} \cdots \sqcup_{\text{RA}} \lambda_{voter(Z)}(q_2)$$

which happens to be equivalent to $\lambda_{voter(x)}(q_2)$ for any $x \in \{A, \ldots, Z\}$. So for any choice of x, assignment (3.2) typechecks according to the rules of §2.3.

A second example sketches RIF automata that enforce integrity policies for a document management system.[4] Given is a set of *original* documents; these are trusted by all principals for all purposes. Operation $ext(D, parms)$ derives a new document by excerpting from document D according to $parms$. Because creative excerpting can be used to generate a document that has different meaning from the original, conservative principals will not use such *derived* documents for certain purposes. Using the terminology of Figure 2.1, excerpting causes deprecation.

One RIF automaton α_D for supporting such a policy might be constructed as follows. It would have a set $\mathcal{F}_{\text{Docs}}$ of reclassifiers that includes ext, which will correspond to excerpting operations. And it would have two automaton states; restrictions indicate whether the associated document D is trusted for *all*

[4]This example is inspired by `TruDocs` [84].

purposes or for *limited* purposes:

$$r_{\mathrm{Docs}}(q) \triangleq \begin{cases} all & \text{if } q = q_1 \\ limited & \text{if } q = q_2 \end{cases}$$

Figure 3.3 gives a graphic depiction for a RIF automaton $\lambda_D(q_1)$ associated with an original document D; $\lambda_D(q_2)$ would be associated with a derived document. The formal definition of λ_D is:

$$\lambda_D \triangleq \langle \{q_1, q_2\}, \mathcal{F}_{\mathrm{Docs}}, \delta_{\mathrm{Docs}}, q_D, r_{\mathrm{Docs}} \rangle$$

where

$$\delta_{\mathrm{Docs}}(q, \mathsf{f}) \triangleq \begin{cases} q_2 & \text{if } q = q_1 \wedge \mathsf{f} = \mathsf{ext} \\ q & \text{othewise} \end{cases}$$

For some applications, we might seek a more refined basis to decide whether a document should be trusted for a specific purpose. The obvious basis for such trust assesments is the set of principals participating in the document's derivation. RIF automata can specify such policies, too. There would be an automaton state q_S for each set S of principals corresponding to a subset of $Prins$. And restrictions being associated with an automaton state q_S would depend on members of S. Transitions are facilitated by having a set $\mathcal{F}_{\mathrm{Doc}}$ of reclassifiers contain an element ext_P for each principal P that invokes an excerpting operation. The formal definition of an automaton λ_D now becomes:

$$\lambda_D \triangleq \langle Q_{Prins}, \mathcal{F}_{\mathrm{Doc}}, \delta_{\mathrm{Doc}}, q_D, r_{\mathrm{Doc}} \rangle$$

where

$$Q_{Prins} \triangleq \{q_S \mid S \in 2^{Prins}\}$$

$$\delta_{\mathrm{Doc}}(q_S, \mathsf{exc}_P) \triangleq q_{S \cup \{P\}}$$

$$r_{\mathrm{Doc}}(q_S) \triangleq \{S\}$$

$$
\begin{array}{lll}
\lambda & ::= & \{\lambda_c; \lambda_i\} \\
\lambda_c & ::= & \texttt{c}\ [\mathit{ListOfTerms}] \\
\lambda_i & ::= & \texttt{i}\ [\mathit{ListOfTerms}] \\
\mathit{ListOfTerms} & ::= & T \mid T, \mathit{ListOfTerms} \\
T & ::= & \mathit{State} \mid \mathit{InitialState} \mid \mathit{Transition} \\
\mathit{State} & ::= & \mathit{ID} : \{\mathit{ListOfPrincipals}\} \\
\mathit{InitialState} & ::= & \mathit{ID}* : \{\mathit{ListOfPrincipals}\} \\
\mathit{Transition} & ::= & \mathit{ID} : \mathit{ID} \rightarrow \mathit{ID}
\end{array}
$$

Figure 3.4: Syntax for JRIF labels, where ID represents an alphanumeric string.

3.2 JRIF

JRIF (Java with Reactive Information Flow) extends Java's types to incorporate RIF automata. Programmers can tag fields, variables, and method signatures with RIF automata, and the JRIF compiler checks whether a program satisfies these RIF automata.

3.2.1 Syntax of JRIF

In JRIF, a *JRIF label* is a pair comprising a c-automaton, which is a RIF automaton for confidentiality, and an i-automaton, which is a RIF automaton for integrity. The JRIF syntax[5] of a JRIF label is given in Figure 3.4. The set of all principals is represented by $\{\_\}$, and the empty set is represented by $\{\}$. Reclassifications that are not given explicitly in a JRIF label are taken to be transitions whose starting and ending states are identical.

Figure 3.5 illustrates how the c-automaton in Figure 3.2 is coded using JRIF

[5]For clarity, the syntax presented in this section simplifies the syntax used in our JRIF implementation.

```
c[s*:{A}, t:{_}, tally:s→t]
```

Figure 3.5: Syntactic representation of a c-automaton

```
boolean{c[q0*:{_}]} check (int{c[q0*:{_}]} in,
    int{c[q1*:{p},q2:{_},C:q1→q2]} PIN)
{   boolean{c[q0*:{_}]} res=false;
    if (reclassify(in==PIN,C))
        res=true;
    return res;}
```

Figure 3.6: PIN check

syntax. The initial state `s` is distinguished by an asterisk $\star$ and maps to principal `A`. State `t` maps to the set of all principals $Prins$ (denoted by $\{\_\}$). Reclassifier `tally` triggers a transition from `s` to `t`.

In JRIF, an expression $\mathcal{E}$ is annotated with reclassifier f by simply writing

$$\textbf{reclassify}(\mathcal{E}, \mathsf{f}).$$

Annotated expressions can appear wherever ordinary Java expression can be used (e.g., in right-hand side expressions of assignments or in guard expressions of conditional commands). Expressions not explicitly annotated trigger no transition on JRIF labels.

A simple method for PIN (personal identification number) checking written in JRIF is shown in Figure 3.6. Here, method `check` takes as arguments an integer input `in` and an integer `PIN`; it checks if these two arguments are equal. The arguments are tagged with different c-automata.[6] Input `in` is, for simplicity, considered public (all principals can read it). `PIN` can initially be read only by principal `p` (the principal that picked this `PIN`), but the result of applying the equality check (annotated with reclassifier `C`) on `PIN` is public. Method `check`

[6] We focus only on confidentiality for this example.

returns the boolean value that results from this equality check, which is considered public. JRIF's compiler decides whether this method is safe, based on typing rules we discuss next.

Label checking

For c-automata, we define λ'_c to be at least as restrictive as λ_c, denoted $\lambda_c \sqsubseteq_c \lambda'_c$, if for all possible sequences of reclassifiers, principals allowed to read the resulting value according to λ'_c are also allowed by λ_c. Relation $\sqsubseteq_c$ is thus formally defined as follows:

$$\lambda_c \sqsubseteq_c \lambda'_c \triangleq (\forall F\colon \mathcal{R}(\mathcal{T}(\lambda_c, F)) \supseteq \mathcal{R}(\mathcal{T}(\lambda'_c, F))). \tag{3.3}$$

For i-automata, λ'_i is at least as restrictive as λ_i, denoted $\lambda_i \sqsubseteq_i \lambda'_i$, if for all possible sequences of reclassifiers, principals that must be trusted according to λ'_i include those that must be trusted according to λ_i. So, relation $\sqsubseteq_i$ is defined as follows:

$$\lambda_i \sqsubseteq_i \lambda'_i \triangleq (\forall F\colon \mathcal{R}(\mathcal{T}(\lambda_i, F)) \subseteq \mathcal{R}(\mathcal{T}(\lambda'_i, F))). \tag{3.4}$$

We extend these restrictiveness relations to JRIF labels by comparing RIF automata pointwise:

$$\{\lambda_c; \lambda_i\} \sqsubseteq \{\lambda'_c; \lambda'_i\} \triangleq (\lambda_c \sqsubseteq_c \lambda'_c) \wedge (\lambda_i \sqsubseteq_i \lambda'_i).$$

The least restrictive JRIF label is denoted with $\{\}$; it allows all principals to read values, and it requires no principal to be trusted. JRIF label $\{\lambda_c\}$ imposes restrictions on confidentiality (according to λ_c), but it imposes no restriction on integrity (no principal is required to be trusted). Similarly, JRIF label $\{\lambda_i\}$ imposes restrictions on integrity, but it imposes no restriction on confidentiality.

The JRIF label inferred by the JRIF compiler for an expression is at least as restrictive as the JRIF labels of all variables in this expression. In particular, the c-automaton of an expression is the join $\sqcup$ of all c-automata of variables in that expression. JRIF constructs the join of two c-automata by taking their product, assigning the intersection of the allowed principals at each state. For integrity, the join of two i-automata is their product, assigning the union of the required principals at each state.

The JRIF label of an annotated expression **`reclassify`**$(\mathcal{E}, \mathsf{f})$ is the JRIF label of expression $\mathcal{E}$ after performing an f transition. Specifically, if $\lambda = \{\lambda_c; \lambda_i\}$ is the JRIF label of $\mathcal{E}$, then $\mathcal{T}(\lambda, \mathsf{f}) \triangleq \{\mathcal{T}(\lambda_c, \mathsf{f}); \mathcal{T}(\lambda_i, \mathsf{f})\}$ is the JRIF label of **`reclassify`**$(\mathcal{E}, \mathsf{f})$.

JRIF extends label checking rules in Figure 2.8 to support all basic Java features, including method overloading, class inheritance, and exceptions.[7] The formal description for all rules employed by JRIF is out of scope for this thesis.

We illustrate label checking by returning to method `check` from Figure 3.6. This method compiles successfully in JRIF, because:

- the c-automaton of `res` is at least as restrictive as the c-automata of `in` and `PIN`, after their taking a `C` transition, and
- the c-automaton of the return value is at least as restrictive as the c-automaton of `res`.

More JRIF examples can be found on the JRIF web page [54].

[7] Label checking rules for Java features already exist in Jif. Their core component is a call to a decision algorithm for the restrictiveness relation. So, we support JRIF labels by substituting Jif's decision algorithm with JRIF's decision algorithm for JRIF label restrictiveness.

Dynamic labels

Sometimes an information flow label becomes known only at execution time. To accommodate this, JRIF adopted Jif's support for *dynamic labels*. JRIF labels in JRIF may be instantiated as runtime values: they may be constructed programmatically, stored in variables, used in static type declarations, and compared dynamically.

Since the actual JRIF label that a dynamic label denotes is not known at compile time, JRIF requires the programmer to provide code that checks for unsafe flows at runtime. For example, consider

$$\texttt{y} = \textbf{reclassify}\,(\texttt{x mod 4}, \texttt{f}) \tag{3.5}$$

where `x` is tagged with dynamic label `L1`, and `y` is tagged with dynamic label `L2`. This assignment statement is secure only when $\mathcal{T}(\texttt{L1}, \texttt{f}) \sqsubseteq \texttt{L2}$ holds. In JRIF, programmers can write `T(L1,f)` to represent a dynamic label whose value is $\mathcal{T}(\texttt{L1}, \texttt{f})$. So, to ensure that $\mathcal{T}(\texttt{L1}, \texttt{f}) \sqsubseteq \texttt{L2}$ holds when (3.5) executes, the programmer must code

$$\textbf{if}\,(\texttt{T(L1,f)} \sqsubseteq \texttt{L2})\,\texttt{y} = \textbf{reclassify}\,(\texttt{x mod 4}, \texttt{f}) \tag{3.6}$$

At compile time, constraint $\texttt{T(L1,f)} \sqsubseteq \texttt{L2}$ informs the type system about the necessary relationship between `L1` and `L2`, because the type system may assume $\texttt{T(L1,f)} \sqsubseteq \texttt{L2}$ holds when the "then" clause starts executing. At runtime the system constructs the JRIF label that results from an `f` transition on `L1` and checks whether `L2` is at least as restrictive. This example also illustrates an interesting property of JRIF labels: the same reclassifier may have different effects on different labels. For some instantiations of `L1`, transitioning according to `f`

may satisfy relation `T(L1,f)` $\sqsubseteq$ `L2`, and for some other instantiations of `L1`, transitioning according to the same reclassifier `f` may not satisfy this relation.

Programming with RIF versus classic labels

We use the term *classic label* to refer to an information flow label that specifies the same restrictions on all values derived from the value with which this label is associated (e.g., [36]).[8] For example, if a user's `PIN` is associated with a classic label specifying that only this user is allowed to read `PIN`, then even the result of a PIN check involving `PIN` is allowed to be read only by that user, instead by everyone. Classic labels often impose more restrictions than needed.

Information flow control systems employing classic labels (e.g., [69, 68]) are forced to use explicit declassification (for confidentiality) and endorsement (for integrity) commands to attach appropriate labels to derived values (i.e., labels that impose weaker restrictions). Reclassifications in JRIF have a concise description in terms of an identifier (i.e., the reclassifier); declassifications and endorsements for classic labels are more verbose, since they glue a target label (i.e., the label that will be attached to the output) and, sometimes, must include the source label (i.e., the label attached to the input) as well.

JRIF labels are more verbose than classic labels, but there is a pay-off—changes to confidentiality and integrity specified in JRIF labels are not expressible by classic labels. Systems using classic labels need additional program code to emulate JRIF labels. This additional code is not automatically checkable for security, and thus, the programmer bears the full responsibility to implement the intended policy correctly.

[8] Classic labels can be simulated by one state RIF automata.

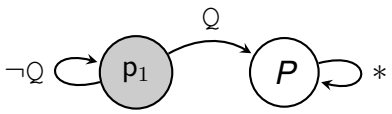
Compared to systems using classic labels, JRIF better separates program logic from information flow policies. This makes programs easier to write and easier to maintain. Suppose, for example, that a programmer decides that some input value—a game player's name—should not be declassified when formerly it was.

- In JRIF, this change to the program involves modifying the JRIF label declaration on any field storing the player's name. The c-automaton of the label would be inspected and edited so that it contains no transitions to automaton states that map to additional principals.
- To accommodate this change in systems that use classic labels, the programmer must not only find and remove all declassification commands that involve the name field explicitly, but she also must remove all declassification commands that involve any expressions to which the game player's name flows. Getting these deletions right is error prone, since the programmer must reason about the flow of information in the code—something the type system was supposed to do.

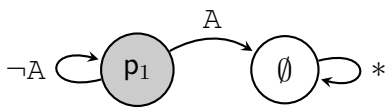
3.2.2 Example applications using JRIF

Battleship

The Battleship game is a good example, because both confidentiality and integrity are important to prevent cheating. Over the course of the game, confidential information is declassified. Ship coordinates are initially fixed and secret, but revealed when opponents guess their coordinates correctly. Also,



(a) A c-automaton for ship-coordinates.



(b) An i-automaton for ship-coordinates.

Figure 3.7: RIF automata for ship-coordinates

players must not be able to change the position of their ships after initial placements.

A simple c-automaton suffices to specify the confidentiality policy for the ship-coordinates of each player. Values derived from ship-coordinates selected by player p_1 should be read only by p_1, because opponent player p_2 is not allowed to learn the position of p_1's ships. The result of whether a ship of p_1 has been hit by the opponent player p_2 may be read by everyone, including p_2. A c-automaton that expresses this policy appears in Figure 3.7a, where $\mathtt{Q}$ is the reclassifier for the operation that checks whether an opponent's attack succeeded, and P is the set of all principals.

The integrity policy of ship-coordinates can be expressed using a simple i-automaton. Once p_1 selects the coordinates of her ships, they are as trusted as p_1. After ship-coordinates are chosen, they may not be changed during the game. So, before the game actually starts, there is a game operation whose reclassifier raises the integrity of all ship-coordinates, thereby ensuring that neither player can make changes. An i-automaton that expresses this policy is presented in Figure 3.7b, where $\mathtt{A}$ is the reclassifier annotating the operation that accepts the initial coordinates.

We borrowed Jif's implementation of Battleship [68] to show that Jif pro-

```
boolean{c[q0*:{_}];i[q1*:{}]} processQuery
(Coordinate[{i[q1*:{}]}]{i[q1*:{}]} query)
{
  Board[{c[q0*:{P},q1:{_},Q:q0→q1];i[q1*:{}]}] brd = this.board;
  List[{i[q1*:{}]}] oppQueries = this.opponentQueries;
  oppQueries.add(query);
  boolean result = brd.testPosition(query);
  return reclassify(result,Q);
}
```

Figure 3.8: Method `processQuery` from JRIF implementation. It checks the success of opponent's hit.

grams can easily be ported to JRIF. We replaced Jif labels with JRIF labels, and we replaced various Jif declassification or endorsement commands with JRIF reclassifications. Methods in the Jif implementation that involved only label parameters and dynamic labels could be used without any modification in the JRIF implementation. Figure 3.8 contains a method of the Battleship implementation in JRIF. This method demonstrates the use of the c-automaton in Figure 3.7a and the application of reclassifier `Q`. The full JRIF source for the Battleship implementation is found on JRIF's web page [54], along with the original Jif source (for comparison).

A Shared Calendar

To explore the expressive power of JRIF labels, we developed a shared calendar application from scratch.[9] The application allows users to create and share events in calendars. Each event consists of fields: time, date, duration, and description. Declassification, classification, endorsement, and deprecation all are employed in this application. Also, users may choose dynamic JRIF labels to

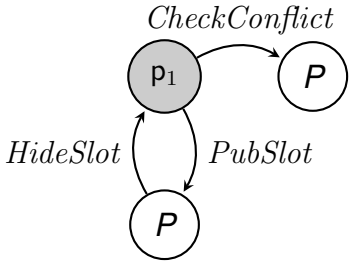
[9]Source code for this shared calendar implementation in JRIF can be found on JRIFs web page [54].

associate with values, so the same reclassifier could have different effects on values with different labels.

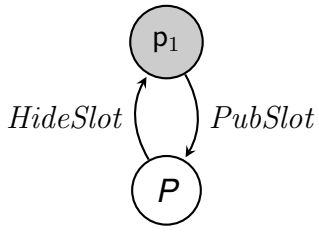
Operations supported by our shared calendar include:

- Create a personal event or a shared event.
- Invite a user to participate in a shared event.
- Accept an invitation to participate in a shared event. Reclassifier: $Accept$
- Cancel a shared event. Reclassifier: $Cancel$
- Check and announce a conflict between personal events (not shared or canceled events) and an invitation for a new shared event. Reclassifier: $CheckConflict$
- Publish an event date and time (but not the event description). Reclassifier: $PubSlot$
- Hide an event date and time. Reclassifier: $HideSlot$

The reclassifiers that annotate these operations change the confidentiality and integrity of events. Once an event is accepted ($Accept$ is applied), the resulting shared event is given the highest integrity, since all of the attendees endorse it. Having the highest integrity implies that no attendee is able to modify this shared event, thereafter. If an event is cancelled ($Cancel$ is applied), then this event is given the lowest integrity, as are all values that subsequently may be derived from it by applying supported operations. With lowest integrity, cancelled events and all values derived from them can be distinguished. If $CheckConflict$ is applied to a personal event and an invitation for a new shared event, then the result gets the lowest confidentiality and the highest integrity. This is because



(a) A c-automaton that permits declassification for conflict checking.



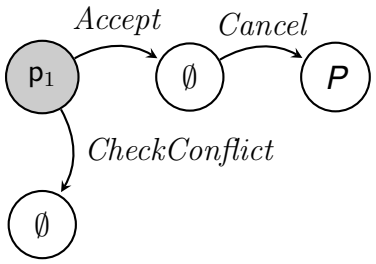
(b) A c-automaton that does not permit declassification for conflict-checking.

Figure 3.9: RIF automata for event confidentiality. Self-loops are omitted for clarity.

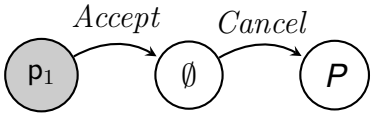
the result is readable and trusted by all principals that learn about the conflict. If $PubSlot$ is applied to an event, then the events date and time can flow to all principals, until a $HideSlot$ is subsequently applied to that event.

Figure 3.9 illustrates c-automata for events created by a principal p_1. The c-automaton in Figure 3.9a permits a full declassification triggered by reclassifier $CheckConflict$; the c-automaton in Figure 3.9b does not. Both c-automata specify a declassification under $PubSlot$, and a classification under $HideSlot$. Figure 3.10 gives corresponding i-automata for the events of p_1. The i-automaton in Figure 3.10a permits a full endorsement triggered by reclassifier $CheckConflict$; the i-automaton in Figure 3.10b does not. Both i-automata specify an endorsement under $Accept$, and a deprecation under $Cancel$. Notice that $CheckConflict$ triggers transitions in both a c-automaton and an i-automaton, contrary to, say, $PubSlot$.

Dynamic labels are used extensively in the shared calendar application. Figure 3.11 excerpts from the conflict-checking method. Here, the label of the event is checked dynamically to see whether it permits the result from the conflict check to be declassified and endorsed before performing the corresponding op-



(a) An i-automaton that permits endorsement for conflict checking.



(b) An i-automaton that does not permit endorsement for conflict-checking.

Figure 3.10: RIF automata for event integrity. Self-loops are omitted; for instance, the result of applying *CheckConflict* to a canceled event has low integrity.

eration. In Figure 3.11, `lEvt` is the dynamic label of the requested shared event `e`, `lCal` is the dynamic label of events in the calendar `cal`, against which the conflict will be checked, and method `hasConflict` returns `true` if a conflict is detected. If `lEvt` and `lCal` after having taken a `C` transition impose no restrictions to the resulting value, and if `hasConflict` is `true`, then a conflict will be announced.

Different users may tag events with different dynamic labels. For example, a user might pick the c-automaton in Figure 3.9a for some events but pick the c-automaton in Figure 3.9b for others. Events can have different i-automata, too. An unshared event has one of the i-automata in Figure 3.10, but an accepted event can be treated with higher integrity and thus tagged with the i-automaton denoted by taking the $Accept$ transition. In addition, the time slot of some events could be either hidden or public. To accommodate these heterogeneously labeled events, we store events in a data structure that makes it easier to aggregate events with different labels. The data structure has two fields: an event and a label. Before processing an event, its label is checked to prevent unspecified flows. Such data structures are common in Jif programs, and they are studied formally in [100].

```
if (T(lEvt,C) ⊑ {c[q0*:{_}];i[q1*:{}]}
    && T(lCal,C) ⊑ {c[q0*:{_}];i[q1*:{}]}){
    if (reclassify(cal.hasConflict(e,lEvt,lCal),C)
        result = true;       // A conflict will be announced
    else
        result = false;     // No conflict will be announced
}
```

Figure 3.11: Checking if the conflict is allowed to be declassified and endorsed, where `C` corresponds to reclassifier *CheckConflict*.

3.2.3 Building a JRIF Compiler

We built a JRIF compiler by modifying the existing Jif compiler in relatively straightforward ways.[10] Extending compilers for other information flow languages ought to be similar. This should not be so surprising: JRIF labels expose the same interface to a type system as native information flow labels.

Our strategy for building JRIF involved three steps:

1. Add syntax for JRIF labels and for annotating expressions with reclassifiers.
2. Add typing rules for annotated expressions (according to §3.2.1).
3. Modify the type checker to handle this more expressive class of labels:
   (a) implement the restrictiveness relation on JRIF labels,
   (b) add an axiom stipulating that this relation is monotone with respect to transition function $\mathcal{T}$.

Item (3b) is essential for supporting our richer language of label comparisons. For example, if relation `l2` $\sqsubseteq$ `l1` holds for two dynamic JRIF labels, then the

[10] The source code for JRIF can be found on JRIF's web page [54].

type checker must be able to deduce that $\mathcal{T}(\texttt{l2}, \texttt{f}) \sqsubseteq \mathcal{T}(\texttt{l1}, \texttt{f})$ holds for every f.

We decided to build JRIF by extending the Jif compiler because Jif is a widely studied language for information flow control and the Jif compiler is readily available. JRIF adds 6k lines of code to Jif (which contains 230k LOC). Out of the 494 Java classes comprising Jif, we modified only 31 and added 48 new classes for JRIF. Of these new classes, 37 are extensions of Jif classes—primarily abstract syntax tree nodes for labels, confidentiality and integrity policies, and code generation classes. Thus, most of the effort in building JRIF focused on extending Jif's functionality rather than on building new infrastructure. Moreover, extending Jif enabled us to harness Jif features, such as dynamic labels, label parameters, and label inference, which reduce the annotation burden on the programmer.

Some features of Jif are orthogonal to enforcing JRIF labels, and JRIF ignores them, for the time being. For instance, Jif uses authority and policy ownership to constrain how labels may be downgraded. Since JRIF labels are concerned with what operation is applied to what value, authority and ownership are ignored for the enforcement of JRIF labels.

3.3 Related Work

Expressive structures, like automata, have previously been used to represent information flow specifications. Program dependence graphs [46, 53], which represent data and flow dependencies between values, specify allowable declassifications. And Rocha et al. [74, 75] employ *policy graphs* to specify sequences of functions that cause declassifications. However, this work does not handle

arbitrary reclassifications; it only handles declassifications.

Many recent systems for information flow control are based on capabilities, including Flume [57], HiStar [99], Asbestos [40], Aeolus [28], Laminar [76], and LIO [88]. We focus our discussion on Flume, but similar arguments apply to other systems.

Flume extends standard operating system abstractions with information flow control. Confidentiality and integrity policies are represented in Flume with unforgeable tokens, called *tags*. System resources are annotated with *labels*, which are collections of tags. Each process has an associated *process label*, which conservatively tracks the confidentiality and integrity policy on the process's memory. When a process performs input operations on sensitive data, the restrictiveness of the process label is raised by adding that resource's tags to the label. Output operations are constrained to affect resources with labels that are at least as restrictive as the current process label. For instance, if a process reads a secret file, then any subsequent attempt to write to a public file will receive an error.

This mechanism alone is usually too restrictive; certain outputs of a program might not actually depend on any secret data, or the purpose of the program may actually be to release secret data in a controlled way. Thus, Flume also assigns to each process a set of capabilities that specify which tags it is permitted to add or remove from its process label. For instance, to add or remove a tag t, a process must have capability t^+ or t^-, respectively. Removing a tag from the process label is equivalent to declassification or endorsement.

Consider the following scenario. Alice has two files: "diary.txt", where she

keeps a personal journal, and "pwds.db", where she stores passwords. Both files contain sensitive information, so she adds a tag, *secret*, to their labels. She gives her editor the $secret^+$ capability, but not $secret^-$. This capability enables the editor to read "diary.txt", but prevents it from outputting its contents to the network or to a file lacking the *secret* tag. In order to read the password file, she gives her password manager the $secret^+$ capability, but also the $secret^-$ capability so that the passwords can be used to log in to remote hosts.

Unfortunately, this scheme gives the password manager more power than Alice might have intended, since it may both read file "diary.txt" and export it to the network. In Flume, Alice's only option is to create separate tags for each file to distinguish secrets that should never be exported and to carefully assign capabilities to processes accordingly.

Extending Flume with RIF automata would provide a better option. As in Jif, we can replace Flume labels with RIF automata, but where the states of these automata are mapped to sets of tags. Thus, each system resource is associated with a RIF automaton, and the process label is a RIF automaton that is at least as restrictive as the current process's memory. Instead of permitting processes to directly add or remove tags, processes receive capabilities for performing transitions on the process label's RIF automaton. Output operations are constrained to resources whose RIF automata are at least as restrictive as the process label.

RIF automata for Flume would allow Alice to express her policies more directly. For "diary.txt", she assigns a RIF automaton with a single state: *secret*. For "pwds.db", she assigns an automaton with two states, *secret* and *public*, and a transition between them called *login*. Then granting the *login* capability to her password manager does not allow it leak "diary.txt", because that file's automa-

ton remains in the *secret* state after the *login* transition.

3.4 Summary

We defined RIF automata, a checkable class of RIF labels, and introduced JRIF. JRIF compiler was implemented by extending the Jif compiler and runtime, thereby demonstrating that RIF automata are easily incorporated into languages that already support information flow types.

JRIF's type system is more expressive than classic information flow type systems. For instance, JRIF allows programmers to specify rich policies based on the sequence of operations used to derive a value. Existing programming languages allow such policies to be emulated in the state and control flow of a program, but doing so invariably makes code more complex and provides few security guarantees.

We illustrated JRIF with an implementation of Battleship and a shared calendar application. Our implementation of Battleship demonstrates that applications developed with Jif may be ported easily to JRIF; the shared calendar demonstrates the separation between policies and program logic that JRIF enables.

CHAPTER 4

κ-LABELS

Encryption causes declassification; decryption causes classification. This chapter presents a checkable class of RIF labels, called κ-*labels*, for specifying and enforcing confidentiality policies in programs that employ such *cryptographic operations*. Functions $\mathcal{R}_\kappa$ and $\mathcal{T}_\kappa$ on κ-labels capture how confidentiality restrictions change with (possibly nested) applications of cryptographic operations.[1] We also give algorithms for deciding a restrictiveness relation $\sqsubseteq_\kappa$ and join $\sqcup_\kappa$ over κ-labels, so that the type system of §2.3 can be used with κ-labels as types. We assume that non-cryptographic operations have no effect on confidentiality restrictions associated with values, and thus, can be ignored.

4.1 Formalization of κ-labels

Our κ-labels are based on Dolev-Yao [39], and they exhibit the limitations of that model. A cryptosystem is characterized in terms of an encryption operation Enc and a decryption operation Dec. Herein, $\Theta(\mathcal{E}, K)$ will range over these operations, where $\mathcal{E}$ is an expression (*viz.* plaintext or ciphertext) and K is a key. Assignments to keys are not permitted. A function KN maps each key K to the set of principals allowed to know the value of K and maps each variable x to the set of principals allowed to know the initial value of x.

$\mathcal{E}$ and K both flow to the result of a cryptographic operation $\Theta(\mathcal{E}, K)$. So, both arguments might be reclassified when $\Theta(\mathcal{E}, K)$ is computed. For exam-

[1] For example, if successive encryptions are followed by the same number of successive decryptions (assuming appropriate keys are used), then the result can be read only by principals knowing the initial value, because the result equals that initial value. Otherwise, the result can be read by everyone.

ple, both $\mathcal{E}$ and K are being declassified from secret to public when ciphertext $Enc(\mathcal{E}, K)$ is considered public. *Cryptographic reclassifiers* from a set $\mathcal{F}_\kappa$ abstract the effect of $\Theta(\mathcal{E}, K)$ on confidentiality restrictions associated with $\mathcal{E}$ and K. In $\Theta(\mathcal{E}, K)$, the reclassifier for $\mathcal{E}$ is denoted $\Theta(\cdot, K)$ and the reclassifier for K is denoted $\Theta(\mathcal{E}, \cdot)$. For the rest of this section, $\Theta(\mathcal{E}, K)$ abbreviates annotated expression $[\Theta(\mathcal{E}, K)]_{\Theta(\cdot,K),\Theta(\mathcal{E},\cdot)}$.

The restrictions associated with outputs of cryptographic operations depend on the restrictions associated with inputs and the cryptographic reclassifiers applied to those inputs. Take the program in Figure 4.1, where Enc and Dec implement symmetric cryptography. Each arrow depicts a value flowing from the expression at the tail of the arrow to the variable at the head; the label on the arrow is the cryptographic reclassifier that corresponds to the flow. From Figure 4.1, we deduce that the value being stored by the first assignment into w might be described as applications of:

$$Enc(\cdot, K) \text{ to } x, \tag{4.1}$$

$$Enc(x, \cdot) \text{ to } K. \tag{4.2}$$

Restrictions on w are derived from these and on any restrictions associated with x and K. In particular, restrictions on w depend on (i) $Enc(\cdot, K)$ and $KN(x)$ due to x as well as (ii) $Enc(x, \cdot)$ and $KN(K)$ due to K. This can be depicted as set of pairs:

$$\{\langle Enc(\cdot, K), KN(x)\rangle, \langle Enc(x, \cdot), KN(K)\rangle\} \tag{4.3}$$

In general, restrictions on computed values depend on sequences of cryptographic operations applied to initial values. Considering again Figure 4.1, restrictions on the computed value in y are represented by extending with $Dec(\cdot, K')$ the reclassifiers in each pair in (4.3), for the flow to y from w, and

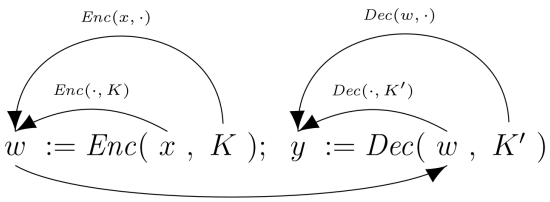


Figure 4.1: A program that implements part of a *cascade protocol* [39].

adding $\langle Dec(w,\cdot), KN(K')\rangle$, for the flow to y from key K' to form the following set:

$$\begin{aligned}&\{\langle Enc(\cdot,K)Dec(\cdot,K'), KN(x)\rangle,\\ &\langle Enc(x,\cdot)Dec(\cdot,K'), KN(K)\rangle,\\ &\langle Dec(w,\cdot), KN(K')\rangle\}.\end{aligned} \tag{4.4}$$

So, (4.4) records the sequences of reclassifiers applied to the initial values of x, K, and K' in computing the final value of y.

Cryptosystems typically define *complement* cryptographic operations. For a symmetric cryptosystem, Dec is the complement of Enc when the following holds:

$$\forall\nu\colon \forall K\colon Dec(Enc(\nu,K),K) = \nu. \tag{4.5}$$

Complements are specified using *rewriting rules* on cryptographic reclassifiers. For a symmetric cryptosystem where (4.5) holds, rewriting rule

$$Enc(\cdot,K)Dec(\cdot,K) \rightarrowtail \epsilon \tag{4.6}$$

signifies that if the computation of a value involves sequence of reclassifiers $Enc(\cdot,K)Dec(\cdot,K)$, then that sequence can be deleted. Deletion is justifiable because whether $Enc(\cdot,K)Dec(\cdot,K)$ or ϵ is applied to a given value, the result will be the same. We assume that we are given set of rewriting rules that completely characterizes all complements. Also, we assume that the complement of cryp-

tographic reclassifier is the only way to efficiently retrieve arguments to which these reclassifiers have been applied.

We write f^{c} to denote the complement of reclassifier f when that complement exists. So, f^{c} is defined iff

$$\mathsf{f}\mathsf{f}^{\mathsf{c}} \rightarrowtail \epsilon \tag{4.7}$$

is a rewriting rule of the given cryptosystem. For example, (4.6) implies that $Dec(\cdot, K)$ is the complement of $Enc(\cdot, K)$: $Enc(\cdot, K)^{\mathsf{c}} = Dec(\cdot, K)$. If $(\mathsf{f}^{\mathsf{c}})^{\mathsf{c}}$ exists, then we require $(\mathsf{f}^{\mathsf{c}})^{\mathsf{c}} = \mathsf{f}$ to hold. This requirement is satisfied by cryptosystems where, for example, $Dec(\cdot, K)^{\mathsf{c}}$ is $Enc(\cdot, K)$. We limit attention to rewriting rules that involve cryptographic reclassifiers of the form $\Theta(\cdot, K)$, and not of the form $\Theta(\mathcal{E}, \cdot)$. Thus, no cryptographic operation is available to recover keys used in computing ciphertexts.[2]

Using rewriting (4.7), a sequence $\mathsf{F}\mathsf{f}\mathsf{f}^{\mathsf{c}}\mathsf{F}'$ of reclassifiers can be rewritten as $\mathsf{F}\mathsf{F}'$. A sequence of reclassifiers is considered *reduced* iff no reclassifier is followed by its complement. Empty sequence ϵ is, by definition, reduced. In Figure 4.1, if K were replaced by K' (so y would equal x when the program terminates) then $Enc(\cdot, K)Dec(\cdot, K')$ in the first pair of (4.4) would be replaced by ϵ.

We use $(\!|\mathsf{F}|\!)$ to denote the reduced sequence to which F can be rewritten.

[2] The limitation rules out cryptographic systems that are vulnerable to plaintext attacks. In such cryptosystems, the following equation would hold: $\Theta'(\mathcal{E}, \Theta(\mathcal{E}, K)) = K$. The rewriting rule that describes this equation is $\Theta(\mathcal{E}, \cdot)\Theta'(\mathcal{E}, \cdot) \rightarrowtail \epsilon$, because $\Theta'(\mathcal{E}, \cdot)$ is the complement of $\Theta(\mathcal{E}, \cdot)$, since applying $\Theta(\mathcal{E}, \cdot)$ to K and then applying $\Theta'(\mathcal{E}, \cdot)$ to that result, yields K.

Specifically, we have:

$$(\!|\epsilon|\!) \triangleq \epsilon \tag{4.8}$$

$$(\!|\mathsf{f}|\!) \triangleq \mathsf{f} \tag{4.9}$$

$$(\!|\mathsf{f}_1\mathsf{f}_2\ldots\mathsf{f}_n|\!) \triangleq \begin{cases} (\!|\mathsf{f}_1\ldots\mathsf{f}_{j-1}\mathsf{f}_{j+2}\ldots\mathsf{f}_n|\!) & \exists j\colon 1 \leq j < n\colon \\ & \quad (\forall i\colon 1 \leq i < j\colon \mathsf{f}_i^{\mathsf{c}} \neq \mathsf{f}_{i+1}) \\ & \quad \wedge\ (\mathsf{f}_j^{\mathsf{c}} = \mathsf{f}_{j+1}) \\ \mathsf{f}_1\mathsf{f}_2\ldots\mathsf{f}_n & \text{otherwise} \end{cases} \tag{4.10}$$

We prove in Appendix A.3 that reduction is associative, which gives:

$$(\!|F_aFF_b|\!) = (\!|F_a(\!|F|\!)F_b|\!) \tag{4.11}$$

Efficient retriever function $\mathcal{X}$ maps each cryptographic reclassifier f to the set of principals that can perform the operation corresponding to cryptographic reclassifier f^{c}. Those principals can retrieve the value of an argument to which f is being applied. So, $\mathsf{p} \in \mathcal{X}(\mathsf{f})$ holds iff (i) f^{c} exists and (ii) principal p is allowed to know the values of all arguments appearing in f^{c}:

$$\mathcal{X}(\mathsf{f}) \triangleq \begin{cases} \emptyset & \text{if } \mathsf{f}^{\mathsf{c}} \text{ does not exist} \\ \{\mathsf{p} \mid \mathsf{f}^{\mathsf{c}} = \Theta(\cdot, K)\ \wedge\ \mathsf{p} \in KN(K)\} & \text{otherwise} \end{cases} \tag{4.12}$$

If $\mathsf{p} \in \mathcal{X}(\mathsf{f})$ holds and ν' is characterized by reclassifier f applied to ν then (4.12) implies that principal p can apply f^{c} to ν' and retrieve ν. We define $\mathcal{X}(\epsilon) \triangleq \boldsymbol{P}$. We write $\overline{\mathcal{X}}(\mathsf{f})$ to denote set complement of $\mathcal{X}(\mathsf{f})$ with respect to set $\boldsymbol{P}$ of all principals. So, $\overline{\mathcal{X}}(\mathsf{f})$ contains those principals that cannot complement f.

For example, in a symmetric cryptosystem where (4.6) is the only rewriting

rule, $\mathcal{X}$ is:

$$\mathcal{X}(Enc(\cdot, K)) \triangleq KN(K) \tag{4.13}$$

$$\mathcal{X}(Enc(\mathcal{E}, \cdot)) \triangleq \emptyset \tag{4.14}$$

$$\mathcal{X}(Dec(\cdot, K)) \triangleq \emptyset \tag{4.15}$$

$$\mathcal{X}(Dec(\mathcal{E}, \cdot)) \triangleq \emptyset \tag{4.16}$$

These equations assert that only those principals allowed to know K can complement reclassifier $Enc(\cdot, K)$ (i.e., decrypt with K); also no principal can complement $Enc(\mathcal{E}, \cdot)$, $Dec(\cdot, K)$, or $Dec(\mathcal{E}, \cdot)$.

A value ν' produced by applying a reduced sequence $(\!|F|\!)$ to some value ν may *safely* be read by two sets of principals:

P_1: Principals allowed to know ν.

P_2: Principals unable to perform the operations to complement all reclassifiers in $(\!|F|\!)$. They learn nothing about ν by reading ν'.

A κ-*atom* $\langle (\!|F|\!), B\rangle$ comprises a reduced sequence $(\!|F|\!)$ and a set B of principals that know ν; the κ-atom characterizes which principals can safely read value ν': P_1 is B, and P_2 is $\bigcup_{\mathsf{f}\in(\!|F|\!)} \overline{\mathcal{X}}(\mathsf{f})$, since for any $\mathsf{f} \in (\!|F|\!)$, if $\mathsf{p} \notin \mathcal{X}(\mathsf{f})$ then p cannot perform an operation that corresponds to f^{c}, so p cannot retrieve ν from ν' by complementing all the reclassifiers recorded in $(\!|F|\!)$.

In general, a set of κ-atoms might be needed to characterize which principals can safely read some value. So, we define a κ-label λ to be a set of κ-atoms. Define Λ_κ to be the powerset of $\{\langle (\!|F|\!), B\rangle \mid F \in \mathcal{F}^*_\kappa, B \subseteq P\}$.

– An initial value in x is tagged with κ-label $\{\langle \epsilon, KN(x)\rangle\}$.

– An initial value in K is tagged with κ-label $\{\langle \epsilon, KN(K)\rangle\}$.

To define a RIF system for κ-labels, we need to give functions $\mathcal{R}_\kappa$ and $\mathcal{T}_\kappa$.

$\mathcal{R}_\kappa(\lambda)$ specifies the set of principals that can safely read a value tagged with $\lambda \in \Lambda_\kappa$. A value ν tagged with λ can be safely read only by principals allowed by all κ-atoms in λ

$$\mathcal{R}_\kappa(\lambda) \triangleq \bigcap_{\langle F,B\rangle\in\lambda} \widehat{\mathcal{R}}_\kappa(\langle F, B\rangle) \tag{4.17}$$

where $\widehat{\mathcal{R}}_\kappa$ is defined by:

$$\begin{aligned} \widehat{\mathcal{R}}_\kappa(\langle \epsilon, B\rangle) &\triangleq B, \\ \widehat{\mathcal{R}}_\kappa(\langle F\mathsf{f}, B\rangle) &\triangleq \widehat{\mathcal{R}}_\kappa(\langle F, B\rangle) \cup \overline{\mathcal{X}}(\mathsf{f}) \end{aligned} \tag{4.18}$$

For example, if a value ν' is tagged with $\{\langle \mathsf{f}_1\mathsf{f}_2\ldots\mathsf{f}_n, B\rangle\}$ then (4.17) asserts that the set of principals that can read ν' is:

$$\begin{aligned} \mathcal{R}_\kappa(\{\langle \mathsf{f}_1\mathsf{f}_2\ldots\mathsf{f}_n, B\rangle\}) =& \widehat{\mathcal{R}}_\kappa(\langle \mathsf{f}_1\mathsf{f}_2\ldots\mathsf{f}_n, B\rangle) \\ =& \widehat{\mathcal{R}}_\kappa(\langle \mathsf{f}_1\mathsf{f}_2\ldots\mathsf{f}_{n-1}, B\rangle) \cup \overline{\mathcal{X}}(\mathsf{f}_n) \\ & \ldots \\ =& \widehat{\mathcal{R}}_\kappa(\langle \epsilon, B\rangle) \cup \overline{\mathcal{X}}(\mathsf{f}_1) \cup \overline{\mathcal{X}}(\mathsf{f}_2) \cup \cdots \cup \overline{\mathcal{X}}(\mathsf{f}_n) \\ =& \underbrace{B}_{P_1} \cup \underbrace{\overline{\mathcal{X}}(\mathsf{f}_1) \cup \overline{\mathcal{X}}(\mathsf{f}_2) \cup \cdots \cup \overline{\mathcal{X}}(\mathsf{f}_n)}_{P_2}. \end{aligned}$$

As another example, consider the value stored in w (Figure 4.1), which is tagged with κ-label (4.3). The set of principals that can read w is:

$$\begin{aligned} & \mathcal{R}_\kappa(\{\langle Enc(\cdot, K), KN(x)\rangle, \langle Enc(x, \cdot), KN(K)\rangle\}) \\ =& \widehat{\mathcal{R}}_\kappa(\langle Enc(\cdot, K), KN(x)\rangle) \cap \widehat{\mathcal{R}}_\kappa(\langle Enc(x, \cdot), KN(K)\rangle) && \text{due to (4.17)} \\ =& (KN(x) \cup \overline{\mathcal{X}}(Enc(\cdot, K))) \cap (KN(K) \cup \overline{\mathcal{X}}(Enc(x, \cdot))) && \text{due to (4.18)} \\ =& (KN(x) \cup \overline{KN(K)}) \cap (KN(K) \cup \overline{\emptyset}) && \text{due to (4.13) and (4.14)} \\ =& KN(x) \cup \overline{KN(K)} && \text{by set theory} \end{aligned}$$

So, w can be safely read by those principals that know x as well as those that do not know K. Principals that do not know x but know K (and thus they could execute $Dec(w, K)$ to retrieve x) are not authorized to read w according to the κ-label in (4.3).

$\mathcal{T}_\kappa(\lambda, \mathsf{f})$ appends f to the reduced sequences of all κ-atoms in λ, when a reclassifier f is applied to a value tagged with $\lambda \in \Lambda_\kappa$. Thus, $\mathcal{T}_\kappa$ is defined as follows:

$$\mathcal{T}_\kappa(\lambda, \mathsf{f}) \triangleq \{\langle (\!|F\mathsf{f}|\!), B\rangle \mid \langle F, B\rangle \in \lambda\}.$$

Finally, restrictiveness relation $\sqsubseteq_\kappa$ instantiates (2.4) using $\supseteq$ for order $\sqsubseteq_R$:

$$\lambda \sqsubseteq_\kappa \lambda' \triangleq (\forall F \in \mathcal{F}^*_\kappa\colon \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda, F)) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda', F))). \tag{4.19}$$

Bottom element $\bot$ is the singleton set $\{\epsilon, P\}$.

There is an algorithm that computes conservative approximation for $\sqsubseteq_\kappa$. This algorithm extends the notion of a *complement sequence* [39] to *maximum complement* F^{c} of a reduced sequence F. F^{c} is constructed by taking the complement of each element of F, starting from the last element of F:

$$\epsilon^{\mathsf{c}} \triangleq \epsilon$$

$$F^{\mathsf{c}} \triangleq \begin{cases} \mathsf{f}^{\mathsf{c}} F_1^{\mathsf{c}} & \text{if } F = F_1\mathsf{f} \text{ and } \mathsf{f}^{\mathsf{c}} \text{ exists,} \\ \epsilon & \text{otherwise.} \end{cases}$$

F is *fully complementable* iff for every f in F, f^{c} exists. Notice that if F_1 and F_2 are fully complementable, then the following holds:

$$(F_1F_2)^{\mathsf{c}} = F_2^{\mathsf{c}}F_1^{\mathsf{c}} \tag{4.20}$$

Decision Algorithm for $\lambda_2 \sqsubseteq_\kappa \lambda_1$. Return *true* iff

$$\forall \langle F_1, B_1\rangle \in \lambda_1, \langle F_2, B_2\rangle \in \lambda_2\colon \mathcal{R}_\kappa(\mathcal{T}_\kappa(\{\langle F_2, B_2\rangle\}, F_2^{\mathsf{c}})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\{\langle F_1, B_1\rangle\}, F_2^{\mathsf{c}}))$$

holds.

The proof in the Appendix A.3 establishes that this algorithm is sound with respect to (4.19) and, therefore, if the algorithm returns *true*, then $\lambda_2 \sqsubseteq_\kappa \lambda_1$ holds according to (4.19).

The above algorithm is complete if λ_1 is a singleton set (i.e., the algorithm returns *true* iff $\lambda_2 \sqsubseteq_\kappa \lambda_1$ holds according to (4.19)). So, type checking an assignment (the only place where the type system in §2.3 invokes this algorithm) whose target variable is tagged with a singleton set λ_1 fails if and only if (4.19) is not satisfied. Consequently, in this case, using the above decision algorithm for deciding $\lambda_2 \sqsubseteq_\kappa \lambda_1$ does not exacerbate the conservatism of the type system.

Join $\sqcup_\kappa$ of two κ-labels is defined to be their union[3]:

$$\lambda \sqcup_\kappa \lambda' \triangleq \lambda \cup \lambda'.$$

This definition of $\sqcup_\kappa$ is sound and complete because, for a κ-label to satisfy the restrictions specified by both λ and λ', it should satisfy the restrictions specified by all κ-atoms in λ and all κ-atoms in λ', so it should satisfy the restrictions specified by all κ-atoms in union $\lambda \cup \lambda'$. We prove in the Appendix A.3 that

[3]The reason to tag values with sets of κ-atoms, instead of one κ-atom $\langle F, B\rangle$ now can be explained. It is not always possible to construct one single κ-atom that is the join of two other κ-atoms. Assume two sets of principals H, P (i.e., the set of all principals) with $H \subset P$ and $KN(K) = H$. Consider symmetric cryptographic operations Enc and Dec, which use only one key K. Consider also definitions (4.13) and (4.14). No κ-atom is equivalent to

$$\langle Enc(\cdot, K)Enc(\cdot, K), H\rangle \sqcup_\kappa \langle Enc(\cdot, K), H\rangle.$$

Assume for contradiction that there is such a κ-atom $\langle F', B\rangle$. By the definition of $\sqsubseteq_\kappa$ and $\sqcup_\kappa$, $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle Enc(\cdot, K)Enc(\cdot, K), H\rangle, F)) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle F', B\rangle, F))$ and $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle Enc(\cdot, K), H\rangle, F)) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle F', B\rangle, F))$ for all F. So, these relations should hold for $F = Dec(\cdot, K)Dec(\cdot, K)$ and $F = Dec(\cdot, K)$. So, $B = H$ and F' should have both exactly one and exactly two $Enc(\cdot, K)$ elements, which is a contradiction.

$\lambda \sqcup_\kappa \lambda'$ is the least upper bound of λ and λ'. So, tuple

$$\langle\langle 2^{\mathsf{P}}, \supseteq\rangle, \langle \Lambda_\kappa,\ \sqcup_\kappa,\ \sqsubseteq_\kappa\rangle, \mathcal{F}_\kappa, \mathcal{R}_\kappa, \mathcal{T}_\kappa\rangle$$

forms a checkable class of RIF labels.

Given these decidable procedures for computing $\sqsubseteq_\kappa$ and $\sqcup_\kappa$, the type system of §2.3 enforces κ-labels. To illustrate, return to the program in Figure 4.1.

We start with the first assignment. Assume principals in P_x are allowed to know the initial value stored in x and principals in P_K are allowed to know key K: $KN(x) = \mathsf{P}_x$ and $KN(K) = \mathsf{P}_K$. So, for mapping Γ from variables to κ-labels, we have $\Gamma(x) = \{\langle \epsilon, \mathsf{P}_x\rangle\}$ and $\Gamma(K) = \{\langle \epsilon, \mathsf{P}_K\rangle\}$. Using rule ANNEXPR-T, the κ-label of expression $Enc(x, K)$ is

$$\mathcal{T}_\kappa(\Gamma(x), Enc(\cdot, K))\ \sqcup_\kappa\ \mathcal{T}_\kappa(\Gamma(K), Enc(x, \cdot))$$

which equals

$$\{\langle Enc(\cdot, K), \mathsf{P}_x\rangle, \langle Enc(x, \cdot), \mathsf{P}_K\rangle\} \tag{4.21}$$

which is equivalent to (4.3). ASGN-T requires the κ-label of w to be at least as restrictive as the κ-label (4.21) of right-hand side expression $Enc(x, K)$. That requirement is satisfied because κ-atom $\langle Enc(x, \cdot), \mathsf{P}_K\rangle$ in (4.21) imposes no restrictions since (4.14) dictates that $Enc(x, \cdot)$ cannot be complemented by any principal, and thus:

$$\forall \mathsf{F} \in \mathcal{F}^*_\kappa\colon \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle Enc(x, \cdot), \mathsf{P}_K\rangle, \mathsf{F})) = \mathsf{P}.$$

So, the type $\Gamma(w)$ of w can be the κ-label:

$$\Gamma(w) = \{\langle Enc(\cdot, K), \mathsf{P}_x\rangle\} \tag{4.22}$$

By definition (4.17) of $\mathcal{R}_\kappa$ and (4.13), the principals that can safely read w are:

$$\mathcal{R}_\kappa(\{\langle Enc(\cdot, K), \mathsf{P}_x\rangle\}) = \mathsf{P}_x \cup \overline{\mathcal{X}}(Enc(\cdot, K)) = \mathsf{P}_x \cup \overline{KN(K)} = \mathsf{P}_x \cup \overline{\mathsf{P}_K}.$$

Thus, the type system correctly deduces that the value in w may be safely read by principals that know x (i.e., P_x) as well as by principals that cannot complement $Enc(\cdot, K)$ (i.e., principals that do not know K). Notice, in this example, cryptographic reclassifier $Enc(\cdot, K)$ triggers a declassification, because:

$$\mathcal{R}_\kappa(\Gamma(x)) \subset \mathcal{R}_\kappa(\mathcal{T}_\kappa(\Gamma(x), Enc(\cdot, K))).$$

We now examine both assignments in Figure 4.1, assuming (4.22), $\Gamma(y) = \{\langle \epsilon, \mathsf{P} \rangle\}$, and $\mathsf{P}_x \subset \mathsf{P}$ hold. If $K = K'$, then by reading y, principals in $\mathsf{P} - \mathsf{P}_x$ could learn the value in x, which they are not supposed to know. We show that in this case the program in Figure 4.1 is not type-correct. We also show that if $K \neq K'$, then the program in Figure 4.1 is type-correct, which is reasonable because principals in $\mathsf{P} - \mathsf{P}_x$ learn nothing about value in x by reading y, since they cannot efficiently retrieve x from y.

- Assume $K = K'$ holds. In this case, the κ-label of expression $Dec(w, K')$ is $\{\langle \epsilon, \mathsf{P}_x \rangle, \langle Dec(w, \cdot), KN(K') \rangle\}$, which is retrieved using rule AnnExpr-T and (4.22). By rule Asgn-T, y should be tagged with a κ-label at least are restrictive as $\langle \epsilon, \mathsf{P}_x \rangle$. This assignment is not type correct, because $\langle \epsilon, \mathsf{P} \rangle$, the κ-label of y, is not at least are restrictive as $\langle \epsilon, \mathsf{P}_x \rangle$. Notice, in this case, cryptographic reclassifier $Dec(\cdot, K')$ triggers a classification, because $\mathcal{R}_\kappa(\Gamma(w)) \supset \mathcal{R}_\kappa(\mathcal{T}_\kappa(\Gamma(w), Dec(\cdot, K')))$ holds.
- Assume $K \neq K'$ holds. Using rule AnnExpr-T, the κ-label of expression $Dec(w, K')$ is $\{\langle Enc(\cdot, K) Dec(\cdot, K'), \mathsf{P}_x \rangle, \langle Dec(w, \cdot), KN(K') \rangle\}$. Due to definitions (4.15) and (4.16), no principal can complement $Dec(\cdot, K')$ or $Dec(w, \cdot)$, and thus, the κ-label of $Dec(w, K')$ imposes no restriction. So, y can be tagged with $\langle \epsilon, \mathsf{P} \rangle$. Thus, the assignment is type correct.

Significance of Type-correctness with κ-labels

To explore the connection between type correct programs involving κ-labels and flows, assume some principal p is not allowed to know a value ν. We argue that, when executing a type-correct program $\mathcal{C}$, if ν flows to a value ν' that can be read by p, then p cannot efficiently retrieve ν from ν'. So p does not learn ν by reading ν'.

Consider a κ-label λ_{p}:

$$\lambda_{\mathsf{p}} \triangleq \{\langle F, B\rangle \mid \langle F, B\rangle \text{ is a } \kappa\text{-atom } \wedge\ \mathsf{p} \in \widehat{\mathcal{R}}_{\kappa}(\langle F, B\rangle)\}.$$

Any value that can be read by p must be tagged with a label $\lambda \in \Lambda_{\kappa}$ where $\lambda \sqsubseteq_{\kappa} \lambda_{\mathsf{p}}$ holds. For purpose of illustration, we will call values tagged with such labels λ *low*. Values that p is not supposed to know (and thus not supposed to read) are tagged with $\lambda' \in \Lambda_{\kappa}$, such that $\lambda' \not\sqsubseteq_{\kappa} \lambda_{\mathsf{p}}$; these values will be called *high*. We now argue that if a high value flows to a low value during execution, then p cannot efficiently retrieve the high value from that low value, which means that in executing type-correct programs, principals do not learn values they are not supposed to know.

Because $\mathcal{C}$ is type-correct, then $\mathcal{C}$ satisfies PWNI, due to Theorem 1. According to PWNI, the only way for a high value to flow to a low value is through a λ_{p}-downgrade. In $\mathcal{C}$, an expression that performs a λ_{p}-downgrade is an annotated expression of the form $[\Theta(\mathcal{E}, K)]_{\Theta(\cdot,K),\Theta(\mathcal{E},\cdot)}$, abbreviated as $\Theta(\mathcal{E}, K)$, where at least one of $\mathcal{E}$ and K is high but $\Theta(\mathcal{E}, K)$ is low. For example, a downgrading expression could be $Enc(x, K)$, where p is not allowed to know x and K but p is allowed to read $Enc(x, K)$. Theorem 2 below implies that for every expression $\Theta(\mathcal{E}, K)$ that performs a λ_{p}-downgrade in $\mathcal{C}$, principal p cannot efficiently

retrieve either $\mathcal{E}$ (i.e., $\mathsf{p} \in \overline{\mathcal{X}}(\Theta(\cdot, K))$) or K (i.e., $\mathsf{p} \in \overline{\mathcal{X}}(\Theta(\mathcal{E}, \cdot))$) by reading $\Theta(\mathcal{E}, K)$.

Theorem 2. *Consider a principal* $\mathsf{p} \in \mathsf{P}$ *and* κ*-label* λ_{p}*. Assume* $\Gamma, \lambda_\kappa \vdash \mathcal{C}$ *for a context-type* $\lambda_\kappa \in \Lambda_\kappa$ *and a mapping* Γ *that maps variables in a command* $\mathcal{C}$ *to* κ*-labels in* Λ_κ*. Let* $\Theta(\mathcal{E}, K)$ *perform a* λ_{p}*-downgrade in* $\mathcal{C}$*. Then,* $\mathsf{p} \in \overline{\mathcal{X}}(\Theta(\cdot, K))$ *and* $\mathsf{p} \in \overline{\mathcal{X}}(\Theta(\mathcal{E}, \cdot))$*.*

Proof. See Appendix A.4. □

Consequently, when executing a type-correct program, a high value (i.e., cannot be read by p) flows to a low value (i.e., can be read by p) only when p cannot efficiently retrieve that high value from the low value.

4.2 Related Work

Variants of noninterference exist to support declassification caused by special commands, such as match queries[4] [94] and one-way functions [91]. Encryption can be viewed as a special command, too. Two approaches have been explored for cryptographic operations: computational and symbolic [32]. The permitted disclosure of secret values only when they are encrypted is formalized as *computational noninterference* (CNI) [58]. CNI handles only encryption (not decryption), and it is enforced by type systems introduced in Laud et al. [60] for *passive* and Fournet et al. [42] for *active* adversaries. Smith et al. [86] propose a variant of noninterference that handles both encryption and decryption, and it is enforced

[4] A *match query* checks whether two objects are equal. For example, a match query is use to check whether a certain string is the password of a given user.

using a type system. All approaches discussed above are computational. In this chapter, we employed a symbolic approach to model cryptographic operations.

Models based on the symbolic analysis first were introduced by Dolev and Yao [39]. *Cryptographically masked flows* [4] uses a symbolic analysis of cryptographic systems, and this is then enforced by a type system. The formalism of cryptographically masked flows is based on a two-level lattice (i.e., secret and public), while our theory handles richer lattices (formed by sets of principals). Also, types in [4] record only sequences of applied encryptions (not decryptions), because a decryption is allowed to be applied only to a value previously encrypted with a proper key. We impose no restriction on the application of cryptographic operations, and thus our κ-labels need to record sequences of any cryptographic operation (encryption and decryption). Laud [59] shows that type correctness according to [4], together with some additional conditions, imply CNI, thereby establishing a connection between cryptographically masked flows (which is based on symbolic analysis) and CNI (which is based on computational analysis). Cortier et al. [32] generalizes this connection by showing that programs secure according to a symbolic analysis are also secure according to a computational analysis.

CHAPTER 5

LABEL CHAINS

This chapter introduces and analyzes dynamic enforcement mechanisms that employ label chains of arbitrary length. We start by formalizing label chains (§5.1) and defining *enforcers* (§5.2). We next (§5.3) extend block-safe noninterference (BNI) [56] to stipulate that sensitive information must not leak to observers of variables and label chains, whether execution terminates normally or is blocked by an enforcer. Enforcer ∞-*Enf* is derived (§5.4); it uses label chains of infinite length to enforce BNI. A family k-*Enf* of enforcers for finite label chains approximate ∞-*Enf*, and these too are shown (§5.5) to satisfy BNI.

5.1 Formalization of Label Chains

We posit that each variable x in a program is associated with a possibly infinite label chain $\langle \ell_1, \ell_2, \ldots, \ell_i, \ell_{i+1}, \ldots \rangle$, where label ℓ_1 specifies sensitivity for the value stored in x and, for $i \geq 1$, label ℓ_{i+1} specifies sensitivity for ℓ_i. Labels come from a possibly infinite *underlying* lattice $\mathcal{L} = \langle L, \sqsubseteq \rangle$, with join operation $\sqcup$ and bottom element $\bot$. For[1] $\ell, \ell' \in \mathcal{L}$, if $\ell \sqsubseteq \ell'$ holds, then ℓ' is *at least as restrictive as* ℓ, signifying that information is allowed to flow from data tagged with ℓ (i.e., data whose sensitivity is ℓ) to data tagged with ℓ'.

Every principal p is assigned a fixed label ℓ that allows p to read variables and labels whose sensitivity is at most ℓ. Thus, if a variable x is tagged with ℓ', then p assigned label ℓ is allowed to read x iff $\ell' \sqsubseteq \ell$ holds.

[1] When $\mathcal{L} = \langle L, \sqsubseteq \rangle$, we write $\ell \in \mathcal{L}$ to assert that $\ell \in L$ holds.

Unless a label chain $\langle \ldots, \ell_i, \ell_{i+1}, \ldots \rangle$ is *monotonically decreasing*

$$\ell_{i+1} \sqsubseteq \ell_i \text{ for } i \geq 1,$$

sensitive information can be leaked. Here is why. Consider a variable x associated with a non-monotonically decreasing label chain $\langle \mathsf{L}, \mathsf{H}, \ldots \rangle$, where $\mathsf{L} \sqsubset \mathsf{H}$. Principals assigned label L are authorized to read the value in x. When the read access to x succeeds, these principals conclude that the label of x is L. Thus, success in reading x leaks to a principal assigned L information about the label of x—even though label chain $\langle \mathsf{L}, \mathsf{H}, \ldots \rangle$ defines the sensitivity of that label to be H. Such leaks cannot occur in monotonically decreasing label chains.

Label chains will be implemented by sequencing individual labels that are stored in a memory M. Domain $dom(M)$ of a memory M includes:

- *Variables* that store (say) integers ($\nu \in \mathbb{Z}$). We use lower case letters (e.g., a, w, x, h, m, l) for variables. So, $M(x)$ denotes the integer stored in a variable x by memory M. Let Var denote the set of variables. *Constants* (e.g., 1, 2, 3) are a subset of Var whose values are fixed.
- *Tags* that store labels ($\ell \in \mathcal{L}$) representing sensitivity. The label for x is stored at tag $T(x)$ in M; its value is $M(T(x))$. Some tags store labels representing the sensitivity of other tags. The label for $T^i(x)$ is stored in tag $T^{i+1}(x)$, for $i \geq 1$. We use the term *value* v when referring to either a label or an integer.
- *Auxiliaries* that store additional information needed by an enforcement mechanism (e.g., a stack to track implicit flows in nested **if** commands). The names of auxiliaries are μ_1, μ_2, etc.

Tags and auxiliaries are called *metadata*. A possibly infinite label chain $\langle T(q), T^2(q), \ldots, T^i(q), \ldots \rangle$ will be associated with each *identifier* q that is either a variable or a tag (but not an auxiliary). For convenience, we adopt the notation $T^0(q) \triangleq q$ and $T^{i+1}(q) \triangleq T(T^i(q))$. We also may write $T^i(q)$ instead of $M(T^i(q))$ for that value in memory M if there will be no ambiguity (e.g., $T^i(q) \sqsubseteq \ell$, $T^i(q) \sqcup T^j(q')$). We require the following:

$$\forall i \geq 1: \quad T^i(q) \in dom(M) \Rightarrow T^{i-1}(q) \in dom(M).$$

The mappings defined by M and T^i extend from identifiers to expressions in the usual way, where $\oplus$ represents an operator:

$$M(e \oplus e') \triangleq M(e) \oplus M(e') \tag{5.1}$$

$$T^i(e \oplus e') \triangleq T^i(e) \sqcup T^i(e'), \text{ for } i \geq 1. \tag{5.2}$$

Variables are categorized according to whether their label chains may change during execution. For a *flexible* variable w, the entire associated label chain might be updated when an assignment to w is executed. For an *anchor* variable a, the label stored in $T(a)$ remains fixed throughout execution, and the remaining elements of the label chain satisfy:

$$M(T^i(a)) = \bot \text{ for any } T^i(a) \in dom(M) \text{ with } i > 1. \tag{5.3}$$

Associating this form of chain with an anchor variable is sensible because $T(a)$ is declared in the program text and thus that label can be considered public (i.e., $T^2(a)$ is $\bot$) when execution starts. No other information can be encoded in $T(a)$ during execution because $T(a)$ remains fixed. So, $T(a)$ ought to be considered public during execution, too. The requirement that label chains be monotonically decreasing then leads to (5.3). A constant ν is a special case of an anchor

variable, so we have

$$M(T^i(\nu)) = \bot, \text{ for any } T^i(\nu) \in dom(M) \text{ with } i \geq 1. \tag{5.4}$$

5.2 Enforcers

Execution of a command C on a memory M can be represented by a *trace* τ, which is a potentially infinite sequence

$$\langle C_1, M_1\rangle \rightarrow \langle C_2, M_2\rangle \rightarrow \ldots \rightarrow \langle C_n, M_n\rangle \rightarrow \ldots$$

where a *state* $\langle C_i, M_i\rangle$ gives the command C_i that will next be executed and gives a memory M_i to be used in that execution. A sequence τ' of states is considered a *subtrace* of τ iff $\tau = \ldots \rightarrow \tau' \rightarrow \ldots$. We write $|\tau|$ to denote the length of τ and $\tau[i]$ to denote the ith state in τ for $1 \leq i \leq |\tau|$. We also write $\langle C, M\rangle =^0 \langle C', M'\rangle$ to denote that two states agree on the command and the values in variables:

- $C = C'$,
- $dom(M) \cap Var \;=\; dom(M') \cap Var$, and
- $\forall x \in dom(M) \cap Var$: $M(x) = M'(x)$.

A set of operational semantics rules is usually employed to formally define traces. This chapter uses a simple **while**-language (Figure 5.1) and operational semantics rules R (Figure 5.2) for this language. Notice that R does not reference metadata. Notation $M[x \mapsto \nu]$ in (AsgnA) and (AsgnF) defines a memory that equals M except x is mapped to ν. *Conditional delimiter* **exit** in rules for *conditional* commands (If1), (If2), (Wl1), and (Wl2) signifies the end of these commands ([77,

$$
\begin{array}{lrcl}
\textit{(Constants)} & \nu & \in & \mathbb{Z} \\
\textit{(Anchor variables)} & a, x & \in & \mathit{Var}_A \\
\textit{(Flexible variables)} & w, x & \in & \mathit{Var}_F \\
\textit{(Expressions)} & e & ::= & \nu \mid x \mid e_1 \oplus e_2 \\
\textit{(Commands)} & C & ::= & \textbf{skip} \mid x := e \mid C_1; C_2 \mid \\
 & & & \textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end} \mid \\
 & & & \textbf{while } e \textbf{ do } C \textbf{ end}
\end{array}
$$

Figure 5.1: Syntax

$$
\text{(SKIP)}\ \frac{}{\langle \textbf{skip}, M\rangle \to \langle \textbf{stop}, M\rangle} \qquad \text{(ASGNA)}\ \frac{\nu = M(e)}{\langle a := e, M\rangle \to \langle \textbf{stop}, M[a \mapsto \nu]\rangle}
$$

$$
\text{(ASGNF)}\ \frac{\nu = M(e)}{\langle w := e, M\rangle \to \langle \textbf{stop}, M[w \mapsto \nu]\rangle}
$$

$$
\text{(IF1)}\ \frac{M(e) \neq 0}{\langle \textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M\rangle \to \langle C_1; \textbf{exit}, M\rangle}
$$

$$
\text{(IF2)}\ \frac{M(e) = 0}{\langle \textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M\rangle \to \langle C_2; \textbf{exit}, M\rangle}
$$

$$
\text{(WL1)}\ \frac{M(e) \neq 0}{\langle \textbf{while } e \textbf{ do } C \textbf{ end}, M\rangle \to \langle C; \textbf{while } e \textbf{ do } C \textbf{ end}; \textbf{exit}, M\rangle}
$$

$$
\text{(WL2)}\ \frac{M(e) = 0}{\langle \textbf{while } e \textbf{ do } C \textbf{ end}, M\rangle \to \langle \textbf{exit}, M\rangle} \qquad \text{(EXIT)}\ \frac{}{\langle \textbf{exit}, M\rangle \to \langle \textbf{stop}, M\rangle}
$$

$$
\text{(SEQ1)}\ \frac{\langle C_1, M\rangle \to \langle \textbf{stop}, M'\rangle}{\langle C_1; C_2, M\rangle \to \langle C_2, M'\rangle} \qquad \text{(SEQ2)}\ \frac{\langle C_1, M\rangle \to \langle C_1', M'\rangle \qquad C_1' \neq \textbf{stop}}{\langle C_1; C_2, M\rangle \to \langle C_1'; C_2, M'\rangle}
$$

Figure 5.2: Structural Operational Semantics R

80]). When execution of the corresponding taken branch completes, rule (EXIT) is triggered.[2] Notice that C_i in a state $\langle C_i, M_i\rangle$ can be a command C as defined in Figure 5.1, a termination delimiter such as **stop**, or a command involving a conditional delimiter **exit**.

Operational semantics rules R define a function $\mathit{trace}_R(C, M)$ that maps a

[2]For a **while** command, the number of times (EXIT) is triggered equals the number of times rules (WL1) and (WL2) are invoked for this command.

command C and a memory M to the trace that represents the entire execution of C started with initial memory M. For $trace_R(C, M)$ to be well-defined, M should be *healthy for* C denoted $M \models \mathcal{H}(C)$ and formalized below, where $x{\in}C$ indicates that $x{\in}\mathit{Var}$ appears in C:

$$M \models \mathcal{H}(C) \;\triangleq\; \forall x{\in}\mathit{Var}\colon (x \in C \Rightarrow x \in \mathit{dom}(M)) \\ \wedge (x \in \mathit{dom}(M) \Rightarrow M(x) \in \mathbb{Z}).$$

If $trace_R(C, M)$ is finite, then by definition it ends with a *normal termination* $\langle \mathbf{stop}, M' \rangle$ state.

Executing command C on memory M under the auspices of an *enforcer* E leads to a trace $\tau = trace_E(C, M)$. We expect that all traces generated by an enforcer will satisfy some policy P of interest, and E blocks traces to ensure that P is satisfied. So, a trace $\tau = trace_E(C, M)$ may end with *blocked state* $\langle \mathbf{block}, M' \rangle$; omitting the blocked state from τ and projecting only commands and variables should yield a prefix of $trace_R(C, M)$.

We now formalize the definition of an enforcer. Define $\mathit{blk}(\tau)$ to hold iff τ ends with a blocked state, and define prefix relation $\tau \preceq \tau'$:

$$\tau \preceq \tau' \;\triangleq\; |\tau| \leq |\tau'| \;\wedge\; l = |\tau| \\ \wedge (\forall 1 \leq i < l\colon \tau[i] =^0 \tau'[i]) \;\wedge\; (\neg \mathit{blk}(\tau) \Rightarrow \tau[l] =^0 \tau'[l]))$$

E is an enforcer on R for P if

- $(\forall C, M\colon trace_E(C, M) \preceq trace_R(C, M))$ and
- the image of $trace_E$ satisfies P.

An enforcer E may employ metadata. For enforcers E that we consider, this metadata includes label chains of size $n_E \geq 1$ and a set Aux_E of auxiliaries.

A memory M is *healthy for E, $\mathcal{L}$, and C* denoted by $M \models \mathcal{H}(E, \mathcal{L}, C)$ iff

- $M \models \mathcal{H}(C)$,
- for each variable x, the domain of M contains exactly n_E tags comprising a label chain, where all tags are mapped to $\mathcal{L}$:

$$\begin{aligned} &\forall x \in dom(M) \cap Var\colon \\ &\quad (\forall 1 \leq i \leq n_E\colon T^i(x) \in dom(M) \wedge M(T^i(x)) \in \mathcal{L}) \\ &\quad \wedge (\forall i > n_E\colon T^i(x) \notin dom(M)) \end{aligned}$$

- the domain of M contains auxiliaries Aux_E:

$$\forall \mu \in Aux_E\colon \mu \in dom(M)$$

- flexible variables in M are associated with monotonically decreasing label chains,
- anchor variables in M are associated with label chains satisfying (5.3), and
- constants in M are associated with label chains satisfying (5.4).

Notice that if $M \models \mathcal{H}(E, \mathcal{L}, C)$ and $x \in dom(M)$, then the sensitivity $T^{n_E+1}(x)$ of the last element $T^{n_E}(x)$ of the label chain associated with x does not belong to $dom(M)$, and thus, $T^{n_E+1}(x)$ is not defined.

For $Init_E$ a mapping from auxiliaries in Aux_E to initial values, M is defined to be *initially healthy* for E, $\mathcal{L}$, and C denoted $M \models \mathcal{H}_0(E, \mathcal{L}, C)$ iff:

- $M \models \mathcal{H}(E, \mathcal{L}, C)$, and
- auxiliaries Aux_E are initialized according to $Init_E$:

$$\forall \mu \in Aux_E\colon M(\mu) = Init_E(\mu).$$

We only consider enforcers E satisfying certain restrictions, where $\mathcal{L}$ is a lattice:

(E1) Trace $trace_E(C, M)$ is defined when $M \models \mathcal{H}_0(E, \mathcal{L}, C)$ holds.

(E2) For memory M_i in a state of τ, $M_i \models \mathcal{H}(E, \mathcal{L}, C)$ holds.

(E3) E updates the label chain of a flexible variable w only in performing an assignment to w, or at **exit** for a conditional command whose branches (taken or untaken) contain an assignment to w.[3]

5.3 Threat Models and BNI

Observations. Our threat model is formulated in terms of principals observing updates to variables. When an assignment to a flexible variable w is executed, each element in set $A(w) \triangleq \{w, T(w), \ldots, T^i(w), \ldots\}$ is updated. When an assignment to an anchor variable a is executed, only $A(a) \triangleq \{a\}$ is updated.

A principal p assigned label ℓ observes updates to variables and tags q, where $T(q)$ is in the domain of a memory M and $M(T(q)) \sqsubseteq \ell$ holds. A similar threat model is used in [10] (i.e., observations at the granularity of individual memory locations, whose sensitivity may change during execution). Principals do not observe updates to an identifier q when $T(q) \notin dom(M)$ holds, indicating q is

[3] More formally, we have:

$\forall C$: $\forall M$: $\forall C_1 \in C$:
  $\forall \tau = \langle C_1; C_2, M_1\rangle \xrightarrow{*} \langle C_1'; C_2, M_2\rangle$ a subtrace of $trace_E(C, M)$:
    $\forall w$ a flexible variable:
      "$w := e$" $\notin C_1 \Rightarrow (\forall i \geq 1$: $T^i(w) \in dom(M_1) \Rightarrow M_1(T^i(w)) = M_2(T^i(w)))$

where $C' \in C$ denotes that C' is a subcommand of C. A conditional delimiter (e.g., **exit**) is not considered a subcommand of C.

not covered by the security policy to be enforced. That implies principals do not observe updates to the last element of a label chain. Also, a principal p assigned ℓ might be allowed to observe updates to an identifier $T^j(q)$ (i.e., $T^{j+1}(q) \sqsubseteq \ell$) but p might not be allowed to observe updates to a preceding identifier $T^i(q)$ (i.e., $T^{i+1}(q) \not\sqsubseteq \ell$) for $0 \leq i < j$, due to monotonically decreasing label chains.

We can now formalize the *observation* available to a principal p assigned label ℓ when an assignment executes. Define the projection of a memory M with respect to label ℓ and a set S of identifiers:

$$M|_\ell^S \triangleq \{\langle q, M(q)\rangle \mid q \in S \wedge T(q) \in dom(M) \wedge M(T(q)) \sqsubseteq \ell\}.$$

If assignment to a variable x is performed and memory M results, then observation $M|_\ell^{A(x)}$ is generated to p. Notice that $M|_\ell^{A(x)}$ can be empty (i.e., $M|_\ell^{A(x)} = \emptyset$). Also, as desired, $M|_\ell^{A(x)}$ does not contain any observation for the last element of the label chain associated with x.

A sequence of observations are generated along with a trace. Given a trace τ, define $\tau|_\ell^S$ to be a sequence $\theta = \Theta_1 \rightharpoonup \ldots \rightharpoonup \Theta_n$ of those observations involving identifiers in set S and having sensitivity at most ℓ:

$$\epsilon|_\ell^S \triangleq \epsilon$$

$$\langle C, M\rangle|_\ell^S \triangleq \epsilon$$

$$(\langle C, M\rangle \rightarrow \langle C', M'\rangle \rightarrow \tau)|_\ell^S \triangleq$$

$$\begin{cases} M'|_\ell^{A(x)\cap S} \rightharpoonup (\langle C', M'\rangle \rightarrow \tau)|_\ell^S, & \text{if } C \text{ is ``}x := e; C'\text{''} \\ (\langle C', M'\rangle \rightarrow \tau)|_\ell^S, & \text{otherwise} \end{cases}$$

We abbreviate $\tau|_\ell^S$ by $\tau|_\ell^k$, when $S = \{T^i(x) \mid 0 \leq i \leq k \ \wedge \ x \in Var\}$. We write $\theta =_{obs} \theta'$ to denote equality of sequences of observations with empty observations omitted, since θ and θ' are then equivalent for principals.

We call this the *strong threat model* to distinguish it from threat models (e.g., section 6.1) that allow observations only on variables. Notice that the strong threat model does not generate observations when identifiers are updated at execution points other than assignments to variables (e.g., no observation is generated when a conditional delimiter **exit** is executed).

Block-safe Noninterference. The goal of an enforcer is to prevent leaks from observations. This can be formalized as Block-safe Noninterference (BNI), which is a form of *noninterference* [44]. Formally, BNI stipulates that if two finite traces (terminated normally or blocked) of the same command agree on initial values whose sensitivity is at most ℓ, then observations (involving variables and tags) visible to a principal assigned label ℓ should be the same in the two executions. We define for $k \geq 0$ specialized k-BNI that restricts observations to variables and up to k^{th} tag (i.e., $T^0, T^1, \ldots, T^k$); and we write $M|_\ell$ to abbreviate $M|_\ell^{dom(M)}$.

$$
\begin{aligned}
k\text{-BNI}(E, \mathcal{L}, C) \quad \triangleq \quad & (\forall \ell \in \mathcal{L}\text{: } \forall M, M'\text{:} \\
& \quad M \models \mathcal{H}_0(E, \mathcal{L}, C) \wedge M' \models \mathcal{H}_0(E, \mathcal{L}, C) \\
& \quad \wedge M|_\ell = M'|_\ell \\
& \quad \wedge \tau = \mathit{trace}_E(C, M) \text{ is finite} \\
& \quad \wedge \tau' = \mathit{trace}_E(C, M') \text{ is finite} \\
& \quad \Rightarrow \tau|_\ell^k =_{obs} \tau'|_\ell^k)
\end{aligned}
$$

If k-BNI$(E, \mathcal{L}, C)$ holds for every C, then we say that E enforces k-BNI$(\mathcal{L})$.[4] We study enforcers on semantics R for security policy k-BNI$(\mathcal{L})$. If for all $k \geq 0$ and $\mathcal{L}$, enforcer E satisfies k-BNI$(\mathcal{L})$, then we say that E enforces BNI.

[4]Notice that if E satisfies $(k+1)$-BNI$(\mathcal{L})$, then E satisfies k-BNI$(\mathcal{L})$.

0-BNI is stronger than *termination insensitive noninterference* (TINI) [95], which is the policy enforced by many dynamic enforcement mechanisms that have been proposed in the past (e.g.[6, 18, 26, 30, 65]). TINI only concerns normally terminated executions but does not consider finite traces that correspond to blocked executions. So TINI ignores flows that occur during a trace that becomes blocked by the enforcement mechanism and thus potentially allows sensitive information to be leaked. 0-BNI considers all finite traces: both terminated normally and blocked traces. So, an enforcement mechanism that satisfies 0-BNI will satisfy TINI, too.

0-BNI is weaker than *termination sensitive noninterference* (TSNI) [92]. TSNI considers all traces: infinite and finite (terminated normally as well as blocked). Because 0-BNI ignores infinite traces, 0-BNI allows leakage through termination channels that already exist in a program (due to non-terminating while-loops). An enforcement mechanism enforcing 0-BNI can be extended into one that enforces TSNI by employing techniques similar to those presented in [12].

5.4 Enforcer ∞-*Enf*

We use familiar insights for handling explicit and implicit information flow to define enforcer ∞-*Enf*, which uses infinite label chains (i.e., $n_{\infty\text{-}Enf} = \infty$) to enforce BNI for programs written in the programming language of Figure 5.1. We later derive from ∞-*Enf* the k-*Enf* family of enforcers that use finite label chains.

5.4.1 Updating Label Chains of Flexible Variables

When $w := e$ executes in isolation, the value of e flows explicitly to flexible variable w. So, w should be at least as sensitive as e. Therefore, just prior to the assignment ∞-*Enf* updates tag $T(w)$ with $T(e)$. But with that update, the value of $T(e)$ flows explicitly to $T(w)$, so ∞-*Enf* also must update tag $T^2(w)$ with $T^2(e)$. Repeating the argument, we conclude that prior to executing $w := e$, ∞-*Enf* should update tag $T^i(w)$ with $T^i(e)$, for $i \geq 0$.

Information can also flow implicitly from the *context* of an assignment to the target variable of that assignment. Context ctx of a command C is a set of boolean expressions that includes all guards involved in determining whether C is reached. So if C appears in the body of a *conditional command* (**if** or **while** commands) having guard e, then e belongs to the context of C. For example, consider:

$$\textbf{if } x > 0 \textbf{ then } w := w' \textbf{ else } w := w'' \textbf{ end} \tag{5.5}$$

Here, context ctx of $w := w'$ and $w := w''$ is $x > 0$. Notice, if prior to (5.5) $T^i(w') \neq T^i(w'')$ holds for some $i \geq 0$, then the value in $T^i(w)$ after the **if** command depends on ctx. For $T(ctx)$ the sensitivity of context ctx, if we require that $T(ctx) \sqsubseteq T^{i+1}(w)$ holds then ctx is prevented from leaking through $T^i(w)$.

In general, for q a flexible variable or a tag, if q is assigned the value of e (e is an expression of variables or tags), then information can flow explicitly from e to q and implicitly from ctx to q. Thus, the sensitivity $T(q)$ of q should be updated to $T(e) \sqcup T(ctx)$. But, this update might also require that $T^i(q)$ have been updated for $i \geq 1$. Recursive function $UT(q, e, ctx)$ below describes tag

updates triggered by q being updated with e in context ctx:

$$UT(q, e, ctx) \triangleq$$
$$T(q) := T(e) \sqcup T(ctx);$$
$$UT(T(q), T(e) \sqcup T(ctx), ctx)$$

For $w := e$ in context ctx, $UT(w, e, ctx)$ expands to[5]

$$\forall i \geq 1\text{: } T^i(w) := T^i(e) \sqcup T(ctx). \tag{5.6}$$

5.4.2 Preventing Leaks through Anchor Variables

Execution of assignments to flexible variables need not be blocked because they cannot cause leaks. But executing an assignment to an anchor variable can cause a leak. So, a prerequisite to executing $a := e$ for an anchor variable a is for an enforcer to check a *block condition* $G_{a:=e}$. $G_{a:=e}$ is defined so that if it holds, then the explicit and implicit flows to a in $a := e$ do not constitute leaks. If $G_{a:=e}$ does not hold, then execution is blocked.

Blocking an execution might cause implicit flow of sensitive information, as seen with (1.8). We address this by generalizing the definition of context ctx for a command to include block conditions that could have been checked before execution reached the current point. This generalization is consistent with the role of ctx: execution of $a := e$ and of any command that might follow is conditioned on whether $G_{a:=e}$ holds. If execution of C depends on $G_{a:=e}$ being *true*, then $G_{a:=e}$ belongs to the context ctx of C.

[5]This expansion uses the fact that the label chain associated with ctx is monotonically decreasing: $T^{i+1}(ctx) \sqsubseteq T^i(ctx)$.

We now show how to construct $G_{a:=e}$ for an assignment $a := e$ in context ctx. $G_{a:=e}$ needs to protect against explicit and implicit flows to a. The value of e explicitly flows to a. So, a should be at least as sensitive as e:

$$T(e) \sqsubseteq T(a).$$

Because execution of $a := e$ depends on $G_{a:=e}$, the context of $a := e$ is $ctx \cup \{G_{a:=e}\}$. Information flows implicitly from this context to a. Variable a should thus be at least as sensitive as $T(ctx \cup \{G_{a:=e}\}) = T(ctx) \sqcup T(G_{a:=e})$:

$$T(ctx) \sqcup T(G_{a:=e}) \sqsubseteq T(a).$$

So, for $G_{a:=e}$ to hold, then $T(e) \sqsubseteq T(a)$ and $T(ctx) \sqcup T(G_{a:=e}) \sqsubseteq T(a)$ should both hold. We conclude

$$G_{a:=e} \;\Rightarrow\; (T(e) \sqsubseteq T(a) \;\wedge\; T(ctx) \sqcup T(G_{a:=e}) \sqsubseteq T(a))$$

or equivalently

$$G_{a:=e} \;\Rightarrow\; (T(e) \sqcup T(ctx) \sqcup T(G_{a:=e}) \sqsubseteq T(a))\,. \tag{5.7}$$

One possible solution for $G_{a:=e}$ in (5.7) is:[6]

$$G_{a:=e} \triangleq (T(e) \sqcup T(ctx) \sqsubseteq T(a)). \tag{5.8}$$

To verify that it is a solution, first compute sensitivity $T(G_{a:=e})$:

$$\begin{aligned} T(G_{a:=e}) &= T(\; T(e) \sqcup T(ctx) \sqsubseteq T(a)\;) \\ &= T^2(e) \sqcup T^2(ctx) \sqcup T^2(a) && \{\text{due to (5.2)}\} \\ &= T^2(e) \sqcup T^2(ctx) \sqcup \bot && \{T^2(a) = \bot\} \\ &= T^2(e) \sqcup T^2(ctx) && \{\ell \sqcup \bot = \ell\} \end{aligned} \tag{5.9}$$

[6] $G_{a:=e}$ in (5.8) is used by all dynamic flow sensitive enforcement mechanisms we know. But, we seem to be the first to give a derivation of this block condition from first principles (i.e., explicit and implicit flows) and monotonically decreasing label chains as a solution of (5.7).

Substituting $T^2(e) \sqcup T^2(ctx)$ for $T(G_{a:=e})$, substituting $T(e) \sqcup T(ctx) \sqsubseteq T(a)$ for $G_{a:=e}$ in (5.7), and noticing that $T^2(e) \sqcup T^2(ctx) \sqsubseteq T(e) \sqcup T(ctx)$ (due to monotonically decreasing label chains), equation (5.7) becomes equivalent to a true statement, which is what we needed to verify solution (5.8).

5.4.3 Operational Semantics for ∞-*Enf*

Enforcer ∞-*Enf* uses (i) UT (see (5.6)) for deducing label chains and (ii) $G_{a:=e}$ (see (5.8)) for blocking possibly unsafe assignments. UT and $G_{a:=e}$ only mention tags for variables and sensitivity $T(ctx)$ of the context (i.e., they do not need ctx, $T^2(ctx)$, $T^3(ctx)$, ...). $T(ctx)$ is the join of the sensitivity of each guard and each block condition that determines the reachability of a command. ∞-*Enf* uses the auxiliaries below to maintain $T(ctx)$:

- cc (conditional context) keeps track of the sensitivity of the guards in all **if** and **while** commands that encapsulate the next command to be executed, and
- bc (blocking context) keeps track of the sensitivity of the information revealed by block conditions that might have influenced the reachability of the next command to be executed.

So, $Aux_{\infty\text{-}Enf} = \{cc, bc\}$. We now show how $T(ctx)$ is defined in terms of cc and bc.

Auxiliary bc is a tag that (conservatively) stores a label at least as restrictive as the sensitivity of all block conditions that could have been evaluated so far in the execution. Any observation after assignment $a := e$ reveals information

about $G_{a:=e}$ and about context ctx in which $G_{a:=e}$ is evaluated. So, whenever a block condition $G_{a:=e}$ is checked, ∞-*Enf* updates bc with $T(G_{a:=e})$ and $T(ctx)$:

$$bc := T(G_{a:=e}) \sqcup T(ctx). \tag{5.10}$$

From (5.9) and monotonicity of label chains (i.e., $T^2(ctx) \sqsubseteq T(ctx)$), we then get:

$$bc := T^2(e) \sqcup T(ctx). \tag{5.11}$$

No block condition has been evaluated at the beginning of execution, so auxiliary bc is initialized to $\bot$: $Init_{\infty\text{-}Enf}(bc) = \bot$.

Auxiliary cc is implemented in ∞-*Enf* using a stack. Whenever execution enters a conditional command (**if** or **while** command), the sensitivity of the corresponding guard is pushed onto cc; when execution exits a conditional command, the top element of cc is popped. We write $\lfloor cc \rfloor$ to denote the join of all labels in cc

$$\lfloor cc \rfloor \triangleq \bigsqcup_{\ell \in cc} \ell \tag{5.12}$$

where $\ell \in cc$ signifies that ℓ appears in some element in stack cc. At the beginning of execution, no conditional command has been entered, and thus auxiliary cc is initialized to the empty stack ϵ with $\lfloor \epsilon \rfloor \triangleq \bot$. So, we have $Init_{\infty\text{-}Enf}(cc) = \epsilon$.

Putting it all together, sensitivity $T(ctx)$ is $\lfloor cc \rfloor \sqcup bc$. Substituting $\lfloor cc \rfloor \sqcup bc$ for $T(ctx)$ in (5.8), block condition $G_{a:=e}$ becomes:

$$T(e) \sqcup \lfloor cc \rfloor \sqcup bc \sqsubseteq T(a). \tag{5.13}$$

Substituting $\lfloor cc \rfloor \sqcup bc$ for $T(ctx)$ in (5.11), the update of bc becomes:

$$bc := T^2(e) \sqcup \lfloor cc \rfloor \sqcup bc. \tag{5.14}$$

So, $G_{a:=e}$ and the update of bc have now been expressed in terms of tags and auxiliaries that ∞-*Enf* uses.

$$\text{(SKIP)}\ \frac{}{\langle \mathbf{skip}, M\rangle \rightarrow \langle \mathbf{stop}, M\rangle}$$

$$\text{(ASGNA)}\ \frac{v = M(e) \qquad G_{a:=e} \qquad \ell = M(T^2(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)}{\langle a := e, M\rangle \rightarrow \langle \mathbf{stop}, M[a \mapsto v, bc \mapsto \ell]\rangle}$$

$$\text{(ASGNAFAIL)}\ \frac{v = M(e) \qquad \neg G_{a:=e} \qquad \ell = M(T^2(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)}{\langle a := e, M\rangle \rightarrow \langle \mathbf{block}, M[bc \mapsto \ell]\rangle}$$

$$\text{(ASGNF)}\ \frac{v_0 = M(e) \qquad \forall i \geq 1\text{: } v_i = M(T^i(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)}{\langle w := e, M\rangle \rightarrow \langle \mathbf{stop}, M[\forall i \geq 0 : T^i(w) \mapsto v_i]\rangle}$$

Figure 5.3: Operational Semantics for **skip** and assignments, where $G_{a:=e}$ is $M(T(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq M(T(a))$

Rule (ASGNA) in Figure 5.3 uses (5.13) and (5.14) to execute $a := e$. If $G_{a:=e}$ does not hold, then rule (ASGNAFAIL) is triggered. Notice that in (ASGNAFAIL), bc is updated with a label representing the sensitivity of the context in which execution is blocked. That label in bc could then dictate which principals are allowed to learn the reason why an execution ended (i.e., due to a **block** versus due to a **stop**) without leaking sensitive information.

Rule (ASGNF) for assignment $w := e$ to flexible variable w implements (5.6), given $T(ctx) = \lfloor cc \rfloor \sqcup bc$. So, the label chain of w is updated as follows:

$$\forall i \geq 1\text{: } T^i(w) := T^i(e) \sqcup \lfloor cc \rfloor \sqcup bc.$$

Rules for conditional commands are given in Figure 5.4. They adopt techniques employed by other dynamic enforcement mechanisms (e.g., [30]) to update auxiliary cc and handle implicit flows to variables and metadata that could have been updated in untaken branches. When execution reaches a conditional command C, tuple $\langle \ell, W, A\rangle$ is pushed onto cc; when execution exits, C tuple $\langle \ell, W, A\rangle$ is popped.

– Element ℓ is the sensitivity of the guard e of C. Including ℓ in cc while taken branch C_t of C is executed signifies that the sensitivity of the context of C_t is the result of augmenting the sensitivity of the context of C with the sensitivity of guard e.

– Element W is the set $targetFlex(C_u)$ of all target flexible variables in the untaken branch C_u of C. If $w \in W$, then $T^i(w)$ for $i \geq 0$ could have been updated if untaken branch C_u were executed. To capture implicit flow from the context of C_u to $T^i(w)$, when execution exits C, sensitivity $T^{i+1}(w)$ is augmented with the sensitivity of the context of C_u, which is the same as the context of C_t.

– Element A is the set $targetAnchor(C_u)$ of all anchor variables in the untaken branch C_u. If A is not empty and if C_u would have been executed, then a block condition could have been evaluated, possibly causing that execution to be blocked. So, the reachability of a command following C might be influenced by whether C_u has been executed, and thus, it might be influenced by the context of C_u. So, in this case, when execution exits C auxiliary bc is augmented with the sensitivity of the context of C_u (which is the same as the context of C_t).

Figure 5.5 gives rules for executing sequences of commands. Rule (SEQF) asserts that the entire execution stops once an assignment is blocked (i.e., **block** has been generated by (ASGNAFAIL)).

Given a lattice $\mathcal{L}$, a command C, and a memory M initially healthy for ∞-*Enf*, $\mathcal{L}$, and C, function $trace_{\infty\text{-}Enf}(C, M)$ is defined by the operational semantics presented in Figures 5.3, 5.4, and 5.5. We have the following theorem.

Theorem 3. *∞-Enf is an enforcer on R for BNI.*

$$\text{(IF1)}\ \frac{M(e) \neq 0 \qquad W = \mathit{targetFlex}(C_2) \qquad A = \mathit{targetAnchor}(C_2) \qquad cc' = M(cc).\mathit{push}(\langle M(T(e)),\ W,\ A\rangle)}{\langle \textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M\rangle \to \langle C_1; \textbf{exit}, M[cc \mapsto cc']\rangle}$$

$$\text{(IF2)}\ \frac{M(e) = 0 \qquad W = \mathit{targetFlex}(C_1) \qquad A = \mathit{targetAnchor}(C_1) \qquad cc' = M(cc).\mathit{push}(\langle M(T(e)),\ W,\ A\rangle)}{\langle \textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M\rangle \to \langle C_2; \textbf{exit}, M[cc \mapsto cc']\rangle}$$

$$\text{(WL1)}\ \frac{M(e) \neq 0 \qquad cc' = M(cc).\mathit{push}(\langle M(T(e)),\ \emptyset,\ \emptyset\rangle)}{\langle \textbf{while } e \textbf{ do } C \textbf{ end}, M\rangle \to \langle C; \textbf{while } e \textbf{ do } C \textbf{ end}; \textbf{exit}, M[cc \mapsto cc']\rangle}$$

$$\text{(WL2)}\ \frac{M(e) = 0 \qquad W = \mathit{targetFlex}(C) \qquad A = \mathit{targetAnchor}(C) \qquad cc' = M(cc).\mathit{push}(\langle M(T(e)),\ W,\ A\rangle)}{\langle \textbf{while } e \textbf{ do } C \textbf{ end}, M\rangle \to \langle \textbf{exit}, M[cc \mapsto cc']\rangle}$$

$$bc' = \begin{cases} M(bc) \sqcup M(\lfloor cc \rfloor), & \text{if } M(cc).\mathit{top}.A \neq \emptyset \\ M(bc), & \text{otherwise} \end{cases}$$

$$\text{(EXIT)}\ \frac{M' = U(M, M(cc).\mathit{top}.W) \qquad cc' = cc.\mathit{pop}}{\langle \textbf{exit}, M\rangle \to \langle \textbf{stop}, M'[cc \mapsto cc', bc \mapsto bc']\rangle}$$

where $U(M, W) \triangleq M[\ \forall w \in W\colon \forall i \geq 1\colon T^i(w) \mapsto T^i(w) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)]$

Figure 5.4: Operational Semantics for conditional commands

$$\text{(SEQ1)}\ \frac{\langle C_1, M\rangle \to \langle \textbf{stop}, M'\rangle}{\langle C_1; C_2, M\rangle \to \langle C_2, M'\rangle}$$

$$\text{(SEQ2)}\ \frac{\langle C_1, M\rangle \to \langle C_1', M'\rangle \qquad C_1' \notin \{\textbf{stop}, \textbf{block}\}}{\langle C_1; C_2, M\rangle \to \langle C_1'; C_2, M'\rangle}$$

$$\text{(SEQF)}\ \frac{\langle C_1, M\rangle \to \langle \textbf{block}, M'\rangle}{\langle C_1; C_2, M\rangle \to \langle \textbf{block}, M'\rangle}$$

Figure 5.5: Operational Semantics for sequences

Proof. See Appendix B.2. □

5.5 Enforcer k-*Enf*

An enforcer that uses infinite label chains cannot be implemented with finite physical memory. But an infinite label chain can be approximated by a finite label chain. Infinite label chain $\Omega = \langle \ell_1, \ldots, \ell_k, \ell_{k+1}, \ell_{k+2}, \ldots \rangle$ is *conservatively approximated* by infinite label chain $\Omega' = \langle \ell_1, \ldots, \ell_k, \ell_k, \ell_k, \ldots \rangle$, where k^{th} label ℓ_k is infinitely repeated. It is a conservative approximation, because if Ω' allows a principal p assigned label ℓ to observe the i^{th} element of Ω', then Ω allows p to observe the i^{th} element of Ω, too (but not vice versa). This is because Ω and Ω' agree up to the k^{th} element and, for $i \geq k$, the i^{th} element in Ω' is at least as restrictive as the corresponding element in Ω due to monotonically decreasing label chains: $\ell_{k+1} \sqsubseteq \ell_k$, $\ell_{k+2} \sqsubseteq \ell_k$, ... Finite label chain with $m \geq 0$:

$$\Omega'' = \langle \ell_1, \ldots, \ell_k, \underbrace{\ell_k, \ldots, \ell_k}_{m} \rangle$$

also is a conservative approximation for Ω' (recall no observation is allowed for identifiers whose sensitivity is not defined). Consequently, an infinite label chain Ω can be approximated by finite label chain Ω''.

We employ such finite approximations to derive from ∞-*Enf* enforcer k-*Enf*. Enforcer k-*Enf* is based on the operational semantics rules of ∞-*Enf* to compute up to the k^{th} tag. Because rule (AsgnA) mentions $T^2(x)$, we require $k \geq 2$. In ∞-*Enf*, only (AsgnF) and function U actually refer to $T^i(x)$ for $i > 2$. So in k-*Enf* rule (AsgnF) and function U are modified to compute labels only for the first k tags. The new rules appear in Figure 5.6.

Enforcer k-*Enf* generates observations involving updates for up to the k^{th} tag. To generate an observation about an update to the k^{th} tag, k-*Enf* conservatively approximates the sensitivity of element $T^k(x)$ to be itself—that is,

$$\text{(AsgnF)}\ \frac{v_0 = M(e) \qquad \forall 1 \le i \le k\colon v_i = M(T^i(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)}{\langle w := e, M\rangle \to \langle \mathbf{stop}, M[\forall i\colon 0 \le i \le k\colon T^i(w) \mapsto v_i]\rangle}$$

$$U(M, W) \triangleq$$
$$M[\ \forall w \in W\colon \forall i\colon 1 \le i \le k\colon T^i(w) \mapsto T^i(w) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)].$$

Figure 5.6: Modified rules for k-*Enf*

$T^{k+1}(x) \triangleq T^k(x)$ (i.e., tag $T^k(x)$ represents tag $T^{k+1}(x)$). So, k-*Enf* actually is using label chains of length $n_{k\text{-}Enf} = k + 1$ and it conservatively approximates an infinite label chain $\Omega = \langle \ell_1, \ldots, \ell_k, \ell_{k+1}, \ell_{k+2}, \ldots \rangle$ that would have been computed by ∞-*Enf* with finite label chain $\Omega'' = \langle \ell_1, \ldots, \ell_k, \ell_k \rangle$.

Similar to ∞-*Enf*, enforcer k-*Enf* has $Aux_{k\text{-}Enf} = \{cc, bc\}$, $Init_{k\text{-}Enf}(cc) = \epsilon$, and $Init_{k\text{-}Enf}(bc) = \bot$. We prove the following theorem.

Theorem 4. *k-Enf is an enforcer on R for k-BNI($\mathcal{L}$), for any $\mathcal{L}$ and $k \ge 2$.*

Proof. See Appendix B.2. □

5.6 Discussion

5.6.1 Inference from Label Chains

Knowledge that label chains must be monotonically decreasing might be used by unauthorized principals to infer information about labels they are not supposed to read. Assume, for example, $T^{i+1}(w) = \mathsf{M}$ and $T^{i+2}(w) = \mathsf{L}$. Principals associated with label L are allowed to read $T^{i+1}(w)$ and learn that $T^{i+1}(w) = \mathsf{M}$. Because label chains are monotonically decreasing, these principals can then

deduce $T^{i+1}(w) \sqsubseteq T^i(w)$, which is $\mathsf{M} \sqsubseteq T^i(w)$. So, they learned something (i.e., $\mathsf{M} \sqsubseteq T^i(w)$) about $T^i(w)$, which they are not supposed to read (because $T^{i+1}(w) = \mathsf{M}$ and $\mathsf{M} \not\sqsubseteq \mathsf{L}$). But, knowing $\mathsf{M} \sqsubseteq T^i(w)$ does not leak anything about initial sensitive values, because the fact that label chains are monotonically decreasing is independent from the actual choice of initial sensitive values.

This kind of inference becomes possible whenever any invariant involving sensitive information is publicly known. In the example above, the invariant was that label chain are monotonically decreasing. In the next example, the publicly known invariant is the program code. Consider a program that includes assignment $h' := h * 2$, where h' and h are tagged with H. Because the program code is publicly known, all principals can infer that high variable h' stores an even value, even if these principals are not allowed to read h'. Such a program might still satisfy noninterference, which implies that these principals learn nothing about initial sensitive values by making the above inference. Perhaps a stronger property than noninterference, which would protect all sensitive values—not only initial sensitive values—could proscribe these inferences.

5.6.2 Reclassification

As a future work, we are planning to extend our family of enforcers to support arbitrary reclassification (e.g., declassification and classification) of both variables and tags. Reclassifying values in variables has been already studied and is well-understood. Reclassifying labels in tags has not been studied yet. And arbitrary reclassifications of labels could produce non-monotonically decreasing label chains. To remedy this, the semantics of label chains would

have to be changed. Here is one solution: define the sensitivity of $T^i(x)$ to be $T^{i+1}(x) \sqcup T^{i+2}(x) \sqcup \ldots$. Now, elements in a label chain may change in an arbitrary way, yet their sensitivity would still be monotonically decreasing.

5.7 Related Work

Dynamic Enforcement Mechanisms and Leaks. The formalization of dynamic information flow enforcement mechanisms dates back to Bell and La-Padula [15]. The research community realized then that dynamic enforcement mechanisms for information flow control might introduce leaks, which were not present in the program itself. Denning [73], for instance, explains that blocking an execution and reporting the underlying violation might leak sensitive information. Denning also gives examples where flow-sensitive labels generated by dynamic enforcement mechanisms violate TINI. Our k-*Enf* enforcers do not report the reason an execution terminates for exactly this reason. Also, they ensure that information is not leaked by observing flow-sensitive labels during normally terminated or blocked executions.

Label Chains of Length One. Most dynamic enforcement mechanisms use label chains of length one. *Purely* dynamic enforcement mechanisms that analyze only code that is executed and employ *no-sensitive-upgrade* (NSU) or *permissive-upgrade* (PU) (e.g., [18], [6],[31], [83], [7],[49]) satisfy TINI but not BNI, because they leak sensitive information when blocking an execution. Previous *hybrid* flow-sensitive enforcement mechanisms (e.g., [67], [30]), which employ program analysis before and during execution, do not satisfy BNI, either. There are

enforcement mechanisms (e.g., [2], [13]) that satisfy BNI, but they either handle only $\mathcal{L}_2 = \langle\{\mathsf{L}, \mathsf{H}\}, \sqsubseteq\rangle$, or lose permissiveness by tagging variables with the same labels at the end of conditional commands independent of which branch is actually taken.

Label Chains of Length Two. Certain dynamic enforcement mechanisms employ label chains of length two. Buiras et al. [26] propose a purely dynamic enforcement mechanism where fixed metalabels capture the implicit flow of information caused by conditional commands. The purely dynamic enforcement mechanism in [26] causes insecure executions to diverge instead of blocking. By enforcing only TINI, no security guarantee is given for executions that are forced to diverge (because TINI considers only finite traces).

By employing hybrid enforcement, Bedford et al. [12] use label chains where the second element is flow sensitive. The hybrid enforcement mechanism presented in [12] enforces TSNI on programs written in a while-language that supports references.

Unbounded Label Chains. Some enforcement mechanisms are expressive enough to support label chains of unbounded length. Zheng et al. [100] employ *dependent types* to tag a label with another label, thus forming chains of labels. Their approach can express a label recursively tagging itself, which can be seen as infinitely repeating the last label of a chain. Examples presented in [100] employ label chains of up to two elements (e.g., $\langle \ell, \bot \rangle$ and $\langle \ell, \ell \rangle$), but the authors acknowledge [100, §3.3.2] that longer chains are sensible. We explain in the next chapter why permissiveness can be lost with label chains of shorter length.

The enforcement mechanism presented in Zheng et al. [100] is mostly static, and thus it does not exhibit the leaks we examine (i.e., through flow sensitive labels and blocking executions). Specifically, label chains in [100] are given as input; they are not deduced by the enforcement mechanism. Conditions on these labels are inlined in the program by the programmer before execution. The program is then statically analyzed. If the analysis succeeds, then the program will satisfy TINI. So, a type-correct program can be safely executed until normal termination. Techniques presented in [100] involving label chains have been implemented in Jif [69, 68]. We believe that any framework that supports dependent types, such as [64] and [27], is likely able to express unbounded label chains.

Actions Other than Blocking. Dynamic enforcement mechanisms can take actions other than blocking when an unsafe command is about to be executed. Enforcement mechanisms presented in [61] and [30], which handle lattices with two elements (i.e., L and H), modify or skip the execution of an unsafe command. Similar to [26], the enforcement mechanism presented in [65] (which handles lattices with two elements) diverges the execution when reaching an unsafe command. Some enforcement mechanisms (e.g., [45],[48], [16],[85]) take no action, because they only update labels on variables; they do not perform any checks.

Certain purely dynamic enforcement mechanisms ([51], [87]) recover from exceptions caused by unsafe commands. The authors acknowledge that exceptions depend on flow-sensitive labels that tag variables and on the control flow of the program. These mechanisms avoid leaking sensitive information to exceptions through flow-sensitive labels and control flow. They enforce *error*

sensitive noninterference, which we believe is stronger that BNI. One of the techniques they employ is assigning the same labels to variables after conditional commands, independent of the branch that is taken. By doing so, some permissiveness might be lost against 2-Enf, which allows labels on variables to depend on taken branches.

Other Dynamic Enforcement Mechanisms. *Secure multi-execution* [38] (SME) is a flow-sensitive dynamic enforcement mechanism that enforces information flow labels by simultaneously executing the same program as many times as the number of labels. An execution that corresponds to a label ℓ sees the actual values, when these values are tagged with labels at most as restrictive as ℓ, and it sees dummy values, otherwise. Using *faceted* values [8], which is a tuple of values, each one corresponding to a different label, one can use one execution to simulate the set of executions generated by SME. These two approaches are permissive and do not introduce information leaks. However, the run-time overhead they introduce increases in proportion to the number of different labels that are used.

5.8 Summary

In this chapter we presented dynamic flow-sensitive enforcement mechanisms k-Enf that associate variables with label chains of arbitrary length. We proved that these mechanisms satisfy BNI, which stipulates that sensitive information is not leaked to principals that observe values in variables and label chains, during normally terminated and blocked executions.

CHAPTER 6

PERMISSIVENESS VERSUS CHAIN LENGTH

An independent criterion for assessing the value of an enforcer is permissiveness. Theorems are proved in this chapter to relate label chain length and permissiveness for k-Enf enforcers (§6.1) and for other enforcers (§6.2) that satisfy BNI. The relationships depend on initialization, threat model, and size of the lattice.

6.1 Permissiveness of k-Enf

An enforcer E' is *at least as permissive as* an enforcer E if, for all executions of each command, E' emits observations involving at least as many identifiers (in the same relative order) as E. This comparison involves deciding whether identifiers (i.e., variables and tags) that appear in a sequence θ of observations produced by E, also appear in a sequence θ' produced by E' (in the same relative order). We write $\theta \trianglelefteq \theta'$ iff

$$|\theta| \leq |\theta'| \ \wedge \ (\forall i\colon 1 \leq i \leq |\theta|\colon dom(\theta[i]) \subseteq dom(\theta'[i]))$$

where $\theta[i]$ is the ith observation in sequence θ. Notice, relation $\trianglelefteq$ does not depend on values being stored in variables because, by definition, enforcers E and E' are required to compute the same values while executing the same command.

We compare permissiveness of enforcers with respect to an underlying lattice and some identifiers of interest; we start this comparison with pairs of memories that satisfy some desired initialization condition, such as equality on initial values and label chains. Formally, an enforcer E' is at least as permissive as an

enforcer E for initialization condition ρ, underlying lattice $\mathcal{L}$, and identifiers up to the kth tag (i.e., T^k) with $k \geq 0$, writing $E \leq_\rho^{k,\mathcal{L}} E'$, iff

$$\begin{aligned} &\forall C, M, M' \text{:} \\ &\quad M \models \mathcal{H}_0(E, \mathcal{L}, C) \wedge M' \models \mathcal{H}_0(E', \mathcal{L}, C) \wedge \rho(M, M') \\ &\quad \Rightarrow (\forall \ell \in \mathcal{L} \text{: } trace_E(C, M)|_\ell^k \trianglelefteq trace_{E'}(C, M')|_\ell^k) \end{aligned} \tag{6.1}$$

Notice, the consequent of (6.1) holds iff labels deduced by E are at least as restrictive as labels deduced by E'. Relation $\leq_\rho^{k,\mathcal{L}}$ is a preorder (i.e., reflexive and transitive relation) on enforcers.

For convenience, we introduce abbreviations:

- $E <_\rho^{k,\mathcal{L}} E' \triangleq E \leq_\rho^{k,\mathcal{L}} E' \wedge E' \not\leq_\rho^{k,\mathcal{L}} E$
- $E \cong_\rho^{k,\mathcal{L}} E' \triangleq E \leq_\rho^{k,\mathcal{L}} E' \wedge E' \leq_\rho^{k,\mathcal{L}} E$

Notice that from (6.1) we can prove that if $\rho \Rightarrow \rho'$, then $E \leq_{\rho'}^{k,\mathcal{L}} E' \Rightarrow E \leq_\rho^{k,\mathcal{L}} E'$. Also, if $k \leq k'$, then $E \leq_\rho^{k',\mathcal{L}} E' \Rightarrow E \leq_\rho^{k,\mathcal{L}} E'$.

We now examine how lengths of label chains relate to the permissiveness of enforcers by comparing the permissiveness of enforcers k-Enf and $(k+1)$-Enf for $k \geq 2$. To perform this comparison, the initial memories considered by k-Enf and $(k+1)$-Enf for executing a command should agree on values in variables and on labels in tags, up to the kth. Define

$$M|^k \triangleq \{\langle T^i(x), M(T^i(x))\rangle \mid 0 \leq i \leq k \ \wedge \ x \in Var \ \wedge \ T^i(x) \in dom(M)\}$$

The desired initialization condition then is:

$$\rho_k(M, M') \quad \triangleq \quad M|^k = M'|^k$$

Initialization condition ρ_k allows a flexible variable w to be initially associated with label chains:

– $\Omega = \langle \ell_1, \ell_2, \ldots, \ell_k, \ell_{k+1}, \ell_{k+1} \rangle$ by $(k+1)$-*Enf*, where $\ell_{k+1} \sqsubset \ell_k$, and

– $\Omega' = \langle \ell_1, \ell_2, \ldots, \ell_k, \ell_k \rangle$ by k-*Enf*.

Here, Ω' is a conservative approximation of Ω. We will say that Ω exhibits a $(k+1)$-*decrease* because $T^{k+1}(w) \sqsubset T^k(w)$. Label chain Ω allows a principal p having label ℓ_{k+1} to observe $T^k(w)$ because $T(T^k(w)) = T^{k+1}(w) = \ell_{k+1}$. But Ω' does not allow p to observe $T^k(w)$ because, by definition of k-*Enf*, we have $T(T^k(w)) = T^k(w)$ and $\ell_{k+1} \sqsubset \ell_k$. Notice that for a label chain to exhibit a $(k+1)$-decrease, the labels in that chain should belong to a lattice with at least one non-bottom element.

Whenever $(k+1)$-*Enf* initially associates flexible variable w with a label chain Ω that exhibits a $(k+1)$-decrease, enforcer k-*Enf* is forced by initialization condition ρ_k to use conservative approximation Ω' for Ω. So, $(k+1)$-*Enf* is strictly more permissive than k-*Enf*.

Theorem 5. *k-Enf $<^{k,\mathcal{L}}_{\rho_k}$ $(k+1)$-Enf, for $k \geq 2$ and any lattice $\mathcal{L}$ with at least one non-bottom element.*

Proof. See Appendix B.5. □

So, longer label chains offer increased permissiveness for the k-*Enf* family of enforcers. From Theorem 5, we conclude by transitivity that k-*Enf* $<^{k,\mathcal{L}}_{\rho_k}$ ∞-*Enf*, for any $k \geq 2$.

There are cases where flexible variables initially store no information, and thus, they are initially associated with bottom-label chains (i.e., $\langle \bot, \ldots, \bot \rangle$). We say memory M is *conventionally initialized* when

$$\alpha_c(M) \triangleq \forall w \in Var_F\colon \forall i \geq 1\colon T^i(w) \in dom(M) \Rightarrow M(T^i(w)) = \bot.$$

We also define initialization condition

$$c(M, M') \triangleq \alpha_c(M) \wedge \rho_1(M, M')$$

which implies that two memories are conventionally initialized and agree on values in anchor variables and on the first labels of these anchor variables.

A result analogous to Theorem 5 does not hold when $<^{k,\mathcal{L}}_{\rho_k}$ is replaced with $<^{k,\mathcal{L}}_{c}$. With initialization condition c, label chains longer than two elements do not enhance the permissiveness of k-*Enf*. This is because, for conventional initialization c, k-*Enf* produces label chains where the second element is always repeated (e.g., $\langle \mathsf{H}, \mathsf{M}, \mathsf{M}, \ldots, \mathsf{M} \rangle$) (See Appendix B.2 for the proof) due to the conservative update of label chains of flexible variable induced by rules (AsgnF) in Figure 5.3 and (Exit) in Figure 5.4. There, all elements of label chains of the involved flexible variables are updated with the same label (i.e., the sensitivity of the context).

Theorem 6. *k-Enf $\cong^{k,\mathcal{L}}_{c}$ $(k+1)$-Enf for any lattice $\mathcal{L}$ and $k \geq 2$.*

Proof. See Appendix B.5. □

Threat model specifics affect the permissiveness of longer label chains, too. Consider a *weakened threat model* that allows observations of variables but disallows observations of updates to tags. Enforcers here would be expected to satisfy 0-BNI. Enforcer k-*Enf* satisfies k-BNI. So, k-*Enf* should also satisfy 0-BNI, because 0-BNI is implied by k-BNI.

Under our weakened threat model, the permissiveness of our enforcers is compared using relation $\leq^{0,\mathcal{L}}_{\rho_k}$, where superscript 0 indicates that only observations involving variables are considered for the comparison. Theorem 5 does

not apply because relation $<^{k,\mathcal{L}}_{\rho_k}$ considers observations up to the kth tag (due to superscript k) where $k \geq 2$. But we do have:

Theorem 7. *k-Enf $\cong^{0,\mathcal{L}}_{\rho_k}$ $(k+1)$-Enf for any lattice $\mathcal{L}$ and $k \geq 2$.*

Proof. See Appendix B.5. □

So, under our weakened threat model, the permissiveness of k-*Enf* does not improve by using label chains of length greater than two. It is tempting to posit a more general result stating that, under the weakened threat model, the permissiveness of all enforcers does not improve when using label chains of length greater than two, but we leave that for future work.

6.2 Other Enforcers

We now turn to the relation between permissiveness and label chain length for enforcers other than k-*Enf*. These other enforcers are categorized first based on the threat model they assume and then, based on particular properties they satisfy.

6.2.1 In the Strong Threat Model

Longer label chains help an enforcer E under the strong threat model provided there are executions of commands for which E produces label chains whose elements

(i) are not redundant—they are not a function of other elements in the same label chain, and

(ii) capture the real sensitivity of the elements they tag rather than conservatively approximating it.

Label chains that can be used as evidence for properties (i) and (ii) are formalized below as being k-*varying* and k-*precise*.

Two label chains $\langle \ell_1, \ell_2, \ldots, \ell_k \rangle$ and $\langle \ell'_1, \ell'_2, \ldots, \ell'_k \rangle$, whose elements belong to a lattice $\mathcal{L}$, are defined to be k-varying for $k \geq 2$ iff

$$(\forall i\colon 1 \leq i < k\colon \ell_i = \ell'_i) \ \wedge \ \ell_k \neq \ell'_k.$$

Notice that k-varying label chains cannot exist when $\mathcal{L}$ has only one element.

We now formalize k-precise. Consider an enforcer E, lattice $\mathcal{L}$, command C, and conventionally initialized memory M such that $M \models \mathcal{H}_0(E, \mathcal{L}, C)$. Assume trace $\tau = \mathit{trace}_E(C, M)$ *produces* label chain prefix $\Omega = \langle \ell_1, \ldots, \ell_n \rangle$ at some state $\tau[j]$ after an assignment to a flexible variable w:

$$\begin{aligned}
&\exists 1 < j \leq |\tau|\colon \exists w \in \mathit{Var}_F\colon \\
&\quad \tau[j-1] = \langle w := e; C_r, M_w \rangle, \\
&\quad \tau[j] = \langle C_r, M_r \rangle, \\
&\quad \forall i\colon 1 \leq i \leq n\colon T^i(w) \in \mathit{dom}(M_r) \ \wedge \ M_r(T^i(w)) = \ell_i.
\end{aligned}$$

Label chain Ω is k-precise (for $1 \leq k \leq n$) at $\tau[j]$ when for each enforcer E':
if

- E' satisfies $(k-1)$-BNI($\mathcal{L}$), and
- $E \leq_c^{k-1,\mathcal{L}} E'$,

then

– trace $\tau' = trace_{E'}(C, M')$ with $M' \models \mathcal{H}_0(E', \mathcal{L}, C)$ and $c(M, M')$ produces label chain $\langle \ell_1, \ldots, \ell_k \rangle$ at $\tau'[j]$.

So, if Ω is k-precise, then an enforcer E' that is sound (i.e., E' satisfies $(k-1)$-BNI($\mathcal{L}$)) and at least as permissive as E (i.e., $E \leq_c^{k-1,\mathcal{L}} E'$) cannot produce (at the same execution point) a label chain whose first k elements are less restrictive than those in Ω. Consequently, the first k elements of Ω captures the real sensitivity of the elements they tag.

For brevity, we say that E *produces some k-precise k-varying label chains* with elements in $\mathcal{L}$ iff there exist commands C, C' whose executions produce label chains Ω, Ω' such that:

– Ω is k-precise at the ith state of $trace_E(C, M)$, for some i and M with $M \models \mathcal{H}_0(E, \mathcal{L}, C)$,

– Ω' is k-precise at the jth state of $trace_E(C', M')$, for some j and M' with $M' \models \mathcal{H}_0(E, \mathcal{L}, C')$,

– Ω and Ω' are k-varying.

We show that longer label chains can offer increased permissiveness for an enforcer E, under the strong threat model, provided E produces some k-precise k-varying label chains. To do so, we compare such an enforcer E with an enforcer E' that approximates the kth element of each label chain as a function of the previous elements, instead of performing, for example, an analysis of the code. We say that E' produces $(k-1)$-*dependent label chains* for $k - 1 \geq 1$ iff E' is

an enforcer and

$$\forall x\colon \forall i > k-1\colon T^i(x) = f_{E'}(T(x), \ldots, T^{k-1}(x))$$

for some function $f_{E'}$. For example, k-*Enf* produces k-dependent label chains, because $T^k = T^{k+1}$. Notice that if an enforcer E' produces $(k-1)$-dependent label chains, then that mechanism cannot produce k-varying label chains.

An enforcer E' cannot both satisfy $(k-1)$-BNI and be at least as permissive as E. Assume for contradiction that E' satisfies $(k-1)$-BNI and is at least as permissive as E. Because the k-varying label chains produced by E are k-precise, E' should then produce the same k-varying label chains. But, we previously saw that if an enforcer E' produces k-varying label chains, then E' does not produce $(k-1)$-dependent label chains, which is a contradiction.

Theorem 8. (i) *For a lattice $\mathcal{L}$, for an enforcer E that satisfies $(k-1)$-BNI($\mathcal{L}$), with $k \geq 2$, and produces some k-precise k-varying label chains with elements in $\mathcal{L}$, and for an enforcer E' that produces $(k-1)$-dependent label chains,*

$$\textit{if } E \leq_c^{k-1,\mathcal{L}} E', \textit{ then } E' \textit{ does not satisfy } (k-1)\textit{-BNI}(\mathcal{L}).$$

(ii) *Enforcer E and lattice $\mathcal{L}$ exist.*

Proof. See Appendix B.6. □

For an enforcer E' that uses label chains of length $k-1$ (i.e., produces $(k-1)$-dependent label chains), Theorem 8 implies that E' cannot be at least as permissive as an enforcer E that uses label chains of length k. So, in contrast to Theorem 6, which stipulates that the family of k-*Enf* does not benefit from longer

label chains under conventional initialization, enforcer E in Theorem 8 does benefit. For example, given that the enforcement mechanism in [12] uses 2-dependent label chains, Theorem 8 implies that this enforcement mechanism loses permissiveness against an enforcer that produces 3-precise 3-varying label chains (e.g., 3-*Eopt*).

Theorem 8 (ii) asserts that such an enforcer E and lattice $\mathcal{L}$ exist. So, it is always possible to define, for each $k > 1$, an enforcer E that can produce k-precise k-varying label chains when executing some command C. Notice, k-*Enf* cannot produce k-precise k-varying label chains.

Witness E and $\mathcal{L}$ for Theorem 8 (ii). In the Appendix B.3, we describe k-*Eopt*, which is an enforcer that satisfies $(k-1)$-BNI and produces some k-precise k-varying label chains during the execution of a certain command C. C involves sequences of assignments and **if** commands whose branches contain only one assignment. Such **if** commands will be called *simple*. We construct k-*Eopt* by optimizing k-*Enf* for deducing k-precise k-varying labels during the execution of such C. The optimization is based on the following observation: ignoring context, if $T^i(w) = \bot$ at the end of both branches of a simple **if** command, then, at the end of that **if** command, $T^{i+1}(w)$ does not need to be updated with the sensitivity $T(e)$ of the guard of that **if** command. This optimization enables k-*Eopt* to produce some k-precise k-varying label chains.

6.2.2 In the Weakened Threat Model

In the weakened threat model, label chains of length two can offer enhanced permissiveness compared to label chains of length one: the second label in a label chain enables the decision to block assignment commands to be more permissive (previous theorems consider label chains with at least two elements). To illustrate, it suffices to consider *anchor-tailed* commands, which are a sequence $C; C'$ of commands where C does not involve any assignment to anchor variables and C' is a sequence of assignments to anchor variables. The second label in a label chain allows the decision to block an assignment in an anchor-tailed command to be more permissive.

Let $G^E_{a:=e}$ denote the condition used by an enforcer E for blocking an assignment $a := e$ to anchor variable a when execution reaches state $\langle a := e; C', M' \rangle$ in a trace $trace_E(C, M)$. $G^E_{a:=e}$ is a boolean expression on the domain of M' such that

$$M'(G^E_{a:=e}) \quad \Leftrightarrow \quad \langle a := e; C', M' \rangle \rightarrow \langle C', M'' \rangle \text{ is a subtrace of } trace_E(C, M).$$

For assignment $a := e$ in an anchor-tailed command, $G^E_{a:=e}$ may depend on label chains of variables in

- the assignment $a := e$ itself (to capture explicit flows), and
- the context of that assignment (to capture implicit flows). By definition of anchor-tailed commands, such an assignment is not encapsulated in any conditional command, but it may follow other assignments to anchor variables. So, the context of $a := e$ only references variables mentioned in assignments to anchor variables that precede $a := e$.

Let $V_{a:=e}$ denote the above set of variables.

We say $G^E_{a:=e}$ is a *k-dependent condition* for $a := e$ in an anchor-tailed command iff $G^E_{a:=e}$ depends at most on the first k elements of the label chains of variables in $V_{a:=e}$

$$G^E_{a:=e} = f_E(\{T^i(x) \mid x \in V_{a:=e} \ \wedge \ 1 \leq i \leq k\}),$$

for some function f_E. For example, 2-*Enf* uses 2-dependent $G_{a:=e}$.

We can now return to our original goal, which is to show how the second label in a label chain makes the decision to block assignment commands more permissive. Theorem 9 states that if an enforcer E uses 1-dependent $G^E_{a:=e}$, then this enforcer cannot both satisfy 0-BNI and be at least as permissive as 2-*Enf*. Here is why. E does not compute the sensitivity of labels referenced by block condition $G^E_{a:=e}$, and thus, E does not compute the sensitivity of the information conveyed by its decision to block a certain assignment $a := e$. In an effort to satisfy 0-BNI and prevent leaking sensitive information, E must decide to always block $a := e$, even though in some executions that assignment is safe and allowed by 2-*Enf*.

Theorem 9. *For an enforcer E and lattice $\mathcal{L}_3$, if $G^E_{a:=e}$ is 1-dependent and 2-Enf $\leq_c^{0,\mathcal{L}_3}$ E, then E does not satisfy 0-BNI($\mathcal{L}_3$).*

Proof. See Appendix B.6. □

So, for the weakened threat model, there is improved permissiveness when using two (instead of one) labels for each variable.

Since most dynamic enforcement mechanisms proposed in the past satisfy TINI, the reader might wonder whether Theorem 9 still holds when 0-BNI is

replaced by TINI. Under the weakened threat model, there are enforcers (e.g., $E_{\mathsf{H,L}}$ in the next section) that use 1-dependent $G^{E}_{a:=e}$, are at least as permissive as 2-Enf and they do satisfy TINI. So, Theorem 9 does not hold when 0-BNI is replaced by TINI.

Familiar Two-level Lattice

Some papers build a theory based on a two-level lattice $\mathcal{L}_2 \triangleq \langle\{\mathsf{L},\mathsf{H}\},\sqsubseteq\rangle$ with $\mathsf{L} \sqsubset \mathsf{H}$, expecting results that will extend to arbitrary lattices. In this section, we give a result expressed in terms of $\mathcal{L}_2$ that does not hold for more complex lattices. Thus, generalizing from $\mathcal{L}_2$ to arbitrary lattices is not always a sound presumption.

Consider $\mathcal{L}_2$ along with the weakened threat model. Previous work [56] proposed a flow-sensitive enforcement mechanism that uses only one label per variable. We denote that enforcement mechanism by $E_{\mathsf{H,L}}$, which is derived from k-Enf by associating each variable with only one tag (i.e., $E_{\mathsf{H,L}}$ does not use $T^i(x)$ when $i > 1$). Figure 6.1 shows the modified rules for $E_{\mathsf{H,L}}$. We prove that $G_{a:=e}$ defined in Figure 6.1 is 1-dependent (Theorem 10).

$E_{\mathsf{H,L}}$ ensures that the sensitivity of each tag $T(w)$ is always L (so there is no need to explicitly keep track of $T^2(w)$). The only way to encode sensitive information (i.e., tagged with H) in $T(w)$ is if $T(w)$ is updated with different labels in a conditional command that has a sensitive (tagged with H) guard. But, if the sensitivity of the guard is H, then due to function U in Figure 6.1, tag $T(w)$ will always be updated to H at the end of that conditional command, because $M(\lfloor cc \rfloor) = \mathsf{H}$. So, $T(w)$ will reveal no information about the value of

that sensitive guard. Thus, the sensitivity of $T(w)$ is L.

Define function $trace_{E_{\mathsf{H,L}}}(C, M)$ to map command C and memory M with $M \models \mathcal{H}_0(E_{\mathsf{H,L}}, \mathcal{L}, C)$ to the entire trace that starts with state $\langle C, M\rangle$. We have $n_{E_{\mathsf{H,L}}} = 1$, $Aux_{E_{\mathsf{H,L}}} = \{cc, bc\}$, $Init_{E_{\mathsf{H,L}}}(cc) = \epsilon$, and $Init_{E_{\mathsf{H,L}}}(bc) = \perp$.

Theorem 9 does not hold when $\mathcal{L}_3$ is replaced with $\mathcal{L}_2$ and E is $E_{\mathsf{H,L}}$. Instead, Theorem 10 below holds; it states that $E_{\mathsf{H,L}}$ satisfies 0-BNI and is strictly more permissive than 2-*Enf* only when $\mathcal{L}_2$ is used. We are also not aware of an enforcement mechanism that uses label chains of length one, enforces $\mathcal{L}_2$, satisfies 0-BNI, and is at least as permissive as $E_{\mathsf{H,L}}$.

Theorem 10. *Enforcer $E_{\mathsf{H,L}}$ uses 1-dependent $G_{a:=e}$, satisfies 0-BNI($\mathcal{L}_2$), and satisfies 2-Enf $<_c^{0,\mathcal{L}_2}$ $E_{\mathsf{H,L}}$.*

Proof. See Appendix B.6. □

So, Theorem 10 contradicts expectations that longer label chains offer increased permissiveness. Moreover, this theorem is an example where a result expressed in terms of $\mathcal{L}_2$ does not necessarily generalize for arbitrary lattices.

Notice, though, that $E_{\mathsf{H,L}}$ does not satisfy 0-BNI for arbitrary lattices. For example, consider (1.8), which employs $\mathcal{L}_3$. Based on rules in Figure 6.1 and rules (IF),(SEQ) in §5.4.3, $E_{\mathsf{H,L}}$ executes (1.8) as described in (i) and (ii) in §1.4. So, executing (1.8) under $E_{\mathsf{H,L}}$ leaks sensitive $m > 0$ to principals observing non-sensitive variable l, and thus, 0-BNI is not satisfied. So, $E_{\mathsf{H,L}}$ illustrates that an enforcer designed to enforce two-level lattices cannot necessarily enforce arbitrary lattices.

$$\text{(AsgnA)}\ \frac{v = M(e) \qquad G_{a:=e} \qquad \ell = M(\lfloor cc \rfloor) \sqcup M(bc)}{\langle a := e, M\rangle \rightarrow \langle \mathbf{stop}, M[a \mapsto v, bc \mapsto \ell]\rangle}$$

$$\text{(AsgnAFail)}\ \frac{v = M(e) \qquad \neg G_{a:=e} \qquad \ell = M(\lfloor cc \rfloor) \sqcup M(bc)}{\langle a := e, M\rangle \rightarrow \langle \mathbf{block}, M[bc \mapsto \ell]\rangle}$$

$$\text{(AsgnF)}\ \frac{v_0 = M(e) \qquad v_1 = M(T(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)}{\langle w := e, M\rangle \rightarrow \langle \mathbf{stop}, M[w \mapsto v_0, T(w) \mapsto v_1]\rangle}$$

$$U(M, W) \triangleq M[\ \forall w{\in}W\colon T(w) \mapsto T(w) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)]$$

$$G_{a:=e} \text{ is } M(T(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq M(T(a))$$

Figure 6.1: Modified rules for $E_{\mathsf{H,L}}$

6.3 Discussion

6.3.1 Extending Finite to Infinite Label Chains

Infinitely repeating the last element of the finite label chain is one approach for safely extending finite label chains into infinite label chains. But this approach is not panacea; it might generate less permissive label chains than others satisfying BNI. As a first example, consider an anchor variable being associated with one-element label chain $\langle \ell \rangle$. Extending that one-element label chain by infinitely repeating ℓ would be safe, but it would lead to a less permissive infinite label chain that, say, $\langle \ell, \bot, \bot, \ldots \rangle$ (used in §5.1). As a second example, recall §6.2.2, where we established that if labels belong to a two-level lattice, then the one-element label chain $\langle \ell \rangle$ of every variable can be extended into infinite label chain $\langle \ell, \mathsf{L}, \mathsf{L}, \ldots \rangle$. Consequently, the rule for extending label chains may depend on the enforcement mechanisms, assumptions on variables, and on the lattice of labels.

6.3.2 Number of Non-bottom Labels in Chains

Under the conventional initialization, flexible variables are initially associated with label chains whose elements equal to bottom label $\bot$. Also, anchor variables are always associated with label chains whose first element is not $\bot$, but all other elements are $\bot$. So, one might wonder whether the number of non-bottom labels increases as a program is executed under an enforcer.

First we consider a case where an enforcer should fill the entire label chain of a variable with non-bottom elements in order to achieve soundness. Consider:

$$\textbf{if } m > 0 \textbf{ then } w := 1;\ x := 3 \textbf{ else } x := 0;\ w := 4 \textbf{ end}$$

where $T(m) = \mathsf{M}$. Because sensitive information $m > 0$ implicitly flows to flexible variables w, x and their label chains, it is safe to associate w and x with label chain $\langle \mathsf{M}, \mathsf{M}, \dots \rangle$, as k-*Enf* actually does. Assume, for contradiction, that an enforcer deduces label chains of length k for w and x, where the label chain contains at least one $\bot$. We show that k-BNI is not satisfied. That is, assume:

- $T(w) = T^2(w) = \ldots = T^i(w) = \mathsf{M}$ and $T^{i+1}(w) = T^{i+2}(w) = \ldots = \bot$,
- $T(x) = T^2(x) = \ldots = T^j(x) = \mathsf{M}$ and $T^{j+1}(x) = T^{j+2}(x) = \ldots = \bot$

for $1 \leq i, j < k$. Then principals assigned $\bot$ make different observations depending on the branch that is taken, and thus, sensitive $m > 0$ is leaked:

- If $m > 0$ holds, then principals assigned $\bot$ first observe $\langle T^i(w), \mathsf{M} \rangle$ and then $\langle T^j(x), \mathsf{M} \rangle$.
- If $m > 0$ does not hold, then principals assigned $\bot$ first observe $\langle T^j(x), \mathsf{M} \rangle$ and then $\langle T^i(w), \mathsf{M} \rangle$.

So, k-BNI is not satisfied. This leak is avoided if at least w or x is associated with a label chain without $\bot$. Assume:

- $T^i(w) = \mathsf{M}$, for all $i > 0$,
- $T(x) = \mathsf{M}$, $T^2(x) = \ldots = \bot$.

Then, in all executions, principals assigned $\bot$ make the same observations (i.e., $\{\langle T(x), \mathsf{M}\rangle, \langle T^2(x), \bot\rangle, \ldots\}$), and thus, m is not leaked.

In other cases, an enforcer gradually fills label chains of variables with distinct non-bottom elements. Consider enforcer k-*Eopt* (see Appendix B.3) and the following program:

$$\begin{array}{l}
\textbf{if } a_1 > 1 \textbf{ then } w_1 := 0 \textbf{ else } w_1 := 1 \textbf{ end};\\
\textbf{if } a_2 > 1 \textbf{ then } w_2 := w_1 \textbf{ else } w_2 := 2 \textbf{ end};\\
\textbf{if } a_3 > 3 \textbf{ then } w_3 := w_2 \textbf{ else } w_3 := 3 \textbf{ end};\\
\ldots\\
\textbf{if } a_i > i \textbf{ then } w_i := w_{i-1} \textbf{ else } w_i := i \textbf{ end};\\
\ldots
\end{array}$$

Figure 6.2 gives the chains associated with each w_i at the end of the i^{th} **if** command, assuming an execution where each $a_i > i$ holds. Notice that the number of non-bottom labels increases as successive **if** commands are executed. Notice, also, that all generated label chains are monotonically decreasing. If $\bot \sqsubset \ldots \sqsubset T(a_i) \sqsubset \ldots \sqsubset T(a_2) \sqsubset T(a_1)$, then these label chains become strictly monotonically decreasing. So, any attempt to approximate these label chains with shorter ones would lose precision and thus harm permissiveness.

w_1	$\langle T(a_1), \bot, \bot, \bot, \ldots \rangle$
w_2	$\langle T(a_1) \sqcup T(a_2), T(a_2), \bot, \bot, \ldots \rangle$
w_3	$\langle T(a_1) \sqcup T(a_2) \sqcup T(a_3), T(a_2) \sqcup T(a_3), T(a_3), \bot \ldots \rangle$
$\ldots$	$\ldots$
w_i	$\langle T(a_1) \sqcup \ldots \sqcup T(a_i), T(a_2) \sqcup \ldots \sqcup T(a_i), \ldots, T(a_i), \bot, \ldots \rangle$

Figure 6.2: Label chains for w_i

6.4 Related Work

Russo et al. [77] study trade-offs between static and dynamic security analysis. They prove the impossibility of a purely dynamic information-flow monitor that satisfies TINI and accepts programs certified by the Hunt and Sands classical flow-sensitive static analysis [52]. They first define some basic semantics (Figure 4 in [77]) that purely dynamic information-flow monitor may extend. Then, they introduce properties for the purely dynamic enforcement mechanisms they consider (i.e., *not look ahead*, *not look aside*). Their main impossibility theorem has the same style as our Theorem 9: an enforcement mechanism with the above properties cannot both satisfy TINI and be at least as permissive as [52]. Our Theorem 8 adopts a more general style, since it compares the permissiveness of any two enforcement mechanisms that satisfy particular properties; it does not assume that one of the two is a fixed known enforcement mechanism. And our permissiveness relation $\leq_{\rho}^{k,\mathcal{L}}$ is more general than the one presented in [77], because it is defined for any two enforcers and handles arbitrary lattices (not only $\mathcal{L}_2$) and initialization conditions.

Bielova et al. [19] present a taxonomy of five representative flow-sensitive information flow enforcement mechanisms (no-sensitive-upgrade, permissive-upgrade, hybrid monitor, secure multi-execution, and multiple facets), in terms

of soundness and *transparency*, which stipulates that enforcement mechanisms do not alter the semantics of safe executions. The authors introduce *Termination-Aware Noninterference* (TANI) as a soundness goal, and it is expressed in terms of knowledge semantics. If an enforcement mechanism that decides to diverge the execution of an unsafe command satisfies TANI, then this mechanism does not leak sensitive information by making this decision. The theoretical framework considered in [19] assumes labels are taken from $\mathcal{L}_2$. Also, it assumes that a terminating execution produces one output, at the end, tagged with L (all principals can read it); if an execution diverges, no output is produced. So, TANI guarantees that dynamic enforcement mechanisms do not introduce leaks when deciding to diverge executions in this particular framework. Because only $\mathcal{L}_2$ is considered in [19], our section 6.2.2 explains that there is no danger for these mechanisms to encode sensitive information in the flow-sensitive labels.

Bielova et al. [19] use several metrics for comparing information flow enforcement mechanisms: precision, permissiveness, and transparency. The comparison results proved in [19] use a style similar to our Theorems 5, 6, 7, and 10, which compare specific enforcement mechanisms.

6.5 Summary

We formalized conditions under which permissiveness is harmed when an enforcer uses shorter label chains. We believe the results presented in this chapter can explain choices made by past work in dynamic information flow control and illustrate how these choices might affect permissiveness.

CHAPTER 7

CONCLUSION

This thesis studied two extensions of classical information flow labels: RIF labels and label chains. We established that the resulting increased expressiveness of these formulations can maintain soundness and increase permissiveness.

RIF labels (chapters 2-4) specify arbitrary reclassifications triggered by operations. With RIF labels, restrictions on derived values can be fewer than those imposed on initial values, and thus permissiveness is enhanced. Also, restrictions on derived values can be increased over those imposed on initial values, and thus soundness is maintained. Of course, supporting labels with increased expressiveness, such as RIF labels, could complicate enforcement. In this thesis, we managed to keep the enforcement mechanism of RIF labels simple and we presented two checkable families of RIF labels where relation $\sqsubseteq$ can be decidable. Our PWNI is a first extension of noninterference to handle arbitrary reclassifications (specified by RIF labels). But as discussed in §2.2, PWNI is conservative.

Label chains (chapters 5-6) enable us to reason about the sensitivity of flow-sensitive labels employed by dynamic enforcement mechanisms. We showed that there are cases where longer label chains can increase permissiveness. As a next step, we plan to support reclassification of any element in a label chain. This step can be achieved by using RIF labels as elements of label chains. Employing *RIF label chains* would give us the opportunity to study RIF labels in a dynamic flow-sensitive enforcement mechanism. It would also force us to interpret label chains in terms of arbitrary restrictions—not only in terms of confidentiality restrictions.

APPENDIX A

PROOFS FOR RIF LABELS

A.1 Type-correctness implies PWNI

We first define $\mathbb{T}(\lambda, \mathcal{C})$. Consider a deterministic generator G of fresh variables. $G.init$ sets G to its initial state. $G.fresh([\mathcal{E}]_{\bar{\mathsf{f}}})$ returns the next fresh variable h. Variable h stores an arbitrary value (of the same type as $[\mathcal{E}]_{\bar{\mathsf{f}}}$) and $\Gamma(h) = \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}})$. $\mathbb{T}(\lambda, \mathcal{C})$ is deterministic (assuming each T is called in the order that it appears in the definitions below).

$$\mathbb{T}(\lambda, \mathcal{C}) \triangleq T(\lambda, \mathcal{C}, G.init)$$

$$\begin{aligned}
T(\lambda, x := [\mathcal{E}]_{\bar{\mathsf{f}}}, G) &\triangleq x := S(\lambda, [\mathcal{E}]_{\bar{\mathsf{f}}}, G) \\
T(\lambda, \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}_e \textbf{ end}, G) &\triangleq \textbf{if } S(\lambda, [\mathcal{E}]_{\bar{\mathsf{f}}}, G) \textbf{ then } T(\lambda, \mathcal{C}_t, G) \\
&\qquad \textbf{else } T(\lambda, \mathcal{C}_e, G) \\
T(\lambda, \textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_w \textbf{ end}, G) &\triangleq \textbf{while } S(\lambda, [\mathcal{E}]_{\bar{\mathsf{f}}}, G) \textbf{ do } T(\lambda, \mathcal{C}_w, G) \textbf{ end} \\
T(\lambda, \mathcal{C}_1; \mathcal{C}_2, G) &\triangleq T(\lambda, \mathcal{C}_1, G); T(\lambda, \mathcal{C}_2, G)
\end{aligned}$$

$$S(\lambda, [\mathcal{E}]_{\bar{\mathsf{f}}}, G) \triangleq \begin{cases} G.fresh([\mathcal{E}]_{\bar{\mathsf{f}}}), & \text{if } \Gamma(\mathcal{E}) \sqsubseteq \lambda \wedge \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda \\ [\mathcal{E}]_{\bar{\mathsf{f}}}, & \text{otherwise} \end{cases}$$

We now proceed with the proofs.

Theorem 1. $\Gamma, \lambda_\kappa \vdash \mathcal{C} \;\Rightarrow\; PWNI(\mathcal{C})$

Proof. Assume $\Gamma, \lambda_\kappa \vdash \mathcal{C}$. To prove $PWNI(\mathcal{C})$, assume $\tau = \Upsilon(\mathbb{T}(\lambda, \mathcal{C}), \mathcal{M})$ and $\tau' = \Upsilon(\mathbb{T}(\lambda, \mathcal{C}), \mathcal{M}')$. We prove $dpNI(\lambda, \tau, \tau')$. From Lemma 3 and $\Gamma, \lambda_\kappa \vdash \mathcal{C}$, we get $\Gamma, \lambda_\kappa \vdash \mathbb{T}(\lambda, \mathcal{C})$. From Lemma 1, we then get $dpNI(\lambda, \tau, \tau')$. □

Lemma 1. *If* $\tau = \Upsilon(\mathcal{C}, \mathcal{M})$, $\tau' = \Upsilon(\mathcal{C}, \mathcal{M}')$, *and* $\Gamma, \lambda_\kappa \vdash \mathcal{C}$, *then* $dpNI(\lambda, \tau, \tau')$, *for all* λ.

Proof. For what follows, we define π to be a λ-*dpiece* iff

- $\pi = \langle \mathcal{C}_1, \mathcal{M}_1 \rangle \rightarrow \ldots \rightarrow \langle \mathcal{C}_n, \mathcal{M}_n \rangle$ with $n \geq 2$,
- $\forall 1 < i < |\pi|$: $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \pi[i].\mathcal{C}) = \emptyset$, and
- $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \pi[n].\mathcal{C}) \neq \emptyset$ or $\pi[n].\mathcal{C} = \bullet$.

Notice that a λ-piece of a trace is a maximal λ-dpiece that starts at the first state of that trace or at a downgrade.

We also define SPNI(λ, π) to stipulate that there is no illicit λ-flow within λ-dpiece π.

SPNI(λ, π): Let $\pi = \langle \mathcal{C}, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}_d, \mathcal{M}_d \rangle$ be a λ-dpiece. For all λ-dpieces $\pi' = \langle \mathcal{C}, \mathcal{M}' \rangle \xrightarrow{*} \langle \mathcal{C}'_d, \mathcal{M}'_d \rangle$ such that

$$\mathcal{M} =_\lambda \mathcal{M}' \qquad (SPNI-H1)$$

$$\forall \mathcal{E} \in \Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}):\ \mathcal{M}(\mathcal{E}) = \mathcal{M}'(\mathcal{E}) \qquad (SPNI-H2)$$

the following should hold

$$\pi =_\lambda \pi', \qquad (SPNI-C1)$$

$$\mathcal{C}_d = \mathcal{C}'_d, \qquad (SPNI-C2)$$

$$\mathcal{M}_d =_\lambda \mathcal{M}'_d. \qquad (SPNI-C3)$$

We prove that if $\tau = \Upsilon(\mathcal{C}, \mathcal{M})$, $\tau' = \Upsilon(\mathcal{C}, \mathcal{M}')$, and $\Gamma, \lambda_\kappa \vdash \mathcal{C}$, then $dpNI(\lambda, \tau, \tau')$, for all λ. By definition, τ and τ' are not empty. Thus, we can

apply induction on the number of λ-pieces that comprise τ and τ' using the following induction hypothesis IH. There, define trace τ_i to be a *strict λ-piece-suffix* of τ iff τ_i is a strict suffix of τ and the first state of τ_i is the first state of a λ-piece in τ.

IH: If τ_i is a strict λ-piece-suffix of τ, τ_i' is a strict λ-piece-suffix of τ', $\tau_i.\mathcal{C} = \tau_i'.\mathcal{C}$, and $\Gamma, \lambda_\kappa \vdash \tau_i.\mathcal{C}$, then $dpNI(\lambda, \tau_i, \tau_i')$, for all λ.

Base case: τ and τ' are λ-pieces.

So, $\overset{\rightarrow\lambda}{\tau} = \tau$, $\overset{\rightarrow\lambda}{\tau'} = \tau'$, $\overset{\lambda\rightarrow}{\tau} = \epsilon$, $\overset{\lambda\rightarrow}{\tau'} = \epsilon$.

We prove $dpNI(\lambda, \tau, \tau')$ assuming $\Gamma, \lambda_\kappa \vdash \mathcal{C}$ holds. Because $\overset{\rightarrow\lambda}{\tau} = \tau$, $\overset{\rightarrow\lambda}{\tau'} = \tau'$, $\overset{\lambda\rightarrow}{\tau} = \epsilon$, $\overset{\lambda\rightarrow}{\tau'} = \epsilon$, and because $dpNI(\lambda, \epsilon, \epsilon)$ holds by definition, we have that $dpNI(\lambda, \tau, \tau')$ is

$$
\begin{aligned}
&(\forall \mathcal{E} \in \Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \tau.\mathcal{C}):\ \tau.\mathcal{M}(\mathcal{E}) = \tau'.\mathcal{M}(\mathcal{E})) \\
&\wedge\ \tau.\mathcal{M} =_\lambda \tau'.\mathcal{M} \\
&\Rightarrow\ \ \tau =_\lambda \tau'
\end{aligned}
$$

So, to prove $dpNI(\lambda, \tau, \tau')$ it suffices to assume

$$(\forall \mathcal{E} \in \Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \tau.\mathcal{C}):\ \tau.\mathcal{M}(\mathcal{E}) = \tau'.\mathcal{M}(\mathcal{E})) \tag{A.1}$$

$$\tau.\mathcal{M} =_\lambda \tau'.\mathcal{M} \tag{A.2}$$

and prove $\tau =_\lambda \tau'$.

Because τ and τ' are λ-pieces, then they are λ-dpieces, too. From Lemma 2, and because τ is a λ-dpiece and $\Gamma, \lambda_\kappa \vdash \mathcal{C}$, we get $SPNI(\lambda, \tau)$. From (A.1) and (A.2), and because τ' is a λ-dpiece, we then get $\tau =_\lambda \tau'$.

Induction case: τ and τ' are not both λ-pieces.

We show $dpNI(\lambda, \tau, \tau')$. By definition of $dpNI$, it suffices to assume

$$(\forall \mathcal{E} \in \Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \tau.\mathcal{C}):\ \tau.\mathcal{M}(\mathcal{E}) = \tau'.\mathcal{M}(\mathcal{E})) \tag{A.3}$$

$$\tau.\mathcal{M} =_{\lambda} \tau'.\mathcal{M} \tag{A.4}$$

and show

$$\tau[|\overset{\rightarrow\lambda}{\tau}|].\mathcal{C} = \tau'[|\overset{\rightarrow\lambda}{\tau'}|].\mathcal{C} \tag{A.5}$$

$$\overset{\rightarrow\lambda}{\tau} =_{\lambda} \overset{\rightarrow\lambda}{\tau'} \tag{A.6}$$

$$dpNI(\lambda, \overset{\lambda\rightarrow}{\tau}, \overset{\lambda\rightarrow}{\tau'}) \tag{A.7}$$

We first show (A.5) and (A.6). We have that $\overset{\rightarrow\lambda}{\tau}$ and $\overset{\rightarrow\lambda}{\tau'}$ are λ-dpieces. So, from $\Gamma, \lambda_{\kappa} \vdash \mathcal{C}$ and Lemma 2, we then get $SPNI(\lambda, \overset{\rightarrow\lambda}{\tau})$. Consequently, from (A.3) and (A.4), we then get (A.5) and (A.6).

We now show (A.7). From (A.5) and the hypothesis of this case, we get that both τ and τ' contain at least two λ-pieces, and thus $\overset{\lambda\rightarrow}{\tau} \neq \epsilon$ and $\overset{\lambda\rightarrow}{\tau'} \neq \epsilon$. From $\Gamma, \lambda_{\kappa} \vdash \mathcal{C}$ and Lemma 4, we get

$$\Gamma, \lambda_{\kappa} \vdash \overset{\lambda\rightarrow}{\tau}.\mathcal{C} \tag{A.8}$$

From (A.5) and definition of λ-pieces, we have $\overset{\lambda\rightarrow}{\tau}.\mathcal{C} = \overset{\lambda\rightarrow}{\tau'}.\mathcal{C}$. From IH and (A.8), we then get $dpNI(\lambda, \overset{\lambda\rightarrow}{\tau}, \overset{\lambda\rightarrow}{\tau'})$. So, we get (A.7).

□

Lemma 2. *For each $\mathcal{C}$, $\mathcal{M}$, and λ, and for each λ-dpiece $\pi = \langle \mathcal{C}, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}_d, \mathcal{M}_d \rangle$:*

$$\Gamma, \lambda_{\kappa} \vdash \mathcal{C} \Rightarrow SPNI(\lambda, \pi).$$

Proof. Consider a memory $\mathcal{M}$, a mapping Γ, a context-type λ_{κ}, and a label λ. Define $\mathcal{C}' \in \mathcal{C}$ to hold iff $\mathcal{C}'$ is a subcommand of $\mathcal{C}$. We use structural induction

on $\mathcal{C}$, using the following induction hypothesis:

IH: For each $\hat{\mathcal{C}} \in \mathcal{C}$, $\hat{\mathcal{M}}$, $\hat{\lambda}$, and for each $\hat{\lambda}$-dpiece $\hat{\pi} = \langle \hat{\mathcal{C}}, \hat{\mathcal{M}} \rangle \xrightarrow{*} \langle \hat{\mathcal{C}}_d, \hat{\mathcal{M}}_d \rangle$:

$$\hat{\Gamma}, \hat{\lambda_\kappa} \vdash \hat{\mathcal{C}} \Rightarrow \mathit{SPNI}(\hat{\lambda}, \hat{\pi}).$$

For what follows, we define $\mathcal{M}|_\lambda \triangleq \{\langle x, \mathcal{M}(x)\rangle \mid \Gamma(x) \sqsubseteq \lambda\}$. Notice that $\mathcal{M}|_\lambda = \mathcal{M}'|_\lambda$ holds iff $\mathcal{M} =_\lambda \mathcal{M}'$ holds.

1. $\mathcal{C}$ is **skip**:

   From rule SKIP, we have:

   $$\pi = \langle \mathbf{skip}, \mathcal{M}\rangle \rightarrow \langle \bullet, \mathcal{M}\rangle. \tag{A.9}$$

   By definition, π is an λ-dpiece. Also, $\Delta^-_{\hat{\mathcal{C}}}(\lambda, \mathbf{skip}) = \emptyset$ and

   $$\Gamma, \lambda_\kappa \vdash \mathbf{skip}. \tag{A.10}$$

   We prove $\mathit{SPNI}(\lambda, \pi)$.

   Using SKIP, consider:

   $$\pi' = \langle \mathbf{skip}, \mathcal{M}'\rangle \rightarrow \langle \bullet, \mathcal{M}'\rangle, \tag{A.11}$$

   where

   $$\mathcal{M} =_\lambda \mathcal{M}'. \tag{A.12}$$

   By definition, π' is an λ-dpiece.

   We prove $(\mathit{SPNI}-\mathit{C1})$, $(\mathit{SPNI}-\mathit{C2})$, and $(\mathit{SPNI}-\mathit{C3})$. We have $\pi|_\lambda = \epsilon$ and $\pi'|_\lambda = \epsilon$. So, $\pi =_\lambda \pi'$, and thus $(\mathit{SPNI}-\mathit{C1})$ holds. From (A.9), (A.11), and (A.12), we get that $(\mathit{SPNI}-\mathit{C2})$ and $(\mathit{SPNI}-\mathit{C3})$ hold.

2. $\mathcal{C}$ is $x := [\mathcal{E}]_{\bar{\mathsf{f}}}$:

   From ASGN, we have:

   $$\pi = \langle x := [\mathcal{E}]_{\bar{\mathsf{f}}}, \mathcal{M}\rangle \rightarrow \langle \bullet, \mathcal{M}[x \mapsto \mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}})]\rangle. \tag{A.13}$$

By definition, π is an λ-dpiece. Assume

$$\Gamma, \lambda_\kappa \vdash \mathcal{C}. \tag{A.14}$$

We prove $SPNI(\lambda, \pi)$.

Using ASGN, consider:

$$\pi' = \langle x := [\mathcal{E}]_{\bar{\mathsf{f}}}, \mathcal{M}' \rangle \rightarrow \langle \bullet, \mathcal{M}'[x \mapsto \mathcal{M}'([\mathcal{E}]_{\bar{\mathsf{f}}})] \rangle, \tag{A.15}$$

where

$$\mathcal{M} =_\lambda \mathcal{M}', \tag{A.16}$$

$$\forall \mathcal{E} \in \Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}):\ \mathcal{M}(\mathcal{E}) = \mathcal{M}'(\mathcal{E}). \tag{A.17}$$

By definition, π' is a λ-dpiece. Also, $(SPNI - H1)$ holds due to (A.16), and $(SPNI - H2)$ holds due to (A.17). To prove $SPNI(\lambda, \pi)$, we must prove $(SPNI - C1)$, $(SPNI - C2)$, and $(SPNI - C3)$.

π in (A.13) instantiates π in $SPNI$ definition with

$$\mathcal{C}_d = \bullet, \tag{A.18}$$

$$\mathcal{M}_d = \mathcal{M}[x \mapsto \mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}})]. \tag{A.19}$$

π' in (A.15) instantiates π' in $SPNI$ definition with

$$\mathcal{C}'_d = \bullet, \tag{A.20}$$

$$\mathcal{M}'_d = \mathcal{M}'[x \mapsto \mathcal{M}'([\mathcal{E}]_{\bar{\mathsf{f}}})]. \tag{A.21}$$

From (A.18) and (A.20), we get $\mathcal{C}_d = \mathcal{C}'_d$, so $(SPNI - C2)$ holds.

To prove $(SPNI - C1)$ and $(SPNI - C3)$, we proceed by cases.

2.1. $\Gamma(x) \not\sqsubseteq \lambda$:

$(SPNI - C1)$ follows because from (A.13) we have $\pi|_\lambda = \epsilon$; from (A.15) we have $\pi'|_\lambda = \epsilon$. So, $\pi|_\lambda = \pi'|_\lambda$.

We prove ($SPNI - C3$). From (A.19) we have $\mathcal{M}_d|_\lambda = \mathcal{M}|_\lambda$. From (A.16), we then have $\mathcal{M}_d|_\lambda = \mathcal{M}'|_\lambda$. From (A.21), we have $\mathcal{M}'_d|_\lambda = \mathcal{M}'|_\lambda$. By transitivity, we then get $\mathcal{M}_d|_\lambda = \mathcal{M}'_d|_\lambda$. So, ($SPNI - C3$) holds.

2.2. $\Gamma(x) \sqsubseteq \lambda$:

2.2.1. $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}) = \{\mathcal{E}\}$:

From (A.17), we get $\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathcal{M}'([\mathcal{E}]_{\bar{\mathsf{f}}})$. From (A.16) we then get $\mathcal{M}[x \mapsto \mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}})]|_\lambda = \mathcal{M}'[x \mapsto \mathcal{M}'([\mathcal{E}]_{\bar{\mathsf{f}}})]|_\lambda$. From (A.19) and (A.21), we now get $\mathcal{M}_d|_\lambda = \mathcal{M}'_d|_\lambda$. So, ($SPNI - C3$) holds.

From hypothesis of case *2.2.* and (A.16), we have $\mathcal{M}(x) = \mathcal{M}'(x)$. If $\mathcal{M}(x) = \mathcal{M}_d(x)$, then $\mathcal{M}'(x) = \mathcal{M}'_d(x)$, $\pi|_\lambda = \epsilon$, and $\pi'|_\lambda = \epsilon$. Thus ($SPNI - C1$) holds. If $\mathcal{M}(x) \neq \mathcal{M}_d(x)$, then $\mathcal{M}'(x) \neq \mathcal{M}'_d(x)$, and $\pi|_\lambda = \pi'|_\lambda = \langle x, \mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}})\rangle$. So, ($SPNI - C1$) holds.

2.2.2. $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}) = \emptyset$:

From the definition of $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C})$ and hypothesis of case *2.2.* we have:

$$\Gamma(\mathcal{E}) \sqsubseteq \lambda \ \vee \ \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda. \tag{A.22}$$

From (A.14) and ASGN-T, we get $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq \Gamma(x)$. From $\Gamma(x) \sqsubseteq \lambda$ (hypothesis of case *2.2.*), we then get $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq \lambda$. From (A.22), we then get $\Gamma(\mathcal{E}) \sqsubseteq \lambda$.

From type rule EXPR-T and definition of $\sqcup$, we get for all $y \in \mathcal{E}$ that $\Gamma(y) \sqsubseteq \Gamma(\mathcal{E})$. Because $\Gamma(\mathcal{E}) \sqsubseteq \lambda$, we then have $\Gamma(y) \sqsubseteq \lambda$. Thus, from (A.16), we then get $\mathcal{M}(\mathcal{E}) = \mathcal{M}'(\mathcal{E})$. So, $\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathcal{M}'([\mathcal{E}]_{\bar{\mathsf{f}}})$. Similarly to case *2.2.1.*, we then have $\pi|_\lambda = \pi'|_\lambda$ and $M_d|_\lambda = M'_d|_\ell$. So, ($SPNI - C1$) and ($SPNI - C3$) hold.

3. $\mathcal{C}$ **is if** $[\mathcal{E}]_{\bar{\mathsf{f}}}$ **then** $\mathcal{C}_t$ **else** $\mathcal{C}_e$ **end**:

Consider a λ-dpiece:

$$\pi = \langle \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}_e \textbf{ end}, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}_d, \mathcal{M}_d \rangle. \tag{A.23}$$

Assume

$$\Gamma, \lambda_\kappa \vdash \mathcal{C}. \tag{A.24}$$

We prove $SPNI(\lambda, \pi)$.

Consider also an λ-dpiece:

$$\pi' = \langle \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}_e \textbf{ end}, \mathcal{M}' \rangle \xrightarrow{*} \langle \mathcal{C}'_d, \mathcal{M}'_d \rangle, \tag{A.25}$$

where

$$\mathcal{M} =_\lambda \mathcal{M}', \tag{A.26}$$

$$\forall \mathcal{E} \in \Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}):\ \mathcal{M}(\mathcal{E}) = \mathcal{M}'(\mathcal{E}). \tag{A.27}$$

($SPNI - H1$) holds due to (A.26), and ($SPNI - H2$) holds due to (A.27). To prove $SPNI(\lambda, \pi)$, we must prove ($SPNI - C1$), ($SPNI - C2$), and ($SPNI - C3$). We proceed by cases.

3.1. $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq \lambda$ and $\exists x \in \mathit{lhs}(\mathcal{C}_t) \cup \mathit{lhs}(\mathcal{C}_e)$: $\Gamma(x) \sqsubseteq \lambda$:

We first prove that $\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathcal{M}'([\mathcal{E}]_{\bar{\mathsf{f}}})$ by considering the following two cases.

- $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}) = \emptyset$:
  From the definition of $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C})$, $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}) = \emptyset$, and (*3.1.*) we get $\Gamma(\mathcal{E}) \sqsubseteq \lambda$. From (A.26), we then get $\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathcal{M}'([\mathcal{E}]_{\bar{\mathsf{f}}})$.
- $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}) = \{\mathcal{E}\}$:
  From (A.27), we get $\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathcal{M}'([\mathcal{E}]_{\bar{\mathsf{f}}})$.

Thus, in all cases $\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathcal{M}'([\mathcal{E}]_{\bar{\mathsf{f}}})$ holds. So, π and π' take the same branch. Say $\mathcal{C}_t$. We prove ($SPNI - C1$), ($SPNI - C2$), and ($SPNI - C3$).

3.1.1. $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_t) \neq \emptyset$:

Due to BRCH1, (A.23) and (A.25) are:

$$\pi = \langle \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}_e \textbf{ end}, \mathcal{M} \rangle \rightarrow \langle \mathcal{C}_t, \mathcal{M} \rangle \tag{A.28}$$

and

$$\pi' = \langle \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}_e \textbf{ end}, \mathcal{M}' \rangle \rightarrow \langle \mathcal{C}_t, \mathcal{M}' \rangle. \tag{A.29}$$

π in (A.28) instantiates π in $SPNI$ definition with

$$\mathcal{C}_d = \mathcal{C}_t, \tag{A.30}$$

$$\mathcal{M}_d = \mathcal{M}. \tag{A.31}$$

π' in (A.29) instantiates π' in $SPNI$ definition with

$$\mathcal{C}'_d = \mathcal{C}_t, \tag{A.32}$$

$$\mathcal{M}'_d = \mathcal{M}'. \tag{A.33}$$

($SPNI - C2$) holds because (A.30) and (A.32) imply $\mathcal{C}_d = \mathcal{C}'_d$.

No update occurs during π and π', so $\pi|_\lambda = \epsilon$ and $\pi'|_\lambda = \epsilon$. Thus, ($SPNI - C1$) holds because $\pi|_\lambda = \pi'|_\lambda$.

We have:

$$\begin{gathered} \mathcal{M}_d|_\lambda \\ =(A.31) \\ \mathcal{M}|_\lambda \\ =(A.26) \\ \mathcal{M}'|_\lambda \\ =(A.33) \\ \mathcal{M}'_d|_\lambda \end{gathered}$$

So, $M_d|_\lambda = M'_d|_\lambda$ and thus ($SPNI - C3$) holds.

3.1.2. $\Delta^{-}_{\hat{\mathcal{C}}}(\lambda, \mathcal{C}_t) = \emptyset$:

So,

$$\forall \mathcal{E} \in \Delta^{-}_{\hat{\mathcal{C}}}(\lambda, \mathcal{C}_t):\ \mathcal{M}(\mathcal{E}) = \mathcal{M}'(\mathcal{E}). \tag{A.34}$$

Due to BRCH1, (A.23) and (A.25) are:

$\pi = \langle \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}_e \textbf{ end}, \mathcal{M}\rangle \rightarrow \langle \mathcal{C}_t, \mathcal{M}\rangle \xrightarrow{*} \langle \mathcal{C}_d, \mathcal{M}_d\rangle$, and

$\pi' = \langle \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}_e \textbf{ end}, \mathcal{M}'\rangle \rightarrow \langle \mathcal{C}_t, \mathcal{M}'\rangle \xrightarrow{*} \langle \mathcal{C}'_d, \mathcal{M}'_d\rangle$,

where $\mathcal{C}_d$ and $\mathcal{C}'_d$ are the first downgrade commands or termination symbols after starting executing the branches (because π and π are λ-dpieces).

Consider suffixes of π and π':

$$\pi_t = \langle \mathcal{C}_t, \mathcal{M}\rangle \xrightarrow{*} \langle \mathcal{C}_d, \mathcal{M}_d\rangle, \tag{A.35}$$

and

$$\pi'_t = \langle \mathcal{C}_t, \mathcal{M}'\rangle \xrightarrow{*} \langle \mathcal{C}'_d, \mathcal{M}'_d\rangle. \tag{A.36}$$

By construction, both π_t and π'_t are λ-dpieces.

Because $\mathcal{C}_t \in \mathcal{C}$, we can instantiate IH with:

$$\hat{\mathcal{C}} = \mathcal{C}_t, \hat{\mathcal{M}} = \mathcal{M}, \hat{\lambda} = \lambda, \hat{\lambda_\kappa} = \lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}), \hat{\pi} = \pi_t. \tag{A.37}$$

We get

$$\Gamma, \lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \vdash \mathcal{C}_t \Rightarrow\ \mathit{SPNI}(\lambda, \pi_t). \tag{A.38}$$

From typing rule BRCH-T and (A.24), we have that

$\Gamma, \lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \vdash \mathcal{C}_t$ holds. So, from (A.38), we get that $\mathit{SPNI}(\lambda, \pi_t)$ holds, too. In the definition of SPNI, we instantiate π with π_t and π' with π'_t. Because $(\mathit{SPNI}-\mathit{H1})$ holds due to (A.26) and $(\mathit{SPNI}-\mathit{H2})$ holds due to (A.34), we get $\pi_t|_\lambda = \pi'_t|_\lambda$, $\mathcal{C}_d = \mathcal{C}'_d$, and $\mathcal{M}_d|_\lambda = \mathcal{M}'_d|_\lambda$. Thus, $(\mathit{SPNI}-\mathit{C2})$ and $(\mathit{SPNI}-\mathit{C3})$ hold.

We now prove $(SPNI-C1)$. The observations generated by π_t are the same with the observations generated by π, so $\pi|_\lambda = \pi_t|_\lambda$. Similarly, $\pi'|_\lambda = \pi'_t|_\lambda$. From $\pi_t|_\lambda = \pi'_t|_\lambda$, and $\pi|_\lambda = \pi_t|_\lambda$ we get $\pi|_\lambda = \pi'_t|_\lambda$. From $\pi'|_\lambda = \pi'_t|_\lambda$, we then get $\pi|_\lambda = \pi'|_\lambda$. So, $(SPNI-C1)$ holds.

3.2. $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda$ or $\forall x \in lhs(\mathcal{C}_t) \cup lhs(\mathcal{C}_e)$: $\Gamma(x) \not\sqsubseteq \lambda$:

We first prove that $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda$ implies $\forall x \in lhs(\mathcal{C}_t) \cup lhs(\mathcal{C}_e)$: $\Gamma(x) \not\sqsubseteq \lambda$. Assume $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda$ holds. From type rule BRCH-T, we get $\Gamma, \lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \vdash \mathcal{C}_t$ and $\Gamma, \lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \vdash \mathcal{C}_e$. From $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq \lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}})$ and $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda$, we get $\lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda$. From Lemma 5, we then get: $\forall x \in lhs(\mathcal{C}_t) \cup lhs(\mathcal{C}_e)$: $\Gamma(x) \not\sqsubseteq \lambda$. So, during execution, no update to an λ variable will happen, and no downgrade to λ will happen. Thus, $\pi|_\lambda = \pi'|_\lambda = \epsilon$, $\mathcal{C}_d = \mathcal{C}'_d = \bullet$ and $M_d|_\lambda = M|_\lambda = M'|_\lambda = M'_d|_\lambda$. Thus, $(SPNI-C1)$, $(SPNI-C2)$, and $(SPNI-C3)$ hold.

4. $\mathcal{C}$ is $\mathcal{C}_1; \mathcal{C}_2$

Consider a λ-dpiece:

$$\pi = \langle \mathcal{C}_1; \mathcal{C}_2, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}_d, \mathcal{M}_d \rangle. \tag{A.39}$$

Assume

$$\Gamma, \lambda_\kappa \vdash \mathcal{C}. \tag{A.40}$$

We prove $SPNI(\lambda, \pi)$.

Consider also a λ-dpiece:

$$\pi' = \langle \mathcal{C}_1; \mathcal{C}_2, \mathcal{M}' \rangle \xrightarrow{*} \langle \mathcal{C}'_d, \mathcal{M}'_d \rangle. \tag{A.41}$$

where

$$\mathcal{M} =_\lambda \mathcal{M}', \tag{A.42}$$

$$\forall \mathcal{E} \in \Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}):\ \mathcal{M}(\mathcal{E}) = \mathcal{M}'(\mathcal{E}). \tag{A.43}$$

($SPNI-H1$) holds due to (A.42), and ($SPNI-H2$) holds due to (A.43). To prove $SPNI(\lambda, \pi)$, we must prove ($SPNI-C1$), ($SPNI-C2$), and ($SPNI-C3$).

π involves the execution of $\mathcal{C}_1$. Consider the λ-dpieces below:

$$\pi_1 = \langle \mathcal{C}_1, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}_{d1}, \mathcal{M}_{d1} \rangle \tag{A.44}$$

and

$$\pi_1' = \langle \mathcal{C}_1, \mathcal{M}' \rangle \xrightarrow{*} \langle \mathcal{C}_{d1}', \mathcal{M}_{d1}' \rangle. \tag{A.45}$$

Because $\mathcal{C}_1 \in \mathcal{C}$, we can instantiate IH with:

$$\hat{\mathcal{C}} = \mathcal{C}_1, \hat{\mathcal{M}} = \mathcal{M}, \hat{\lambda} = \lambda, \hat{\lambda_\kappa} = \lambda_\kappa, \hat{\pi} = \pi_1. \tag{A.46}$$

We get

$$\Gamma, \lambda_\kappa \vdash \mathcal{C}_1 \Rightarrow \ SPNI(\lambda, \pi_1). \tag{A.47}$$

From typing rule SEQ-T and (A.40), we have that $\Gamma, \lambda_\kappa \vdash \mathcal{C}_1$ holds. So, from (A.47), we get that $SPNI(\lambda, \pi_1)$ holds, too.

In the definition of $SPNI$, we instantiate π with π_1 and π' with π_1'. ($SPNI-H1$) holds because $\Delta^-_{\hat{\mathcal{C}}}(\lambda, \mathcal{C}_1; \mathcal{C}_2) = \Delta^-_{\hat{\mathcal{C}}}(\lambda, \mathcal{C}_1)$ (by definition) and (A.43) hold. ($SPNI-H2$) holds due to (A.42). Thus, we get:

$$\pi_1|_\lambda = \pi_1'|_\lambda, \quad \mathcal{C}_{d1} = \mathcal{C}_{d1}', \quad \mathcal{M}_{d1}|_\lambda = \mathcal{M}_{d1}'|_\lambda. \tag{A.48}$$

For what follows, we write $\mathcal{C} = \ldots \mathcal{C}'$ to denote $\exists \mathcal{M}, \mathcal{M}'$: $\langle \mathcal{C}, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}', \mathcal{M}' \rangle$.

Due to SEQ1, SEQ2, and because $\mathcal{C}_d$ is a downgrade or termination, $\mathcal{C}_d$ is either

- partial execution of $\mathcal{C}_1$ followed by $\mathcal{C}_2$:
  $\mathcal{C}_d = \mathcal{C}_1'; \mathcal{C}_2$ and $\mathcal{C}_1 = \ldots \mathcal{C}_1'$, or
- $\mathcal{C}_2$, or

- partial execution of $\mathcal{C}_2$ or termination:

  $\mathcal{C}_2 = \ldots \mathcal{C}_d$ or $\mathcal{C}_d = \bullet$.

We prove $(SPNI - C1)$, $(SPNI - C2)$, and $(SPNI - C3)$, by examining the above cases.

4.1. $\mathcal{C}_d$ is $\mathcal{C}_1'; \mathcal{C}_2$ or $\mathcal{C}_2$.

So, π involves the execution of only $\mathcal{C}_1$, and not $\mathcal{C}_2$. We distinguish two cases based on the last command $\mathcal{C}_{d1}$ of λ-dpieces π_1 and π_1'.

4.1.1. $\mathcal{C}_{d1} \neq \bullet$: So, while executing $\mathcal{C}_1$ in π_1, $\mathcal{C}_{d1}$ is the first downgrade that occurs. Thus, while executing $\mathcal{C}_1$ in π, $\mathcal{C}_{d1}; \mathcal{C}_2$ should be the first downgrade that occurs. Similarly, while executing $\mathcal{C}_1$ in π', $\mathcal{C}_{d1'}; \mathcal{C}_2$ should be the first downgrade that occurs. From SEQ, and using π_1, π_1', we get: $\pi = \langle \mathcal{C}_1; \mathcal{C}_2, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}_{d1}; \mathcal{C}_2, \mathcal{M}_{d1} \rangle$, and
$\pi' = \langle \mathcal{C}_1; \mathcal{C}_2, \mathcal{M}' \rangle \xrightarrow{*} \langle \mathcal{C}_{d1'}; \mathcal{C}_2, \mathcal{M}'_{d1} \rangle$.
π in instantiates π in $SPNI$ definition with

$$\mathcal{C}_d = \mathcal{C}_{d1}; \mathcal{C}_2, \tag{A.49}$$

$$\mathcal{M}_d = \mathcal{M}_{d1}. \tag{A.50}$$

π' instantiates π' in $SPNI$ definition with

$$\mathcal{C}'_d = \mathcal{C}_{d1'}; \mathcal{C}_2, \tag{A.51}$$

$$\mathcal{M}'_d = \mathcal{M}'_{d1}. \tag{A.52}$$

$(SPNI - C2)$ holds because from $\mathcal{C}_{d1} = \mathcal{C}'_{d1}$ in (A.48), (A.49), and (A.51), we have $\mathcal{C}_d = \mathcal{C}'_d$.
We now prove $(SPNI - C3)$. From (A.48), we get $\mathcal{M}_{d1}|_\lambda = \mathcal{M}'_{d1}|_\lambda$. From (A.50), we get $\mathcal{M}_d|_\lambda = \mathcal{M}_{d1}|_\lambda$. And thus, by transitivity, we

have $\mathcal{M}_d|_\lambda = \mathcal{M}'_{d1}|_\lambda$. From (A.52), we get $\mathcal{M}'_d|_\lambda = \mathcal{M}'_{d1}|_\lambda$. And thus, by transitivity, we have $\mathcal{M}_d|_\lambda = \mathcal{M}'_d|_\lambda$.

We now prove $(SPNI-C1)$. Pieces π and π_1 generate the same observations, so $\pi|_\lambda = \pi_1|_\lambda$. From $\pi_1|_\lambda = \pi'_1|_\lambda$ in (A.48), we then get $\pi|_\lambda = \pi'_1|_\lambda$. Pieces π' and π'_1 generate the same observations, so $\pi'|_\lambda = \pi'_1|_\lambda$. By transitivity, we get $\pi|_\lambda = \pi'|_\lambda$. Thus, $(SPNI-C1)$ holds.

4.1.2. $\mathcal{C}_{d1} = \bullet$: Due to (A.48), we also get $\mathcal{C}'_{d1} = \bullet$. So, λ-dpieces π_1 (in (A.44)) and π'_1 (in (A.45)) are:

$$\pi_1 = \langle \mathcal{C}_1, \mathcal{M} \rangle \xrightarrow{*} \langle \bullet, \mathcal{M}_{d1} \rangle \tag{A.53}$$

and

$$\pi'_1 = \langle \mathcal{C}_1, \mathcal{M}' \rangle \xrightarrow{*} \langle \bullet, \mathcal{M}'_{d1} \rangle. \tag{A.54}$$

Because $\mathcal{C}_{d1} = \bullet$ ($\mathcal{C}'_{d1} = \bullet$) holds, and because π_1 and π'_1 are λ-dpieces, no downgrade occurs while executing $\mathcal{C}_1$ in π_1 and π'_1. So, no downgrade occurs while executing $\mathcal{C}_1$ in π and π'. By assumption *4.1.*, $\mathcal{C}_d$ is either $\mathcal{C}'_1; \mathcal{C}_2$ or $\mathcal{C}_2$, and because no downgrade occurs while executing $\mathcal{C}_1$ in π and π', $\mathcal{C}_d$ (in (A.39)) and $\mathcal{C}'_d$ (in (A.41)) should be $\mathcal{C}_2$. Thus, $(SPNI-C2)$ holds because $\mathcal{C}_d = \mathcal{C}'_d = \mathcal{C}_2$.

We now prove $(SPNI-C3)$. From SEQ, and using π_1 in (A.53), π'_1 in in (A.54), pieces π and π' are: $\pi = \langle \mathcal{C}_1; \mathcal{C}_2, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}_2, \mathcal{M}_{d1} \rangle$, and $\pi' = \langle \mathcal{C}_1; \mathcal{C}_2, \mathcal{M}' \rangle \xrightarrow{*} \langle \mathcal{C}_2, \mathcal{M}'_{d1} \rangle$.

π instantiates π in $SPNI$ definition with

$$\mathcal{M}_d = \mathcal{M}_{d1}. \tag{A.55}$$

π' instantiates π' in $SPNI$ definition with

$$\mathcal{M}'_d = \mathcal{M}'_{d1}. \tag{A.56}$$

We have:

$$\begin{aligned}
&\mathcal{M}_d|_\lambda \\
=&(A.55) \\
&\mathcal{M}_{d1}|_\lambda \\
=&(A.48) \\
&\mathcal{M}'_{d1}|_\lambda \\
=&(A.56) \\
&\mathcal{M}'_d|_\lambda.
\end{aligned}$$

So, $\mathcal{M}_d|_\lambda = \mathcal{M}'_d|_\lambda$ and thus $(SPNI - C3)$ holds.

We now prove $(SPNI - C1)$. Pieces π and π_1 generate the same observations because both execute only $\mathcal{C}_1$, so $\pi|_\lambda = \pi_1|_\lambda$. Similarly, pieces π' and π'_1 generate the same observations, so $\pi'|_\lambda = \pi'_1|_\lambda$. From $\pi_1|_\lambda = \pi'_1|_\lambda$ in (A.48), we then get $\pi|_\lambda = \pi'|_\lambda$.

4.2. For $\mathcal{C}_d$ (in (A.39)), either $\mathcal{C}_2 = \ldots \mathcal{C}_d$ or $\mathcal{C}_d = \bullet$ holds.

So, π involves the execution of both $\mathcal{C}_1$ and some of $\mathcal{C}_2$. Because $\mathcal{C}_d$ is not $\mathcal{C}'_1; \mathcal{C}_2$ (where $\mathcal{C}_1 = \ldots \mathcal{C}'_1$), no downgrade occurs while executing $\mathcal{C}_1$. Thus, λ-dpiece π_1 (in A.44) that involves the execution of only $\mathcal{C}_1$ should end at a termination. So, $\mathcal{C}_{d1}$ (in (A.44)) is $\bullet$. From $\mathcal{C}_{d1} = \mathcal{C}'_{d1}$ in (A.48), we then get that $\mathcal{C}'_{d1}$ (in (A.45)) is $\bullet$. So, π_1 (in (A.44)) and π'_1 (in (A.44)) are:

$\pi_1 = \langle \mathcal{C}_1, \mathcal{M} \rangle \xrightarrow{*} \langle \bullet, \mathcal{M}_{d1} \rangle$, and

$\pi'_1 = \langle \mathcal{C}_1, \mathcal{M}' \rangle \xrightarrow{*} \langle \bullet, \mathcal{M}'_{d1} \rangle$.

Consider now pieces:

$\pi_2 = \langle \mathcal{C}_2, \mathcal{M}_{d1} \rangle \xrightarrow{*} \langle \mathcal{C}_{d2}, \mathcal{M}_{d2} \rangle$, and

$\pi'_2 = \langle \mathcal{C}_2, \mathcal{M}'_{d1} \rangle \xrightarrow{*} \langle \mathcal{C}'_{d2}, \mathcal{M}'_{d2} \rangle$.

Because $\mathcal{C}_2 \in \mathcal{C}$, we can instantiate IH with:

$$\hat{\mathcal{C}} = \mathcal{C}_2, \hat{\mathcal{M}} = \mathcal{M}_{d1}, \hat{\lambda} = \lambda, \hat{\lambda_\kappa} = \lambda_\kappa, \hat{\pi} = \pi_2. \tag{A.57}$$

We get

$$\Gamma, \lambda_\kappa \vdash \mathcal{C}_2 \Rightarrow \ \mathit{SPNI}(\lambda, \pi_2). \tag{A.58}$$

From typing rule SEQ-T and (A.40), we have that $\Gamma, \lambda_\kappa \vdash \mathcal{C}_2$ holds. So, from (A.58), we get that $\mathit{SPNI}(\lambda, \pi_2)$ holds, too. In the definition of SPNI, we instantiate π with π_2 and π' with π'_2. $(\mathit{SPNI}-\mathit{H1})$ holds due to (A.48).

We prove $(\mathit{SPNI}-\mathit{H2})$. $\mathcal{C}_d$ is not $\mathcal{C}_2$, and thus, no downgrade happens at $\mathcal{C}_2$. So, by the definition of $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C})$ we have $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_2) = \emptyset$.
Thus $\forall \mathcal{E} \in \Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_2)$: $\mathcal{M}_{d1}(\mathcal{E}) = \mathcal{M}'_{d1}(\mathcal{E})$. Thus, we get:

$$\pi_2|_\lambda = \pi'_2|_\lambda, \quad \mathcal{C}_{d2} = \mathcal{C}'_{d2}, \quad \mathcal{M}_{d2}|_\lambda = \mathcal{M}'_{d2}|_\lambda. \tag{A.59}$$

From SEQ, π is constructed out of π_1 and π_2, and π' is constructed out of π'_1 and π'_2:
$\pi = \langle \mathcal{C}_1; \mathcal{C}_2, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}_2, \mathcal{M}_{d1} \rangle \xrightarrow{*} \langle \mathcal{C}_{d2}, \mathcal{M}_{d2} \rangle$, and
$\pi' = \langle \mathcal{C}_1; \mathcal{C}_2, \mathcal{M}' \rangle \xrightarrow{*} \langle \mathcal{C}_2, \mathcal{M}'_{d1} \rangle \xrightarrow{*} \langle \mathcal{C}'_{d2}, \mathcal{M}'_{d2} \rangle$.
π instantiates π in SPNI definition with:

$$\mathcal{C}_d = \mathcal{C}_{d2}, \tag{A.60}$$

$$\mathcal{M}_d = \mathcal{M}_{d2}. \tag{A.61}$$

π' instantiates π' in SPNI definition with

$$\mathcal{C}'_d = \mathcal{C}'_{d2}, \tag{A.62}$$

$$\mathcal{M}'_d = \mathcal{M}'_{d2}. \tag{A.63}$$

From (A.60), (A.62) and (A.59), we get $\mathcal{C}_d = \mathcal{C}'_d$. So, $(\mathit{SPNI}-\mathit{C2})$ holds.

We prove $(SPNI-C3)$. We have:

$$\begin{aligned}
&\mathcal{M}_d|_\lambda \\
=&(A.61) \\
&\mathcal{M}_{d2}|_\lambda \\
=&(A.59) \\
&\mathcal{M}'_{d2}|_\lambda \\
=&(A.63) \\
&\mathcal{M}'_d|_\lambda.
\end{aligned}$$

So, $\mathcal{M}_d|_\lambda = \mathcal{M}'_d|_\lambda$. Thus, $(SPNI-C3)$ holds.

We now prove $(SPNI-C1)$. We have $\pi|_\lambda = (\pi_1 \to \pi_2)|_\lambda$ and $\pi'|_\lambda = (\pi'_1 \to \pi'_2)|_\lambda$. So, from $\pi_1|_\lambda = \pi'_1|_\lambda$ in (A.48) and $\pi_2|_\lambda = \pi'_2|_\lambda$ in (A.59), we then have $\pi|_\lambda = \pi'|_\lambda$.

5. $\mathcal{C}$ is **while** $[\mathcal{E}]_{\bar{\mathsf{f}}}$ **do** $\mathcal{C}_t$ **end**.

   Consider a λ-dpiece:

$$\pi = \langle \textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_t \textbf{ end}, \mathcal{M}\rangle \xrightarrow{*} \langle \mathcal{C}_d, \mathcal{M}_d\rangle. \tag{A.64}$$

   Assume

$$\Gamma, \lambda_\kappa \vdash \mathcal{C}. \tag{A.65}$$

   We prove $SPNI(\lambda, \pi)$.

   Consider also a λ-dpiece:

$$\pi' = \langle \textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_t \textbf{ end}, \mathcal{M}'\rangle \xrightarrow{*} \langle \mathcal{C}'_d, \mathcal{M}'_d\rangle, \tag{A.66}$$

   where

$$\mathcal{M} =_\lambda \mathcal{M}', \tag{A.67}$$

$$\forall \mathcal{E} \in \Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}):\ \mathcal{M}(\mathcal{E}) = \mathcal{M}'(\mathcal{E}). \tag{A.68}$$

($SPNI - H1$) holds due to (A.67), and ($SPNI - H2$) holds due to (A.68). To prove $SPNI(\lambda, \pi)$, we must prove ($SPNI - C1$), ($SPNI - C2$), and ($SPNI - C3$).

We use induction on the maximum number of instantiations of rule LOOP1 for $\mathcal{C}$ in π, with the following induction hypothesis:

IH$_w$: If $\hat{\pi} \triangleq \langle \textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_t \textbf{ end}, \hat{\mathcal{M}} \rangle \xrightarrow{*} \langle \hat{\mathcal{C}}_d, \hat{\mathcal{M}}_d \rangle$ involves strictly less iterations than those involved in π, then $SPNI(\lambda, \hat{\pi})$ holds.

Base case: π involves no iteration.

So, π executes only LOOP2.

5.1. $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq \lambda$ and $\exists x \in lhs(\mathcal{C}_t)$: $\Gamma(x) \sqsubseteq \ell$.

We first prove that $\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathcal{M}'([\mathcal{E}]_{\bar{\mathsf{f}}})$ by considering the following two cases.

5.1.1. $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}) = \emptyset$:

From the definition of $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C})$, (*5.1.1.*), and (*5.1.*) we get $\Gamma(\mathcal{E}) \sqsubseteq \lambda$.

From (A.67), we then get $\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathcal{M}'([\mathcal{E}]_{\bar{\mathsf{f}}})$.

5.1.2. $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}) = \{\mathcal{E}\}$:

From (A.68), we get $\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathcal{M}'([\mathcal{E}]_{\bar{\mathsf{f}}})$.

So, in all cases, we have $\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathcal{M}'([\mathcal{E}]_{\bar{\mathsf{f}}})$, and thus, π' executes only LOOP2. Then, π and π' are:

$\pi = \langle \textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_t \textbf{ end}, \mathcal{M} \rangle \rightarrow \langle \bullet, \mathcal{M} \rangle$, and

$\pi' = \langle \textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_t \textbf{ end}, \mathcal{M}' \rangle \rightarrow \langle \bullet, \mathcal{M}' \rangle$.

π instantiates π in $SPNI$ definition with

$$\mathcal{C}_d = \bullet, \tag{A.69}$$

$$\mathcal{M}_d = \mathcal{M}. \tag{A.70}$$

π' instantiates π' in $SPNI$ definition with

$$\mathcal{C}'_d = \bullet, \tag{A.71}$$

$$\mathcal{M}'_d = \mathcal{M}'. \tag{A.72}$$

($SPNI - C2$) holds because from (A.69) and (A.71), we get $\mathcal{C}_d = \mathcal{C}'_d$.

We prove ($SPNI - C3$). We have:

$$\begin{aligned}
&\mathcal{M}_d|_\lambda \\
=&(A.70) \\
&\mathcal{M}|_\lambda \\
=&(A.67) \\
&\mathcal{M}'|_\lambda \\
=&(A.72) \\
&\mathcal{M}'_d|_\lambda.
\end{aligned}$$

So, $\mathcal{M}_d|_\lambda = \mathcal{M}'_d|_\lambda$.

For ($SPNI - C1$), no observations are generated during π and π', so $\pi|_\lambda = \epsilon$ and $\pi'|_\lambda = \epsilon$, and thus $\pi|_\lambda = \pi'|_\lambda$.

5.2. $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda$ or $\forall x \in \mathit{lhs}(\mathcal{C}_t)\colon \Gamma(x) \not\sqsubseteq \lambda$:

We first prove that $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda$ implies $\forall x \in \mathit{lhs}(\mathcal{C}_t)\colon \Gamma(x) \not\sqsubseteq \lambda$. Assume $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda$. From type rule LOOP-T, we get $\Gamma, \lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \vdash \mathcal{C}_t$. From $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq \lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}})$ and $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda$, we get $\lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda$. From Lemma 5, we then get:

$$\forall x \in \mathit{lhs}(\mathcal{C}_t)\colon \Gamma(x) \not\sqsubseteq \lambda. \tag{A.73}$$

Thus, we get ($SPNI - C1$), because $\pi|_\lambda = \epsilon$ and $\pi'|_\lambda = \epsilon$, and we get ($SPNI - C3$), because $M_d|_\lambda = M|_\lambda$, $M'_d|_\lambda = M'|_\lambda$ and (A.67). Also, during

execution no downgrade to λ will happen, due to (A.73). Thus, we get ($SPNI - C2$), because $\mathcal{C}_d = \bullet$ and $\mathcal{C}'_d = \bullet$.

Induction case: π involves the execution of at least one iteration.

So, π executes at least one LOOP1.

5.1. $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq \lambda$ and $\exists x \in lhs(\mathcal{C}_t)$: $\Gamma(x) \sqsubseteq \lambda$:

Similar to (*5.1.*) of the base case, we have $\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathcal{M}'([\mathcal{E}]_{\bar{\mathsf{f}}})$. Thus, both π and π' take LOOP1. Thus, both $\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}})$ and $\mathcal{M}'([\mathcal{E}]_{\bar{\mathsf{f}}})$ evaluate to *true*.

Using rules BRCH and LOOP, we get that executing

$$\mathcal{C} = \textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_t \textbf{ end} \tag{A.74}$$

on $\mathcal{M}$ or $\mathcal{M}'$ is equivalent to executing

$$\mathcal{C}_f = \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}^0 \textbf{ end}; \textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_t \textbf{ end} \tag{A.75}$$

on $\mathcal{M}$ or $\mathcal{M}'$, where $\mathcal{C}^0$ is a **skip** that is not executed (because $\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}})$ and $\mathcal{M}'([\mathcal{E}]_{\bar{\mathsf{f}}})$ evaluate to *true*). This means that λ-dpieces of $\mathcal{C}$ on $\mathcal{M}$ and $\mathcal{M}'$ (i.e., π and π') are the same with the λ-dpieces of $\mathcal{C}_f$ on $\mathcal{M}$ and $\mathcal{M}'_t$, except for the first state.

Consider λ-dpieces for $\mathcal{C}_f$:

$$\pi^* = \langle \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}^0 \textbf{ end}; \textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_t \textbf{ end}, \mathcal{M}\rangle \xrightarrow{*} \langle \mathcal{C}_d, \mathcal{M}_d\rangle, \tag{A.76}$$

$$\pi'^* = \langle \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}^0 \textbf{ end}; \textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_t \textbf{ end}, \mathcal{M}'\rangle \xrightarrow{*} \langle \mathcal{C}'_d, \mathcal{M}'_d\rangle, \tag{A.77}$$

Because $\mathcal{C}$ and $\mathcal{C}_f$ are equivalent on $\mathcal{M}$ and $\mathcal{M}'$, we have

$$\pi^*|_\lambda = \pi|_\lambda \ , \ \pi'^*|_\lambda = \pi'|_\lambda. \tag{A.78}$$

π^* and π'^* first execute the if-statement.

Consider two λ-dpieces of that if-statement:

$$\pi_1 = \langle \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}^0 \textbf{ end}, \mathcal{M}\rangle \xrightarrow{*} \langle \mathcal{C}_{d1}, \mathcal{M}_{d1}\rangle, \tag{A.79}$$

$$\pi'_1 = \langle \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}^0 \textbf{ end}, \mathcal{M}'\rangle \xrightarrow{*} \langle \mathcal{C}'_{d1}, \mathcal{M}'_{d1}\rangle. \tag{A.80}$$

Due to case 2, $SPNI(\lambda, \pi_1)$ holds. π_1 instantiates π in $SPNI$ definition. π'_1 instantiates π' in $SPNI$ definition. $(SPNI - H1)$ holds due to (A.67). $(SPNI - H2)$ holds due to (A.68) and because
$\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}^0 \textbf{ end}) = \Delta^-_{\widehat{\mathcal{C}}}(\lambda, \textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_t \textbf{ end})$. So, we get:

$$\pi_1|_\lambda = \pi'_1|_\lambda,\ \mathcal{C}_{d1} = \mathcal{C}'_{d1},\ \mathcal{M}_{d1}|_\lambda = \mathcal{M}'_{d1}|_\lambda. \tag{A.81}$$

5.1.1. $\mathcal{C}_{d1} \neq \bullet$:

From (A.81), we then have $\mathcal{C}'_{d1} \neq \bullet$. So, a downgrade occurs while executing the if-statement. Thus, π^* and π^* does not reach the while-statement. Instead, π^* and π^* execute only the steps that are executed in π_1 and π'_1, correspondingly. So, π^* and π^* are constructed from π_1 and π'_1 by concatenating all commands with "$;\mathcal{C}$":

$$\pi^* = \langle \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}^0 \textbf{ end}; \mathcal{C}, \mathcal{M}\rangle \xrightarrow{*} \langle \mathcal{C}_{d1}; \mathcal{C}, \mathcal{M}_{d1}\rangle, \tag{A.82}$$

$$\pi'^* = \langle \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}^0 \textbf{ end}; \mathcal{C}, \mathcal{M}'\rangle \xrightarrow{*} \langle \mathcal{C}'_{d1}; \mathcal{C}, \mathcal{M}'_{d1}\rangle. \tag{A.83}$$

Also,

$$\pi^*|_\lambda = \pi_1|_\lambda,\ \pi'^*|_\lambda = \pi'_1|_\lambda. \tag{A.84}$$

Because π and π' are the same as π^* and π'^*, except for the first state, we have:

$$\pi = \langle \mathcal{C}, \mathcal{M}\rangle \xrightarrow{*} \langle \mathcal{C}_{d1}; \mathcal{C}, \mathcal{M}_{d1}\rangle, \tag{A.85}$$

$$\pi' = \langle \mathcal{C}, \mathcal{M}' \rangle \xrightarrow{*} \langle \mathcal{C}'_{d1}; \mathcal{C}, \mathcal{M}'_{d1} \rangle. \tag{A.86}$$

π instantiates π in $SPNI$ definition with:

$$\mathcal{C}_d = \mathcal{C}_{d1}; \mathcal{C} \tag{A.87}$$

$$\mathcal{M}_d = \mathcal{M}_{d1}. \tag{A.88}$$

π' instantiates π' in $SPNI$ definition with:

$$\mathcal{C}'_d = \mathcal{C}'_{d1}; \mathcal{C} \tag{A.89}$$

$$\mathcal{M}'_d = \mathcal{M}'_{d1}. \tag{A.90}$$

From $\mathcal{C}_{d1} = \mathcal{C}'_{d1}$ in (A.81), from (A.87) and (A.89), we get $\mathcal{C}_d = \mathcal{C}'_d$. Thus, $(SPNI - C2)$ holds.

From $\mathcal{M}_{d1}|_\lambda = \mathcal{M}'_{d1}|_\lambda$ in (A.81), from (A.88) and (A.90), we get $\mathcal{M}_d|_\lambda = \mathcal{M}'_d|_\lambda$. Thus, $(SPNI - C3)$ holds.

From (A.78), $\pi_1|_\lambda = \pi'_1|_\lambda$ in (A.81), and (A.84), we get $\pi|_\lambda = \pi'|_\lambda$. Thus, $(SPNI - C1)$ holds.

5.1.2. $\mathcal{C}_{d1} = \bullet$:

From (A.81), we then have $\mathcal{C}'_{d1} = \bullet$. Consider suffixes of π^* (in (A.76)) and π'^* (in (A.77)):

$$\pi_2 = \langle \textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_t \textbf{ end}, \mathcal{M}_{d1} \rangle \xrightarrow{*} \langle \mathcal{C}_d, \mathcal{M}_d \rangle, \tag{A.91}$$

$$\pi'_2 = \langle \textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_t \textbf{ end}, \mathcal{M}'_{d1} \rangle \xrightarrow{*} \langle \mathcal{C}'_d, \mathcal{M}'_d \rangle. \tag{A.92}$$

π_2 and π'_2 are λ-dpieces. Because π_2 involves strictly fewer iterations than those involved in π, we can apply IH_w to get $SPNI(\lambda, \pi_2)$. π_2 instantiates π in $SPNI$ definition. π'_2 instantiates π' in $SPNI$ definition. $(SPNI - H1)$ holds due to (A.81). $(SPNI - H2)$ holds due to (A.68). Thus, we get:

$$\pi_2|_\lambda = \pi'_2|_\lambda,\ \mathcal{C}_d = \mathcal{C}'_d,\ \mathcal{M}_d|_\lambda = \mathcal{M}'_d|_\lambda. \tag{A.93}$$

Thus ($SPNI-C2$) and ($SPNI-C3$) hold.

Observations in π^* are the observations in π_1 followed by the observations in π_2: $\pi^*|_\lambda = (\pi_1 \to \pi_2)|_\lambda$. Similarly, $\pi'^*|_\lambda = (\pi_1' \to \pi_2')|_\lambda$. Because $\pi|_\lambda = \pi^*|_\lambda = (\pi_1 \to \pi_2)|_\lambda$ and $\pi'|_\lambda = \pi^*|_\lambda = (\pi_1' \to \pi_2')|_\lambda$, equations (A.81) and (A.93) imply $\pi|_\lambda = \pi'|_\lambda$. So, ($SPNI-C1$) holds.

5.2. $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda$ or $\forall x \in \mathit{lhs}(\mathcal{C}_t)$: $\Gamma(x) \not\sqsubseteq \lambda$:

Similar to case *5.2.* in Base case.

□

Lemma 3. *If* $\Gamma, \lambda_\kappa \vdash \mathcal{C}$*, then* $\Gamma, \lambda_\kappa \vdash \mathbb{T}(\lambda, \mathcal{C})$.

Proof. By structural induction on $\mathcal{C}$. □

Lemma 4. *If* $\Gamma, \lambda_\kappa \vdash \mathcal{C}$ *and* $\langle \mathcal{C}, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}', \mathcal{M}' \rangle$ *and* $\mathcal{C}' \neq \bullet$*, then* $\Gamma, \lambda_\kappa \vdash \mathcal{C}'$.

Proof. Assume $\Gamma, \lambda_\kappa \vdash \mathcal{C}$. We use structural induction on $\mathcal{C}$.

IH: If $\hat{\mathcal{C}} \in \mathcal{C}$ and $\Gamma, \lambda_\kappa \vdash \hat{\mathcal{C}}$ and $\langle \hat{\mathcal{C}}, \hat{\mathcal{M}} \rangle \xrightarrow{*} \langle \hat{\mathcal{C}}', \hat{\mathcal{M}}' \rangle$ and $\mathcal{C}' \neq \bullet$, then $\Gamma, \lambda_\kappa \vdash \hat{\mathcal{C}}'$.

1. $\mathcal{C}$ is **skip**

   Trivially true, because $\mathcal{C}' = \bullet$.

2. $\mathcal{C}$ is $x := [\mathcal{E}]_{\bar{\mathsf{f}}}$

   Trivially true, because $\mathcal{C}' = \bullet$.

3. $\mathcal{C}$ is **if** $[\mathcal{E}]_{\bar{\mathsf{f}}}$ **then** $\mathcal{C}_t$ **else** $\mathcal{C}_e$ **end**

   From typing rule BRCH-T and $\Gamma, \lambda_\kappa \vdash \mathcal{C}$, we have

$$\Gamma, \lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \vdash \mathcal{C}_t \text{ and } \Gamma, \lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \vdash \mathcal{C}_e.$$

From Lemma 6, we then get:

$$\Gamma, \lambda_\kappa \vdash \mathcal{C}_t \text{ and } \Gamma, \lambda_\kappa \vdash \mathcal{C}_e. \tag{A.94}$$

From BRCH1 and BRCH2, we have $\mathcal{C}' = \mathcal{C}_t$, or $\mathcal{C}' = \mathcal{C}_e$, or $\langle \mathcal{C}_t, \mathcal{M}_t \rangle \xrightarrow{*} \langle \mathcal{C}', \mathcal{M}' \rangle$, or $\langle \mathcal{C}_e, \mathcal{M}_e \rangle \xrightarrow{*} \langle \mathcal{C}', \mathcal{M}' \rangle$.

3.1. $\mathcal{C}' = \mathcal{C}_t$ or $\mathcal{C}' = \mathcal{C}_e$:

From (A.94), we get $\Gamma, \lambda_\kappa \vdash \mathcal{C}'$.

3.2. $\langle \mathcal{C}_t, \mathcal{M}_t \rangle \xrightarrow{*} \langle \mathcal{C}', \mathcal{M}' \rangle$ or $\langle \mathcal{C}_e, \mathcal{M}_e \rangle \xrightarrow{*} \langle \mathcal{C}', \mathcal{M}' \rangle$:

Assume $\langle \mathcal{C}_i, \mathcal{M}_i \rangle \xrightarrow{*} \langle \mathcal{C}', \mathcal{M}' \rangle$, where $\mathcal{C}_i$ is $\mathcal{C}_t$ or $\mathcal{C}_e$. We instantiate IH with $\hat{\mathcal{C}} = \mathcal{C}_i$. Because $\mathcal{C}_i \in \mathcal{C}$, (A.94) holds, and $\langle \mathcal{C}_i, \mathcal{M}_i \rangle \xrightarrow{*} \langle \mathcal{C}', \mathcal{M}' \rangle$ (hypothesis of this subcase), we get $\Gamma, \lambda_\kappa \vdash \mathcal{C}'$.

4. $\mathcal{C}$ is $\mathcal{C}_1; \mathcal{C}_2$

From typing rule SEQ-T and $\Gamma, \lambda_\kappa \vdash \mathcal{C}$, we have

$$\Gamma, \lambda_\kappa \vdash \mathcal{C}_1 \text{ and } \Gamma, \lambda_\kappa \vdash \mathcal{C}_2. \tag{A.95}$$

From SEQ1 and SEQ2, and because $\langle \mathcal{C}, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}', \mathcal{M}' \rangle$, we have:

- $\mathcal{C}'$ is partial execution of $\mathcal{C}_1$, followed by $\mathcal{C}_2$:
  $\langle \mathcal{C}_1, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}'', \mathcal{M}' \rangle$ and $\mathcal{C}' = \mathcal{C}''; \mathcal{C}_2$, or
- $\mathcal{C}'$ is $\mathcal{C}_2$, or
- $\mathcal{C}'$ is partial execution of $\mathcal{C}_2$:
  $\langle \mathcal{C}_2, \mathcal{M}_2 \rangle \xrightarrow{*} \langle \mathcal{C}', \mathcal{M}' \rangle$.

We examine these cases here:

4.1. $\langle \mathcal{C}_1, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}'', \mathcal{M}' \rangle$ and $\mathcal{C}' = \mathcal{C}''; \mathcal{C}_2$:

We instantiate IH with $\hat{\mathcal{C}} = \mathcal{C}_1$. From $\mathcal{C}_1 \in \mathcal{C}$, (A.95), and $\langle \mathcal{C}_1, \mathcal{M} \rangle \xrightarrow{*}$

$\langle \mathcal{C}'', \mathcal{M}' \rangle$ (hypothesis of this subcase), we get $\Gamma, \lambda_\kappa \vdash \mathcal{C}''$. From (A.95), $\mathcal{C}' = \mathcal{C}''; \mathcal{C}_2$, and type rule SEQ-T, we then have $\Gamma, \lambda_\kappa \vdash \mathcal{C}'$.

4.2. $\mathcal{C}' = \mathcal{C}_2$:

From (A.95) we get $\Gamma, \lambda_\kappa \vdash \mathcal{C}'$.

4.3. $\langle \mathcal{C}_2, \mathcal{M}_2 \rangle \xrightarrow{*} \langle \mathcal{C}', \mathcal{M}' \rangle$:

We instantiate IH with $\hat{\mathcal{C}} = \mathcal{C}_2$. From $\mathcal{C}_2 \in \mathcal{C}$, (A.95), and $\langle \mathcal{C}_2, \mathcal{M}_2 \rangle \xrightarrow{*} \langle \mathcal{C}', \mathcal{M}' \rangle$ (hypothesis of this subcase), we get $\Gamma, \lambda_\kappa \vdash \mathcal{C}'$.

5. $\mathcal{C}$ is **while** $[\mathcal{E}]_{\bar{\mathsf{f}}}$ **do** $\mathcal{C}_t$ **end**

From Lemma typing rule LOOP-T and $\Gamma, \lambda_\kappa \vdash \mathcal{C}$, we have

$$\Gamma, \lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \vdash \mathcal{C}_t.$$

From Lemma 6, we then get:

$$\Gamma, \lambda_\kappa \vdash \mathcal{C}_t. \tag{A.96}$$

From LOOP1, and because $\langle \mathcal{C}, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}', \mathcal{M}' \rangle$, we have:

- $\mathcal{C}' = \mathcal{C}_t;$ **while** $[\mathcal{E}]_{\bar{\mathsf{f}}}$ **do** $\mathcal{C}_t$ **end**, or
- $\mathcal{C}'$ is partial execution of $\mathcal{C}_t$ followed by the loop:
  $\langle \mathcal{C}_t, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}'', \mathcal{M}' \rangle$ and $\mathcal{C}'$ is $\mathcal{C}''$; **while** $[\mathcal{E}]_{\bar{\mathsf{f}}}$ **do** $\mathcal{C}_t$ **end**, or
- $\mathcal{C}' =$ **while** $[\mathcal{E}]_{\bar{\mathsf{f}}}$ **do** $\mathcal{C}_t$ **end**

We examine these cases here:

5.1. $\mathcal{C}' = \mathcal{C}_t;$ **while** $[\mathcal{E}]_{\bar{\mathsf{f}}}$ **do** $\mathcal{C}_t$ **end**:

From (A.96), $\Gamma, \lambda_\kappa \vdash \mathcal{C}$, and type rule SEQ-T, we get $\Gamma, \lambda_\kappa \vdash \mathcal{C}'$.

5.2. $\langle \mathcal{C}_t, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}'', \mathcal{M}' \rangle$ and $\mathcal{C}' = \mathcal{C}''$; **while** $[\mathcal{E}]_{\bar{\mathsf{f}}}$ **do** $\mathcal{C}_t$ **end**:

We instantiate IH with $\hat{\mathcal{C}} = \mathcal{C}_t$. From $\mathcal{C}_t \in \mathcal{C}$, (A.96), and $\langle \mathcal{C}_t, \mathcal{M} \rangle \xrightarrow{*} \langle \mathcal{C}'', \mathcal{M}' \rangle$ (hypothesis of this subcase), we get $\Gamma, \lambda_\kappa \vdash \mathcal{C}''$. From $\Gamma, \lambda_\kappa \vdash \mathcal{C}$, and type rule SEQ-T, we get $\Gamma, \lambda_\kappa \vdash \mathcal{C}'$.

5.3. $\mathcal{C}' = \textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_t \textbf{ end}$:

From $\Gamma, \lambda_\kappa \vdash \mathcal{C}$, we get $\Gamma, \lambda_\kappa \vdash \mathcal{C}'$.

□

Lemma 5. *If* $\Gamma, \lambda_\kappa \vdash \mathcal{C}$ *and* $\lambda_\kappa \not\sqsubseteq \lambda$*, then* $\forall y \in \mathit{lhs}(\mathcal{C})$: $\Gamma(y) \not\sqsubseteq \lambda$.

Proof. Assume $\Gamma, \lambda_\kappa \vdash \mathcal{C}$ and $\lambda_\kappa \not\sqsubseteq \lambda$.

We prove $\forall y \in \mathit{lhs}(\mathcal{C})$: $\Gamma(y) \not\sqsubseteq \lambda$ by structural induction on $\mathcal{C}$.

IH: If $\hat{\mathcal{C}} \in \mathcal{C}$ and $\Gamma, \hat{\lambda_\kappa} \vdash \hat{\mathcal{C}}$ and $\hat{\lambda_\kappa} \not\sqsubseteq \lambda$, then $\forall y \in \mathit{lhs}(\hat{\mathcal{C}})$: $\Gamma(y) \not\sqsubseteq \lambda$.

1. $\mathcal{C}$ is **skip**:

   Trivially true, because $\mathit{lhs}(\mathcal{C}) = \emptyset$.

2. $\mathcal{C}$ is $x := [\mathcal{E}]_{\bar{\mathsf{f}}}$:

   Here, $\mathit{lhs}(\mathcal{C}) = \{x\}$. We prove that $\Gamma(x) \not\sqsubseteq \lambda$. From ASGN-T, we get $\lambda_\kappa \sqsubseteq \Gamma(x)$. From $\lambda_\kappa \not\sqsubseteq \lambda$, we then get $\Gamma(x) \not\sqsubseteq \lambda$.

3. $\mathcal{C}$ is $\textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}_e \textbf{ end}$:

   We have $\mathit{lhs}(\mathcal{C}) = \mathit{lhs}(\mathcal{C}_t) \cup \mathit{lhs}(\mathcal{C}_t)$. From BRCH we get $\Gamma, \lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \vdash \mathcal{C}_t$ and $\Gamma, \lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \vdash \mathcal{C}_e$. From $\lambda_\kappa \sqsubseteq \lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}})$ and $\lambda_\kappa \not\sqsubseteq \lambda$, we get $\lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda$. And because $\mathcal{C}_t, \mathcal{C}_e \in \mathcal{C}$, we apply IH on $\mathcal{C}_t$ and $\mathcal{C}_e$: we instantiate $\hat{\mathcal{C}}$ with $\mathcal{C}_t$ or $\mathcal{C}_e$ and $\hat{\lambda_\kappa}$ with $\lambda_\kappa \sqcup \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}})$. So, $\forall y \in \mathit{lhs}(\mathcal{C}_t)$: $\Gamma(y) \not\sqsubseteq \lambda$ and $\forall y \in \mathit{lhs}(\mathcal{C}_e)$: $\Gamma(y) \not\sqsubseteq \lambda$. Thus, $\forall y \in \mathit{lhs}(\mathcal{C})$: $\Gamma(y) \not\sqsubseteq \lambda$.

4. $\mathcal{C}$ is $\textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_t \textbf{ end}$:

   Similar to the above case.

5. $\mathcal{C}$ is $\mathcal{C}_1; \mathcal{C}_2$:

   Similar to the above case.

□

Lemma 6. *If* $\Gamma, \lambda_\kappa \vdash \mathcal{C}$ *and* $\lambda'_\kappa \sqsubseteq \lambda_\kappa$*, then* $\Gamma, \lambda'_\kappa \vdash \mathcal{C}$*.*

Proof. By structural induction on $\mathcal{C}$. □

A.2 Completeness of $\sqsubseteq_{\mathrm{RA}}$ and $\sqcup_{\mathrm{RA}}$ for RIF automata

Proposition 1. *Consider algorithm* $rstr$*, which takes as input two RIF automata* $\lambda_1 = \langle Q_1, \mathcal{F}, \delta_1, q_1, r_1 \rangle$*,* $\lambda_2 = \langle Q_2, \mathcal{F}, \delta_2, q_2, r_2 \rangle$*, and a sequence* $visited$ *of pairs of automata states:*

$rstr(\lambda_1, \lambda_2, visited) \triangleq$

1. *if* $(\langle q_1, q_2 \rangle \in visited)$ *then return* $true$*.*
2. *if* $r_1(q_1) \not\sqsubseteq_R r_2(q_2)$ *then return* $false$*.*
3. $visited' := visited \cup \{\langle q_1, q_2 \rangle\}$
4. *return* $(\forall \mathsf{f} \in \mathcal{F}$: $rstr(\mathcal{T}_{\mathrm{RA}}(\lambda_1, \mathsf{f}), \mathcal{T}_{\mathrm{RA}}(\lambda_2, \mathsf{f}), visited'))$

Then, $rstr(\lambda_1, \lambda_2, \epsilon)$ *holds iff* $\lambda_1 \sqsubseteq_{\mathrm{RA}} \lambda_2$ *holds.*

Proof.

1. Soundness

   Assume $rstr(\lambda_1, \lambda_2, \epsilon)$ holds. (h1)

   We prove that $\lambda_1 \sqsubseteq_{\mathrm{RA}} \lambda_2$ holds. So, we prove that

$$\forall \mathsf{F}\text{: } \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_1, \mathsf{F})) \sqsubseteq_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_2, \mathsf{F})).$$

Fix F. We should prove that $\mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_1, F)) \sqsubseteq_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_2, F))$.

Assume, for contradiction, that $\mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_1, F)) \not\sqsubseteq_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_2, F))$. (h2)
Due to Lemma 7 and (h1), $rstr(\mathcal{T}_{\mathrm{RA}}(\lambda_1, F), \mathcal{T}_{\mathrm{RA}}(\lambda_2, F), visited)$ must have been called. If step 1 of executing $rstr(\mathcal{T}_{\mathrm{RA}}(\lambda_1, F), \mathcal{T}_{\mathrm{RA}}(\lambda_2, F), visited)$ holds, then there must have been a previous call $rstr(\mathcal{T}_{\mathrm{RA}}(\lambda_1, F), \mathcal{T}_{\mathrm{RA}}(\lambda_2, F), visited')$, such that step 1 does not hold.

So, consider a call $rstr(\mathcal{T}_{\mathrm{RA}}(\lambda_1, F), \mathcal{T}_{\mathrm{RA}}(\lambda_2, F), visited)$ where step 1 does not hold. From (h2) we get that step 2 returns $false$. So, $rstr(\mathcal{T}_{\mathrm{RA}}(\lambda_1, F), \mathcal{T}_{\mathrm{RA}}(\lambda_2, F), visited)$ does not hold, and thus $rstr(\lambda_1, \lambda_2, \epsilon)$ does not hold, which contradicts (h1). So, we proved that $\mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_1, F)) \sqsubseteq_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_2, F))$ holds.

2. Completeness

   Assume $\lambda_1 \sqsubseteq_{\mathrm{RA}} \lambda_2$ holds. (h3)
   We should prove that $rstr(\lambda_1, \lambda_2, \epsilon)$ holds. Assume for contradiction that $rstr(\lambda_1, \lambda_2, \epsilon)$ does not hold. So, there is an F such that $rstr(\mathcal{T}_{\mathrm{RA}}(\lambda_1, F), \mathcal{T}_{\mathrm{RA}}(\lambda_2, F), visited)$ does not hold. So, when executing $rstr(\mathcal{T}_{\mathrm{RA}}(\lambda_1, F), \mathcal{T}_{\mathrm{RA}}(\lambda_2, F), visited)$, step 2 must have returned false. This means that $\mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_1, F)) \not\sqsubseteq_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_2, F))$, which contradicts (h3). So, we proved that $rstr(\lambda_1, \lambda_2, \epsilon)$ holds.

□

Lemma 7. *If $rstr(\lambda_1, \lambda_2, \epsilon)$ holds, then for every sequence F there exists a recursive call of $rstr$ with the first two arguments being equal to $\mathcal{T}_{\mathrm{RA}}(\lambda_1, F)$ and $\mathcal{T}_{\mathrm{RA}}(\lambda_2, F)$.*

Proof. Assume $rstr(\lambda_1, \lambda_2, \epsilon)$ holds (h1).
Fix an F'.

We prove that, when $rstr(\lambda_1, \lambda_2, \epsilon)$ is executed, $rstr(\mathcal{T}_{\mathrm{RA}}(\lambda_1, \mathsf{F}'), \mathcal{T}_{\mathrm{RA}}(\lambda_2, \mathsf{F}'), visited)$ is called, for some $visited$. We use induction on the length of F', with induction hypothesis:

$$IH : \text{If } |\mathsf{F}| < |\mathsf{F}'|, \text{ and if } rstr(\lambda_1, \lambda_2, \epsilon) \text{ is executed,}$$
$$\text{then } rstr(\mathcal{T}_{\mathrm{RA}}(\lambda_1, \mathsf{F}), \mathcal{T}_{\mathrm{RA}}(\lambda_2, \mathsf{F}), visited) \text{ is called.}$$

Assume $rstr(\lambda_1, \lambda_2, \epsilon)$ is executed. If $\mathsf{F}' = \epsilon$, then $rstr(\mathcal{T}_{\mathrm{RA}}(\lambda_1, \mathsf{F}), \mathcal{T}_{\mathrm{RA}}(\lambda_2, \mathsf{F}), visited)$ is $rstr(\lambda_1, \lambda_2, \epsilon)$.

Now, assume that $\mathsf{F}' = \mathsf{F}\mathsf{f}$. By IH, we get that $rstr(\mathcal{T}_{\mathrm{RA}}(\lambda_1, \mathsf{F}), \mathcal{T}_{\mathrm{RA}}(\lambda_2, \mathsf{F}), visited)$ has been called. We prove that $rstr(\mathcal{T}_{\mathrm{RA}}(\lambda_1, \mathsf{F}\mathsf{f}), \mathcal{T}_{\mathrm{RA}}(\lambda_2, \mathsf{F}\mathsf{f}), visited')$ has been called, too. Consider the execution of $rstr(\mathcal{T}_{\mathrm{RA}}(\lambda_1, \mathsf{F}), \mathcal{T}_{\mathrm{RA}}(\lambda_2, \mathsf{F}), visited)$, and assume $\mathcal{T}_{\mathrm{RA}}(\lambda_1, \mathsf{F}) = \langle Q_1, \mathcal{F}, \delta_1, q_1, r_1 \rangle$ and $\mathcal{T}_{\mathrm{RA}}(\lambda_2, \mathsf{F}) = \langle Q_2, \mathcal{F}, \delta_2, q_2, r_2 \rangle$

1. Assume step 1 does not return $false$.

   Due to (h1), step 2 does not return $false$, either. So, step 4 is executed. For $\mathsf{f} \in \Sigma$, we have that $rstr(\mathcal{T}_{\mathrm{RA}}(\lambda_1, \mathsf{F}\mathsf{f}), \mathcal{T}_{\mathrm{RA}}(\lambda_2, \mathsf{F}\mathsf{f}), visited')$ is called.

2. Assume step 1 returns $false$.

   Then, there must have been a previous call $rstr(\lambda_1', \lambda_2', visited')$, such that steps 3 and 4 are executed, where $\lambda_1' = \langle Q_1', \mathcal{F}', \delta_1', q_1', r_1' \rangle$, $\lambda_2' = \langle Q_2', \mathcal{F}', \delta_2', q_2', r_2' \rangle$, and $q_1' = q_1$ and $q_2' = q_2$. By the definition of $rstr$ and $\mathcal{T}_{\mathrm{RA}}$, we get:

   $Q_1' = Q_1$, $\Sigma' = \Sigma$, $\delta_1' = \delta_1$, $Prins_1' = Prins_1$,

   $Q_2' = Q_2$, $\delta_2' = \delta_2$, $Prins_2' = Prins_2$.

   So, $\lambda_1' = \mathcal{T}_{\mathrm{RA}}(\lambda_1, \mathsf{F})$ and $\lambda_2' = \mathcal{T}_{\mathrm{RA}}(\lambda_2, \mathsf{F})$. Thus, when step 4 in call $rstr(\lambda_1', \lambda_2', visited')$ is executed we have, for $\mathsf{f} \in F$, that $rstr(\mathcal{T}_{\mathrm{RA}}(\lambda_1, \mathsf{F}\mathsf{f}), \mathcal{T}_{\mathrm{RA}}(\lambda_2, \mathsf{F}\mathsf{f}), visited'')$ is called.

□

Proposition 2. *$\lambda \sqcup_{\mathrm{RA}} \lambda'$ is the least upper bound of λ and λ'.*

Proof. We first prove that $\lambda \sqsubseteq_{\mathrm{RA}} \lambda \sqcup_{\mathrm{RA}} \lambda'$ and $\lambda' \sqsubseteq_{\mathrm{RA}} \lambda \sqcup_{\mathrm{RA}} \lambda'$.

W.l.o.g., we prove $\lambda \sqsubseteq_{\mathrm{RA}} \lambda \sqcup_{\mathrm{RA}} \lambda'$.

So, we should prove

$$\forall F\colon \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda, F)) \sqsubseteq_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda \sqcup_{\mathrm{RA}} \lambda', F)).$$

Fix an F.

We prove $\mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda, F)) \sqsubseteq_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda \sqcup_{\mathrm{RA}} \lambda', F))$.

By definition of $\sqcup_{\mathrm{RA}}$, we have that the current state of $\mathcal{T}_{\mathrm{RA}}(\lambda \sqcup_{\mathrm{RA}} \lambda', F)$ is $\langle q, q' \rangle$, where q is the current state of $\mathcal{T}_{\mathrm{RA}}(\lambda, F)$, and q' is the current state of $\mathcal{T}_{\mathrm{RA}}(\lambda', F)$.

So,

$$\mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda \sqcup_{\mathrm{RA}} \lambda', F)) = \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda, F)) \sqcup_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda', F)). \tag{A.97}$$

Thus, $\mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda, F)) \sqsubseteq_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda \sqcup_{\mathrm{RA}} \lambda', F))$.

Now, we prove that if $\lambda \sqsubseteq_{\mathrm{RA}} \lambda''$ and $\lambda' \sqsubseteq_{\mathrm{RA}} \lambda''$, then $\lambda'' \not\sqsubset_{\mathrm{RA}} \lambda \sqcup_{\mathrm{RA}} \lambda'$.

Assume for contradiction that $\lambda'' \sqsubset_{\mathrm{RA}} \lambda \sqcup_{\mathrm{RA}} \lambda'$.

So, there is an F such that $\mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda'', F)) \sqsubset_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda \sqcup_{\mathrm{RA}} \lambda', F))$.

From (A.97), we then have

$$\mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda'', F)) \sqsubset_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda, F)) \sqcup_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda', F)). \tag{A.98}$$

From $\lambda \sqsubseteq_{\mathrm{RA}} \lambda''$ and $\lambda' \sqsubseteq_{\mathrm{RA}} \lambda''$, we have $\mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda, F)) \sqsubseteq_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda'', F))$ and $\mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda', F)) \sqsubseteq_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda'', F))$.

So, $\mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda, F)) \sqcup_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda', F)) \sqsubseteq_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda'', F))$, which contradicts (A.98).

So, we proved that $\lambda'' \not\sqsubset_{\mathrm{RA}} \lambda \sqcup_{\mathrm{RA}} \lambda'$. □

A.3 Completeness of $\sqsubseteq_\kappa$ and $\sqcup_\kappa$ for κ-labels

We first prove completeness of $\sqsubseteq_\kappa$ for κ-atoms that are singleton sets. For brevity, we write $\langle \mathsf{F}, B\rangle$ instead of $\{\langle \mathsf{F}, B\rangle\}$.

Proposition 3.

$$\mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \mathsf{F}_2, B_2\rangle, \mathsf{F}_2^{\mathsf{c}})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \mathsf{F}_1, B_1\rangle, \mathsf{F}_2^{\mathsf{c}}))$$

$$\Leftrightarrow$$

$$\forall \mathsf{F}\colon \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \mathsf{F}_2, B_2\rangle, \mathsf{F})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \mathsf{F}_1, B_1\rangle, \mathsf{F}))$$

Proof.

1. $\Rightarrow$

Assume the following holds:

$$\mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \mathsf{F}_2, B_2\rangle, \mathsf{F}_2^{\mathsf{c}})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \mathsf{F}_1, B_1\rangle, \mathsf{F}_2^{\mathsf{c}})). \tag{A.99}$$

We prove that:

$$\forall \mathsf{F}\colon \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \mathsf{F}_2, B_2\rangle, \mathsf{F})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \mathsf{F}_1, B_1\rangle, \mathsf{F})).$$

Fix F. We prove $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \mathsf{F}_2, B_2\rangle, \mathsf{F})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \mathsf{F}_1, B_1\rangle, \mathsf{F}))$. By definition of $\mathcal{T}_\kappa$ and (4.11), it suffices to prove

$$\mathcal{R}_\kappa(\langle(\!|\mathsf{F}_2\mathsf{F}|\!), B_2\rangle) \supseteq \mathcal{R}_\kappa(\langle(\!|\mathsf{F}_1\mathsf{F}|\!), B_1\rangle). \tag{A.100}$$

1.1. Assume F_2 is not fully complementable.

By definition of $\langle \mathsf{F}_2, B_2\rangle$ we have that F_2 is reduced. Lemma 11 then gives $\mathcal{R}_\kappa(\langle(\!|\mathsf{F}_2\mathsf{F}|\!), B_2\rangle) = \mathsf{F}$. So, (A.100) holds.

1.2. Assume instead that F_2 is fully complementable.

By definition of $\mathcal{T}_\kappa$ and (4.11), (A.99) becomes

$$\mathcal{R}_\kappa(\langle(\!|F_2F_2^c|\!), B_2\rangle) \supseteq \mathcal{R}_\kappa(\langle(\!|F_1F_2^c|\!), B_1\rangle)$$

Because $(\!|F_2F_2^c|\!) = \epsilon$, we then have

$$\mathcal{R}_\kappa(\langle\epsilon, B_2\rangle) \supseteq \mathcal{R}_\kappa(\langle(\!|F_1F_2^c|\!), B_1\rangle) \tag{A.101}$$

From (A.101), Lemma 9, and (4.11), we get

$$\mathcal{R}_\kappa(\langle(\!|F_2F|\!), B_2\rangle) \supseteq \mathcal{R}_\kappa(\langle(\!|F_1F_2^cF_2F|\!), B_1\rangle) \tag{A.102}$$

From Lemma 8, we get

$$\mathcal{R}_\kappa(\langle(\!|F_1F_2^cF_2F|\!), B_1\rangle) \supseteq \mathcal{R}_\kappa(\langle(\!|F_1F|\!), B_1\rangle) \tag{A.103}$$

By transitivity on (A.102) and (A.103), we get (A.100).

2. $\Leftarrow$

Assume the following holds:

$$\forall F\colon \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle F_2, B_2\rangle, F)) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle F_1, B_1\rangle, F)). \tag{A.104}$$

We prove that the algorithm decides that $\langle F_2, B_2\rangle \sqsubseteq_\kappa \langle F_1, B_1\rangle$ holds. Let F be F_2^c. From (A.104), we have $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle F_2, B_2\rangle, F_2^c)) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle F_1, B_1\rangle, F_2^c))$. So, the algorithm decides that $\langle F_2, B_2\rangle \sqsubseteq_\kappa \langle F_1, B_1\rangle$ holds.

□

For the following proof we define F to be a *subcollection* of F' iff:

$$\mathsf{f} \in F \Rightarrow \mathsf{f} \in F'$$

Lemma 8. $\mathcal{R}_\kappa(\langle(\!|F_2F_1^{\mathsf{c}}F_1F|\!), B\rangle) \supseteq \mathcal{R}_\kappa(\langle(\!|F_2F|\!), B\rangle$ *if* F_1 *is fully complementable.*

Proof. Assume F_1 is fully complementable. We first prove that $(\!|F_2F|\!)$ is a subcollection of $(\!|F_2F_1^{\mathsf{c}}F_1F|\!)$. We use induction on $|F_1|$, with induction hypothesis:

> IH: If F_i is fully complementable and $|F_i| < |F_1|$, then for all F_a, F_b, $(\!|F_aF_b|\!)$ is a subcollection of $(\!|F_aF_i^{\mathsf{c}}F_iF_b|\!)$.

1. $|F_1| = 0$.

So, $F_1 = \epsilon$. We have $(\!|F_2F_1^{\mathsf{c}}F_1F|\!) = (\!|F_2\epsilon^{\mathsf{c}}\epsilon F|\!) = (\!|F_2\epsilon\epsilon F|\!) = (\!|F_2F|\!)$.

2. $|F_1| = 1$.

So $F_1 = \mathsf{f}$. From (4.11), we have:

$$(\!|F_2F|\!) = (\!|(\!|F_2|\!)(\!|F|\!)|\!) = (\!|F_{2r}F_r|\!) \tag{A.105}$$

and

$$(\!|F_2\mathsf{f}^{\mathsf{c}}\mathsf{f}F|\!) = (\!|(\!|F_2|\!)\mathsf{f}^{\mathsf{c}}\mathsf{f}(\!|F|\!)|\!) = (\!|F_{2r}\mathsf{f}^{\mathsf{c}}\mathsf{f}F_r|\!) \tag{A.106}$$

It suffices to prove that $(\!|F_{2r}F_r|\!)$ is a subcollection of $(\!|F_{2r}\mathsf{f}^{\mathsf{c}}\mathsf{f}F_r|\!)$.

2.1. $(\mathsf{f}^{\mathsf{c}})^{\mathsf{c}}$ exists.

So, $(\mathsf{f}^{\mathsf{c}})^{\mathsf{c}} = \mathsf{f}$. From (4.11), we then have $(\!|F_{2r}F_r|\!) = (\!|F_{2r}\mathsf{f}^{\mathsf{c}}\mathsf{f}F_r|\!)$.

2.2. $F_{2r} = F'_{2r}\mathsf{f}$

Using (4.11), we then have

$$(\!|F_{2r}\mathsf{f}^{\mathsf{c}}\mathsf{f}F_r|\!) = (\!|F'_{2r}\mathsf{f}\mathsf{f}^{\mathsf{c}}\mathsf{f}F_r|\!) = (\!|F'_{2r}\mathsf{f}F_r|\!) = (\!|F_{2r}F_r|\!) \tag{A.107}$$

2.3. $F_r = \mathsf{f}^{\mathsf{c}}F'_r$

Using (4.11), we then have

$$(\!|F_{2r}\mathsf{f}^{\mathsf{c}}\mathsf{f}F_r|\!) = (\!|F_{2r}\mathsf{f}^{\mathsf{c}}\mathsf{f}\mathsf{f}^{\mathsf{c}}F'_r|\!) = (\!|F_{2r}\mathsf{f}^{\mathsf{c}}F'_r|\!) = (\!|F_{2r}F_r|\!) \tag{A.108}$$

2.4. $(\mathsf{f}^{\mathsf{c}})^{\mathsf{c}}$ does not exist, $F_{2r} \neq F'_{2r}\mathsf{f}$, and $F_r \neq \mathsf{f}^{\mathsf{c}}F'_r$

By definition (4.10) of reduction we have that $(\!|F_{2r}\mathsf{f}^{\mathsf{c}}\mathsf{f}F_r|\!) = F_{2r}\mathsf{f}^{\mathsf{c}}\mathsf{f}F_r$. Also $(\!|F_{2r}F_r|\!)$ is a subcollection of $F_{2r}F_r$. Because $F_{2r}F_r$ is a subcollection of $F_{2r}\mathsf{f}^{\mathsf{c}}\mathsf{f}F_r$, we then get by transitivity that $F_{2r}F_r$ is a subcollection of $(\!|F_{2r}\mathsf{f}^{\mathsf{c}}\mathsf{f}F_r|\!)$, too.

3. $|F_1| > 1$

Let $F_1 = F_{1h}F_{1t}$. From (4.20) and because F_1 is fully complementable, we then have $(\!|F_2F_1^{\mathsf{c}}F_1F|\!) = (\!|F_2F_{1t}^{\mathsf{c}}F_{1h}^{\mathsf{c}}F_{1h}F_{1t}F|\!)$. Applying IH with $F_i = F_{1h}$, $F_a = F_2F_{1t}^{\mathsf{c}}$, and $F_b = F_{1t}F$, we have that $(\!|F_2F_{1t}^{\mathsf{c}}F_{1t}F|\!)$ is a subcollection of $(\!|F_2F_{1t}^{\mathsf{c}}F_{1h}^{\mathsf{c}}F_{1h}F_{1t}F|\!)$. Applying IH with $F_i = F_{1t}$, $F_a = F_2$, and $F_b = F$, we have that $(\!|F_2F|\!)$ is a subcollection of $(\!|F_2F_{1t}^{\mathsf{c}}F_{1t}F|\!)$. So, by transitivity, $(\!|F_2F|\!)$ is a subcollection of $(\!|F_2F_1^{\mathsf{c}}F_1F|\!)$.

So, $(\!|F_2F|\!)$ is a subcollection of $(\!|F_2F_1^{\mathsf{c}}F_1F|\!)$. From Lemma 10 we then have: $\bigcup_{\mathsf{f}\in(\!|F_2F_1^{\mathsf{c}}F_1F|\!)} \overline{\mathcal{X}}(\mathsf{f}) \supseteq \bigcup_{\mathsf{f}\in(\!|F_2F|\!)} \overline{\mathcal{X}}(\mathsf{f})$. So, $\bigcup_{\mathsf{f}\in(\!|F_2F_1^{\mathsf{c}}F_1F|\!)} \overline{\mathcal{X}}(\mathsf{f}) \cup B \supseteq \bigcup_{\mathsf{f}\in(\!|F_2F|\!)} \overline{\mathcal{X}}(\mathsf{f}) \cup B$. Thus, $\mathcal{R}_\kappa(\langle(\!|F_2F_1^{\mathsf{c}}F_1F|\!), B\rangle) \supseteq \mathcal{R}_\kappa(\langle(\!|F_2F|\!), B\rangle$. □

Lemma 9. $\mathcal{R}_\kappa(\langle\epsilon, B_2\rangle) \supseteq \mathcal{R}_\kappa(\langle F_1, B_1\rangle) \;\Rightarrow\; \forall F\text{: } \mathcal{R}_\kappa(\langle(\!|F|\!), B_2\rangle) \supseteq \mathcal{R}_\kappa(\langle(\!|F_1F|\!), B_1\rangle)$

Proof. Assume

$$\mathcal{R}_\kappa(\langle\epsilon, B_2\rangle) \supseteq \mathcal{R}_\kappa(\langle F_1, B_1\rangle). \tag{A.109}$$

Fix F. We prove $\mathcal{R}_\kappa(\langle(\!|F|\!), B_2\rangle) \supseteq \mathcal{R}_\kappa(\langle(\!|F_1F|\!), B_1\rangle)$.

Let $q \notin \mathcal{R}_\kappa(\langle(\!|F|\!), B_2\rangle)$. We prove

$$q \notin \mathcal{R}_\kappa(\langle(\!|F_1F|\!), B_1\rangle). \tag{A.110}$$

From $q \notin \mathcal{R}_\kappa(\langle(\!|F|\!), B_2\rangle)$, we get $q \notin B_2 \cup \bigcup_{\mathsf{f}\in(\!|F|\!)} \overline{\mathcal{X}}(\mathsf{f})$. So, $q \notin B_2$ and $q \notin \bigcup_{\mathsf{f}\in(\!|F|\!)} \overline{\mathcal{X}}(\mathsf{f})$. Thus, $q \notin \mathcal{R}_\kappa(\langle\epsilon, B_2\rangle)$ and $q \notin \bigcup_{\mathsf{f}\in(\!|F|\!)} \overline{\mathcal{X}}(\mathsf{f})$. From (A.109), we then

get $q \notin \mathcal{R}_\kappa(\langle F_1, B_1\rangle)$ and $q \notin \bigcup_{\mathsf{f}\in(\!|F|\!)} \overline{\mathcal{X}}(\mathsf{f})$. So, $q \notin \mathcal{R}_\kappa(\langle F_1, B_1\rangle) \cup \bigcup_{\mathsf{f}\in(\!|F|\!)} \overline{\mathcal{X}}(\mathsf{f})$, which gives $q \notin B_1 \cup \bigcup_{\mathsf{f}\in F_1} \overline{\mathcal{X}}(\mathsf{f}) \cup \bigcup_{\mathsf{f}\in(\!|F|\!)} \overline{\mathcal{X}}(\mathsf{f})$. Thus,

$$q \notin (B_1 \cup \bigcup_{\mathsf{f}\in F_1(\!|F|\!)} \overline{\mathcal{X}}(\mathsf{f})). \tag{A.111}$$

We have that $(\!|F_1 F|\!)$ is a subcollection of $F_1(\!|F|\!)$. From (A.111) and Lemma 10 we then get that $q \notin (B_1 \cup \bigcup_{\mathsf{f}\in(\!|F_1 F|\!)} \overline{\mathcal{X}}(\mathsf{f}))$. So, $q \notin \mathcal{R}_\kappa(\langle(\!|F_1 F|\!), B_1\rangle)$, which is (A.110). □

Lemma 10. *If F' is a subcollection of F, then $\bigcup_{\mathsf{f}\in F}\overline{\mathcal{X}}(\mathsf{f}) \supseteq \bigcup_{\mathsf{f}\in F'} \overline{\mathcal{X}}(\mathsf{f})$*

Proof. Assume $\mathsf{p} \in \bigcup_{\mathsf{f}\in F'} \overline{\mathcal{X}}(\mathsf{f})$. Then $\mathsf{p} \in \overline{\mathcal{X}}(\mathsf{f})$ for some $\mathsf{f} \in F'$. Because F' is a subcollection of F, we then have $\mathsf{f} \in F$. So, $\mathsf{p} \in \overline{\mathcal{X}}(\mathsf{f})$ for $\mathsf{f} \in F$. Thus, $\mathsf{p} \in \bigcup_{\mathsf{f}\in F} \overline{\mathcal{X}}(\mathsf{f})$. So, $\bigcup_{\mathsf{f}\in F} \overline{\mathcal{X}}(\mathsf{f}) \supseteq \bigcup_{\mathsf{f}\in F'} \overline{\mathcal{X}}(\mathsf{f})$. □

Lemma 11. *If F' is reduced and not fully complementable, then*

$$\forall F\text{: } \mathcal{R}_\kappa(\langle(\!|F' F|\!), B\rangle) = P.$$

Proof. Assume F' is reduced and not fully complementable. Then, there is f_u in F' such that $\mathsf{f}_u^{\mathsf{c}}$ does not exist. Fix F. We prove $\mathcal{R}_\kappa(\langle(\!|F' F|\!), B\rangle) = P$.

We first prove that $\mathsf{f}_u \in (\!|F' F|\!)$, by induction on $|F|$, with induction hypothesis IH.

$$\text{IH: If } |F_i| < |F|, \text{ then } \mathsf{f}_u \in (\!|F' F_i|\!).$$

1. $|F| = 0$.

   So, $F = \epsilon$. We have $(\!|F' F|\!) = (\!|F'\epsilon|\!) = (\!|F'|\!) = F'$. So, $\mathsf{f}_u \in (\!|F' F|\!)$.

2. $|F| > 0$.

   So, $F = F''\mathsf{f}$. We prove $\mathsf{f}_u \in (\!|F' F|\!)$.

We have $(\!|F'F|\!) = (\!|F'F''\mathsf{f}|\!)$. From (4.11), we have $(\!|F'F''\mathsf{f}|\!) = (\!|(\!|F'F''|\!)\mathsf{f}|\!)$. So, by transitivity, we have

$$(\!|F'F|\!) = (\!|(\!|F'F''|\!)\mathsf{f}|\!). \tag{A.112}$$

By IH on F'', we have that $\mathsf{f}_u \in (\!|F'F''|\!)$, and thus, $(\!|F'F''|\!) \neq \epsilon$.

2.1. $(\!|F'F''|\!) = F_a\mathsf{f}_a$ and $\mathsf{f} \neq \mathsf{f}_a^{\mathsf{c}}$.

Then $(\!|(\!|F'F''|\!)\mathsf{f}|\!) = (\!|F'F''|\!)\mathsf{f}$. Because $\mathsf{f}_u \in (\!|F'F''|\!)$, we get $\mathsf{f}_u \in (\!|(\!|F'F''|\!)\mathsf{f}|\!)$. From (A.112), we then get $\mathsf{f}_u \in (\!|F'F|\!)$.

2.2. $(\!|F'F''|\!) = F_a\mathsf{f}_a$ and $\mathsf{f} = \mathsf{f}_a^{\mathsf{c}}$.

Then $\mathsf{f}_a \neq \mathsf{f}_u$, because f_u does not have a complement, while f_a has a complement, namely f. Because $\mathsf{f}_u \in (\!|F'F''|\!)$ and $\mathsf{f}_a \neq \mathsf{f}_u$, we then get $\mathsf{f}_u \in F_a$.

We have $(\!|(\!|F'F''|\!)\mathsf{f}|\!) = (\!|F_a\mathsf{f}_a\mathsf{f}|\!) = (\!|F_a|\!) = F_a$, because F_a is by assumption reduced.

Because $\mathsf{f}_u \in F_a$, we then get $\mathsf{f}_u \in (\!|(\!|F'F''|\!)\mathsf{f}|\!)$. From (A.112), we then get $\mathsf{f}_u \in (\!|F'F|\!)$.

Thus, $\mathsf{f}_u \in (\!|F', F|\!)$.

We now prove $\mathcal{R}_\kappa(\langle(\!|F'F|\!), B\rangle) = P$. By definition of $\mathcal{R}_\kappa$, we have $\mathcal{R}_\kappa(\langle(\!|F'F|\!), B\rangle) = B \cup \bigcup_{\mathsf{f}\in(\!|F'F|\!)} \overline{\mathcal{X}}(\mathsf{f})$. Because $\mathsf{f}_u \in (\!|F'F|\!)$, we then have: $\mathcal{R}_\kappa(\langle(\!|F'F|\!), B\rangle) \supseteq \overline{\mathcal{X}}(\mathsf{f}_u)$. Because f_u does not have a complement, we get that $\mathcal{X}(\mathsf{f}_u) = \emptyset$. So, $\overline{\mathcal{X}}(\mathsf{f}_u) = P$. Thus, $\mathcal{R}_\kappa(\langle(\!|F'F|\!), B\rangle) \supseteq P$. So, $\mathcal{R}_\kappa(\langle(\!|F'F|\!), B\rangle) = P$. □

Proposition 4. *The decision algorithm for $\lambda_2 \sqsubseteq_\kappa \lambda_1$ is sound with respect to (4.19), but not complete. The decision algorithm for $\lambda_2 \sqsubseteq_\kappa \lambda_1$ is complete with respect to (4.19) when λ_1 is a singleton set.*

Proof. First we prove that the algorithm for $\lambda_2 \sqsubseteq_\kappa \lambda_1$ is sound with respect to (4.19). Assume the algorithm decides that $\lambda_2 \sqsubseteq_\kappa \lambda_1$ holds. So, the following holds:

$$\begin{aligned} &\forall \langle \boldsymbol{F}_1, \boldsymbol{B}_1 \rangle \in \lambda_1, \langle \boldsymbol{F}_2, \boldsymbol{B}_2 \rangle \in \lambda_2: \\ &\quad \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_2, \boldsymbol{B}_2 \rangle, \boldsymbol{F}_2^{\mathsf{c}})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_1, \boldsymbol{B}_1 \rangle, \boldsymbol{F}_2^{\mathsf{c}})) \end{aligned} \tag{A.113}$$

We prove $\forall \boldsymbol{F}$: $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_2, \boldsymbol{F})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_1, \boldsymbol{F}))$, which is the right-hand side of (4.19).

Fix $\boldsymbol{F}$. We prove $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_2, \boldsymbol{F})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_1, \boldsymbol{F}))$. From (A.113) and Proposition 3, we get:

$$\begin{aligned} &\forall \langle \boldsymbol{F}_1, B_1 \rangle \in \lambda_1, \langle \boldsymbol{F}_2, B_2 \rangle \in \lambda_2: \\ &\quad \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_2, B_2 \rangle, \boldsymbol{F})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_1, B_1 \rangle, \boldsymbol{F})) \end{aligned} \tag{A.114}$$

We have $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_1, \boldsymbol{F})) = \bigcap_{\langle \boldsymbol{F}_1, B_1 \rangle \in \lambda_1} \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_1, B_1 \rangle, \boldsymbol{F}))$.
Similarly, $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_2, \boldsymbol{F})) = \bigcap_{\langle \boldsymbol{F}_2, B_2 \rangle \in \lambda_2} \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_2, B_2 \rangle, \boldsymbol{F}))$.
Assume $\mathsf{p} \in \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_1, \boldsymbol{F}))$. We prove $\mathsf{p} \in \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_2, \boldsymbol{F}))$.
From $\mathsf{p} \in \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_1, \boldsymbol{F}))$, we get $\forall \langle \boldsymbol{F}_1, B_1 \rangle \in \lambda_1$: $\mathsf{p} \in \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_1, B_1 \rangle, \boldsymbol{F}))$. From (A.114), we then get $\forall \langle \boldsymbol{F}_2, B_2 \rangle \in \lambda_2$: $\mathsf{p} \in \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_2, B_2 \rangle, \boldsymbol{F}))$. So, $\mathsf{p} \in \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_2, \boldsymbol{F}))$.

Now, we prove that the algorithm for $\lambda_2 \sqsubseteq_\kappa \lambda_1$ is not complete. Let $\lambda_1 = \{\langle \epsilon, B_a \rangle, \langle \epsilon, B_b \rangle\}$ and $\lambda_2 = \{\langle \epsilon, B'_a \rangle, \langle \epsilon, B'_b \rangle\}$. Assume $B_a = \emptyset$, $B_b = \{\mathsf{p}\}$, $B'_a = \{\mathsf{p}\}$, and $B'_b = \emptyset$. We have: $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_1, \boldsymbol{F})) = \overline{\mathcal{X}}(\boldsymbol{F}) \cup B_a \cap B_b$ and $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_2, \boldsymbol{F})) = \overline{\mathcal{X}}(\boldsymbol{F}) \cup B'_a \cap B'_b$. So, (4.19) holds. But $\mathcal{R}_\kappa(\langle \epsilon, B_b \rangle) = \{\mathsf{p}\}$ and $\mathcal{R}_\kappa(\langle \epsilon, B'_b \rangle) = \emptyset$. So, $\mathcal{R}_\kappa(\langle \epsilon, B'_b \rangle) \not\supseteq \mathcal{R}_\kappa(\langle \epsilon, B_b \rangle)$. Thus, the algorithm decides that $\lambda_2 \sqsubseteq_\kappa \lambda_1$ does not hold.

Now, we prove that the decision algorithm for $\lambda_2 \sqsubseteq_\kappa \lambda_1$ is complete with respect to (4.19) when λ_1 is a singleton set. Assume $\lambda_1 = \{\langle \boldsymbol{F}_1, B_1 \rangle\}$. Assume

the right-hand side of (4.19) holds:

$$\forall \boldsymbol{F}\colon \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_2, \boldsymbol{F})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_1, \boldsymbol{F})) \tag{A.115}$$

We prove that the algorithm decides that $\lambda_2 \sqsubseteq_\kappa \lambda_1$ holds. So, we prove $\forall \langle \boldsymbol{F}_2, B_2\rangle \in \lambda_2\colon \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_2, \boldsymbol{B}_2\rangle, \boldsymbol{F}_2^{\mathsf{c}})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_1, \boldsymbol{B}_1\rangle, \boldsymbol{F}_2^{\mathsf{c}}))$.

Assume, for contradiction, that

$\exists \langle \boldsymbol{F}_2, B_2\rangle \in \lambda_2\colon \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_2, \boldsymbol{B}_2\rangle, \boldsymbol{F}_2^{\mathsf{c}})) \not\supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_1, \boldsymbol{B}_1\rangle, \boldsymbol{F}_2^{\mathsf{c}}))$.

We have $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_2, \boldsymbol{F}_2^{\mathsf{c}})) = \bigcap_{\langle \boldsymbol{F}_2, B_2\rangle \in \lambda_2} \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_2, B_2\rangle, \boldsymbol{F}_2^{\mathsf{c}}))$. From (A.115), we have $\bigcap_{\langle \boldsymbol{F}_2, B_2\rangle \in \lambda_2} \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_2, B_2\rangle, \boldsymbol{F}_2^{\mathsf{c}})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_1, \boldsymbol{F}_2^{\mathsf{c}}))$.

So, $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_2, B_2\rangle, \boldsymbol{F}_2^{\mathsf{c}})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda_1, \boldsymbol{F}_2^{\mathsf{c}}))$.

Thus $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_2, B_2\rangle, \boldsymbol{F}_2^{\mathsf{c}})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}_1, B_1\rangle, \boldsymbol{F}_2^{\mathsf{c}}))$, which is a contradiction. □

Proposition 5. *$\lambda \sqcup_\kappa \lambda'$ is the least upper bound of λ and λ'.*

Proof. First we prove that $\lambda \sqsubseteq_\kappa \lambda \sqcup_\kappa \lambda'$ and $\lambda' \sqsubseteq_\kappa \lambda \sqcup_\kappa \lambda'$.

Fix $\boldsymbol{F}$.

We have:

$$\begin{aligned}
\mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda \sqcup_\kappa \lambda', \boldsymbol{F})) &= \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda \cup \lambda', \boldsymbol{F})) \\
&= \bigcap_{\langle \boldsymbol{F}', B'\rangle \in \lambda \cup \lambda'} \widehat{\mathcal{R}}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}', B'\rangle, \boldsymbol{F})) \\
&= \bigcap_{\langle \boldsymbol{F}', B'\rangle \in \lambda} \widehat{\mathcal{R}}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}', B'\rangle, \boldsymbol{F})) \cap \bigcap_{\langle \boldsymbol{F}', B'\rangle \in \lambda'} \widehat{\mathcal{R}}_\kappa(\mathcal{T}_\kappa(\langle \boldsymbol{F}', B'\rangle, \boldsymbol{F})) \\
&= \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda, \boldsymbol{F})) \cap \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda', \boldsymbol{F}))
\end{aligned}$$

Thus

$$\mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda \sqcup_\kappa \lambda', \boldsymbol{F})) = \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda, \boldsymbol{F})) \cap \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda', \boldsymbol{F})) \tag{A.116}$$

So, $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda, \boldsymbol{F})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda \sqcup_\kappa \lambda', \boldsymbol{F}))$ and $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda', \boldsymbol{F})) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda \sqcup_\kappa \lambda', \boldsymbol{F}))$.

So, $\lambda \sqsubseteq_\kappa \lambda \sqcup_\kappa \lambda'$ and $\lambda' \sqsubseteq_\kappa \lambda \sqcup_\kappa \lambda'$.

Now, we prove that if $\lambda \sqsubseteq_\kappa \lambda''$ and $\lambda' \sqsubseteq_\kappa \lambda''$, then $\lambda'' \not\sqsubset_\kappa \lambda \sqcup_\kappa \lambda'$. Assume for contradiction that $\lambda'' \sqsubset_\kappa \lambda \sqcup_\kappa \lambda'$. So, there is an F such that we have $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda'', F)) \supset \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda \sqcup_\kappa \lambda', F))$. From (A.116), we then have

$$\mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda'', F)) \supset \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda, F)) \cap \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda', F)). \tag{A.117}$$

From $\lambda \sqsubseteq_\kappa \lambda''$ and $\lambda' \sqsubseteq_\kappa \lambda''$, we have $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda, F)) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda'', F))$ and $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda', F)) \supseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda'', F))$. So, $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda'', F)) \subseteq \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda, F)) \cap \mathcal{R}_\kappa(\mathcal{T}_\kappa(\lambda', F))$, which contradicts (A.117). So, we proved that $\lambda'' \not\sqsubset_\kappa \lambda \sqcup_\kappa \lambda'$. □

To prove that (4.11) holds, we introduce the folowing definitions:

- $F \overset{i}{\rightarrowtail} F'$ iff $F[i+1] = F[i]^{\mathsf{c}} \ \wedge \ F' = F[1..i-1]F[i+2..]$.
- $F \rightarrowtail F'$ iff

$$\exists i\colon 1 \leq i < |F|\colon (\forall j\colon 1 \leq j < i\colon F[j+1] \neq F[j]^{\mathsf{c}}) \\ \wedge \ F \overset{i}{\rightarrowtail} F'$$

  Define $F \overset{*}{\rightarrowtail} F''$ iff $F \rightarrowtail F' \rightarrowtail \ldots \rightarrowtail F''$.

  Notice that $\rightarrowtail$ is a deterministic function.

- We denote with G a sequence of reclassifiers that is reduced.

  Notice that if $F \overset{*}{\rightarrowtail} G$, then $(\!|F|\!) = G$.

Lemma 12. $F \overset{*}{\rightarrowtail} G \ \wedge \ F \overset{i}{\rightarrowtail} F' \overset{*}{\rightarrowtail} G' \ \Rightarrow \ G = G'$

Proof. By definition of $F \overset{i}{\rightarrowtail} F'$, we have that $1 \leq i < |F|$, $F[i+1] = F[i]^{\mathsf{c}}$, and $F' = F[1..i-1]F[i+2..]$. Let $F_1 = F[1..i-1]$ and $F_2 = F[i+2..]$. So, we have:

$$F = F_1 F[i..] \text{ and } F' = F_1 F_2. \tag{A.118}$$

We examine two cases based on the value of i.

1. $i = 1$

   So, we have that hypothesis $\mathsf{F}[i+1] = \mathsf{F}[i]^{\mathsf{c}}$ becomes $\mathsf{F}[2] = \mathsf{F}[1]^{\mathsf{c}}$, and hypothesis $\mathsf{F} \overset{i}{\rightarrowtail} \mathsf{F}' \overset{*}{\rightarrowtail} G'$ becomes $\mathsf{F} \overset{1}{\rightarrowtail} \mathsf{F}' \overset{*}{\rightarrowtail} G'$. By definition of $\rightarrowtail$ and because $\mathsf{F}[2] = \mathsf{F}[1]^{\mathsf{c}}$, we have that $\mathsf{F} \overset{1}{\rightarrowtail} \mathsf{F}' \overset{*}{\rightarrowtail} G'$ implies $\mathsf{F} \rightarrowtail \mathsf{F}' \overset{*}{\rightarrowtail} G'$. By definition of $\mathsf{F} \overset{*}{\rightarrowtail} G$ and because $\rightarrowtail$ is deterministic we then get $G = G'$.

2. $1 < i < |\mathsf{F}|$

   Due to (A.118), hypothesis $\mathsf{F} \overset{*}{\rightarrowtail} G$ becomes

   $$\mathsf{F}_1\mathsf{F}[i..] \overset{*}{\rightarrowtail} G \tag{A.119}$$

   and hypothesis $\mathsf{F} \overset{i}{\rightarrowtail} \mathsf{F}' \overset{*}{\rightarrowtail} G'$ becomes

   $$\mathsf{F}_1\mathsf{F}[i..] \overset{i}{\rightarrowtail} \mathsf{F}_1\mathsf{F}_2 \overset{*}{\rightarrowtail} G'. \tag{A.120}$$

   So, given (A.119) and (A.120), we prove $G = G'$.

   F_1 is either reduced or not reduced. So, there exists a reduced sequence G_1 such that either $G_1 = \mathsf{F}_1$ or $\mathsf{F}_1 \overset{*}{\rightarrowtail} G_1$. For sequence $\mathsf{F}_1\mathsf{F}[i..]$, we then get that either $\mathsf{F}_1\mathsf{F}[i..] = G_1\mathsf{F}[i..]$ (due to $G_1 = \mathsf{F}_1$) or $\mathsf{F}_1\mathsf{F}[i..] \overset{*}{\rightarrowtail} G_1\mathsf{F}[i..]$ (due to $\mathsf{F}_1 \overset{*}{\rightarrowtail} G_1$). From (A.119) and because $G_1\mathsf{F}[i..]$ is not reduced (since $\mathsf{F}[i+1] = \mathsf{F}[i]^{\mathsf{c}}$), we then get for both cases that the following holds:

   $$G_1\mathsf{F}[i..] \overset{*}{\rightarrowtail} G \tag{A.121}$$

   Also, for sequence $\mathsf{F}_1\mathsf{F}_2$, we then get that either $\mathsf{F}_1\mathsf{F}_2 = G_1\mathsf{F}_2$ (due to $G_1 = \mathsf{F}_1$) or $\mathsf{F}_1\mathsf{F}_2 \overset{*}{\rightarrowtail} G_1\mathsf{F}_2$ (due to $\mathsf{F}_1 \overset{*}{\rightarrowtail} G_1$). We examine these two cases.

   - $\mathsf{F}_1\mathsf{F}_2 = G_1\mathsf{F}_2$

     From $\mathsf{F}_1\mathsf{F}_2 \overset{*}{\rightarrowtail} G'$ in (A.120), we then get $G_1\mathsf{F}_2 \overset{*}{\rightarrowtail} G'$.

   - $\mathsf{F}_1\mathsf{F}_2 \overset{*}{\rightarrowtail} G_1\mathsf{F}_2$

     We examine two cases based on $G_1\mathsf{F}_2$.

– $G_1\mathsf{F}_2$ is reduced.

From $\mathsf{F}_1\mathsf{F}_2 \overset{*}{\rightarrowtail} G'$ in (A.120) and $\mathsf{F}_1\mathsf{F}_2 \overset{*}{\rightarrowtail} G_1\mathsf{F}_2$, we then get that $G' = G_1\mathsf{F}_2$.

– $G_1\mathsf{F}_2$ is not reduced.

From $\mathsf{F}_1\mathsf{F}_2 \overset{*}{\rightarrowtail} G'$ in (A.120) and $\mathsf{F}_1\mathsf{F}_2 \overset{*}{\rightarrowtail} G_1\mathsf{F}_2$, we then have $G_1\mathsf{F}_2 \overset{*}{\rightarrowtail} G'$.

So, we have that

$$\text{either } G_1\mathsf{F}_2 \overset{*}{\rightarrowtail} G' \text{ or } G' = G_1\mathsf{F}_2. \tag{A.122}$$

To show $G = G'$, it suffices to use (A.121), (A.122), and prove that $G_1\mathsf{F}[i..] \overset{*}{\rightarrowtail} G_1\mathsf{F}_2$ holds. So, we first show that (A.121), (A.122), and $G_1\mathsf{F}[i..] \overset{*}{\rightarrowtail} G_1\mathsf{F}_2$ imply $G = G'$; we then prove $G_1\mathsf{F}[i..] \overset{*}{\rightarrowtail} G_1\mathsf{F}_2$. We examine two cases.

- $G_1\mathsf{F}_2$ is reduced.

  From $G_1\mathsf{F}[i..] \overset{*}{\rightarrowtail} G_1\mathsf{F}_2$ and (A.121), we then get that $G = G_1\mathsf{F}_2$. From (A.122) and because $G_1\mathsf{F}_2$ is reduced, we get that $G' = G_1\mathsf{F}_2$ holds. By transitivity, we then get $G = G'$.

- $G_1\mathsf{F}_2$ is not reduced.

  From $G_1\mathsf{F}[i..] \overset{*}{\rightarrowtail} G_1\mathsf{F}_2$ and (A.121), we then get $G_1\mathsf{F}[i..] \overset{*}{\rightarrowtail} G_1\mathsf{F}_2 \overset{*}{\rightarrowtail} G$. From (A.122) and because $G_1\mathsf{F}_2$ is not reduced, we get that $G_1\mathsf{F}_2 \overset{*}{\rightarrowtail} G'$ holds. Because $\rightarrowtail$ is deterministic, we then get $G = G'$.

So, in both cases we have $G = G'$.

We now prove $G_1\mathsf{F}[i..] \overset{*}{\rightarrowtail} G_1\mathsf{F}_2$ by examining the cases below.

2.1. $G_1 = \epsilon$ or $G_1 = G_1'\mathsf{f}$ and $\mathsf{F}[i] \neq \mathsf{f}^{\mathsf{c}}$.

Because $\mathsf{F}[i+1] = \mathsf{F}[i]^{\mathsf{c}}$, we then have $G_1\mathsf{F}[i..] \rightarrowtail G_1\mathsf{F}_2$.

2.2. $G_1 = G_1'\mathsf{f}$ and $\boldsymbol{F}[i] = \mathsf{f}^{\mathsf{c}}$.

So, we have $G_1\boldsymbol{F}[i..] \rightarrowtail G_1'\boldsymbol{F}[i{+}1..]$. Because $\boldsymbol{F}[i] = \mathsf{f}^{\mathsf{c}}$ and $\boldsymbol{F}[i{+}1] = \boldsymbol{F}[i]^{\mathsf{c}}$, we get $\boldsymbol{F}[i+1] = \mathsf{f}$. So, $G_1'\boldsymbol{F}[i+1..] = G_1'\boldsymbol{F}[i+1]\boldsymbol{F}_2 = G_1\boldsymbol{F}_2$. Thus $G_1\boldsymbol{F}[i..] \rightarrowtail G_1\boldsymbol{F}_2$.

□

Proposition 6. $(\!|\boldsymbol{F}_1\boldsymbol{F}\boldsymbol{F}_2|\!) = (\!|\boldsymbol{F}_1(\!|\boldsymbol{F}|\!)\boldsymbol{F}_2|\!)$

Proof. If $\boldsymbol{F}$ is reduced, then it is trivially true, because $(\!|\boldsymbol{F}|\!) = \boldsymbol{F}$.

Assume $\boldsymbol{F}$ is not reduced. By definition of $\overset{*}{\rightarrowtail}$, there exists G' such that $\boldsymbol{F}_1\boldsymbol{F}\boldsymbol{F}_2 \overset{*}{\rightarrowtail} G'$ and $(\!|\boldsymbol{F}_1\boldsymbol{F}\boldsymbol{F}_2|\!) = G'$. Also, there exits G such that $\boldsymbol{F} \overset{*}{\rightarrowtail} G$ and $(\!|\boldsymbol{F}|\!) = G$. Thus $(\!|\boldsymbol{F}_1(\!|\boldsymbol{F}|\!)\boldsymbol{F}_2|\!) = (\!|\boldsymbol{F}_1G\boldsymbol{F}_2|\!)$. By definition of $\overset{*}{\rightarrowtail}$, there exists G'' such that $\boldsymbol{F}_1G\boldsymbol{F}_2 \overset{*}{\rightarrowtail} G''$ and $(\!|\boldsymbol{F}_1G\boldsymbol{F}_2|\!) = G''$.

To prove $(\!|\boldsymbol{F}_1\boldsymbol{F}\boldsymbol{F}_2|\!) = (\!|\boldsymbol{F}_1(\!|\boldsymbol{F}|\!)\boldsymbol{F}_2|\!)$, it suffices to prove $G' = G''$. So, given

$$\boldsymbol{F}_1\boldsymbol{F}\boldsymbol{F}_2 \overset{*}{\rightarrowtail} G',\ \boldsymbol{F}_1G\boldsymbol{F}_2 \overset{*}{\rightarrowtail} G'' \text{ and } \boldsymbol{F} \overset{*}{\rightarrowtail} G, \tag{A.123}$$

we prove that $G' = G''$ holds. We use induction on the number n of $\rightarrowtail$ steps in $\boldsymbol{F} \overset{*}{\rightarrowtail} G$, with the following induction hypothesis, where H is a sequence of cryptographic reclassifiers.

$$\text{IH: If } H \overset{*}{\rightarrowtail} G \text{ with } n-1 \text{ steps, } \boldsymbol{F}_1H\boldsymbol{F}_2 \overset{*}{\rightarrowtail} G',$$
$$\text{and } \boldsymbol{F}_1G\boldsymbol{F}_2 \overset{*}{\rightarrowtail} G'', \text{ then } G' = G''.$$

1. Base case: $\boldsymbol{F} \rightarrowtail G$

So, there is i such that $\boldsymbol{F} \overset{i}{\rightarrowtail} G$. Thus, we can write $\boldsymbol{F}_1\boldsymbol{F}\boldsymbol{F}_2 \overset{i+|\boldsymbol{F}_1|}{\rightarrowtail} \boldsymbol{F}_1G\boldsymbol{F}_2$. From $\boldsymbol{F}_1G\boldsymbol{F}_2 \overset{*}{\rightarrowtail} G''$ in (A.123), we then have $\boldsymbol{F}_1\boldsymbol{F}\boldsymbol{F}_2 \overset{i+|\boldsymbol{F}_1|}{\rightarrowtail} \boldsymbol{F}_1G\boldsymbol{F}_2 \overset{*}{\rightarrowtail} G''$. From Lemma 12 and $\boldsymbol{F}_1\boldsymbol{F}\boldsymbol{F}_2 \overset{*}{\rightarrowtail} G'$ in (A.123), we then get $G' = G''$.

2. Inductive case: $F \rightarrowtail F' \overset{*}{\rightarrowtail} G$

   So, there is i such that $F \overset{i}{\rightarrowtail} F' \overset{*}{\rightarrowtail} G$. Let $F_1F'F_2 \overset{*}{\rightarrowtail} G'''$. Because $F' \overset{*}{\rightarrowtail} G$ involves $n-1$ steps (since $F \rightarrowtail F' \overset{*}{\rightarrowtail} G$ involves n steps), $F_1F'F_2 \overset{*}{\rightarrowtail} G'''$, and $F_1GF_2 \overset{*}{\rightarrowtail} G''$ in (A.123), we can apply IH with $H = F'$ and get $G''' = G''$.

   Using $F \overset{i}{\rightarrowtail} F'$ from $F \overset{i}{\rightarrowtail} F' \overset{*}{\rightarrowtail} G$, we can write $F_1FF_2 \overset{i+|F_1|}{\longrightarrow} F_1F'F_2$. From $F_1F'F_2 \overset{*}{\rightarrowtail} G'''$, we then have $F_1FF_2 \overset{i+|F_1|}{\longrightarrow} F_1F'F_2 \overset{*}{\rightarrowtail} G'''$. From $F_1FF_2 \overset{*}{\rightarrowtail} G'$ in (A.123) and Lemma 12, we then have $G''' = G'$. By transitivity on $G''' = G''$ and $G''' = G'$, we get $G' = G''$.

□

A.4 Significance of Type-correctness with κ-labels

Theorem 2. *Consider a principal* $\mathsf{p} \in \mathsf{P}$ *and* κ*-label* λ_{p}*. Assume* $\Gamma, \lambda_\kappa \vdash \mathcal{C}$ *for a context-type* $\lambda_\kappa \in \Lambda_\kappa$ *and a mapping* Γ *that maps variables in a command* $\mathcal{C}$ *to* κ*-labels in* Λ_κ*. Let* $\Theta(\mathcal{E}, K)$ *perform a* λ_{p}*-downgrade in* $\mathcal{C}$*. Then,* $\mathsf{p} \in \overline{\mathcal{X}}(\Theta(\cdot, K))$ *and* $\mathsf{p} \in \overline{\mathcal{X}}(\Theta(\mathcal{E}, \cdot))$*.*

Proof. By the definition of λ_{p}-downgrade, and because $\Theta(\mathcal{E}, K)$ performs a λ_{p}-downgrade, we conclude that $\mathcal{E}$ or K is high and $\Theta(\mathcal{E}, K)$ is low.

We first show that $\mathsf{p} \in \overline{\mathcal{X}(\Theta(\mathcal{E}, \cdot))}$. Because no principal can complement $\Theta(\mathcal{E}, \cdot)$, we have $\overline{\mathcal{X}(\Theta(\mathcal{E}, \cdot))} = \mathsf{P}$. So, $\mathsf{p} \in \overline{\mathcal{X}(\Theta(\mathcal{E}, \cdot))}$.

We now show that $\mathsf{p} \in \overline{\mathcal{X}(\Theta(\cdot, K))}$. Because $\Theta(\mathcal{E}, K)$ is low, we have that $\mathsf{p} \in \mathcal{R}_\kappa(\Gamma(\Theta(\mathcal{E}, K)))$. By the rule AnnExpr-T, we have

$$\Gamma(\Theta(\mathcal{E}, K)) = \mathcal{T}_\kappa(\Gamma(\mathcal{E}), \Theta(\cdot, K)) \sqcup_\kappa \mathcal{T}_\kappa(\Gamma(K), \Theta(\mathcal{E}, \cdot)).$$

By definition of $\mathcal{R}_\kappa$, we then get that

$$\mathcal{R}_\kappa(\Gamma(\Theta(\mathcal{E}, K))) = \mathcal{R}_\kappa(\mathcal{T}_\kappa(\Gamma(\mathcal{E}), \Theta(\cdot, K))) \cap \mathcal{R}_\kappa(\mathcal{T}_\kappa(\Gamma(K), \Theta(\mathcal{E}, \cdot))).$$

Because $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\Gamma(K), \Theta(\mathcal{E}, \cdot))) = \mathsf{P}$, since no principal can complement $\Theta(\mathcal{E}, \cdot)$, we then get

$$\mathcal{R}_\kappa(\Gamma(\Theta(\mathcal{E}, K))) = \mathcal{R}_\kappa(\mathcal{T}_\kappa(\Gamma(\mathcal{E}), \Theta(\cdot, K))).$$

From $\mathsf{p} \in \mathcal{R}_\kappa(\Gamma(\Theta(\mathcal{E}, K)))$ we then have

$$\mathsf{p} \in \mathcal{R}_\kappa(\mathcal{T}_\kappa(\Gamma(\mathcal{E}), \Theta(\cdot, K))). \tag{A.124}$$

Because $\mathcal{E}$ is high, we have

$$\mathsf{p} \notin \mathcal{R}_\kappa(\Gamma(\mathcal{E})). \tag{A.125}$$

Due to the definition of $\mathcal{R}_\kappa$ and $\mathcal{T}_\kappa$, (A.124) and (A.125), it should be the case that $\mathcal{R}_\kappa(\mathcal{T}_\kappa(\Gamma(\mathcal{E}), \Theta(\cdot, K))) = \mathcal{R}_\kappa(\Gamma(\mathcal{E})) \cup \overline{\mathcal{X}(\Theta(\cdot, K))}$. From the above equation, (A.124), and (A.125) we get $\mathsf{p} \in \overline{\mathcal{X}(\Theta(\cdot, K))}$. □

APPENDIX B

PROOFS FOR LABEL CHAINS

B.1 Definitions

The following definitions are used in the proofs appearing in this appendix.

- We abbreviate $\tau|_\ell^S$ by $\tau|_\ell$, when S is the set of all variables and their tags.
- We extend the projection of a memory with respect to a label ℓ to include bc (blocking context) and cc (conditional context):

$$\begin{aligned} &M|_\ell = \\ &\{\langle q, M(q)\rangle \mid q, T(q) \in dom(M) \ \wedge \ M(T(q)) \sqsubseteq \ell\} \cup \\ &\{\langle bc, M(bc)\rangle \mid M(bc) \sqsubseteq \ell\} \cup \\ &\{\langle cc, M(cc)\rangle \mid M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq \ell\} \end{aligned} \tag{B.1}$$

- Define equality of sequences of observations when omitting the empty sets:

$$\epsilon =_{obs} \epsilon$$

$$\emptyset =_{obs} \epsilon$$

$$\epsilon =_{obs} \emptyset$$

$$\Theta \rightharpoonup \theta =_{obs} \Theta' \rightharpoonup \theta' \text{ iff}$$

$$\begin{cases} \Theta = \Theta' \ \wedge \ \theta =_{obs} \theta', \text{ or} \\ \Theta = \emptyset \ \wedge \ \Theta' \neq \emptyset \ \wedge \ \theta =_{obs} \Theta' \rightharpoonup \theta', \text{ or} \\ \Theta \neq \emptyset \ \wedge \ \Theta' = \emptyset \ \wedge \ \Theta \rightharpoonup \theta =_{obs} \theta' \end{cases}$$

- Define $kstut(M)$ to hold when the k^{th} label in all label chains in M is infinitely repeated:

  $kstut(M) \triangleq$

  $$\forall x{\in}dom(M){:}\ \forall i > k{:}\ T^i(x){\in}dom(M) \Rightarrow M(T^i(x)) = M(T^k(x))$$

  for $k \geq 1$

- We write $mon(M)$ to denote that all label chains in M are monotonically decreasing:

  $mon(M) \triangleq$

  $$\forall x{\in}dom(M){:}\ \forall i \geq 1{:}\ T^{i+1}(x){\in}dom(M) \Rightarrow M(T^{i+1}(x)) \sqsubseteq M(T^i(x))$$

- We write $M =_k M'$ iff

  1. $\forall x{:}\ \forall 0 \leq i \leq k{:}\ M(T^i(x)) = M'(T^i(x))$,
  2. $M(cc) = M'(cc)$, and
  3. $M(bc) = M'(bc)$.

- To prove inductively that our enforcers satisfy BNI, we strengthen BNI to BNI+ and prove BNI+ instead.

  BNI+$(E, \mathcal{L}, C)$**.** $\forall \ell \in \mathcal{L}{:}\ \forall M, M'{:}$

  If

  $$M \models \mathcal{H}(E, \mathcal{L}, C)\ \wedge\ M' \models \mathcal{H}(E, \mathcal{L}, C),$$
  $$M|_\ell = M'|_\ell, M(cc) = M'(cc), mon(M), mon(M')$$
  $$\tau = trace_E(C, M),$$
  $$\tau' = trace_E(C, M'),$$

  where $\tau = \langle C, M\rangle \xrightarrow{*} \langle C_t, M_t\rangle$, $\tau' = \langle C, M'\rangle \xrightarrow{*} \langle C'_t, M'_t\rangle$, *and* C_t, C'_t *are terminations (i.e.,* **stop** *or* **block***),*

  then:

$c1$ *If* C_t *and* C'_t *are both* **stop**, *then* $\tau|_\ell =_{obs} \tau'|_\ell$, $M_t|_\ell = M'_t|_\ell$, *and* $M_t(cc) = M'_t(cc)$.

$c2$ *If* C_t *or* C'_t *is* **block**, *then* $\tau|_\ell =_{obs} \tau'|_\ell$.

$c3$ *If* C_t *is* **stop**, C'_t *is* **block**, *and* $M'_t(bc) \not\sqsubseteq \ell$, *then* $M_t(bc) \not\sqsubseteq \ell$.

$c4$ *If* C_t *is* **stop**, C'_t *is* **block**, $\langle C'_{tp}, M'_{tp}\rangle \rightarrow \langle C'_t, M'_t\rangle$ *are the last two states of* τ', *and* $M'_{tp}(\lfloor cc \rfloor) \sqsubseteq \ell$, *then there exists* $\langle C'', M''\rangle \in \tau$, *with* $C'' = C'_{tp}$ *and* $M''|_\ell = M'_{tp}|_\ell$.

- For enforcers k-*Enf* with $2 \leq k \leq \infty$, $E_{\mathsf{H,L}}$, and k-*Eopt* with $k \geq 2$, we extend the domain of function $trace_E(C, M)$ to also include any memory M such that $M \models \mathcal{H}(E, \mathcal{L}, C)$ holds.
  Also, if $M \models \mathcal{H}(E, \mathcal{L}, C)$ holds, then for any M' in $trace_E(C, M)$ and any subcommand C' of C, we have $M' \models \mathcal{H}(E, \mathcal{L}, C')$ (based on the corresponding operational semantics).

B.2 Soundness of ∞-*Enf* and k-*Enf* for $k \geq 2$

Theorem 3. ∞-*Enf is an enforcer on* R *for BNI.*

Proof. It is easy to prove that ∞-*Enf* is an enforcer on R and satisfies restrictions (E1), (E2), and (E3) by induction on the rules of ∞-*Enf*. We omit the details.

To prove that ∞-*Enf* satisfies BNI, we will prove that k-BNI(∞-*Enf*$, \mathcal{L}, C$) holds for a lattice $\mathcal{L}$, a command C, and $k \geq 0$. From Lemma 13, we have that BNI+(∞-*Enf*,$\mathcal{L}$,C) holds. By definition, $M \models \mathcal{H}_0(\infty\text{-}Enf, \mathcal{L}, C)$ and $M' \models \mathcal{H}_0(\infty\text{-}Enf, \mathcal{L}, C)$ imply $M(cc) = M'(cc), mon(M), mon(M')$. Also, $\tau|_\ell =_{obs} \tau'|_\ell$

implies $\tau|^k_\ell =_{obs} \tau'|^k_\ell$. Thus, BNI+($\infty$-*Enf*,$\mathcal{L}$,$C$) implies k-BNI(∞-*Enf*$, \mathcal{L}, C$). Because k, C, and $\mathcal{L}$ were arbitrary, we then get that ∞-*Enf* enforces BNI. □

Lemma 13. *For a command C and lattice $\mathcal{L}$, BNI+(∞-Enf,$\mathcal{L}$,C) holds.*

Proof. Let $\ell \in \mathcal{L}$. We use structural induction on C.

1. C is **skip**:

   From rule SKIP, we get $C_t = C'_t = \mathbf{stop}$. So, $c2$, $c3$, and $c4$ are trivially true.

   We prove $c1$. We have, $\tau|_\ell =_{obs} \epsilon$ and $\tau'|_\ell =_{obs} \epsilon$. Because $M_t = M$ and $M'_t = M'$, we get $M_t|_\ell = M'_t|_\ell$ and $M_t(cc) = M'_t(cc)$. So, $c1$ holds.

2. C is $a := e$:

   From $M|_\ell = M'|_\ell$ and $M(T^2(a)) = M'(T^2(a)) = \bot$, we get

   $$M(T(a)) = M'(T(a)). \tag{B.2}$$

   2.1. $M(T(a)) \sqsubseteq \ell$

   We first prove that the command is executed normally in both memories or blocked in both memories. W.l.o.g, assume that the command is executed normally in M. We prove that the command is executed normally in M', too. Because the command is executed normally in M, rule ASGNA in Figure 5.3 has been triggered, meaning that $M(T(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq M(T(a))$ holds. From hypothesis (*2.1.*), we then get $M(T(e)) \sqsubseteq \ell$, $M(\lfloor cc \rfloor) \sqsubseteq \ell$, and $M(bc) \sqsubseteq \ell$. Because $M|_\ell = M'|_\ell$, we then get $M(cc) = M'(cc)$ and $M(bc) = M'(bc)$. From $mon(M)$ and $M(T(e)) \sqsubseteq \ell$, we get $M(T^2(e)) \sqsubseteq \ell$ and $M(T^3(e)) \sqsubseteq \ell$. From Lemma 17 and $M|_\ell = M'|_\ell$, we then get $M(T^2(e)) = M'(T^2(e))$, $M(T(e)) = M'(T(e))$, $M(e) = M'(e)$. From (B.2), we then get that

$M'(T(e)) \sqcup M'(\lfloor cc \rfloor) \sqcup M'(bc) \sqsubseteq M'(T(a))$ holds. So, rule ASGNA is triggered, meaning that the command is executed normally in M'. Taking the contrapositive of the statement we just proved (i.e., if the command is executed normally in M, then it will be executed normally in M'), we get that if the command is blocked in M', then it will be blocked in M. Because M and M' are arbitrary, we consequently have that the command is either (i) executed normally in both memories or (ii) blocked in both memories. So, $c3$ and $c4$ are trivially true. To prove $c1$ and $c2$, we examine cases (i) and (ii).

2.1.1. The command is executed normally in both memories.

$c2$ is trivially true.

We prove $c1$. We have:

$\tau = \langle a := e, \mathcal{M} \rangle \rightarrow \langle \mathbf{stop}, M[a \mapsto M(e), bc \mapsto \ell_g] \rangle$

$\tau' = \langle a := e, \mathcal{M}' \rangle \rightarrow \langle \mathbf{stop}, M'[a \mapsto M'(e), bc \mapsto \ell'_g] \rangle$,

where

$$\ell_g = M(T^2(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc) \text{ and}$$
$$\ell'_g = M'(T^2(e)) \sqcup M'(\lfloor cc \rfloor) \sqcup M'(bc).$$

We have $\tau|_\ell = \tau'|_\ell = \langle a, M(e) \rangle$, because $M(e) = M'(e)$. Also, $\ell_g = \ell'_g$. So, $M_t(bc) = M'_t(bc)$. Because $M_t(cc) = M(cc) = M'(cc) = M'_t(cc)$, then $M_t(cc) = M'_t(cc)$ holds. Because $M_t(a) = M'_t(a)$, $M_t(bc) = M'_t(bc)$, and $M_t(cc) = M'_t(cc)$, then we get $M_t|_\ell = M'_t|_\ell$. So $c1$ holds.

2.1.2. The command is blocked in both memories.

$\tau = \langle a := e, \mathcal{M} \rangle \rightarrow \langle \mathbf{block}, M[bc \mapsto \ell_g] \rangle$

$\tau' = \langle a := e, \mathcal{M}' \rangle \rightarrow \langle \mathbf{block}, M'[bc \mapsto \ell'_g] \rangle$.

So, $\tau|_\ell = \tau'|_\ell = \epsilon$, and thus $c2$ holds. Also, $c1$ is trivially true.

2.2. $M(T(a)) \not\sqsubseteq \ell$

We then have $\tau|_\ell =_{obs} \epsilon$ and $\tau'|_\ell =_{obs} \epsilon$, and thus $c2$ holds.

To prove $c1$, assume $C_t = C'_t = \textbf{stop}$. We show $M_t(cc) = M'_t(cc)$ and $M_t|_\ell = M'_t|_\ell$.

Because $M(cc) = M'(cc)$ and $M'_t(cc) = M'(cc)$, we have $M_t(cc) = M'_t(cc)$.

We now prove $M_t|_\ell = M'_t|_\ell$ as per (B.1). Because $M(T(a)), M'(T(a)) \not\sqsubseteq \ell$, then M_t and M'_t do not need to agree on a. If $M_t(bc) \not\sqsubseteq \ell$, then $M_t|_\ell = M'_t|_\ell$ trivially holds. Assume instead $M_t(bc) \sqsubseteq \ell$. So, rule ASGNA in Figure 5.3 then gives $M(T^2(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq \ell$. Thus, $M(T^2(e)) \sqsubseteq \ell$, $M(\lfloor cc \rfloor) \sqsubseteq \ell$, and $M(bc) \sqsubseteq \ell$. From $mon(M)$ and $M(T^2(e)) \sqsubseteq \ell$, we get $M(T^3(e)) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$, $M(T^3(e)) \sqsubseteq \ell$, and Lemma 17, we get $M(T^2(e)) = M'(T^2(e))$. From $M|_\ell = M'|_\ell$, $M(\lfloor cc \rfloor) \sqsubseteq \ell$, and $M(bc) \sqsubseteq \ell$, we get: $M(\lfloor cc \rfloor) = M'(\lfloor cc \rfloor)$ and $M(bc) = M'(bc)$. Because ASGNA in Figure 5.3 gives $M_t(bc) = M(T^2(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)$ and $M'_t(bc) = M'(T^2(e)) \sqcup M'(\lfloor cc \rfloor) \sqcup M'(bc)$, we then have $M_t(bc) = M'_t(bc)$. From $M_t(cc) = M'_t(cc)$, we then get $M_t|_\ell = M'_t|_\ell$. So, $c1$ holds.

To prove $c3$, assume C_t is **stop**, C'_t is **block**, and $M'_t(bc) \not\sqsubseteq \ell$. We must show $M_t(bc) \not\sqsubseteq \ell$. We show the contrapositive. Assume $M_t(bc) \sqsubseteq \ell$, then following the same arguments as above, we get $M_t(bc) = M'_t(bc)$, and thus, $M'_t(bc) \sqsubseteq \ell$, as wanted. So, $c3$ holds.

To prove $c4$, assume C_t is **stop**, C'_t is **block**, $\langle C'_{tp}, M'_{tp} \rangle \rightarrow \langle C'_t, M'_t \rangle$ are the last two states of τ', and $M'_{tp}(\lfloor cc \rfloor) \sqsubseteq \ell$. So, $M'_{tp} = M'$ and $C'_{tp} = a := e$. We have that $\langle C'', M'' \rangle$ in $c4$ is $\langle a := e, M \rangle$, which satisfies $C'' = C'_{tp}$ and $M''|_\ell = M'_{tp}|_\ell$. Thus $c4$ holds.

3. C is $w := e$

$\tau = \langle w := e, M\rangle \to \langle \mathbf{stop}, M_t\rangle$

$\tau' = \langle w := e, M'\rangle \to \langle \mathbf{stop}, M'_t\rangle$

$c2$, $c3$, and $c4$ are trivially true.

We prove $c1$. $M_t(cc) = M'_t(cc)$ holds because we have $M(cc) = M'(cc)$, $M_t(cc) = M(cc)$, and $M'_t(cc) = M'(cc)$.

3.1. $\exists r \geq 1$: $M_t(T^r(w)) \sqsubseteq \ell$

From $mon(M)$, we then get $\forall i \geq r$: $M_t(T^i(w)) \sqsubseteq \ell$. Then we have $\forall i \geq r$: $M(T^i(e)) \sqsubseteq \ell$, $M(\lfloor cc \rfloor) \sqsubseteq \ell$, and $M(bc) \sqsubseteq \ell$. From, $M|_\ell = M'|_\ell$ and Lemma 17, we then get $\forall i \geq r-1 : M(T^i(e)) = M'(T^i(e))$, $M(cc) = M'(cc)$, and $M(bc) = M'(bc)$. So, $\forall i \geq r-1 : M_t(T^i(w)) = M'_t(T^i(w))$, and thus, $\forall i \geq r$: $M'_t(T^i(w)) \sqsubseteq \ell$.

- If $r = 1$, we then get $M_t|_\ell = M'_t|_\ell$ and $\tau|_\ell = \tau'|_\ell$. Thus $c1$ holds.
- Assume $r > 1$ holds. Then we have $\forall i$: $1 < i < r$: $M_t(T^i(w)) \not\sqsubseteq \ell$. Because $\forall i \geq r-1 : M_t(T^i(w)) = M'_t(T^i(w))$, we then get $M'_t(T^{r-1}(w)) \not\sqsubseteq \ell$. From $mon(M')$ we then get $\forall i < r : M'_t(T^i(w)) \not\sqsubseteq \ell$. Thus, $M_t|_\ell = M'_t|_\ell$ and $\tau|_\ell = \tau'|_\ell$. Thus $c1$ holds.

3.2. $\forall i \geq 1$: $M_t(T^i(w)) \not\sqsubseteq \ell$

By symmetry of preceding case, $\forall i \geq 1$: $M'_t(T^i(w)) \not\sqsubseteq \ell$. So,$\tau|_\ell =_{obs} \epsilon$ and $\tau'|_\ell =_{obs} \epsilon$. Because $\forall i \geq 1$: $M_t(T^i(w)) \not\sqsubseteq \ell$ and $\forall i \geq 1$: $M'_t(T^i(w)) \not\sqsubseteq \ell$ holds, we get $M_t|_\ell = M'_t|_\ell$. Thus $c1$ holds.

4. $C_1; C_2$

4.1. C_1 terminates normally in τ and τ'.

Let

$\tau = \langle C_1; C_2, M\rangle \xrightarrow{*} \langle C_2, M_2\rangle \xrightarrow{*} \langle C_{2t}, M_t\rangle$ and

$\tau' = \langle C_1; C_2, M'\rangle \xrightarrow{*} \langle C_2, M'_2\rangle \xrightarrow{*} \langle C'_{2t}, M'_t\rangle$.

Consider:

$\tau_1 = \langle C_1, M\rangle \xrightarrow{*} \langle \mathbf{stop}, M_2\rangle$,

$\tau_2 = \langle C_2, M_2\rangle \xrightarrow{*} \langle C_{2t}, M_t\rangle$,

$\tau_1' = \langle C_1, M'\rangle \xrightarrow{*} \langle \mathbf{stop}, M_2'\rangle$,

$\tau_2' = \langle C_2, M_2'\rangle \xrightarrow{*} \langle C_{2t}', M_t'\rangle$.

From $c1$ of IH on C_1, we get $\tau_1|_\ell =_{obs} \tau_1'|_\ell$, $M_2|_\ell = M_2'|_\ell$, and $M_2(cc) = M_2'(cc)$. From $mon(M)$, $mon(M')$ and Lemma 14, we get $mon(M_2)$, $mon(M_2')$. So, we can apply IH on C_2.

To prove $c1$, assume $C_{2t} = C_{2t}' = \mathbf{stop}$. From IH on C_2, we get $\tau_2|_\ell =_{obs} \tau_2'|_\ell$, $M_t|_\ell = M_t'|_\ell$, and $M_t(cc) = M_t'(cc)$. From $\tau_1|_\ell =_{obs} \tau_1'|_\ell$ and $\tau_2|_\ell =_{obs} \tau_2'|_\ell$, we get $\tau|_\ell =_{obs} \tau'|_\ell$. Thus $c1$ holds.

To prove $c2$, assume $C_{2t} = \mathbf{block}$ or $C_{2t}' = \mathbf{block}$. From IH on C_2, we get $\tau_2|_\ell =_{obs} \tau_2'|_\ell$. From $\tau_1|_\ell =_{obs} \tau_1'|_\ell$ and $\tau_2|_\ell =_{obs} \tau_2'|_\ell$, we get $\tau|_\ell =_{obs} \tau'|_\ell$. Thus $c2$ holds.

To prove $c3$, assume $C_{2t} = \mathbf{stop}$, $C_{2t}' = \mathbf{block}$, and $M_t'(bc) \not\sqsubseteq \ell$. From IH on C_2, we get $M_t(bc) \not\sqsubseteq \ell$. Thus $c3$ holds.

To prove $c4$, assume $C_{2t} = \mathbf{stop}$, $C_{2t}' = \mathbf{block}$, $\langle C_{tp}', M_{tp}'\rangle \rightarrow \langle C_{2t}', M_t'\rangle$ are the last two states of τ', and $M_{tp}'(\lfloor cc \rfloor) \sqsubseteq \ell$. Then $\langle C_{tp}', M_{tp}'\rangle \rightarrow \langle C_{2t}', M_t'\rangle$ are the last two states of τ_2'. From IH on C_2, we get that there exists $\langle C'', M''\rangle \in \tau_2$, with $C'' = C_{tp}'$ and $M''|_\ell = M_{tp}'|_\ell$. Thus, there exists $\langle C'', M''\rangle \in \tau$, with $C'' = C_{tp}'$ and $M''|_\ell = M_{tp}'|_\ell$. Thus $c4$ holds.

4.2. C_1 is blocked in both τ, τ'

Similar to the above case, we apply IH on C_1.

4.3. C_1 is blocked in τ, terminates normally in τ' (C_2 may term/block in τ').

$c1$ is trivially true.

Let:

$\tau = \langle C_1; C_2, M\rangle \xrightarrow{*} \langle C_{1t}; C_2, M_t\rangle$ and

$\tau' = \langle C_1; C_2, M'\rangle \xrightarrow{*} \langle C_2, M_2'\rangle \xrightarrow{*} \langle C_{2t}', M_t'\rangle$.

Consider:

$\tau_1 = \langle C_1, M\rangle \xrightarrow{*} \langle C_{1t}, M_t\rangle$,

$\tau_1' = \langle C_1, M'\rangle \xrightarrow{*} \langle \mathbf{stop}, M_2'\rangle$,

$\tau_2' = \langle C_2, M_2'\rangle \xrightarrow{*} \langle C_{2t}', M_t'\rangle$.

So, we have

$$\tau|_\ell = \tau_1|_\ell \text{ and } \tau'|_\ell = \tau_1'|_\ell \rightharpoonup \tau_2'|_\ell. \tag{B.3}$$

We prove $c3$. We have $C_t = C_{1t} = \mathbf{block}$. Assume $C_t' = C_{2t}' = \mathbf{stop}$ and $M_t(bc) \not\sqsubseteq \ell$. We prove $M_t'(bc) \not\sqsubseteq \ell$. For IH on C_1, we get $M_2'(bc) \not\sqsubseteq \ell$. From Lemma 19, we then get $M_t'(bc) \not\sqsubseteq \ell$. Thus $c3$ holds.

We prove $c4$. We have $C_t = \mathbf{block}$. Assume $C_{2t}' = \mathbf{stop}$, $\langle C_{tp}; C_2, M_{tp}\rangle \rightarrow \langle C_{1t}; C_2, M_t\rangle$ are the last two states of τ, and $M_{tp}(\lfloor cc \rfloor) \sqsubseteq \ell$. Then $\langle C_{tp}, M_{tp}\rangle \rightarrow \langle C_{1t}, M_t\rangle$ are the last two states of τ_1. From IH on C_1, we get that there exists $\langle C'', M''\rangle \in \tau_1'$, with $C'' = C_{tp}$ and $M''|_\ell = M_{tp}|_\ell$. Thus, there exists $\langle C''; C_2, M''\rangle \in \tau'$, with $C'' = C_{tp}$ and $M''|_\ell = M_{tp}|_\ell$. Thus $c4$ holds.

We prove $c2$. From IH on C_1, we get we get $\tau_1|_\ell =_{obs} \tau_1'|_\ell$. Given also (B.3), to prove $\tau|_\ell =_{obs} \tau'|_\ell$, it suffices that $\tau_2'|_\ell = \epsilon$. So, we prove $\tau_2'|_\ell = \epsilon$. Assume the last transition of τ_1 is $\langle C_{1tp}, M_{tp}\rangle \rightarrow \langle C_{1t}, M_t\rangle$. τ_1 is blocked due to AsgnaFail, so C_{1tp} is $a := e; C'$.

4.3.1. $M_{tp}(bc) \not\sqsubseteq \ell$

From Lemma 19 and $\langle C_{1tp}, M_{tp}\rangle \rightarrow \langle C_{1t}, M_t\rangle$, we have $M_{tp}(bc) \sqsubseteq M_t(bc)$. Hypothesis (*4.3.1.*) then gives $M_t(bc) \not\sqsubseteq \ell$. From IH[$c3$] on C_1, we get $M_2'(bc) \not\sqsubseteq \ell$. From Lemma 16, $\tau_2'|_\ell = \epsilon$. Thus $c2$ holds.

4.3.2. $M_{tp}(bc) \sqsubseteq \ell$ and $M_{tp}(cc) \sqsubseteq \ell$

From IH[$c4$] on C_1, there exists $\langle C_{1tp}, M_1' \rangle \in \tau_1'$ such that $M_{tp}|_\ell = M_1'|_\ell$. So, $M_{tp}(bc) = M_1'(bc)$ and $M_{tp}(cc) = M_1'(cc)$. Because τ_1 is blocked, we have $M_{tp}(T(e)) \sqcup M_{tp}(\lfloor cc \rfloor) \sqcup M_{tp}(bc) \not\sqsubseteq M_{tp}(T(a))$. Since the inequality is satisfied in τ_1', it means that the value of $T(e)$ is different in M_1' and M_{tp}. So, $M_1'(T^2(e)) \not\sqsubseteq \ell$. Consider $\langle C_1, M' \rangle \xrightarrow{*} \langle C_{1tp}, M_1' \rangle \rightarrow \langle C_n, M_n \rangle$ a prefix of τ_1'. From ASGNA, we then have $M_n(bc) \not\sqsubseteq \ell$. From Lemma 19, we then get $M_2'(bc) \not\sqsubseteq \ell$. From Lemma 16, we then have $\tau_2'|_\ell = \epsilon$. Thus $c2$ holds.

4.3.3. $M_{tp}(bc) \sqsubseteq \ell$ and $M_{tp}(\lfloor cc \rfloor) \not\sqsubseteq \ell$

From ASGNAFAIL, $M_{tp}(\lfloor cc \rfloor) \sqsubseteq M_t(bc)$. So, $M_t(bc) \not\sqsubseteq \ell$. We work similarly to case *4.3.1.*.

5. **if** e **then** C_1 **else** C_2 **end**

5.1. $M(\lfloor cc \rfloor) \sqsubseteq \ell$ and $M(T(e)) \sqsubseteq \ell$

From $mon(M)$ and $M(T(e)) \sqsubseteq \ell$, we have $M(T^2(e)) \sqsubseteq \ell$. Because $M|_\ell = M'|_\ell$ and Lemma 17, we then have $M(T(e)) = M'(T(e))$ and $M(e) = M'(e)$. So, τ and τ' get the same branch, say C_1.

$\tau = \langle \textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M \rangle \rightarrow \langle C_1; \textbf{exit}, M_1 \rangle \xrightarrow{*} \langle C_t, M_t \rangle$

$\tau' = \langle \textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M' \rangle \rightarrow \langle C_1; \textbf{exit}, M_1' \rangle \xrightarrow{*} \langle C_t', M_t' \rangle$.

From $M_1(cc) = M(cc).push(\langle M(T(e)), W, A \rangle)$, $M(cc) = M'(cc)$, $M(T(e)) = M'(T(e))$, and $M_1'(cc) = M'(cc).push(\langle M'(T(e)), W, A \rangle)$, we get $M_1(cc) = M_1'(cc)$.

We prove $M_1|_\ell = M_1'|_\ell$ as per (B.1). Because $M|_\ell = M'|_\ell$, it is trivially true when $M_1(bc) \not\sqsubseteq \ell$. Assume $M_1(bc) \sqsubseteq \ell$. Because $M_1(cc) = M_1'(cc)$ holds, it suffices to also prove $M_1(bc) = M_1'(bc)$. Because $M_1(bc) = M(bc)$, we then get $M(bc) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$, we then get $M(bc) = M'(bc)$. Because $M_1'(bc) = M'(bc)$, we then get $M_1(bc) = M_1'(bc)$. So, $M_1|_\ell = M_1'|_\ell$.

From $mon(M)$, $mon(M')$ and Lemma 14, we get $mon(M_1)$, $mon(M'_1)$. We apply IH on $C_1; \textbf{exit}$ and get $c1$, $c2$, $c3$, and $c4$.

5.2. $M(\lfloor cc \rfloor) \not\sqsubseteq \ell$ or $M(T(e)) \not\sqsubseteq \ell$

We first prove that $M'(\lfloor cc \rfloor) \not\sqsubseteq \ell$ or $M'(T(e)) \not\sqsubseteq \ell$ holds. If $M(T(e)) \not\sqsubseteq \ell$, then from $M|_\ell = M'|_\ell$ and Lemma 18, we get $M'(T(e)) \not\sqsubseteq \ell$. Now, if $M(\lfloor cc \rfloor) \not\sqsubseteq \ell$, then $M(cc) = M'(cc)$ gives $M'(\lfloor cc \rfloor) \not\sqsubseteq \ell$. Thus, we have $M'(\lfloor cc \rfloor) \not\sqsubseteq \ell$ or $M'(T(e)) \not\sqsubseteq \ell$.

So, Lemma 15 gives:

(i) $\tau|_\ell =_{obs} \epsilon$ and $\tau'|_\ell =_{obs} \epsilon$.

(ii) If $C_t = \textbf{stop}$ (or $C'_t = \textbf{stop}$), and $w \in targetFlex(C)$, then $\forall i$: $M_t(T^i(w)) \not\sqsubseteq \ell$ (or $\forall i : M'_t(T^i(w)) \not\sqsubseteq \ell$).

(iii) If $C_t = \textbf{stop}$ (or $C'_t = \textbf{stop}$), and $targetAnchor(C) \neq \emptyset$ then $M_t(bc) \not\sqsubseteq \ell$ or ($M'_t(bc) \not\sqsubseteq \ell$).

So, $c2$ holds.

We prove $c1$. Assume $C_t = C'_t = \textbf{stop}$. Because $\tau|_\ell =_{obs} \epsilon$ and $\tau'|_\ell =_{obs} \epsilon$, it suffices to prove $M_t|_\ell = M'_t|_\ell$ and $M_t(cc) = M'_t(cc)$. From Lemma 20, we get $M(cc) = M_t(cc)$ and $M'(cc) = M'_t(cc)$. From $M(cc) = M'(cc)$, we then get $M_t(cc) = M'_t(cc)$. We prove $M_t|_\ell = M'_t|_\ell$. If $M_t(T^{i+1}(x)) \sqsubseteq \ell$, then (ii) gives $x \notin targetFlex(C)$. So, $M_t(T^i(x)) = M(T^i(x))$, $M'(T^i(x)) = M'_t(T^i(x))$, $M_t(T^{i+1}(x)) = M(T^{i+1}(x))$, $M'(T^{i+1}(x)) = M'_t(T^{i+1}(x))$. Thus, $M(T^{i+1}(x)) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$, we then have $M'(T^{i+1}(x)) \sqsubseteq \ell$ and $M(T^i(x)) = M'(T^i(x))$. By transitivity, $M_t(T^i(x)) = M'_t(T^i(x))$ and $M'_t(T^{i+1}(x)) \sqsubseteq \ell$. Assume $M_t(bc) \sqsubseteq \ell$. So, (iii) gives $targetAnchor(C) = \emptyset$. So, $M(bc) = M_t(bc)$ and $M'(bc) = M'_t(bc)$. From $M_t(bc) \sqsubseteq \ell$, we then get $M(bc) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$, we then get $M(bc) = M'(bc)$. Thus, $M_t(bc) = M'_t(bc)$. So, $M_t|_\ell = M'_t|_\ell$. Thus $c1$ holds.

We prove $c3$. If $C'_t = \textbf{block}$, then $targetAnchor(C) \neq \emptyset$. So, (iii) gives $M_t(bc) \not\sqsubseteq \ell$. Thus $c3$ holds.

We prove $c4$. We have that $M'(\lfloor cc \rfloor) \not\sqsubseteq \ell$ or $M'(T(e)) \not\sqsubseteq \ell$.

So, if $C'_t = \textbf{block}$, then $M'_{tp}(\lfloor cc \rfloor) \not\sqsubseteq \ell$. Thus $c4$ holds.

6. $\textbf{while}\ e\ \textbf{do}\ C_1\ \textbf{end}$

Induction on the maximum number of iterations in τ and τ'.

Base case: Both τ and τ' take (Wl2).

$\tau = \langle \textbf{while}\ e\ \textbf{do}\ C_1\ \textbf{end}, \mathcal{M} \rangle \rightarrow \langle \textbf{exit}, M_e \rangle \rightarrow \langle \textbf{stop}, M_t \rangle$

$\tau' = \langle \textbf{while}\ e\ \textbf{do}\ C_1\ \textbf{end}, \mathcal{M}' \rangle \rightarrow \langle \textbf{exit}, M'_e \rangle \rightarrow \langle \textbf{stop}, M'_t \rangle$.

So, $c2$, $c3$, $c4$ are trivially true.

We prove $c1$. We have $\tau|_\ell =_{obs} \epsilon$ and $\tau'|_\ell =_{obs} \epsilon$.

6.1. $M(T(e)) \sqsubseteq \ell$:

We prove $M_e(cc) = M'_e(cc)$. From $M(T(e)) \sqsubseteq \ell$ and $mon(M)$, we then have $M(T^2(e)) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$ and Lemma 17, we then get $M(T(e)) = M'(T(e))$. Both τ and τ' have the same W and A. So, from $M(cc) = M'(cc)$ and $M(T(e)) = M'(T(e))$, we then get $M_e(cc) = M'_e(cc)$.

We now prove $M_e|_\ell = M'_e|_\ell$. If $M_e(bc) \not\sqsubseteq \ell$, then it is trivial, given $M|_\ell = M'|_\ell$. Assume $M_e(bc) \sqsubseteq \ell$. Given $M_e(cc) = M'_e(cc)$, it suffices to prove $M_e(bc) = M'_e(bc)$. We have $M(bc) = M_e(bc)$ and $M'(bc) = M'_e(bc)$. Because $M_e(bc) \sqsubseteq \ell$, we then get $M(bc) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$, we then get $M(bc) = M'(bc)$. Because $M(bc) = M_e(bc)$ and $M'(bc) = M'_e(bc)$, we then get by transitivity $M_e(bc) = M'_e(bc)$.

So, $M_e(cc) = M'_e(cc)$ and $M_e|_\ell = M'_e|_\ell$. From Lemma 14, $mon(M)$, and $mon(M')$, we get $mon(M_e)$ and $mon(M_e)$. We use the proof for **exit** to get $M_t|_\ell = M'_t|_\ell$ and $M_t(cc) = M'_t(cc)$. Thus $c1$ holds.

6.2. $M(T(e)) \not\sqsubseteq \ell$:

In case (*6.1.*), we showed that $M(T(e)) \sqsubseteq \ell$ implies $M(T(e)) = M'(T(e))$, which gives $M'(T(e)) \sqsubseteq \ell$. The contrapositive of this statement is that $M'(T(e)) \not\sqsubseteq \ell$ gives $M(T(e)) \not\sqsubseteq \ell$. Because M, M' are arbitrary and because $M(T(e)) \not\sqsubseteq \ell$ (hypothesis of this case), we then get $M'(T(e)) \not\sqsubseteq \ell$.

We prove $M_t(cc) = M'_t(cc)$. From Lemma 20, we get $M(cc) = M_t(cc)$ and $M'(cc) = M'_t(cc)$. From $M(cc) = M'(cc)$, we then get $M_t(cc) = M'_t(cc)$.

We prove $M_t|_\ell = M'_t|_\ell$. Using Lemma 15, if $M_t(T^{i+1}(x)) \sqsubseteq \ell$, then $x \notin \mathit{targetFlex}(C)$. So, $M_t(T^i(x)) = M(T^i(x))$, $M'(T^i(x)) = M'_t(T^i(x))$, $M_t(T^{i+1}(x)) = M(T^{i+1}(x))$, $M'(T^{i+1}(x)) = M'_t(T^{i+1}(x))$. Thus, $M(T^{i+1}(x)) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$, we then have $M'(T^{i+1}(x)) \sqsubseteq \ell$ and $M(T^i(x)) = M'(T^i(x))$. By transitivity, $M_t(T^i(x)) = M'_t(T^i(x))$ and $M'_t(T^{i+1}(x)) \sqsubseteq \ell$. Assume $M_t(bc) \sqsubseteq \ell$. From Lemma 15, we then get that $\mathit{targetAnchor}(C) = \emptyset$. So, $M(bc) = M_t(bc)$ and $M'(bc) = M'_t(bc)$. From $M_t(bc) \sqsubseteq \ell$ and Lemma 19, we get $M(bc) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$, we then get $M(bc) = M'(bc)$. Thus, by transitivity, we get $M_t(bc) = M'_t(bc)$. And because $M_t(cc) = M'_t(cc)$, we consequently have $M_t|_\ell = M'_t|_\ell$. Thus $c1$ holds.

Induction case:

6.1. $M(\lfloor cc \rfloor) \sqsubseteq \ell$ and $M(T(e)) \sqsubseteq \ell$

From $mon(M)$, we then have $M(T^2(e)) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$ and Lemma 17, we then get $M(T(e)) = M'(T(e))$ and $M(e) = M'(e)$. So, τ and τ' take the same branch. If both take (WL2), then we follow the Base case.

Assume that both take (WL1):

$\tau = \langle \mathbf{while}\ e\ \mathbf{do}\ C_1\ \mathbf{end}, M\rangle \rightarrow \langle C_1; \mathbf{while}\ e\ \mathbf{do}\ C_1\ \mathbf{end}; \mathbf{exit}, M_1\rangle \stackrel{*}{\rightarrow}$

$\langle C_t, M_t\rangle$,

$\tau' = \langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}, M'\rangle \rightarrow \langle C_1; \textbf{while } e \textbf{ do } C_1 \textbf{ end}; \textbf{exit}, M_1'\rangle \stackrel{*}{\rightarrow}$ $\langle C_t', M_t'\rangle$.

We get $M_1(cc) = M_1'(cc)$ from $M(cc) = M'(cc)$ and $M(T(e)) = M'(T(e))$.

We prove $M_1|_\ell = M_1'|_\ell$. Assume $M_1(bc) \sqsubseteq \ell$. Because $M_1(bc) = M(bc)$, we then get $M(bc) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$, we then get $M(bc) = M'(bc)$. Because $M_1'(bc) = M'(bc)$, we then get $M_1(bc) = M_1'(bc)$. So, $M_1|_\ell = M_1'|_\ell$.

We get $mon(M_1)$ and $mon(M_1)$, from Lemma 14, $mon(M)$, and $mon(M')$.

6.1.1. C_1 terminates normally in the 1st iteration in τ and τ'.

$\tau = \langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}, M\rangle \rightarrow \langle C_1; \textbf{while } e \textbf{ do } C_1 \textbf{ end}; \textbf{exit}, M_1\rangle \stackrel{*}{\rightarrow}$ $\langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}; \textbf{exit}, M_2\rangle \stackrel{*}{\rightarrow} \langle C_t, M_t\rangle$,

$\tau' = \langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}, M'\rangle \rightarrow \langle C_1; \textbf{while } e \textbf{ do } C_1 \textbf{ end}; \textbf{exit}, M_1'\rangle \stackrel{*}{\rightarrow}$ $\langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}; \textbf{exit}, M_2'\rangle \stackrel{*}{\rightarrow} \langle C_t', M_t'\rangle$.

Consider:

$\tau_1 = \langle C_1, M_1\rangle \stackrel{*}{\rightarrow} \langle \textbf{stop}, M_2\rangle$

$\tau_1' = \langle C_1, M_1'\rangle \stackrel{*}{\rightarrow} \langle \textbf{stop}, M_2'\rangle$

Because $M_1(cc) = M_1'(cc)$, $M_1|_\ell = M_1'|_\ell$, $mon(M_1)$, and $mon(M_1)$, we can apply IH[$c1$] on C_1. So, we get $\tau_1|_\ell =_{obs} \tau_1'|_\ell$, $M_2|_\ell = M_2'|_\ell$, and $M_2(cc) = M_2'(cc)$. From $mon(M_1)$, $mon(M_1')$ and Lemma 14, we get $mon(M_2)$, $mon(M_2')$.

Consider traces:

$\tau_2 = \langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}, M_2\rangle \stackrel{*}{\rightarrow} \langle C_3, M_3\rangle$

$\tau_2' = \langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}, M_2'\rangle \stackrel{*}{\rightarrow} \langle C_3', M_3'\rangle$

that terminate (normally or blocked). Because $M_2|_\ell = M_2'|_\ell$, $M_2(cc) = M_2'(cc)$, $mon(M_2)$, and $mon(M_2')$, we can apply IH on the max-number of iterations on τ_2 and τ_2'.

We prove $c2$. Say that C_t is **block**. Then C_3 should be **block**. From IH[$c2$] on τ_2 and τ_2', we then get $\tau_2|_\ell =_{obs} \tau_2'|_\ell$. Because we have $\tau|_\ell =_{obs} \tau_1|_\ell \rightharpoonup \tau_2|_\ell$, $\tau'|_\ell =_{obs} \tau_1'|_\ell \rightharpoonup \tau_2'|_\ell$, $\tau_1|_\ell =_{obs} \tau_1'|_\ell$, and $\tau_2|_\ell =_{obs} \tau_2'|_\ell$, we get $\tau|_\ell =_{obs} \tau'|_\ell$. So, $c2$ holds.

We similarly prove $c3$ and $c4$.

We prove $c1$. Assume τ and τ' terminate normally:

$\tau = \langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}, M\rangle \rightarrow \langle C_1; \textbf{while } e \textbf{ do } C_1 \textbf{ end}; \textbf{exit}, M_1\rangle \xrightarrow{*}$
$\langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}; \textbf{exit}, M_2\rangle \xrightarrow{*} \langle \textbf{exit}, M_3\rangle \rightarrow \langle \textbf{stop}, M_t\rangle$,

$\tau' = \langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}, M'\rangle \rightarrow \langle C_1; \textbf{while } e \textbf{ do } C_1 \textbf{ end}; \textbf{exit}, M_1'\rangle \xrightarrow{*}$
$\langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}; \textbf{exit}, M_2'\rangle \xrightarrow{*} \langle \textbf{exit}, M_3'\rangle \rightarrow \langle \textbf{stop}, M_t'\rangle$.

Then τ_2 and τ_2' terminate normally. So, we have

$\tau_2 = \langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}, M_2\rangle \xrightarrow{*} \langle \textbf{stop}, M_3\rangle$

$\tau_2' = \langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}, M_2'\rangle \xrightarrow{*} \langle \textbf{stop}, M_3'\rangle$

By IH[$c1$] on τ_2 and τ_2', we then get $\tau_2|_\ell =_{obs} \tau_2'|_\ell$, $M_3|_\ell = M_3'|_\ell$ and $M_3(cc) = M_3'(cc)$. Because $\tau|_\ell =_{obs} \tau_1|_\ell \rightharpoonup \tau_2|_\ell$, $\tau'|_\ell =_{obs} \tau_1'|_\ell \rightharpoonup \tau_2'|_\ell$, $\tau_1|_\ell =_{obs} \tau_1'|_\ell$, and $\tau_2|_\ell =_{obs} \tau_2'|_\ell$, we get $\tau|_\ell =_{obs} \tau'|_\ell$.

To prove $c1$, we also need to prove that $M_t|_\ell = M_t'|_\ell$ and $M_t(cc) = M_t'(cc)$. Consider:

$\tau_3 = \langle \textbf{exit}, M_3\rangle \rightarrow \langle \textbf{stop}, M_t\rangle$

$\tau_3' = \langle \textbf{exit}, M_3'\rangle \rightarrow \langle \textbf{stop}, M_t'\rangle$.

From $mon(M_2)$, $mon(M_2')$ and Lemma 14, we get $mon(M_3)$, $mon(M_3')$. Because $M_3|_\ell = M_3'|_\ell$ and $M_3(cc) = M_3'(cc)$, $mon(M_3)$, and $mon(M_3')$, we can use the proof for **exit** (case 7.) to get $M_t|_\ell = M_t'|_\ell$ and $M_t(cc) = M_t'(cc)$. So, $c1$ holds.

6.1.2. C_1 blocked in both τ and τ' during 1st iteration.

We use IH on C_1.

6.1.3. C_1 blocked in τ, terminates normally in τ'.

$\tau = \langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}, M\rangle \to \langle C_1; \textbf{while } e \textbf{ do } C_1 \textbf{ end}; \textbf{exit}, M_1\rangle \xrightarrow{*}$
$\langle C_{1t}; \textbf{while } e \textbf{ do } C_1 \textbf{ end}; \textbf{exit}, M_t\rangle$,
$\tau' = \langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}, M'\rangle \to \langle C_1; \textbf{while } e \textbf{ do } C_1 \textbf{ end}; \textbf{exit}, M'_1\rangle \xrightarrow{*}$
$\langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}; \textbf{exit}, M'_2\rangle \xrightarrow{*} \langle C'_t, M'_t\rangle$.

Consider:

$\tau_1 = \langle C_1, M_1\rangle \xrightarrow{*} \langle C_{1t}, M_t\rangle$
$\tau'_1 = \langle C_1, M'_1\rangle \xrightarrow{*} \langle \textbf{stop}, M'_2\rangle$
$\tau'_2 = \langle \textbf{while } e \textbf{ do } C_1 \textbf{ end}; \textbf{exit}, M'_2\rangle \xrightarrow{*} \langle C'_t, M'_t\rangle$

IH can be applied to τ_1 and τ'_1, because $M_1|_\ell = M'_1|_\ell$, $M_1(cc) = M'_1(cc)$, $mon(M'_1)$, and C_1 is a subcommand of **while** e **do** C_1 **end**.

We invoke (*4.3.*) for τ_1, τ'_1, and τ'_2.

6.2. $M(\lfloor cc \rfloor) \not\sqsubseteq \ell$ or $M(T(e)) \not\sqsubseteq \ell$

Consider:

$\tau_w = \langle \textbf{if } e \textbf{ then } (C_1; \textbf{while } e \textbf{ do } C_1 \textbf{ end}) \textbf{ else skip end}, M\rangle \xrightarrow{*}$
$\langle C_{wt}, M_{wt}\rangle$, and
$\tau_w = \langle \textbf{if } e \textbf{ then } (C_1; \textbf{while } e \textbf{ do } C_1 \textbf{ end}) \textbf{ else skip end}, M'\rangle \xrightarrow{*}$
$\langle C'_{wt}, M'_{wt}\rangle$.

We invoke case *5.2.* for τ_w, τ'_w, and we get:

c1w If C_{wt} and C'_{wt} are both **stop**, then $\tau_w|_\ell =_{obs} \tau'_w|_\ell$, $M_{wt}|_\ell = M'_{wt}|_\ell$, and $M_{wt}(cc) = M'_{wt}(cc)$.

c2w If C_{wt} or C'_{wt} is **block**, then $\tau_w|_\ell =_{obs} \tau'_w|_\ell$.

c3w If C_{wt} is **stop**, C'_{wt} is **block**, and $M'_{wt}(bc) \not\sqsubseteq \ell$, then $M_{wt}(bc) \not\sqsubseteq \ell$.

We prove $c1$. Assume C_t and C'_t are both **stop**. Because $C_t = C_{wt}$ and $C'_t = C'_{wt}$, we have that C_{wt} and C'_{wt} are both **stop**. From **c1w**, we have $\tau_w|_\ell = \tau'_w|_\ell$, $M_{wt}|_\ell = M'_{wt}|_\ell$, and $M_{wt}(cc) = M'_{wt}(cc)$. We have $M_t = M_{wt}$,

$M'_t = M'_{wt}$, $\tau|_\ell =_{obs} \tau_w|_\ell$, and $\tau'|_\ell =_{obs} \tau'_w|_\ell$. So, $\tau|_\ell =_{obs} \tau'|_\ell$, $M_t|_\ell = M'_t|_\ell$, and $M_t(cc) = M'_t(cc)$. Thus $c1$ holds.

Similarly, we get $c2$ and $c3$.

$c4$ is trivially true: from $M(\lfloor cc \rfloor) \not\sqsubseteq \ell$ or $M(T(e)) \not\sqsubseteq \ell$ we get $M'_{tp}(\lfloor cc \rfloor) \not\sqsubseteq \ell$.

7. **exit**

We have:

$\tau = \langle \mathbf{exit}, M \rangle \rightarrow \langle \mathbf{stop}, M_t \rangle$

$\tau' = \langle \mathbf{exit}, M' \rangle \rightarrow \langle \mathbf{stop}, M'_t \rangle$.

$c2$, $c3$, $c4$ are trivially true.

We prove $c1$. We have $\tau|_\ell =_{obs} \epsilon$ and $\tau'|_\ell =_{obs} \epsilon$. So, we need to prove $M_t|_\ell = M_t|_\ell$ and $M_t(cc) = M'_t(cc)$. Because $M(cc) = M'(cc)$, $M_t(cc) = M(cc).pop$, and $M'_t(cc) = M'(cc).pop$, we then get $M_t(cc) = M'_t(cc)$. We now prove $M_t|_\ell = M_t|_\ell$.

7.1. $M_t(\lfloor cc \rfloor) \sqcup M_t(bc) \not\sqsubseteq \ell$ and $M(cc).top.A \neq \emptyset$.

We first prove that $M_t(bc) \not\sqsubseteq \ell$ and $M'_t(bc) \not\sqsubseteq \ell$. Because $M(cc).top.A \neq \emptyset$, we have $M_t(bc) = M(bc) \sqcup M(\lfloor cc \rfloor)$. Because $M_t(\lfloor cc \rfloor) \sqsubseteq M(\lfloor cc \rfloor)$, we get $M_t(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq M(\lfloor cc \rfloor) \sqcup M(bc)$, which becomes $M_t(\lfloor cc \rfloor) \sqcup M(bc) \sqcup M(\lfloor cc \rfloor) \sqsubseteq M(\lfloor cc \rfloor) \sqcup M(bc) \sqcup M(\lfloor cc \rfloor)$, which becomes $M_t(\lfloor cc \rfloor) \sqcup M_t(bc) \sqsubseteq M(\lfloor cc \rfloor) \sqcup M(bc)$, due to $M_t(bc) = M(bc) \sqcup M(\lfloor cc \rfloor)$. From $M_t(\lfloor cc \rfloor) \sqcup M_t(bc) \not\sqsubseteq \ell$ we get $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$. From $M_t(bc) = M(bc) \sqcup M(\lfloor cc \rfloor)$, we then have $M_t(bc) \not\sqsubseteq \ell$. From $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$ and $M|_\ell = M'|_\ell$, we get $M'(\lfloor cc \rfloor) \sqcup M'(bc) \not\sqsubseteq \ell$. Because $M(cc).top.A \neq \emptyset$ and $M(cc) = M'(cc)$, we have $M'(cc).top.A \neq \emptyset$, too. So, $M'_t(bc) = M'(bc) \sqcup M'(\lfloor cc \rfloor)$. From $M'(\lfloor cc \rfloor) \sqcup M'(bc) \not\sqsubseteq \ell$, we then have $M'_t(bc) \not\sqsubseteq \ell$. So, $M_t(bc) \not\sqsubseteq \ell$ and $M'_t(bc) \not\sqsubseteq \ell$.

Only variables in W change their labels. Let $x \in M(cc).top.W$. Be-

cause $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$, we have $\forall i \geq 1. M_t(T^i(x)) \not\sqsubseteq \ell$. Because $M(cc) = M'(cc)$, we get $x \in M'(cc).top.W$, too. From $M'_t(bc) \not\sqsubseteq \ell$, we have $M'(\lfloor cc \rfloor) \sqcup M'(bc) \not\sqsubseteq \ell$, and thus, $\forall i \geq 1. M'_t(T^i(x)) \not\sqsubseteq \ell$. So, $M_t|_\ell = M'_t|_\ell$. Thus $c1$ holds.

7.2. $M_t(\lfloor cc \rfloor) \sqcup M_t(bc) \not\sqsubseteq \ell$ and $M(cc).top.A = \emptyset$.

Because $M(cc).top.A = \emptyset$, we have $M_t(bc) = M(bc)$. We have $M_t(\lfloor cc \rfloor) \sqsubseteq M(\lfloor cc \rfloor)$. We get $M_t(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq M(\lfloor cc \rfloor) \sqcup M(bc)$, which becomes $M_t(\lfloor cc \rfloor) \sqcup M_t(bc) \sqsubseteq M(\lfloor cc \rfloor) \sqcup M(bc)$, due to $M_t(bc) = M(bc)$. From $M_t(\lfloor cc \rfloor) \sqcup M_t(bc) \not\sqsubseteq \ell$, we then have $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$. Because $M|_\ell = M'|_\ell$, we also get $M'(\lfloor cc \rfloor) \sqcup M'(bc) \not\sqsubseteq \ell$.

We prove that if $M_t(bc) \sqsubseteq \ell$, then $M_t(bc) = M'_t(bc)$. Assume $M_t(bc) \sqsubseteq \ell$. From $M_t(bc) = M(bc)$, we then get $M(bc) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$, we then get $M(bc) = M'(bc)$. Because $M(cc).top.A = \emptyset$ and $M(cc) = M'(cc)$, we have that $M'(cc).top.A = \emptyset$. So, $M'_t(bc) = M'(bc)$. By transitivity, we then get $M_t(bc) = M'_t(bc)$.

From $M_t(cc) = M'_t(cc)$, $M_t(bc) = M'_t(bc)$, and $M_t(\lfloor cc \rfloor) \sqcup M_t(bc) \not\sqsubseteq \ell$, we then get $M'_t(\lfloor cc \rfloor) \sqcup M'_t(bc) \not\sqsubseteq \ell$.

Only variables in W change their labels. Let $x \in M(cc).top.W$. Because $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$, we have $\forall i \geq 1. M_t(T^i(x)) \not\sqsubseteq \ell$. Because $M(cc) = M'(cc)$, we get $x \in M'(cc).top.W$, too. Because $M'(\lfloor cc \rfloor) \sqcup M'(bc) \not\sqsubseteq \ell$, we have $\forall i \geq 1. M'_t(T^i(x)) \not\sqsubseteq \ell$. So, $M_t|_\ell = M'_t|_\ell$. Thus $c1$ holds.

7.3. $M_t(\lfloor cc \rfloor) \sqcup M_t(bc) \sqsubseteq \ell$

So, $M_t(bc) \sqsubseteq \ell$. From Lemma 19, we get $M(bc) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$ and $M(bc) \sqsubseteq \ell$, we also get $M(bc) = M'(bc)$. From $M(cc) = M'(cc)$, we then get $M_t(bc) = M'_t(bc)$.

- Let $M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq \ell$. Let $x \in M(cc).top.W$. Because $M(cc) =$

$M'(cc)$, we get $x \in M'(cc).top.W$. We have:

$M_t(T^i(x)) \sqsubseteq \ell \Rightarrow M(T^i(x)) \sqsubseteq \ell \Rightarrow M(T^{i-1}(x)) = M'(T^{i-1}(x))$ and $M'(T^i(x)) \sqsubseteq \ell$.

Because $M(cc) = M'(cc)$ and $M(bc) = M'(bc)$, we then have $M_t(T^{i-1}(x)) = M'_t(T^{i-1}(x))$ and $M'_t(T^i(x)) \sqsubseteq \ell$. Thus, $M_t|_\ell = M'_t|_\ell$. Thus $c1$ holds.

- Let $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$.

  So, $M'(\lfloor cc \rfloor) \sqcup M'(bc) \not\sqsubseteq \ell$. Let $x \in M(cc).top.W$. Consequently, we have that $\forall i \geq 1$: $M_t(T^i(x)) \not\sqsubseteq \ell$. Because $M(cc) = M'(cc)$, we get $x \in M'(cc).top.W$, and thus $\forall i \geq 1$: $M'_t(T^i(x)) \not\sqsubseteq \ell$. Thus, $M_t|_\ell = M'_t|_\ell$. Thus $c1$ holds.

□

Lemma 14. *Let $\langle C, M\rangle \xrightarrow{*} \langle C', M'\rangle$ be a trace generated by ∞-Enf.*
If $mon(M)$, then $mon(M')$.

Proof. We first prove the statement for one-step transition: $\langle C, M\rangle \rightarrow \langle C', M'\rangle$, and then we use induction on the number of steps in $\langle C, M\rangle \xrightarrow{*} \langle C', M'\rangle$. We prove the statement for one-step transition, using induction on the rules of ∞-*Enf*'s operational semantics. Assume $mon(M)$. We prove $mon(M')$.

1. (SKIP):

   Because $M = M'$, we then get $mon(M')$.

2. (ASGNA):

   Trivially true, because no label chain is being updated, and thus, $mon(M')$.

3. (ASGNAFAIL):

   Same arguments as in above case.

4. (ASGNF):

   From $mon(M')$, we have that for every x we have

   $\forall i \geq 1$: $M(T^{i+1}(x)) \sqsubseteq M(T^i(x))$,

   So, we get $\forall i \geq 1$: $M(T^{i+1}(e)) \sqsubseteq M(T^i(e))$. We then have

   $\forall i \geq 1$: $M(T^{i+1}(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq M(T^i(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)$,

   and thus $\forall i \geq 1$: $M'(T^{i+1}(w)) \sqsubseteq M'(T^i(w))$. So, $mon(M')$.

5. (IF1),(IF2),(WL1),(WL2):

   Trivially true, because no label chain is being updated, and thus, $mon(M')$.

6. (EXIT):

   Only label chains of $w \in V$ change. From $mon(M)$, we have for $i \geq 1$, $M(T^{i+1}(w)) \sqsubseteq M(T^i(w))$. Thus, $M(T^{i+1}(w)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq M(T^i(w)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)$. So, $M'(T^{i+1}(w)) \sqsubseteq M'(T^i(w))$. So, $mon(M')$.

7. (SEQ1),(SEQ2),(SEQF):

   We use the IH.

□

Lemma 15. *If C does not include* **exit** *(or i-***exit***), if $\tau = \langle C, M \rangle \xrightarrow{*} \langle C', M' \rangle$ is generated by ∞-Enf and $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$, or C is a conditional command (executed under* IF, IF′, *or* WL *rule—not* IFS *rules) with guard e and $M(T(e)) \not\sqsubseteq \ell$, then*

- *(i)* $\tau|_\ell =_{obs} \epsilon$.
- *(ii) if $C' =$* **stop** *and $w \in targetFlex(C)$, then $M'(T^i(w)) \not\sqsubseteq \ell$, for all $T^i(w) \in dom(M')$ where $i \geq 1$.*
- *(iii) if $C' =$* **stop** *and $targetAnchor(C) \neq \emptyset$, then $M'(bc) \not\sqsubseteq \ell$.*

Proof. Induction on C (which should not include **exit**).

1. $a := e$

   Assume $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$.

   If $C' = \textbf{block}$, then $\tau|_\ell = \epsilon$, so (i) holds, and (ii), (iii) are trivially true.

   Assume $C' = \textbf{stop}$. So, $M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq M(T(a))$. Because $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$, we then get $M(T(a)) \not\sqsubseteq \ell$. Thus $\tau|_\ell =_{obs} \epsilon$ and (i) holds.

   (ii) is trivially true.

   We have $M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq M'(bc)$. Because $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$, we then get $M'(bc) \not\sqsubseteq \ell$. Thus (iii) holds.

2. $w := e$

   Assume $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$.

   (iii) is trivially true.

   Because $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$ and $\forall i \geq 1$: $M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq M'(T^i(w))$, we get $\forall i \geq 1$: $M'(T^i(w)) \not\sqsubseteq \ell$. Thus (ii) holds.

   Also, $\tau|_\ell =_{obs} \epsilon$. Thus (i) holds.

3. $C_1; C_2$

   Assume $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$.

   - Assume τ involves only the execution of C_1.

     So, τ is blocked, and thus, (ii), (iii) are trivially true.

     We prove (i). Because τ involves only the blocked execution of C_1, consider $\tau_1 = \langle C_1, M \rangle \xrightarrow{*} \langle \textbf{block}, M' \rangle$. From IH on C_1, we get $\tau_1|_\ell =_{obs} \epsilon$. Because $\tau|_\ell = \tau_1|_\ell$, we then get $\tau|_\ell =_{obs} \epsilon$. Thus (i) holds.

   - Assume τ involves execution of C_1 and C_2.

     Then C_1 is executed to normal termination. Consider:

     $\tau_1 = \langle C_1, M \rangle \xrightarrow{*} \langle \textbf{stop}, M_1 \rangle$.

Also, C_2 might be executed to termination or blocked. Consider:

$\tau_2 = \langle C_2, M_1\rangle \xrightarrow{*} \langle C', M'\rangle$.

From Lemma 20, we get $M(cc) = M_1(cc)$. From Lemma 19, we get $M(bc) \sqsubseteq M_1(bc)$. So, $M(bc) \sqcup M(cc) \sqsubseteq M_1(bc) \sqcup M_1(\lfloor cc \rfloor)$. From $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$, we then have $M_1(bc) \sqcup M_1(\lfloor cc \rfloor) \not\sqsubseteq \ell$.

We prove (i). From IH on C_1, we get $\tau_1|_\ell =_{obs} \epsilon$. From IH on C_2, we get $\tau_2|_\ell =_{obs} \epsilon$. Because $\tau|_\ell =_{obs} \tau_1|_\ell \rightharpoonup \tau_2|_\ell$, we get $\tau|_\ell =_{obs} \epsilon$. Thus (i) holds.

We prove (ii). Assume $C' = \mathbf{stop}$ and $w \in \mathit{targetFlex}(C_1; C_2)$. Then $w \in \mathit{targetFlex}(C_1)$ or $w \in \mathit{targetFlex}(C_2)$. Assume $w \in \mathit{targetFlex}(C_2)$. From IH on C_2, we get $\forall i \geq 1$: $M'(T^i(w)) \not\sqsubseteq \ell$. Thus (ii) holds. Assume $w \notin \mathit{targetFlex}(C_2)$. From $w \in \mathit{targetFlex}(C_1; C_2)$, we get $w \in \mathit{targetFlex}(C_1)$. From IH on C_1, we get $\forall i \geq 1$: $M_1(T^i(w)) \not\sqsubseteq \ell$. Because $w \notin \mathit{targetFlex}(C_2)$, we have $\forall i \geq 1$: $M_1(T^i(w)) = M'(T^i(w))$. So, $\forall i \geq 1$: $M'(T^i(w)) \not\sqsubseteq \ell$. Thus (ii) holds.

We prove (iii). Assume $C' = \mathbf{stop}$ and $\mathit{targetAnchor}(C_1; C_2) \neq \emptyset$. Then $\mathit{targetAnchor}(C_1) \neq \emptyset$ or $\mathit{targetAnchor}(C_2) \neq \emptyset$. Assume that $\mathit{targetAnchor}(C_2) \neq \emptyset$. From IH on C_2, we get $M'(bc) \not\sqsubseteq \ell$. Thus (iii) holds. Assume $\mathit{targetAnchor}(C_2) = \emptyset$. From $\mathit{targetAnchor}(C_1; C_2) \neq \emptyset$, we then have $\mathit{targetAnchor}(C_1) \neq \emptyset$. From IH on C_1, we get $M_1(bc) \not\sqsubseteq \ell$. From Lemma 19, we have $M_1(bc) \sqsubseteq M'(bc)$. So, we have $M'(bc) \not\sqsubseteq \ell$. Thus (iii) holds.

4. $\mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{end}$

   Assume $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$ or $M(T(e)) \not\sqsubseteq \ell$. W.l.o.g. assume that τ involves the execution of C_1.

   So:

   $\tau = \langle \mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{end}, M\rangle \rightarrow \langle C_1; \mathbf{exit}, M_1\rangle \xrightarrow{*} \langle C', M'\rangle$.

Consider:

$\tau_1 = \langle C_1, M_1\rangle \xrightarrow{*} \langle C_1', M_1'\rangle$.

We have $M(bc) = M_1(bc)$ and $M(\lfloor cc \rfloor) \sqsubseteq M_1(\lfloor cc \rfloor)$. So, $M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq M_1(\lfloor cc \rfloor) \sqcup M_1(bc)$. If $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$, we then have $M_1(bc) \sqcup M_1(\lfloor cc \rfloor) \not\sqsubseteq \ell$. If $M(T(e)) \not\sqsubseteq \ell$, we then have $M_1(bc) \sqcup M_1(\lfloor cc \rfloor) \not\sqsubseteq \ell$, because $M(T(e)) \sqsubseteq M_1(\lfloor cc \rfloor)$. So, in any case, $M_1(bc) \sqcup M_1(\lfloor cc \rfloor) \not\sqsubseteq \ell$. So, we can apply IH on C_1.

We prove (i). From IH on C_1, we get $\tau_1|_\ell =_{obs} \epsilon$. Because $\tau|_\ell = \tau_1|_\ell$, we have $\tau|_\ell =_{obs} \epsilon$. Thus (i) holds.

We prove (ii). Assume $C' = \mathbf{stop}$ and $w \in \mathit{targetFlex}(C)$. Then $C_1' = \mathbf{stop}$ and $w \in \mathit{targetFlex}(C_1)$ or $w \in \mathit{targetFlex}(C_2)$. We have:

$\tau = \langle \mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{end}, M\rangle \to \langle C_1; \mathbf{exit}, M_1\rangle \xrightarrow{*} \langle \mathbf{exit}, M_1'\rangle \to \langle C', M'\rangle$

and

$\tau_1 = \langle C_1, M_1\rangle \xrightarrow{*} \langle \mathbf{stop}, M_1'\rangle$.

Assume $w \in \mathit{targetFlex}(C_1)$. From IH on C_1, we get $\forall i \geq 1$: $M_1'(T^i(w)) \not\sqsubseteq \ell$. Due to the rule for **exit**, we get $\forall i \geq 1$: $M_1'(T^i(w)) \sqsubseteq M'(T^i(w))$. So, $\forall i \geq 1$: $M'(T^i(w)) \not\sqsubseteq \ell$. Thus (ii) holds.

Assume $w \notin \mathit{targetFlex}(C_1)$. From $w \in \mathit{targetFlex}(C)$, we then have $w \in \mathit{targetFlex}(C_2)$. So, $w \in M_1(cc).\mathit{top}.W$. From Lemma 20, we then have $w \in M_1'(cc).\mathit{top}.W$. Due to the rule for **exit**, we get $\forall i \geq 1$: $M_1'(\lfloor cc \rfloor) \sqcup M_1'(bc) \sqsubseteq M'(T^i(w))$. We have $M_1(\lfloor cc \rfloor) \sqcup M_1(bc) \sqsubseteq M_1'(\lfloor cc \rfloor) \sqcup M_1'(bc)$. So, $M_1'(\lfloor cc \rfloor) \sqcup M_1'(bc) \not\sqsubseteq \ell$. Thus, $\forall i \geq 1$: $M'(T^i(w)) \not\sqsubseteq \ell$. Thus (ii) holds.

We prove (iii). Assume $C' = \mathbf{stop}$ and $\mathit{targetAnchor}(C) \neq \emptyset$ Then $C_1' = \mathbf{stop}$ and $\mathit{targetAnchor}(C_1) \neq \emptyset$ or $\mathit{targetAnchor}(C_2) \neq \emptyset$.

Assume $\mathit{targetAnchor}(C_1) \neq \emptyset$. From IH on C_1, we get $M_1'(bc) \not\sqsubseteq \ell$. From Lemma 19, we get $M_1'(bc) \sqsubseteq M'(bc)$. Thus, we have $M'(bc) \not\sqsubseteq \ell$. Thus (iii)

holds.

Assume $targetAnchor(C_1) = \emptyset$. From $targetAnchor(C) \neq \emptyset$, we then have $targetAnchor(C_2) \neq \emptyset$. So, $M_1(cc).top.A \neq \emptyset$. From Lemma 20, we then have $M_1'(cc).top.A \neq \emptyset$. Due to the rule for **exit**, we get $M'(bc) = M_1'(\lfloor cc \rfloor) \sqcup M_1'(bc)$. We have $M_1(\lfloor cc \rfloor) \sqcup M_1(bc) \sqsubseteq M_1'(\lfloor cc \rfloor) \sqcup M_1'(bc)$. So, $M_1'(\lfloor cc \rfloor) \sqcup M_1'(bc) \not\sqsubseteq \ell$. Thus, $M'(bc) \not\sqsubseteq \ell$. Thus (iii) holds.

5. **while** e **do** C_1 **end**

   We use induction on the number of iterations executed in τ, and IH on C_1, similar to the above cases.

□

Lemma 16. *If $\tau = \langle C, M \rangle \xrightarrow{*} \langle C', M' \rangle$ is generated by ∞-Enf, and $M(bc) \not\sqsubseteq \ell$, then $\tau|_\ell = \epsilon$.*

Proof. By structural induction on C and Lemma 19. □

Lemma 17. *If $M|_\ell = M'|_\ell$ and $M(T^i(e)) \sqsubseteq \ell$, for $i \geq 1$, then $M(T^{i-1}(e)) = M'(T^{i-1}(e))$.*

Proof. We use structural induction on e.

1. e is n:

   By definition $M(n) = M'(n) = n$. By definition $M(T^i(n)) = M'(T^i(n)) = \bot$, for $i \geq 1$.

2. e is x:

   From $M|_\ell = M'|_\ell$ and $M(T^i(x)) \sqsubseteq \ell$, we have $M(T^{i-1}(x)) = M'(T^{i-1}(x))$.

3. e is $e_1 \oplus e_2$:

   We have $M(T^i(e)) = M(T^i(e_1)) \sqcup M(T^i(e_2))$. From $M(T^i(e)) \sqsubseteq \ell$, we then get $M(T^i(e_1)) \sqsubseteq \ell$ and $M(T^i(e_2)) \sqsubseteq \ell$. By IH on e_1 and e_2, we get $M(T^{i-1}(e_1)) = M'(T^{i-1}(e_1))$ and $M(T^{i-1}(e_2)) = M'(T^{i-1}(e_2))$. We have $M(e) = M(e_1) \oplus M(e_2) = M'(e_1) \oplus M'(e_2) = M'(e)$. For $i \geq 1$, we have $M(T^i(e)) = M(T^i(e_1)) \sqcup M(T^i(e_2)) = M'(T^i(e_1)) \sqcup M'(T^i(e_2)) = M'(T^i(e))$.

□

Lemma 18. *If $M|_\ell = M'|_\ell$, $mon(M)$, $mon(M')$ and $M(T^i(e)) \not\sqsubseteq \ell$, then $M'(T^i(e)) \not\sqsubseteq \ell$.*

Proof. We prove it by contradiction. Assume $M'(T^i(e)) \sqsubseteq \ell$. From $mon(M)$, we then have $M'(T^{i+1}(e)) \sqsubseteq \ell$. From Lemma 17, we then get $M(T^i(e)) = M'(T^i(e))$. So, $M(T^i(e)) \sqsubseteq \ell$, which is a contradiction. □

Lemma 19. *Let $\langle C, M\rangle \xrightarrow{*} \langle C', M'\rangle$ be a trace generated by ∞-Enf.*
Then $M(bc) \sqsubseteq M'(bc)$.

Proof. We first prove that: if $\langle C, M\rangle \rightarrow \langle C', M'\rangle$ is generated by ∞-*Enf*, then $M(bc) \sqsubseteq M'(bc)$. To prove this, we use induction on the rules for ∞-*Enf*'s operational semantics. We then use induction on the number of steps in $\langle C, M\rangle \rightarrow \langle C', M'\rangle$. □

Lemma 20. *Let $\langle C, M\rangle \xrightarrow{*} \langle \mathbf{stop}, M_t\rangle$ be a trace generated by ∞-Enf and let C have no* **exit** *(or i-***exit***), then $M(cc) = M_t(cc)$.*

Proof. We use structural induction on C. □

Lemma 21. *Let* $\langle C, M\rangle \xrightarrow{*} \langle C', M'\rangle$ *be a trace generated by* ∞*-Enf.*
If $2stut(M)$*, then* $2stut(M')$*.*

Proof. We first prove the statement for one-step transition:
$\langle C, M\rangle \rightarrow \langle C', M'\rangle$, and then we use induction on the number of steps in $\langle C, M\rangle \xrightarrow{*} \langle C', M'\rangle$. We prove the statement for one-step transition, using induction on the rules of ∞-*Enf*'s operational semantics. Assume, for all x, we have $M(T^2(x)) \sqsubseteq M(T(x))$ and $\forall i > 1$: $M(T^2(x)) = M(T^i(x))$.

1. (SKIP):

   Because $M = M'$, we then get $M'(T^2(x)) \sqsubseteq M'(T(x))$ and
   $\forall i > 1$: $M'(T^2(x)) = M'(T^i(x))$.

2. (ASGNA):

   Trivially true, because no label chain is being updated, and thus, we have
   $\forall i \geq 1$: $M(T^i(x)) = M'(T^i(x))$.

3. (ASGNAFAIL):

   Same arguments as in above case.

4. (ASGNF):

   Because for every x we have $M(T^2(x)) \sqsubseteq M(T(x))$, we get $M(T^2(e)) \sqsubseteq M(T(e))$. We then have
   $M(T^2(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq M(T(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)$,
   and thus $M'(T^2(w)) \sqsubseteq M'(T(w))$.
   Similarly, we get $\forall i > 1$: $M'(T^2(w)) = M'(T^i(w))$.

5. (IF1),(IF2),(WL1),(WL2):

   Trivially true, because no label chain is being updated, and thus,
   $\forall i \geq 1$: $M(T^i(x)) = M'(T^i(x))$.

6. (EXIT):

   Only label chains of $w \in W$ change. We have $M'(T^2(w)) = M(T^2(w)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)$ and $M'(T(w) = M(T(w)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)$. Because $M(T^2(w)) \sqsubseteq M(T(w))$, we then get $M'(T^2(w)) \sqsubseteq M'(T(w))$. Similarly, we get $\forall i > 1$: $M'(T^2(w)) = M'(T^i(w))$.

7. (SEQ1),(SEQ2),(SEQF):

   We use IH.

□

Theorem 4. *k-Enf is an enforcer on R for k-BNI($\mathcal{L}$), for any $\mathcal{L}$ and $k \geq 2$.*

Proof. It is easy to prove that k-*Enf* is an enforcer on R and satisfies restrictions (E1), (E2), and (E3) by induction on the rules for k-*Enf*. We omit the details.

We now prove k-BNI(k-*Enf*, $\mathcal{L}, C$), for a command C, a lattice $\mathcal{L}$, and $k \geq 2$. Consider $\ell \in \mathcal{L}$. Take M, M' with $M \models \mathcal{H}_0(k\text{-}Enf, \mathcal{L}, C)$, $M' \models \mathcal{H}_0(k\text{-}Enf, \mathcal{L}, C)$, $M|_\ell = M'|_\ell$, and finite traces $\tau = trace_{k\text{-}Enf}(C, M)$ and $\tau' = trace_{k\text{-}Enf}(C, M')$, where $\tau = \langle C, M\rangle \xrightarrow{*} \langle C_t, M_t\rangle$, and $\tau' = \langle C, M'\rangle \xrightarrow{*} \langle C'_t, M'_t\rangle$.

We prove $\tau|^k_\ell =_{obs} \tau'|^k_\ell$ or equivalently $\tau|_\ell =_{obs} \tau'|_\ell$, because k-*Enf* generates observation up to kth tag. From $M \models \mathcal{H}_0(k\text{-}Enf, \mathcal{L}, C)$ and $M' \models \mathcal{H}_0(k\text{-}Enf, \mathcal{L}, C)$, we get $M(cc) = M'(cc)$, $mon(M)$, and $mon(M')$. There exists M_I such that $M_I \models \mathcal{H}_0(\infty\text{-}Enf, \mathcal{L}, C)$, $M_I =_k M$, and $kstut(M_I)$. Similarly, there exists M'_I such that $M'_I \models \mathcal{H}_0(\infty\text{-}Enf, \mathcal{L}, C)$, $M'_I =_k M'$, and $kstut(M'_I)$.

We prove $M_I|_\ell = M'_I|_\ell$. Due to $M|_\ell = M'|_\ell$, $M_I =_k M$, and $M'_I =_k M'$, it suffices to examine $\forall x$: $\forall i > k$: $T^i(x)$. Assume $M_I(T^i(x)) \sqsubseteq \ell$. From $kstut(M_I)$, we then get $M_I(T^k(x)) \sqsubseteq \ell$. From $M_I =_k M$, we then have $M(T^k(x)) \sqsubseteq \ell$. By

definition of k-*Enf*, we have $T^{k+1}(x) = T^k(x)$, and thus $M(T^{k+1}(x)) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$, we then get $M(T^k(x)) = M'(T^k(x))$. From $M_I =_k M$, $kstut(M_I)$, $M'_I =_k M'$, and $kstut(M'_I)$, we have $M_I(T^{i-1}(x)) = M'_I(T^{i-1}(x))$ and $M_I(T^i(x)) = M'_I(T^i(x))$. So, $M_I|_\ell = M'_I|_\ell$.

We have $M_I(cc) = M'_I(cc)$, due to $M(cc) = M'(cc)$, $M_I =_k M$, and $M'_I =_k M'$. We have $mon(M_I)$, due to $mon(M)$, $M_I =_k M$, and $kstut(M_I)$. Similarly, we have $mon(M'_I)$.

Consider

$\tau_I = trace_{\infty\text{-}Enf}(C, M_I) = \langle C, M_I\rangle \xrightarrow{*} \langle C_t, M_{It}\rangle$, and

$\tau'_I = trace_{\infty\text{-}Enf}(C, M'_I) = \langle C, M'_I\rangle \xrightarrow{*} \langle C'_t, M'_{It}\rangle$.

From Lemma 22, and applying induction on the number of steps in τ and τ', we get $\tau_I|^k_\ell = \tau|_\ell$ and $\tau'_I|^k_\ell = \tau'|_\ell$. Because ∞-*Enf* satisfies BNI+ (Lemma 13), we get $\tau_I|_\ell =_{obs} \tau'_I|_\ell$. We then get $\tau_I|^k_\ell =_{obs} \tau'_I|^k_\ell$. So, $\tau|_\ell =_{obs} \tau'|_\ell$. □

Lemma 22. *Consider $\tau = \langle C, M\rangle \to \langle C_n, M_n\rangle$ be a step under k-Enf with $k \geq 2$. Let $M' =_k M$, $kstut(M')$, and $\tau' = \langle C, M'\rangle \to \langle C'_n, M'_n\rangle$ be a step under ∞-Enf. Then, $M'_n =_k M_n$, $kstut(M'_n)$, $C_n = C'_n$, and $\tau|_\ell = \tau'|^k_\ell$.*

Proof. Structural induction on C.

1. C is **skip**:

   We have $M = M_n$ and $M' = M'_n$. So, $M'_n =_k M_n$ and $kstut(M'_n)$. Also, $C_n = C'_n =$ **stop** and $\tau|_\ell = \tau'|^k_\ell = \epsilon$.

2. C is $a := e$:

   $G_{a:=e}$ is $T(e) \sqcup M(\lfloor cc\rfloor) \sqcup bc \sqsubseteq T(a)$. Because $M' =_k M$ and $k \geq 2$, we have $M(e) = M'(e)$, $M(T(e)) = M'(T(e))$, $M(T^2(e)) = M'(T^2(e))$, $M(T(a)) =$

$M'(T(a))$, $M(cc) = M'(cc)$, $M(bc) = M'(bc)$. So, τ and τ' both are generated from ASGNA, or both from ASGNAFAIL. So, $C_n = C'_n$. Also, $M_n(bc) = M'_n(bc)$. Thus, $M'_n =_k M_n$. Because no label chain changed, $kstut(M')$ gives $kstut(M'_n)$. Because $M(e) = M'(e)$, we also get $\tau|_\ell = \tau'|^k_\ell$.

3. C is $w := e$:

   We have $C_n = C'_n = \textbf{stop}$. We have $\forall 0 \leq i \leq k$:
   $M'_n(T^i(w)) = M'(T^i(e)) \sqcup M'(cc) \sqcup M'(bc) =$
   $M(T^i(e)) \sqcup M(cc) \sqcup M(bc) = M_n(T^i(w))$.
   Also, $\forall i > k$, we have $M'_n(T^i(w)) = M'_n(T^k(w))$. So, $M'_n =_k M_n$ and $kstut(M'_n)$.
   We have $\tau|_\ell = \tau'|^k_\ell$, because $\forall 0 \leq i \leq k+1$ we have $M_n(T^i(w)) = M'_n(T^i(w))$.

4. C is **if** e **then** C_1 **else** C_2 **end**:

   From $M' =_k M$, we get $M(e) = M'(e)$, $M(T(e)) = M'(T(e))$, $M(cc) = M'(cc)$, $M(bc) = M'(bc)$. Both τ and τ' take the same branch. Say C_1:
   $\tau = \langle C, M\rangle \to \langle C_1; \textbf{exit}, M_1\rangle$
   $\tau' = \langle C, M'\rangle \to \langle C_1; \textbf{exit}, M'_1\rangle$.
   We have $M_1(cc) = M'_1(cc)$. So, $M'_1 =_k M_1$. Because no label chain changed, $kstut(M')$ gives $kstut(M'_1)$. Also, $\tau|_\ell = \tau'|_\ell = \epsilon$ and $C_n = C'_n = C_1; \textbf{exit}$.

5. C is **while** e **do** C_1 **end**:

   Similarly to above.

6. C is $C_1; C_2$:

   By IH on C_1.

7. C is **exit**:

   We have $C_n = C'_n = \textbf{stop}$. Also $\tau|_\ell = \tau'|_\ell = \epsilon$. From $M' =_k M$, we get $M(cc) = M'(cc)$ and $M(bc) = M'(bc)$. So, $M_t(cc) = M'_t(cc)$ and $M_t(bc) = M'_t(bc)$. Only variables in W change. Assume $w \in W$. We have that
   $\forall 0 \leq i \leq k$: $M'_n(T^i(w)) = M'(T^i(w)) \sqcup M'(cc) \sqcup M'(bc) =$

$M(T^i(w)) \sqcup M(cc) \sqcup M(bc) = M_n(T^i(w))$.

Alos, $\forall i > k$, we have $M'_n(T^i(w)) = M'_n(T^k(w))$. So, $M'_n =_k M_n$ and $kstut(M'_n)$.

□

Lemma 23. *If* $\langle C, M\rangle \to \langle C', M'\rangle$ *according to* k*-Enf, and* $2stut(M)$*, then* $2stut(M')$.

Proof. Assume $\langle C, M_I\rangle \to \langle C'_I, M'_I\rangle$ according to ∞-*Enf*, where $M_I =_k M$ and $kstut(M_I)$. From Lemma 22, we get $M'_I =_k M'$ and $kstut(M'_I)$. Because $2stut(M)$, $M_I =_k M$, and $kstut(M_I)$, we have $2stut(M_I)$. From Lemma 21, we get $2stut(M'_I)$. From $M'_I =_k M'$, we then get $2stut(M')$. □

B.3 Optimized Enforcer k-*Eopt*

We sketch the construction of k-*Eopt*. We add two rules for **if** command (one for each truth value of the guard) to k-*Enf*. These new rules apply to a simple **if** command. We add a premise to the existing rules for **if** command, so that these rules are triggered when this **if** command is not simple. The new rules for simple **if** command augment the taken branch with a new delimiter i-**end**, and we add one rule for i-**end** to k-*Enf*; this rule sets certain labels of label chains to $\perp$. Notice, there are programs where k-*Eopt* produces more permissive label chains than those produced by k-*Enf*.

Figure B.1 gives the rules for augmenting k-*Enf* in order to obtain k-*Eopt*. Function $isSimple(C, M, i)$ decides whether a command C is simple:

(i) C is of the form **if** $a > 0$ **then** $w_i := e$ **else** $w_i := n$ **end**,

$$\text{(IFS1)}\ \frac{\exists i\colon 1 \le i \le k\colon \mathit{isSimple}(\textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M, i) \quad M(e) \neq 0 \qquad cc' = M(cc).\mathit{push}(\langle M(T(e)),\ \emptyset,\ \emptyset\rangle}{\langle \textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M\rangle \to \langle C_1; i\textbf{-exit}, M[cc \mapsto cc']\rangle}$$

$$\text{(IFS2)}\ \frac{\exists i\colon 1 \le i \le k\colon \mathit{isSimple}(\textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M, i) \quad M(e) = 0 \qquad cc' = M(cc).\mathit{push}(\langle M(T(e)),\ \emptyset,\ \emptyset\rangle}{\langle \textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M\rangle \to \langle C_2; i\textbf{-exit}, M[cc \mapsto cc']\rangle}$$

$$\text{(EXIT\_IFS)}\ \frac{cc' = cc.\mathit{pop}}{\langle i\textbf{-exit}, M\rangle \to \langle \textbf{stop}, M[\forall j\colon i < j \le k\colon T^j(w_i) \mapsto \bot, cc \mapsto cc']\rangle}$$

Figure B.1: Rules for simple **if** command

(ii) a is an anchor variable,

(iii) w_i is a flexible variable,

(iv) $i = 1$ and $M(T(e)) = \bot$, or
$i > 1$, $M(T^{i-1}(e)) \neq \bot$, and $M(T^i(e)) = \bot$,

(v) n is a constant,

(vi) C is context-free (e.g., $M(cc) = \epsilon$ and $M(bc) = \bot$).

Notice that if $\mathit{isSimple}(C, M, i)$ holds, then $\mathit{isSimple}(C, M, j)$ does not hold for $j \neq i$, due to (iv) and monotonically decreasing label chains.

As an example, we show how k-*Eopt* deduces label chains for the following simple **if**:

$$\textbf{if } m > 0 \textbf{ then } w := h \textbf{ else } w := 4 \textbf{ end} \tag{B.4}$$

where anchor variable m is associated with $\langle \mathsf{M}, \bot, \bot, \bot\rangle$, anchor variable h is associated with $\langle \mathsf{H}, \bot, \bot, \bot\rangle$, and $h \neq 4$. Without considering the context (i.e., $m > 0$), flexible variable w would be associated with either $\langle \mathsf{H}, \bot, \bot, \bot\rangle$ (due to $w := h$) or $\langle \bot, \bot, \bot, \bot\rangle$ (due to $w := 4$), when execution of assignments ends. Here, only w and $T(w)$ reveal information about guard $m > 0$. So, at the end

of the conditional command, only $T(w)$ and $T^2(w)$ should be updated with the sensitivity of the context $T(m) = \mathsf{M}$. Thus, if $m > 0$, then w is associated with $\langle \mathsf{H}, \mathsf{M}, \bot, \bot \rangle$, at the end of the conditional command. Otherwise, w is associated with $\langle \mathsf{M}, \mathsf{M}, \bot, \bot \rangle$. Notice that, in both cases, the meta-meta label of w is strictly less restrictive than its metalabel. So, using the metalabel to specify its own sensitivity would be conservative. In particular, using rules from k-*Enf*, w would be associated with $\langle \mathsf{M}, \mathsf{M}, \mathsf{M}, \mathsf{M} \rangle$ or $\langle \mathsf{H}, \mathsf{M}, \mathsf{M}, \mathsf{M} \rangle$ at the end of the execution. Consequently, k-*Enf* deduces less permissive label chains than k-*Eopt*.

Consider now how k-*Eopt* produces label chains for the following simple **if**:

$$\textbf{if } a > 0 \textbf{ then } w_i := e \textbf{ else } w_i := n \textbf{ end}$$

where $T(a) = A$, $T^j(e) \neq \bot$ for $j < i$, and $T^j(e) = \bot$ for $j \geq i$. Without considering the context, we have

$$\forall j \geq i \colon T^j(w_i) = \bot$$

at the end of both branches. Only $T^j(w_i)$, for $j < i$, reveal information about guard $a > 0$. So, at the end of the conditional command, only $T^{j+1}(w_i)$, for $j < i$, should be updated with $T(a) = A$. Thus, at the end of the conditional command, we always have

$$\forall j > i \colon T^j(w_i) = \bot.$$

So, when execution exits a simple **if** command, $T^j(w_i)$ can be set to $\bot$, for every $j > i$.

Consider now lattice $\mathcal{L}_3 \triangleq \langle \{\mathsf{H}, \mathsf{M}, \mathsf{L}\}, \sqsubseteq \rangle$ with $\bot = \mathsf{L} \sqsubset \mathsf{M} \sqsubset \mathsf{H}$ and the

following program:

$$\textbf{if } m > 0 \textbf{ then } w := h \textbf{ else } w := 4 \textbf{ end};$$

$$\textbf{if } l > 0 \textbf{ then } w' := w \textbf{ else } w := m \textbf{ end};$$

$$w'' := w'$$

where l is anchor variable with $T(l) = \bot$ and w', w'' are flexible variables. If $m \not> 0$ and $l > 0$, then w'' is associated with $\langle \mathsf{M}, \mathsf{M}, \bot, \bot \rangle$. If $l \not> 0$, then w'' is associated with $\langle \mathsf{M}, \bot, \bot, \bot \rangle$. So, k-*Eopt* produces 2-precise 2-varying label chains for the target variable w''. Such an example can be extended to show that k-*Eopt* can produce k-precise k-varying label chains.

For enforcer k-*Eopt*, we have $n_{k\text{-}Eopt} = k + 1$, $Aux_{k\text{-}Eopt} = \{cc, bc\}$, $Init_{k\text{-}Eopt}(cc) = \epsilon$, and $Init_{k\text{-}Eopt}(bc) = \bot$.

B.4 Soundness of k-*Eopt*

Lemma 24. *k-Eopt is an enforcer on R for k-BNI with $k \geq 2$.*

Proof. We first add the rules in Figure B.1 to k-*Enf* and retrieve k-*Eopt*. Also,

we substitute (IF1) and (IF2) with:

$$(\text{IF1}')\ \frac{\begin{array}{c}\nexists i\colon 1 \le i \le k\colon \mathit{isSimple}(\textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M, i) \\ M(e) \neq 0 \qquad W = \mathit{targetFlex}(C_2) \\ A = \mathit{targetAnchor}(C_2) \qquad cc' = M(cc).\mathit{push}(\langle M(T(e)),\ W,\ A\rangle)\end{array}}{\langle \textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M\rangle \to \langle C_1; \textbf{exit}, M[cc \mapsto cc']\rangle}$$

$$(\text{IF2}')\ \frac{\begin{array}{c}\nexists i\colon 1 \le i \le k\colon \mathit{isSimple}(\textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M, i) \\ M(e) = 0 \qquad W = \mathit{targetFlex}(C_1) \\ A = \mathit{targetAnchor}(C_1) \qquad cc' = M(cc).\mathit{push}(\langle M(T(e)),\ W,\ A\rangle)\end{array}}{\langle \textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M\rangle \to \langle C_2; \textbf{exit}, M[cc \mapsto cc']\rangle}$$

It is easy to prove that k-*Eopt* is an enforcer on R and satisfies restrictions (E1), (E2), and (E3) by induction on the rules for k-*Eopt*.[1] We omit the details.

We prove that BNI+(k-*Eopt*,$\mathcal{L}$,C) holds, for a command C and a lattice $\mathcal{L}$. Assume $\ell \in \mathcal{L}$ and

$$M|_\ell = M'|_\ell,$$

$$M(cc) = M'(cc), mon(M), mon(M')$$

$$\tau = \langle C, M\rangle \xrightarrow{*} \langle C_t, M_t\rangle \text{ according to } k\text{-}\mathit{Eopt},$$

$$\tau' = \langle C, M'\rangle \xrightarrow{*} \langle C'_t, M'_t\rangle \text{ according to } k\text{-}\mathit{Eopt}$$

where C_t and C'_t are terminations (normal or blocked).

We prove $c1$, $c2$, $c3$, and $c4$. We use structural induction on C. We build on the proof of Lemma 13. That proof uses lemmata 14, 15, 16, 17, 18, 19, and 20, which all still hold for k-*Eopt*.

If C is **skip** or $a := e$, then k-*Eopt* and ∞-*Enf* use the same rules. So, we follow

[1] For k-*Eopt*, we could use **exit** and introduce a new auxiliary for tracking when a simple **if** is executed. For simplicity, we instead introduce a new conditional delimiter i-**exit** and extend definition $\langle C, M\rangle =^0 \langle C', M'\rangle$ to hold even if the syntax of conditional delimiters that appear in C and C' is different.

the same proof as in Case 1 and Case 2 of Lemma 13.

If C is $w := e$, then k-*Eopt* and ∞-*Enf* use the same rule up to the kth tag. So, we follow the same proof as in Case 3 of Lemma 13 by bounding $r \leq k$ and $i \leq k$ and recalling $T^k(x) = T^{k+1}(x)$ (dy definition of k-*Eopt*).

If C is $C_1; C_2$, **while** e **do** C_t **end**, or **exit**, then k-*Eopt* and ∞-*Enf* use the same rules up to the kth tag. We follow the same proof as in Cases 4,6,7 of Lemma 13 by bounding $i \leq k$ and recalling $T^k(x) = T^{k+1}(x)$ (dy definition of k-*Eopt*).

Now, it suffices to prove that BNI+(k-*Eopt*,$\mathcal{L}$,C) holds, where C is an **if**. We first prove that if $isSimple(C, M, i)$ holds for some $1 \leq i \leq k$, then $isSimple(C, M', i)$ holds, too. Because $isSimple(C, M, i)$ holds, we get:

$$C \text{ is of the form } \textbf{if } a > 0 \textbf{ then } w_i := e_i \textbf{ else } w_i := n \textbf{ end}, \tag{B.5}$$

$$a \text{ is an anchor variable,} \tag{B.6}$$

$$w_i \text{ is a flexible variable,} \tag{B.7}$$

$$i = 1 \text{ and } M(T(e_i)) = \bot, \text{ or}$$
$$i > 1 \text{ and } M(T^{i-1}(e_i)) \neq \bot \text{ and } M(T^i(e_i)) = \bot \tag{B.8}$$

$$n \text{ is a constant,} \tag{B.9}$$

$$C \text{ is context-free (e.g., } M(cc) = \epsilon \text{ and } M(bc) = \bot). \tag{B.10}$$

- From (B.8), $mon(M)$, and $M|_\ell = M'|_\ell$, we have $M(T^i(e_i)) = M'(T^i(e_i))$ and if $i > 1$, then $M(T^{i-1}(e_i)) = M'(T^{i-1}(e_i))$.

  So,

  $$\begin{aligned} &i = 1 \text{ and } M'(T(e_i)) = \bot, \text{ or} \\ &i > 1 \text{ and } M'(T^{i-1}(e_i)) \neq \bot \text{ and } M'(T^i(e_i)) = \bot \end{aligned} \tag{B.11}$$

- From (B.10), we have $M(cc) = \epsilon$. From $M(cc) = M'(cc)$, we then get

  $$M'(cc) = \epsilon. \tag{B.12}$$

- From (B.10), we have $M(bc) = \bot$. From $M|_\ell = M'|_\ell$, we then get

$$M'(bc) = \bot. \tag{B.13}$$

From (B.5), (B.6), (B.7), (B.11), (B.9), (B.12), and (B.13), we get that $isSimple(C, M', i)$ holds. So, if $isSimple(C, M, i)$ holds, then $isSimple(C, M', i)$ holds, too. Similarly, if $isSimple(C, M', i)$ holds, then $isSimple(C, M, i)$ holds. Thus, τ and τ' both use IFS or IF′. If both τ and τ' use IF′ (i.e, IF1′ or IF2′), then we follow Case 5 of Lemma 13.

So, it remains to handle the case where τ and τ' both use IFS. A simple **if** does not contain assignments to anchor variables. So, a trace of a simple **if** never stops before normal termination. Thus, $c2$, $c3$, and $c4$ are trivially true.

We prove $c1$. Assume C is **if** $a > 0$ **then** $w_i := e_i$ **else** $w_i := n$ **end** and

$$M|_\ell = M'|_\ell,$$
$$M(cc) = M'(cc), mon(M), mon(M')$$
$$\tau = \langle C, M\rangle \to \langle C_b; i\text{-}\mathbf{exit}, M_b\rangle \to \langle i\text{-}\mathbf{exit}, M_e\rangle \to \langle \mathbf{stop}, M_t\rangle,$$
$$\tau' = \langle C, M'\rangle \to \langle C'_b; i\text{-}\mathbf{exit}, M'_b\rangle \to \langle i\text{-}\mathbf{exit}, M'_e\rangle \to \langle \mathbf{stop}, M'_t\rangle$$

where C_b and C'_b are either $w_i := e_i$ or $w_i := n$.

We prove $\tau|_\ell =_{obs} \tau'|_\ell$, $M_t|_\ell = M'_t|_\ell$, and $M_t(cc) = M'_t(cc)$, in the case τ and τ' both use IFS (i.e, IFS1 or IFS2). From $M(cc) = M'(cc)$, IFS, and EXIT_IFS, we get $M_t(cc) = M'_t(cc)$. It remains to prove that $\tau|_\ell =_{obs} \tau'|_\ell$ and $M_t|_\ell = M'_t|_\ell$.

We first compute the possible label chains that w_i may be associated with at different points of the execution of C. Notice that by the definition of simple **if** we have $\lfloor cc \rfloor = \bot$ and $bc = \bot$ at the beginning of its execution.

(I) After execution of $w_i := e_i$:

From AsgnF, IfS, and (B.8) we have:

$\forall j\colon 1 \leq j < i\colon T^j(w_i) = T^j(e_i) \sqcup T(a)$ and

$\forall j\colon i \leq j \leq k\colon T^j(w_i) = T(a)$.

(II) After execution of $w_i := n$:

From AsgnF and IfS we have:

$\forall j\colon 1 \leq j \leq k\colon T^j(w_i) = T(a)$.

(III) After execution of i-**exit** when $a > 0$ holds:

From Exit_IfS and (I) we have:

$\forall j\colon 1 \leq j < i\colon T^j(w_i) = T^j(e_i) \sqcup T(a)$ and $T^i(w_i) = T(a)$ and

$\forall j\colon i < j \leq k\colon T^j(w_i) = \bot$.

(IV) After execution of i-**exit** when $a \not> 0$ holds:

From Exit_IfS and (II) we have:

$\forall j\colon 1 \leq j \leq i\colon T^j(w_i) = T(a)$ and $\forall j\colon i < j \leq k\colon T^j(w_i) = \bot$.

By definition of anchor variables, $M(T^2(a)) = \bot$. From $M|_\ell = M'|_\ell$, we then get $M(T(a)) = M'(T(a))$.

We prove $\tau|_\ell =_{obs} \tau'|_\ell$ and $M_t|_\ell = M'_t|_\ell$.

1. $M(T(a)) \sqsubseteq \ell$:

From $M|_\ell = M'|_\ell$, we get $M(a) = M'(a)$. So, τ and τ' take the same branch. We first prove $\tau|_\ell =_{obs} \tau'|_\ell$. Because these observations might involve only w_i and its associated label chain, it suffices to show that: for j such that $1 \leq j \leq k+1$, if $M_e(T^j(w_i)) \sqsubseteq \ell$, then $M'_e(T^j(w_i)) \sqsubseteq \ell$ and $M_e(T^{j-1}(w_i)) = M'_e(T^{j-1}(w_i))$. We examine two cases based on the branch that is executed.

- Branch $w_i := e_i$ is executed.

  Consider j such that $i+1 \leq j \leq k+1$.

  Due to (I), we have $M_e(T^{j-1}(w_i)) = M'_e(T^{j-1}(w_i)) = M(T(a))$. Because $T^{k+1}(w_i) = T^k(w_i)$, we also have $M_e(T^{k+1}(w_i)) = M'_e(T^{k+1}(w_i)) = M(T(a))$.

  Consider $j = i$. $M_e(T^j(w_i)) \sqsubseteq \ell$ and $M'_e(T^j(w_i)) \sqsubseteq \ell$ hold because from (I), we have $M_e(T^j(w_i)) = M'_e(T^j(w_i)) = M(T(a))$ and $M(T(a)) \sqsubseteq \ell$. From (B.8) and $M|_\ell = M'|_\ell$ we have $M(T^{i-1}(e_i)) = M'(T^{i-1}(e_i))$, and thus, (I) gives $M_e(T^{j-1}(w_i)) = M(T^{i-1}(e_i)) \sqcup M(T(a)) = M'(T^{i-1}(e_i)) \sqcup M'(T(a)) = M'_e(T^{j-1}(w_i))$.

  Consider $j < i$. Assume $M_e(T^j(w_i)) \sqsubseteq \ell$. Then, from (I), we have $M(T^j(e_i)) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$ and Lemma 17, we then have $M(T^{j-1}(e_i)) = M'(T^{j-1}(e_i))$. For $j = 1$, we then have $M(e_i) = M'(e_i)$. For $j \neq 1$, (I) gives $M_e(T^{j-1}(w_i)) = M(T^{j-1}(e_i)) \sqcup M(T(a)) = M'(T^{j-1}(e_i)) \sqcup M'(T(a)) = M'_e(T^{j-1}(w_i))$.

- Branch $w_i := n$ is executed.

  From (II) and because $T^{k+1}(w_i) = T^k(w_i)$, we get

  $\forall j$: $1 \leq j \leq k+1$: $M_e(T^j(w_i)) = M'_e(T^j(w_i)) = M(T(a))$.

  Also, $M_e(w_i) = M'_e(w_i) = n$.

So, $\tau|_\ell =_{obs} \tau'|_\ell$ holds.

We now prove $M_t|_\ell = M'_t|_\ell$. Because $M_t(cc) = M'_t(cc)$ and because bc is not modified, it suffices to prove $M_t|_\ell = M'_t|_\ell$ for the label chain of w_i. We prove for j with $1 \leq j \leq k+1$ that if $M_t(T^j(w_i)) \sqsubseteq \ell$, then $M'_t(T^j(w_i)) \sqsubseteq \ell$ and $M_t(T^{j-1}(w_i)) = M'_t(T^{j-1}(w_i))$. We examine two cases based on the branch that is executed.

- Branch $w_i := e_i$ is executed.

  From (III) and because $T^{k+1}(w_i) = T^k(w_i)$, we get

  $\forall j$: $i \leq j \leq k+1$: $M_t(T^j(w_i)) = M'_t(T^j(w_i))$.

  So, it suffices to examine $j \leq i$.

  Consider $j = i$. From (B.8) and $M|_\ell = M'|_\ell$ we have $M(T^{i-1}(e_i)) = M'(T^{i-1}(e_i))$, and thus, (III) gives $M_t(T^{j-1}(w_i)) = M(T^{i-1}(e_i)) \sqcup M(T(a)) = M'(T^{i-1}(e_i)) \sqcup M(T(a)) = M'_t(T^{j-1}(w_i))$.

  Consider $j < i$. Assume $M_t(T^j(w_i)) \sqsubseteq \ell$. Then, from (III), we have $M(T^j(e_i)) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$, we then have $M(T^{j-1}(e_i)) = M'(T^{j-1}(e_i))$. For $j = 1$, we then have $M(e_i) = M'(e_i)$. For $j \neq 1$, we then have $M_t(T^{j-1}(w_i)) = M(T^{j-1}(e_i)) \sqcup M(T(a)) = M'(T^{j-1}(e_i)) \sqcup M'(T(a)) = M'_t(T^{j-1}(w_i))$.

- Branch $w_i := n$ is executed.

  From (IV) and because $T^{k+1}(w_i) = T^k(w_i)$, we get

  $\forall j$: $1 \leq j \leq k+1$: $M_t(T^j(w_i)) = M'_t(T^j(w_i))$.

  Also, $M_t(w_i) = M'_t(w_i) = n$.

Thus, $M_t|_\ell = M'_t|_\ell$ holds.

2. $M(T(a)) \not\sqsubseteq \ell$:

   Traces τ and τ' may take different branches. From (I), (II), $M(T(a)) \not\sqsubseteq \ell$, and because $T^{k+1}(w_i) = T^k(w_i)$, we get that:

   $\forall j$: $1 \leq j \leq k+1$: $M_e(T^j(w_i)) \not\sqsubseteq \ell \wedge M'_e(T^j(w_i)) \not\sqsubseteq \ell$.

   Thus, $\tau|_\ell =_{obs} \epsilon$ and $\tau'|_\ell =_{obs} \epsilon$.

   We now prove $M_t|_\ell = M'_t|_\ell$. Because $M_t(cc) = M'_t(cc)$ and because bc is not modified, it suffices to prove $M_t|_\ell = M'_t|_\ell$ for the label chain of w_i. We prove for j with $1 \leq j \leq k+1$ that if $M_t(T^j(w_i)) \sqsubseteq \ell$, then $M'_t(T^j(w_i)) \sqsubseteq \ell$ and $M_t(T^{j-1}(w_i)) = M'_t(T^{j-1}(w_i))$. (III) and (IV) give $\forall j$: $1 \leq j \leq i$: $T^j(w_i) \not\sqsubseteq \ell$. It

then suffices to prove that the following holds:

$\forall j$: $i \leq j \leq k+1$: $M_t(T^j(w_i)) = M'_t(T^j(w_i))$.

For $j = i$, we have $M_t(T^j(w_i)) = M(T(a)) = M'_t(T^j(w_i))$. For j with $i < j < k+1$, we have $M_t(T^j(w_i)) = \bot = M'_t(T^j(w_i))$. Because $T^{k+1}(w_i) = T^k(w_i)$, we also have $M_t(T^{k+1}(w_i)) = \bot = M'_t(T^{k+1}(w_i))$. So, $M_t|_\ell = M'_t|_\ell$ holds.

So, BNI+(k-*Eopt*,$\mathcal{L}$,C) holds. Because BNI+(k-*Eopt*,$\mathcal{L}$,C) implies k-BNI(k-*Eopt*,$\mathcal{L}$,C), we get that k-BNI(k-*Eopt*,$\mathcal{L}$,C) holds, too. □

B.5 Permissiveness of k-*Enf* versus Chain Length

Theorem 5. *k-Enf $<^{k,\mathcal{L}}_{\rho_k}$ $(k+1)$-Enf, for $k \geq 2$ and any lattice $\mathcal{L}$ with at least one non-bottom element.*

Proof. We first prove k-*Enf* $\leq^{k,\mathcal{L}}_{\rho_k}$ $(k+1)$-*Enf*. Consider $\tau = \mathit{trace}_{k\text{-}Enf}(C, M)$ with $M \models \mathcal{H}_0(k\text{-}Enf, \mathcal{L}, C)$. Consider M' such that $M' \models \mathcal{H}_0((k+1)\text{-}Enf, \mathcal{L}, C)$, $M|^k = M'|^k$, and $\tau' = \mathit{trace}_{(k+1)\text{-}Enf}(C, M')$. Using induction on the number of steps in τ and Lemma 25, we get that $\tau|^k_\ell \preceq \tau'|^k_\ell$, for all $\ell \in \mathcal{L}$. So, k-*Enf* $\leq^{k,\mathcal{L}}_{\rho_k}$ $(k+1)$-*Enf*.

To prove k-*Enf* $<^{k,\mathcal{L}}_{\rho_k}$ $(k+1)$-*Enf*, it suffices to also show that $(k+1)$-*Enf* $\not\leq^{k,\mathcal{L}}_{\rho_k}$ k-*Enf*. Because $\mathcal{L}$ contains at least one non-bottom element ℓ, with $\bot \sqsubset \ell$, there exists M_1 such that $M_1 \models \mathcal{H}_0((k+1)\text{-}Enf, \mathcal{L}, C)$,
$\tau = \mathit{trace}_{(k+1)\text{-}Enf}(w := w_1, M_1) = \langle w := w_1, M_1\rangle \rightarrow \langle \mathbf{stop}, M_2\rangle$,
and $M_1(T^{k+1}(w_1)) \sqsubset M_1(T^k(w_1))$. There exists M'_1 such that $M'_1 \models \mathcal{H}_0(k\text{-}Enf, \mathcal{L}, C)$, $M_1|^k = M'_1|^k$, and

$\tau' = \mathit{trace}_{k\text{-}Enf}(w := w_1, M_1') = \langle w := w_1, M_1' \rangle \rightarrow \langle \mathbf{stop}, M_2' \rangle$.

From $M_1|^k = M_1'|^k$, we have $M_1'(T^k(w_1)) = M_1(T^k(w_1))$. By definition of k-*Enf* (i.e., $\forall x : T^{k+1}(x) = T^k(x)$), we then have

$M_1'(T^{k+1}(w_1)) = M_1'(T^k(w_1)) = M_1(T^k(w_1)) \sqsupset M_1(T^{k+1}(w_1))$,

and thus, we get $M_2'(T^{k+1}(w)) \sqsupset M_2(T^{k+1}(w))$. So, τ generates observation involving $T^k(w)$ to label $M_2(T^{k+1}(w))$, but τ' does not generate observation involving $T^k(w)$ to label $M_2(T^{k+1}(w))$. So, $(k+1)$-*Enf* $\not\leq_{\rho_k}^{k,\mathcal{L}}$ k-*Enf*. Thus, k-*Enf* $<_{\rho_k}^{k,\mathcal{L}}$ $(k+1)$-*Enf*. □

Lemma 25. *Consider step* $\tau = \langle C, M_1 \rangle \rightarrow \langle C_2, M_2 \rangle$ *generated by* k-*Enf for* $k \geq 2$. *Consider step* $\tau' = \langle C, M_1' \rangle \rightarrow \langle C_2', M_2' \rangle$ *generated by* $(k+1)$-*Enf*.
If $M_1 =_k M_1'$, *then* $C_2 = C_2'$, $M_2 =_k M_2'$, *and* $\tau|_\ell^k \preceq \tau'|_\ell^k$, *for all* ℓ.

Proof. By structural induction on C. □

Theorem 6. k-*Enf* $\cong_c^{k,\mathcal{L}}$ $(k+1)$-*Enf for any lattice* $\mathcal{L}$ *and* $k \geq 2$

Proof. We prove k-*Enf* $\leq_c^{k,\mathcal{L}}$ $(k+1)$-*Enf* and $(k+1)$-*Enf* $\leq_c^{k,\mathcal{L}}$ k-*Enf*.
Consider conventionally initialized memory M with $M \models \mathcal{H}_0(k\text{-}Enf, \mathcal{L}, C)$ and $\tau = \mathit{trace}_{k\text{-}Enf}(C, M)$. Consider M' such that $M' \models \mathcal{H}_0((k+1)\text{-}Enf, \mathcal{L}, C)$, $\rho_1(M, M')$, and $\tau' = \mathit{trace}_{(k+1)\text{-}Enf}(C, M')$. So, M' is conventionally initialized, too. Thus, we have $\mathit{2stut}(M)$ and $\mathit{2stut}(M')$. Also, because M and M' are conventionally initialized and $\rho_1(M, M')$ holds, we get that $M =_k M'$ holds. Using induction on the number of steps in τ and Lemma 26, we get that $\tau|_\ell^k = \tau'|_\ell^k$, for all $\ell \in \mathcal{L}$. So, k-*Enf* $\leq_c^{k,\mathcal{L}}$ $(k+1)$-*Enf* and $(k+1)$-*Enf* $\leq_c^{k,\mathcal{L}}$ k-*Enf*. Thus, k-*Enf* $\cong_c^{k,\mathcal{L}}$ $(k+1)$-*Enf*. □

Lemma 26. *Consider step* $\tau = \langle C, M_1 \rangle \rightarrow \langle C_2, M_2 \rangle$ *generated by* k-*Enf for* $k \geq 2$. *Consider step* $\tau' = \langle C, M_1' \rangle \rightarrow \langle C_2', M_2' \rangle$ *generated by* $(k+1)$-*Enf*.

If $M_1 =_k M_1'$, $2stut(M_1)$, *and* $2stut(M_1')$, *then* $C_2 = C_2'$, $M_2 =_k M_2'$, $2stut(M_2)$, $2stut(M_2')$, *and* $\tau|_\ell^k = \tau'|_\ell^k$, *for all* ℓ.

Proof. By structural induction on C. □

Theorem 7. k-*Enf* $\cong_{\rho_k}^{0,\mathcal{L}}$ $(k+1)$-*Enf for any lattice* $\mathcal{L}$ *and* $k \geq 2$.

Proof. Lemma 27 gives 2-*Enf* $\cong_{\rho_2}^{0,\mathcal{L}}$ k-*Enf* for any lattice $\mathcal{L}$ and $k \geq 2$. Because $\rho_k \Rightarrow \rho_2$, we then get 2-*Enf* $\cong_{\rho_k}^{0,\mathcal{L}}$ k-*Enf* for $k \geq 2$. By transitivity, we then have k-*Enf* $\cong_{\rho_k}^{0,\mathcal{L}}$ $(k+1)$-*Enf*. □

Lemma 27. 2-*Enf* $\cong_{\rho_2}^{0,\mathcal{L}}$ k-*Enf for any lattice* $\mathcal{L}$ *and* $k > 2$.

Proof. We prove k-*Enf* $\leq_{\rho_2}^{0,\mathcal{L}}$ 2-*Enf* and 2-*Enf* $\leq_{\rho_2}^{0,\mathcal{L}}$ k-*Enf*. Consider memory M with $M \models \mathcal{H}_0(\text{2-}\textit{Enf}, \mathcal{L}, C)$ and $\tau = trace_{\text{2-}\textit{Enf}}(C, M)$. Consider memory M' such that $M' \models \mathcal{H}_0(k\text{-}\textit{Enf}, \mathcal{L}, C)$, $\rho_2(M, M')$, and $\tau' = trace_{k\text{-}\textit{Enf}}(C, M')$. Using induction on the number of steps in τ and Lemma 28, we get that $\tau|_\ell^0 = \tau'|_\ell^0$, for all $\ell \in \mathcal{L}$. So, k-*Enf* $\leq_{\rho_2}^{0,\mathcal{L}}$ 2-*Enf* and 2-*Enf* $\leq_{\rho_2}^{0,\mathcal{L}}$ k-*Enf*. Thus, 2-*Enf* $\cong_{\rho_2}^{0,\mathcal{L}}$ k-*Enf*. □

Lemma 28. *Consider step* $\tau = \langle C, M_1\rangle \to \langle C_2, M_2\rangle$ *generated by* k-*Enf for* $k \geq 2$.
Consider step $\tau' = \langle C, M_1'\rangle \to \langle C_2', M_2'\rangle$ *generated by* 2-*Enf*.
If $M_1 =_2 M_1'$, *then* $C_2 = C_2'$, $M_2 =_2 M_2'$, *and* $\tau|_\ell^0 = \tau'|_\ell^0$, *for all* ℓ.

Proof. By structural induction on C. □

B.6 Other Enforcers

Strong Threat Model

Theorem 8. *For a lattice $\mathcal{L}$, for an enforcer E that satisfies $(k-1)$-BNI($\mathcal{L}$), with $k \geq 2$, and produces some k-precise k-varying label chains with elements in $\mathcal{L}$, and for an enforcer E' that produces $(k-1)$-dependent label chains,*

$$\textit{if } E \leq_c^{k-1,\mathcal{L}} E', \textit{ then } E' \textit{ does not satisfy } (k-1)\textit{-BNI}(\mathcal{L}).$$

Enforcer E and lattice $\mathcal{L}$ exist.

Proof. First we prove that E and $\mathcal{L}$ exist. Lemma 29 gives that k-$Eopt$ is an enforcer, satisfies $(k-1)$-BNI($\mathcal{L}_k$), and produces some k-precise k-varying label chains with elements in $\mathcal{L}_k$, which is defined in (B.14). So, $\mathcal{L}$ exists and it can be $\mathcal{L}_k$, and E exists and it can be k-$Eopt$.

Assume a lattice $\mathcal{L}$ and an enforcer E that satisfies $(k-1)$-BNI($\mathcal{L}$) and produces some k-precise k-varying label chains with elements in $\mathcal{L}$:

$$\Omega = \langle \ell_1, \ell_2, \ldots, \ell_k \rangle \text{ and } \Omega' = \langle \ell_1, \ell_2, \ldots, \ell'_k \rangle$$

with $\ell_k \neq \ell'_k$. Assume an enforcer E' that produces $(k-1)$-dependent label chains and $E \leq_c^{k-1,\mathcal{L}} E'$.

We prove that E' does not satisfy $(k-1)$-BNI($\mathcal{L}$). Assume for contradiction that E' satisfies $(k-1)$-BNI($\mathcal{L}$). We have that there are j, C, and $M \models \mathcal{H}_0(E, \mathcal{L}, C)$ such that Ω is k-precise at the jth state of $\tau = trace_E(C, M)$. There exists a memory M_1 such that M_1 is conventionally initialized, $M_1 \models \mathcal{H}_0(E', \mathcal{L}, C)$ holds,

and $\rho_1(M, M_1)$. Let $\tau_1 = trace_{E'}(C, M_1)$. By definition of k-precise and because $E \leq_c^{k-1,\mathcal{L}} E'$ and E' satisfies $(k-1)$-BNI($\mathcal{L}$), we then get that τ_1 produces Ω at the jth state. So, by definition, $1 \leq j \leq |\tau_1|$ and there exists w such that:

$$\tau_1[j-1] = \langle w := e; C_r, M_w \rangle,\ \tau_1[j] = \langle C_r, M_r \rangle,$$
$$\forall i\colon 1 \leq i \leq k\colon M_r(T^i(w)) = \ell_i.$$

Working similarly for Ω', we get:

$$\tau_2[s-1] = \langle w' := e'; C'_r, M'_w \rangle,\ \tau_2[s] = \langle C'_r, M'_r \rangle,$$
$$\forall i\colon 1 \leq i < k\colon M'_r(T^i(w')) = \ell_i,\ M'_r(T^k(w')) = \ell'_k$$

for $\tau_2 = trace_{E'}(C', M_2)$, conventionally initialized memory M_2 with $M_2 \models \mathcal{H}_0(E', \mathcal{L}, C')$, and $1 \leq s \leq |\tau_2|$. Because E' uses $(k-1)$-dependent label chains, there exists a function f such that:

$$M_r(T^k(w)) = f(M_r(T(w)), \ldots, M_r(T^{k-1}(w)))$$
$$M'_r(T^k(w')) = f(M'_r(T(w')), \ldots, M'_r(T^{k-1}(w'))).$$

Because $\forall i\colon 1 \leq i < k\colon M_r(T^i(w)) = M'_r(T^i(w')) = \ell_i$, we then have $M_r(T^k(w)) = M'_r(T^k(w'))$. But $M_r(T^k(w)) = \ell_k$, $M'_r(T^k(w')) = \ell'_k$, and $\ell_k \neq \ell'_k$ give $M_r(T^k(w)) \neq M'_r(T^k(w'))$, which is a contradiction. □

Lemma 29. *For $k \geq 2$, k-Eopt is an enforcer, satisfies $(k-1)$-BNI($\mathcal{L}_k$), and produces k-precise k-varying label chains with elements in $\mathcal{L}_k$, which is defined in (B.14).*

Proof. Lemma 24 gives that k-$Eopt$ is an enforcer on R for k-BNI. Thus, k-$Eopt$ satisfies $(k-1)$-BNI($\mathcal{L}_k$).

Lemma 30 gives the possible label chains that k-$Eopt$ produces for each z_j in pgm_k, which is defined below. Lemma 31 gives that these label chains are k-precise. The last label chain in (Z_{k-1}) is $\langle \ell_{k-1}, \ell_{k-1}, \ldots, \ell_{k-1}, \bot \rangle$ and has length k.

The penultimate label chain in (Z_k) is $\langle \ell_{k-1}, \ell_{k-1}, \ldots, \ell_{k-1}, \ell_k \rangle$ and has length k. The above two label chains take elements from $\mathcal{L}_k$ and they are k-varying. $\square$

Definition of pgm_k for $k \geq 2$

Let pgm_k be the following program:

$$\begin{array}{l}
\textbf{if } a_1 > 0 \textbf{ then } w_1 := 0 \textbf{ else } w_1 := 1 \textbf{ end}; \\
z_1 := w_1; \\
\textbf{if } a_2 > 0 \textbf{ then } w_2 := z_1 \textbf{ else } w_2 := 2 \textbf{ end}; \\
z_2 := w_2; \\
\ldots \\
\textbf{if } a_{k-1} > 0 \textbf{ then } w_{k-1} := z_{k-2} \textbf{ else } w_{k-1} := k-1 \textbf{ end}; \\
z_{k-1} := w_{k-1}; \\
\textbf{if } a_k > 0 \textbf{ then } w_k := z_{k-1} \textbf{ else } w_k := k \textbf{ end}; \\
z_k := w_k;
\end{array}$$

where all w_k and z_k are flexible variables. Assume lattice $\mathcal{L}_k$ of labels such that

$$\ell_0 \sqsupset \ell_1 \sqsupset \ell_2 \sqsupset \ldots \sqsupset \ell_k \sqsupset \bot \tag{B.14}$$

Assume $\mathcal{L}_k$ consists only of $\bot, \ell_j$, for $0 \leq j \leq k$. Assume pgm_k is executed with conventionally initialized memory M under k-*Eopt*, where $M(T(a_j)) = \ell_j$, for $0 \leq j \leq k$ and $k \geq 2$.

($\mathbf{Z_1}$) After the execution of $z_1 := w_1$, this is the possible chain for z_1:

$$\begin{array}{cccccc}
 & T(z_1) & T^2(z_1) & \ldots & T^k(z_1) & \\
\langle & \ell_1 & \bot & \ldots & \bot & \rangle
\end{array}$$

($\mathbf{Z_2}$) After the execution of $z_2 := w_2$, these are the possible 2 chains for z_2:

$$\begin{array}{cccccc||l} & T(z_2) & T^2(z_2) & T^3(z_2) & \dots & & \\ \langle & \ell_1 & \ell_2 & \bot & \dots & \rangle & a_2 > 0 \\ \langle & \ell_2 & \ell_2 & \bot & \dots & \rangle & a_2 \not> 0 \end{array}$$

($\mathbf{Z_3}$) After the execution of $z_3 := w_3$, these are the possible 3 chains for z_3:

$$\begin{array}{cccccc||l} & T(z_3) & T^2(z_3) & T^3(z_3) & T^4(z_3) & \dots & \\ \langle & \ell_1 & \ell_2 & \ell_3 & \bot & \dots & \rangle & a_2 > 0 \wedge a_3 > 0 \\ \langle & \ell_2 & \ell_2 & \ell_3 & \bot & \dots & \rangle & a_2 \not> 0 \wedge a_3 > 0 \\ \langle & \ell_3 & \ell_3 & \ell_3 & \bot & \dots & \rangle & a_3 \not> 0 \end{array}$$

...

($\mathbf{Z_j}$) After the execution of $z_j := w_j$, these are the possible j chains for z_j:

$$\begin{array}{cccccccccc||l} & T & T^2 & T^3 & \dots & T^{j-1} & T^j & T^{j+1} & \dots & & \\ \langle & \ell_1 & \ell_2 & \ell_3 & \dots & \ell_{j-1} & \ell_j & \bot & \dots & \rangle & a_2, a_3, \dots, a_j > 0 \\ \langle & \ell_2 & \ell_2 & \ell_3 & \dots & \ell_{j-1} & \ell_j & \bot & \dots & \rangle & a_2 \not> 0 \wedge a_3, \dots, a_j > 0 \\ \langle & \ell_3 & \ell_3 & \ell_3 & \dots & \ell_{j-1} & \ell_j & \bot & \dots & \rangle & a_3 \not> 0 \wedge a_4, \dots, a_j > 0 \\ \dots & & & & & & & & & & \\ \langle & \ell_{j-1} & \ell_{j-1} & \ell_{j-1} & \dots & \ell_{j-1} & \ell_j & \bot & \dots & \rangle & a_{j-1} \not> 0 \wedge a_j > 0 \\ \langle & \ell_j & \ell_j & \ell_j & \dots & \ell_j & \ell_j & \bot & \dots & \rangle & a_j \not> 0 \end{array}$$

($\mathbf{Z_k}$) After the execution of $z_k := w_k$, these are the possible k chains for z_i:

$$\begin{array}{ccccccccc||l} & T & T^2 & T^3 & \dots & T^{k-1} & T^k & & & \\ \langle & \ell_1 & \ell_2 & \ell_3 & \dots & \ell_{k-1} & \ell_k & \rangle & a_2 > 0 \wedge a_3 > 0 \wedge \dots \wedge a_j > 0 \\ \langle & \ell_2 & \ell_2 & \ell_3 & \dots & \ell_{k-1} & \ell_k & \rangle & a_2 \not> 0 \wedge a_3 > 0 \wedge \dots \wedge a_k > 0 \\ \langle & \ell_3 & \ell_3 & \ell_3 & \dots & \ell_{k-1} & \ell_k & \rangle & a_3 \not> 0 \wedge \dots \wedge a_k > 0 \\ \dots & & & & & & & & & \\ \langle & \ell_{k-1} & \ell_{k-1} & \ell_{k-1} & \dots & \ell_{k-1} & \ell_k & \rangle & a_{k-1} \not> 0 \wedge a_k > 0 \\ \langle & \ell_k & \ell_k & \ell_k & \dots & \ell_k & \ell_k & \rangle & a_k \not> 0 \end{array}$$

Lemma 30. *The label chains presented after pgm_k are the only possible chains that k-Eopt produces for variables z_j, where $k \geq 2$ and $1 \leq j \leq k$.*

Proof. Induction on j.

Base case: $j = 1$.

When execution reaches C_1:

$$\textbf{if } a_1 > 0 \textbf{ then } w_1 := 0 \textbf{ else } w_1 := 1 \textbf{ end}$$

with a memory M', then $isSimple(C_1, M', 1)$ is always satisfied. Using IFS, EXIT_IFS, and ASGNF rules, we get that z_1 is always associated with: $\langle \ell_1, \bot, \ldots, \bot \rangle$.

Induction case:

IH: chains presented after pgm_k for z_{j-1}, with $j > 1$, are the only possible chains that k-Eopt produces for z_{j-1}.

We prove that chains presented after pgm_k are the only possible chains that k-Eopt produces for z_j. When execution reaches C_j:

$$\textbf{if } a_j > 0 \textbf{ then } w_j := z_{j-1} \textbf{ else } w_j := j \textbf{ end}$$

with some memory M', then using IH on z_{j-1} we get that $isSimple(C_j, M', j)$ is always satisfied. So, rules IFS and EXIT_IFS are used while executing C_j.

1. $a_j > 0$

Assignment $w_j := z_{j-1}$ is executed. From IH, IFS and ASGNF, we have that w_j is

associated after $w_j := z_{j-1}$ with one of the following $j-1$ label chains:

$$\begin{array}{cccccccccc||l}
 & T & T^2 & T^3 & \dots & T^{j-1} & T^j & T^{j+1} & \dots & & \\
\langle & \ell_1 & \ell_2 & \ell_3 & \dots & \ell_{j-1} & \ell_j & \ell_j & \dots & \rangle & a_2, a_3, \dots, a_{j-1} > 0 \\
\langle & \ell_2 & \ell_2 & \ell_3 & \dots & \ell_{j-1} & \ell_j & \ell_j & \dots & \rangle & a_2 \not> 0 \wedge a_3, \dots, a_{j-1} > 0 \\
\langle & \ell_3 & \ell_3 & \ell_3 & \dots & \ell_{j-1} & \ell_j & \ell_j & \dots & \rangle & a_3 \not> 0 \wedge a_4, \dots, a_{j-1} > 0 \\
\dots & & & & & & & & & & \\
\langle & \ell_{j-1} & \ell_{j-1} & \ell_{j-1} & \dots & \ell_{j-1} & \ell_j & \ell_j & \dots & \rangle & a_{j-1} \not> 0
\end{array}$$

From EXIT_IFS, we have that w_j is associated at the end of C_j with one of the following $j-1$ label chains:

$$\begin{array}{cccccccccc||l}
 & T & T^2 & T^3 & \dots & T^{j-1} & T^j & T^{j+1} & \dots & & \\
\langle & \ell_1 & \ell_2 & \ell_3 & \dots & \ell_{j-1} & \ell_j & \bot & \dots & \rangle & a_2, a_3, \dots, a_{j-1} > 0 \\
\langle & \ell_2 & \ell_2 & \ell_3 & \dots & \ell_{j-1} & \ell_j & \bot & \dots & \rangle & a_2 \not> 0 \wedge a_3, \dots, a_{j-1} > 0 \\
\langle & \ell_3 & \ell_3 & \ell_3 & \dots & \ell_{j-1} & \ell_j & \bot & \dots & \rangle & a_3 \not> 0 \wedge a_4, \dots, a_{j-1} > 0 \\
\dots & & & & & & & & & & \\
\langle & \ell_{j-1} & \ell_{j-1} & \ell_{j-1} & \dots & \ell_{j-1} & \ell_j & \bot & \dots & \rangle & a_{j-1} \not> 0
\end{array}$$

2. $a_j \not> 0$

   Assignment $w_j := j$ is executed. From IFS and ASGNF, we have that w_j is associated after $w_j := j$ with the following label chain:

$$\begin{array}{cccccccccc||}
 & T & T^2 & T^3 & \dots & T^{j-1} & T^j & T^{j+1} & \dots & & \\
\langle & \ell_j & \ell_j & \ell_j & \dots & \ell_j & \ell_j & \ell_j & \dots & \rangle &
\end{array}$$

   From EXIT_IFS, we have that w_j is associated at the end of C_j with the following label chain:

$$\begin{array}{cccccccccc||}
 & T & T^2 & T^3 & \dots & T^{j-1} & T^j & T^{j+1} & \dots & & \\
\langle & \ell_j & \ell_j & \ell_j & \dots & \ell_j & \ell_j & \bot & \dots & \rangle &
\end{array}$$

So, using ASGNF and the j above possible label chains produced for w_j at the end of C_j, we get that label chains presented after pgm_k for z_j are the only possible label chains that k-Eopt produces for z_j. □

Lemma 31. *The label chains produced by k-Eopt for each z_j in pgm_k are k-precise, where $k \geq 2$ and $1 \leq j \leq k$.*

Proof. Consider j such that $1 \leq j \leq k$ and $k \geq 2$. We prove that the label chains for z_j produced by k-*Eopt* are k-precise. We use induction on the number n of elements in these label chains, where $1 \leq n \leq k$.

Base case: $n = 1$

We prove that the label chains produced by k-*Eopt* for z_j are 1-precise. Consider $\tau = trace_{k\text{-}Eopt}(pgm_k, M)$, where M is conventionally initialized and $M \models \mathcal{H}_0(k\text{-}Eopt, \mathcal{L}_k, pgm_k)$. Then $\tau[s-1] = \langle z_j := w_j; C, M_0\rangle$ and $\tau[s] = \langle C, M_1\rangle$ for some $1 < s \leq |\tau|$. Trace τ produces label chain $\Omega = \langle M_1(T(z_j)), M_1(T^2(z_j)), \ldots, M_1(T^k(z_j))\rangle$ at the sth state.

We prove that Ω is 1-precise. Ω may be one of the j possible label chains in $(\mathbf{Z_j})$. So, the only possible labels for $M_1(T(z_j))$ are $\ell_1, \ell_2, \ldots, \ell_j$. Let $M_1(T(z_j)) = \ell_i$ for $1 \leq i \leq j$. Only the ith label chain in $(\mathbf{Z_j})$ has $T(z_j) = \ell_i$. So, it should be the case that

$$\begin{aligned} M(a_i) \not> 0, M(a_{i+1}) > 0, \ldots M(a_j) > 0 \quad &\text{if } i \neq j \\ M(a_j) \not> 0 \quad &\text{if } i = j \end{aligned} \tag{B.15}$$

Consider an enforcer D that satisfies 0-BNI($\mathcal{L}_k$) and k-*Eopt* $\leq_c^{0,\mathcal{L}_k} D$. Consider memory M' with $M' \models \mathcal{H}_0(D, \mathcal{L}_k, pgm_k)$, $\rho_1(M, M')$ and $\tau' = trace_D(pgm_k, M')$. It should be the case that $\tau'[s] = \langle C, M_1'\rangle$, because otherwise $\forall \ell \in \mathcal{L}_k$: $\tau|_\ell^0 \trianglelefteq \tau'|_\ell^0$ (and k-*Eopt* $\leq_c^{0,\mathcal{L}_k} D$) would not hold.

We prove $M_1'(T(z_j)) = M_1(T(z_j)) = \ell_i$. Assume for contradiction that $M_1'(T(z_j)) \neq M_1(T(z_j))$. Either $M_1'(T(z_j)) \not\sqsubset M_1(T(z_j))$ or $M_1'(T(z_j)) \sqsubset M_1(T(z_j))$ holds. From $M_1'(T(z_j)) \not\sqsubset M_1(T(z_j))$ and $M_1'(T(z_j)) \neq M_1(T(z_j))$, we get $\tau|_{\ell_i}^0 \not\trianglelefteq \tau'|_{\ell_i}^0$, which implies that $k\text{-}Eopt \not\leq_c^{0,\mathcal{L}_k} D$, which is a contradiction. Assume $M_1'(T(z_j)) \sqsubset M_1(T(z_j))$. From $M_1'(T(z_j)) \sqsubset M_1(T(z_j))$ and $M_1(T(z_j)) = \ell_i$, we then have $M_1'(T(z_j)) \sqsubset \ell_i$. Let $\ell = M_1'(T(z_j))$. There exists M''' such that $M''' \models \mathcal{H}_0(k\text{-}Eopt, \mathcal{L}_k, pgm_k)$ and

$$M \text{ and } M''' \text{ agree on everything except for } a_i > 0 \tag{B.16}$$

So M''' is conventionally initialized.

Let $\tau''' = trace_{k\text{-}Eopt}(pgm_k, M''')$. We have $\tau'''[s-1] = \langle z_j := w_j; C, M_0''' \rangle$ and $\tau'''[s] = \langle C, M_1''' \rangle$. There exists M'' such that $M'' \models \mathcal{H}_0(D, \mathcal{L}_k, pgm_k)$ and $\rho_1(M''', M'')$ hold.

Let $\tau'' = trace_D(pgm_k, M'')$. We should have $\tau''[s-1] = \langle z_j := w_j; C, M_0'' \rangle$ and $\tau''[s] = \langle C, M_1'' \rangle$, because otherwise $\forall \ell \in \mathcal{L}_k$: $\tau'''|_\ell^0 \trianglelefteq \tau''|_\ell^0$ (and $k\text{-}Eopt \leq_c^{0,\mathcal{L}_k} D$) would not hold.

Because M is conventionally initialized, and because we have $\rho_1(M, M')$, $\rho_1(M''', M'')$, and (B.16), we get that M' and M'' agree on everything except for $a_i > 0$. In particular, from $\rho_1(M, M')$ and (B.15) we get $M'(T(a_i)) = M(T(a_i)) = \ell_i$ and

$$\begin{aligned} M'(a_i) \not> 0, M'(a_{i+1}) > 0, \ldots M'(a_j) > 0 \text{ if } i \neq j \\ M'(a_j) \not> 0 \text{ if } i = j \end{aligned} \tag{B.17}$$

From (B.15), (B.16), and $\rho_1(M''', M'')$, we get

$$\begin{aligned} M''(a_i) > 0, M''(a_{i+1}) > 0, \ldots M''(a_j) > 0 \text{ if } i \neq j \\ M''(a_j) > 0 \text{ if } i = j \end{aligned} \tag{B.18}$$

Because M' and M'' agree on everything except for a_i and because $M'(T(a_i)) = \ell_i \not\sqsubseteq \ell$, we get $M'|_\ell = M''|_\ell$. So, based on pgm_k, we get $M_1''(z_j) < i$ and $M_1'(z_j) = i$ for $1 \leq i \leq j$. Thus, $M_1'(z_j) \neq M_1''(z_j)$.

Because $M_1'(T(z_j)) = \ell$, observation $\langle z_j, M_1'(z_j)\rangle$ appears in $\tau'|_\ell^0$. If $M_1''(z_j) \not\sqsubseteq \ell$, then observation $\langle z_j, M_1''(z_j)\rangle$ does not appear in $\tau''|_\ell^0$, and thus D does not satisfy 0-BNI($\mathcal{L}_k$). If $M_1''(z_j) \sqsubseteq \ell$, then observation $\langle z_j, M_1''(z_j)\rangle$ appears in $\tau''|_\ell^0$. But because $M_1'(z_j) \neq M_1''(z_j)$, D does not satisfy 0-BNI($\mathcal{L}_k$). So, in any case D does not satisfy 0-BNI($\mathcal{L}_k$), which is a contradiction. So, $M_1'(T(z_j)) = M_1(T(z_j))$. Thus, the label chain produced by k-*Eopt* for z_j is 1-precise.

Induction case:
IH: The label chains for z_j are n-precise, for $1 \leq n < k$.
We prove that the label chains for z_j are $(n+1)$-precise.

If $n + 1 \geq j + 1$, then $T^{n+1}(z_j) = \bot$, and thus using IH we get that the label chains for z_j are $(n+1)$-precise.

Assume $1 \leq n < j$. Consider $\tau = trace_{k\text{-}Eopt}(pgm_k, M)$ for a conventionally initialized memory M with $M \models \mathcal{H}_0(k\text{-}Eopt, \mathcal{L}_k, pgm_k)$. Then $\tau[s-1] = \langle z_j := w_j; C, M_0\rangle$ and $\tau[s] = \langle C, M_1\rangle$ for some $1 < s \leq |\tau|$. Trace τ produces label chain $\Omega = \langle M_1(T(z_j)), M_1(T^2(z_j)), \ldots, M_1(T^k(z_j))\rangle$ at the sth state.

We prove that Ω is $(n+1)$-precise. Ω may be one of the j possible label chains in ($\mathbf{Z_j}$). So, we get $M_1(T^{n+1}(z_j)) = \ell_m$, for $1 \leq m \leq j$. Consider an enforcer D that satisfies n-BNI($\mathcal{L}_k$) and k-*Eopt* $\leq_c^{n,\mathcal{L}_k} D$. Assume $\tau' = trace_D(pgm_k, M')$, for memory M' with $M' \models \mathcal{H}_0(D, \mathcal{L}_k, pgm_k)$ and $\rho_1(M, M')$. It should be the case that $\tau'[s-1] = \langle z_j := w_j; C, M_0'\rangle$ and $\tau'[s] = \langle C, M_1'\rangle$, because otherwise k-*Eopt* $\leq_c^{n,\mathcal{L}_k} D$ would not hold. From IH, n-BNI($\mathcal{L}_k, D$), and k-*Eopt* $\leq_c^{n,\mathcal{L}_k} D$ we

get that

$$\forall t\colon 1 \leq t \leq n\colon M_1'(T^t(z_j)) = M_1(T^t(z_j)). \tag{B.19}$$

We prove $M_1'(T^{n+1}(z_j)) = M_1(T^{n+1}(z_j))$. Assume for contradiction that $M_1'(T^{n+1}(z_j)) \neq M_1(T^{n+1}(z_j))$. Either $M_1'(T^{n+1}(z_j)) \not\sqsubseteq M_1(T^{n+1}(z_j))$ or $M_1'(T^{n+1}(z_j)) \sqsubset M_1(T^{n+1}(z_j))$ holds. From $M_1'(T^{n+1}(z_j)) \not\sqsubseteq M_1(T^{n+1}(z_j))$ and $M_1'(T^{n+1}(z_j)) \neq M_1(T^{n+1}(z_j))$, we get $k\text{-}Eopt \not\leq_c^{n,\mathcal{L}_k} D$, which is a contradiction.

Assume $M_1'(T^{n+1}(z_j)) \sqsubset M_1(T^{n+1}(z_j))$. So $M_1'(T^{n+1}(z_j)) \sqsubset \ell_m$. Let $\ell = M_1'(T^{n+1}(z_j))$. There exists memory M_m such that $M_m \models \mathcal{H}_0(k\text{-}Eopt, \mathcal{L}_k, pgm_k)$ and $M_m = M[\neg(M(a_m) > 0)]$, which denotes that M_m and M agree on everything expect for $a_m > 0$: $(M_m(a_m) > 0) = \neg(M(a_m) > 0)$. So, M_m is conventionally initialized.

Let $\tau_m = trace_{k\text{-}Eopt}(pgm_k, M_m)$. So, $\tau_m[s-1] = \langle z_j := w_j; C, M_{m0}\rangle$ and $\tau_m[s] = \langle C, M_{m1}\rangle$. From Lemma 32, we get

$$M_{m1}(T^n(z_j)) \neq M_1(T^n(z_j)). \tag{B.20}$$

There exists M'' with $M'' \models \mathcal{H}_0(D, \mathcal{L}_k, pgm_k)$, $\rho_1(M_m, M'')$, and $\tau'' = trace_D(pgm_k, M'')$. We should have $\tau''[s-1] = \langle z_j := w_j; C, M_0''\rangle$ and $\tau''[s] = \langle C, M_1''\rangle$, because otherwise $\forall \ell \in \mathcal{L}_k\colon \tau_m|_\ell^n \trianglelefteq \tau''|_\ell^n$ (and $k\text{-}Eopt \leq_c^{n,\mathcal{L}_k} D$) would not hold. From $\rho_1(M_m, M'')$, $\rho_1(M, M')$, and $M_m = M[\neg(M(a_m) > 0)]$, and because M, M_m are conventionally initialized, we get that M' and M'' agree on everything except for a_m: $M'(a_m > 0) = \neg M''(a_m > 0)$. From $\rho_1(M, M')$ and $M(T(a_m)) = \ell_m$, we get $M'(T(a_m)) = \ell_m$ and then $M''(T(a_m)) = \ell_m$. From $\ell_m \not\sqsubseteq \ell$, we then get $M'|_\ell = M''|_\ell$. By IH, n-BNI($\mathcal{L}_k, D$), and $k\text{-}Eopt \leq_c^{n,\mathcal{L}_k} D$ we get $M_1''(T^n(z_j)) = M_{m1}(T^n(z_j))$. From (B.19) and (B.20) we then get $M_1'(T^n(z_j)) \neq M_1''(T^n(z_j))$.

Because $M_1'(T^{n+1}(z_j)) = \ell$, observation $\langle T^n(z_j), M_1'(T^n(z_j))\rangle$ appears in $\tau'|^n_\ell$. If $M_1''(T^{n+1}(z_j)) \not\sqsubseteq \ell$, then observation $\langle T^n(z_j), M_1''(T^n(z_j))\rangle$ does not appear in $\tau''|^n_\ell$, and thus n-BNI($\mathcal{L}_k, D$) does not hold. If $M_1''(T^{n+1}(z_j)) \sqsubseteq \ell$, then observation $\langle T^n(z_j), M_1''(T^n(z_j))\rangle$ appears in $\tau''|^n_\ell$. But because $M_1'(T^n(z_j)) \neq M_1''(T^n(z_j))$, n-BNI($\mathcal{L}_k, D$) does not hold. So, in any case n-BNI($\mathcal{L}_k, D$) does not hold, which is a contradiction. So, $M_1'(T^{n+1}(z_j)) = M_1(T^{n+1}(z_j))$. Thus, the label chains produced by k-*Eopt* for z_j are $(n+1)$-precise. □

Lemma 32. *Consider* $\tau = trace_{k\text{-}Eopt}(pgm_k, M)$*, where* M *is conventionally initialized,* $\tau[s-1] = \langle z_j := w_j; C, M_0\rangle$ *and* $\tau[s] = \langle C, M_1\rangle$*. Assume* $M_1(T^{n+1}(z_j)) = \ell_m$*, with* $1 \leq n < j$*,* $1 \leq j \leq k$*, and* $1 \leq m \leq j$*. Then for* $\tau' = trace_{k\text{-}Eopt}(pgm_k, M')$ *with* $M' = M[\neg(M(a_m > 0))]$*,* $\tau'[s-1] = \langle z_j := w_j; C, M_0'\rangle$*, and* $\tau'[s] = \langle C, M_1'\rangle$*, we get* $M_1(T^n(z_j)) \neq M_1'(T^n(z_j))$*.*

Proof. 1. $M_1(T(z_j)) = \ell_m$

From ($\mathbf{Z_j}$) and $M_1(T^{n+1}(z_j)) = \ell_m$, we then get that $2 \leq n+1 \leq m$. So, $1 \leq n \leq m-1$. Also, $M_1(T^n(z_j)) = \ell_m$. Such a label chain $M_1(\Omega_{z_j})$ for z_j is generated when $M(a_m) \not> 0 \wedge M(a_{m+1}) > 0 \wedge \ldots \wedge M(a_j) > 0$. So, for M' we should have $M'(a_m) > 0 \wedge M'(a_{m+1}) > 0 \wedge \ldots \wedge M'(a_j) > 0$, because $M' = M[\neg(M(a_m > 0))]$. Thus, $M_1'(\Omega_{z_j})$ should be one of the label chains that appear above $M_1(\Omega_{z_j})$ in ($\mathbf{Z_j}$). From $1 \leq n \leq m-1$, we then have $M_1'(T^n(z_j)) \neq \ell_m$. So, $M_1(T^n(z_j)) \neq M_1'(T^n(z_j))$.

2. $M_1(T(z_j)) \neq \ell_m$

From ($\mathbf{Z_j}$) and $M_1(T^{n+1}(z_j)) = \ell_m$, we then have $M_1(T^n(z_j)) = \ell_{m-1}$ and $n = m-1$ (so $m \neq 1$). Also, M should have $M'(a_m) > 0 \wedge M'(a_{M+1}) > 0 \wedge \ldots \wedge M'(a_j) > 0$. So, M' should have $M'(a_m) \not> 0 \wedge M'(a_{M+1}) > 0 \wedge \ldots \wedge M'(a_j) > 0$, because $M' = M[\neg(M(a_m > 0))]$. Thus, $M_1'(\Omega_{z_j})$ is the mth label chain in ($\mathbf{Z_j}$).

So, $M_1'(T^n(z_j)) = M_1'(T^{m-1}(z_j)) = \ell_m$. Thus, $M_1(T^n(z_j)) \neq M_1'(T^n(z_j))$.

□

Weakened Threat Model

Theorem 9. *For an enforcer E and lattice $\mathcal{L}_3$, if $G^E_{a:=e}$ is 1-dependent and 2-Enf $\leq_c^{0,\mathcal{L}_3}$ E, then E does not satisfy 0-BNI($\mathcal{L}_3$).*

Proof. We have $\mathcal{L} = \langle\{\mathsf{L}, \mathsf{M}, \mathsf{H}\}, \sqsubseteq\rangle$ where $\mathsf{L} \sqsubset \mathsf{M} \sqsubset \mathsf{H}$ and $\bot = \mathsf{L}$.
Consider program pgm:

$$
\begin{aligned}
&w_a := m_a;\\
&\textbf{if } m_b > 0 \textbf{ then } w_b := m_b \textbf{ else } w_b := h \textbf{ end};\\
&\textbf{if } l > 1 \textbf{ then } w_c := w_a \textbf{ else } w_c := w_b \textbf{ end};\\
&w := w_c;\\
&m := w;\\
&l := 1
\end{aligned}
$$

where l, m_a, m_b, m, h are anchor variables with $T(l) = \mathsf{L}$, $T(m_a) = T(m_b) = T(m) = \mathsf{M}$, and $T(h) = \mathsf{H}$, and w, w_a, w_b, w_c are flexible variables.

2-*Enf* produces the following labels when executes pgm with a convention-

ally initialized memory:

	$m_b > 0$	$m_b \not> 0$
$l > 1$	$w_a : \langle \mathsf{M}, \mathsf{L} \rangle$	$w_a : \langle \mathsf{M}, \mathsf{L} \rangle$
	$w_b : \langle \mathsf{M}, \mathsf{M} \rangle$	$w_b : \langle \mathsf{H}, \mathsf{M} \rangle$
	$w : \langle \mathsf{M}, \mathsf{L} \rangle$	$w : \langle \mathsf{M}, \mathsf{L} \rangle$
$l \not> 1$	$w_a : \langle \mathsf{M}, \mathsf{L} \rangle$	$w_a : \langle \mathsf{M}, \mathsf{L} \rangle$
	$w_b : \langle \mathsf{M}, \mathsf{M} \rangle$	$w_b : \langle \mathsf{H}, \mathsf{M} \rangle$
	$w : \langle \mathsf{M}, \mathsf{M} \rangle$	$w : \langle \mathsf{H}, \mathsf{M} \rangle$

We first prove that, for all executions, $T(w)$ produced by 2-Enf is 1-precise. Consider $\tau = trace_{2\text{-}Enf}(pgm, M)$, where M is conventionally initialized and $M \models \mathcal{H}_0(2\text{-}Enf, \mathcal{L}, pgm)$. There exists s such that $\tau[s-1] = \langle w := w_c; C_r, M_w \rangle$ and $\tau[s] = \langle C_r, M_r \rangle$.

We prove that $M_r(T(w))$ is 1-precise. Consider E' an enforcer that satisfies 0-BNI($\mathcal{L}$) and 2-E $\preceq_c^{0,\mathcal{L}}$ E'. Assume $\tau' = trace_{E'}(pgm, M')$ where $M' \models \mathcal{H}_0(E', \mathcal{L}, pgm)$ and

$$\rho_1(M, M'). \tag{B.21}$$

From 2-Enf $\preceq_c^{0,\mathcal{L}}$ E', we get $\forall \ell \in \mathcal{L}$: $\tau|_\ell^0 \trianglelefteq \tau'|_\ell^0$. So, it should be the case that $\tau'[s-1] = \langle w := w_c; C_r, M'_w \rangle$ and $\tau'[s] = \langle C_r, M'_r \rangle$. We prove $M'_r(T(w)) = M_r(T(w))$.

1. $M(l) \not> 1$ and $M(m_b) > 0$ (a1)

   We have $M_r(T(w)) = \mathsf{M}$. We prove $M'_r(T(w)) = \mathsf{M}$. It should be the case that $M'_r(T(w)) \sqsubseteq \mathsf{M}$, because otherwise $\tau|_{\mathsf{M}}^0 \trianglelefteq \tau'|_{\mathsf{M}}^0$ would not hold, since observation $\langle w, M_r(w) \rangle$ would belong to $\tau|_{\mathsf{M}}^0$, but no observation involving w would belong to $\tau'|_{\mathsf{M}}^0$.

Assume for contradiction that $M'_r(T(w)) \sqsubset \mathsf{M}$. So, $M'_r(T(w)) = \mathsf{L}$. From (B.21) and (a1), we get

$$M'(l) \not> 1 \text{ and } M'(m_b) > 0 \tag{B.22}$$

There exists a memory M'' such that $M'' \models \mathcal{H}_0(E', \mathcal{L}, pgm)$ and:

$$M'|_{\mathsf{L}} = M''|_{\mathsf{L}}, \tag{B.23}$$

$$M''(m_b) \not> 0, \tag{B.24}$$

$$M'(m_b) \neq M''(h) \tag{B.25}$$

Let $\tau'' = trace_{E'}(pgm, M'')$. From (B.23) and because $T(l) = \mathsf{L}$, we get

$$M'(l) = M''(l). \tag{B.26}$$

From $M'(l) \not> 1$, we then get

$$M''(l) \not> 1. \tag{B.27}$$

If $M''_r(T(w)) \neq \mathsf{L}$, then E' does not satisfy 0-BNI($\mathcal{L}$), because (B.23) and $M'_r(T(w)) = \mathsf{L}$, and thus, observation $\langle w, M'_r(w)\rangle$ is included in $\tau'|^0_{\mathsf{L}}$ but no observation involving w is included in $\tau''|^0_{\mathsf{L}}$. So, $M''_r(T(w)) = \mathsf{L}$. Because E' is an enforcer and due to (B.27), (B.22), and (B.24), we have:
$M'_r(w) = M'_r(w_c) = M'_r(w_b) = M'(m_b)$ and
$M''_r(w) = M''_r(w_c) = M''_r(w_b) = M''(h)$.
From (B.25), we then have $M'_r(w) \neq M''_r(w)$. So, E' does not satisfy 0-BNI given (B.23), because $\tau'|^0_{\mathsf{L}}$ includes observation $\langle w, M'_r(w)\rangle$, $\tau''|^0_{\mathsf{L}}$ includes observation $\langle w, M''_r(w)\rangle$, and $M'_r(w) \neq M''_r(w)$. But this is a contradiction. Thus, $M'_r(T(w)) = \mathsf{M}$.

2. $M(l) \not> 1$ and $M(m_b) \not> 0$ (a2)

   We have $M_r(T(w)) = \mathsf{H}$. We prove that $M'_r(T(w)) = \mathsf{H}$. If $M'_r(T(w)) = \mathsf{L}$, then

we follow the arguments of the above case, where (B.22) would be $M'(m_b) \not> 0$ and (B.24) would be $M''(m_b) > 0$, and we are lead to a contradiction.

Assume for contradiction that $M'_r(T(w)) = \mathsf{M}$. There exists M'' such that $M'' \models \mathcal{H}_0(E', \mathcal{L}, pgm)$ and

$$M'|_{\mathsf{M}} = M''|_{\mathsf{M}}, \tag{B.28}$$

$$M'(h) \neq M''(h). \tag{B.29}$$

Let $\tau'' = trace_{E'}(pgm, M'')$. From (B.28), we get

$$M'(l) = M''(l), \tag{B.30}$$

$$M'(m_b) = M''(m_b). \tag{B.31}$$

From (B.21), (a2), (B.30), and (B.31), we get

$$M'(l) \not> 1, M''(l) \not> 1, M'(m_b) \not> 0, \text{ and } M''(m_b) \not> 0. \tag{B.32}$$

It should be the case that $M''_r(T(w)) = \mathsf{M}$, because otherwise E' would not satisfy 0-BNI($\mathcal{L}$), which is a contradiction. Because E' is an enforcer and due to (B.32), we have:

$M'_r(w) = M'_r(w_c) = M'_r(w_b) = M'(h)$ and

$M''_r(w) = M''_r(w_c) = M''_r(w_b) = M''(h)$.

From (B.29), we then have $M'_r(w) \neq M''_r(w)$. So, given (B.28), E' does not satisfy 0-BNI($\mathcal{L}$), which is a contradiction. Thus, $M'_r(T(w)) = \mathsf{H}$.

3. $M(l) > 1$ (a3)

We have $M_r(T(w)) = \mathsf{M}$. We prove that $M'_r(T(w)) = \mathsf{M}$. It should be the case that $M'_r(T(w)) \sqsubseteq \mathsf{M}$, because otherwise $\tau|^0_{\mathsf{M}} \trianglelefteq \tau'|^0_{\mathsf{M}}$ would not hold.

Assume for contradiction that $M'_r(T(w)) \sqsubset \mathsf{M}$. So, $M'_r(T(w)) = \mathsf{L}$. There exists

M'' such that

$$M'|_{\mathsf{L}} = M''|_{\mathsf{L}}, \tag{B.33}$$

$$M'(m_a) \neq M''(m_a) \tag{B.34}$$

Let $\tau'' = trace_{E'}(C, M'')$. From (B.33), we get

$$M'(l) = M''(l). \tag{B.35}$$

From (B.21) and (a3), we then have

$$M'(l) > 1 \text{ and } M''(l) > 1. \tag{B.36}$$

If $M''(T(w)) \neq \mathsf{L}$, then E' does not satisfy 0-BNI($\mathcal{L}$), which is a contradiction. Assume $M''(T(w)) = \mathsf{L}$. Because E' is an enforcer and due to (B.36), we have $M'_r(w) = M'_r(w_c) = M'_r(w_a) = M'(m_a)$ and
$M''_r(w) = M''_r(w_c) = M''_r(w_a) = M''(m_a)$.
From (B.34), we have $M'_r(w) \neq M''_r(w)$. So, given (B.33), E' does not satisfy 0-BNI($\mathcal{L}$), which is a contradiction. Thus, $M'_r(T(w)) = \mathsf{M}$.

So, for all executions, $T(w)$ produced by 2-Enf is 1-precise.

Consider an enforcer E that uses 1-dependent $G^E_{a:=e}$ and 2-Enf $\leq^{0,\mathcal{L}}_c$ E. We prove that E does not satisfy 0-BNI($\mathcal{L}$). Assume for contradiction that E satisfies 0-BNI($\mathcal{L}$). We examine whether E decides to block the execution of pgm for the following exhaustive list of cases: $l > 1$, $l \not> 1 \wedge m_b > 0$, and $l \not> 1 \wedge m_b \not> 0$. From that, we will show that E does not satisfy 0-BNI($\mathcal{L}$), which is a contradiction.

1. $l > 1$:

There exists a conventionally initialized memory M with $M \models$

$\mathcal{H}_0(2\text{-}Enf, \mathcal{L}, pgm)$ and $M(l) > 1$.

Let $\tau' = trace_{2\text{-}Enf}(pgm, M)$. There exists memory M_1 such that $M_1 \models \mathcal{H}_0(E, \mathcal{L}, pgm)$, $\rho_1(M, M_1)$, and $\tau_1 = trace_E(pgm, M_1)$. Because $2\text{-}Enf \leq_c^{0,\mathcal{L}} E$, we have $\forall \ell \in \mathcal{L}$: $\tau'|_\ell^0 \trianglelefteq \tau_1|_\ell^0$. From $\rho_1(M, M_1)$, we have

$$M_1(l) > 1. \tag{B.37}$$

Because $M(l) > 1$, enforcer $2\text{-}Enf$ executes $l := 1$, and thus, $\langle l, 1\rangle \in \tau'|_{\mathsf{L}}^0$. From $\forall \ell$: $\tau'|_\ell^0 \trianglelefteq \tau_1|_\ell^0$, we then get $\langle l, 1\rangle \in \tau_1|_{\mathsf{L}}^0$. Thus, $\langle l := 1, M_r\rangle \rightarrow \langle \mathbf{stop}, M_s\rangle$ should be a subtrace of τ_1. Thus, we have

$$M_r(G^E_{l:=1}) = true. \tag{B.38}$$

Because pgm is an anchor-tailed command, we can have $V_{l:=1} = \{l, m, w\}$. Because E uses 1-dependent $G^E_{l:=1}$, we get that

$$M_r(G^E_{l:=1}) = f(M_r(T(l)), M_r(T(m)), M_r(T(w))). \tag{B.39}$$

Because, for all executions, $T(w)$ produced by $2\text{-}Enf$ is 1-precise, and because $2\text{-}Enf \leq_c^{0,\mathcal{L}} E$, E satisfies 0-BNI($\mathcal{L}$), and $\rho_1(M, M_1)$, we get

$$M_r(T(l)) = \mathsf{L}, M_r(T(m)) = \mathsf{M}, M_r(T(w)) = \mathsf{M}. \tag{B.40}$$

From (B.38) and (B.39), we then get

$$f(\mathsf{L}, \mathsf{M}, \mathsf{M}) = true. \tag{B.41}$$

2. $l \not> 1$ and $m_b > 0$:

   There exists a conventionally initialized memory M' with $M' \models \mathcal{H}_0(2\text{-}Enf, \mathcal{L}, pgm)$ $M'(l) \not> 1$ and $M'(m_b) > 0$.

Let $\tau' = trace_{2\text{-}Enf}(pgm, M')$. There exists M_2 such that $M_2 \models \mathcal{H}_0(E, \mathcal{L}, pgm)$, $\rho_1(M', M_2)$, and $\tau_2 = trace_E(pgm, M_2)$. Because 2-$Enf$ $\leq_c^{0,\mathcal{L}} E$, we have that $\forall \ell \in \mathcal{L}$: $\tau'|_\ell^0 \trianglelefteq \tau_2|_\ell^0$. From $\rho_1(M', M_2)$, we have

$$M_2(l) \not> 1 \ \wedge \ M_2(m_b) > 0. \tag{B.42}$$

Because $M'(l) \not> 1$, and $M'(m_b) > 0$, enforcer 2-Enf executes $m := w$, and thus, $\langle m, \nu \rangle \in \tau'|_{\mathsf{M}}^0$. From $\forall \ell$: $\tau'|_\ell^0 \trianglelefteq \tau_2|_\ell^0$, we then get $\langle m, \nu \rangle \in \tau_2|_{\mathsf{M}}^0$. Thus, $\langle m := w; l := 1, M_m \rangle \rightarrow \langle l := 1, M_r \rangle$ should be a subtrace of τ_2. Because pgm is an anchor-tailed command, we can have $V_{l:=1} = \{l, m, w\}$. Because E uses 1-dependent $G_{l:=1}^E$, we get that

$$M_r(G_{l:=1}^E) = f(M_r(T(l)), M_r(T(m)), M_r(T(w))). \tag{B.43}$$

Because, for all executions, $T(w)$ produced by 2-Enf is 1-precise, and because 2-Enf $\leq_c^{0,\mathcal{L}} E$, E satisfies 0-BNI($\mathcal{L}$), and $\rho_1(M', M_2)$, we get

$$M_r(T(l)) = \mathsf{L}, M_r(T(m)) = \mathsf{M}, M_r(T(w)) = \mathsf{M}. \tag{B.44}$$

So, $M_r(G_{l:=1}^E) = f(\mathsf{L}, \mathsf{M}, \mathsf{M}) = true$, due to (B.41). So, E executes $l := 1$, and thus, $\langle l, 1 \rangle \in \tau_2|_{\mathsf{L}}^0$. (c1)

3. $l \not> 1$ and $m_b \not> 0$:

   There exists a conventionally initialized memory M'' with $M'' \models \mathcal{H}_0(2\text{-}Enf, \mathcal{L}, pgm)$, $dom(M'') = dom(M')$, $M'|_{\mathsf{L}} = M''|_{\mathsf{L}}$, $M''(l) \not> 1$, and $M''(m_b) \not> 0$.

   Let $\tau'' = trace_{2\text{-}Enf}(pgm, M'')$. There exists M_3 such that $M_3 \models \mathcal{H}_0(E, \mathcal{L}, pgm)$, $\rho_1(M'', M_3)$, and $\tau_3 = trace_E(pgm, M_3)$. Because 2-$Enf$ $\leq_c^{0,\mathcal{L}} E$, we have that $\forall \ell \in \mathcal{L}$: $\tau''|_\ell^0 \trianglelefteq \tau_3|_\ell^0$. From $\rho_1(M'', M_3)$, we have

   $$M_3(l) \not> 1 \ \wedge \ M_3(m_b) \not> 0. \tag{B.45}$$

From $dom(M'') = dom(M')$, $M'|_{\mathsf{L}} = M''|_{\mathsf{L}}$, $\rho_1(M'', M_3)$, $c(M'')$, $\rho_1(M', M_2)$, $c(M')$, we then get $M_2|_{\mathsf{L}} = M_3|_{\mathsf{L}}$. Enforcer 2-*Enf* executes $w := w_c$, and thus, $\langle w, \nu \rangle \in \tau''|^0_{\mathsf{H}}$. From $\forall \ell$: $\tau''|^0_\ell \trianglelefteq \tau_3|^0_\ell$, we then get $\langle w, \nu \rangle \in \tau_3|^0_{\mathsf{H}}$. Thus, $\langle w := w_c; m := w; l := 1, M_w \rangle \rightarrow \langle m := w; l := 1, M_m \rangle$ should be a subtrace of τ_3. We examine two cases: τ_3 either executes $m := w$ or not.

3.1. τ_3 does not execute $m := w$.

So, $l := 1$ is not executed either. (c2)

Thus, $\tau_2|^0_{\mathsf{L}} \neq \tau_3|^0_{\mathsf{L}}$, due to (c1) and (c2).

From $M_2|_{\mathsf{L}} = M_3|_{\mathsf{L}}$, we then get that E does not satisfy 0-BNI($\mathcal{L}$).

3.2. τ_3 executes $m := w$.

Then h is leaked to m. There exists M_4 such that $M_4 \models \mathcal{H}_0(E, \mathcal{L}, pgm)$ and

$$M_3|_{\mathsf{M}} = M_4|_{\mathsf{M}}, \tag{B.46}$$

$$M_3(h) \neq M_4(h). \tag{B.47}$$

From (B.46) and (B.45), we get

$$M_3(l) = M_4(l) \not> 1, \tag{B.48}$$

$$M_3(m_b) = M_4(m_b) \not> 0. \tag{B.49}$$

Let $\tau_4 = trace_E(pgm, M_4)$. If τ_4 does not execute $m := w$, then $\tau_3|_{\mathsf{M}} \neq \tau_4|_{\mathsf{M}}$. If τ_4 executes $m := w$, then (B.47) implies $\tau_3|_{\mathsf{M}} \neq \tau_4|_{\mathsf{M}}$, because in both traces τ_3 and τ_4 the value of m equals the value of h. So, in both cases, E does not satisfy 0-BNI($\mathcal{L}$).

□

$$(\textsc{AsgnA})\ \frac{v = M(e) \qquad G_{a:=e} \qquad \ell = M(\lfloor cc \rfloor) \sqcup M(bc)}{\langle a := e, M\rangle \to \langle \mathbf{stop}, M[a \mapsto v, bc \mapsto \ell]\rangle}$$

$$(\textsc{AsgnAFail})\ \frac{v = M(e) \qquad \neg G_{a:=e} \qquad \ell = M(\lfloor cc \rfloor) \sqcup M(bc)}{\langle a := e, M\rangle \to \langle \mathbf{block}, M[bc \mapsto \ell]\rangle}$$

$$(\textsc{AsgnF})\ \frac{\nu_0 = M(e) \qquad \nu_1 = M(T(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)}{\langle w := e, M\rangle \to \langle \mathbf{stop}, M[w \mapsto \nu_0, T(w) \mapsto \nu_1]\rangle}$$

$$U(M, W) \triangleq M[\ \forall w{\in}W\colon T(w) \mapsto T(w) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)]$$

Figure B.2: Modified rules for $E_{\mathsf{H,L}}$

Familiar Two-level Lattice

Theorem 10. *Enforcer $E_{\mathsf{H,L}}$ uses 1-dependent $G_{a:=e}$, satisfies 0-BNI($\mathcal{L}_2$), and satisfies 2-Enf $<_c^{0,\mathcal{L}_2}$ $E_{\mathsf{H,L}}$.*

Proof. Lemma 36 gives that $E_{\mathsf{H,L}}$ is an enforcer and uses 1-dependent $G_{a:=e}$. Lemma 33 gives that $E_{\mathsf{H,L}}$ satisfies 0-BNI($\mathcal{L}_2$). Lemma 34 gives that 2-*Enf* $<_c^{0,\mathcal{L}_2}$ $E_{\mathsf{H,L}}$ holds. □

Lemma 33. *$E_{\mathsf{H,L}}$ satisfies 0-BNI($\mathcal{L}_2$).*

Proof. We retrieve $E_{\mathsf{H,L}}$ from ∞-*Enf* by replacing rules for assignments and function U used in **exit** with the corresponding definitions in Figure B.2. Because BNI+(∞-*Enf*,$\mathcal{L}_2$,C) holds, then BNI+($E_{\mathsf{H,L}}$,$\mathcal{L}_2$,C) holds where C is any command different from assignment and **exit**, provided all Lemmata used to prove Lemma 13 hold for rules in Figure B.2. In particular, Lemmata 14, 15, 16, 19, 20, 17, and 18 still hold. Now, it suffices to prove BNI+($E_{\mathsf{H,L}}$,$\mathcal{L}_2$,C) where C is an assignment or **exit**.

For $E_{\mathsf{H,L}}$, the domain of memories contain only variables and tags of these variables. So, in the definition of BNI+, projections $M|_\ell$ and $\tau|^k_\ell$ equal projection $M|^0_\ell$ and $\tau|^0_\ell$, correspondingly. Also, $mon(M)$ is trivially true for any such memory So, the definition on BNI+ becomes: For all $\ell \in \mathcal{L}_2, M, M'$ if

$$
\begin{aligned}
&M|^0_\ell = M'|^0_\ell,\\
&M(cc) = M'(cc)\\
&\tau = trace_{E_{\mathsf{H,L}}}(C, M) = \langle C, M\rangle \xrightarrow{*} \langle C_t, M_t\rangle,\\
&\tau' = trace_{E_{\mathsf{H,L}}}(C, M') = \langle C, M'\rangle \xrightarrow{*} \langle C'_t, M'_t\rangle
\end{aligned}
$$

where C_t and C'_t are terminations (i.e., **stop** or **block**), then:

c1 If C_t and C'_t are both **stop**, then $\tau|^0_\ell =_{obs} \tau'|^0_\ell$, $M_t|^0_\ell = M'_t|^0_\ell$, and $M_t(cc) = M'_t(cc)$.

c2 If C_t or C'_t is **block**, then $\tau|^0_\ell =_{obs} \tau'|^0_\ell$.

c3 If C_t is **stop**, C'_t is **block**, and $M'_t(bc) \not\sqsubseteq \ell$, then $M_t(bc) \not\sqsubseteq \ell$.

c4 If C_t is **stop**, C'_t is **block**, $\langle C'_{tp}, M'_{tp}\rangle \rightarrow \langle C'_t, M'_t\rangle$ are the last two states of τ', and $M'_{tp}(\lfloor cc \rfloor) \sqsubseteq \ell$, then there exists $\langle C'', M''\rangle \in \tau$, with $C'' = C'_{tp}$ and $M''|_\ell = M'_{tp}|_\ell$.

Because $\ell \in \mathcal{L}_2$,we have that ℓ is either L or H. If $\ell = \mathsf{H}$, then BNI+ is trivially satisfied, because hypothesis $M|^0_\ell = M'|^0_\ell$ implies $M = M'$. Thus, we prove BNI+ for

$$\ell = \mathsf{L}. \tag{B.50}$$

1. C is $a := e$:

We prove that $M(T(a)) = M'(T(a))$. If $M(T(a)) = \mathsf{L}$, then $M|_\ell = M'|_\ell$ gives

$M'(T(a)) = \mathsf{L}$. If $M(T(a)) = \mathsf{H}$, then $M|_\ell = M'|_\ell$ gives $M'(T(a)) = \mathsf{H}$. So, $M(T(a)) = M'(T(a))$.

1.1. $M(T(a)) \sqsubseteq \ell$

We first prove that the command is executed normally in both memories or blocked in both memories. W.l.o.g, assume that the command is executed normally in M. That is, $M(T(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq M(T(a))$ holds. This implies that $M(T(e)) \sqsubseteq \ell$, $M(\lfloor cc \rfloor) \sqsubseteq \ell$, and $M(bc) \sqsubseteq \ell$. Because $M|_\ell = M'|_\ell$, we get $M(cc) = M'(cc)$ and $M(bc) = M'(bc)$. From $M(T(e)) \sqsubseteq \ell$ and $M|_\ell = M'|_\ell$, we then get $M'(T(e)) \sqsubseteq \ell$. From (B.50), we then have $M(T(e)) = M'(T(e)) = \mathsf{L}$.

Thus, $M'(T(e)) \sqcup M'(\lfloor cc \rfloor) \sqcup M'(bc) \sqsubseteq M'(T(a))$ holds. So, in both cases the command is executed normally. Thus, the command is executed normally in both memories or blocked in both memories. So, $c3$ and $c4$ are trivially true.

1.1.1. The command is executed normally in both memories.

$c2$ is trivially true.

We prove $c1$. We have:

$\tau = \langle a := e, \mathcal{M} \rangle \rightarrow \langle \mathbf{stop}, M[a \mapsto M(e), bc \mapsto \ell_g] \rangle$

$\tau' = \langle a := e, \mathcal{M}' \rangle \rightarrow \langle \mathbf{stop}, M'[a \mapsto M'(e), bc \mapsto \ell'_g] \rangle$,

where $\ell_g = M(\lfloor cc \rfloor) \sqcup M(bc)$ and $\ell'_g = M'(\lfloor cc \rfloor) \sqcup M'(bc)$. We have $\tau|_\ell = \tau'|_\ell = \langle a, M(e) \rangle$, because $M(e) = M'(e)$. Also, $\ell_g = \ell'_g$. So, $M_t(bc) = M'_t(bc)$. Because $M_t(cc) = M(cc) = M'(cc) = M'_t(cc)$, $M_t(cc) = M'_t(cc)$ holds. Because $M_t(a) = M'_t(a)$, $M_t(bc) = M'_t(bc)$, and $M_t(cc) = M'_t(cc)$, we get $M_t|_\ell = M'_t|_\ell$. Thus $c1$ holds.

1.1.2. The command is blocked in both memories.

$\tau = \langle a := e, \mathcal{M} \rangle \rightarrow \langle \mathbf{block}, M[bc \mapsto \ell_g] \rangle$

$\tau' = \langle a := e, \mathcal{M}' \rangle \to \langle \mathbf{block}, M'[bc \mapsto \ell'_g] \rangle$.

So, $\tau|_\ell =_{obs} \epsilon$ and $\tau'|_\ell =_{obs} \epsilon$. Thus $c2$ holds. $c1$ is trivially true.

1.2. $M(T(a)) \not\sqsubseteq \ell$

We have $\tau|_\ell =_{obs} \epsilon$ and $\tau'|_\ell =_{obs} \epsilon$. Thus $c2$ holds.

We prove $c1$. Assume $C_t = C'_t = \mathbf{stop}$. Because $M(T(a)), M'(T(a)) \not\sqsubseteq \ell$, M_t and M'_t do not need to agree on a. Assume $M_t(bc) \sqsubseteq \ell$. So, $M(T^2(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq \ell$. Thus, $M(T^2(e)) \sqsubseteq \ell$, $M(\lfloor cc \rfloor) \sqsubseteq \ell$, and $M(bc) \sqsubseteq \ell$. From $mon(M)$ and $M(T^2(e)) \sqsubseteq \ell$, we get $M(T^3(e)) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$, $M(T^3(e)) \sqsubseteq \ell$, $M(\lfloor cc \rfloor) \sqsubseteq \ell$, $M(bc) \sqsubseteq \ell$, and Lemma 17, we get: $M(T^2(e)) = M'(T^2(e))$, $M(\lfloor cc \rfloor) = M'(\lfloor cc \rfloor)$, and $M(bc) = M'(bc)$. So, $M_t(bc) = M'_t(bc)$. From $M(cc) = M'(cc)$, we get $M(cc) = M'(cc)$. From $M_t(cc) = M(cc)$ and $M'_t(cc) = M'(cc)$, we then get $M_t(cc) = M'_t(cc)$. Thus, $M_t|_\ell = M'_t|_\ell$ and $M_t(cc) = M'_t(cc)$. Thus $c1$ holds.

We prove $c3$. Assume C_t is **stop**, C'_t is **block**, and $M'_t(bc) \not\sqsubseteq \ell$. We prove $M_t(bc) \not\sqsubseteq \ell$. We prove the contrapositive. Assume $M_t(bc) \sqsubseteq \ell$, then following the same arguments as above, we get $M_t(bc) = M'_t(bc)$, and thus, $M'_t(bc) \sqsubseteq \ell$, as wanted.

We prove $c4$. Assume C_t is **stop**, C'_t is **block**, $\langle C'_{tp}, M'_{tp} \rangle \to \langle C'_t, M'_t \rangle$ are the last two states of τ', and $M'_{tp}(\lfloor cc \rfloor) \sqsubseteq \ell$. So, $M'_{tp} = M'$ and $C'_{tp} = a := e$. We have that $\langle C'', M'' \rangle = \langle a := e, M \rangle$, which satisfies $C'' = C'_{tp}$ and $M''|_\ell = M'_{tp}|_\ell$. Thus $c4$ holds.

2. C is $w := e$

$\tau = \langle w := e, M \rangle \to \langle \mathbf{stop}, M_t \rangle$

$\tau' = \langle w := e, M' \rangle \to \langle \mathbf{stop}, M'_t \rangle$

$c2$, $c3$, and $c4$ are trivially true.

We prove $c1$. $M_t(cc) = M'_t(cc)$ holds due to $M(cc) = M'(cc)$, $M_t(cc) = M(cc)$,

and $M'_t(cc) = M'(cc)$.

2.1. $M_t(T(w)) \sqsubseteq \ell$

Then $M(T(e)) \sqsubseteq \ell$, $M(\lfloor cc \rfloor) \sqsubseteq \ell$, and $M(bc) \sqsubseteq \ell$. From, $M|_\ell = M'|_\ell$ and (B.50), we then get $M(e) = M'(e)$, $M(T(e)) = M'(T(e))$, $M(cc) = M'(cc)$, and $M(bc) = M'(bc)$. So, $M_t(w) = M'_t(w)$ and $M_t(T(w)) = M'_t(T(w))$. Thus, $M_t|_\ell = M'_t|_\ell$ and $\tau|_\ell = \tau'|_\ell$. Thus $c1$ holds.

2.2. $M_t(T(w)) \not\sqsubseteq \ell$

By symmetry of preceding case, $M'_t(T(w)) \not\sqsubseteq \ell$. So, $\tau|_\ell =_{obs} \epsilon$ and $\tau'|_\ell =_{obs} \epsilon$. Also $M_t|_\ell = M'_t|_\ell$. Thus $c1$ holds.

3. **exit**

$\tau = \langle \mathbf{exit}, M \rangle \rightarrow \langle \mathbf{stop}, M_t \rangle$

$\tau' = \langle \mathbf{exit}, M' \rangle \rightarrow \langle \mathbf{stop}, M'_t \rangle$.

$c2$, $c3$, $c4$ are trivially true.

We prove $c1$. We have $\tau|_\ell =_{obs} \epsilon$ and $\tau'|_\ell =_{obs} \epsilon$. So, we need to prove $M_t|_\ell = M_t|_\ell$ and $M_t(cc) = M'_t(cc)$. From $M(cc) = M'(cc)$, we get $M(cc) = M'(cc)$. Because $M_t(cc) = M(cc).pop$ and $M'_t(cc) = M'(cc).pop$, we then get $M_t(cc) = M'_t(cc)$. We now prove $M_t|_\ell = M_t|_\ell$.

3.1. $M_t(\lfloor cc \rfloor) \sqcup M_t(bc) \not\sqsubseteq \ell$ and $M(cc).top.A \neq \emptyset$.

Because $M(cc).top.A \neq \emptyset$, we have $M_t(bc) = M(bc) \sqcup M(\lfloor cc \rfloor)$. We have $M_t(\lfloor cc \rfloor) \sqsubseteq M(\lfloor cc \rfloor)$. We get $M_t(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq M(\lfloor cc \rfloor) \sqcup M(bc)$, which becomes $M_t(\lfloor cc \rfloor) \sqcup M(bc) \sqcup M(\lfloor cc \rfloor) \sqsubseteq M(\lfloor cc \rfloor) \sqcup M(bc) \sqcup M(\lfloor cc \rfloor)$, which becomes $M_t(\lfloor cc \rfloor) \sqcup M_t(bc) \sqsubseteq M(\lfloor cc \rfloor) \sqcup M(bc)$. So, $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$. So, $M_t(bc) \not\sqsubseteq \ell$. Because $M|_\ell = M'|_\ell$, we also get $M'(\lfloor cc \rfloor) \sqcup M'(bc) \not\sqsubseteq \ell$.

Because $M(cc).top.A \neq \emptyset$ and $M(cc) = M'(cc)$, we have $M'(cc).top.A \neq \emptyset$, too. So, $M'_t(bc) = M'(bc) \sqcup M'(\lfloor cc \rfloor)$. Thus, $M'_t(bc) \not\sqsubseteq \ell$.

From $M_t(bc) \not\sqsubseteq \ell$ and $M'_t(bc) \not\sqsubseteq \ell$, we get $M_t(\lfloor cc \rfloor) \sqcup M_t(bc) \not\sqsubseteq \ell$ and $M'_t(\lfloor cc \rfloor) \sqcup M'_t(bc) \not\sqsubseteq \ell$.

Only variables in W change their labels. Let $x \in M(cc).top.W$. Because $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$, we have $M_t(T(x)) \not\sqsubseteq \ell$. Because $M(cc) = M'(cc)$, we get $x \in M'(cc).top.W$, too. Because $M'(\lfloor cc \rfloor) \sqcup M'(bc) \not\sqsubseteq \ell$, we have $M'_t(T(x)) \not\sqsubseteq \ell$. So, $M_t|_\ell = M'_t|_\ell$. Thus $c1$ holds.

3.2. $M_t(\lfloor cc \rfloor) \sqcup M_t(bc) \not\sqsubseteq \ell$ and $M(cc).top.A = \emptyset$.

Because $M(cc).top.A = \emptyset$, we have $M_t(bc) = M(bc)$. We have $M_t(\lfloor cc \rfloor) \sqsubseteq M(\lfloor cc \rfloor)$. We get $M_t(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq M(\lfloor cc \rfloor) \sqcup M(bc)$, which becomes $M_t(\lfloor cc \rfloor) \sqcup M_t(bc) \sqsubseteq M(\lfloor cc \rfloor) \sqcup M(bc)$. So, $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$. Because $M|_\ell = M'|_\ell$, we also get $M'(\lfloor cc \rfloor) \sqcup M'(bc) \not\sqsubseteq \ell$.

Because $M(cc).top.A = \emptyset$ and $M(cc) = M'(cc)$, we have that $M'(cc).top.A = \emptyset$. So, $M'_t(bc) = M'(bc)$.

Assume $M_t(bc) \sqsubseteq \ell$. Then $M(bc) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$, we then get $M(bc) = M'(bc)$. Thus, $M_t(bc) = M'_t(bc)$.

We prove $M'_t(\lfloor cc \rfloor) \sqcup M'_t(bc) \not\sqsubseteq \ell$. Assume for contradiction that $M'_t(\lfloor cc \rfloor) \sqcup M'_t(bc) \sqsubseteq \ell$. Then $M'_t(bc) \sqsubseteq \ell$. Following the same arguments as above, we get $M_t(bc) = M'_t(bc)$. Because $M_t(cc) = M'_t(cc)$, we then get $M_t(bc) \sqcup M_t(cc) = M'_t(bc) \sqcup M'_t(cc)$, which is a contradiction. So, $M'_t(\lfloor cc \rfloor) \sqcup M'_t(bc) \not\sqsubseteq \ell$.

Only variables in V change their labels. Let $x \in M(cc).top.W$. Because $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$, we have $M_t(T(x)) \not\sqsubseteq \ell$. Because $M(cc) = M'(cc)$, we get $x \in M'(cc).top.W$, too. Because $M'(\lfloor cc \rfloor) \sqcup M'(bc) \not\sqsubseteq \ell$, we have $M'_t(T(x)) \not\sqsubseteq \ell$. So, $M_t|_\ell = M'_t|_\ell$. Thus $c1$ holds.

3.3. $M_t(\lfloor cc \rfloor) \sqcup M_t(bc) \sqsubseteq \ell$

So, $M_t(bc) \sqsubseteq \ell$. From Lemma 19, we get $M(bc) \sqsubseteq \ell$. From $M|_\ell = M'|_\ell$

and $M(bc) \sqsubseteq \ell$, we also get $M(bc) = M'(bc)$. From $M(cc) = M'(cc)$, we then get $M_t(bc) = M'_t(bc)$.

- Let $M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq \ell$.

  Let $x \in M(cc).top.W$. Because $M(cc) = M'(cc)$, we get $x \in M'(cc).top.W$. We have:

  $M_t(T(x)) \sqsubseteq \ell \Rightarrow M(T(x)) \sqsubseteq \ell \Rightarrow M(x) = M'(x)$ and $M'(T(x)) \sqsubseteq \ell$.

  Because $M(cc) = M'(cc)$ and $M(bc) = M'(bc)$, we then have $M_t(x) = M'_t(x)$ and $M'_t(T(x)) \sqsubseteq \ell$. Thus, $M_t|_\ell = M'_t|_\ell$. Thus $c1$ holds.

- Let $M(\lfloor cc \rfloor) \sqcup M(bc) \not\sqsubseteq \ell$.

  So, $M'(\lfloor cc \rfloor) \sqcup M'(bc) \not\sqsubseteq \ell$. Let $x \in M(cc).top.W$. So, $M_t(T(x)) \not\sqsubseteq \ell$. Because $M(cc) = M'(cc)$, we get $x \in M'(cc).top.V$, and thus $M'_t(T(x)) \not\sqsubseteq \ell$. Thus, $M_t|_\ell = M'_t|_\ell$. Thus $c1$ holds.

Case *4.3.2.* in the proof of Lemma 13 mentions ASGNA.

We reexamine this case when rule ASGNA in Figure B.2 is instead used:

– $M_{tp}(bc) \sqsubseteq \ell$ and $M_{tp}(cc) \sqsubseteq \ell$

  From IH[$c4$] on C_1, there exists $\langle C_{1tp}, M'_1 \rangle \in \tau'_1$ such that $M_{tp}|_\ell = M'_1|_\ell$. So, $M_{tp}(bc) = M'_1(bc)$ and $M_{tp}(cc) = M'_1(cc)$. Because τ_1 is blocked, we have $M_{tp}(T(e)) \sqcup M_{tp}(\lfloor cc \rfloor) \sqcup M_{tp}(bc) \not\sqsubseteq M_{tp}(T(a))$. Since the inequality is satisfied in τ'_1, it means that the value of $T(e)$ is different in M'_1 and M_{tp}. But this contradicts $M_{tp}|_\ell = M'_1|_\ell$. Indeed, $M_{tp}|_\ell = M'_1|_\ell$ implies that $M'_1(T(e))$ and $M_{tp}(T(e))$ should either be both L or both H. Thus, this case is no longer possible once a two-level lattice is considered.

Thus, BNI+($E_{\mathsf{H,L}}$,$\mathcal{L}_2$,C) holds. BNI+ implies 0-BNI. So, $E_{\mathsf{H,L}}$ satisfies 0-BNI. □

Lemma 34. $2\text{-}Enf <_c^{0,\mathcal{L}_2} E_{\mathsf{H,L}}$

Proof. We first prove 2-*Enf* $\leq_c^{0,\mathcal{L}_2}$ $E_{\mathsf{H,L}}$. Consider conventionally initialized memory M with $M \models \mathcal{H}_0(\text{2-}Enf, \mathcal{L}_2, C)$. Consider memory M' with $M' \models \mathcal{H}_0(E_{\mathsf{H,L}}, \mathcal{L}_2, C)$ and $\rho_1(M, M')$. Assume $\tau' = trace_{E_{\mathsf{H,L}}}(C, M')$ and $\tau = trace_{\text{2-}Enf}(C, M)$. By definition, we have $M'(cc) = M(cc) = \epsilon$ and $M'(bc) = M(bc) = \bot$.

We write $M' \sqsubseteq_e M$ iff

1. $\forall x$: $M'(x) = M(x)$,
2. $\forall a$: $M'(T(a)) = M(T(a))$,
3. $\forall w$: $M'(T(w)) \sqsubseteq M(T(w))$,
4. $M'(bc) \sqsubseteq M(bc)$,
5. $\forall i \geq 0$: $M'(\lfloor cc.pop^i \rfloor) \sqsubseteq M(\lfloor cc.pop^i \rfloor) \wedge$
   $M'(cc.pop^i.top.A) = M(cc.pop^i.top.A) \wedge$
   $M'(cc.pop^i.top.W) = M(cc.pop^i.top.W)$

where $cc.pop^i$ pops the top i elements from cc and $cc.pop^0 = cc$.

So, $M' \sqsubseteq_e M$ holds. By induction on the number of steps in τ and Lemma 35, we get that $\forall \ell \in \mathcal{L}_2$: $\tau|_\ell^0 \trianglelefteq \tau'|_\ell^0$. So, 2-*Enf* $\leq_c^{0,\mathcal{L}_2}$ $E_{\mathsf{H,L}}$.

Now, we prove $E_{\mathsf{H,L}} \not\leq_c^{0,\mathcal{L}_2}$ 2-Enf. Consider program pgm:

if $h > 0$

  $w := h$

else

  $w := l$

end;

$h := w$;

$l := 1$

For every execution under 2-Enf we have:

- w is associated with $\langle \mathsf{H}, \mathsf{H} \rangle$,
- bc is H when reaching $l := 1$,
- so $l := 1$ is blocked.

For every execution under $E_{\mathsf{H,L}}$ we have:

- w is associated with H,
- bc is L when reaching $l := 1$,
- so $l := 1$ is allowed.

Thus, $E_{\mathsf{H,L}}$ produces observation $\langle l, 1 \rangle$, but 2-Enf does not. So, $E_{\mathsf{H,L}} \not\leq_c^{0,\mathcal{L}_2}$ 2-Enf. From 2-Enf $\leq_c^{0,\mathcal{L}_2}$ $E_{\mathsf{H,L}}$, we then have 2-Enf $<_c^{0,\mathcal{L}_2}$ $E_{\mathsf{H,L}}$. Notice that the same program pgm works for 2-$Eopt$, too. And thus we get 2-$Eopt$ $<_c^{0,\mathcal{L}_2}$ $E_{\mathsf{H,L}}$. □

Lemma 35. *If* $\langle C_1, M_1 \rangle \rightarrow \langle C_2, M_2 \rangle$ *under* 2-*Enf*, *and* $\langle C_1, M_1' \rangle \rightarrow \langle C_2', M_2' \rangle$ *under* $E_{\mathsf{H,L}}$, *and* $M_1' \sqsubseteq_e M_1$, *then* $C_2 = C_2'$ *and* $M_2' \sqsubseteq_e M_2$ *hold or* $C_2 =$ **block** *holds.*

Proof. We use induction on C_1. Assume $C_2 \neq \textbf{block}$. We prove $C_2 = C_2'$ and $M_2' \sqsubseteq_e M_2$.

1. C_1 is $a := e$.

   We have $C_2 = \textbf{stop}$. Then $M_1(T(e)) \sqcup M_1(\lfloor cc \rfloor) \sqcup M_1(bc) \sqsubseteq M_1(T(a))$ holds. Due to $M_1' \sqsubseteq_e M_1$, we then get $M_1'(T(e)) \sqcup M_1'(\lfloor cc \rfloor) \sqcup M_1'(bc) \sqsubseteq M_1'(T(a))$. So, $C_2' = \textbf{stop}$. Also, $M_2(a) = M_1(e) = M_1'(e) = M_2'(a)$. Because $M_2'(bc) = M_1'(\lfloor cc \rfloor) \sqcup M_1'(bc)$ and $M_2(bc) = M_1(T(e)) \sqcup M_1(\lfloor cc \rfloor) \sqcup M_1(bc)$, hypothesis $M_1' \sqsubseteq_e M_1$ gives $M_2'(bc) \sqsubseteq M_2(bc)$. So, $M_2' \sqsubseteq_e M_2$.

2. C_1 is $w := e$.

   We have $C_2 = C_2' = \textbf{stop}$. Hypothesis $M_1' \sqsubseteq_e M_1$ implies $M_1(e) = M_1'(e)$, and thus, $M_2(w) = M_2'(w)$. Because $M_2'(T(w)) = M_1'(T(e)) \sqcup M_1'(\lfloor cc \rfloor) \sqcup M_1'(bc)$ and $M_2(T(w)) = M_1(T(e)) \sqcup M_1(\lfloor cc \rfloor) \sqcup M_1(bc)$, hypothesis $M_1' \sqsubseteq_e M_1$ gives $M_2'(T(w)) \sqsubseteq M_2(T(w))$. So, $M_2' \sqsubseteq_e M_2$.

3. C_1 is **exit**

   We have $C_2 = C_2' = \textbf{stop}$. Because $M_2(cc) = M_1(cc).pop$ and $M_2'(cc) = M_1'(cc).pop$, hypothesis $M_1' \sqsubseteq_e M_1$ gives $\forall i\colon M_2'(\lfloor cc.pop^i \rfloor) \sqsubseteq M_2(\lfloor cc.pop^i \rfloor)$. Hypothesis $M_1' \sqsubseteq_e M_1$ also gives $M_1'(cc) \sqsubseteq M_1(cc)$, $M_1'(cc.top.A) = M_1(cc.top.A)$, and $M_1'(bc) \sqsubseteq M_1(bc)$, and thus, we get $M_2'(bc) \sqsubseteq M_2(bc)$. From $M_1' \sqsubseteq_e M_1$, we also get $M_1'(cc.top.W) = M_1(cc.top.W)$. Because, for all $w \in W$, we have $M_2'(T(w)) = M_1'(T(w)) \sqcup M_1'(\lfloor cc \rfloor) \sqcup M_1'(bc)$ and $M_2(T(w)) = M_1(T(w)) \sqcup M_1(\lfloor cc \rfloor) \sqcup M_1(bc)$, hypothesis $M_1' \sqsubseteq_e M_1$ gives $M_2'(T(w)) \sqsubseteq M_2(T(w))$. So, $M_2' \sqsubseteq_e M_2$.

2-*Enf* and $E_{\mathsf{H,L}}$ use the same rules for other commands, so $M_2' \sqsubseteq_e M_2$ follows easily. □

Lemma 36. *$E_{\mathsf{H,L}}$ is an enforcer and uses 1-dependent $G_{a:=e}$.*

Proof. It is easy to prove that $E_{\mathsf{H,L}}$ is an enforcer on R and satisfies restrictions (E1), (E2), and (E3) by induction on the operational semantics rules. We omit the details.

We now prove that $E_{\mathsf{H,L}}$ uses 1-dependent $G_{a:=e}$. Consider an assignment $a := e$ in an anchor-tailed command $C; C'$, where C does not involve any assignment to anchor variable and C' is a sequence of assignments to anchor variables. From rule (ASGNA) of $E_{\mathsf{H,L}}$ we get that $G_{a:=e}$ is $M(T(e)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc) \sqsubseteq M(T(a))$. Because $a := e$ is in an anchor-tailed command, we get that $a := e$ is not encapsulated in any conditional command. So, $M(cc) = \epsilon$. Because C does not involve any assignment to anchor variable, bc is $\perp$ when execution reaches C'. While executing C', cc remains ϵ, and thus, from rule (ASGNA) of $E_{\mathsf{H,L}}$, we get that bc remains $\perp$. So, $M(bc) = \perp$. Thus, $G_{a:=e}$ is $M(T(e)) \sqsubseteq M(T(a))$. So, $E_{\mathsf{H,L}}$ uses 1-dependent $G_{a:=e}$. □

BIBLIOGRAPHY

[1] Martin Abadi and Cedric Fournet. Access control based on execution history. In *In Proceedings of the 10th Annual Network and Distributed System Security Symposium*, pages 107–121, 2003.

[2] A. Askarov, S. Chong, and H. Mantel. Hybrid monitors for concurrent noninterference. In *2015 IEEE 28th Computer Security Foundations Symposium*, pages 137–151, July 2015.

[3] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *IEEE Symp. on Security and Privacy*, pages 207–221, 2007.

[4] Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. Cryptographically-masked flows. *Theor. Comput. Sci.*, 402(2-3):82–101, July 2008.

[5] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 113–124, New York, NY, USA, 2009. ACM.

[7] Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 3:1–3:12, New York, NY, USA, 2010. ACM.

[8] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 165–178, New York, NY, USA, 2012. ACM.

[9] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. Faceted execution of policy-agnostic programs. pages 15–26, 2013.

[10] A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-Policies: Formally verified, tag-based security monitors. In *2015 IEEE Symposium on Security and Privacy*, pages 813–830, May 2015.

[11] A. Banerjee, D.A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *IEEE Symp. on Security and Privacy*, pages 339–353, 2008.

[12] Andrew Bedford, Stephen Chong, Jose Desharnais, Elisavet Kozyri, and Nadia Tawbi. A progress-sensitive flow-sensitive inlined information-flow control monitor (extended version). *Computers & Security*, 71:114–131, 2017.

[13] Andrew Bedford, Stephen Chong, Josée Desharnais, and Nadia Tawbi. A progress-sensitive flow-sensitive inlined information-flow control monitor. In *ICT Systems Security and Privacy Protection: 31st IFIP TC 11 International Conference, SEC 2016, Ghent, Belgium, May 30–June 1, 2016, Proceedings*, pages 352–366, Cham, 2016. Springer International Publishing.

[14] E. D. Bell and J. L. La Padula. Secure computer systems: Unified exposition and MULTICS interpretation. Technical Report ESD-TR-75306, Bedford, MA, 1976.

[15] E. D. Bell and J. L. LaPadula. Secure computer systems: Mathematical foundations, 1973.

[16] Lennart Beringer. End-to-end multilevel hybrid information flow control. In *Programming Languages and Systems: 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*, pages 50–65. Springer Berlin Heidelberg, 2012.

[17] Kenneth J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, 1977.

[18] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. Generalizing permissive-upgrade in dynamic information flow analysis. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*, PLAS'14, pages 15:15–15:24, New York, NY, USA, 2014. ACM.

[19] Nataliia Bielova and Tamara Rezk. A taxonomy of information flow monitors. In *Proceedings of the 5th International Conference on Principles of Security and Trust - Volume 9635*, pages 46–67, Berlin, Heidelberg, 2016. Springer-Verlag.

[20] Eleanor Birrell and Fred B. Schneider. A reactive approach to use-based privacy. Technical Report 54843, Cornell University, Computing and Information Science, November 2017.

[21] D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *Proceedings. 1989 IEEE Symposium on Security and Privacy*, pages 206–214, May 1989.

[22] Niklas Broberg, Bart Delft, and David Sands. Paragon for practical programming with information-flow control. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems*, volume 8301, pages 217–232, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[23] Niklas Broberg and David Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 180–196. Springer-Verlag, 2006.

[24] Niklas Broberg and David Sands. Paralocks: Role-based information flow control and beyond. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 431–444, New York, NY, USA, 2010. ACM.

[25] Niklas Broberg, Bart van Delft, and David Sands. The anatomy and facets of dynamic policies. In *IEEE Symp. on Computer Security Foundations (CSF)*, 2015.

[26] Pablo Buiras, Deian Stefan, and Alejandro Russo. On dynamic flow-sensitive floating-label systems. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium*, CSF '14, pages 65–79, Washington, DC, USA, 2014. IEEE Computer Society.

[27] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 289–301, New York, NY, USA, 2015. ACM.

[28] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in aeolus. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[29] S. Chong and A.C. Myers. End-to-end enforcement of erasure and declassification. In *Computer Security Foundations Symposium, 2008. CSF '08. IEEE 21st*, pages 98–111, 2008.

[30] A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 200–214, July 2010.

[31] Andrey Chudnov and David A. Naumann. Inlined information flow monitoring for JavaScript. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 629–643, New York, NY, USA, 2015. ACM.

[32] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *J. Autom. Reason.*, 46(3-4):225–259, April 2011.

[33] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), pages 213–229, New York, NY, USA, 1965. ACM.

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

[35] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[36] Dorothy Elizabeth Robling Denning. *Secure information flow in computer systems.* PhD thesis, West Lafayette, IN, USA, 1975.

[37] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, March 1966.

[38] Dominique Devriese and Frank Piessens. Noninterference through secure

multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 109–124, Washington, DC, USA, 2010. IEEE Computer Society.

[39] D. Dolev and Andrew C. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, Mar 1983.

[40] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 17–30, New York, NY, USA, 2005. ACM.

[41] Eslam Elnikety, Deepak Garg, and Peter Druschel. SHAI: Enforcing Data-Specic Policies with Near-Zero Runtime Overhead. Technical report, Max Planck Institute for Software Systems, Saarland Informatics Campus, Germany, January 2018.

[42] Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 323–335, New York, NY, USA, 2008. ACM.

[43] Joseph A. Goguen and José Mesegue. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–87, 1984.

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

[45] G. Le Guernic. Precise dynamic verification of confidentiality. In *Proceedings of the 5th International Verification Workshop*, pages 82–96, 2008.

[46] Christian Hammer and Gregor 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, 2009.

[47] H. Rex Hartson and David K. Hsiao. Full protection specifications in the semantic model for database protection languages. In *Proceedings of the 1976 Annual Conference*, ACM '76, pages 90–95, New York, NY, USA, 1976. ACM.

[48] D. Hedin, L. Bello, and A. Sabelfeld. Value-sensitive hybrid information flow control for a JavaScript-like language. In *2015 IEEE 28th Computer Security Foundations Symposium*, pages 351–365, July 2015.

[49] Daniel Hedin and Andrei Sabelfeld. Information-flow security for a core of JavaScript. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, pages 3–18, Washington, DC, USA, 2012. IEEE Computer Society.

[50] Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification: High-level policy for a security-typed language. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*, PLAS '06, pages 65–74, New York, NY, USA, 2006. ACM.

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

[52] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 79–90, New York, NY, USA, 2006. ACM.

[53] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. Exploring and enforcing security guarantees via program dependence graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 291–302, New York, NY, USA, 2015. ACM.

[54] Elisavet Kozyri, Owen Arden, Andrew C. Myers, and Fred B. Schneider. JRIF: Java with Reactive Information Flow. Software release, at http://www.cs.cornell.edu/jrif/, February 2016.

[55] Elisavet Kozyri, Owen Arden, Andrew C. Myers, and Fred B. Schneider. JRIF: Reactive Information Flow Control for Java. Technical report, Cornell Univarsity, February 2016.

[56] Elisavet Kozyri, Jose Desharnais, and Nadia Tawbi. Block-safe information flow control. Technical report, Cornell University, 2016.

[57] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control

for standard OS abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 321–334, New York, NY, USA, 2007. ACM.

[58] Peeter Laud. Semantics and program analysis of computationally secure information flow. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, ESOP '01, pages 77–91, London, UK, UK, 2001. Springer-Verlag.

[59] Peeter Laud. On the computational soundness of cryptographically masked flows. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 337–348, New York, NY, USA, 2008. ACM.

[60] Peeter Laud and Varmo Vene. A type system for computationally secure information flow. In *Proceedings of the 15th International Conference on Fundamentals of Computation Theory*, FCT'05, pages 365–377, Berlin, Heidelberg, 2005. Springer-Verlag.

[61] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues*, ASIAN'06, pages 75–89, Berlin, Heidelberg, 2007. Springer-Verlag.

[62] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 158–170, New York, NY, USA, 2005. ACM.

[63] Peng Li and Steve Zdancewic. Practical information-flow control in web-based information systems. In *Proceedings of the 18th IEEE Workshop on Computer Security Foundations*, CSFW '05, pages 2–15, Washington, DC, USA, 2005. IEEE Computer Society.

[64] Luísa Lourenço and Luís Caires. Dependent information flow types. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 317–328, New York, NY, USA, 2015. ACM.

[65] Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld. On-the-fly inlining of dynamic security monitors. *Comput. Secur.*, 31(7):827–843, October 2012.

[66] Kristopher Micinski, Jonathan Fetter-Degges, Jinseong Jeon, Jeffrey S. Foster, and Michael R. Clarkson. Checking interaction-based declassification policies for Android using symbolic execution. In Günther Pernul, Peter Y A Ryan, and Edgar Weippl, editors, *Computer Security – ESORICS 2015*, pages 520–538, Cham, 2015. Springer International Publishing.

[67] S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th*, pages 146–160, 2011.

[68] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif 3.0: Java information flow. Software release. http://www.cs.cornell.edu/jif, July 2006.

[69] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[70] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 129–142, New York, NY, USA, 1997. ACM.

[71] Department of Defense. Department of Defense Trusted Computer System Evaluation Criteria. DoD 5200.28-STD. Supercedes CSC-STD-001-83 dated 15 August 1984, Library Number S225,711.

[72] Department of Defense. Department of Defense Trusted Computer System Evaluation Criteria. CSC-STD-001-83, Library Number S225,711, August 1983.

[73] Dorothy Elizabeth Robling Denning. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

[74] B.P.S. Rocha, S. Bandhakavi, J. den Hartog, W.H. Winsborough, and S. Etalle. Towards static flow-based declassification for legacy and untrusted programs. In *IEEE Symp. on Security and Privacy*, pages 93–108, 2010.

[75] B.P.S. Rocha, M. Conti, S. Etalle, and B. Crispo. Hybrid static-runtime information flow and declassification enforcement. *Information Forensics and Security, IEEE Transactions on*, 8(8):1294–1305, 2013.

[76] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 63–74, New York, NY, USA, 2009. ACM.

[77] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 186–199, Washington, DC, USA, 2010. IEEE Computer Society.

[78] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[79] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *In Proc. International Symp. on Software Security (ISSS03), volume 3233 of LNCS*, pages 174–191. Springer-Verlag, 2004.

[80] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proceedings of the 7th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics*, PSI'09, pages 352–365, Berlin, Heidelberg, 2010. Springer-Verlag.

[81] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17(5):517–548, October 2009.

[82] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, February 1996.

[83] José Fragoso Santos and Tamara Rezk. An information flow monitor-inlining compiler for securing a core of JavaScript. In *ICT Systems Security and Privacy Protection: 29th IFIP TC 11 International Conference, SEC 2014, Marrakech, Morocco, June 2-4, 2014. Proceedings*, pages 278–292. Springer Berlin Heidelberg, 2014.

[84] Fred B. Schneider, Kevin Walsh, and Emin Gün Sirer. Nexus Authorization Logic (NAL): Design rationale and applications. *ACM Trans. Inf. Syst. Secur.*, 14(1):8:1–8:28, June 2011.

[85] Paritosh Shroff, Scott Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE*

Computer Security Foundations Symposium, CSF '07, pages 203–217, Washington, DC, USA, 2007. IEEE Computer Society.

[86] Geoffrey Smith and Rafael Alpízar. Secure information flow with random assignment and encryption. In *Proceedings of the Fourth ACM Workshop on Formal Methods in Security*, FMSE '06, pages 33–44, New York, NY, USA, 2006. ACM.

[87] Deian Stefan, David Mazières, John C. Mitchell, and Alejandro Russo. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming*, 27:e5, 2017.

[88] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 95–106, New York, NY, USA, 2011. ACM.

[89] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, January 1986.

[90] Panagiotis Vasilikos, Flemming Nielson, and Hanne Riis Nielson. Secure information release in timed automata. In *Principles of Security and Trust*, pages 28–52, Cham, 2018. Springer International Publishing.

[91] D. Volpano. Secure introduction of one-way functions. In *Computer Security Foundations Workshop, 2000. CSFW-13. Proceedings. 13th IEEE*, pages 246–254, 2000.

[92] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proceedings 10th Computer Security Foundations Workshop*, pages 156–168, Jun 1997.

[93] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996.

[94] Dennis Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 268–276, New York, NY, USA, 2000. ACM.

[95] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *Proceedings of the 7th International Joint Conference CAAP/-FASE on Theory and Practice of Software Development*, TAPSOFT '97, pages 607–621, London, UK, UK, 1997. Springer-Verlag.

[96] Dan S. Wallach, Jim A. Roskind, and Edward W. Felten. Flexible, extensible Java security using digital signatures. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 38:59–74, December 1996.

[97] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, June 2016. ACM Press.

[98] Stephan Arthur Zdancewic. *Programming Languages for Information Security*. PhD thesis, Ithaca, NY, USA, 2002. AAI3063751.

[99] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[100] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2):67–84, Mar 2007.