Reconciling progress-insensitive noninterference and declassification

Johan Bay
Aarhus University
bay@cs.au.dk

Aslan Askarov
Aarhus University
aslan@cs.au.dk

***Abstract*—Practitioners of secure information flow often face a design challenge: what is the right semantic treatment of leaks via termination? On the one hand, the potential harm of untrusted code calls for strong progress-sensitive security. On the other hand, when the code is trusted to not aggressively exploit termination channels, practical concerns, such as permissiveness of the enforcement, make a case for settling for weaker, progress-insensitive security. This binary situation, however, provides no suitable middle point for systems that mix trusted and untrusted code. This paper connects the two extremes by reframing progress-insensitivity as a particular form of declassification. Our novel semantic condition reconciles progress-insensitive security as a declassification bound on the so-called progress knowledge in an otherwise progress or timing sensitive setting. We show how the new condition can be soundly enforced using a mostly standard information-flow monitor. We believe that the connection established in this work will enable other applications of ideas from the literature on declassification to progress-insensitivity.**

I. Introduction

Progress-insensitive noninterference (PINI) is a popular semantic condition for secure information flow. PINI generalizes the classical termination-insensitive noninterference to accommodate I/O interactions and provides a practical foundation for many information flow systems. A known downside of PINI is that it permits leaking arbitrary amounts of information [6]. Malicious code may launder data through termination channels by unary encoding the information in the length of the trace or via timing channels. For these reasons, the consensus in the information flow community is to use PINI for trusted settings, where the goal is to prevent accidental information leaks. For untrusted settings, stronger notions of security, such as progress or timing sensitivity, are necessary.

Many practical scenarios, however, combine both trusted and untrusted code. Such combinations are natural to browser mashups, mobile apps, and just about any system that embeds third-party code. The binary consensus provides no suitable middle ground here. Progress-insensitivity is too permissive, whereas progress and timing-sensitivity is too restrictive.

Consider one such example scenario of a mashup that embeds a third-party newsfeed widget. The widget downloads the latest newsfeed from the news server and displays the favorite topic of the user. The choice of the favorite topic is sensitive and, therefore, must not leak to the news server. Figure 1 presents a pseudo-code for such a widget. The widget implements a custom caching logic by maintaining a counter and re-fetching the news

```
1 function newsWidget (userFavTopic) {
2   if (counter % 10 == 0) {
3     feed = receive (newsfeed_server_url)
4   }
5   counter ++;
6   newstext = feed[userFavTopic]
7 }
```

Fig. 1. Newsfeed widget code

on every tenth invocation. For the purpose of this example, we regard the counter as sensitive as well.

The code in Figure 1 is straightforward and unproblematic. We can imagine crafting a tool that analyzes (statically or dynamically) the code in Figure 1 for potential information flow violations. But if we are to take the next step and try to prove our tool sound, we hit a semantic conundrum. Because Line 3 contains a potentially blocking network operation, it is unclear how long it may take for the server to respond, if ever. This means that if we want our tool to accept programs such as Figure 1, we cannot use progress and timing-sensitive security as the basis for soundness. With the binary consensus, the only other option is progress-insensitive security. This option permits blocking and divergence, making it suitable for Figure 1. However, it also forces us to place the termination and timing attacks outside of the formal threat model, which weakens our tool.

This paper addresses the problem of the binary situation by presenting a novel semantic definition that connects the two extremes by reconciling progress-insensitive security as a particular form of declassification. This reframing means that we can treat progress-insensitivity just like any other declassification – a selective weakening of a baseline end-to-end security policy. It also means that we can transfer insights about declassification policies, such as their dimensions and principles [30], to progress-insensitivity. The key to the new definition is the use of the epistemic approach to information flow, which allows us to specify a bound on the knowledge the attacker learns from observing the progress of the computation in an otherwise progress or timing-sensitive setting.

Two meta-level points about our definition are worth highlighting. First, we note that the practice of declassifying termination leaks by itself is not novel. This idea appears in the literature as early as two decades ago in Jif [28] in the context of

DOI 10.1109/CSF49147.2020.00015

programming languages and later in HiStar [36] in the context of operating systems. Here, our work provides a firm theoretical basis that this practice lacked. In fact, we show that a mostly standard flow-insensitive dynamic monitor soundly enforces the new definition.

Second, we stress the value of the epistemic approach in formulating a concise and intuitive definition. It is not clear to us whether the definition can be reformulated in a classical two-trace style while retaining the same degree of clarity. The discussion of the soundness of our monitor presents an operational security invariant that does have the classical two-trace formulation, but that invariant is far from intuitive.

We present our condition in the setting of a simple imperative language with a standard flow-insensitive dynamic monitor, which conveys the condition in a clean form. The simple language does not contain networking or blocking primitives. This omission does not remove generality from our setup because the language already contains the possibility of divergence via infinite loops. We have implemented the enforcement of this condition in Troupe [12] – a research programming language with dynamic information flow control, actor-based concurrency, and primitives for distributed programming.

The rest of the paper is structured as follows: Section II introduces the formal setting of a small imperative language we use in this work. The presentation of the security condition is split across two sections. Section III presents the security for a progress-sensitive attacker and presents how a mostly standard dynamic monitor can soundly enforce this condition; Section IV presents the security condition for a timing-sensitive attacker. We discuss our definitions in Section V and report on the implementation experience in Section VI. Finally, in Sections VII and VIII we discuss related work and conclude.

II. The security model and the language

A. Security model

We assume a standard security lattice $\mathcal{L}$ of security levels ℓ, with distinguished bottom and top levels $\bot$ and $\top$, and the operations for least upper bound $\sqcup$ and the lattice order $\sqsubseteq$.

Our language is a standard imperative language extended with capability-based declassification, and a special purpose tini command for bounded progress-insensitivity that we explain below. Each variable in the program has a fixed security level $lev(x)$ that does not change throughout the execution. An attacker associated with a security level ℓ observes updates to variables with levels up to ℓ; they additionally observe the reachability of the tini blocks, as we explain below.

In the examples we show here, we use a two or three-level lattice with levels L, M, H, where $L \sqsubseteq M \sqsubseteq H$, and $\ell \sqsubseteq \ell$ for each $\ell \in \{L, M, H\}$. We adopt the convention of using upper-case letters to denote concrete lattice elements of $\mathcal{L}$ and lower-case letters to denote variables of said level. As such, h_1 and h_2 are variables such that $lev(h_1) = lev(h_2) = H$.

$$
\begin{aligned}
e &::= n \mid x \mid e \ op \ e \mid \mathsf{attenuate}\ e\ \mathsf{to}\ (\ell, p) \\
c &::= \mathsf{skip} \mid c; c \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \\
&\quad \mid x = e \mid \mathsf{tini}_\eta\ \mathsf{to}\ \ell\ \mathsf{with}\ e\ \mathsf{do}\ c \\
&\quad \mid x = \mathsf{decl}\ e\ \mathsf{to}\ \ell\ \mathsf{with}\ e \\
&\quad \mid \mathsf{eval}\ e\ \{x_1, \ldots, x_n\}
\end{aligned}
$$

Fig. 2. Syntax of the language

B. The language and the monitoring semantics

Figure 2 presents the syntax of our language. We explain the formal semantics of the language and then discuss the non-standard features.

a) Monitoring semantics: For evaluating commands we use a small-step semantics transition $\langle c, m, pc\rangle \longrightarrow_\alpha \langle c', m', pc'\rangle$, where pc is the security level of the program counter, and α is the event generated by the step. The events can be empty events, denoted by ϵ, and assignments and declassifications per the following grammar:

$$\alpha ::= \epsilon \mid a(x, v) \mid d(x, \ell, \ell) \mid \bar{t}_\eta(\ell, \ell)$$

The stop and pcdecl commands are only used internally, and therefore not part of the syntax of the language. Command stop denotes final configurations that cannot step any further. For evaluating expressions we use a big-step relation $\langle e, m\rangle \Downarrow \langle base; \ell\rangle$ that relates an expression with a labeled value. Labeled values $\langle base; \ell\rangle$ consists of a base value and a level, where ℓ denotes the confidentiality-level of the base value $base$. Base values include integers n, strings s, and authority values $\mathsf{auth}\ \ell\ p$. In our semantics, we denote the base type (integer, string, or authority) of a base value $base$ as $type(base)$, and we furthermore assign a predetermined type for each program variable such that $type(x)$ denotes the type of variable x. The types of variables are static and cannot be changed during the execution. Fig. 3 presents the rules for expression evaluation and Fig. 4 presents the command evaluation rules for our language. Note how a tini statement reduces to the sequential composition of its argument and a special pcdecl command. The syntactic structure imposed by the tini blocks ensures that the use of pcdecl is always well-bracketed since the pcdecl-command is not part of the surface language. At runtime, the expanded pcdecl-commands exhibit a stack-like behavior reminiscent of pc-stacks in other monitor designs from the literature.

The monitor is inherently progress-sensitive: barring any pcdecl commands, the pc never goes down during the execution. A reader familiar with the literature on information flow monitors may spot deficiencies in the monitor's precision – for example, it rejects program $(\mathsf{if}\ h\ \mathsf{then}\ \mathsf{skip}\ \mathsf{else}\ \mathsf{skip}); l = 0$. This simple monitor is picked for the purpose of exposition to allow us to focus on the presentation of the security condition and the soundness proof in Section III. We further note that while it is possible to add extra precision to this monitor, unlike progress-insensitive monitors that benefit from hybrid analysis, it is difficult to avoid pc creep in progress-sensitive monitors.

$$\frac{}{\langle base, m\rangle \Downarrow \langle base; \bot\rangle} \qquad \frac{m(x) = base}{\langle x, m\rangle \Downarrow \langle base; lev(x)\rangle}$$

$$\frac{\langle e_1, m\rangle \Downarrow \langle base_1; \ell_1\rangle \quad \langle e_2, m\rangle \Downarrow \langle base_2; \ell_2\rangle \quad type(base_1) = type(base_2) \quad base = base_1 \oplus base_2}{\langle e_1 \oplus e_2, m\rangle \Downarrow \langle base; \ell_1 \sqcup \ell_2\rangle}$$

$$\frac{\langle e_1, m\rangle \Downarrow \langle \mathsf{auth}\ \ell_{auth_1}\ p_1; \ell\rangle \quad \ell_{auth_2} \sqsubseteq \ell_{auth_1} \quad p_2 \leq p_1}{\langle \mathsf{attenuate}\ e_1\ \mathsf{to}\ (\ell_{auth_2}, p_2), m\rangle \Downarrow \langle \mathsf{auth}\ \ell_{auth_2}\ p; \ell\rangle}$$

Fig. 3. Semantics of evaluating expressions

b) Declassifications: Our language has two different constructs for downgrading: one for downgrading values (decl), and one for downgrading the termination of a region of the program (tini). We include two constructs to highlight differences and parallels between the two kinds of declassifications. Both constructs reveal information by design, but in different ways. Whereas declassification is a way for the programmer to indicate that an otherwise secret value is public, the tini constructs allows the programmer to indicate that a program block (identified by a unique tag η) should be treated in a progress-insensitive way, which means that the information about the termination of the block is public. In the jargon of information flow control systems, this exactly amounts to lowering the pc-label at the end of the block.

c) Authority: Our language restricts the use of declassifications via a capability-like mechanism that we refer to as *authority* [28]. Given a value at level ℓ_{from}, an authority of level ℓ_{auth} permits a declassification to level ℓ_{to} if $\ell_{from} \sqsubseteq \ell_{to} \sqcup \ell_{auth}$. At run-time, an authority value $\mathsf{auth}\ \ell\ p$ consists of an authority level ℓ and a purpose bit p. The purpose bit 1 means that the authority can be used for general purpose declassification, while the purpose bit 0 means that the authority can only be used for tini-statements. For example, assuming that variable $auth_M$ contains the value $\mathsf{auth}\ M\ 1$, the language allows the declassification

$$l = \mathsf{decl}\ m\ \mathsf{to}\ L\ \mathsf{with}\ auth_M$$

but not

$$l = \mathsf{decl}\ h\ \mathsf{to}\ L\ \mathsf{with}\ auth_M$$

d) Attenuate and running untrusted code: The only way to create an authority value in the language is by attenuation of another authority value. Initially, the special variable rootauth contains the full authority $\mathsf{auth}\ \top\ 1$. Our language contains primitives for restricting the access, level, and purpose of authority, namely attenuate and eval.

For example, $\langle \mathsf{attenuate\ rootauth\ to}\ (M, 0), m\rangle$ evaluates to a value $\langle \mathsf{auth}\ M\ 0; \bot\rangle$ that can only be used for declassifying progress up to level M. For running untrusted code, we provide an eval command that takes a string s and a set of variables $\{x_1, \ldots, x_n\}$. The semantics of eval is, that it parses the string to a command c (denoted $c = parse(s)$) under the condition that c is only allowed to use variables explicitly mentioned in $\{x_1, \ldots, x_n\}$ and must not contain nested evals. In this way, our eval-command can be seen as a "poor man's"-scoping, which we capture in the following Lemma:

Lemma 1 (eval memory safety)**.** *Suppose* $\langle \mathsf{eval}\ e\ X, m, pc\rangle \longrightarrow_t^* \langle c', m', pc'\rangle$. *Then it holds for all* s *where* $x \in X \implies m(x) = s(x)$ *that*

$$\langle \mathsf{eval}\ e\ X, s, pc\rangle \longrightarrow_t^* \langle c', s', pc'\rangle$$

and

$$x \in X \implies m'(x) = s'(x)$$

Proof. By induction in the program resulting from $parse(s)$ using that no variables except those occurring in X is used. □

The combination of eval and attenuate allows us to attenuate the root-authority by storing it in some variable, e.g., x, and run untrusted code while only permitting access to x. For example, we may restrict declassifications in the evaluation of the command stored in variable m_{code} up to level M as follows.

$$auth_M = \mathsf{attenuate\ rootauth\ to}\ (M, 1);$$
$$\mathsf{eval}\ m_{code}\ \{auth_M, l_1, l_2, m_1, m_2, h_1, h_2\}$$

Note that the program in m_{code} may access high variables h_1 and h_2 but cannot declassify them since it does not have access to sufficient authority.

e) tini*-blocks:* The tini-construct allows us to embed progress-insensitive code in an otherwise progress-sensitive setting. To give some intuition about the tini-construct, suppose we have the following program that loops if a variable of level H is positive; or makes an assignment at level L otherwise:

$$\mathsf{while}\ h > 0\ \mathsf{do\ skip}$$
$$l = 0$$

This program is acceptable in a progress-insensitive setting, but is rejected by progress-sensitive security conditions, since the assignments to l leaks information about the reachability of the join-point. The tini construct allows us to embed such code in a progress-sensitive setting by explicitly declassifying the reachability of the end of the block. Just like regular declassification, the tini-block also requires an authority argument. Hence, the example above can be written instead as:

$$\mathsf{tini}_\eta\ \mathsf{to}\ L\ \mathsf{with\ rootauth\ do}$$
$$\quad \mathsf{while}\ h > 0\ \mathsf{do\ skip};$$
$$l = 0$$

The design of the tini block is inspired by similar constructs in large-scale information flow systems: Jif [28] implements pc-declassification by a single command for declassifying the pc-label although the syntax does not limit the scope of the progress that is declassified. HiStar [36] implements a similar thing through "untainting" gates that can be restricted to only untaint the control flow.

Attenuation of the purpose can be used in conjunction with eval and the tini block. Revisiting the news widget example from

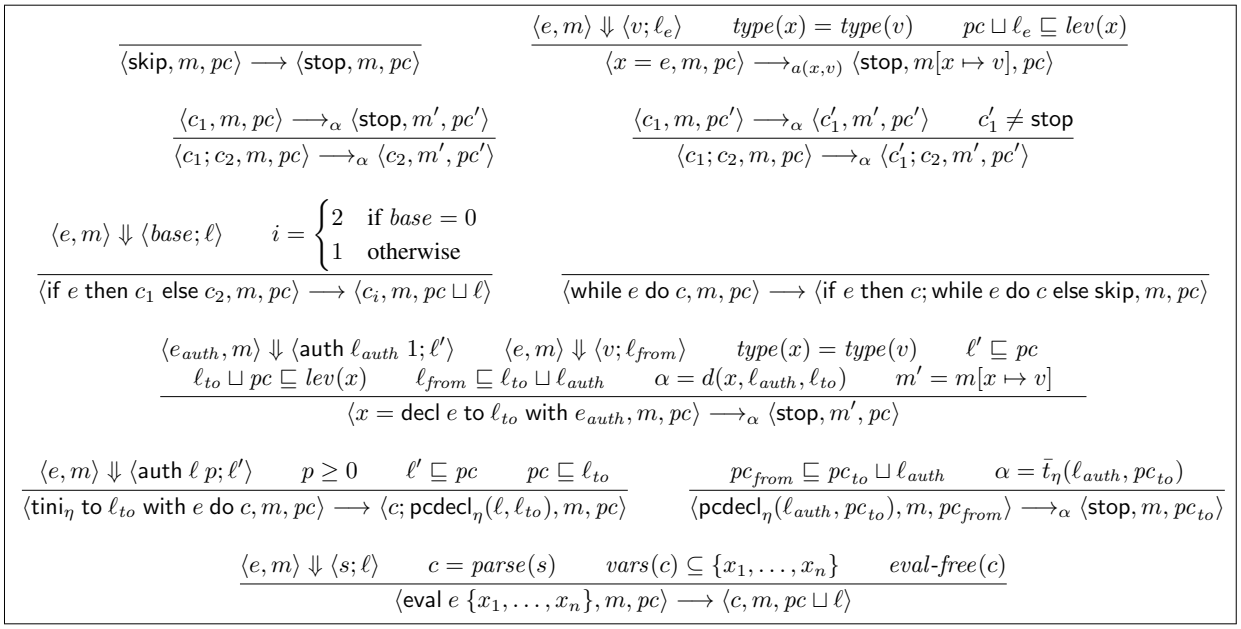

Fig. 4. Monitored operational semantics

Section I, the trusted code may evaluate the widget by passing it access to an attenuated authority. To bring the example closer to the language we have presented, we let receive fetch the untrusted widget code from a network connection and run it by using eval:

$$
\begin{array}{l}
\mathit{untrustedWidget} = \mathsf{receive}\ \mathit{newsfeed\_server\_url};\\
\mathit{userFavTopic} = \text{"}\mathsf{Politics}\text{"};\\
\mathit{authNews} = \mathsf{attenuate}\ \mathsf{rootauth}\ \mathsf{to}\ (\mathit{newslev}, 0);\\
\mathsf{tini}_\eta\ \mathsf{to}\ \bot\ \mathsf{with}\ \mathit{authNews}\ \mathsf{do}\\
\quad \mathsf{eval}\ \mathit{untrustedWidget}\ \{\mathit{userFavTopic}\}
\end{array}
$$

III. Security condition

This section presents a security definition for embedding tini-blocks when the baseline security is progress-sensitive.

A. Auxiliary definitions

We use the knowledge-based [5] approach to define our security condition. The high-level idea behind the approach is that we consider an attacker that can observe the execution of the program and define the knowledge that such attacker obtains as the set of memories that are consistent with seeing the execution up to this point. The security condition is defined as a bound on how much the knowledge is allowed to change at each step of the execution.

To define such bounds, we first define what it means for memories to be equivalent and define which execution steps are visible to the adversary.

In the following, we write $m \sim_\ell s$ to denote that two memories are equal up to ℓ (Definition 1 below), and $\lfloor t \rfloor_\ell$ to denote a filtering of the trace t that only includes the events that are observable at level ℓ (Definition 2 below).

Definition 1 (Memory equivalence)**.** *Two memories m and s are equivalent up to level ℓ, written $m \sim_\ell s$, if $\mathsf{dom}(m) = \mathsf{dom}(s)$ and it holds that for all $x \in \mathsf{dom}(m)$,*

$$lev(x) \sqsubseteq \ell \implies m(x) = s(x)$$

We define level of an event, denoted $lev(\alpha)$, as the level of the updated variable for assignment and declassify events, level ℓ_{to} for tini events $\bar{t}_\eta(\ell, \ell_{to})$, and $\top$ otherwise:

$$
\begin{aligned}
lev(\epsilon) &= \top\\
lev(a(x, \_)) &= lev(x)\\
lev(d(x, \_, \_)\ |) &= lev(x)\\
lev(\bar{t}_\eta(\_, \ell_{to})) &= \ell_{to}
\end{aligned}
$$

Definition 2 (Trace filtering)**.** *The filtering of a trace t at level ℓ written $\lfloor t \rfloor_\ell$ is defined as*

$$
\begin{aligned}
\lfloor [] \rfloor_\ell &= []\\
\lfloor t' \cdot \alpha \rfloor_\ell &= \begin{cases} \lfloor t' \rfloor_\ell \cdot \alpha & \textit{if } lev(\alpha) \sqsubseteq \ell \\ \lfloor t' \rfloor_\ell & \textit{otherwise} \end{cases}
\end{aligned}
$$

We use the above to define two technical definitions of knowledge. First, we define attacker knowledge which defines the knowledge of an attacker observing a trace t.

Definition 3 (Attacker knowledge [3])**.** *Given a program c, initial memory m, initial program counter level pc, such that $\langle c, m, pc\rangle \longrightarrow_t^* \langle c', m', pc'\rangle$, define* attacker knowledge at

level ℓ_{adv} *to be the set of memories* m' *that are consistent with the observations of the adversary:*

$$k(c, m, t, \ell_{adv}) \triangleq \{m' \mid m \sim_{\ell_{adv}} m' \wedge \langle c, m', pc\rangle \longrightarrow^*_{t'} \langle c'', m'', pc''\rangle \wedge \lfloor t'\rfloor_{\ell_{adv}} = \lfloor t\rfloor_{\ell_{adv}}\}$$

We can now use this definition as a building block for defining security conditions. We can, for example, define progress-sensitive noninterference as follows:

Definition 4 (Progress-sensitive noninterference)**.** *Given a program* c*, initial memory* m *and initial program counter label* pc *such that*

$$\langle c, m, pc\rangle \longrightarrow^*_{t\cdot\alpha} \langle c', m', pc'\rangle$$

the run satisfies progress-sensitive noninterference *if it holds that for all* ℓ_{adv}*, if* $lev(\alpha) \sqsubseteq \ell_{adv}$ *then*

$$k(c, m, t\cdot\alpha, \ell_{adv}) \supseteq k(c, m, t, \ell_{adv})$$

Note how this definition bounds the knowledge from seing $t\cdot\alpha$ with the knowledge of seeing t. This essentially means that all the memories that the attacker considered possible when seeing t are still considered possible after also observing the event α. Note that his is a very strong security condition. To define more lenient conditions, we use another building block: the progress knowledge.

Definition 5 (Progress knowledge [4])**.** *Given a program* c*, initial memory* m*, initial program counter level* pc*, such that* $\langle c, m, pc\rangle \longrightarrow^*_t \langle c', m', pc'\rangle$*, define* progress knowledge at level ℓ_{adv} *to be the set of memories* m' *that are consistent with the knowledge up to* t *followed further by one more event:*

$$k_{\rightarrow}(c, m, t, \ell_{adv}) \triangleq \{m' \mid m \sim_{\ell_{adv}} m' \wedge \langle c, m', pc\rangle \longrightarrow^*_{t'} \langle c'', m'', pc''\rangle \wedge \lfloor t'\rfloor_{\ell_{adv}} = \lfloor t\rfloor_{\ell_{adv}} \cdot \alpha\}$$

The above allows us to express the standard progress-insensitive noninterference:

Definition 6 (Progress-insensitive noninterference)**.** *Given a program* c*, initial memory* m *and initial program counter label* pc *such that*

$$\langle c, m, pc\rangle \longrightarrow^*_{t\cdot\alpha} \langle c', m', pc'\rangle$$

the run satisfies progress-insensitive noninterference *if it holds that for all* ℓ_{adv}*, if* $lev(\alpha) \sqsubseteq \ell_{adv}$ *then*

$$k(c, m, t\cdot\alpha, \ell_{adv}) \supseteq k_{\rightarrow}(c, m, t, \ell_{adv})$$

Here, the knowledge of an attacker that observes $t\cdot\alpha$ is bounded by the progress knowledge from seeing just t. This exactly captures that the attacker is allowed to rule out the the memories that do not make progress.

B. Progress-sensitive security with declassification and locally-bound progress-insensitivity

Armed with the above definitions, we define our main security condition as follows.

Definition 7 (Progress-sensitive security with declassification and locally-bound progress-insensitivity)**.** *Given a program* c*, initial memory* m *and initial program counter label* pc *such that*

$$\langle c, m, pc\rangle \longrightarrow^*_{t\cdot\alpha} \langle c', m', pc'\rangle$$

define the run as secure *if it holds that for all* ℓ_{adv}*, if* $lev(\alpha) \sqsubseteq \ell_{adv}$ *then*

1) *if* $\alpha = d(\_, \ell_{auth}, \ell_{to})$ *then it should hold that:*
   a) $k_{\rightarrow}(c, m, t, \ell_{adv}) \supseteq k(c, m, t, \ell_{adv})$*, and*
   b) $k(c, m, t\cdot\alpha, \ell_{adv}) \supseteq k(c, m, t, \ell_{auth} \sqcup \ell_{adv})$
2) *if* $\alpha = \bar{t}_\eta(\ell_{auth}, \ell_{to})$ *then it should hold that:*
   a) $k(c, m, t\cdot\alpha, \ell_{adv}) \supseteq k_{\rightarrow}(c, m, t, \ell_{adv})$
   b) $k_{\rightarrow}(c, m, t, \ell_{adv}) \supseteq k(c, m, t, \ell_{auth} \sqcup \ell_{adv})$
3) *otherwise, it should hold that:*

$$k(c, m, t\cdot\alpha, \ell_{adv}) \supseteq k(c, m, t, \ell_{adv})$$

The security condition specifies what information the attacker may learn from observing the program events. The baseline of progress-sensitive security is captured in item 3 of the definition stating that the attacker learns nothing from non-declassify events. This rules out many standard examples of direct and indirect flows, as well as the termination leaks such as

$$l = 0; (\mathsf{while}\ h > 0\ \mathsf{do\ skip}); l = 1$$

The other two items weaken the baseline as follows. For declassifications (item 1) we have two clauses: Clause 1a says that reachability of the declassification conveys no knowledge to the attacker. Observe that this is expressed as a bound on the progress knowledge! This clause rules out programs such as

$$l = 0; (\mathsf{while}\ h > 0\ \mathsf{do\ skip}); l = \mathsf{decl}\ h\ \mathsf{to}\ L\ \mathsf{with}\ auth_H$$

that leak via termination without a tini-statement.

Clause 1b specifies an upper bound on the information the attacker learns from the event to be no more the knowledge at level $\ell_{auth} \sqcup \ell_{adv}$ before the event. This clause has a flavor of language-based intransitive noninterference [23], because it does not otherwise bound what information from the permitted level is declassified. For example, assuming $auth_M$ and $auth_H$ are authorities with purpose bit one, this definition accepts the program

$$m = \mathsf{decl}\ h\ \mathsf{to}\ M\ \mathsf{with}\ auth_H;$$
$$l = \mathsf{decl}\ m\ \mathsf{to}\ L\ \mathsf{with}\ auth_M$$

Both declassifications above are allowed. At the time of the second declassification, the adversary at L learns the original value of h despite only using the authority of M. This is accepted because the earlier declassification of h to m happened with sufficient authority.

Clause 1b does not regulate exactly *what* information from the level of $\ell_{auth} \sqcup \ell_{adv}$ may be declassified; however, prior work on using knowledge-based conditions for further constraining what and where to declassify can be easily applied here in an orthogonal manner [4], [16].

For tini-events (item 2), we also have two constraints. The first constraint corresponds to standard progress-insensitive noninterference [6]: knowledge of the event must reveal no more than knowledge of the event's existence. The second constraint is interesting, because it specifies an upper bound on the information leaked by the termination to be no more than the knowledge at level $\ell_{auth} \sqcup \ell_{adv}$ before the event. This is again expressed as a bound on progress knowledge. This clause rules out programs with insufficient authority for the pc-declassification such as

$$l = 0; (\mathsf{tini}_{\eta} \text{ to } L \text{ with } auth_M \text{ do while } h > 0 \text{ do skip}); l = 1$$

The definition accepts programs that use tini blocks as long as the authority for the pc-declassification is sufficient. This includes nested tini blocks. The following program is accepted.

$l = 0;$
tini_{η_1} to L with $auth_M$ do {
  if $m > 0$ then
    tini_{η_2} to M with $auth_H$ do
      while $h > 0$ do skip
  else skip };
$l = 1$

C. A note on the design of item 2

For the simple language of this section, the two clauses of item 2 can be simplified to require that for $\alpha = \bar{t}_{\eta}(\ell_{auth}, \ell_{to})$ it must hold that

$$k(c, m, t \cdot \alpha, \ell_{adv}) \supseteq k(c, m, t, \ell_{auth} \sqcup \ell_{adv})$$

We opted to present the definition without this simplification, because in more realistic settings, this simplification is dangerous and leads to occlusion.

The simplification is possible in our language, because pcdecl events are attacker-observable and convey little information other than their reachability, thanks to syntactically enforced well-bracketedness of tini/pcdecl commands.[1]

However, in reality, it may be unfair to assume that attacker observes internal events such as pcdecl. Suppose indeed that pcdecl has no manifestation in the attacker-observable projection of the trace. How would we need to change Definition 7 to accommodate this? One option is to rephrase item 2 of Definition 7 so that event α refers to *the first observable event after executing* pcdecl. But such events can communicate more than a unit of information, as in the program below.

tini to L with rootauth do {skip}
if $h > 0$ then $l = 0$ else $l = 1$

[1]These conveniences help us minimize technical clutter in the paper.

$$\frac{cfg \longrightarrow_{\alpha} cfg' \qquad lev(\alpha) \sqsubseteq \ell}{cfg \curvearrowright_{\alpha}^{0,\ell} cfg'}$$

$$\frac{cfg \longrightarrow_{\beta} \langle \mathsf{stop}, m, pc \rangle \qquad lev(\beta) \not\sqsubseteq \ell}{cfg \curvearrowright_{\beta}^{0,\ell} \langle \mathsf{stop}, m, pc \rangle}$$

$$\frac{cfg_1 \longrightarrow_{\beta} cfg_2 \qquad lev(\beta) \not\sqsubseteq \ell \qquad cfg_2 \curvearrowright_{\alpha}^{n,\ell} cfg_3}{cfg_1 \curvearrowright_{\alpha}^{n+1,\ell} cfg_3}$$

Fig. 5. Bridge-step relation

This program would reduce to

$$\mathsf{pcdecl}(\mathsf{rootauth}, L); \mathsf{if}\ h > 0\ \mathsf{then}\ l = 0\ \mathsf{else}\ l = 1$$

Here, the first event after pcdecl is one of the low assignments. The approach of the simplified definition accepts this program because it mistakenly applies the declassification condition to reveal the choice of the high branch. On the other hand, the two-clause approach that explicitly constraints the progress knowledge rejects this program.

D. Soundness of the enforcement

Next, we formally connect the monitoring semantics of Section II with Definition 7. We do this by showing the following statement:

Theorem 1 (Soundness of the monitoring semantics)**.** *Given a program c, memory m, and level pc then all runs $\langle c, m, pc \rangle \longrightarrow_t^* \langle c', m', pc' \rangle$ satisfy Definition 7.*

To get some intuition about the proof, let us think how classical noninterference proofs usually proceed. The security invariant of such proofs boils down to the reasoning along the lines of "a pair of low-equivalent configurations that each emit attacker-observable events transition to low-equivalent configurations plus the attacker cannot discriminate between the two events." Note how low-equivalence is used in both the precondition and the post-condition of such a statement. For declassification, we need to weaken the invariant, which is typically done by strengthening the precondition to relate fewer configurations. Set-theoretically, this strengthening corresponds to picking a relation that is smaller than low-equivalence. Exactly how small is an important design criterion that is dictated by the top-level security requirement such as our Definition 7. One challenge that we have encountered in the proof is finding the right equivalence relation for the precondition that is compositional in the applications of the inductive hypothesis. Our solution to this challenge is to engineer relations that are smaller than low-equivalence, subject to additional constraints we explain below.

First, we define an auxiliary relation that characterizes the intuition of "configuration emitting an attacker-observable event." We call this relation *bridge-step*. Operationally it is defined as a relation between two configurations where the first configuration reaches the second one by taking n intermediate "secret" steps

$$\frac{\langle c,m,pc\rangle \curvearrowright_{\alpha}^{n,\ell} \langle c',m',pc'\rangle \qquad \langle c,s,pc\rangle \curvearrowright_{\alpha}^{n',\ell} \langle c',s',pc'\rangle}{\langle c,m \mid s,pc\rangle \Rightarrow_{\alpha}^{\ell} \langle c',m' \mid s',pc'\rangle}$$

$$\frac{k>1 \qquad \langle c,m \mid s,pc\rangle \Rightarrow_{\alpha_1}^{\ell} \langle c',m' \mid s',pc'\rangle \quad \langle c',m' \mid s',pc'\rangle \Rightarrow_{\alpha_2\ldots\alpha_k}^{\ell} \langle c'',m'' \mid s'',pc''\rangle}{\langle c,m \mid s,pc\rangle \Rightarrow_{\alpha_1\ldots\alpha_k}^{\ell} \langle c'',m'' \mid s'',pc''\rangle}$$

Fig. 6. Synchronized bridging

(without producing any observable events) and then either emits an observable step or terminates. This relation is shown in Fig. 5. The security intuition behind the bridge relation is that the attacker only observes the configurations related by the bridge relation. Hence, we formulate our security invariant around that relation.

We furthermore define *indistinguishability restriction* $\langle I\rangle^{c,pc}_{\ell|\alpha_1\ldots\alpha_k}$ as the restriction of the relation I to only contain all pairs of the memories that can emit $\alpha_1\ldots\alpha_k$ (in that order) when evaluating c using initial program-counter pc. To formally define $\langle I\rangle^{c,pc}_{\ell|\alpha_1\ldots\alpha_k}$, we introduce another auxiliary definition that synchronizes two bridge-step runs on a list of events. The synchronized bridge has the effect of demanding that two runs proceed in lock-step w.r.t to their individual bridge-steps. The rules for synchronized bridging can be seen in Fig. 6.

We can now define indistinguishability restriction as per Definition 8 below.

Definition 8 (Indistinguishability restriction $\langle I\rangle^{c,pc}_{\ell|\alpha}$)**.** *Consider a potentially empty sequence of events $\alpha_1\ldots\alpha_k$. Define the relation $m\ \langle I\rangle^{c,pc}_{\ell|\alpha_1\ldots\alpha_k}\ s$ as follows:*

$$\frac{m\ I\ s}{m\ \langle I\rangle^{c,pc}_{\ell|\mathsf{nil}}\ s}$$

$$\frac{m\ I\ s \qquad \langle c,m \mid s,pc\rangle \Rightarrow_{\alpha_1\ldots\alpha_k}^{\ell} \langle c',m' \mid s',pc''\rangle}{m\ \langle I\rangle^{c,pc}_{\ell|\alpha_1\ldots\alpha_k}\ s}$$

With all this auxiliary infrastructure, we now state the operational definition of security.

Lemma 2 (Security for monitored evaluations)**.** *Suppose $\langle c,m,pc\rangle \curvearrowright_{\alpha}^{n,\ell_{adv}} \langle c',m',pc'\rangle$. Then the following holds:*

1) ***if*** $\alpha = d(x,\ell_{auth},\ell_{to})$ ***and*** $lev(x)\sqsubseteq\ell_{adv}$***:***
   Let
   $$I = \langle{\sim_{\ell_{auth}\sqcup\ell_{adv}}}\rangle^{c,pc}_{\ell_{auth}\sqcup\ell_{adv}|\beta_1,\ldots,\beta_j}$$
   where
   $$\langle c,m,pc\rangle \curvearrowright_{\beta_1}^{i_1,\ell_{auth}\sqcup\ell_{adv}} \langle c_1,m_1,pc_1\rangle \curvearrowright_{\beta_2}^{i_2,\ell_{auth}\sqcup\ell_{adv}} \ldots \curvearrowright_{\beta_j}^{i_j,\ell_{auth}\sqcup\ell_{adv}} \langle c_j,m_j,pc_j\rangle$$
   such that
   $$\langle c_j,m_j,pc_j\rangle \curvearrowright_{\alpha}^{i',\ell_{auth}\sqcup\ell_{adv}} \langle c',m',pc'\rangle$$
   then it must hold that for all s where $m\ I\ s$ there exists α' such that
   $$\langle c,s,pc\rangle \curvearrowright_{\alpha'}^{n',\ell_{adv}} \langle c',s',pc'\rangle$$
   and $m'\sim_{\ell_{adv}} s'$.
2) ***if*** $\alpha = \bar{t}_{\eta}(\ell_{auth},\ell_{to})$ ***and*** $\ell_{to}\sqsubseteq\ell_{adv}$***:***
   Let
   $$I = \langle{\sim_{\ell_{auth}\sqcup\ell_{adv}}}\rangle^{c,pc}_{\ell_{auth}\sqcup\ell_{adv}|\beta_1,\ldots,\beta_j}$$
   where
   $$\langle c,m,pc\rangle \curvearrowright_{\beta_1}^{i_1,\ell_{auth}\sqcup\ell_{adv}} \langle c_1,m_1,pc_1\rangle \curvearrowright_{\beta_2}^{i_2,\ell_{auth}\sqcup\ell_{adv}} \ldots \curvearrowright_{\beta_j}^{i_j,\ell_{auth}\sqcup\ell_{adv}} \langle c_j,m_j,pc_j\rangle$$
   such that
   $$\langle c_j,m_j,pc_j\rangle \curvearrowright_{\alpha}^{i',\ell_{auth}\sqcup\ell_{adv}} \langle c',m',pc'\rangle$$
   then it must hold that for all s where $m\ I\ s$ there exists α' such that
   $$\langle c,s,pc\rangle \curvearrowright_{\alpha'}^{n',\ell_{adv}} \langle c',s',pc'\rangle$$
   and $m'\sim_{\ell_{adv}} s'$.
3) ***if*** $\alpha\neq\bar{t}\_(\_,\_)$ ***and*** $pc'\sqsubseteq\ell_{adv}$***:***
   For all s where $m\sim_{\ell_{adv}} s$ it holds that
   $$\langle c,s,pc\rangle \curvearrowright_{\alpha}^{n',\ell_{adv}} \langle c',s',pc'\rangle$$
   and if α is not a declassify event $d(x,\_,\_)$ where $lev(x)\sqsubseteq\ell_{adv}$ then $m'\sim_{\ell_{adv}} s'$.
4) ***if*** $\alpha=\bar{t}_{\eta}(\_,\_)$ ***or*** $pc'\not\sqsubseteq\ell_{adv}$***:***
   It holds that for all s where $m\sim_{\ell_{adv}} s$,
   $$\begin{aligned}\langle c,s,pc\rangle \curvearrowright_{\alpha'}^{n',\ell_{adv}} \langle c'',s',pc''\rangle \implies\ & m'\sim_{\ell_{adv}} s' \wedge c' = c'' \\ \wedge\ pc'\not\sqsubseteq\ell_{adv} \implies\ & (pc''\not\sqsubseteq\ell_{adv}\wedge c' = \mathsf{stop}) \\ \wedge\ pc'\sqsubseteq\ell_{adv} \implies\ & (pc''\sqsubseteq\ell_{adv}\wedge\alpha=\alpha')\end{aligned}$$

The indistinguishability restriction of $\sim_{\ell}$, which we alluded to earlier, appears in two out of four sub-cases of the invariant. This is crucial in the proof when showing the clauses related to declassify-events, since this allows us to account for earlier unobservable declassifies that might become observable through the latest event. For example, suppose we have an attacker at level L and that we earlier on declassified a value v from H to M. Now if we later declassify that same value from M to L, it is not enough to only assume that the initial memories satisfy memory-equivalence up to M to prove the clause for declassify. Instead, we "replay" the trace at a higher attacker-level – in this case M – which reveals the events that are otherwise only observable at this higher level – such as declassifications from H to M. We use these events to synchronize the memories, and then conclude that the two runs must declassify the same value. This is exactly what the indistinguishability restriction condition $\langle{\sim_{\ell_{auth}\sqcup\ell_{to}}}\rangle^{c,pc}_{\ell_{auth}\sqcup\ell_{to}|\alpha_1,\ldots,\alpha_k}$ provides. The events $\alpha_1,\ldots,\alpha_k$ here range over the M-level events including H to M declassifications; none of these events are typically observable by L.

The detailed proof of Lemma 2 can be found in the extended version of this paper [11], where we also prove Theorem 1 by showing that runs satisfying Lemma 2 satisfy Definition 7.

IV. Timing Sensitivity

The security condition that we present for progress-sensitive noninterference in Definition 7 can be naturally strengthened to also cover timing-sensitive noninterference. The cautious reader might have noticed already that the monitor we present in our language is actually already enforcing this stronger notion of noninterference. As an example, the program

$$\begin{array}{l}\mathsf{if}\ h > 0\\ \quad \mathsf{then\ skip}\\ \quad \mathsf{else\ skip; skip; skip;}\\ l = 0\end{array}$$

is accepted by our progress-sensitive security condition, but is not allowed by our monitor. This shows that our monitoring leaves room for strengthening the security condition so that examples like above are also rejected by the definition.

To formalize this observation, we add a clock ts to our configurations $\langle c, m, pc \mid ts\rangle$ and timestamps to events (ts, α) such that our evaluation steps are now defined by the following rule

$$\frac{\langle c, m, pc\rangle \longrightarrow_{\alpha} \langle c', m', pc'\rangle}{\langle c, m, pc \mid ts\rangle \longrightarrow_{(ts+1,\alpha)} \langle c', m', pc' \mid ts+1\rangle}$$

We extend the definition of when events are observable in the obvious way: a timestamped event (ts, α) is observable at level ℓ if α is observable at level ℓ. The definitions of attacker knowledge and progress knowledge from the previous section are also ported to the new setting in a straightforward manner, noting that the initial clock value is 0. However, we need a new knowledge combinator, that we dub *clock knowledge*.

Definition 9 (Clock knowledge)**.** *Given a program c, initial memory m, initial program counter level pc, and initial timestamp ts such that $\langle c, m, pc \mid 0\rangle \longrightarrow_t^* \langle c', m', pc' \mid ts'\rangle$, define* clock knowledge at level ℓ_{adv} *to be the set of memories m' that are consistent with the knowledge up to t followed further by one more event with timestamp ts:*

$$\begin{aligned} k_{\rightarrow}^{\textcircled{1}}(c, m, t, \ell_{adv}, ts) \triangleq \{m' \mid\ & m \sim_{\ell_{adv}} m'\\ & \wedge \langle c, m', pc \mid 0\rangle \longrightarrow_{t'}^* \langle c'', m'', pc'' \mid ts\rangle\\ & \wedge \lfloor t'\rfloor_{\ell_{adv}} = \lfloor t\rfloor_{\ell_{adv}} \cdot (ts, \alpha)\}\end{aligned}$$

Observe that the clock knowledge, the progress knowledge, and the attacker knowledge are related by their definitions as follows: $k(c, m, t, \ell_{adv}) \supseteq k_{\rightarrow}(c, m, t, \ell_{adv}) \supseteq k_{\rightarrow}^{\textcircled{1}}(c, m, t, \ell_{adv}, ts')$.

We can now give a more precise top-level security condition:

Definition 10 (Timing-sensitive security with declassification and locally-bound progress-insensitivity)**.** *Given a program c, initial memory m, initial program counter label pc, and initial clock ts such that*

$$\langle c, m, pc \mid ts\rangle \longrightarrow_{t\cdot(ts',\alpha)}^* \langle c', m', pc' \mid ts'\rangle$$

define the run as secure *if it holds that for all ℓ_{adv}, if $lev(\alpha) \sqsubseteq \ell_{adv}$ then*

1) if $\alpha = d(\_, \ell_{auth}, \ell_{to})$ then it should hold that:
   a) $k_{\rightarrow}^{\textcircled{1}}(c, m, t, \ell_{adv}, ts') \supseteq k_{\rightarrow}(c, m, t, \ell_{adv}) \supseteq k(c, m, t, \ell_{adv})$
   b) $k(c, m, t \cdot (ts', \alpha), \ell_{adv}) \supseteq k(c, m, t, \ell_{auth} \sqcup \ell_{adv})$
2) if $\alpha = \bar{t}_{\eta}(\_, \_)$ then it should hold that:
   a) $k(c, m, t \cdot (ts', \alpha), \ell_{adv}) \supseteq k_{\rightarrow}^{\textcircled{1}}(c, m, t, \ell_{adv}, ts')$
   b) $k_{\rightarrow}^{\textcircled{1}}(c, m, t, \ell_{adv}, ts') \supseteq k(c, m, t, \ell_{auth} \sqcup \ell_{adv})$
3) otherwise it should hold that:

$$k(c, m, t \cdot (ts', \alpha), \ell_{adv}) \supseteq k(c, m, t, \ell_{adv})$$

Observe that Clause 3 of the above definition now requires timing-sensitivity since it explicitly states that an attacker must not learn anything from observing an event α and its timestamp. Another notable change is Clause 1a that specifies that the timing of a regular declassification must not convey information. Finally, this definition also changes the semantics of the tini-construct (cf. Clause 2b). Instead of declassifying the progress knowledge it now declassifies the timing behavior of the code block guarded by tini.

V. Discussion

a) Dimensions and principles of declassification: The reframing of the progress-insensitive security as declassification allows us to think about it in terms of declassification principles and dimensions. The locality-driven aspect of our definition places it in a *where* dimension, while the use of authority-based bounds naturally has a clear *what* flavor. While we do not specify any bounds on what information can be learned via a tini-declassification as long as the authority is sufficient, the prior work on tight specification of *what* information is released through declassifications [4], [16] should compose with our definition. Our authority model is inspired by the expressive label models such as DLM [25] and FLAM [1]; and studying our condition in the formal frameworks of these label models will lead to *who* characterizations of the tini-declassifications.

Another interesting angle to explore is the integration of integrity into the formal model, which would allow one to study the robustness [35] of declassifications via progress-insensitivity. Here, a potentially desirable semantic characterization is that attacker-controlled input does not influence information leaked through termination channels. A knowledge-based approach to robustness [7] can provide a starting point for such a definition.

With respect to the four principles of declassification, we believe that the principles of semantic consistency – namely that security definition should be invariant under equivalence-preserving transformations – and of conservativity – namely that the definition of security should be a weakening of noninterference – follow directly from the knowledge-based nature of the definition that is inherently attacker-driven [10]. The principle of monotonicity of release – namely that adding a declassification should not make a secure program insecure – is also satisfied by our definition: adding a tini block to a program that is already accepted by Definitions 7 does not change how the definition

treats this program, because all knowledge containments for the declassification cases are weaker than Clause 3 of the definition (a similar argument applies to the normal declassification). Finally, our definition also satisfies the non-occlusion principle – namely, that the presence of declassifications should not mask other covert leaks. This one has two subtleties. The first one is already discussed in Section III-C. The second one is that without Clause 1a of the definition, we would have violated non-occlusion, as examples that reach an explicit declassification after a high loop would have been accepted.

Similar arguments apply to Definition 10.

b) Design principle for pc-declassifications: In the information flow community, pc-declassifications have a poor reputation because their security characterization has been not well understood. Our work provides a principle for understanding security of pc-declassifications that can answer the following question: *given a programming language or a system that has a primitive for pc-declassification, how dangerous is it?* The key to answering this question is bounding the progress knowledge.

If the security of pc-declassification can be characterized as a bound on progress knowledge – as we do in Definition 7 – then these pc-declassifications are as dangerous as leaks through progress. However, if progress knowledge cannot be bounded, then these pc-declassifications are more dangerous. For example, in a system designed to allow any pc-declassifications, programs such as

$$\mathsf{if}\ h\ \mathsf{then}\ \mathsf{pcdecl}(H_{auth}, L); l = 0\ \mathsf{else}\ \mathsf{pcdecl}(H_{auth}, L); l = 1$$

can leak information indirectly more efficiently than just encoding the secret in the length of the trace.

c) Access control to authority: Neither our security policy nor the language provides guarantees about programs that misuse authority if they have access to it. To that extent, our approach leaves it to the programmers to ensure that untrusted code does not have access to authority above the code's intended security clearance. However, the capability-based nature of the authority means that a complementary technique for principled control of capabilities can be used. One candidate approach is the work by Dimoulas *et al.* [18] that uses access control and integrity policies to restrict capability use. Another is the mechanism of bounded privileges for LIO proposed by Waye *et al.* [34].

d) Enforcement techniques: We choose a simple runtime monitor to showcase the enforcement of the new definition. While the monitor is fully dynamic and flow-insensitive, we believe that other single-trace monitoring techniques such as hybrid information flow monitors [2], [21], [26] as well as Denning-style static techniques can be easily adapted. Static approaches may have an added benefit of helping infer the location of tini statements. An interesting prospect for future work is extension of monitors designed for declassification for secure multi execution [24], [29] to enforce our definition.

e) Timing treatment: Our treatment of timing-sensitivity in Section IV via a simple step counter is admittedly academic, given the plethora of architectural and runtime side channels today. We nevertheless believe that the formulation of the timing-sensitive condition is useful, and can be combined with other proposals to mitigate practical timing attacks such as predictive mitigation [8], [37], [38].

VI. Implementation experience

We implemented the tini-based enforcement as a part of Troupe [12]. This language enforces progress-sensitive security, but allows tini-scoped initialization as a variation of let-declarations

```
let tini auth  (* tini declaration *)
    val v1 = e1
    val v2 = e2
    ...
in (* the point of pcdecl *)
    e
end
```

This construct declassifies the termination of the initialization expressions e1, e2, ... using authority auth before evaluating the body e.

Figure 7 presents a snippet from the code of the news widget example in our language. The top listing is the source of the news widget itself. When invoked with the favorite topic and its current state as arguments, it updates the counter, fetching updated news from the remote servers if necessary. Finally, it returns the result together with the updated state. Fetching the news is potentially blocking and implemented in the function fetch_news (omitted from the listing but it uses the networking primitives of the language). The news value is an associative list, and the secret-dependent lookup is done using the built-in function list_lookup_with_default. The initial state of the widget is an empty list, with the counter set to zero. The security level of the initial state is NEWS.

The bottom listing in the figure displays how this widget is used by user at level ALICE. The important part is the invocation of the news_widget is placed in the let tini block with attenuated authority NEWS, which limits the termination leakage of the news_widget function.

The actual example is about 80 lines of code. As another data point for the readers, a different case study in our language of roughly 500LOC uses the let tini construct 9 times.

VII. Related work

a) pc-declassification: Jif provides a mechanism for pc-downgrading in the form of a declassify statement that lowers that pc-label that is tracked by the type system. Unlike other features of Jif that are proven sound, e.g., dependent labels [39] or robust declassification [15], there is no soundness theorem for the pc-declassifications.

Both the Asbestos [19] and the HiStar [36] operating systems also allow downgrading of the control-flow. In Asbestos a process with *privilege*, a related notion to our authority, can *decontaminate* other processes' *send* label which has the effect of allowing the other process to "forget" that it has previously seen secret data from the privileged process. In our setup, this corresponds to passing an authority that allows declassifying control-flow up to the senders level. HiStar similarly makes it possible to lower the accrued taint by passing on untainting

```
fun news_widget fav state =
  let
    val (news, update_counter) = state
    val news = if update_counter %10 = 0
               then fetch_news() (* Blocking *)
               else news
    val update_counter = update_counter + 1
    (* Operation on the secret *)
    val fav_news = list_lookup_with_default
                          news fav "no news"
  in (fav_news, (news,update_counter))
  end
val init_state = ([], 0) raisedTo {NEWS}
```

```
(* Receiving widget and initial state *)
val (news_widget, state0) = fetch_widget ()

(* Usage of the widget by user ALICE *)
val news_auth = attenuate(rootauth, {NEWS})
val fav_topic = "#politics" raisedTo {ALICE}

(* Calling untrusted widget code *)
val (fav_news1, state1) =
       let tini news_auth
         val res = news_widget fav_topic state0
       in res
       end
```

Fig. 7. News widget (top) and its usage (bottom) code snippets

gates that act as a capability for lowering the pc. Both of these systems provide this functionality because it is a practical feature to have, but neither of them presents a security condition that encapsulates what this feature entails regarding leakage.

Chandra and Franz [14] present an information flow framework for the Java Virtual Machine with a hybrid monitoring that uses a static analysis to reason about when it is safe to declassify the pc. Similarly to earlier work by, for example, Denning [17], they statically find the immediate postdominator (the nearest join-point that all execution paths must pass through) to any branch-point and insert a pc-lowering command at this point. Their security condition is intended to only allow lowering the pc when no knowledge is revealed by doing so, but since they are in a setting where almost any bytecode can throw unchecked exceptions, this is not generally feasible. Instead, they disregard all implicit flows through unchecked exceptions and accept these leaks as a limitation of the security the system provides. We believe one could extend this line of work by applying our bound on what is learned through such flows, and thereby gain a stronger guarantee for the system as a whole.

The idea of control flow declassification also appears in the discussions of information flow control vs. taint tracking. For example, Schoepe *et al.* [31] use an observational approach where every branch decision is declassified.

b) Knowledge-based policies: The methodology and the experience of this paper is in line with the argument by Broberg *et al.* [13] that epistemic specifications is the most natural way to specify information flow properties:

> The notion of security intrinsically has nothing to do with observing two separate runs – but rather what can be deduced from observing a single run. [...] A two-run formulation could certainly be very useful as part of the strategy to prove e.g. the correctness of an enforcement mechanism. [...] But that property is then only a stepping stone, and should, for completeness, be shown to imply the natural epistemic property.

In our case, it is the operational security (cf. Lemma 2) that has the two-run formulation.

The knowledge-based approach we use in this work follows the style of definitions of gradual release [3]. Logical epistemic approaches include the work by Halpern and O'Neill [20] that use epistemic logic to specify noninterference, and that of Balliu *et al.* [9] that uses epistemic temporal logic used to reason about knowledge acquired by observing program outputs.

Chudnov and Naumann [16] define an epistemic semantics for relational assumptions and guarantees in a progress-insensitive setting. To specify the allowed knowledge at a particular point in the trace they define a notion of release policy of a trace, where relational assumptions are interpreted as an annotation permitting the attacker to learn new information. The insight of our work suggests the direction of lifting their approach into a progress-sensitive setting and treating progress leaks as another form of relational assumptions.

McCall et al. [24] propose a model for enforcing information flow control in the setting of webpages that must handle execution of untrusted scripts. Their approach enforces robust declassification such that untrusted code cannot influence what is declassified by extending prior work on secure-multi-execution. They show their enforcement sound with respect to a knowledge-based progress-insensitive noninterference condition with declassification. They also present a progress-sensitive notion of noninterference, but restrict their focus to the weaker progress-insensitive condition, because IO-operations can use potentially looping event handlers that leak information through progress (a design decision somewhat reminiscent of the scenario in the Introduction). In the context of their work, the bridge between progress-sensitive and progress-insensitive security provided by our definition, can allow programmers to explicitly state when, and how much, information an event handler is allowed to leak through divergence.

c) Leakage via termination: Moore *et al.* [27] propose a type-based enforcement combined with a runtime mechanism for budgeting the amount of information leaked through termination at runtime. The idea is to use a termination oracle that uses maximum available runtime public information to deduce the termination behavior of secret-dependent code. The budgets mechanism allows for a quantitative interpretation of the leakage.

d) Untrusted code: LIO [32], MAC [33], and related programming models side step the issue of label creep via a programming discipline where high computations are forked into separate processes. A consequence of this programming model however is that consuming the result of the forked computation requires process synchronization followed by explicit declassification. Fabric [22] contains a number of mechanisms for

confining untrusted code downloaded over a network, including limits on authority that the code can use and access labels that limit when the untrusted code can read remote objects. As Fabric is based on Jif, it also places timing and progress channels outside of its threat model.

VIII. Conclusion

This paper proposes two novel knowledge-based security conditions that capture the semantic meaning of declassifying the progress knowledge in information flow control systems. While many language-based and architectural systems allows such declassification there is, to the best of our knowledge, no formal characterization of it. We present a language construct, tini, that exactly captures the embedding of progress-insensitive code in a stricter setting and show how this can be used in the presence of potentially blocking or diverging untrusted code. We furthermore show that our conditions are enforceable by a mostly standard dynamic monitor. For future work we conjecture that our epistemic definitions can form a foundation for further studies by extending it with for example integrity and robust downgrades, principled usage of authority-capabilities, or more elaborate label models. Finally, we believe that a large body of techniques that rely on progress-insensitive security can use the insight of our work to accommodate stronger adversary models.

IX. Acknowledgements

We thank Mathias Vorreiter Pedersen for his help with the technical aspects of this work at an earlier stage, Alix Trieu and Andrei Sabelfeld for their comments and insights, and the anonymous reviewers for their suggestions for improving the presentation of this paper. This work is supported by the DFF project 6108-00363 from The Danish Council for Independent Research for the Natural Sciences (FNU) and Aarhus University Research Foundation.

References

[1] O. Arden, J. Liu, and A. C. Myers, "Flow-limited authorization," in *Proceedings of the 2015 IEEE 28th Computer Security Foundations Symposium*, ser. CSF '15, Washington, DC, USA: IEEE Computer Society, 2015, pp. 569–583.

[2] A. Askarov, S. Chong, and H. Mantel, "Hybrid monitors for concurrent noninterference," in *2015 IEEE 28th Computer Security Foundations Symposium*, Jul. 2015, pp. 137–151.

[3] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *2007 IEEE Symposium on Security and Privacy (SP '07)*, May 2007, pp. 207–221.

[4] A. Askarov and A. Sabelfeld, "Tight enforcement of information-release policies for dynamic languages," in *2009 22nd IEEE Computer Security Foundations Symposium*, Jul. 2009, pp. 43–59.

[5] A. Askarov and S. Chong, "Learning is change in knowledge: Knowledge-based security for dynamic policies," in *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, 2012, pp. 308–322.

[6] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, 2008, pp. 333–348.

[7] A. Askarov and A. Myers, "Attacker control and impact for confidentiality and integrity," *Logical Methods in Computer Science*, vol. 7, no. 3, M. Hicks, Ed., Sep. 2011.

[8] A. Askarov, D. Zhang, and A. C. Myers, "Predictive black-box mitigation of timing channels," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 297–307.

[9] M. Balliu, M. Dam, and G. Le Guernic, "Epistemic temporal logic for information flow security," in *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, ACM, 2011, p. 6.

[10] I. Bastys, F. Piessens, and A. Sabelfeld, "Prudent design principles for information flow control," in *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, ACM, 2018, pp. 17–23.

[11] J. Bay and A. Askarov, "Reconciling progress-insensitive noninterference and declassification," Tech. Rep., 2020, https://arxiv.org/pdf/2005.01977.

[12] J. Bay and A. Askarov. (Jul. 2019). "Troupe programming language." Software release and user manual, available at http://troupe.cs.au.dk.

[13] N. Broberg, B. van Delft, and D. Sands, "The anatomy and facets of dynamic policies," in *2015 IEEE 28th Computer Security Foundations Symposium*, IEEE, 2015, pp. 122–136.

[14] D. Chandra and M. Franz, "Fine-grained information flow analysis and enforcement in a java virtual machine," Jan. 2008, pp. 463–475.

[15] S. Chong and A. C. Myers, "Decentralized robustness," in *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, ser. CSFW '06, Washington, DC, USA: IEEE Computer Society, 2006, pp. 242–256.

[16] A. Chudnov and D. A. Naumann, "Assuming you know: Epistemic semantics of relational annotations for expressive flow policies," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, IEEE, 2018, pp. 189–203.

[17] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977.

[18] C. Dimoulas, S. Moore, A. Askarov, and S. Chong, "Declarative policies for capability control," in *2014 IEEE 27th Computer Security Foundations Symposium*, Jul. 2014, pp. 3–17.

[19] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris, "Labels and event processes in the asbestos operating system," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05, Brighton, United Kingdom: ACM, 2005, pp. 17–30.

[20] J. Halpern and K. O'Neill, "Secrecy in multiagent systems," in *Proceedings 15th IEEE Computer Security Foundations Workshop. CSFW-15*, IEEE, 2002, pp. 32–46.

[21] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "Jsflow: Tracking information flow in javascript and its apis," *Proceedings of the ACM Symposium on Applied Computing*, Mar. 2014.

[22] J. Liu, O. Arden, M. D. George, and A. C. Myers, "Fabric: Building open distributed systems securely by construction," *Journal of Computer Security*, vol. 25, no. 4-5, pp. 367–426, 2017.

[23] H. Mantel and D. Sands, "Controlled declassification based on intransitive noninterference," vol. 3302, Nov. 2004, pp. 129–145.

[24] M. McCall, H. Zhang, and L. Jia, "Knowledge-based security of dynamic secrets for reactive programs," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, Jul. 2018, pp. 175–188.

[25] B. Montagu, B. C. Pierce, and R. Pollack, "A theory of information-flow labels," in *2013 IEEE 26th Computer Security Foundations Symposium*, Jun. 2013, pp. 3–17.

[26] S. Moore, A. Askarov, and S. Chong, "Precise enforcement of progress-sensitive security," in *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, 2012, pp. 881–893.

[27] S. Moore, A. Askarov, and S. Chong, "Precise enforcement of progress-sensitive security," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 881–893.

[28] A. C. Myers and A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL)*, ACM, 1999.

[29] W. Rafnsson and A. Sabelfeld, "Secure multi-execution: Fine-grained, declassification-aware, and transparent," in *2013 IEEE 26th Computer Security Foundations Symposium*, Jun. 2013, pp. 33–48.

[30] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, Jun. 2005, pp. 255–269.

[31] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld, "Explicit secrecy: A policy for taint tracking," in *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, Mar. 2016, pp. 15–30.

[32] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, *Flexible dynamic information flow control in the presence of exceptions*, 2012. arXiv: 1207.1457 [cs.CR].

[33] M. Vassena, A. Russo, P. Buiras, and L. Waye, "Mac a verified static information-flow control library," *Journal of Logical and Algebraic Methods in Programming*, vol. 95, Dec. 2017.

[34] L. Waye, P. Buiras, D. King, S. Chong, and A. Russo, "It's my privilege: Controlling downgrading in dc-labels," Sep. 2015, pp. 203–219.

[35] S. Zdancewic and A. C. Myers, "Robust declassification," in *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, ser. CSFW '01, Washington, DC, USA: IEEE Computer Society, 2001, pp. 5–.

[36] N. Zeldovich, S. Boyd-wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in histar," in *In Proc. 7th OSDI*, 2006.

[37] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 99–110.

[38] D. Zhang, A. Askarov, and A. C. Myers, "Predictive mitigation of timing channels in interactive systems," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 563–574.

[39] L. Zheng and A. C. Myers, "Dynamic security labels and static information flow control," *Int. J. Inf. Secur.*, vol. 6, no. 2-3, pp. 67–84, Mar. 2007.