2019 IEEE 32nd Computer Security Foundations Symposium (CSF)

Beyond Labels: Permissiveness for Dynamic Information Flow Enforcement

Elisavet Kozyri, Fred B. Schneider
Cornell University
Ithaca, USA
{ekozyri,fbs}@cs.cornell.edu

Andrew Bedford, Josée Desharnais and Nadia Tawbi
Laval University
Quebec City, Canada
{andrew.bedford.1@, josee.desharnais@ift., nadia.tawbi@ift.}ulaval.ca

***Abstract*—Flow-sensitive labels used by dynamic enforcement mechanisms might themselves encode sensitive information, which can leak. Metalabels, employed to represent the sensitivity of labels, exhibit the same problem. This paper derives a new family of *enforcers*—k-Enf, for $2 \leq k \leq \infty$—that uses *label chains*, where each label defines the sensitivity of its predecessor. These enforcers satisfy *Block-safe Noninterference* (BNI), which proscribes leaks from observing variables, label chains, and blocked executions. Theorems in this paper characterize where longer label chains can improve the *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 lattice size.**

***Index Terms*—label chain, dynamic information flow control, permissiveness**

I. Introduction

Dynamic enforcement mechanisms (which might involve static analysis) for information flow control employ tags containing labels to represent the sensitivity[1] of what variables store. These labels can be *flow-sensitive*, meaning that they change when a value with different sensitivity is assigned to the tagged variable during program execution. Sensitive information might influence which assignments execute and, consequently, determine how and when the flow-sensitive label tagging a variable changes. So flow-sensitive labels can depend on sensitive information.

Inspecting or directly observing flow-sensitive labels might itself leak sensitive information [20]. Consider a program

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

Suppose w is tagged with a flow-sensitive label, but the other variables, are tagged with *fixed* labels[2], which do not change during execution: 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.

Kozyri and Schneider are supported in part by AFOSR grant F9550-16-0250 and NSF grant 1642120. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government. Bedford, Desharnais, and Tawbi are supported by NSERC grants RGPIN-239294-2012 and RGPIN-04461-2015.

[1]In this paper, *sensitivity* refers to *confidentiality level*.

[2]Note that variables with fixed labels can model *sources* or *sinks* of information.

(i) If $m > 0$ holds, then information flows *explicitly* from h to w and *implicitly* from m (in $m > 0$) to w. When (1) 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 w should be tagged with flow-sensitive label M when (1) terminates, because M is at least as restrictive as the labels that tag l and m.

So, the flow-sensitive label tagging w depends on whether $m > 0$ holds. Information about m, which is sensitive, leaks to observers that can learn that label.

Blocking an execution based on flow-sensitive labels might leak sensitive information, too. Consider (1), extended with two assignments:

$$\begin{array}{l}\textbf{if } m > 0 \textbf{ then } w := h \textbf{ else } w := l \textbf{ end};\\ m := w;\ l := 1\end{array} \tag{2}$$

(i) If $m > 0$ holds, then $m := w$ should be blocked to prevent information tagged H and stored in w from flowing to m, which is tagged with fixed label 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 $l := 1$ will 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 if observers can detect that $m := w$ is blocked.[3]

To prevent such leaks, *metalabels* (e.g., [6]) might be introduced to represent the sensitivity of information encoded in flow-sensitive labels. For example, the metalabel for w in (2) would be M, corresponding to the sensitivity of information 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

[3]In fact, an arbitrary number of bits can be leaked through blocking executions [2].

DOI 10.1109/CSF.2019.00031

flow of that information (which is tagged with M) to variable l (tagged L), assignment $l := 1$ must not be executed.

Since metalabels are flow-sensitive, they too could encode sensitive information that might leak 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.

This paper introduces and analyzes dynamic enforcement mechanisms that employ label chains of arbitrary length. These mechanisms protect against a threat model where principals observe updates to variables and to elements of label chains—attackers thus co-resident with the program being executed.

We start by formalizing label chains (§II) and defining enforcers (§III). We next (§IV) extend block-safe noninterference (BNI) [22] to stipulate that sensitive information does not leak to observers of variables and label chains. BNI extends *termination insensitive noninterference* (TINI) [38] in order to proscribe leaks to principals that can observe variables and label chains along normally terminated and blocked traces. Enforcer ∞-*Enf* is derived (§V); it uses label chains of infinite length to enforce BNI. A family k-*Enf* of enforcers use finite label chains to approximate the infinite label chains of ∞-*Enf*. Our k-*Enf* enforcers also are shown (§VI) to satisfy BNI.

A loss of *permissiveness* could result when shorter label chains approximate longer ones. In particular, with a shorter label chain, execution of some program might be blocked sooner or fewer principals might be allowed to observe elements in a label chain—either brings a loss of permissiveness. This paper formally characterizes the relationship between permissiveness and storage overhead of label chains having different lengths. We present theorems that relate label chain length and permissiveness for k-*Enf* enforcers (§VII) as well as for other enforcers (§VIII) that satisfy BNI. The relationships between permissiveness and storage overhead depend on initialization, threat model, size of the lattice, as well as, on certain semantic attributes of enforcement mechanisms: k-*precise*, k-*varying*, and k-*dependent*.

Our theorems show that approximating longer label chains with shorter ones can harm permissiveness. Specifically, we show:

- For k-*Enf* enforcers, if flexible variables initially store information associated with given label chains, then approximating these label chains with shorter ones causes fewer principals to read chain elements, leading to a permissiveness loss.
- For other enforcers, if flexible variables initially store no information, then the generated label chains cannot be shortened without a permissiveness loss; an example in §VIII illustrates.
- An enforcer that uses only one label for each variable blocks some executions sooner than an enforcer that uses two labels for each variable. This blocking harms permissiveness: principals will make fewer observations in the blocked execution. But, when labels are drawn from a 2-level lattice, associating variables with only one label does not harm permissiveness.

In summary, we are identifying conditions under which longer label chains are useful. Moreover, our conditions apply even if labels are not first-class entities in the provided programming language but instead are internal to an enforcement mechanism.

II. Label Chains

Each variable x in a program will be 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 label ℓ_{i+1} specifies sensitivity for ℓ_i. Of course, actual implementation of label chains may only use finite space. Labels come from a possibly infinite *underlying* lattice $\mathcal{L} = \langle L, \sqsubseteq, \sqcup \rangle$ with bottom element $\bot$. For[4] $\ell, \ell' \in \mathcal{L}$, 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 to data tagged with ℓ'.

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

Unless a label chain $\langle \ldots, \ell_i, \ell_{i+1}, \ldots \rangle$ is *monotonically decreasing*—$\ell_{i+1} \sqsubseteq \ell_i$ for $i \geq 1$— then sensitive information can be leaked. Here is why. Consider a variable x having 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 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 are implemented by sequencing individual labels stored in a memory M. Domain $dom(M)$ of a memory M includes:

- *Variables* that store (say) integers ($\nu \in \mathbb{Z}$). Lower case letters (e.g., a, w, x, h, m, l) denote variables. $M(x)$ is the integer stored in variable x by M. Let Var be 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 $T(T^i(x))$, for $i \geq 1$. We say *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

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

not an auxiliary). For convenience, we define $T^0(q) \triangleq q$ 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:

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

The mappings defined by M and T^i extend from identifiers to expressions (of variables or tags) $e \oplus e'$ in the usual way:

$$M(e \oplus e') \triangleq M(e) \oplus M(e') \tag{3}$$

$$T^i(e \oplus e') \triangleq T^i(e) \sqcup T^i(e'), \text{ for } i \geq 1. \tag{4}$$

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 a value is assigned to w. So, the label chain of flexible variable w is flow-sensitive. For an *anchor* variable a, the label stored in $T(a)$ remains fixed throughout execution, and the remaining elements of the label chain satisfy:

$$M(T^i(a)) = \bot \text{ for any } T^i(a) \in dom(M) \text{ with } i > 1. \tag{5}$$

This form of chain is sensible for an anchor variable 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). A constant ν is a special case of an anchor variable:

$$M(T^i(\nu)) = \bot, \text{ for any } T^i(\nu) \in dom(M) \text{ with } i \geq 1. \tag{6}$$

III. Enforcers

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

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

with $C_1 = C$. 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 employed to formally define traces. This paper uses a **while**-language (Figure 1) with operational semantics rules R (Figure 2). Notice, 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 If1, If2, Wl1, and Wl2 marks the end of *conditional commands* (similar to [31], [33]). When execution of the corresponding taken branch completes, rule Exit is triggered.[5] Notice that C_i in a state $\langle C_i, M_i\rangle$ can be a command C as defined in Figure 1, a termination delimiter such as **stop**, or a command involving a conditional delimiter **exit**.

(Constants)	ν	$\in$	$\mathbb{Z}$
(Anchor variables)	a, x	$\in$	Var_A
(Flexible variables)	w, x	$\in$	Var_F
(Expressions)	e	$::=$	$\nu \mid x \mid e_1 \oplus e_2$
(Commands)	C	$::=$	$\mathbf{skip} \mid x := e \mid C_1; C_2 \mid$ $\mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{end} \mid$ $\mathbf{while}\ e\ \mathbf{do}\ C\ \mathbf{end}$

Fig. 1. Syntax

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

$$\text{(AsgnA)}\ \frac{\nu = M(e)}{\langle a := e, M\rangle \rightarrow \langle \mathbf{stop}, M[a \mapsto \nu]\rangle}$$

$$\text{(AsgnF)}\ \frac{\nu = M(e)}{\langle w := e, M\rangle \rightarrow \langle \mathbf{stop}, M[w \mapsto \nu]\rangle}$$

$$\text{(If1)}\ \frac{M(e) \neq 0}{\langle \mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{end}, M\rangle \rightarrow \langle C_1; \mathbf{exit}, M\rangle}$$

$$\text{(If2)}\ \frac{M(e) = 0}{\langle \mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{end}, M\rangle \rightarrow \langle C_2; \mathbf{exit}, M\rangle}$$

$$\text{(Wl1)}\ \frac{M(e) \neq 0}{\langle \mathbf{while}\ e\ \mathbf{do}\ C\ \mathbf{end}, M\rangle \rightarrow \langle C; \mathbf{while}\ e\ \mathbf{do}\ C\ \mathbf{end}; \mathbf{exit}, M\rangle}$$

$$\text{(Wl2)}\ \frac{M(e) = 0}{\langle \mathbf{while}\ e\ \mathbf{do}\ C\ \mathbf{end}, M\rangle \rightarrow \langle \mathbf{exit}, M\rangle}$$

$$\text{(Exit)}\ \frac{}{\langle \mathbf{exit}, M\rangle \rightarrow \langle \mathbf{stop}, M\rangle}$$

$$\text{(Seq1)}\ \frac{\langle C_1, M\rangle \rightarrow \langle \mathbf{stop}, M'\rangle}{\langle C_1; C_2, M\rangle \rightarrow \langle C_2, M'\rangle}$$

$$\text{(Seq2)}\ \frac{\langle C_1, M\rangle \rightarrow \langle C_1', M'\rangle \qquad C_1' \neq \mathbf{stop}}{\langle C_1; C_2, M\rangle \rightarrow \langle C_1'; C_2, M'\rangle}$$

Fig. 2. Structural Operational Semantics R

The rules comprising R define a function $trace_R(C, M)$ that maps a 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 as follows.

Definition 1 (Healthy memory for C)**.** *Define*

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

where $x \in C$ indicates that $x \in Var$ appears in C.

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

[5] 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.

Executing command C on memory M under the auspices of an *enforcer* E leads to a trace $\tau = trace_E(C, M)$. Memories in states of $trace_E(C, M)$ are expected to store, among other identifiers, metadata employed by E. In particular, these memories should store label chains of size $n_E \geq 1$ and a set Aux_E of auxiliaries, as characterized by:

Definition 2 (Healthy memory for E, $\mathcal{L}$, and C)**.** *A memory M is* healthy *for enforcer E, lattice $\mathcal{L}$, and command C denoted by $M \models \mathcal{H}(E, \mathcal{L}, C)$, iff*

- $M \models \mathcal{H}(C)$,
- *for each variable x, $dom(M)$ includes exactly n_E tags comprising a label chain, where all tags are mapped to lattice $\mathcal{L}$:*

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

- *$dom(M)$ contains requisite auxiliaries Aux_E:*

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

- *flexible variables in M have monotonically decreasing label chains,*
- *anchor variables in M have label chains satisfying (5), and*
- *constants in M have label chains satisfying (6).*

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

An enforcer is also accompanied by mapping $Init_E$ from auxiliaries in Aux_E to initial values. Initial memories should satisfy this mapping.

Definition 3 (Initially healthy memory for E, $\mathcal{L}$, and C)**.** *A memory M is defined to be* initially healthy *for enforcer E, lattice $\mathcal{L}$, and command 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 expect that traces $trace_E(C, M)$ will satisfy some policy P of interest, where E blocks traces if needed to satisfy P. 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 *trace prefix* of $trace_R(C, M)$.

Definition 4 (Trace prefix)**.**

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

where $blk(\tau)$ holds iff τ ends with a blocked state.

We now can give the formal definition for the enforcers E being considered in this paper.

Definition 5 (Enforcer)**.** $E \triangleq \langle trace_E, n_E, Aux_E, Init_E \rangle$ *is an* enforcer on R *if $trace_E$ satisfies the following reasonable conditions:*

- **(E0)** $(\forall C, M\colon trace_E(C, M) \preceq trace_R(C, M))$
- **(E1)** *Trace $trace_E(C, M)$ is defined when $M \models \mathcal{H}_0(E, \mathcal{L}, C)$ holds.*
- **(E2)** *For a memory M_i in any state of $trace_E(C, M)$, condition $M_i \models \mathcal{H}(E, \mathcal{L}, C)$ 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.*[6]

We say that E is an enforcer on R for P, if the image of function $trace_E$, which is a set of traces, satisfies P.

IV. Threat Models and BNI

a) Observations: Our threat model has principals observing updates to identifiers. When an assignment to a flexible variable w is executed, each element in set $O(w) \triangleq \{w, T(w), \ldots, T^i(w), \ldots\}$ is updated. When an assignment to an anchor variable a is executed, only $O(a) \triangleq \{a\}$ 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 [5].

Principals do not observe updates to an identifier q when $T(q) \notin dom(M)$ holds, because q then is 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 now formalize the *observation* available to a principal p assigned label ℓ when an assignment executes. Define the projection $M|_\ell^S$ 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 an assignment to a variable x is performed and memory M results, then observation $M|_\ell^{O(x)}$ is generated to p. Notice, $M|_\ell^{O(x)}$ can be empty.

A *sequence of observations* is generated along with a trace when assignments are performed.

[6]More formally, if $\langle C_1; C_2, M_1 \rangle \xrightarrow{*} \langle C_1'; C_2, M_2 \rangle$ is a subtrace of $trace_E(C, M)$, where C_1 is a subcommand of C, and if "$w := e$"$\notin C_1$ holds for a flexible variable w, then the following should hold:

$$(\forall i \geq 1\colon T^i(w) \in dom(M_1) \Rightarrow M_1(T^i(w)) = M_2(T^i(w)))$$

Note, a conditional delimiter (e.g., **exit**) is not considered a subcommand of C.

Definition 6 (Sequence of observations)**.** *Given a trace τ, define $\tau|_\ell^S$ to be sequence $\theta = \Theta_1 \rightharpoonup \ldots \rightharpoonup \Theta_n$ of observations involving identifiers in set S and having sensitivity at most ℓ:*

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

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

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

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

This *strong threat model* produces observations for both variables and tags. But observations are not generated 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). This is because enforcers may differ about whether these other updates are performed. And those differences would make comparison of observations (employed in §VII) problematic.

b) Block-safe Noninterference: Block-safe Noninterference (BNI) is a form of *noninterference* [16] that incorporates observations on tags and considers all finite traces—normally terminated and blocked by the enforcement mechanism. Formally, BNI stipulates that if two finite traces 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. We define k-BNI for $k \geq 0$ for settings where observations are limited to variables and tags $T^0, T^1, \ldots, T^k$. Note $M|_\ell$ abbreviates $M|_\ell^{dom(M)}$.

Definition 7 (k-BNI)**.**

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

If k-BNI$(E, \mathcal{L}, C)$ holds for every C, then E enforces k-BNI$(\mathcal{L})$.[7] If for all $k \geq 0$ and $\mathcal{L}$, enforcer E satisfies k-BNI$(\mathcal{L})$, then we say that E enforces BNI. Notice that, by definition, k-BNI ignores observations generated by infinite traces. k-BNI could be strengthened to handle such observations by in addition requiring that $\tau|_\ell^k$ in Definition 7 be a prefix of $\tau'|_\ell^k$ when τ' is infinite. Such a strengthening of Definition 7 does not affect the theory presented in this paper.

0-BNI is stronger than TINI enforced by [3], [10], [12], [14], [26]. TINI concerns normally terminated executions but does not consider finite traces that correspond to blocked executions. So TINI ignores traces that become blocked by the enforcement mechanism and thereby leak sensitive information. 0-BNI considers all finite traces. So, an enforcement mechanism that satisfies 0-BNI will satisfy TINI, too. But, an enforcement mechanism that satisfies TINI might not satisfy 0-BNI. An example is the enforcement mechanism that executes program (2) as outlined by (i) and (ii) given below that example in §I. This mechanism satisfies TINI, but it does not satisfy 0-BNI because the value of m is leaked. Notice that 0-BNI is equivalent to TINI (extended with observations along traces), when no enforcer is being employed during program execution (i.e., E is the trivial enforcer that accepts all commands).

0-BNI is weaker than the natural extension of *termination sensitive noninterference* (TSNI) [37] for generating observations throughout traces. TSNI considers infinite and finite traces (terminated normally as well as blocked), but because 0-BNI ignores infinite traces, 0-BNI allows leaks through termination channels that already exist in a program (due to non-terminating while-loops) [15].

We chose to study 0-BNI, so we could focus on leaks introduced by the enforcer itself. The enforcement techniques of the next section prevent those leaks. Moreover, they can be extended to enforce TSNI (e.g., would leak through non-terminating while-loops) using techniques similar to those given in [6].

V. Enforcer ∞-*Enf*

We use familiar insights about information flow to formulate an enforcer ∞-*Enf* that uses infinite label chains (i.e., $n_{\infty\text{-}Enf} = \infty$) to enforce BNI for programs written in the programming language of Figure 1. We later derive from ∞-*Enf* the k-*Enf* family of enforcers that use finite label chains.

A. Updating Label Chains of Flexible Variables

When assignment $w := e$ 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 when executing $w := e$, enforcer ∞-*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 that C should be reached. If C appears in the body of a conditional command having guard e, then e belongs to the context of C. For example, consider:

$$\textbf{if } x > 0 \textbf{ then } w := w' \textbf{ else } w := w'' \textbf{ end} \qquad (7)$$

Here, context ctx of $w := w'$ and $w := w''$ is $\{x > 0\}$. Notice, if $T^i(w') \neq T^i(w'')$ holds prior to (7) for some $i \geq 0$, then the value in $T^i(w)$ after the **if** command depends on ctx.

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

Context ctx is prevented from leaking through $T^i(w)$ if we require that $T(ctx) \sqsubseteq T^{i+1}(w)$ holds, where $T(ctx)$ is the sensitivity of ctx.

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

$$\begin{aligned} UT(q, e, ctx) \triangleq\ & T(q) := T(e) \sqcup T(ctx); \\ & UT(T(q), T(e) \sqcup T(ctx), ctx) \end{aligned}$$

For $w := e$ in context ctx, $UT(w, e, ctx)$ expands to[8]

$$\forall i \geq 1:\ T^i(w) := T^i(e) \sqcup T(ctx). \tag{8}$$

Here, universal quantifier $\forall$ denotes simultaneous update of infinitely many identifiers. So, enforcer ∞-*Enf* will produce a new label chain for w; each label in that chain is computed according to (8).

B. Preventing Leaks through Anchor Variables

Prior to executing $a := e$ for an anchor variable a, an enforcer checks a *block condition* $G_{a:=e}$. If $G_{a:=e}$ 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 blocks.

But blocking execution might cause implicit flow of sensitive information, as seen with (2). We avoid this flow by generalizing the definition of ctx to include block conditions that could have already been checked. 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.

We now show how to construct $G_{a:=e}$ for an assignment $a := e$ in context ctx. 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 $T(ctx \cup \{G_{a:=e}\})$. We thus require $T(ctx) \sqcup T(G_{a:=e}) \sqsubseteq T(a)$. So, for $G_{a:=e}$ to hold, both $T(e) \sqsubseteq T(a)$ and $T(ctx) \sqcup T(G_{a:=e}) \sqsubseteq T(a)$ should 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

$$G_{a:=e} \ \Rightarrow\ (T(e) \sqcup T(ctx) \sqcup T(G_{a:=e}) \sqsubseteq T(a)). \tag{9}$$

One possible solution for $G_{a:=e}$ in (9) is:

$$G_{a:=e} \triangleq (T(e) \sqcup T(ctx) \sqsubseteq T(a)). \tag{10}$$

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

To verify that (10) is a solution, first compute sensitivity:

$$\begin{aligned} T(G_{a:=e}) &= T(\ T(e) \sqcup T(ctx) \sqsubseteq T(a)\ ) \\ &= T^2(e) \sqcup T^2(ctx) \sqcup T^2(a) && \{\text{due to (4)}\} \\ &= T^2(e) \sqcup T^2(ctx) \sqcup \bot && \{T^2(a) = \bot\} \\ &= T^2(e) \sqcup T^2(ctx) && \{\ell \sqcup \bot = \ell\} \end{aligned} \tag{11}$$

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 (9), and noticing that $T^2(e) \sqcup T^2(ctx) \sqsubseteq T(e) \sqcup T(ctx)$ (due to monotonically decreasing label chains), equation (9) becomes equivalent to a true statement, which is what we needed to verify solution (10).

$G_{a:=e}$ in (10) is used by all dynamic flow sensitive enforcement mechanisms we know. But, we seem to be the first to present it as a solution of (9).

C. Operational Semantics for ∞-Enf

Enforcer ∞-*Enf* uses (i) UT (see (8)) for deducing label chains and (ii) $G_{a:=e}$ (see (10)) for blocking possibly unsafe assignments. UT and $G_{a:=e}$ mention tags for variables and sensitivity $T(ctx)$ of the context but do not need ctx, $T^2(ctx)$, $T^3(ctx)$, etc. $T(ctx)$ is the join of the sensitivity of each guard and each block condition that determines the reachability of a command. ∞-*Enf* uses auxiliaries to maintain $T(ctx)$:

- cc (conditional context) keeps track of the sensitivity of the guards in all conditional commands that encapsulate the next command to be executed, and
- bc (blocking context) keeps track of the sensitivity of information revealed by block conditions that might influence reachability of the next command 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 already been evaluated. 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 $T(ctx)$:

$$bc := T(G_{a:=e}) \sqcup T(ctx). \tag{12}$$

From (11) and monotonicity of label chains (i.e., $T^2(ctx) \sqsubseteq T(ctx)$), we then get

$$bc := T^2(e) \sqcup T(ctx) \tag{13}$$

which is equivalent to (12). No block condition has been evaluated before execution starts, so 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, the sensitivity of the corresponding guard is pushed onto cc; upon exit the top element of cc is popped. $\lfloor cc \rfloor$ will denote the join of all labels in cc. At the beginning of execution, no conditional command has been entered, so cc is initialized to the empty stack ϵ with $\lfloor \epsilon \rfloor \triangleq \bot$. So, we have $Init_{\infty\text{-}Enf}(cc) = \epsilon$.

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

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

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

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

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

Fig. 3. Operational semantics for **skip** and assignments.

Putting all together, sensitivity $T(ctx)$ is $\lfloor cc \rfloor \sqcup bc$. Substituting $\lfloor cc \rfloor \sqcup bc$ for $T(ctx)$ in (10), block condition $G_{a:=e}$ becomes:

$$T(e) \sqcup \lfloor cc \rfloor \sqcup bc \sqsubseteq T(a). \tag{14}$$

Substituting $\lfloor cc \rfloor \sqcup bc$ for $T(ctx)$ in (13), the update of bc becomes:

$$bc := T^2(e) \sqcup \lfloor cc \rfloor \sqcup bc. \tag{15}$$

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

Rule ASGNA in Figure 3 uses (14) and (15). 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 dictates which principals are allowed to learn why an execution ended (i.e., due to a **block** versus due to a **stop**) without sensitive information leaking.

In Figure 3, Rule ASGNF for assignment $w := e$ to flexible variable w implements (8), given $T(ctx) = \lfloor cc \rfloor \sqcup bc$. So, the label chain of w is updated as follows:

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

Rules for conditional commands are given in Figure 4. They adopt techniques employed by other dynamic enforcement mechanisms (e.g., [14]) 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 (writing $M(cc).push(\langle \ell, W, A\rangle)$); when execution exits C, tuple $\langle \ell, W, A\rangle$ is popped. Here we define the elements of tuple $\langle \ell, W, A\rangle$.

- Element ℓ is the sensitivity of the guard e of conditional command 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 set $targetFlex(C_u)$ of target flexible variables in untaken branch C_u of C. If $w \in W$, then $T^i(w)$ for $i \geq 0$ could have been updated if 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 set $targetAnchor(C_u)$ of all anchor variables in 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, 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, 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).

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

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

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

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

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

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

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

Fig. 4. Operational semantics for conditional commands.

Figure 5 gives rules for executing sequences of commands. Rule SEQF asserts that execution stops once an assignment is blocked.

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 3, 4, and 5. We prove the following Theorem in [23].

Theorem 1. *∞-Enf is an enforcer on R for BNI.*

Here is how ∞-*Enf* handles program (2).

- If $m > 0$ holds, then rule IF1 is invoked. Having $T(m > 0) = \mathsf{M}$, $W = \{w\}$, and $A = \emptyset$, causes triple

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

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

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

Fig. 5. Operational semantics for sequences

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

$$\begin{array}{l} U(M, W) \triangleq \\ \quad M[\ \forall w \in W\text{: } \forall i\text{:} 1 \le i \le k\text{:} \\ \qquad T^i(w) \mapsto M(T^i(w)) \sqcup M(\lfloor cc \rfloor) \sqcup M(bc)]. \end{array}$$

Fig. 6. Modified rules for k-*Enf*

$\langle \mathsf{M},\ \{w\},\ \emptyset\rangle$ to be pushed onto conditional context cc, which was empty. To execute taken branch $w := h$, rule ASGNF is invoked, which causes w to be associated with label chain $\langle \mathsf{H}, \mathsf{M}, \mathsf{M}, \ldots\rangle$. Before execution exits the **if**-statement, rule EXIT is invoked, leaving the label chain of w unchanged and restoring cc to the empty stack. Rule ASGNAFAIL is then invoked for assignment $m := w$, because $T(m)$ is not as restrictive as $T(w)$. So, execution blocks without assigning w to m.

– If $m > 0$ does not hold, then w is associated with label chain $\langle \mathsf{M}, \mathsf{M}, \mathsf{M}, \ldots\rangle$ at the end of the **if**-statement. Rule ASGNA is then invoked for assignment $m := w$, which sets blocking context bc to M. Because $bc = \mathsf{M}$ holds, ASGNAFAIL is invoked for assignment $l := 1$, and thus, execution blocks before performing the update.

Notice that principals assigned label L do not observe any updates to variables and elements of label chains. So, the leak of m (as described in §I) is prevented when (2) is executed with ∞-*Enf*.

VI. Enforcer k-*Enf*

An enforcer that uses infinite label chains cannot always be implemented with finite memory. But an infinite label chain can be approximated by a finite label chain. First notice that 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 $i \ge k$, the i^{th} element in Ω' is at least as restrictive as the corresponding element in Ω due to monotonically decreasing label chains: $\ell_{k+1} \sqsubseteq \ell_k$, $\ell_{k+2} \sqsubseteq \ell_k$, etc. Finite label chain with $m \ge 0$:

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

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 enforcer k-*Enf* from ∞-*Enf*. Enforcer k-*Enf* uses the operational semantics rules of ∞-*Enf* to compute up to the k^{th} tag. Because rule ASGNA mentions $T^2(x)$, we require $k \ge 2$. In ∞-*Enf*, only ASGNF and function U refers 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. See Figure 6 for the revised rule.

Enforcer k-*Enf* generates observations for updates 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. 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 in [23].

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

VII. Permissiveness of k-*Enf* versus Chain Length

Approximation by shorter label chains has a penalty: permissiveness. The details however are not straightforward. For k-*Enf* enforcers, the penalty of shorter label chains will depend on the threat model and on assumptions about initialization. This section gives theorems to characterize that trade-off. And in the next section, we examine other classes of enforcers.

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 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'. To formalize this, we define $\theta \trianglelefteq \theta'$:

$$\begin{array}{l} \theta \trianglelefteq \theta' \triangleq \\ \quad |\theta| \le |\theta'|\ \wedge\ (\forall i\text{: } 1 \le i \le |\theta|\text{: } dom(\theta[i]) \subseteq dom(\theta'[i])) \end{array}$$

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

We compare permissiveness of enforcers relative to an underlying lattice and some identifiers of interest. We start the comparison with pairs of memories that satisfy an initialization condition, such as equality on initial values and label chains.

Definition 8 (At least as permissive as)**.** *Define an enforcer E' to be* 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 $k \geq 0$:*

$$\begin{aligned} E \leq_{\rho}^{k,\mathcal{L}} E' \triangleq\ & \\ & \forall C, M, M' \colon \rho(M, M') \\ & \quad \wedge M \models \mathcal{H}_0(E, \mathcal{L}, C) \wedge M' \models \mathcal{H}_0(E', \mathcal{L}, C) \\ & \quad \Rightarrow (\forall \ell \in \mathcal{L} \colon \mathit{trace}_E(C, M)|_\ell^k \trianglelefteq \mathit{trace}_{E'}(C, M')|_\ell^k) \end{aligned} \tag{16}$$

Notice, the consequent in definition (16) 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 (16) we can prove that if $\rho \Rightarrow \rho'$, then

$$E \leq_{\rho'}^{k,\mathcal{L}} E' \Rightarrow E \leq_{\rho}^{k,\mathcal{L}} E'. \tag{17}$$

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

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

The desired initialization condition then is:

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

Thus, initialization condition ρ_k allows a flexible variable w to be initially associated with label chains, where $\ell_{k+1} \sqsubset \ell_k$:

- $\Omega = \langle \ell_1, \ell_2, \ldots, \ell_{k-1}, \ell_k, \ell_{k+1}, \ell_{k+1}\rangle$ by $(k+1)$-*Enf*,
- $\Omega' = \langle \ell_1, \ell_2, \ldots, \ell_{k-1}, \ell_k, \ell_k\rangle$ by k-*Enf*.

We say that Ω exhibits a $(k+1)$-*decrease* because $T^{k+1}(w) \sqsubset T^k(w)$. Notice that for a label chain to exhibit a $(k+1)$-decrease, the labels should belong to a lattice with at least one non-bottom element. Here, Ω' is a conservative approximation of Ω.

Consequently, 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, as we prove in [23], $(k+1)$-*Enf* is strictly more permissive than k-*Enf*.

Theorem 3. *k-Enf $<_{\rho_k}^{k,\mathcal{L}}$ $(k+1)$-Enf, for $k \geq 2$ and any lattice $\mathcal{L}$ with at least one non-bottom element.*

Thus, longer label chains offer increased permissiveness for the k-*Enf* family of enforcers, because they allow more principals to observe elements of these label chains. Moreover, we conclude by transitivity that k-*Enf* $<_{\rho_k}^{k,\mathcal{L}}$ ∞-*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 for set Var_F of flexible variables

$$\begin{aligned} \alpha_c(M) \triangleq \forall w \in \mathit{Var}_F \colon \forall i \geq 1 \colon \\ T^i(w) \in \mathit{dom}(M) \Rightarrow M(T^i(w)) = \bot. \end{aligned}$$

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 3 does not hold when $<_{\rho_k}^{k,\mathcal{L}}$ is replaced with $<_{c}^{k,\mathcal{L}}$. With initialization condition c, label chains longer than two elements do not enhance the permissiveness of k-*Enf*. This is because, for initialization condition c, enforcer k-*Enf* produces label chains where the second element is always repeated[9] (e.g., $\langle \mathsf{H}, \mathsf{M}, \mathsf{M}, \ldots, \mathsf{M}\rangle$) due to the conservative update of label chains of flexible variable induced by rules ASGNF in Figure 3 and EXIT in Figure 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). We prove the following theorem in [23].

Theorem 4. *k-Enf $\cong_{c}^{k,\mathcal{L}}$ $(k+1)$-Enf for any lattice $\mathcal{L}$ and $k \geq 2$.*

Threat model specifics affect the permissiveness of longer label chains, too. Consider a *weakened threat model* that allows observations of updates to variables but not to tags. This model characterizes attackers that are co-resident with program execution and can access the memory modified by the target program. Label chains here are assumed to be stored in a protected memory that only the enforcer can access. Enforcers here would be expected to satisfy 0-BNI. Enforcer k-*Enf* satisfies k-BNI. So, k-*Enf* satisfies 0-BNI, because 0-BNI is implied by k-BNI.

Under the weakened threat model, permissiveness of our enforcers is compared using relation $\leq_{\rho_k}^{0,\mathcal{L}}$, where superscript 0 indicates that only observations involving variables are considered for the comparison. Theorem 3 does not apply, because relation $<_{\rho_k}^{k,\mathcal{L}}$ considers observations up to the kth tag (due to superscript k) where $k \geq 2$. But we do have the following Theorem, which is proved in [23].

Theorem 5. *k-Enf $\cong_{\rho_k}^{0,\mathcal{L}}$ $(k+1)$-Enf for any lattice $\mathcal{L}$ and $k \geq 2$.*

Because $c \Rightarrow \rho_k$ holds, property (17) and Theorem 5 gives k-*Enf* $\cong_{c}^{0,\mathcal{L}}$ $(k+1)$-*Enf* for any lattice $\mathcal{L}$ and $k \geq 2$. So, under the weakened threat model and for both initialization conditions (i.e, ρ_k and c), the permissiveness of k-*Enf* does not improve by using label chains of length greater than two.

[9] See Lemma 11 in [23].

Threat Model	Initialization Condition: ρ_k	Initialization Condition: c
Strong	✓	x
Weak	x	x

Fig. 7. ✓ indicates enhanced permissiveness from label chains with more than two elements; x indicates no permissiveness gains for our family of k-*Enf* enforcers with $k \geq 2$.

Figure 7 summarizes the results presented in this section. These results apply only to k-*Enf*. Thus, Theorems 4 and 5 do not preclude other enforcers (e.g., optimizations of k-*Enf* enforcers) where longer label chains increase permissiveness.

VIII. Other Enforcers

We now relate permissiveness with label chain length for enforcers other than k-*Enf*. Here longer label chains might increase permissiveness. But the results depend on the threat model, lattice size, and certain semantic properties of enforcers: *k-precise*, *k-varying*, and *k-dependent*.

A. In the Strong Threat Model

Longer label chains are useful for 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 characterized below as being *k-varying* and *k-precise*. Notice that k-varying label chains cannot exist when $\mathcal{L}$ has only one element.

Definition 9 (k-varying)**.** *Label chains $\langle \ell_1, \ell_2, \ldots, \ell_k \rangle$ and $\langle \ell'_1, \ell'_2, \ldots, \ell'_k \rangle$ with labels from lattice $\mathcal{L}$, are defined to be* k-varying *for $k \geq 2$ iff*

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

Definition 10 (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 $\tau = trace_E(C, M)$* 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:*

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

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 any enforcer E' that satisfies $(k-1)$-BNI($\mathcal{L}$) and is at least as permissive as E (i.e., $E \leq_c^{k-1,\mathcal{L}} E'$) will produce (at the same execution point) the same first k elements that appear in Ω. Consequently, the first k elements of Ω capture 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.

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 see this, 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.

Definition 11 ($(k-1)$-dependent label chains)**.** *E' produces* $(k-1)$-dependent label chains *for $k - 1 \geq 1$ iff E' is an enforcer and for some function $f_{E'}$:*

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

For example, k-*Enf* produces k-dependent label chains, because k-*Enf* uses $f_{k\text{-}Enf}(T(x), \ldots, T^k(x)) \triangleq T^k(x)$ for computing $T^{k+1}(x)$. Notice, if an enforcer E' produces $(k-1)$-dependent label chains, then that mechanism cannot produce k-varying label chains.

An enforcer E' that produces $(k-1)$-dependent label chains cannot both satisfy $(k-1)$-BNI and be at least as permissive as E, which produces some k-precise k-varying label chains: 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. A detailed proof of Theorem 6 is found in [23].

Theorem 6. *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,*

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

For an enforcer E' that uses label chains of length $k-1$ (i.e., produces $(k-1)$-dependent label chains), Theorem 6 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 4, which stipulates that k-*Enf* does not benefit from longer label chains under conventional initialization, enforcer E in Theorem 6 does benefit.

Theorem 6 (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 6 (ii): In the Appendix, 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.

As Theorem 6 stipulates, any mechanism that approximates the third label by repeating the second label in the label chain (i.e., produces 2-dependent label chains) loses permissiveness against 3-*Eopt* (which can produce 3-precise 3-varying label chains). Example program (1)

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

illustrates. Assume anchor variable m is associated with label chain $\langle \mathsf{M}, \mathsf{L}, \mathsf{L} \rangle$, anchor variable h is associated with $\langle \mathsf{H}, \mathsf{L}, \mathsf{L} \rangle$, anchor variable l is associated with $\langle \mathsf{L}, \mathsf{L}, \mathsf{L} \rangle$, and $l \neq h$ holds. Without considering context $m > 0$, flexible variable w would be associated either with $\langle \mathsf{H}, \mathsf{L}, \mathsf{L} \rangle$ (due to $w := h$) or with $\langle \mathsf{L}, \mathsf{L}, \mathsf{L} \rangle$ (due to $w := l$), when execution of one of these assignments ends. Here, only w and $T(w)$ reveal information about guard $m > 0$. So, at the end of the **if**-statement, only $T(w)$ and $T^2(w)$ should be augmented with $T(m) = \mathsf{M}$. Thus, if $m > 0$ holds, then 3-*Eopt* associates w with $\langle \mathsf{H}, \mathsf{M}, \mathsf{L} \rangle$ at the end of the **if**-statement. Otherwise, 3-*Eopt* associates w with $\langle \mathsf{M}, \mathsf{M}, \mathsf{L} \rangle$ at the end of the **if**-statement.

Notice that, starting from a common initialization, label chain $\langle \mathsf{H}, \mathsf{M}, \mathsf{L} \rangle$ that 3-*Eopt* produces for w (when $m > 0$ holds) has a the second label that is not repeated. Instead, the third label in that chain is strictly less restrictive than the second label. However, 2-*Enf* (which actually computes the first two elements in a label chains and approximates the third element by repeating the second one) would have associated w with $\langle \mathsf{H}, \mathsf{M}, \mathsf{M} \rangle$ when $m > 0$ holds. Thus, the label chain deduced by 3-*Eopt* (which actually computes the first three elements in a label chains) for w is strictly more permissive than the label chain deduced by 2-*Enf*. So, this examples shows the usefulness of computing label chains with at least 3 elements. The Appendix extends this example to show the usefulness of label chains with at least k elements, for all $k > 3$.

B. 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 metalabel enables the decision to block assignment commands to be more permissive. (Previous theorems concerned 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.

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 $trace_E(C, M)$. Boolean expression $G^E_{a:=e}$ is satisfied in a memory M' according to

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

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

- 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. Then $G^E_{a:=e}$ will be characterized by:

Definition 12 (k-dependent condition)**.** *$G^E_{a:=e}$ 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 now show how the second label in a label chain makes the decision to block assignment commands more permissive. Theorem 7, which is proved in [23], states that if an enforcer E uses 1-dependent $G^E_{a:=e}$, then E 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 always to block $a := e$, even though in some executions that assignment is safe and allowed by 2-*Enf*.

Theorem 7. *For an enforcer E and lattice $\mathcal{L}_3 \triangleq \langle \{\mathsf{L}, \mathsf{M}, \mathsf{H}\}, \sqsubseteq, \sqcup \rangle$, if $G^E_{a:=e}$ is 1-dependent and 2-Enf $\leq^{0,\mathcal{L}_3}_c$ E, then E does not satisfy 0-BNI($\mathcal{L}_3$).*

Thus, enforcer E that uses label chains of length one (i.e., $G^E_{a:=e}$ is 1-dependent) cannot be at least as permissive as 2-*Enf*, which uses label chains of length two (i.e., $G_{a:=e}$ is 2-dependent). So, for the weakened threat model, permissiveness can be improved when using two (instead of one) labels for each variable.

Since most dynamic enforcement mechanisms proposed in the past satisfy TINI, we might wonder whether Theorem 7 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 do satisfy TINI. So, Theorem 7 does not hold when 0-BNI is replaced by TINI.

Familiar Two-level Lattice: Some authors use a two-level lattice $\mathcal{L}_2 \triangleq \langle\{\mathsf{L},\mathsf{H}\}, \sqsubseteq, \sqcup\rangle$ with $\mathsf{L} \sqsubset \mathsf{H}$, believing that their results will extend to arbitrary lattices. In this section, we give a result for $\mathcal{L}_2$ that does not hold for more complex lattices. Thus, generalizing from $\mathcal{L}_2$ to arbitrary lattices is not always a sound proposition.

Consider $\mathcal{L}_2$ with the weakened threat model. Previous work [22] 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. Figure 8 shows the modified rules for $E_{\mathsf{H,L}}$. We prove below (Theorem 8) that $G_{a:=e}$ defined in Figure 8 is 1-dependent.

$E_{\mathsf{H,L}}$ 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 information tagged with H in $T(w)$ is if $T(w)$ is updated with different labels in a conditional command that has a guard tagged with H. But, if the sensitivity of the guard is H, then due to function U in Figure 8, 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 $Init_{E_{\mathsf{H,L}}}(bc) = \bot$.

Theorem 7 does not hold when $\mathcal{L}_3$ is replaced with $\mathcal{L}_2$ if E is $E_{\mathsf{H,L}}$. Instead, Theorem 8 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.

Theorem 8. *Enforcer $E_{\mathsf{H,L}}$ uses 1-dependent $G_{a:=e}$, satisfies 0-BNI($\mathcal{L}_2$), and satisfies 2-Enf $<_c^{0,\mathcal{L}_2}$ $E_{\mathsf{H,L}}$.*

So Theorem 8, which is proved in [23], contradicts expectations that longer label chains can 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 (2), which employs $\mathcal{L}_3$. Based on rules in Figure 8 and rules IF, SEQ in §V-C, $E_{\mathsf{H,L}}$ executes (2) as described in (i) and (ii) in §I. So, executing

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

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

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

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

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

Fig. 8. Modified rules for $E_{\mathsf{H,L}}$

		Label Chain Length	
		Greater than 1	Greater than 2
Threat Model	Strong	✓	✓
	Weak ($\mathcal{L}_3$)	✓	?
	Weak ($\mathcal{L}_2$)	x	x

Fig. 9. ✓ indicates where labels chains with length greater than the one indicated in the corresponding column can provide enhanced permissiveness; x indicates where longer label chains do not enhance permissiveness; ? indicates an open question.

(2) under $E_{\mathsf{H,L}}$ leaks sensitive $m > 0$ to principals observing nonsensitive variable l, and thus, 0-BNI is not satisfied. $E_{\mathsf{H,L}}$ thus illustrates that an enforcer designed to enforce two-level lattices cannot necessarily enforce arbitrary lattices.

Figure 9 summarizes the results presented in this section. We do not have a proof but we conjecture that label chains with more than two elements do not improve permissiveness for lattices with more than two elements under the weakened threat model.

IX. Related Work

Dynamic Enforcement Mechanisms and Leaks: The formalization of dynamic information flow enforcement mechanisms dates back to Bell and LaPadula [8]. The community realized early that dynamic enforcement mechanisms for information flow control might introduce leaks not present in the program itself. Denning [30], 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., [3], [4], [10], [19], [34]) satisfy TINI but not BNI, because they leak sensitive information when blocking an execution. In particular, NSU and PU are shown

in [4, Figure 1] to block execution depending on values of high confidentiality. There, BNI is not satisfied because the final output of low confidentiality will be observed depending on a value of high confidentiality. Other *hybrid* flow-sensitive enforcement mechanisms (e.g., [14], [27]), which employ some static analysis during execution, do not satisfy BNI, either, because observations of blocked traces might only be a strict prefix of those generated by normally terminated traces, whereas BNI requires equality. There are enforcement mechanisms (e.g., [1], [7]) that satisfy BNI, but they either handle only $\mathcal{L}_2$ or lose permissiveness by tagging variables with the same labels at the end of conditional commands independent of which branch is actually taken. We are not aware of an enforcement mechanism that uses label chains of length one, enforces labels from an arbitrary lattice, satisfies BNI, and is at least as permissive as 2-Enf. We are also not aware of an enforcement mechanism that uses label chains of length one, enforces $\mathcal{L}_2$, satisfies BNI, and is at least as permissive as $E_{\mathsf{H},\mathsf{L}}$.

Label Chains of Length Two: Certain dynamic enforcement mechanisms use label chains of length two. Buiras et al. [12] propose a purely dynamic enforcement mechanism that employs fixed metalabels to capture implicit flows caused by conditional commands. The purely dynamic enforcement mechanism in [12] 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).

Bedford et al. [6] use label chains where the second element is flow sensitive. That hybrid enforcement mechanism enforces TSNI on programs written in a **while**-language that supports references. The enforcement mechanism uses 2-dependent label chains and, therefore, Theorem 6 implies that this enforcement mechanism is not more permissive than an enforcer that produces 3-precise 3-varying label chains (e.g., 3-$Eopt$).

Unbounded Label Chains: Some enforcement mechanisms support label chains of unbounded length. Zheng et al. [39] 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 [39] employ label chains of up to two elements (e.g., $\langle \ell, \bot \rangle$ and $\langle \ell, \ell \rangle$), but the authors acknowledge [39, §3.3.2] that longer chains are sensible but do not show—as we do in this paper—that permissiveness can benefit from longer label chains. We explained (§VII) why permissiveness can be lost when using label chains of fixed length (instead of using longer label chains).

The enforcement mechanism presented in Zheng et al. [39] is mostly static, so it does not exhibit the kinds of leaks our paper examines through flow sensitive labels and blocking executions. Specifically, label chains in [39] are given as input; they are not deduced by the enforcement mechanism. Conditions on these labels are inlined by the programmer. If the static analysis succeeds, then the program will satisfy TINI. So, a type-correct program can be safely executed until normal termination. Techniques presented in [39] involving label chains have been implemented in Jif [28], [29]. We believe that any framework that supports dependent types, such as [13] and [25], is likely capable of expressing 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 [14] and [24], which handle $\mathcal{L}_2$, modify or skip the execution of an unsafe command. Similar to [12], the enforcement mechanism presented in [26] (which enforces labels from $\mathcal{L}_2$) diverges when reaching an unsafe command. Some enforcement mechanisms (e.g., [9], [17], [18], [35]) take no action, because they only update labels on variables; they do not perform any checks.

Certain purely dynamic enforcement mechanisms (e.g., [20], [36]) recover from exceptions caused by unsafe commands. They enforce *error sensitive noninterference*, which we believe is stronger than BNI. One technique they employ is assigning the same labels to variables after conditional commands, independent of the branch that is taken. Here, some permissiveness might be lost against 2-Enf, which allows labels on variables to depend on taken branches.

Comparing Enforcement Mechanisms: Russo et al. [32] study trade-offs between static and dynamic security analysis. They prove 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 [21]. They first define basic semantics that purely dynamic information-flow monitor may extend. Then, they introduce properties (i.e., *not look ahead*, *not look aside*) for the purely dynamic enforcement mechanisms they consider. Their impossibility theorem has the same style as our Theorem 7: an enforcement mechanism with the above properties cannot both satisfy TINI and be at least as permissive as [21]. Our Theorem 6 instead compares permissiveness of any two enforcement mechanisms that satisfy particular properties. And our permissiveness relation $\leq_{\rho}^{k,\mathcal{L}}$ is more general than the one presented in [32], because it is defined on any two enforcers and handles arbitrary lattices (not just $\mathcal{L}_2$) and initialization conditions.

Bielova et al. [11] 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, *precision*, and *transparency*, which stipulates that enforcement mechanisms do not alter the semantics of safe executions. *Termination-Aware Noninterference* (TANI) is the soundness goal, and it is expressed in terms of knowledge semantics. If an enforcement mechanism diverges the execution of an unsafe command satisfies TANI, then this mechanism does not leak sensitive information by taking this action. The theoretical framework considered in [11] assumes labels are taken from $\mathcal{L}_2$. Also, it assumes that a terminating execution produces one output, at the end, tagged L; if an execution diverges, no output is produced. So, TANI guarantees that dynamic enforcement mechanisms do not introduce leaks when it diverges executions

in the framework of [11]. Our section VIII-B explains that there is no danger these mechanisms could encode sensitive information in the flow-sensitive labels, because the framework in [11] is restricted to $\mathcal{L}_2$.

Acknowledgements

The authors would like to thank Andrew Hirsch and Drew Zagieboylo for comments on an earlier version of this paper, and the CSF '19 reviewers for suggestions that improved the clarity of this paper.

References

[1] A. Askarov, S. Chong, and H. Mantel. Hybrid monitors for concurrent noninterference. In *Proceedings of the 28th IEEE Computer Security Foundations Symposium*, pages 137–151, July 2015.

[2] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security*, ESORICS '08, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 113–124, New York, NY, USA, 2009. ACM.

[4] T. H. Austin and C. 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.

[5] 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 *Proceedings of the IEEE Symposium on Security and Privacy*, pages 813–830, May 2015.

[6] A. Bedford, S. Chong, J. Desharnais, E. Kozyri, and N. Tawbi. A progress-sensitive flow-sensitive inlined information-flow control monitor (extended version). *Computers & Security*, 71:114–131, 2017.

[7] A. Bedford, S. Chong, J. Desharnais, and N. Tawbi. A progress-sensitive flow-sensitive inlined information-flow control monitor. In *Proceedings of the 31st IFIP TC 11 International Conference, SEC 2016*, pages 352–366. Springer International Publishing, 2016.

[8] E. D. Bell and J. L. LaPadula. Secure computer systems: Mathematical foundations, 1973.

[9] L. Beringer. End-to-end multilevel hybrid information flow control. In *Proceedings of the 10th Asian Symposium on Programming Languages and Systems*, pages 50–65, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[10] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Generalizing permissive-upgrade in dynamic information flow analysis. In *Proceedings of the 9th Workshop on Programming Languages and Analysis for Security*, PLAS '14, pages 15:15–15:24, New York, NY, USA, 2014. ACM.

[11] N. Bielova and T. 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.

[12] P. Buiras, D. Stefan, and A. Russo. On dynamic flow-sensitive floating-label systems. In *Proceedings of the 27th IEEE Computer Security Foundations Symposium*, CSF '14, pages 65–79, Washington, DC, USA, 2014. IEEE Computer Society.

[13] P. Buiras, D. Vytiniotis, and A. 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.

[14] A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium*, pages 200–214, July 2010.

[15] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, Jan 1974.

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

[17] G. L. Guernic. Precise dynamic verification of confidentiality. In *Proceedings of the 5th International Verification Workshop*, pages 82–96, 2008.

[18] D. Hedin, L. Bello, and A. Sabelfeld. Value-sensitive hybrid information flow control for a JavaScript-like language. In *Proceedings of the 28th IEEE Computer Security Foundations Symposium*, pages 351–365, July 2015.

[19] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, CSF '12, pages 3–18, Washington, DC, USA, 2012. IEEE Computer Society.

[20] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCException are belong to us. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 3–17, May 2013.

[21] S. Hunt and D. 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.

[22] E. Kozyri, J. Desharnais, and N. Tawbi. Block-safe information flow control. Technical report, Cornell University, 2016.

[23] E. Kozyri, F. B. Schneider, A. Bedford, J. Desharnais, and N. Tawbi. Beyond labels: Permissiveness for dynamic information flow enforcement. Technical report, Cornell University, 2019.

[24] G. Le Guernic, A. Banerjee, T. Jensen, and D. 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.

[25] L. Lourenço and L. 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.

[26] J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. *Computers and Security*, 31(7):827–843, Oct. 2012.

[27] S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF)*, pages 146–160, 2011.

[28] A. 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.

[29] 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.

[30] D. E. Robling Denning. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

[31] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 186–199, Washington, DC, USA, 2010. IEEE Computer Society.

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

[33] A. Sabelfeld and A. 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.

[34] J. F. Santos and T. Rezk. An information flow monitor-inlining compiler for securing a core of JavaScript. In *Proceedings of the 29th IFIP TC 11 International Conference, SEC 2014*, pages 278–292, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[35] P. Shroff, S. Smith, and M. 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.

[36] D. Stefan, D. Mazières, J. C. Mitchell, and A. Russo. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming*, 27:e5, 2017.

[37] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 156–168, June 1997.

$$\text{(IFS1)}\ \frac{\exists i\colon 1 \le i \le k\colon \mathit{isSimple}(\textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M, i) \quad M(e) \neq 0 \quad cc' = M(cc).\mathit{push}(\langle M(T(e)),\ \emptyset,\ \emptyset\rangle}{\langle \textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M\rangle \rightarrow \langle C_1; i\textbf{-exit}, M[cc \mapsto cc']\rangle}$$

$$\text{(IFS2)}\ \frac{\exists i\colon 1 \le i \le k\colon \mathit{isSimple}(\textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M, i) \quad M(e) = 0 \quad cc' = M(cc).\mathit{push}(\langle M(T(e)),\ \emptyset,\ \emptyset\rangle}{\langle \textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ end}, M\rangle \rightarrow \langle C_2; i\textbf{-exit}, M[cc \mapsto cc']\rangle}$$

$$\text{(EXIT\_IFS)}\ \frac{cc' = cc.pop}{\langle i\textbf{-exit}, M\rangle \rightarrow \langle \textbf{stop}, M[\forall j\colon i < j \le k\colon T^j(w_i) \mapsto \bot, cc \mapsto cc']\rangle}$$

Fig. 10. Rules for simple **if** command

[38] D. M. Volpano and G. 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, 1997. Springer-Verlag.

[39] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2):67–84, Mar 2007.

APPENDIX

A. 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 $\bot$. Notice, there are programs where k-*Eopt* produces more permissive label chains than those produced by k-*Enf*.

Figure 10 gives the rules for augmenting k-*Enf* in order to obtain k-*Eopt*. Function $\mathit{isSimple}(C, M, i)$ decides whether a command C is simple:

(i) C is of the form

$$\textbf{if } a > 0 \textbf{ then } w_i := e \textbf{ else } w_i := n \textbf{ end}$$

(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**:

$$\textbf{if } m > 0 \textbf{ then } w := h \textbf{ else } w := 4 \textbf{ end} \qquad (18)$$

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. See [23] for the proof.

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 $Init_{k\text{-}Eopt}(bc) = \bot$. The soundness proof for k-*Eopt* can be found in [23].

B. Producing useful label chains with length $k \geq 2$

We use pgm_k defined below to show how k-*Eopt* can produce useful label chains with length $k \geq 2$. In particular,

we show that starting from a conventional initialization (i.e., flexible variables are initially associated with bottom label chains $\langle \bot, \bot, \ldots, \bot \rangle$) k-*Eopt* can gradually produce label chains such that

- an increasing number of elements are non-bottom,
- the non-bottom elements are not the same.

$$\begin{array}{l}
\textbf{if } a_1 > 0 \textbf{ then } w_1 := 0 \textbf{ else } w_1 := 1 \textbf{ end}; \\
z_1 := w_1; \\
\textbf{if } a_2 > 0 \textbf{ then } w_2 := z_1 \textbf{ else } w_2 := 2 \textbf{ end}; \\
z_2 := w_2; \\
\ldots \\
\textbf{if } a_{k-1} > 0 \textbf{ then } w_{k-1} := z_{k-2} \textbf{ else } w_{k-1} := k-1 \textbf{ end}; \\
z_{k-1} := w_{k-1}; \\
\textbf{if } a_k > 0 \textbf{ then } w_k := z_{k-1} \textbf{ else } w_k := k \textbf{ end}; \\
z_k := w_k;
\end{array}$$

Here all w_k and z_k are flexible variables. Assume lattice $\mathcal{L}_k$ of labels such that

$$\ell_0 \sqsupset \ell_1 \sqsupset \ell_2 \sqsupset \ldots \sqsupset \ell_k \sqsupset \bot \tag{19}$$

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 flexible variable 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:

$$\begin{array}{cccccc||l}
 & T(z_2) & T^2(z_2) & T^3(z_2) & \ldots & & \\
\langle & \ell_1 & \ell_2 & \bot & \ldots & \rangle & a_2 > 0 \\
\langle & \ell_2 & \ell_2 & \bot & \ldots & \rangle & a_2 \not> 0
\end{array}$$

($\mathbf{Z_3}$) After the execution of $z_3 := w_3$, these are the possible 3 chains for z_3:

$$\begin{array}{cccccc||l}
 & T(z_3) & T^2(z_3) & T^3(z_3) & T^4(z_3) & \ldots & & \\
\langle & \ell_1 & \ell_2 & \ell_3 & \bot & \ldots & \rangle & a_2 > 0 \wedge a_3 > 0 \\
\langle & \ell_2 & \ell_2 & \ell_3 & \bot & \ldots & \rangle & a_2 \not> 0 \wedge a_3 > 0 \\
\langle & \ell_3 & \ell_3 & \ell_3 & \bot & \ldots & \rangle & a_3 \not> 0
\end{array}$$

$\ldots$

($\mathbf{Z_j}$) After the execution of $z_j := w_j$, these are the possible j chains for z_j:

$$\begin{array}{cccccccc||l}
T & T^2 & T^3 & \ldots & T^{j-1} & T^j & T^{j+1} \ldots & & \\
\langle \ell_1 & \ell_2 & \ell_3 & \ldots & \ell_{j-1} & \ell_j & \bot \ldots \rangle & & a_2, a_3, \ldots, a_j > 0 \\
\langle \ell_2 & \ell_2 & \ell_3 & \ldots & \ell_{j-1} & \ell_j & \bot \ldots \rangle & & a_2 \not> 0 \wedge a_3, \ldots, a_j > 0 \\
\langle \ell_3 & \ell_3 & \ell_3 & \ldots & \ell_{j-1} & \ell_j & \bot \ldots \rangle & & a_3 \not> 0 \wedge a_4, \ldots, a_j > 0 \\
\ldots & & & & & & & & \\
\langle \ell_{j-1} & \ell_{j-1} & \ell_{j-1} & \ldots & \ell_{j-1} & \ell_j & \bot \ldots \rangle & & a_{j-1} \not> 0 \wedge a_j > 0 \\
\langle \ell_j & \ell_j & \ell_j & \ldots & \ell_j & \ell_j & \bot \ldots \rangle & & a_j \not> 0
\end{array}$$

$\ldots$

($\mathbf{Z_k}$) After the execution of $z_k := w_k$, these are the possible k chains for z_i:

$$\begin{array}{cccccc||l}
T & T^2 & T^3 & \ldots & T^{k-1} & T^k & \\
\langle \ell_1 & \ell_2 & \ell_3 & \ldots & \ell_{k-1} & \ell_k \rangle & a_2 > 0 \wedge a_3 > 0 \wedge \ldots \wedge a_j > 0 \\
\langle \ell_2 & \ell_2 & \ell_3 & \ldots & \ell_{k-1} & \ell_k \rangle & a_2 \not> 0 \wedge a_3 > 0 \wedge \ldots \wedge a_k > 0 \\
\langle \ell_3 & \ell_3 & \ell_3 & \ldots & \ell_{k-1} & \ell_k \rangle & a_3 \not> 0 \wedge \ldots \wedge a_k > 0 \\
\ldots & & & & & & \\
\langle \ell_{k-1} & \ell_{k-1} & \ell_{k-1} & \ldots & \ell_{k-1} & \ell_k \rangle & a_{k-1} \not> 0 \wedge a_k > 0 \\
\langle \ell_k & \ell_k & \ell_k & \ldots & \ell_k & \ell_k \rangle & a_k \not> 0
\end{array}$$

Notice that for $0 \leq j \leq k$ and $k \geq 2$, the label chains that k-*Eopt* produced for flexible variable z_j start with j non-bottom elements. Also, the non-bottom elements of the first label chain for z_j:

$$\langle \ell_1, \ell_2, \ell_3, \ldots, \ell_{j-1}, \ell_j, \bot, \ldots \rangle$$

are strictly monotonically decreasing:

$$\ell_1 \sqsupset \ell_2 \sqsupset \ell_3 \sqsupset \ldots \sqsupset \ell_{j-1} \sqsupset \ell_j$$

due to hypothesis (19). In [23], we prove that the label chains presented in ($\mathbf{Z_1}$) – ($\mathbf{Z_k}$) are the only possible chains that k-*Eopt* produces for variables z_j and that they are k-precise.