Nonmalleable Progress Leakage

Ethan Cecchetti
University of Wisconsin–Madison*
cecchetti@wisc.edu

***Abstract*—Information-flow control systems often enforce progress-insensitive noninterference, as it is simple to understand and enforce. Unfortunately, real programs need to declassify results and endorse inputs, which noninterference disallows, while preventing attackers from controlling leakage, including through progress channels, which progress-insensitivity ignores.**

This work combines ideas for progress-*sensitive* security with secure downgrading (declassification and endorsement) to identify a notion of securely downgrading progress information. We use hyperproperties to distill the separation between progress-sensitive and progress-insensitive noninterference and combine it with nonmalleable information flow, an existing (progress-insensitive) definition of secure downgrading, to define *nonmalleable progress leakage* (NMPL). We present the first information-flow type system to allow some progress leakage while enforcing NMPL, and we show how to infer the location of secure progress downgrades. All theorems are verified in Rocq.

I. Introduction

Information flow control (IFC) is a powerful tool for enforcing information security. The most common guarantee is *noninterference*, which prohibits a program's more-sensitive inputs from influencing—interfering with—its less-sensitive outputs [20]. Different requirements give rise to different formulations, the most popular being *progress-insensitive noninterference* (PINI)—which allows the program's termination behavior to leak information—and *progress-sensitive noninterference* (PSNI)—which does not. While PSNI provides stronger security, enforcing it requires extreme limits on any program construct that may not terminate, including simple while loops, leading many tools to enforce PINI instead.

Unfortunately, PINI is unsuitable in many real systems for two opposite reasons. First, noninterference is too restrictive. Many applications need untrusted inputs to sometimes influence decisions and allow controlled release of outputs derived using secrets. Second, the termination channel left open by progress-insensitivity allows arbitrary leakage, in theory in a single run [8], or easily across multiple runs if the system automatically restarts when hanging. This leakage makes PINI too permissive in the presence of active attackers.

Applications mixing secrets with code from untrusted sources provide an illustrative example. Arden et al. [4] suggest IFC types to enforce security in such a setting, but nontermination poses a concern. Consider the pseudocode in Figure 1 for a mobile app that displays nearby attractions on a map, revealing only the user's region to the server. IFC types can ensure that the application-supplied `buildMap` does not send `loc` to the server, only `region`, but there remain termination channels.

*Work done in part while author was at the University of Maryland.

```
1 loc = system.getLocation()
2 region = system.getRegion()
3 // Fetch attractions in region and loop
4 // over them, placing nearby ones on map
5 render(appCode.buildMap(loc, region))
```

Fig. 1. Pseudocode for app mapping nearby attractions.

Lines 1 or 2 might hang due to poor signal, revealing precise location details. Line 5 could hang benignly, due to a code bug or a non-responsive server, or maliciously, in an attempt to reveal `loc` through a termination channel.

An application might reasonably choose to accept unlikely and minimal leakage due to poor signal and ignore the app server hanging, which leaks nothing, but disallow the last, malicious, option. Unfortunately, enforcing PSNI would break the application, disallowing the minimal poor-signal leakage, while enforcing only PINI would allow the malicious progress leakage. This work shows how to differentiate these termination channels, formally define the desired restriction, and enforce it with a type system.

To accomplish this goal, we turn to a long line of work generalizing noninterference to support different notions of secure declassification and endorsement, known collectively as *downgrading* [12–15, 23, 24, 38, 39, 41, 48, 50]. Unfortunately, these conditions are generally progress-insensitive, at best suggesting enforcing a progress-sensitive variant by preventing any progress leakage [6]. This solution, while effective, imposes the same constraints as PSNI, either requiring all loop conditions to be fully public values [31, 46], or prohibiting any publicly visible operation after a loop with a non-public condition [9, 27], making programming nearly untenable and failing to address the above example.

To circumvent this crushing limitation, we allow programs to explicitly downgrade progress information. Some prior work supports progress downgrades, but they either appeal to halting oracles [27], making them unrealistic, quantify how much information leaks [27], making them non-compositional, or provide intensional definitions of security based on the authority of the declassifier [9]. Perhaps more importantly, they all consider only confidentiality and are thus unable to create, or even express, restrictions on who can influence the timing or content of declassifications, precisely what is needed to safely mix secrets with code and inputs from untrusted sources.

This work addresses this shortcoming by defining progress-based variants of robust declassification (RD) [50], and its extensions transparent endorsement and nonmalleable informa-

tion flow (NMIF) [12], and showing how to restrict progress downgrades to enforce them. These conditions constrain the impact of attackers on declassifications and secrets on endorsements, exactly what is needed when combining secrets and untrusted code. However, all existing enforcement mechanisms are progress-insensitive [6, 12, 14, 25, 29], seriously limiting their power to secure complex systems.

To build these conditions, we formalize *leakage-free progress* (LFP)—the distinction between PSNI and PINI—as a hyperproperty [16]. Hyperproperties, or sets of sets of traces, provide a framework for relating multiple executions of the same program, making them ideal for defining complex information security properties. Our approach draws insights from the definitions of LFP as well as RD and NMIF to define *nonmalleable progress leakage* (NMPL), a progress-based variant of NMIF that separates progress-sensitive and progress-insensitive NMIF.

We also define the first information-flow type system to enforce NMPL without prohibiting all progress leakage. This result shows that meaningful end-to-end security is achievable in the presence of malicious code and data without imposing draconian restrictions on basic looping constructs. For instance, it supports the desired policy in the mapping example above.

Finally, we leverage the static type system to *infer* progress downgrades. The inference procedure is efficient, adds a minimal number of downgrades, and is sound and complete with respect to the type system—any program it produces is well-typed, and it will find a way to add well-typed progress downgrades if one exists. The existence of such a procedure is a powerful result. It can automatically verify that all progress leakage satisfies NMPL while identifying where leakage can occur to support programmer audits.

The main contributions of this work are as follows.

- Section III defines LFP as a hyperproperty and generalizes it to define NMPL and progress-sensitive NMIF.
- Section IV defines a calculus and type system to enforce progress-sensitive NMIF and proves it secure.
- Section V presents a sound, complete, and efficient procedure for inferring a syntactically minimal placement of progress downgrade instructions within a program.
- All theorems in this paper have been mechanically verified in the Rocq Prover (formerly Coq) [34], the first mechanized statements or proofs of RD and NMIF of which we are aware.

Section VI gives an overview of the verification effort. Section VII presents related work and Section VIII concludes.

II. Label Model

Before defining new notions of security, we first describe the structure of our security policies. As is standard, we express information flow policies through a set of labels $\mathcal{L}$ that form a preorder. That is, there must be a reflexive and transitive ordering relation $\sqsubseteq$ (pronounced "flows-to") where $\ell_1 \sqsubseteq \ell_2$ means that ℓ_2 is at least as restrictive as ℓ_1, so data with label ℓ_1 may safely influence data with label ℓ_2.

Because the primary security conditions in this work—robust declassification, transparent endorsement, and nonmalleable information flow (NMIF)—concern the interaction of confidentiality and integrity, the labels must express both. Prior work accomplishes this goal by demanding a distributive lattice over a set of principals representing their authority. A label is then an ordered pair of the confidentiality authority and the integrity authority [3, 12, 14, 49]. This structure is simple and creates an obvious way to convert between confidentiality and integrity, which is necessary for defining and enforcing security. However, it also forces the space of confidentiality labels and integrity labels to be the same, limiting its applications.

This work extends NMIF-style definitions and results to a wider range of information-flow policy spaces by decoupling the confidentiality policies $\mathcal{C}$ from the integrity policies $\mathcal{I}$. Each must form a preorder (denoted $\sqsubseteq_{\mathcal{C}}$ and $\sqsubseteq_{\mathcal{I}}$, respectively), but the sets of policies need not be the same, or even contain any overlapping elements. Intuitively, $c \sqsubseteq_{\mathcal{C}} c'$ means the policy c' demands at least as much secrecy as c, while $i \sqsubseteq_{\mathcal{I}} i'$ means that i is at least as trusted as i'. It is therefore safe to use data labeled c (or i) in a context expecting data labeled c' (or i'). Labels are simply ordered pairs $\mathcal{L} = \mathcal{C} \times \mathcal{I}$ with $\sqsubseteq$ defined pointwise. We also require a least label $\bot \in \mathcal{L}$ such that $\bot \sqsubseteq \ell$ for any label ℓ.

Though confidentiality policies and integrity policies come from different spaces, we still need a way to relate the two. We thus require mappings between them in both directions. Using the language of Cecchetti et al. [12], we call these the *voice*, $\nabla : \mathcal{C} \to \mathcal{I}$, and the *view*, $\Delta : \mathcal{I} \to \mathcal{C}$. The voice of a confidentiality policy c represents the least trustworthy integrity level such that everyone who can write to $\nabla(c)$ can read c. Similarly, the view of i is the most secret confidentiality level where everyone who can write i can also read $\Delta(i)$.

To formalize this intuition, the mappings must satisfy two properties. First, they must be *anti-monotonic*. Since $c_1 \sqsubseteq_{\mathcal{C}} c_2$ means anyone who can read c_2 can also read c_1, any integrity level where everyone can read c_2 also guarantees everyone can read c_1. In other words, $\nabla(c_2) \sqsubseteq_{\mathcal{I}} \nabla(c_1)$. A dual argument holds for Δ. Second, $i \sqsubseteq_{\mathcal{I}} \nabla(c)$ means that anyone who can write to data labeled i must be able to read data labeled c. But $c \sqsubseteq_{\mathcal{C}} \Delta(i)$ means exactly the same thing! We therefore require $i \sqsubseteq_{\mathcal{I}} \nabla(c) \iff c \sqsubseteq_{\mathcal{C}} \Delta(i)$. Together, these properties make (∇, Δ) an *antitone Galois connection* [see e.g., 19].

As in prior work [3, 49], the voice and view combine to form a *reflection* operator $\mathbb{X} : \mathcal{L} \to \mathcal{L}$ that flips the confidentiality and integrity of a label: $\mathbb{X}(c, i) \stackrel{\text{def}}{=} (\Delta(i), \nabla(c))$. This simple construction extends the properties of ∇ and Δ to $\mathbb{X}$, making $\mathbb{X}$ an antitone Galois connection between $\mathcal{L}$ and itself: $\ell \sqsubseteq \mathbb{X}(\ell') \iff \ell' \sqsubseteq \mathbb{X}(\ell)$.

This reflection operator is critical for enforcing downgrade-tolerant security conditions. Prior work has shown downgrading to be secure only when anyone who can influence the data can also read it—$i \sqsubseteq_{\mathcal{I}} \nabla(c)$ for data labeled (c, i) [3, 12, 49]. We will see in Section IV that the same holds for downgrading progress. Since (∇, Δ) form an antitone Galois connection, this requirement is equivalent to the label flowing to its own reflection. Using the terminology of Zagieboylo et al. [49], we refer to labels that fail this requirement as *compromised*.

Definition 1 (Compromised Label [49])**.** A label ℓ is *compromised* if $\ell \not\sqsubseteq \overline{\ell}$—not everyone who can write to it can read from it. A label is *non-compromised* if $\ell \sqsubseteq \overline{\ell}$.

Further Generality: The results of this paper actually hold for an even more general structure than the one described above. The labels $\mathcal{L}$ need only form an arbitrary preorder, without explicit separation of confidentiality and integrity. Every pair of labels must have some lower bound, but there need not be a global least element.[1] Finally, the reflection operator $\overline{\cdot}$ simply needs to be anti-monotonic, not an antitone Galois connection.

This added generality does not complicate the proofs, but it does complicate intuition. In fact, I have yet to find a reasonable intuitive interpretation of the security results with a label model that does not fit the separate confidentiality and integrity structure above. However, there is no reason to constrain the technical result by the limits of the author's imagination, so all formal definitions and theorems are stated and proven in the more general model, except where explicitly noted otherwise.

III. Progress-Sensitive Hyperproperties

Progress-(in)sensitive security conditions generalize the classic notion of termination-(in)sensitive security [37, 46, 47]. They consider traces of effectful programs and properly account for leakage during execution, even when programs might not terminate. Existing work on progress-sensitivity has primarily focused on noninterference and models attacker knowledge by the set of possible initial memories consistent with the attacker's observations [6–9, 27]. Progress-sensitive noninterference (PSNI) then requires that the set of memories remain constant as the program executes—and the attacker observes more. Progress-insensitive noninterference (PINI) is more permissive, requiring only that an attacker learn no more from observing an event than from knowing the event exists. This formulation provides a strong intuition for noninterference. However, its extensions to robust declassification either treat confidentiality and integrity asymmetrically [6], making it unclear if it extends to nonmalleable information flow, or assume traces identify syntactic downgrades [25], requiring the language to track leakage for the definition to make sense.

We instead turn to *hyperproperties* [16], another framework commonly used for information security conditions, including noninterference, robust declassification, and nonmalleable information flow. A hyperproperty is a set of sets of traces, and various forms of noninterference are classic examples of hyperproperties [e.g., 2, 10, 16, 42, 44]. For instance, PSNI—an attacker learning nothing—requires any two traces with public-equivalent inputs to look the same at every point in execution, lest the adversary distinguish the inputs, thereby learning a secret. Notably, while many information-security formalizations use hyperproperties, most are not progress sensitive. They either ignore nonterminating programs entirely—resulting in progress- and termination-insensitive security definitions—or require termination behaviors to match, but ignore the effects of nonterminating programs, creating *termination*-sensitive definitions unsuitable for effectful programs.

To formalize progress-sensitive security through hyperproperties, we view program behaviors as traces. A trace t must include the program's inputs, denoted $\mathrm{in}(t)$, and any visible effects the program produces during execution. A trace may be either finite—if the execution terminates—or infinite—if it diverges. We also require an equivalence relation $\approx_{\mathcal{D}}$ on both inputs and finite prefixes of traces, where $p_1 \approx_{\mathcal{D}} p_2$ means p_1 is *indistinguishable* from p_2 to a low observer $\mathcal{D}$.

Many definitions separate "low" from "high" by a single label L—anything that flows to L is "low" and all other labels are "high"—we follow a more expressive approach and define security relative to an arbitrary downward-closed set of labels $\mathcal{D} \subseteq \mathcal{L}$, which need not have a maximal element. Labels in $\mathcal{D}$ are considered "low" while labels not in $\mathcal{D}$ are "high."

These basic building blocks are sufficient to define our security properties. Section IV-E below shows one way to instantiate these primitives and enforce security.

A. Noninterference and Leakage-Free Progress

We begin with noninterference, partially to fill a minor gap in the space of hyperproperty-based noninterference definitions, but primarily for expository reasons. The definition of nonmalleable information flow, and thus nonmalleable progress leakage, builds directly on that of noninterference while adding significant complexity.

The strongest condition, PSNI, is also the simplest to define. As described above, PSNI requires two traces with indistinguishable inputs to produce indistinguishable executions. To formalize that as a hyperproperty, recall that a hyperproperty is a set of sets of traces. $\mathbf{PsNi}_{\mathcal{D}}$ is then the hyperproperty such that, for any set of traces $T \in \mathbf{PsNi}_{\mathcal{D}}$ and any pair of traces $t_1, t_2 \in T$, if those traces have $\mathcal{D}$-equivalent inputs—$\mathrm{in}(t_1) \approx_{\mathcal{D}} \mathrm{in}(t_2)$—then every observation in one trace must also appear in the other, up to $\mathcal{D}$-equivalence. In other words, an adversary who can see only data and events with labels in $\mathcal{D}$ learns nothing from a full execution trace beyond what they could learn from that execution's input.

As is standard [16], we model observations as finite prefixes of a trace, resulting in the following definition, where $p \leq t$ denotes that p is a finite prefix of t, and $\mathbb{T}$ is the set of all possible traces.

$$\mathbf{PsNi}_{\mathcal{D}} \stackrel{\text{def}}{=} \{T \subseteq \mathbb{T} \mid \forall t_1, t_2 \in T.\, \mathrm{in}(t_1) \approx_{\mathcal{D}} \mathrm{in}(t_2) \implies \forall p_1 \leq t_1.\, \exists p_2 \leq t_2.\, p_1 \approx_{\mathcal{D}} p_2\}$$

Progress-insensitivity, by contrast, allows an attacker to gain information by learning an event exists, but no additional information from seeing its contents [7]. The hyperproperty must therefore allow traces to "stop early" from $\mathcal{D}$'s perspective due to infinite loops. When this happens, the trace $t^{\uparrow}$ with an infinite loop will appear to be a prefix of $t_{\downarrow}$, the one without. When considering finite observations, we do not know, a priori, which trace is which. However, every prefix of $t^{\uparrow}$ must be indistinguishable from some prefix of $t_{\downarrow}$, and any sufficiently short prefix of $t_{\downarrow}$ must be indistinguishable from a prefix of $t^{\uparrow}$.

[1] If $\mathcal{L}$ is finite, pairwise lower bounds are equivalent to a global lower bound, but if $\mathcal{L}$ is infinite, pairwise bounds are weaker.

To make this idea formal, given prefixes p_1 and p_2 of traces with $\mathcal{D}$-equivalent inputs, we require one of p_1 and p_2 to be $\mathcal{D}$-equivalent to a prefix of the other. Letting $p_1 \leq_{\mathcal{D}} p_2$ denote that p_1 is indistinguishable from a prefix of p_2—there is some $p_2' \leq p_2$ such that $p_1 \approx_{\mathcal{D}} p_2'$—we define $\mathbf{PiNi}_{\mathcal{D}}$ as follows.

$$\mathbf{PiNi}_{\mathcal{D}} \stackrel{\text{def}}{=} \{T \subseteq \mathbb{T} \mid \forall t_1, t_2 \in T.\, \mathrm{in}(t_1) \approx_{\mathcal{D}} \mathrm{in}(t_2) \implies \forall p_1 \leq t_1, p_2 \leq t_2.\, p_1 \leq_{\mathcal{D}} p_2 \vee p_1 \geq_{\mathcal{D}} p_2\}$$

Leakage-free progress: The distinction between PSNI and PINI is whether progress itself can leak information. A program does not leak information through progress if, after any sequence of events, the existence of another event reveals no further information about the program's secret inputs. More generally—including both confidentiality and integrity—the sequence of low (public or trusted) events a program produces must entirely determine if another low event occurs. We call this security property *leakage-free progress* (LFP) and present the first hyperproperty formalization of it.

To capture the above intuition, begin with the same setup as in PINI: finite prefixes $p_1 \leq t_1$ and $p_2 \leq t_2$ from traces where $\mathrm{in}(t_1) \approx_{\mathcal{D}} \mathrm{in}(t_2)$. If p_1 appears to be a *strict* prefix of p_2, denoted $p_1 <_{\mathcal{D}} p_2$, that means it is possible for an execution producing the $\mathcal{D}$-visible events in p_1 to visibly progress—produce another $\mathcal{D}$-visible event. Since LFP demands that the visible events determine the progress behavior, t_1 *must* progress, including some $\mathcal{D}$-visible event beyond the end of p_1. We thus define LFP as follows, letting $\mathrm{Prog}_{\mathcal{D}}(p, t)$ denote that trace t has another $\mathcal{D}$-visible event beyond the end of prefix p. Formally, $\mathrm{Prog}_{\mathcal{D}}(p, t) \stackrel{\text{def}}{=} \exists p' \leq t.\, p <_{\mathcal{D}} p'$,

$$\mathbf{Lfp}_{\mathcal{D}} \stackrel{\text{def}}{=} \{T \subseteq \mathbb{T} \mid \forall t_1, t_2 \in T, p_1 \leq t_1, p_2 \leq t_2.\ p_1 <_{\mathcal{D}} p_2 \implies \mathrm{Prog}_{\mathcal{D}}(p_1, t_1)\}$$

The requirement that $\mathrm{in}(t_1) \approx_{\mathcal{D}} \mathrm{in}(t_2)$ is implicit here; the premise $p_1 <_{\mathcal{D}} p_2$ can only hold with indistinguishable inputs.

Notably, the event t_1 progresses with need not match that of t_2; LFP allows leakage through the *content* of events, just not their existence. A program that outputs its secret to a public channel and then terminates satisfies LFP, as progress is not the leakage vector. On its own, LFP is therefore unlikely to be a useful security condition, but it precisely defines the distinction between PINI, which allows leakage *only* through progress, and PSNI, which disallows all leakage. Indeed, satisfying PSNI is the same as satisfying both PINI and LFP.

Theorem 1. *For any* $\mathcal{D} \subseteq \mathcal{L}$, $\mathbf{PsNi}_{\mathcal{D}} = \mathbf{PiNi}_{\mathcal{D}} \cap \mathbf{Lfp}_{\mathcal{D}}$.

These definitions are parameterized for a single attacker, but each extends simply to *all* attackers by taking the intersection over all downward-closed sets $\mathcal{D}$:

$$\mathbf{PsNi} \stackrel{\text{def}}{=} \bigcap \mathbf{PsNi}_{\mathcal{D}} \quad \mathbf{PiNi} \stackrel{\text{def}}{=} \bigcap \mathbf{PiNi}_{\mathcal{D}} \quad \mathbf{Lfp} \stackrel{\text{def}}{=} \bigcap \mathbf{Lfp}_{\mathcal{D}}$$

Hypersafety and Hyperliveness: Hyperproperties are often divided into *hypersafety*, requiring that bad things don't happen, and *hyperliveness*, requiring that good things do happen. Noninterference is a classic example of not just a hyperproperty, but specifically hypersafety [e.g., 2, 10, 42]. Indeed, the more common progress-*insensitive* noninterference is hypersafety.

Progress-*sensitive* noninterference, however, is not. It is subset-closed, but it both prohibits bad things—attackers cannot learn from events they see—and requires good things—both executions make progress or both silently diverge. Every hyperproperty is the intersection of a hypersafety property and a hyperlivness property [16], and the decomposition of $\mathbf{PsNi}$ is precisely progress insensitivity and progress leakage: $\mathbf{PiNi}$ is hypersafety, while $\mathbf{Lfp}$ is hyperliveness. This decomposition and a similar note in Section III-B, which are not stated as theorems, are the only claims in this paper not verified in Rocq.

B. Robust Declassification

Robust declassification (RD) recognizes that many programs need to declassify information to function, inherently violating noninterference. Instead, it prohibits an untrusted attacker from influencing the timing or content of declassifications. Identifying influence requires multiple program executions: two runs with different attacks must produce the same declassifications. This idea seems to suggest a definition similar to noninterference, where traces with the same trusted inputs must produce the same declassifications. Unfortunately, semantically, declassifications are defined as (confidentiality) violations of noninterference, meaning detecting them requires two executions with different secrets. Existing formalizations of RD therefore use four runs [12, 14, 29], comparing two pairs of inputs where secret inputs differ within a pair and an attack varies across the pairs.

These definitions model attacks by leaving holes in commands and inserting low-integrity attacker code. To keep our hyperproperties language-agnostic, we instead vary untrusted values in the initial input. These formulations produce equivalent guarantees for languages with conditional branches when enforcement theorems quantify over all programs, as is the case for our example language in Section IV-E. To model a hole with either of two attacks, simply hard code the attacks as branches of an if statement conditioned on part of the low-integrity input not used elsewhere, and vary only that value.

Defining Attackers: Making these ideas precise requires dividing confidentiality into "public" and "secret" and integrity into "trusted" and "untrusted." Prior approaches that demand confidentiality and integrity be dual policy lattices require public labels—the attacker's ability to read data—and untrusted labels—the attacker's ability to write data—to be the same policy sets [3, 6, 12, 14, 25, 29, 49]. Since our confidentiality and integrity policies may come from disperate spaces (see Section II), we require a more general definition. We bound an attacker's power by two downward-closed sets of labels, representing public and trusted policies, respectively, and require only that the attacker can read any security level they can write; the view of untrusted integrity must be public.

Definition 2 (Attacker)**.** A pair of label sets $\mathcal{A} = (\mathcal{P}, \mathcal{T})$ is an *attacker* if $\mathcal{P} = P \times \mathcal{I}$ and $\mathcal{T} = \mathcal{C} \times T$ for downward-closed sets $P \subseteq \mathcal{C}$ and $T \subseteq \mathcal{I}$ such that $\Delta(\overline{T}) \subseteq P$.

Note that we could have required the voice of secret confidentiality be trusted—$\nabla(\overline{P}) \subseteq T$. The definitions are equivalent since (∇, Δ) form an antitone Galois connection.

Recall from Definition 1 that non-compromised labels—those where $\ell \sqsubseteq \mathbb{X}(\ell)$—aim to capture labels where anyone who can influence the data can also read it. That means, for every attacker, a non-compromised label should be either public—the attacker *can* read it—or trusted—the attacker *cannot* write it—or both. Indeed, Definition 2 enforces this property.

Proposition 1. *For any attacker $(\mathcal{P}, \mathcal{T})$ and any label $\ell \in \mathcal{L}$, if $\ell \sqsubseteq \mathbb{X}(\ell)$, then $\ell \in \mathcal{P} \cup \mathcal{T}$.*

As with the label model itself (Section II), our theorems support a more general definition, requiring only that $\mathcal{P}$ and $\mathcal{T}$ be downward-closed and satisfy Proposition 1. It is again unclear, intuitively, what this more general structure represents. The proofs, however, rely only on Proposition 1, so there is no reason to restrict the formalism.

Defining Robust Declassification: This notion of an attacker allows us to formalize the above intuition for RD that an attacker should have no influence over the timing or content of declassifications. Prior definitions only reason about the behavior of terminating traces [12, 14, 29], making them not only progress-insensitive, but entirely unable to reason about progress leaks. Their structure, however, provides useful intuition. They require that, given any set of four traces $t_{11}, t_{12}, t_{21}, t_{22}$ with inputs $\sigma_{ij} = \text{in}(t_{ij})$, if

1) $\sigma_{11} \approx_{\mathcal{P}} \sigma_{21}$ and $\sigma_{12} \approx_{\mathcal{P}} \sigma_{22}$—$(\sigma_{11}, \sigma_{21})$ and $(\sigma_{12}, \sigma_{22})$ are the pairs of inputs varying secrets,
2) $\sigma_{11} \approx_{\mathcal{T}} \sigma_{12}$ and $\sigma_{21} \approx_{\mathcal{T}} \sigma_{22}$—only the attack varies across the pairs, and
3) all traces terminate,

then the second attack cannot leak secrets unless the first does as well. That is, $t_{11} \approx_{\mathcal{P}} t_{21}$ implies $t_{12} \approx_{\mathcal{P}} t_{22}$.

This formulation has two major shortcomings. First, it cannot reason about any divergent programs or constrain progress leakage. Second, enforcing it requires prohibiting endorsement, as endorsed data may safely influence future declassifications.

The key to solving both problems lies in explicitly considering partial executions and only restricting declassification prior to any (semantic) endorsements. More formally, given any program point in the first trace $p \leq t_{11}$, if no (semantic) endorsements have occurred by p and the first attack cannot yet differentiate the secrets then the second attack must not leak information to that point.

Checking for semantic endorsements is simple: only consider prefixes in the second attack $p_{12} \leq t_{12}$ when $p_{12} \approx_{\mathcal{T}} p$. If an endorsement has already occurred, there will be no such prefixes and the condition will hold vacuously.

Checking if the first attack leaks nothing up to p is more complicated for two reasons. First, we must pick an appropriate definition for "leaks nothing." One might assume that we should use whatever notion of leakage we aim to constrain: a PSNI-like structure for constraining all leakage, a PINI-like structure to allow progress leakage but nothing else, or an LFP-like structure to prohibit only progress leakage. Unfortunately, using this approach for both attacks does not give the desired result.

Consider the following program where a has a public–untrusted label and y and z have secret labels.

$$\begin{array}{l}\mathsf{while}\ a = y\ \mathsf{do}\ \mathsf{skip}\,;\\ \mathsf{declassify}\ z\ \mathsf{to}\ \textsc{PublicTrusted}\end{array}$$

A progress-insensitive definition should ignore the progress leak in the first line, correctly identify that the attacker cannot influence the second line, and consider this program (progress-insensitively) robust. However, an RD definition using a progress-insensitive notion of leakage for the first attack—the one used to check if leakage is allowed—would incorrectly classify the program. Given secrets y_1 and y_2, the first attack could set $a = y_1$, sending t_{11} into an infinite loop and causing all leakage in the first attack to stem from progress. The result would satisfy any progress-insensitive notion of "leaks nothing." However, a second attack where a differs from both y_1 and y_2 would cause t_{12} and t_{22} to both execute line two, producing more direct (non-progress) leakage and appear insecure.

Sending the program into an infinite loop causes the attacker to (voluntarily) learn *less* than the developer intended, and should be irrelevant to a progress-insensitive condition. Prior work rules out such nontermination-based irrelevant attacks by demanding termination [14, 29]. The corresponding progress formulation would require the first attack to progress beyond the current execution point p for both secrets.

Such a condition ensures the first attack does exhibit a progress leak up to p. When combined with the existing PINI-style assumption, Theorem 1 tells us the attack cannot leak anything, regardless of the channel; it must exhibit PSNI up to p. We therefore require this condition directly and prohibit *all* leakage between t_{11} and t_{21}. The definition of "leaks nothing" in the *second* attack, however, determines the version of RD.

The second complication of considering leakage up to some prefix $p \leq t_{11}$ is that p is only meant to limit the impact of *endorsements*. It is insufficient to simply require t_{21}, the trace with the other secret, match p exactly. Consider the following program with the same labels as above.

$$(\mathsf{while}\ y\ \mathsf{do}\ \mathsf{skip})\,;\, a \coloneqq 5$$

All leakage in this program is robust, as the attack has no impact on the behavior. However, using secret values 0 and 1, if $p \leq t_{11}$ happens to not include the assignment to a—though it exists in t_{11}—t_{21} could match it, and the requirement suggested above would demand the second attack leak nothing, which is clearly false.

To handle this situation correctly, we formalize the intuition that the first attack satisfies PSNI up to p by requiring it to hold for *every* prefix of t_{11} that is $\mathcal{T}$-equivalent to p. Since a is untrusted, some $p_{11} \leq t_{11}$ includes the assignment and satisfies $p_{11} \approx_{\mathcal{T}} p$, but t_{21} is stuck in an infinite loop, so it has no prefix $\mathcal{P}$-equivalent to p_{11}. This more expansive premise recognizes the leakage in the first attack, thereby allowing the same leakage to occur in the second.

Taking this structure and using PSNI as the notion of "leaks nothing" for the second attack gives us a complete definition of progress-sensitive robust declassification (PSRD).

$$\begin{aligned}\mathbf{PsRd}_{(\mathcal{P},\mathcal{T})} \stackrel{\text{def}}{=} \{T \subseteq \mathbb{T} \mid \forall t_{11}, t_{12}, t_{21}, t_{22} \in T.\\ \mathit{nmif\text{-}eq\text{-}in}_{(\mathcal{P},\mathcal{T})}(t_{11}, t_{12}, t_{21}, t_{22})\\ \implies \forall p \le t_{11}.\ (\forall p_{11} \le t_{11}.\, p_{11} \approx_{\mathcal{T}} p\\ \implies \exists p_{21} \le t_{21}.\, p_{11} \approx_{\mathcal{P}} p_{21})\\ \implies (\forall p_{12} \le t_{12}.\, p_{12} \approx_{\mathcal{T}} p\\ \implies \exists p_{22} \le t_{22}.\, p_{12} \approx_{\mathcal{P}} p_{22})\}\end{aligned}$$

Here we use $\mathit{nmif\text{-}eq\text{-}in}_{(\mathcal{P},\mathcal{T})}$ to indicate that the initial states of all four traces properly correspond to pairs of attacks and secrets, defined by the following equivalences.

$$\mathit{nmif\text{-}eq\text{-}in}_{(\mathcal{P},\mathcal{T})}(t_{11}, t_{12}, t_{21}, t_{22}) \stackrel{\text{def}}{=} \begin{array}{ccc}\mathrm{in}(t_{11}) & \text{–}\approx_{\mathcal{P}}\text{–} & \mathrm{in}(t_{21})\\ \downarrow & & \downarrow\\ \approx_{\mathcal{T}} & & \approx_{\mathcal{T}}\\ \downarrow & & \downarrow\\ \mathrm{in}(t_{12}) & \text{–}\approx_{\mathcal{P}}\text{–} & \mathrm{in}(t_{22})\end{array}$$

This definition follows exactly the intuition above. Take any four traces whose inputs match as two pairs, with only secrets differing within a pair and only untrusted (attacker) inputs differing between pairs. For every endorsement-alignment prefix $p \le t_{11}$ of one trace, if the first pair (attack) leaks nothing up to p, then neither does the second.

Like noninterference, this definition immediately extends to security against all attacks by taking the intersection over the set $\mathbb{A}$ of all attackers: $\mathbf{PsRd} \stackrel{\text{def}}{=} \bigcap_{\mathcal{A}\in\mathbb{A}} \mathbf{PsRd}_{\mathcal{A}}$.

Defining progress-insensitive robust declassification (PIRD) requires only changing the definition of "leaks nothing" in the second attack. Instead of prohibiting leakage through any channel up to p, PIRD allows leakage be due to progress. We use the same approach as $\mathbf{PiNi}$ and require the trace prefixes appear as prefixes of each other to a public observer.

$$\begin{aligned}\mathbf{PiRd}_{(\mathcal{P},\mathcal{T})} \stackrel{\text{def}}{=} \{T \subseteq \mathbb{T} \mid \forall t_{11}, t_{12}, t_{21}, t_{22} \in T.\\ \mathit{nmif\text{-}eq\text{-}in}_{(\mathcal{P},\mathcal{T})}(t_{11}, t_{12}, t_{21}, t_{22})\\ \implies \forall p \le t_{11}.\ (\forall p_{11} \le t_{11}.\, p_{11} \approx_{\mathcal{T}} p\\ \implies \exists p_{21} \le t_{21}.\, p_{11} \approx_{\mathcal{P}} p_{21})\\ \implies (\forall p_{12} \le t_{12}, p_{22} \le t_{22}.\, p_{12} \approx_{\mathcal{T}} p\\ \implies p_{12} \le_{\mathcal{P}} p_{22} \vee p_{12} \ge_{\mathcal{P}} p_{22})\}\end{aligned}$$

By changing the notion of leakage in the second attack instead to prohibit leakage through progress, but allow leakage through event contents, we acquire a robust declassification analogue of LFP we call *robust progress leakage* (RPL).

$$\begin{aligned}\mathbf{Rpl}_{(\mathcal{P},\mathcal{T})} \stackrel{\text{def}}{=} \{T \subseteq \mathbb{T} \mid \forall t_{11}, t_{12}, t_{21}, t_{22} \in T.\\ \mathit{nmif\text{-}eq\text{-}in}_{(\mathcal{P},\mathcal{T})}(t_{11}, t_{12}, t_{21}, t_{22})\\ \implies \forall p \le t_{11}.\ (\forall p_{11} \le t_{11}.\, p_{11} \approx_{\mathcal{T}} p\\ \implies \exists p_{21} \le t_{21}.\, p_{11} \approx_{\mathcal{P}} p_{21})\\ \implies (\forall p_{12} \le t_{12}, p_{22} \le t_{22}.\, p_{12} \approx_{\mathcal{T}} p\\ \implies p_{22} <_{\mathcal{P}} p_{12} \implies \mathrm{Prog}_{\mathcal{P}}(p_{22}, t_{22}))\}\end{aligned}$$

As with noninterference, enforcing PSRD is equivalent to enforcing both PIRD and RPL.

Theorem 2. *For any attacker $\mathcal{A}$, $\mathbf{PsRd}_{\mathcal{A}} = \mathbf{PiRd}_{\mathcal{A}} \cap \mathbf{Rpl}_{\mathcal{A}}$.*

Hypersafety and Hyperliveness: Much like $\mathbf{PsNi}$, $\mathbf{PsRd}$ is subset-closed, but neither hypersafety nor hyperliveness. Unlike noninterference, however, the progress insensitivity and progress leakage split is not a decomposition into hypersafety and hyperliveness; $\mathbf{PiRd}$ is not hypersafety.

Formally, a hyperproperty H is hypersafety if, for any trace set T violating H ($T \notin H$), there is a finite set $\{p_i\}$ of finite prefixes such that (1) for each p_i, there is some $t_i \in T$ such that $p_i \le t_i$, and (2) if T' satisfies property 1, then $T' \notin H$ [16]. $\mathbf{PiRd}_{\mathcal{A}}$ does not satisfy this requirement. Consider a set $T^\star$ of four traces, t_{11}, t_{12}, t_{21}, and t_{22} where the trace inputs match as required by $\mathbf{PiRd}_{\mathcal{A}}$, t_{11} and t_{21} are infinite but contain no visible events at all, and t_{12} and t_{22} have different public-untrusted events—no traces contain trusted events. Here t_{11} and t_{21} leak no information relative to each other, even through progress, but t_{12} and t_{22} do, meaning $T^\star \notin \mathbf{PiRd}_{\mathcal{A}}$. However, any finite prefix of t_{11} can be extended with some public-untrusted event, rendering the PSNI-style premise false, creating a trace set that *does* satisfy $\mathbf{PiRd}_{\mathcal{A}}$. This counterexample is not verified in Rocq.

We leave decomposing these more complicated hyperproperties into hypersafety and hyperliveness as future work.

C. Nonmalleable Information Flow

While RD constrains declassification based on integrity, recall that transparent endorsement (TE) constrains endorsement based on confidentiality and prohibits endorsement of secret information. As in the original formulation [12], TE is precisely dual to RD, switching the roles of the attacks and secrets.

This duality allows for immediate definitions of $\mathbf{PsTe}_{\mathcal{A}}$, $\mathbf{PiTe}_{\mathcal{A}}$, and *transparent progress control*, $\mathbf{Tpc}_{\mathcal{A}}$, a prohibition on secrets influencing an attacker's control over progress.

$$\begin{aligned}\mathbf{Tpc}_{(\mathcal{P},\mathcal{T})} \stackrel{\text{def}}{=} \{T \subseteq \mathbb{T} \mid \forall t_{11}, t_{12}, t_{21}, t_{22} \in T.\\ \mathit{nmif\text{-}eq\text{-}in}_{(\mathcal{P},\mathcal{T})}(t_{11}, t_{12}, t_{21}, t_{22})\\ \implies \forall p \le t_{11}.\ (\forall p_{11} \le t_{11}.\, p_{11} \approx_{\mathcal{P}} p\\ \implies \exists p_{21} \le t_{21}.\, p_{11} \approx_{\mathcal{T}} p_{21})\\ \implies (\forall p_{12} \le t_{12}, p_{22} \le t_{22}.\, p_{12} \approx_{\mathcal{P}} p\\ \implies p_{22} <_{\mathcal{T}} p_{12} \implies \mathrm{Prog}_{\mathcal{T}}(p_{22}, t_{22}))\}\end{aligned}$$

Similarly, nonmalleable information flow is the intersection of RD and TE, immediately giving progress-sensitive and insensitive definitions, and a definition of *nonmalleable progress leakage*, $\mathbf{NmPl}_{\mathcal{A}}$, the progress-only counterpart to NMIF.

$$\begin{aligned}\mathbf{PsNmif}_{\mathcal{A}} &\stackrel{\text{def}}{=} \mathbf{PsRd}_{\mathcal{A}} \cap \mathbf{PsTe}_{\mathcal{A}}\\ \mathbf{PiNmif}_{\mathcal{A}} &\stackrel{\text{def}}{=} \mathbf{PiRd}_{\mathcal{A}} \cap \mathbf{PiTe}_{\mathcal{A}}\\ \mathbf{NmPl}_{\mathcal{A}} &\stackrel{\text{def}}{=} \mathbf{Rpl}_{\mathcal{A}} \cap \mathbf{Tpc}_{\mathcal{A}}\end{aligned}$$

IV. A Core Calculus for Secure Progress Leakage

We now describe a core calculus that securely constrains progress leakage without eliminating it. The calculus is a simple

$$\text{[E-STOP]}\ \frac{}{\langle \mathsf{skip}, \sigma\rangle \xrightarrow{\mathrm{stp}} \langle \mathsf{stop}, \sigma\rangle} \qquad \text{[E-ASSIGN]}\ \frac{\langle e, \sigma\rangle \Downarrow n \qquad x \in \mathrm{dom}(\sigma)}{\langle x := e, \sigma\rangle \xrightarrow{\mathrm{a}(x,n)} \langle \mathsf{skip}, \sigma[x \mapsto n]\rangle}$$

$$\text{[E-SEQSTEP]}\ \frac{\langle c_1, \sigma\rangle \xrightarrow{\alpha} \langle c_1', \sigma'\rangle \qquad c_1' \neq \mathsf{stop}}{\langle c_1 \,;\, c_2, \sigma\rangle \xrightarrow{\alpha} \langle c_1' \,;\, c_2, \sigma'\rangle} \qquad \text{[E-SEQSKIP]}\ \frac{}{\langle \mathsf{skip} \,;\, c, \sigma\rangle \xrightarrow{\bullet} \langle c, \sigma\rangle}$$

$$\text{[E-PDOWNSTEP]}\ \frac{\langle c, \sigma\rangle \xrightarrow{\alpha} \langle c', \sigma'\rangle \qquad c' \neq \mathsf{stop}}{\langle \mathsf{pdown}_\ell\, c, \sigma\rangle \xrightarrow{\alpha} \langle \mathsf{pdown}_\ell\, c', \sigma'\rangle}$$

$$\text{[E-PDOWNSKIP]}\ \frac{}{\langle \mathsf{pdown}_\ell\, \mathsf{skip}, \sigma\rangle \xrightarrow{\mathrm{pd}(\ell)} \langle \mathsf{skip}, \sigma\rangle}$$

Fig. 2. Selected operational semantic rules

imperative calculus with only numeric values. The syntax below contains the standard IMP commands plus pdown and stop.

$$\begin{array}{rcl} e & ::= & n \mid x \mid e \otimes e \\ c & ::= & \mathsf{skip} \mid x := e \mid c \,;\, c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \\ & \mid & \mathsf{while}\ e\ \mathsf{do}\ c \mid \mathsf{pdown}_\ell\, c \mid \mathsf{stop} \end{array}$$

The stop command signals whole program termination. It differs from skip, which indicates only that a given operation has no more steps. Notably, stop should appear only on its own, does not appear in the surface language, and can never be well-typed (see Section IV-B).

The pdown, or *progress-downgrade*, operation is the main addition to the language. It downgrades (declassifies and endorses) control flow, and thus termination behavior. The command $\mathsf{pdown}_\ell\ c$ means: run c, then explicitly declassify (or endorse) the termination behavior of c to label ℓ.

To focus on progress leakage, we keep the calculus simple and omit data declassification and endorsement instructions. Adding them with typing and semantic rules similar to pdown would be straightforward and, while it would likely not hamper security, the proofs would become considerably more involved.

A. Operational Semantics

The semantics of the core calculus is mostly standard. Expressions, which always terminate, use a big-step semantics, and commands, which may diverge, use a small-step semantics. A semantic configuration $\langle c, \sigma\rangle$ consists of a pair of a command c (or expression e) and a memory σ, where $\sigma : \mathcal{V} \rightharpoonup \mathbb{N}$ is a partial function from variable names to values (natural numbers).

Figure 2 presents selected small-step operational semantics. The complete semantics can be found in Appendix A. Note that both E-SEQSTEP and E-PDOWNSTEP require that the inductive step not produce stop, ensuring that stop appears only for full program termination. Instead, E-SEQSKIP and E-PDOWNSKIP proceed directly when there is a skip.

Each semantic step also produces an event that will form the elements of an execution trace. All but three steps produce $\bullet$, indicating no effects or information. E-STOP produces stp, indicating that the program has terminated. E-ASSIGN produces an assignment event, $\mathrm{a}(x, n)$, indicating that variable x has been assigned value n. Finally, E-PDOWNSKIP produces a progress downgrade event, $\mathrm{pd}(\ell)$, explicitly making visible to label ℓ that the program has reached this point in execution, and therefore the body of the pdown_ℓ statement terminated.

$$\text{[SKIP]}\ \frac{}{\Gamma; pc \vdash \mathsf{skip} \diamond nt} \qquad \text{[ASSIGN]}\ \frac{\Gamma(x) = \ell \qquad \Gamma \vdash e : \ell}{\Gamma; \ell \vdash x := e \diamond nt}$$

$$\text{[IF]}\ \frac{\Gamma \vdash e : pc \qquad \Gamma; pc \vdash c_1 \diamond nt \qquad \Gamma; pc \vdash c_2 \diamond nt}{\Gamma; pc \vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \diamond nt}$$

$$\text{[SEQ]}\ \frac{\Gamma; pc_1 \vdash c_1 \diamond nt_1 \qquad \Gamma; pc_2 \vdash c_2 \diamond nt_2 \qquad pc_1 \sqsubseteq pc_2 \qquad nt_1 \sqsubseteq pc_2 \qquad nt_1 \sqsubseteq nt_2}{\Gamma; pc_1 \vdash c_1 \,;\, c_2 \diamond nt_2}$$

$$\text{[WHILE]}\ \frac{\Gamma \vdash e : pc \qquad \Gamma; pc \vdash c \diamond pc}{\Gamma; pc \vdash \mathsf{while}\ e\ \mathsf{do}\ c \diamond pc}$$

$$\text{[PDOWN]}\ \frac{\Gamma; pc \vdash c \diamond nt \qquad nt \sqsubseteq \mathbb{X}(nt) \qquad pc \sqsubseteq \ell}{\Gamma; pc \vdash \mathsf{pdown}_\ell\, c \diamond \ell} \qquad \text{[VARIANCE]}\ \frac{\Gamma; pc' \vdash c \diamond nt' \qquad pc \sqsubseteq pc' \qquad nt' \sqsubseteq nt}{\Gamma; pc \vdash c \diamond nt}$$

Fig. 3. Command typing rules

B. Type System

The type system has different judgments for expressions and commands. Since all values are numeric, the types are simply information-flow labels. Expression judgments take the form $\Gamma \vdash e : \ell$, where Γ is a partial map from variable names to labels, and ℓ is a label, and mean that label ℓ is at least as restrictive as the security policy of any input to expression e. That is, it is safe to treat e as having policy ℓ.

Command judgments take the form $\Gamma; pc \vdash c \diamond nt$, where Γ is as before, and pc and nt are labels used to constrain effects. The pc, or *program counter*, label is standard [21, 37], and it serves as both a lower bound on the visibility of a command's effects and a means of controlling implicit information flows. Flows can be either *explicit*, when resulting from a direct assignment like $x := y$, or *implicit* when stemming from control flow. Consider the following program.

$$\mathsf{if}\ y\ \mathsf{then}\ x := 0\ \mathsf{else}\ x := 1$$

This program only directly assigns constant values to x, but the value of y implicitly influences x. The pc label constrains these flows through two requirements: (1) the effects of an if statement must be no more public than the condition, and (2) an assignment is an effect, meaning the pc must flow to the label of the variable being assigned. Type-checking the above example then requires $\Gamma(y) \sqsubseteq pc$ and $pc \sqsubseteq \Gamma(x)$, transitivity ensuring $\Gamma(y) \sqsubseteq \Gamma(x)$ and preventing information leakage.

The nt, or *nontermination*, label constraints progress leakage. It serves as an upper bound on the sensitivity of termination

behavior of c. That is, anyone at or above nt may safely learn if c terminates without leaking information.

Figure 3 presents the full type system. VARIANCE formalizes the notion that pc is a *lower* bound while nt is an *upper* bound. If c only produces high effects and its termination behavior is only influenced by low data, it is safe to treat it as if it may produce lower effects—$pc \sqsubseteq pc'$—and has termination influenced by higher data—$nt' \sqsubseteq nt$. Including this rule allows ASSIGN, IF, and WHILE to demand equality of labels, rather than flows, simplifying the presentation and analysis.

SKIP and ASSIGN leave nt unconstrained, as skip and assignments always terminate. ASSIGN constrains explicit flows by requiring $\Gamma \vdash e : \ell$ where ℓ is the label of the assigned variable. Since assignments are effects, it also sets the pc bound at ℓ. IF requires $\Gamma \vdash e : pc$, meaning the effects of the branches must be bounded below by the label of the condition, completing the implicit flow restriction. As if's termination behavior is that of its branches, nt remains unchanged.

The other rules more directly constrain termination leakage. When sequentially composing commands $c_1 ; c_2$, if c_1 diverges, then c_2 will never execute, meaning visible effects of c_2 can leak whether or not c_1 terminated. To maintain security, SEQ therefore requires the effects of c_2 to be bounded below by nt_1. To ensure pc_1 and nt_2 correctly bound the composed command, pc_1 must be a lower bound on the effects of both c_1 and c_2—$\Gamma; pc_1 \vdash c_1 \diamond nt_1$ and $pc_1 \sqsubseteq pc_2$, respectively. Similarly, nt_2 must be an upper bound on their termination sensitivity—$nt_1 \sqsubseteq nt_2$ and $\Gamma; pc_2 \vdash c_2 \diamond nt_2$, respectively.

WHILE uses the same label for the condition, the pc, and nontermination labels for three reasons. First, the premise $\Gamma \vdash e : pc$ ensures that the guard can safely influence effects in the loop. Second, requiring c to type-check with a nontermination label of pc ensures that learning if c terminates does not leak more information than c's own effects. This requirement is necessary because the termination behavior of one execution of c can influence whether or not c executes again. Third, both the loop condition and the termination behavior of c can influence if the entire loop terminates. Since both have label pc, the nontermination label of the entire loop is pc.

Lastly, PDOWN concerns explicit downgrades of progress information. Recall that E-PDOWNSKIP, when executed, directly reveals to label ℓ that the body of the pdown_ℓ statement terminated. As a result, the termination behavior of $\mathsf{pdown}_\ell\, c$ leaks no *additional* information to anyone at or above ℓ, so PDOWN sets the new nontermination label to ℓ.

Since pdown_ℓ creates a visible effect at ℓ, the pc must properly bound it: $pc \sqsubseteq \ell$. Despite having a variance rule, this premise is a flow rather than an equality. Downgrading to pc and varying the nontermination label to ℓ is semantically different from downgrading directly to ℓ. The former releases information to pc, which may be lower than ℓ.

Enforcing NMIF requires restricting influence over both the content and timing of downgrades. For a pdown instructions, the old nontermination label nt bounds what information might be released, so we require it be non-compromised. Restricting the timing would normally involve similarly checking the pc to avoid improper influence over the control flow. However, if $\Gamma; pc \vdash c \diamond nt$, then either $pc \sqsubseteq nt$ or $\Gamma; pc \vdash c \diamond \ell'$ for any ℓ'. In the first case, the antimonotonicity of $\mathbb{X}$ together with nt being non-compromised ensure $pc \sqsubseteq \mathbb{X}(pc)$, making such a premise redundant. In the second case, setting $\ell' = \ell$ shows that there is no actual leakage to constrain.

Type Soundness: This type system satisfies the basic soundness property that well-typed programs do not get stuck. Since the type system presumes a mapping Γ of variables names to labels and the semantics operates over a memory σ, we require that all names referenced in Γ exist in σ.

Theorem 3 (Type Soundness)**.** *If $\Gamma; pc \vdash c \diamond nt$, then for any memory σ where $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\sigma)$, if $\langle c, \sigma\rangle \longrightarrow^* \langle c', \sigma'\rangle$, then either $c' = \mathsf{stop}$ or $\langle c', \sigma'\rangle$ can step.*

C. Example Revisited

To see the use of these typing rules, and in particular pdown, we look at the attraction mapping example from Section I using simple public/secret and trusted/untrusted policies. We assumed that lines 1 and 2 could both hang due to signal failures, revealing precise location. The nontermination label nt_{loc} of both lines would be secret, but it would remain trusted, as the attacker cannot influence the user's location or the control flow to this point. Line 5 will contact the server—a publicly-visible effect—requiring the pc label to be public.

With these labels, SEQ rule would identify the potential progress leakage and require an explicit declassification. Since nt_{loc} is secret–trusted, it is not compromised, so PDOWN allows wrapping lines 1 and 2 in pdown to public–trusted.

The code for line 5 comes from an untrusted source. To prevent it from modifying trusted data, it must type-check with an untrusted pc. Importantly, if it contains a loop depending on secrets—opening the attack we aim to prevent—WHILE requires the nt label to be secret–untrusted—*a compromised label*. PDOWN does not allow downgrading this compromised control flow, so SEQ ensures no later operation can create public or trusted effects. Embedded in a larger system, this would inevitably fail to compile, identifying the dangerous termination channel. Requiring line 5 to type-check with a *non-compromised* nt label prevents attacker-controlled termination leakage, but allows a benign loop over attractions to place nearby ones on a map, which has a public–untrusted nt label.

D. Program Behavior and Indistinguishability

To prove the security of this calculus, we must first define traces and low-equivalence in this setting. A trace consists of the program inputs, which we consider to be the initial memory σ, and a possibly-infinite stream of events st. Command c *produces* a trace (σ, st), denoted $c \rightsquigarrow (\sigma, st)$, if c outputs precisely st when run with initial memory σ. That is, if $\langle c, \sigma\rangle$ terminates, then st is the full list of events it produces, ending with stp. If $\langle c, \sigma\rangle$ diverges, then st is infinite and contains all events emitted during the execution.

The set of all traces c can produce is its *behavior*:

$$\mathrm{Behav}(c) \stackrel{\text{def}}{=} \{t \mid c \rightsquigarrow t\}$$

Recall from Section III that our security hyperproperties assume an indistinguishability relation $\approx_{\mathcal{D}}$ on finite trace prefixes that is parameterized by a downward-closed label set $\mathcal{D}$ representing low-sensitivity policies. A finite trace prefix consists of an input—the initial memory—and a finite list of events, so we need low-equivalence relations for each.

Two memories are $\mathcal{D}$-equivalent if they contain the same values for locations with labels in $\mathcal{D}$, though they may differ elsewhere. Formalizing this idea requires labels for memory locations, so we parameterize the equivalence on both $\mathcal{D}$ and a context Γ mapping locations to labels, as in the type system. $\mathcal{D}$-equivalence, denoted $\sigma_1 \cong^{\Gamma}_{\mathcal{D}} \sigma_2$, then demands only that σ_1 and σ_2 agree on x when $\Gamma(x) \in \mathcal{D}$. Formally,

$$\sigma_1 \cong^{\Gamma}_{\mathcal{D}} \sigma_2 \stackrel{\text{def}}{=} \forall x.\, \Gamma(x) \in \mathcal{D} \implies \sigma_1(x) = \sigma_2(x).$$

Two finite sequences of events are $\mathcal{D}$-equivalent if, at $\mathcal{D}$, they appear to contain the same events in the same order. We model a $\mathcal{D}$-observer as being unable to gain information from internal steps, signified by $\bullet$ events, and both assignments and progress downgrades at labels not in $\mathcal{D}$. To make this intuition precise, we define a silent-at-$\mathcal{D}$ predicate $Sil^{\Gamma}_{\mathcal{D}}(\alpha)$ as follows.

$$\frac{}{Sil^{\Gamma}_{\mathcal{D}}(\bullet)} \qquad \frac{\Gamma(x) \notin \mathcal{D}}{Sil^{\Gamma}_{\mathcal{D}}(\mathrm{a}(x,n))} \qquad \frac{\ell \notin \mathcal{D}}{Sil^{\Gamma}_{\mathcal{D}}(\mathrm{pd}(\ell))}$$

Note two important design decisions. First, the termination event stp is always visible, regardless of the label, formalizing that a progress-sensitive observer can always distinguish termination from silent infinite loops. Second, all assignments to low memory locations are visible, providing a strong security guarantee by modeling a powerful attacker that can continuously monitor public areas in memory. Modeling explicit output is possible using distinguished memory addresses and making more events silent, which cannot leak more information.

Equivalence of finite event sequences s_1 and s_2, denoted $s_1 \cong^{\Gamma}_{\mathcal{D}} s_2$, then simply ignores silent events and requires the rest to be identical.

$$\frac{}{\epsilon \cong^{\Gamma}_{\mathcal{D}} \epsilon} \qquad \frac{s_1 \cong^{\Gamma}_{\mathcal{D}} s_2}{\alpha \cdot s_1 \cong^{\Gamma}_{\mathcal{D}} \alpha \cdot s_2} \qquad \frac{s_1 \cong^{\Gamma}_{\mathcal{D}} s_2 \quad Sil^{\Gamma}_{\mathcal{D}}(\alpha)}{\alpha \cdot s_1 \cong^{\Gamma}_{\mathcal{D}} s_2} \qquad \frac{s_1 \cong^{\Gamma}_{\mathcal{D}} s_2 \quad Sil^{\Gamma}_{\mathcal{D}}(\alpha)}{s_1 \cong^{\Gamma}_{\mathcal{D}} \alpha \cdot s_2}$$

We can now define equivalence of trace prefixes, also parameterized by Γ and suggestively denoted $\approx^{\Gamma}_{\mathcal{D}}$, by requiring the initial memories and event sequences both be equivalent. That is, $(\sigma_1, s_1) \approx^{\Gamma}_{\mathcal{D}} (\sigma_2, s_2) \stackrel{\text{def}}{=} \sigma_1 \cong^{\Gamma}_{\mathcal{D}} \sigma_2$ and $s_1 \cong^{\Gamma}_{\mathcal{D}} s_2$.

Using these definitions of traces, prefixes, and equivalences is sufficient to state and prove the security of our core calculus. Since our equivalences are all parameterized on Γ, we will write $\mathbf{PsNi}^{\Gamma}$, and similarly for other hyperproperties, to indicate that we are using $\approx^{\Gamma}_{-}$ as the equivalence relation.

E. Proving Security

While this calculus and its type system are designed to enforce progress-sensitive NMIF, which allows for controlled data leakage that violates noninterference, it remains useful to confirm that explicit downgrades are the *only* way to violate noninterference. The omission of data downgrade operations means all leakage should be through progress channels. We verify this by proving that well-typed commands enforce PINI, which does not consider progress leakage.

Theorem 4 (PINI)**.** *If* $\Gamma; pc \vdash c \diamond nt$*, then* $\mathrm{Behav}(c) \in \mathbf{PiNi}^{\Gamma}$.

While some progress leakage is allowed, it should only come through two channels explicit in the type system: pdown instructions and programs with high nt labels. To eliminate leakage that the type system tracks and reports in the nt labels, we can demand the command type-check with a low nt label. To ensure that all remaining leakage arises from pdown instructions, we define a notion of *downgrade freedom* from a downward-closed set $\mathcal{D}$. Intuitively, a command that does not downgrade from outside $\mathcal{D}$—a "high" label—to inside $\mathcal{D}$—a "low" label—should also enforce LFP, and thus PSNI, at $\mathcal{D}$. We formalize this lack of downgrading as follows.

Definition 3 ($\mathcal{D}$-downgrade Freedom)**.** The proof $\Gamma; pc \vdash c \diamond nt$ is *$\mathcal{D}$-downgrade free* if, for every subcommand $\mathsf{pdown}_{\ell}\, c'$, either $\ell \notin \mathcal{D}$ or the subproof of $\Gamma; pc \vdash c' \diamond nt'$ has $nt' \in \mathcal{D}$.

$\mathcal{D}$-downgrade freedom and a low nt label are together sufficient to prevent progress leakage, proving that all leakage is properly accounted for. Together with Theorem 4, this guarantees PSNI.

Theorem 5 ($\mathcal{D}$-downgrade-free PSNI)**.** *For any downward-closed label set $\mathcal{D}$, if* $\Gamma; pc \vdash c \diamond nt$ *with* $nt \in \mathcal{D}$ *and the typing proof is $\mathcal{D}$-downgrade free, then* $\mathrm{Behav}(c) \in \mathbf{PsNi}^{\Gamma}_{\mathcal{D}}$.

Enforcing Nonmalleable Information Flow: For the same reason that Theorem 5 requires $nt \in \mathcal{D}$, a compromised nt label—which cannot be safely downgraded—signals potentially insecure progress leakage. As a result, not every well-typed command enforces nonmalleable progress leakage against every attacker. However, if $nt \in \mathcal{P} \cup \mathcal{T}$ for an attacker $\mathcal{A} = (\mathcal{P}, \mathcal{T})$, then secret influence has been safely declassified, attacker influence has been safely endorsed, or both. In each case, progress leakage will be robust, guaranteeing security.

Theorem 6 (Low-nt NMPL)**.** *Given an attacker* $\mathcal{A} = (\mathcal{P}, \mathcal{T})$*, if* $\Gamma; pc \vdash c \diamond nt$ *with* $nt \in \mathcal{P} \cup \mathcal{T}$*, then* $\mathrm{Behav}(c) \in \mathbf{NmPl}^{\Gamma}_{\mathcal{A}}$.

Also recall that non-compromised labels are public, trusted, or both for *every* attacker (Proposition 1). Combined with Theorem 6, this provides the condition necessary to ensure security against all attackers.

Theorem 7 (Nonmalleable Progress Leakage)**.** *If* $\Gamma; pc \vdash c \diamond nt$ *with* $nt \sqsubseteq \mathbb{X}(nt)$*, then* $\mathrm{Behav}(c) \in \mathbf{NmPl}^{\Gamma}$.

Because noninterference (in particular $\mathbf{PiNi}$) is strictly stronger than the corresponding form of NMIF ($\mathbf{PiNmif}$), Theorems 2, 4, and 7 combine to show that all well-typed programs enforce progress-sensitive NMIF.

Corollary 1 (Progress-Sensitive NMIF)**.** *If* $\Gamma; pc \vdash c \diamond nt$ *with* $nt \sqsubseteq \mathbb{X}(nt)$*, then* $\mathrm{Behav}(c) \in \mathbf{PsNmif}^{\Gamma}$.

V. Inferring Progress Downgrades

One major advantage of enforcing security with a static type system is support for inference procedures. To reduce programmer burden, they can omit explicit progress downgrades and instead the compiler can infer their locations if any secure placement is possible. Downgrades are generally considered sensitive operations requiring audits, but a constructive inference procedure can direct the programmer to specific code points, minimizing manual effort. We now present such an algorithm that is sound and complete with respect to the type system, highly efficient, and infers a minimal set of downgrades.

A. Label Structure

The extreme generality of the label model in Section II is good for expressive power, but its lack of structure makes using it for computations challenging. To make inference tractable, we require somewhat more structure on the labels.

First, flows-to ($\sqsubseteq$) must to be antisymmetric in addition to reflexive and transitive. There must be binary meet ($\sqcap$) and join ($\sqcup$) functions that compute the greatest lower bound and least upper bound of two labels, respectively. These changes make $\mathcal{L}$ a lattice. Second, along with the global least label $\bot$, we require a global greatest label $\top$ where $\ell \sqsubseteq \top$ for all $\ell \in \mathcal{L}$.

B. Inference Algorithm

The inference algorithm, pd-inf, consists of three parts. The first, elab, computes the minimum label of an expression e or determines that it cannot be typed. The second, pd-place, does most of the work. It places the progress downgrades, or determines that no placement will generate well-typed code. It also records auxiliary information that the third part, pd-lab-set, uses to set the label on downgrades placed by pd-place.

The goal is to infer a *secure* placement of pdown instructions, slightly different than producing any well-typed command. The type system guarantees progress-sensitive security only when the nt label is non-compromised (see Section IV-E), so pd-inf places pdown statements such that the resulting command is well-typed *with a non-compromised nontermination label*. Since pd-inf is complete with respect to the type system (Theorem 9 below), if any such placement exists, it will find one. Otherwise, the type system cannot prove that all of the program's leakage is nonmalleable, and pd-inf will fail, indicating this fact.

Expression Labels: Computing labels of expressions is straightforward. We parameterize elab on a typing context Γ, and specify it as a partial function that is defined precisely when e is well-typed. It produces the most permissive (lowest) label consistent with the typing context. It is defined as follows.

$$\begin{aligned} \mathsf{elab}_\Gamma(x) &= \Gamma(x) \\ \mathsf{elab}_\Gamma(n) &= \bot \\ \mathsf{elab}_\Gamma(e_1 \otimes e_2) &= \mathsf{elab}_\Gamma(e_1) \sqcup \mathsf{elab}_\Gamma(e_2) \end{aligned}$$

Downgrade Placement: The pd-place algorithm determines both the placement of pdown instructions and the nontermination label nt if inference is possible. It is also parameterized on Γ, and is a partial function from a pc label and a command c (free from progress downgrades) to a triple: a partial command $\tilde{c}$, a bound label b, and nt. A *partial command* is an intermediate data structure with the same structure as a command, but without labels on pdown and with auxiliary label information for control structures—conditionals, sequences, and loops. We write partial commands in blue and with a tilde.

$\mathsf{pd\text{-}place}_\Gamma(pc, c) =$ **match** c **with**
- **case** skip **do** $(\mathsf{skip}, \top, \bot)$
- **case** $(x := e)$ **do**
  - $\ell \leftarrow \mathsf{elab}_\Gamma(e)$;
  - **assert** $pc \sqcup \ell \sqsubseteq \Gamma(x)$;
  - $(x := e, \Gamma(x), \bot)$
- **case** (if e then c_1 else c_2) **do**
  - $\ell \leftarrow \mathsf{elab}_\Gamma(e)$;
  - $(\tilde{c}_1, b_1, nt_1) \leftarrow \mathsf{pd\text{-}place}_\Gamma(pc \sqcup \ell, c_1)$;
  - $(\tilde{c}_2, b_2, nt_2) \leftarrow \mathsf{pd\text{-}place}_\Gamma(pc \sqcup \ell, c_2)$;
  - **if** $nt_1 \sqcup nt_2 \sqsubseteq \mathbb{X}(nt_1 \sqcup nt_2)$ **then**
    - $(\mathsf{if}\{\ell\}\ e\ \mathsf{then}\ \tilde{c}_1\ \mathsf{else}\ \tilde{c}_2, b_1 \sqcap b_2, nt_1 \sqcup nt_2)$
  - **else**
    - $(\mathsf{if}\{\ell\}\ e\ \mathsf{then}\ (\mathsf{pdown}\ \tilde{c}_1)\ \mathsf{else}\ \tilde{c}_2, b_1 \sqcap b_2, nt_2)$
- **case** $(c_1 ; c_2)$ **do**
  - $(\tilde{c}_1, b_1, nt_1) \leftarrow \mathsf{pd\text{-}place}_\Gamma(pc, c_1)$;
  - $(\tilde{c}_2, b_2, nt_2) \leftarrow \mathsf{pd\text{-}place}_\Gamma(pc, c_2)$;
  - **if** $nt_1 \sqsubseteq b_2$ **then**
    - $(\tilde{c}_1 ;\{nt_1\}\ \tilde{c}_2, b_1 \sqcap b_2, nt_1 \sqcup nt_2)$
  - **else**
    - $((\mathsf{pdown}\ \tilde{c}_1) ;\{\bot\}\ \tilde{c}_2, b_1 \sqcap b_2, pc \sqcup nt_2)$
- **case** (while e do c') **do**
  - $\ell \leftarrow \mathsf{elab}_\Gamma(e)$;
  - **assert** $pc \sqcup \ell \sqsubseteq \mathbb{X}(pc \sqcup \ell)$;
  - $(\tilde{c}', b, nt) \leftarrow \mathsf{pd\text{-}place}_\Gamma(pc \sqcup \ell, c')$;
  - **if** $nt \sqsubseteq b$ **then**
    - $(\mathsf{while}\ e\ \mathsf{do}\{\ell \sqcup nt\}\ \tilde{c}', b \sqcap \mathbb{X}(pc \sqcup \ell), nt \sqcup pc \sqcup \ell)$
  - **else**
    - $(\mathsf{while}\ e\ \mathsf{do}\{\ell\}\ (\mathsf{pdown}\ \tilde{c}'), b \sqcap \mathbb{X}(pc \sqcup \ell), pc \sqcup \ell)$

Fig. 4. Procedure for inferring types and pdown placement

The bound label b indicates how much the pc can rise before inference will fail. That is, no pdown placement can allow c to type-check in context Γ with a non-compromised nt label and a pc label above $pc \sqcup b$. This label is important for efficiency in the sequence and while cases. Placing a progress downgrade around the first command or the loop body, respectively, can allow the second command or loop body to use a more permissive pc. A naive algorithm would thus make two recursive calls, one for each value of pc, resulting in exponential run time. The bound label allows us to replace the second recursive call with a single flow check, drastically improving efficiency.

Figure 4 presents the full pd-place algorithm. For skip, the bound label is $\top$ and nontermination label of $\bot$, since skip type-checks with any pc and nt. The assignment case is only slightly more complex. It asserts that $pc \sqcup \ell \sqsubseteq \Gamma(x)$, as the command will never type-check if that flow does not hold. If it does hold, pd-place sets $b = \Gamma(x)$, as the pc cannot rise above that level and still produce a valid typing proof, and $nt = \bot$, as the command always terminates.

For conditionals, pd-place first determines the label ℓ of the condition, before making recursive calls on both branches, using $pc \sqcup \ell$ as the pc label. Inference fails if the condition is

ill-typed or either recursive call fails. If they all succeed, the only remaining check is whether a downgrade is required.

Without a downgrade, the nontermination label of the conditional will be $nt_1 \sqcup nt_2$, where nt_1 and nt_2 are the nontermination labels of the branches. If that join is non-compromised, no downgrade is needed. If the join *is* compromised, a downgrade is required around one branch. Because both nt_i are outputs of pd-place, either $nt_i = \bot$ or $pc \sqcup \ell \sqsubseteq nt_i$ and both are non-compromised. That means whenever $nt_1 \sqcup nt_2$ is compromised, neither label is $\bot$, so $pc \sqcup \ell \sqsubseteq nt_i$. Downgrading the progress of either branch to $pc \sqcup \ell$ will therefore result in the entire if statement having a non-compromised nt label, so we arbitrarily choose to downgrade the then branch. Some programs may also need to downgrade the termination behavior of the full if statement—and thus the else branch. Inserting pdown into only the then branch preserves the ability to safely perform such a larger downgrade while optimistically hoping it will be unnecessary.

For the conditional's bound label, a higher pc must flow to both b_1 and b_2 for inference to succeed, which corresponds precisely to a bound of their meet $b_1 \sqcap b_2$.

Sequential composition is where the bound label becomes important. SEQ requires the pc of c_2 to be at least as high both the pc and nt labels of c_1. In the absence of a downgrade, c_2 must therefore type-check with $pc \sqcup nt_1$. A naive strategy would make a recursive call on c_1 and then try to infer downgrades for c_2 with $pc \sqcup nt_1$. If this inference on c_2 fails, however, it may still be possible by downgrading c_1's progress and infer downgrades on c_2 using pc, requiring a second recursive call.

The bound label allows us to avoid this double recursive call and the resulting exponential running time. Instead, pd-place makes one recursive call on each of c_1 and c_2 using pc for both. If the nontermination label nt_1 of c_1 flows to the bound label b_2 of c_2, inference will still succeed on c_2 when run with $pc \sqcup nt_1$, and no downgrade is needed. If $nt_1 \not\sqsubseteq b_2$, inference with $pc \sqcup nt_1$ would fail and a progress downgrade is required.

Additionally, the partial command for sequence includes an auxiliary label. This label represents the amount c_2's program counter must increase beyond pc in the typing proof. Without a downgrade, this value is nt_1. With a downgrade, no increase is required, so $\bot$ suffices.

The while case first computes the label ℓ of the condition and asserts that $pc \sqcup \ell \sqsubseteq \underline{\mathsf{X}}(pc \sqcup \ell)$. The nontermination label of the loop cannot be lower than $pc \sqcup \ell$, so this check is needed to ensure pd-place only outputs non-compromised nontermination labels. If the check passes, then it makes a recursive call on c using $pc \sqcup \ell$. Since while loops can sequence c with itself, we again need to check if the first command's nt label flows to the second's bound. As both commands are the same, this check becomes $nt \sqsubseteq b$. As in the sequence case above, a downgrade is necessary if and only if the flow does not hold.

The bound label for while is the meet of the recursively computed bound b and $\underline{\mathsf{X}}(pc \sqcup \ell)$. The first part captures the bound for the subcommand c. The second part ensures that $pc \sqcup \ell$ will remain non-compromised even when raising the pc.

```
pd-lab-set(pc, c̃) = match c̃ with
  case skip do skip
  case (x :=̃ e) do x := e
  case (if{ℓ} e then c̃1 else c̃2) do
    if e then (pd-lab-set(pc ⊔ ℓ, c̃1))
         else (pd-lab-set(pc ⊔ ℓ, c̃2))
  case (c̃1 ;{ℓ} c̃2) do
    pd-lab-set(pc, c̃1) ; pd-lab-set(pc ⊔ ℓ, c̃2)
  case (while e do{ℓ} c̃′) do
    while e do pd-lab-set(pc ⊔ ℓ, c̃′)
  case (pdown c̃′) do
    pdown_pc pd-lab-set(pc, c̃′)
```

Fig. 5. Procedure for setting pdown labels

The nontermination label is either $nt \sqcup pc \sqcup \ell$ if there is no downgrade, as all three impact termination, or $pc \sqcup \ell$ if a progress downgrade lowered the inner nontermination label.

Finally, we include an auxiliary label as in the sequence case. With no downgrade, the pc in the body must rise by $\ell \sqcup nt$, and with a downgrade, only ℓ.

Setting Downgrade Labels: For sequential composition, pd-place cannot determine the pc of c_2 until after the recursive call completes (and similarly for loops). That means it does not have enough information to set the labels on pdown instructions as it places them. Instead, it embeds auxiliary label information in its output, which pd-lab-set (Figure 5) uses on a second pass to set those labels. For each command, it recursively executes on each sub-command, increasing the pc by the label specified in the auxiliary information, and sets the label of each pdown command to the current pc as it goes.

The full inference algorithm first runs pd-place and then pd-lab-set on its output.

$$\text{pd-inf}_\Gamma(pc, c) = (\tilde{c}, b, nt) \leftarrow \text{pd-place}_\Gamma(pc, c)\ ; \\ (\text{pd-lab-set}(pc, \tilde{c}), nt)$$

C. Soundness, Completeness, and Correctness

The inference algorithm is sound and complete with respect to the type system using only non-compromised nontermination labels. Formally, soundness says that, if the algorithm succeeds, then the resulting command is well-typed with a non-compromised nt label.

Theorem 8 (Sound Inference)**.** *If* $\text{pd-inf}_\Gamma(pc, c) = (c', nt)$*, then* $\Gamma; pc \vdash c' \diamond nt$ *and* $nt \sqsubseteq \underline{\mathsf{X}}(nt)$*.*

Intuitively, completeness says that, if there is any way to place pdown instructions such that the resulting command is well-typed with a non-compromised nt label, then pd-inf will find one. To formalize this intuition, we use a downgrade-erasure operation, denoted $\lfloor c \rfloor$ and defined as follows.

$$\begin{aligned} \lfloor \mathsf{skip} \rfloor &= \mathsf{skip} \\ \lfloor x := e \rfloor &= x := e \\ \lfloor c_1\ ;\ c_2 \rfloor &= \lfloor c_1 \rfloor\ ;\ \lfloor c_2 \rfloor \\ \lfloor \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \rfloor &= \mathsf{if}\ e\ \mathsf{then}\ \lfloor c_1 \rfloor\ \mathsf{else}\ \lfloor c_2 \rfloor \\ \lfloor \mathsf{while}\ e\ \mathsf{do}\ c \rfloor &= \mathsf{while}\ e\ \mathsf{do}\ \lfloor c \rfloor \\ \lfloor \mathsf{pdown}_\ell\ c \rfloor &= \lfloor c \rfloor \end{aligned}$$

The formal definition of completeness is then as follows.

Theorem 9 (Complete Inference)**.** *For any command c and label nt such that $\Gamma; pc \vdash c \diamond nt$ and $nt \sqsubseteq \mathbb{X}(nt)$, there is some c' and nt' such that* $\mathsf{pd\text{-}inf}_\Gamma(pc, \lfloor c \rfloor) = (c', nt')$.

Finally, pd-inf is *correct* in that it does not modify commands aside from possibly adding downgrade instructions.

Theorem 10 (Correct Inference)**.** *If* $\mathsf{pd\text{-}inf}_\Gamma(pc, c) = (c', nt)$, *then* $c = \lfloor c' \rfloor$.

D. Efficiency and Minimality

The inference algorithm also aims to be efficient and only insert downgrades where necessary. Achieving either goals is simple: inserting downgrades everywhere is efficient but not minimal, while trying every combination of downgrades and selecting a minimal well-typed one is minimal but highly inefficient. The pd-inf algorithm accomplishes both.

To remain efficient, pd-inf consists of two sequential linear passes: pd-place and pd-lab-set. In each pass, the only operations that could impact performance are label operations (join, meet, reflection, and flows-to checks).

For simple label models, like small finite lattices, these operations are constant time, producing linear efficiency. More complicated label models may result in more overhead, but the structure of the algorithm mitigates some concern. All flows-to checks query if a nontermination label—always the join of (at most) linearly many labels—flows to a bound label—always the meet of (at most) linearly many labels. Common security lattices, including subset lattices of permissions [52], and free distributive lattices over a set of principals [5, 28, 43] can run these checks very efficiently.

Minimality: Even nonmalleable downgrades represent possible points of data leakage or corruption, so pd-inf aims to insert a minimal set. However, there are many ways to define "minimal." Semantic minimality—executing as few downgrades as possible—is appealing, but the semantically-minimal set of downgrades for a program might not be statically well-defined, as it could depend on the inputs.

Instead, we achieve a local syntactic notion of minimality: removing any downgrade inserted by pd-inf (without adding another elsewhere) will always be ill-typed. We formalize the idea of "removing a downgrade" using a relation $\preccurlyeq_{\text{PD}}$ to denote that two commands have the same structure, but one may have more syntactic downgrade instructions than the other and the pdown labels may not match. We define $\preccurlyeq_{\text{PD}}$ as the smallest structurally compatible preorder on commands—it is reflexive, transitive, and admits structurally recursive rules like $\frac{c_1' \preccurlyeq_{\text{PD}} c_1 \quad c_2' \preccurlyeq_{\text{PD}} c_2}{c_1'; c_2' \preccurlyeq_{\text{PD}} c_1; c_2}$—admitting the following rules.

$$\frac{c' \preccurlyeq_{\text{PD}} c}{c' \preccurlyeq_{\text{PD}} \mathsf{pdown}_\ell\, c} \qquad \frac{c' \preccurlyeq_{\text{PD}} c}{\mathsf{pdown}_{\ell'}\, c' \preccurlyeq_{\text{PD}} \mathsf{pdown}_\ell\, c}$$

Letting $c \equiv_{\text{PD}} c'$ denote commands with identical structure—they are the same except for the labels on pdown instructions—this relation allows us to formalize syntactic minimality.

Theorem 11 (Minimal Inference)**.** *If* $\mathsf{pd\text{-}inf}_\Gamma(pc, c_{\text{in}}) = (c, nt)$, *then for any command c' where $c' \preccurlyeq_{\text{PD}} c$, if $\Gamma; pc \vdash c' \diamond nt'$ with $nt' \sqsubseteq \mathbb{X}(nt')$, then $c' \equiv_{\text{PD}} c$.*

VI. Proof Approach and Rocq Details

All theorems in this paper are mechanically verified in The Rocq Prover [34] and are available online [11]. We encode expressions and commands with a deep embedding as inductive types and use `option` types to encode partial functions, including typing contexts, memories, and pd-inf. We assume decidable equality for variable names, decidable set inclusion for label sets and attackers in the security theorems, and decidable flows-to in the inference algorithm.

The proofs use the most general label model presented, but we verify that the less-general model that explicitly separates confidentiality and integrity is actually a special case (see Section II). There are also minor differences in the definitions of $\mathcal{D}$-equivalence of events and stores to make proofs simpler, so we include definition equivalence proofs for each.

A. Proving Security

The proofs of noninterference (Theorems 4 and 5) and NMPL (Theorem 7) use the bridge-step relation introduced by Bay and Askarov [9], which defines a configuration emitting a $\mathcal{D}$-visible event. That is, $\langle c, \sigma\rangle$ takes one or more steps, where only the last one produces something visible to $\mathcal{D}$. Formally, bridge steps are defined by the following inductive relation.

$$\frac{\langle c,\sigma\rangle \xrightarrow{\alpha} \langle c',\sigma'\rangle \quad \neg Sil^\Gamma_{\mathcal{D}}(\alpha)}{\langle c,\sigma\rangle \curvearrowright^{\mathcal{D}}_{\alpha} \langle c',\sigma'\rangle} \qquad \frac{\langle c,\sigma\rangle \xrightarrow{\alpha'} \langle c',\sigma'\rangle \quad Sil^\Gamma_{\mathcal{D}}(\alpha') \quad \langle c',\sigma'\rangle \curvearrowright^{\mathcal{D}}_{\alpha} \langle c'',\sigma''\rangle}{\langle c,\sigma\rangle \curvearrowright^{\mathcal{D}}_{\alpha} \langle c'',\sigma''\rangle}$$

A critical lemma shows that running one command with two $\mathcal{D}$-equivalent memories produces the same bridge step unless one configuration silently diverges, in which case we can bound the visible events and nontermination label.

Lemma 1 (Matching Bridge Step)**.** *For any command c where $\Gamma; pc \vdash c \diamond nt$ and memories σ_1 and σ_2 where $\sigma_1 \cong^\Gamma_{\mathcal{D}} \sigma_2$, if $\langle c, \sigma_1\rangle \curvearrowright^{\mathcal{D}}_{\alpha} \langle c', \sigma_1'\rangle$, then either*

- *$\langle c, \sigma_2\rangle \curvearrowright^{\mathcal{D}}_{\alpha} \langle c', \sigma_2'\rangle$ with $\sigma_1' \cong^\Gamma_{\mathcal{D}} \sigma_2'$, or*
- *$\langle c, \sigma_2\rangle$ silently diverges—diverges without ever producing a $\mathcal{D}$-visible event—and either $\alpha = \text{pd}(\ell)$ or both $\alpha = \mathsf{stop}$ and $nt \notin \mathcal{D}$.*

This lemma relies on a standard containment lemma.

Lemma 2 (Containment)**.** *For a downward-closed set $\mathcal{D}$, if $\Gamma; pc \vdash c \diamond nt$ with $pc \notin \mathcal{D}$ and $\langle c, \sigma\rangle \xrightarrow{\alpha} \langle c', \sigma'\rangle$, then $\sigma \cong^\Gamma_{\mathcal{D}} \sigma'$ and either $Sil^\Gamma_{\mathcal{D}}(\alpha)$ or $\alpha = \mathsf{stop}$.*

Nontermination: Since our theorems focus on distinguishing terminating executions from nonterminating ones, their proofs rely on a similar differentiation. The proof of progress-sensitive NMPL (Theorem 7) uses the following lemma.

Lemma 3 (While Trilemma)**.** *If a* while *loop is never stuck, then either (i) it terminates, or (ii) after some finite number of iterations, the body diverges, or (iii) the body converges on every iteration, but the loop executes infinitely many times.*

This lemma is not provable in a constructive logic like Rocq; deciding which branch holds requires solving the halting problem. Instead, we verify that it holds classically—ensuring logical consistency—and take it as an axiom for the theorems.

A note on computability theory: Lemma 3 is not just undecidable, no recursive enumeration procedure can determine the cause of nontermination and separate cases (*ii*) and (*iii*). One can, however, recursively enumerate the loops satisfying (*ii*) *using a halting oracle*. Deciding the while trilemma is thus under $\mathbf{0}''$ [for background, see, e.g., 40]. Indeed, the Rocq proof of Lemma 3 uses the classical assumption exactly twice: once to separate terminating loops from divergent ones, and a second time to differentiate the cause of nontermination.

B. Inference Properties

Recall from Section V-B that, pd-place returns a bound label b in addition to a command and nontermination label. The soundness and minimality of pd-inf rely on b properly bounding how much the pc can rise before inference fails. The following lemma formalizes this requirement.

Lemma 4 (Bound Validity)**.** *If* $\mathsf{pd\text{-}place}_\Gamma(pc, c) = (c', b, nt)$*, then for any non-compromised label* ℓ*,* $\ell \sqsubseteq b$ *if and only if* $\mathsf{pd\text{-}place}_\Gamma(pc \sqcup \ell, c)$ *is defined.*

Minimality also requires that the inferred nontermination label nt be the smallest possible label.

Lemma 5 (Least nt)**.** *If* $\mathsf{pd\text{-}inf}_\Gamma(pc, c_{\text{in}}) = (c, nt)$*, then for any* c' *and* nt' *if* $c' \equiv_{\text{PD}} c$ *and* $\Gamma; pc \vdash c' \diamond nt'$*, then* $nt \sqsubseteq nt'$*.*

VII. Related Work

We now discuss prior work on progress-sensitive security, secure downgrading, and information-security hyperproperties.

Progress-Sensitive Security: Early type-based enforcement of termination-sensitive noninterference either limit loops to only execute in fully public environments ($pc = \bot$) and have fully public conditions [31, 46] or operate in a pure λ-calculus with call-by-name semantics [1]. These constraints led most work to target progress-insensitive security. However, Askarov et al. [8] show how progress channels can leak arbitrary data in effectful languages, identifying a major risk of using progress-insensitive security with active attackers.

Moore et al. [27] provide a more precise type system similar to the one in Section IV-B. To further relax the type system's restrictions, they include a progress downgrade operation they call *cast*. Notably, the type system does not restrict *cast*. Instead, there is one semantics that appeals to a halting oracle and gets stuck if *cast* could leak anything, and a second that tracks a quantitative leakage budget.

Bay and Askarov [9] show how to define progress leakage as declassification in a progress-sensitive context and introduce tini blocks much like our pdown. They consider only confidentiality and bound declassifications by a separate notion of declassifier authority, with no ability to ensure robustness.

Secure Downgrading: Prior work on secure downgrades in IFC systems is extensive. Some allow labels to specify what downgrades directly [15, 23, 33]. Delimited release allows declassifications based on syntactic code structure [38]. Intransitive information flow restricts flows based on policies that may not be transitive [24, 32, 35, 45]. None of these ideas use confidentiality and integrity to constrain each other, and so cannot consider nonmalleability concerns.

The original formulation of robust declassification (RD) appears progress-sensitive, but gives no suggestions for enforcement [50]. The first enforcement mechanisms limit declassifications based on integrity levels, but both enforce only progress-insensitive definitions of RD [14, 29]. Askarov and Myers [6] use knowledge-based formulations to define both progress-sensitive and progress-insensitive forms of RD, but the definitions and enforcement are tailored to a four-point lattice and the only suggestion for enforcing progress-sensitive RD is to eliminate progress leakage entirely. McCall et al. [25] provide a knowledge-based definition of RD for use in the challenging setting of reactive web applications. Their definition is progress-insensitive and relies on trace events that track syntactic downgrades, making it intensional and hard to generalize to languges that do not directly track leakage.

Cecchetti et al. [12] define transparent endorsement as the dual of RD, combine the two into nonmalleable information flow, and present all three as hyperproperties. Their definition relies on traces having matching public-trusted events, making it inherently progress-insensitive, and they only enforce it in a fully terminating language. Soloviev et al. [41] reformulate RD and NMIF in modal logic using Kripke frames, including both progress-sensitive and progress-insensitive formulations, but like the progress-sensitive RD definitions, there is no enforcement mechanism.

Information-Security Hyperproperties: Information-security conditions have been key examples of hyperproperties since their introduction. Clarkson and Schneider [16] show how to express multiple versions of noninterference as hyperproperties. One such notion is *observational determinism* (OD) [26, 36], which Zdancewic and Myers [51] formulate similarly to noninterference, with explicit consideration of trace prefixes. Clarkson and Schneider present OD as a hyperproperty similar to our definition of $\mathbf{PiNi}_{\mathcal{D}}$ in Section III-A.

A variety of tools aim to specify or verify security-oriented hyperproperties. Relational Hoare Type Theory (RHTT) [30] allows for precise specification of 2-hypersafety properties like noninterference. Cartesian Hoare Logic (CHL) [42] generalizes RHTT to arbitrary k-hypersafety properties for relational traces (input/output pairs). As our main hyperproperties require considering intermediate trace prefixes of four traces, neither RHTT nor CHL can represent them.

Other tools and techniques aim to verify various forms of hyperproperties [10, 17, 18, 22, 44], with entirely different goals from this work. They aim to verify application-specific hyperproperties (within a certain class), sometimes at great computational expense. This work identifies highly-general

security hyperproperties and enforces them through inexpensive type checking. I hope the active research into verification will complement the results in this paper.

VIII. Conclusion

Information-flow control systems have long faced a tension between providing strong whole-program security guarantees and supporting necessary programming constructs, like declassification and loops. Noninterference is too restrictive, but uncontrolled downgrading complicates stating and proving security. Similarly, progress-insensitive security can leak arbitrary data to attackers who can influence nontermination, but requiring loops to condition only on public data makes many programs difficult to write.

To resolve this tension, we distilled the separation between progress-sensitivity and progress-insensitivity into a new hyperproperty called *leakage-free progress*, and generalized it as *nonmalleable progress leakage* (NMPL), an adaptation of the intuitions of nonmalleable information flow (NMIF) to secure progress channels. We explored how to enforce NMPL with a simple information-flow type system, and finally we showed how to efficiently infer the locations of necessary progress downgrade operations to aid developers or verify NMPL without explicit annotations.

We hope these foundations will support more expressive and powerful security-typed languages in the future. This work used an imperative core calculus without data downgrades to better focus on the core contribution and simplify formalisms and proofs. Extending these ideas to more practical languages would be valuable future work, with higher-order stateful languages posing a particularly interesting challenge.

Acknowledgments

This paper has only one author, but many people helped make it possible. Mike Hicks provided direction about which questions would be most impactful. Leo Lampropoulos suggested a suitable notion for minimality of the downgrade inference algorithm. Andrew K. Hirsch provided valuable comments on the text and, along with Tej Chajed, helped work through multiple challenges in the Rocq code. Additional thanks to Andrew C. Myers and Ashley Samuelson for help editing. Finally, the shepherd and other anonymous reviewers provided valuable feedback and suggestions for improving presentation.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke, "A core calculus of dependency," in *26th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '99)*, Jan. 1999.

[2] C. Abate, R. Blanco, D. Garg, C. Hriţcu, M. Patrignani, and J. Thibault, "Journey beyond full abstraction: Exploring robust property preservation for secure compilation," in *32nd IEEE Computer Security Foundations Symposium (CSF '19)*, Jun. 2019.

[3] C. Acay, R. Recto, J. Gancher, A. C. Myers, and E. Shi, "Viaduct: An extensible, optimizing compiler for secure distributed programs," in *42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '21)*, Jun. 2021.

[4] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers, "Sharing mobile code securely with information flow control," in *33rd IEEE Symposium on Security and Privacy (S&P '12)*, May 2012.

[5] O. Arden, J. Liu, and A. C. Myers, "Flow-limited authorization," in *28th IEEE Computer Security Foundations Symposium (CSF '15)*, Jul. 2015.

[6] A. Askarov and A. C. Myers, "Attacker control and impact for confidentiality and integrity," *Logical Methods in Computer Science (LMCS)*, vol. 7, no. 3, Sep. 2011.

[7] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *28th IEEE Symposium on Security and Privacy (S&P '07)*, May 2007.

[8] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *13th European Symposium on Research in Computer Security (ESORICS '08)*, Oct. 2008.

[9] J. Bay and A. Askarov, "Reconciling progress-insensitive noninterference and declassification," in *33rd IEEE Computer Security Foundations Symposium (CSF '20)*, Jun. 2020.

[10] R. Beutner and B. Finkbeiner, "Software verification of hyperproperties beyond k-safety," in *34th International Conference on Computer Aided Verification (CAV '22)*, Aug. 2022.

[11] E. Cecchetti, "Nonmalleable progress leakage rocq proofs," https://zenodo.org/records/15384760.

[12] E. Cecchetti, A. C. Myers, and O. Arden, "Nonmalleable information flow control," in *24th ACM Conference on Computer and Communication Security (CCS '17)*, Oct. 2017.

[13] E. Cecchetti, S. Yao, H. Ni, and A. C. Myers, "Compositional security for reentrant applications," in *42nd IEEE Symposium on Security and Privacy (S&P '21)*, May 2021.

[14] S. Chong and A. C. Myers, "Decentralized robustness," in *19th IEEE Computer Security Foundations Workshop (CSFW '06)*, Jul. 2006.

[15] ——, "End-to-end enforcement of erasure and declassification," in *21st IEEE Computer Security Foundations Symposium (CSF '08)*, Jun. 2008.

[16] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security (JCS)*, vol. 18, no. 6, pp. 1157–1210, 2010.

[17] N. Coenen, B. Finkbeiner, C. Sánchez, and L. Tentrup, "Verifying hyperliveness," in *31st International Conference on Computer Aided Verification (CAV '19)*, Jul. 2019.

[18] A. Farzan and A. Vandikas, "Automated hypersafety verification," in *31st International Conference on Computer Aided Verification (CAV '19)*, Jul. 2019.

[19] N. Galatos, *Residuated Lattices: An Algebraic Glimpse at Substructural Logics*, ser. Studies in Logic and the

Foundations of Mathematics. Elsevier Sience, 2007.

[20] J. A. Goguen and J. Meseguer, "Security policies and security models," in *3rd IEEE Symposium on Security and Privacy (S&P '82)*, Apr. 1982.

[21] A. K. Hirsch and E. Cecchetti, "Giving semantics to program-counter labels via secure effects," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, Jan. 2021.

[22] L. Lamport and F. B. Schneider, "Verifying hyperproperties with TLA," in *34th IEEE Computer Security Foundations Symposium (CSF '21)*, Jun. 2021.

[23] P. Li and S. Zdancewic, "Downgrading policies and relaxed noninterference," in *32nd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '05)*, Jan. 2005.

[24] H. Mantel and D. Sands, "Controlled declassification based on intransitive noninterference," in *2nd Asian Symposium on Programming Languages and Systems (APLAS '04)*, Nov. 2004.

[25] M. McCall, A. Bichhawat, and L. Jia, "Tainted secure multi-execution to restrict attacker influence," in *30th ACM Conference on Computer and Communication Security (CCS '23)*, Nov. 2023.

[26] J. McLean, "Proving noninterference and functional correctness using traces," *Journal of Computer Security (JCS)*, vol. 1, no. 1, pp. 37–57, Jan. 1992.

[27] S. Moore, A. Askarov, and S. Chong, "Precise enforcement of progress-sensitive security," in *19th ACM Conference on Computer and Communication Security (CCS '12)*, Oct. 2012.

[28] A. C. Myers and B. Liskov, "Complete, safe information flow with decentralized labels," in *19th IEEE Symposium on Security and Privacy (S&P '98)*, May 1998.

[29] A. C. Myers, A. Sabelfeld, and S. Zdancewic, "Enforcing robust declassification and qualified robustness," *Journal of Computer Security (JCS)*, vol. 14, no. 2, pp. 157–196, 2006.

[30] A. Nanevski, A. Banerjee, and D. Garg, "Verification of information flow and access control policies with dependent types," in *32nd IEEE Symposium on Security and Privacy (S&P '11)*, May 2011.

[31] K. R. O'Neill, M. R. Clarkson, and S. Chong, "Information-flow security for interactive programs," in *19th IEEE Computer Security Foundations Workshop (CSFW '06)*, Jul. 2006.

[32] S. Pinsky, "Absorbing covers and intransitive noninterference," in *16th IEEE Symposium on Security and Privacy (S&P '95)*, May 1995.

[33] N. Polikarpova, D. Stefan, J. Yang, S. Itzhaky, T. Hance, and A. Solar-Lezama, "Liquid information flow control," *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, Aug. 2020.

[34] Rocq development team, *The Rocq Prover*, 2025, version 8.20.1. [Online]. Available: https://rocq-prover.org/

[35] A. W. Roscoe and M. H. Goldsmith, "What is intransitive noninterference?" in *12th IEEE Computer Security Foundations Workshop (CSFW '99)*, Jun. 1999.

[36] A. Roscoe, "CSP and determinism in security modelling," in *16th IEEE Symposium on Security and Privacy (S&P '95)*, May 1995.

[37] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[38] ——, "A model for delimited information release," in *International Symposium on Software Security*, Nov. 2003.

[39] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *18th IEEE Computer Security Foundations Workshop (CSFW '05)*, Jun. 2005.

[40] R. I. Soare, *Turing Computability: Theory and Applications*. Springer, 2016.

[41] M. Soloviev, M. Balliu, and R. Guanciale, "Security properties through the lens of modal logic," in *37th IEEE Computer Security Foundations Symposium (CSF '24)*, Jul. 2024, to Appear.

[42] M. Sousa and I. Dillig, "Cartesian hoare logic for verifying k-safety properties," in *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*, Jun. 2016.

[43] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in Haskell," in *4th ACM SIGPLAN Haskell Symposium (HASKELL '11)*, Sep. 2011.

[44] H. Unno, T. Terauchi, and E. Koskinen, "Constraint-based relational verification," in *33rd International Conference on Computer Aided Verification (CAV '21)*, Jul. 2021.

[45] R. van der Meyden, "What, indeed, is intransitive noninterference?" in *12th European Symposium on Research in Computer Security (ESORICS '07)*, Sep. 2007.

[46] D. Volpano and G. Smith, "Eliminating covert flows with minimum typings," in *10th IEEE Computer Security Foundations Workshop (CSFW '97)*, Jun. 1997.

[47] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," *Journal of Computer Security (JCS)*, vol. 4, no. 2–3, pp. 167–187, 1996.

[48] L. Waye, P. Buiras, D. King, S. Chong, and A. Russo, "It's my privilege: Controlling downgrading in DC-labels," in *11th International Workshop on Security and Trust Management (STM '15)*, Sep. 2015.

[49] D. Zagieboylo, G. E. Suh, and A. C. Myers, "Using information flow to design an ISA that controls timing channels," in *32nd IEEE Computer Security Foundations Symposium (CSF '19)*, Jun. 2019.

[50] S. Zdancewic and A. C. Myers, "Robust declassification," in *14th IEEE Computer Security Foundations Workshop (CSFW '01)*, Jun. 2001.

[51] ——, "Observational determinism for concurrent program security," Jun. 2003.

[52] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," *Communications of the ACM*, vol. 54, no. 11, pp. 93–101, Nov. 2011.

APPENDIX A
FULL CALCULUS RULES

We now present the full semantic and typing rules of the core calculus from Section IV. The big-step semantic rules for expression are in Figure 6 and the small-step rules for all commands are in Figure 7. The typing rules for expressions are in Figure 8. Figure 3 in Section IV-B contains all typing rules for commands.

$\boxed{\langle e, \sigma\rangle \Downarrow n}$

$$\frac{}{\langle x, \sigma\rangle \Downarrow \sigma(x)} \qquad \frac{}{\langle n, \sigma\rangle \Downarrow n} \qquad \frac{\langle e_1, \sigma\rangle \Downarrow n_1 \qquad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1 \otimes e_2, \sigma\rangle \Downarrow (n_1 \otimes n_2)}$$

Fig. 6. Big-step operational semantics for expressions.

$\boxed{\langle e, \sigma\rangle \xrightarrow{\alpha} \langle e', \sigma'\rangle}$

[E-STOP]
$$\frac{}{\langle \mathsf{skip}, \sigma\rangle \xrightarrow{\mathrm{stp}} \langle \mathsf{stop}, \sigma\rangle}$$

[E-ASSIGN]
$$\frac{\langle e, \sigma\rangle \Downarrow n \qquad x \in \mathrm{dom}(\sigma)}{\langle x \coloneqq e, \sigma\rangle \xrightarrow{\mathrm{a}(x,n)} \langle \mathsf{skip}, \sigma[x \mapsto n]\rangle}$$

[E-SEQSTEP]
$$\frac{\langle c_1, \sigma\rangle \xrightarrow{\alpha} \langle c_1', \sigma'\rangle \qquad c_1' \neq \mathsf{stop}}{\langle c_1 \,;\, c_2, \sigma\rangle \xrightarrow{\alpha} \langle c_1' \,;\, c_2, \sigma'\rangle}$$

[E-SEQSKIP]
$$\frac{}{\langle \mathsf{skip} \,;\, c, \sigma\rangle \xrightarrow{\bullet} \langle c, \sigma\rangle}$$

[E-IFN]
$$\frac{\langle e, \sigma\rangle \Downarrow n \qquad n \neq 0}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle \xrightarrow{\bullet} \langle c_1, \sigma\rangle}$$

[E-IF0]
$$\frac{\langle e, \sigma\rangle \Downarrow 0}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle \xrightarrow{\bullet} \langle c_2, \sigma\rangle}$$

[E-WHILE]
$$\frac{}{\langle \mathsf{while}\ e\ \mathsf{do}\ c, \sigma\rangle \xrightarrow{\bullet} \langle \mathsf{if}\ e\ \mathsf{then}\ (c \,;\, \mathsf{while}\ e\ \mathsf{do}\ c)\ \mathsf{else}\ \mathsf{skip}, \sigma\rangle}$$

[E-PDOWNSTEP]
$$\frac{\langle c, \sigma\rangle \xrightarrow{\alpha} \langle c', \sigma'\rangle \qquad c' \neq \mathsf{stop}}{\langle \mathsf{pdown}_\ell\, c, \sigma\rangle \xrightarrow{\alpha} \langle \mathsf{pdown}_\ell\, c', \sigma'\rangle}$$

[E-PDOWNSKIP]
$$\frac{}{\langle \mathsf{pdown}_\ell\, \mathsf{skip}, \sigma\rangle \xrightarrow{\mathrm{pd}(\ell)} \langle \mathsf{skip}, \sigma\rangle}$$

Fig. 7. Full small-step operational semantics for commands.

$\boxed{\Gamma \vdash e : \ell}$

$$\frac{\Gamma(x) = \ell}{\Gamma \vdash x : \ell} \qquad \frac{}{\Gamma \vdash n : \ell} \qquad \frac{\Gamma \vdash e_1 : \ell \qquad \Gamma \vdash e_2 : \ell}{\Gamma \vdash e_1 \otimes e_2 : \ell} \qquad \frac{\Gamma \vdash e : \ell' \qquad \ell' \sqsubseteq \ell}{\Gamma \vdash e : \ell}$$

Fig. 8. Typing rules for expressions.