Tight Enforcement of Information-Release Policies for Dynamic Languages

Aslan Askarov Andrei Sabelfeld

Abstract

This paper studies the problem of securing information release in dynamic languages. We propose (i) an intuitive framework for information-release policies expressing both what *can be released by an application and* where *in the code this release may take place and (ii) tight and modular enforcement by hybrid mechanisms that combine monitoring with on-the-fly static analysis for a language with dynamic code evaluation and communication primitives. The policy framework and enforcement mechanisms support both termination-sensitive and insensitive security policies.*

1. Introduction

As computing systems are becoming increasingly extensible and interconnected, the challenge of securing applications written in dynamic and distributed languages is becoming increasingly important. This challenge is particularly pressing for web applications that critically rely on dynamism and distribution.

Information-flow tracking in web applications provides a viable, and increasingly popular, alternative for enforcing end-to-end confidentiality and integrity. Information-flow tracking is ubiquitous in several recent practical approaches to web security. To give a few recent examples, these approaches include server-side mechanisms (e.g., [19], [10]), client-side mechanisms for JavaScript (e.g., [37]) and JVM (e.g., [8]), as well as mechanisms that combine protection for servers and clients [9]. However, while promising, this line of work lacks soundness guarantees and sound support for information-release (or *declassification*) policies.

On the other side of the spectrum, much progress has been made on formal reasoning about security policies and (mostly static) enforcement [28], [30]. However, dynamic code evaluation has been out of reach for the mostly static techniques developed so far.

Recently, several dynamic techniques for program security have emerged [38], [21], [32], [20]. Yet they have two limitations: (i) they target restrictive *noninterference* [17] policies (stipulating that there should be no dependence of public outputs on secret inputs) and (ii) they do not handle dynamic code evaluation.

The first limitation poses problems for practical use because noninterference is too strong for many applications that intentionally release some secret information [28], [30]. For example, programs that need to release an average salary or the result of password checking would be ruled out as insecure.

The second limitation (which is also a fundamental limitation for static enforcement mechanisms) prevents us from applying the technology of information-flow sensitive languages [25], [33] to widely-used dynamic languages. In the context of web applications, where there are requirements on information flow control of sensitive data, dynamic code evaluation is a popular feature. A quick check reveals that the `eval` primitive is used in nearly a quarter of the pages with embedded JavaScript indexed by Google code search.

Clearly, there is a gap between formal, mostly static, approaches—that lack support for dynamic code evaluation—and practical, mostly dynamic, approaches—that lack soundness and support for flexible information-release policies. Bridging this gap is a research program that requires a substantial research and engineering effort. This paper obviously does not have an ambition to do it all, but we believe it can at least make a step in this direction: we propose an intuitive and general framework for reasoning about information-release policies for expressing both *what* can be released by an application and *where* in the code this release may take place.

The framework makes it possible to express not only such simple information-release policies as for password-checking and average-salary programs, but also more complex and dynamic ones. For example, a form-validating script should not be able to steal a credit card number. We give a server- and a client-side scenario as further examples:

Server-side scenario A third-party service (cf. [1]) offers users help in bidding for auctioning web sites. The user specifies the maximum amount A he is prepared to pay for the goods, and the server participates in bidding on the user's behalf by minimally increasing the current bid B as long as it does not exceed what the user is prepared to pay. The policy for information release from the user to the auction is to only reveal whether $A > B$ and nothing else about A. Both A and B can be changed dynamically.

Client-side scenario A third-party service provides an API for embedding maps in web pages. APIs as the Google Maps API [2] rely on inclusion of their scripts in trusted pages, which gives full trust to these scripts by today's browsers (e.g., the scripts get access to the entire document, including possibly sensitive data). But our approach allows limiting the trust by the following policy: release the coordinates of the objects to be displayed to the third party but allow no other user data to be leaked. Note that dynamic code generation needs to be addressed by enforcement in this scenario: new code for map rendering is requested and run in response to user events such as moving the map [2].

Not only is the policy framework general, but also tightly enforceable: we present a permissive yet sound hybrid of monitoring and on-the-fly static analysis that enforces security for a language with web-style primitives for dynamic code evaluation and communication.

2. Security specification

In this section, we state our assumptions on the semantics of programs and present a security specification for information release.

Semantics Without loss of generality, we assume a two-element security lattice Lev with levels L (for *low*, or public) and H (for *high*, or secret), where $L \sqsubseteq H$ and $H \not\sqsubseteq L$. Assume Γ is a security environment $\Gamma : Vars \rightarrow Lev$ that maps variable names $Vars$ to security levels Lev.

Program *configurations* are triples of the form $cfgc = \langle c, m, E\rangle$ where c is a command (program), m is a memory (mapping variables to values $m : Vars \rightarrow Vals$), and E is a set of released expressions. The set of released expressions E is updated every time an expression is declassified. We assume mapping m can be extended to expressions.

Small-step semantic transitions between configurations have the form $cfgc \longrightarrow_{\alpha} cfgc'$, where α is a *low event*. Low events describe an attacker's capability to observe changes in low memory. This allows modeling a powerful attacker and accommodates a straightforward treatment of input and output (which we present in Section 4). Some program transitions do not generate any low events, which, when there is need to be explicit, we denote as ϵ. A distinguished form of a low event is program termination $\downarrow$. We have:

$$\alpha ::= \ell \mid \epsilon \qquad \ell ::= (x, v) \mid \downarrow$$

Here (x, v) corresponds to an assignment of value v to a low variable x. A program trace $cfgc_0 \longrightarrow_{\alpha_1} \dots \longrightarrow_{\alpha_n} cfgc_n$ produces a sequence of low events $\vec{\ell}$, where $\vec{\ell}$ is defined as the subsequence of all non-empty events among $\alpha_1, \dots, \alpha_n$, if there is any, and ϵ otherwise. This will also be denoted as $cfgc_0 \longrightarrow_{\vec{\ell}} cfgc_n$.

Security condition Declassification primitives in a program specify *what* information about the initial values of secret variables is disclosed and *where* in a program run this declassification happens. These primitives have the form $\texttt{declassify}(e)$ for some expression e.

We treat expressions e appearing in $\texttt{declassify}(e)$ primitives as *escape hatches* [29] that describe what is released. These expressions induce an indistinguishability relation on memories corresponding to what the attacker can and cannot distinguish given that the expressions have been released: Given a set of escape hatches E, two memories m_1 and m_2 are indistinguishable by E, written $m_1 \ I(E) \ m_2$, if the memories agree on all expressions from the escape-hatch set: $\forall e \in E \ . \ m_1(e) = m_2(e)$. For example, if the only escape hatch is the average of two high variables h and h', then the average value is visible to the attacker, but memories that agree on $(h+h')/2$ are indistinguishable by the attacker. If the set of escape hatches is empty, then nothing can be learned by the attacker: the indistinguishability relation relates all memories.

Our security condition is based on reasoning about the attacker's *knowledge* about the initial values of high variables (cf. [15], [4], [6]). Initially, this knowledge corresponds to all possible values. As the computation goes along, the knowledge can be refined, i.e., the attacker may learn that some values are *not* possible. The essence of our condition is that at any given point the attacker may not learn more than what is allowed by the escape-hatch expressions that have been released so far. A useful feature of the condition is that it is defined per individual run of a program, which makes it amenable for runtime monitoring.

Given a trace $t = cfgc_0 \longrightarrow_{\vec{\ell}} cfgc_n$ that produces a sequence of low events $\vec{\ell}$, assume m is the initial memory in $cfgc_0$, and E is the set of escape hatches accumulated in the last configuration $cfgc_n$. Assume m_L denotes the low projection of memory m. Define the *release policy* $p(m, E)$ as the set of all memories that agree on the low variables with m and that are indistinguishable from m by the escape hatches from E:

$$p(m, E) = \{m' \mid m'_L = m_L \wedge m' \ I(E) \ m\}$$

The release policy describes what is visible by the attacker with access to the initial values of low variables and escape-hatch expressions. The release policy can be

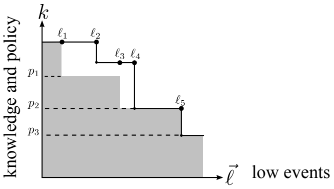

Fig. 1. Security condition

equivalently defined as $p(m, E) = \{m' \mid m' \; I(E \cup \Gamma_L) \; m\}$, where Γ_L is the set of low variables.

The attacker can gain knowledge by observing low events. Given a command c, the low part i_L of the initial memory i, and a sequence of low events $\vec{\ell}$, the *knowledge* is the set of memories that agree with i on the low variables and can lead to generating $\vec{\ell}$:

$$k(c, i_L, \vec{\ell}) = \{m \mid m_L = i_L \wedge \langle c, m, \emptyset\rangle \longrightarrow_{\vec{\ell}} \langle c', m', E'\rangle\}$$

The condition that the attacker may not learn more than what is allowed by the escape-hatch expressions can be specified by a straightforward set inclusion of the knowledge allowed by the policy into the knowledge the attacker may derive at a given step. Figure 1 illustrates how the attacker's knowledge can be refined over time. At each event of a sequence $\vec{\ell}$, the attacker may gain some knowledge. This happens at events ℓ_2, ℓ_4, and ℓ_5 in the figure, when the solid line drops to a more refined knowledge. The gray area corresponds to the evolution of the release policy over time. As the set of escape-hatch expressions grows, the policy allows the attacker to distinguish more and more data. The key condition is that the attacker may not learn more than what is allowed by the policy: the solid line may never cross into the gray area.

Definition 1. (TERMINATION-SENSITIVE SECURITY). *A program c is* secure *with respect to a sequence of low events $\vec{\ell}$ and initial low memory i_L, denoted $TSec(c, i_L, \vec{\ell})$, if for all memories $m \in k(c, i_L, \vec{\ell})$ that produce $\vec{\ell}$ we have:*

$$\forall i \;.\; 1 \leq i \leq n \;.\; p(m, E_i) \subseteq k(c, m_L, \vec{\ell_i})$$

where $\vec{\ell_i}$ is the i-prefix of $\vec{\ell}$, $\vec{\ell} = \vec{\ell_n}$ for some n, and E_i is extracted from the configuration that generated the last event in $\vec{\ell_i}$.

The above definition is simple yet powerful. It allows runs of the following program:

$$l := \texttt{declassify}(h)$$

Although the attacker can refine the knowledge about h to a single value, this refinement is allowed by the policy because it only allows memories that agree on h. We have $p(m, \{h\}) = \{m' \mid m'_L = m_L \wedge m' \; I(\{h\}) \; m\} = \{m' \mid m'_L = m_L \wedge m'(h) = m(h)\}$, which is included in (in fact, equal to) $k(c, m_L, (l, m(h)))$.

Consider the following laundering attack. The escape-hatch expression is h (e.g., storing the expiry date of a credit card), but it is the initial value of h' (e.g., storing the credit card number) that is leaked:

$$h := h'; l := \texttt{declassify}(h)$$

Here, the attacker may learn the initial value of h'. However, this is not allowed by the policy, which demands agreement on h but allows memories with all possible values for h'. We have $p(m, \{h\}) = \{m' \mid m'_L = m_L \wedge m' \; I(\{h\}) \; m\} = \{m' \mid m'_L = m_L \wedge m'(h) = m(h)\}$. Take memory $m'' \in p(m, \{h\})$ so that $m''(h') \neq m(h')$. But $m'' \notin k(c, m_L, (l, m(h')))$ by the choice of m'', and thus $p(m, \{h\}) \not\subseteq k(c, m_L, (l, m(h)))$. Therefore, such runs are rightfully rejected by the definition.

Note that a simple approach to avoid possibilities of this kind of laundering would be do force the programmer to declare escape hatches in a global declaration block and "activate" them by declassification, along the lines of:

$$let \; \texttt{hatch} \; ha = h; \ldots \; in \ldots l := \texttt{declassify}(ha); \ldots$$

where ha is an immutable high variable. However, this solution does not scale well, when the escape hatches depend on dynamically received input (subject of Section 4).

Consider an example where the escape-hatch expression is the average $avg(h, h')$ of two variables h and h':

$$t := h'; h' := h; h := t; l := \texttt{declassify}(avg(h, h'))$$

Although the variables h and h' are swapped, the value of the expression $(h + h')/2$ at the declassification time equals its initial value. Therefore, the inclusion of the policy into the knowledge set holds, and this program is accepted by the definition.

The above security definition is *termination-sensitive* (cf. [28]): from seeing a low event in a program with possible divergence, the attacker may learn some sensitive information. For example, when running the program $(\texttt{while} \; h \; \texttt{do} \; skip); l := 5$, if the attacker observes that l has been assigned 5, the attacker learns that h was 0. Sometimes, a weaker security definition is desirable, which accepts the program as secure on the grounds that leaks via termination are hard to exploit. Thus, we also present a *termination-insensitive* definition. Askarov et al. [3] provide a formal justification for a declassification-free version of this definition: if a program satisfies the definition, then the attacker may not learn the secret in polynomial running time in the size of the secret; and, for uniformly-distributed secrets, the probability of guessing the secret in polynomial running time is negligible.

We cast insensitivity to (non)termination by allowing new knowledge when observing the next output, but only as much as can be learned from the fact that there is *some* next output. This knowledge, dubbed *progress knowledge*, can be expressed as the union of all knowledge sets that correspond to extending a low trace $\vec{\ell}$ with next output: $\bigcup_{\ell'} k(c, m_L, \vec{\ell}\ell')$.

With this notion at hand, we have a way of ignoring leaks due to nontermination at each step. We refer to this notion as *progress-insensitive* security or, for the sake of compatibility with the declassification-free version [3], *termination-insensitive* security.

Definition 2. (TERMINATION-INSENSITIVE SECURITY). *A program c is* secure *with respect to a sequence of low events $\vec{\ell}$ and initial low memory i_L, denoted $TISec(c, i_L, \vec{\ell})$, if for all memories $m \in k(c, i_L, \vec{\ell})$ that produce $\vec{\ell}$, we have: $\forall i \;.\; 1 \leq i \leq n$.*

$$p(m, E_i) \cap \bigcup_{\ell'} k(c, m_L, \vec{\ell}_{i-1}\ell') \subseteq k(c, m_L, \vec{\ell}_i)$$

where $\vec{\ell}_i$ is the i-prefix of $\vec{\ell}$, $\vec{\ell} = \vec{\ell}_n$ for some n, and E_i is extracted from the configuration that generated the last event in $\vec{\ell}_i$.

As intended, runs of the program $(\texttt{while}\ h\ \texttt{do}\ \mathit{skip}); l := 5$ are accepted by Definition 2. Memories that lead to diverging traces are ruled out by the progress knowledge; hence, there is no refinement on observing that 5 has been assigned to l. On the other hand, runs of the program $(\texttt{while}\ h\ \texttt{do}\ \mathit{skip}); l := h$ are rejected by Definition 2. The progress knowledge allows refinement of the loop guard $h = 0$, but the low assignment gives the exact value of h which is much more precise than what progress knowledge allows.

3. Enforcement

This section shows how to enforce security policies by a hybrid of monitoring and on-the-fly static analysis for a language with dynamic code evaluation. Because termination-insensitive enforcement is simpler (it requires no major static analysis), we present it first.

Language We consider a simple imperative language with an $\texttt{eval}(s)$ primitive for dynamic code evaluation of string s. Figure 7 in Appendix B displays the syntax of the language. Expressions consist of constant integers n and strings s, variables x, and composite expressions $e\ op\ e$, where op ranges over total operations. For simplicity, we assume declassifications have the form $x := \texttt{declassify}(e)$ for some low variable x and declassification-free expression e.

The semantics of expressions extends mapping m from variables to arbitrary expressions as follows: $m(e_1\ op\ e_2) = m(e_1)\ op\ m(e_2)$. As before, we write $m(e) = v$ whenever expression e evaluates to value v (either integer n or string s) under memory m.

The semantics of commands is similar to standard small-step semantics (see Figure 8 in Appendix B for the details). Low events $\ell = (x, v)$ are generated by assignments (with and without declassification) whenever the assigned variable x is low. Value v is the result of evaluating the expression on the right-hand side of the assignment in the current memory. The rule for declassification includes the underlying expression in the escape-hatch set. Dynamic code evaluation of an expression e succeeds when e evaluates to a string in the current memory, and this string can be successfully parsed. (Failing to parse the string would result in a dynamic error in a realistic language, but we represent such an event by a stuck state for simplicity: turning the execution into the stuck state can be achieved in the monitor anyway.) We assume that program termination event $\downarrow$ is generated when the program reaches terminal configuration $\langle \mathit{stop}, m, E\rangle$ for some m and E and no other transitions are possible.

We also equip the semantics with monitor events β that represent the interface of the execution with a monitor. The executions of a configuration and the monitor are synchronized via these events.

We instantiate monitor events β for our language as follows. Event nop signals that the program performs a $\texttt{skip}$. Event $a(x, e)$ records that the program assigns the value of e in the current memory to variable x. Event $d(x, e, m)$ reports a declassification of expression e in the current memory m into variable x. Sequential composition propagates monitor events similarly to low events. Event $b(e, c_1; c_2)$ indicates that the program branches on expression e and is about to enter one of the branches c_1 or c_2. This information is important for the static analysis part of the monitor, as we explain later. When the program either enters or skips a $\texttt{while}$ loop with guard e or when it runs $\texttt{eval}(e)$ event $we(e)$ is triggered. In all three cases it is important to communicate to the monitor the security level of expression e: when it is high, then implicit flows via loops and via dynamic code generation (exemplified below) need to be prevented by the monitor. Finally, event f is generated when the structure block of a conditional, loop, or $\texttt{eval}$ has finished evaluation.

Termination-insensitive enforcement The semantics of the monitor is reported in Figure 2. A monitor configuration has the form $cfgm = \langle i, st\rangle$. The monitor is parameterized in the initial memory i and contains a *context stack* [16], [21], a stack of security levels st, which is initially empty (denoted ϵ). The stack helps tracking control flow (cf. *implicit flows* below).

$$\langle i, st\rangle \xrightarrow{nop} \langle i, st\rangle \qquad \frac{lev(e) \sqsubseteq \Gamma(x) \qquad lev(st) \sqsubseteq \Gamma(x)}{\langle i, st\rangle \xrightarrow{a(x,e)} \langle i, st\rangle}$$

$$\langle i, hd : st\rangle \xrightarrow{f} \langle i, st\rangle \qquad \frac{m(e) = i(e) \qquad lev(st) \sqsubseteq \Gamma(x)}{\langle i, st\rangle \xrightarrow{d(x,e,m)} \langle i, st\rangle}$$

$$\langle i, st\rangle \xrightarrow{b(e,c)} \langle i, lev(e) : st\rangle \qquad \langle i, st\rangle \xrightarrow{we(e)} \langle i, lev(e) : st\rangle$$

Fig. 2. Termination-insensitive enforcement

When reaching an `if` $e\ldots$, `while` $e\ldots$, or `eval`(s), the security level of e, or s, respectively, is pushed onto the stack. When reaching an end instruction that closes the scope of one of the above commands, the topmost security level is popped from the stack. Based on the events generated by the program, the monitor may stop its execution or allow it, while keeping track of the context stack. Assume function $lev(e)$ returns the highest level of a variable encountered in expression e. Similarly, $lev(st)$ returns H if there is an H element in stack st, and L otherwise.

Event nop (that originates from a `skip`) is always accepted without changes in the monitor state. Event $a(x, e)$ (that originates from an assignment) is accepted without changes in the monitor state but with two conditions: (i) that the security level of expression e is no greater than the security level of variable x and (ii) that the highest level of the context stack is no greater than the security level of variable x. The former prevents *explicit* flows of the form $l := h$, whereas the latter prevents *implicit* [14] flows of the form `if` h `then` $l := 1$ `else` $l := 0$, where depending on the high guard, the execution of the program leads to different low events.

Events $b(e, c)$ and $we(e)$ result in pushing the security level of e onto the stack of the monitor. This is a part of implicit-flow prevention: runs of program `if` h `then` $l := 1$ `else` $l := 0$ are stopped before performing an assignment l because the level of the stack is high when reaching the execution the assignment. The stack structure avoids overrestrictive enforcement. For example, runs of program (`if` h `then` $h := 1$ `else` $h := 0$); $l := 1$ are allowed. This is because by the time the assignment to l is reached, the execution has left the high context: the high security level has been popped from the stack in response to event f, which the program generates on exiting the `if`.

We have seen that runs of programs like `if` h `then` $l := 1$ `else` $l := 0$ are rejected by the monitor. But what about a program like `if` h `then` $l := 1$ `else skip`, a common example for illustrating that dynamic information-flow enforcement is delicate? If h is non-zero, the monitor stops the execution. However, if h is 0, the program proceeds normally. Are we accepting an insecure program? It turns out that the slight difference between unmonitored and monitored runs (stopping in case h is non-zero) is sufficient for termination-insensitive security. In effect, the monitor prevents implicit flows by *collapsing the implicit-flow channel into the termination channel*; it does not introduce any more bandwidth than the termination channel already permits. Indeed, implicit flows in unmonitored runs can be magnified by a loop so that secrets can be leaked bit-by-bit in the linear time of the secret. On the other hand, implicit flows in monitored runs cannot be magnified because execution is stopped whenever it attempts entering a branch with a public side effect. For example, one implication for uniformly-distributed secrets is that they cannot be leaked on the termination channel in polynomial time [3].

Information release is controlled by the rule for the declassification event $d(x, e, m)$. Similarly to the rule for assignment, this rule prevents implicit flows. Explicit flows from an escape-hatch expression to a low variable are allowed, but only if the value of the escape-hatch expression is the same as it was in the initial memory. This prevents laundering because at the time of declassification we only release what is described by the escape-hatch expression with respect to the initial memory and nothing else. Revisiting the examples of Section 2, runs of program:

$$l := \texttt{declassify}(h)$$

are accepted by the monitor because the value of the declassified expression h at the time of declassification is the same as initially. On the other hand, the laundering attack:

$$h := h'; l := \texttt{declassify}(h)$$

where h' is leaked instead of h, is prevented. Suppose for the initial memory i we have $i(h) = 2$ and $i(h') = 3$. Then for memory m, which is obtained after the first assignment, we have $m(h) = m(h') = 3$. Thus, check $m(h) = i(h)$ fails, and therefore the dangerous declassification is disallowed by the monitor.

The monitor prevents leaks due to dynamic code evaluation. For example, consider program:

$$(\texttt{if}\ h\ \texttt{then}\ s := "l := 1"\ \texttt{else}\ s := "l := 0"); \texttt{eval}(s)$$

where s is a high string. Clearly, this program is insecure, as the information about variable h is encoded in string s, which, when evaluated, reflects it in the generated low events. Because s is high, the monitor pushes a high security level in the context stack before executing the assignment. Consequently, assignment to a low variable in a context with a high level will be prevented by the monitor.

Note that getting stuck while parsing a high string or evaluating in a high context is not a problem in the same way as diverging in high context is not a problem because abnormal termination is ignored in the same way as nontermination is ignored by termination-insensitive security.

Thanks to the modularity of our approach, the semantics of monitored execution boils down to a single rule:

$$\frac{\mathit{cfgc} \xrightarrow{\beta}_{\alpha} \mathit{cfgc}' \qquad \mathit{cfgm} \xrightarrow{\beta} \mathit{cfgm}'}{(\mathit{cfgc}, \mathit{cfgm}) \longrightarrow_{\alpha} (\mathit{cfgc}', \mathit{cfgm}')}$$

The rule ensures that a program configuration is allowed to perform a step with monitor event β only if the monitor accepts event β. Stopping the execution (or not) is the only channel for information flow in the direction from the monitor to the attacker. Note that no attacker-observable event is generated when the monitor has stopped execution under this rule. However, this choice is not fundamental: traces with a special abnormal termination event at the next step can be "ignored" by progress knowledge in the same way as traces are ignored if they get stuck at the next step.

In the following we let $\mathit{TISec}(c, i_L, \vec{\ell})$ refer to an execution of c monitored by a termination-insensitive monitor, and to $\mathit{TSec}(c, i_L, \vec{\ell})$ to an execution of c monitored by a termination-sensitive monitor. The security of monitored executions is guaranteed by the following soundness result:

Proposition 1. (SOUNDNESS OF TERMINATION-INSENSITIVE ENFORCEMENT). *Given a program c, initial memory i, and a sequence $\vec{\ell}$ of low events produced by $\langle c, i, \emptyset\rangle$ while monitored by the termination-insensitive monitor $\langle i, \epsilon\rangle$, we have that c satisfies termination-insensitive security with respect to $\vec{\ell}$ and i_L, that is, $\mathit{TISec}(c, i_L, \vec{\ell})$.*

Proofs of this and Proposition 2 follow from Propositions 3 and 4 for the more general language in Section 4.

Note that the monitor gives quite a bit of precision compared to typical static systems: it approximates the security condition more tightly. For example, a static treatment of `eval` would most likely demand no `eval` in high context (unless it is combined with dynamic security context tracking), while the monitor allows `eval` in high context. In addition, the handling of declassification would be different in precision: for example, a program where two values are swapped before releasing their average would most likely be rejected by a static analysis (e.g., [29]), while the monitor allows such a program.

Termination-sensitive enforcement The monitor above tightly enforces termination-insensitive security. But it is not sufficient to guarantee termination-sensitive security. For example, if the value of h is initially 0, the execution of this program is not stopped:

$$\texttt{while}\ h\ \texttt{do skip}$$

However, the observation of a termination event teaches the attacker that h was indeed initially 0. Furthermore, while appropriate for termination-insensitive security, stopping (or not) the program execution in a high context can break termination-sensitive security. For example, if h is initially 0, the monitor of the previous paragraph will not stop the following program:

$$\texttt{if}\ h\ \texttt{then}\ l := 1\ \texttt{else skip}$$

Again, the observation of a termination event teaches the attacker that h was indeed initially 0. Similar problems occur when declassifying, looping, and dynamically evaluating code in a high context.

This motivates a *hybrid* enforcement mechanism, which utilizes on-the-fly static analysis to guarantee that when branching on high, there are no low-observable side effects that can increase the attacker's knowledge.

Figure 3 shows the semantics of the hybrid mechanism. A monitor configuration has now the form $\mathit{cfgm} = \langle \mathit{st}, U\rangle$, where st is the stack of security levels (as before) and U is the set of updated variables (tracked to prevent laundering). We define function $\mathit{lev}(c)$ on commands by assuming it returns the lowest level of a variable assigned in c and returns H if there are no assignments. The function gives a lower bound on side effects produced by c. Assume $\mathit{noeval}(c)$ is true whenever no `eval` statements occur in c. Similarly, $\mathit{noloop}(c)$ holds if no `while` loops occur in c. Let function $\mathit{upd}(c)$ return the set of variables assigned to in c.

As before, event nop (that originates from a `skip`) is accepted without changes in the monitor state. Event $a(x, e)$ (that originates from an assignment) is accepted if x is high. If x is low, then the monitor only checks for explicit flows (implicit flows are checked statically once a branching point is reached). In both cases set U is extended with the variable that has been updated. It is important to update U even for assignment to low variables. For example, if U is not updated after the first assignment when running the program:

$$l := 1; l := \texttt{declassify}(h * l)$$

when l is 0 initially, then h would be allowed to leak although it is not allowed by the escape hatch.

Recall that event $b(e, c)$ indicates that the program branches on expression e and program c contains both branches. If guard e is high and program c contains assignments to low variables, then there is a risk of an implicit flow. Therefore, the monitor performs static analysis of c by computing the lower bound $lev(c)$ on the side effects of c. The execution is allowed to enter a high context only if there are no low assignments in the branches, i.e., $lev(e) \sqsubseteq lev(c)$. Compared to the permissive treatment of `eval` by the termination-insensitive monitor, the termination-sensitive monitor needs to be more conservative about `eval` in sensitive context. To prevent divergence/abnormal termination in sensitive context, we disallow `eval` in branches of conditionals with high guards. Similar restrictions are placed on loops, as is standard [39]. The rule for branching on high data ensures that there are no loops, declassifications, or `eval` statements in the branches. In addition, before branching on high data, the rule records all variables that can possibly be updated in both branches in the set U'. This set is passed along in the updated monitor configuration, preventing future declassification of variables that could possibly be updated in high context. On the other hand, when branching on low data, the set of updated variables need not be changed because updates in low context are treated by the rule $a(x, e)$ as described above. In this case, we simply let $U' = \emptyset$.

The treatment of declassification is another part of the monitor that relies on static analysis. The simple mechanism of the termination-insensitive monitor is not sufficient for termination-sensitive security. Recall the laundering attack:

$$h := h'; l := \texttt{declassify}(h)$$

The progress-insensitive monitor rejects declassification attempts by most runs of this program. However, if the attacker gets lucky and h and h' were the same initially, then the monitor allows the execution. By observing successful declassification in this program, the attacker can learn information about h', and not only about h as was intended. While we can argue along the lines of [3] that the impact of this attack is limited, we also have a solution that provides protection from this kind of attack. This solution corresponds to a dynamic version of the type system for delimited information release [29]. We keep track of a set U of variables that might have been updated and make sure that at each declassification point, no updated variables are involved in declassified expressions. The latter requirement appears in the rule for the declassification event $d(x, e, m)$. The variables $vars(e)$ of the declassified expression e must not have been updated: $vars(e) \cap U = \emptyset$.

The rule for event $we(e)$ disallows loops with high guards [39] as well as `eval` of high strings.

The security of the hybrid mechanism is assured by the following soundness result:

$$\langle st, U\rangle \xrightarrow{nop} \langle st, U\rangle \qquad \langle hd : st, U\rangle \xrightarrow{f} \langle st, U\rangle$$

$$\frac{lev(st) = L \Longrightarrow lev(e) \sqsubseteq lev(x)}{\langle st, U\rangle \xrightarrow{a(x,e)} \langle st, U \cup \{x\}\rangle}$$

$$\frac{\begin{array}{c} lev(e) \sqsubseteq lev(c) \qquad lev(e) = L \Longrightarrow U' = \emptyset \\ lev(e) = H \Longrightarrow noeval(c) \wedge noloop(c) \wedge U' = upd(c) \end{array}}{\langle st, U\rangle \xrightarrow{b(e,c)} \langle lev(e) : st, U \cup U'\rangle}$$

$$\frac{vars(e) \cap U = \emptyset}{\langle st, U\rangle \xrightarrow{d(x,e,m)} \langle st, U \cup \{x\}\rangle} \qquad \frac{lev(e) = L}{\langle st, U\rangle \xrightarrow{we(e)} \langle L : st, U\rangle}$$

Fig. 3. Termination-sensitive enforcement

Proposition 2. (SOUNDNESS OF TERMINATION-SENSITIVE ENFORCEMENT). *Given a program c, initial memory i, and a sequence $\vec{\ell}$ of low events produced by $\langle c, i, \emptyset\rangle$ while monitored by the termination-sensitive monitor $\langle \epsilon, \emptyset\rangle$, we have that c satisfies termination-sensitive security with respect to $\vec{\ell}$ and i_L, that is, $TSec(c, i_L, \vec{\ell})$.*

Compared to the termination-insensitive enforcement, the termination-sensitive one needs to be more conservative. However, it is still not as conservative as static analysis. The difference is particularly dramatic in the presence of communication. For example, a typical static analysis would flatly reject useful programs that dynamically evaluate newly received input (as in the map scenario in Section 1). The next section introduces communication primitives, and Section 5 discusses an implementation for the map scenario that is accepted by both termination-sensitive and -insensitive monitors.

4. Communication primitives

This section extends the security condition and enforcement from Sections 2 and 3, respectively, with communication primitives. For simplicity, we consider a single communication channel per security level (but discuss a straightforward extension to multiple channels in Appendix C).

The release policy and security conditions in Section 2 are based on the indistinguishability of initial memories. Applying these conditions to a system with inputs is only partially satisfactory. One can still reason about noninterference-like policies, but the restriction of release policy to initial memories does not allow declassification of expressions with variables that have been updated by inputs.

An example of such a program is password checking, where a user is prompted to enter a password several times, and after every input the program declassifies if the user's guess matches the secret password.

We observe that, contrary to updates, inputs introduce fresh data into the program and, therefore, distinguish them from ordinary updates. One could also let the programmer control which of the inputs are treated specially, but, for simplicity, we assume that every input introduces fresh data.

We define a variant of indistinguishability that takes inputs into account. In contrast to inputs, outputs are straightforward to adapt. This is due to the intentionally powerful attacker model, introduced in Section 2, that allows the attacker to inspect low updates.

Semantics We introduce channels $\hat{L}$ and $\hat{H}$ for low and high communication, respectively. O'Neill et al. [26] model interaction by *strategies* in their work on termination-sensitive noninterference. But Clark and Hunt observe [11] that it makes no difference for deterministic programs whether communication is modeled by streams or strategies; and so we model channels as *streams*. Each of $\hat{L}$ and $\hat{H}$ is modeled as a pair of streams—one for input, and the other one for output:

$$\hat{L} = [L_{\mathtt{I}}, L_{\mathtt{O}}] \qquad \hat{H} = [H_{\mathtt{I}}, H_{\mathtt{O}}]$$

Next, we keep track of the *input history*—with every input we record a pair of the channel name that is read and the variable that is updated by this input. These pairs are stored in the input history sequence $hist$. Moreover, escape hatches are now pairs of the form (e, r) where e is, as previously, an expression that is declassified, and r is the length of the input history at the time of declassification.

The new configurations have the form $\langle c, m, E, \hat{L}, \hat{H}, hist\rangle$, where c and m are, as before, the current program and memory. For simplicity, we limit inputs from the low channel to low variables and inputs from the high channel to high variables only.

Security condition We let the attacker observe low communication, which is reflected in the definition of low events:

$$\ell ::= \ldots \mid (\mathtt{I}, x, v) \mid (\mathtt{O}, v)$$

$(\mathtt{I}, x, v)$ corresponds to an observation of low input of the value v into the variable x. $(\mathtt{O}, v)$ is an observation of low output of the value v. The attacker may learn new information based on low observations. The attacker's knowledge for the extended language is:

$$k(c, i_L, \hat{L}, \vec{\ell}) = \{(m, \hat{H}) \mid m_L = i_L \wedge \\ \langle c, m, \emptyset, \hat{L}, \hat{H}, \epsilon\rangle \longrightarrow_{\vec{\ell}} \langle c', m', E', \hat{L}', \hat{H}', hist'\rangle\}$$

Note that the domain of the knowledge is now a Cartesian product of two sets: (i) the set of initial memories and (ii) the set of initial high channels. We refer to this domain as a set of *initial high environments*, and define indistinguishability relation $I(E, \hat{L}, hist)$ on it:

$$(m_1, \hat{H}_1)\ I(E, \hat{L}, hist)\ (m_2, \hat{H}_2) \Leftrightarrow \\ \forall (e, r) \in E \ .\ m_1^{hist[r]}(e) = m_2^{hist[r]}(e)$$

Here E is the set of escape hatches of the form (e, r), where the value of r tells us how many inputs have happened before e has been declassified. This includes inputs at all levels. We refer to r to look up the input history up to that point and "replay" effects of these inputs on the initial memories m_1 and m_2, but using $\hat{H}_1$ for high input in m_1, $\hat{H}_2$ for high input in m_2, and $\hat{L}$ for low input in both of them. In other words, $m_j^{hist[r]}$ is a memory m_j with input history $hist$ applied up to the event number r using input values from channels $\hat{L}$ and $\hat{H}_j$, for $j = 1, 2$.

When the set of escape hatches E is empty, this relation relates all initial high environments. If no input is recorded, the input channels are not essential for this relation, and then it coincides with the simpler indistinguishability relation $I(E)$ from Section 2 (modulo Cartesian product with the set of initial high channels). The release policy is based on the indistinguishability:

$$p(m, \hat{L}, \hat{H}, E, hist) = \{(m', \hat{H}') \mid m_L = m'_L \wedge \\ (m, \hat{H})\ I(E, \hat{L}, hist)\ (m', \hat{H}')\}$$

As in Section 2, the security conditions specify bounds on the attacker's knowledge in terms of the release policy:

Definition 3. (Termination-sensitive security). *A program c is* secure *with respect to a sequence of low events $\vec{\ell}$, initial low-memory i_L, and initial low-communication environment $\hat{L}$, denoted $\widehat{TSec}(c, i_L, \hat{L}, \vec{\ell})$, if for all environments $(m, \hat{H}) \in k(c, i_L, \hat{L}, \vec{\ell})$ that produce low events $\vec{\ell}$ we have:*

$$\forall i\ .\ 1 \le i \le n\ .\ p(m, \hat{L}, \hat{H}, E_i, hist_i) \subseteq k(c, m_L, \hat{L}, \vec{\ell_i})$$

where $\vec{\ell_i}$ is the i-prefix of $\vec{\ell}$, $\vec{\ell} = \vec{\ell_n}$ for some n, and E_i and $hist_i$ are extracted from the configuration that generated the last event in $\vec{\ell_i}$.

Consider an example for illustrating the definition:

$$h' := h; \mathtt{input}(h, H); l := \mathtt{declassify}(h); l' := h'$$

Because declassification happens after the input, it refers to the value of h that has been read from the channel H, rather than its initial value. The last assignment gives to the attacker the knowledge about the initial value of h, which is not what has been released by the declassification. Therefore, this program is rejected.

We now define termination-insensitive security:

Definition 4. (Termination-insensitive security). *A program c is* secure *with respect to a sequence of low events $\vec{\ell}$, initial low-memory i_L and low-communication environment $\hat{L}$, denoted $\widehat{TISec}(c, i_L, \hat{L}, \ell)$, $\forall(m, \hat{H}) \in k(c, i_L, \hat{L}, \vec{\ell})$ that produce low events $\vec{\ell}$ we have: $\forall i \;.\; 1 \leq i \leq n \;.\; p(m, \hat{L}, \hat{H}, E_i, hist_i) \cap \bigcup_{\ell'} k(c, m_L, \hat{L}, \vec{\ell}_{i-1}\ell') \subseteq k(c, m_L, \hat{L}, \vec{\ell}_i)$ where $\vec{\ell}_i$ is the i-prefix of $\vec{\ell}$, $\vec{\ell} = \vec{\ell}_n$ for some n, and E_i and $hist_i$ are extracted from the configuration that generated the last event in $\vec{\ell}_i$.*

Let us consider another example program:

$$\begin{array}{l}\texttt{input}(h, H); l_1 := \texttt{declassify}(h);\\ \texttt{if } h' \texttt{ then input}(h, H) \texttt{ else skip};\\ l_2 := \texttt{declassify}(h)\end{array}$$

Given an initial environment with initial input stream $H_{\texttt{I}} = 1, 2, \ldots$, and initial memory i where $i(h') \neq 0$, this program produces low events $\vec{\ell} = (l_1, 1)(l_2, 2) \downarrow$. We observe that for any i_L and $\hat{L}$ this program satisfies neither $\widehat{TSec}(c, i_L, \hat{L}, \vec{\ell})$ nor $\widehat{TISec}(c, i_L, \hat{L}, \vec{\ell})$.

For the termination-sensitive condition it is sufficient to note that with the second declassification the attacker deduces that h' is non-zero. But h' is never part of any declassification expression, and, hence, the release policy does not place any bounds on it.

For the termination-insensitive condition we also need to argue that permitted values for h' (the left hand side of $\subseteq$ in the definition of $\widehat{TISec}$) are not affected by the progress knowledge. Indeed, termination of this program does not depend on the value of h'.

Enforcement: language We extend the syntax with keywords $\texttt{input}(x, ch)$ and $\texttt{output}(e, ch)$ where ch is either L or H corresponding to the channel level.

The syntax for monitor events is also extended with events $in(x, v)$ for input and $out(ch, e)$ for output. Figure 4 shows monitored semantics for the extended language with the new rule for declassification and the rules for input and output. The new rule for declassification now records the length of the current input history sequence together with the escape-hatch expression.

Rules for input and output are presented separately for every channel. When reading a value from a channel we communicate to the monitor an intention to perform an input and pass the name of the variable which will store the result. If the execution is allowed, we read the value v from the low channel and update the memory. Moreover, we record this input in the input history and communicate back to the monitor the value that has been just read. This interaction with the monitor is denoted by $in(x, v)$ in the input rule. For the low channel we also produce a low-observable event $(\texttt{I}, x, v)$ indicating that a low input has just happened. No low events are produced for inputs on the high channel.

$$\frac{m(e) = v}{\begin{array}{c}\langle x := \texttt{declassify}(e), m, E, \hat{L}, \hat{H}, hist\rangle \xrightarrow{d(x,e,m)}_{(x,v)}\\ \langle stop, m[x \mapsto v], E \cup \{(e, |hist|)\}, \hat{L}, \hat{H}, hist\rangle\end{array}}$$

$$\frac{\hat{L} = [v : L_{\texttt{I}}, L_{\texttt{O}}]}{\begin{array}{c}\langle \texttt{input}(x, L), m, E, \hat{L}, \hat{H}, hist\rangle \xrightarrow{in(x,v)}_{(\texttt{I},x,v)}\\ \langle stop, m[x \mapsto v], E, [L_{\texttt{I}}, L_{\texttt{O}}], \hat{H}, (x, L) : hist\rangle\end{array}}$$

$$\frac{\hat{H} = [v : H_{\texttt{I}}, H_{\texttt{O}}]}{\begin{array}{c}\langle \texttt{input}(x, H), m, E, \hat{L}, \hat{H}, hist\rangle \xrightarrow{in(x,v)}\\ \langle stop, m[x \mapsto v], E, \hat{L}, [H_{\texttt{I}}, H_{\texttt{O}}], (x, H) : hist\rangle\end{array}}$$

$$\frac{m(e) = v \qquad \hat{L} = [L_{\texttt{I}}, L_{\texttt{O}}]}{\begin{array}{c}\langle \texttt{output}(e, L), m, E, \hat{L}, \hat{H}, hist\rangle \xrightarrow{out(L,e)}_{(\texttt{O},v)}\\ \langle stop, m, E, [L_{\texttt{I}}, v : L_{\texttt{O}}], \hat{H}, hist\rangle\end{array}}$$

$$\frac{m(e) = v \qquad \hat{H} = [H_{\texttt{I}}, H_{\texttt{O}}]}{\begin{array}{c}\langle \texttt{output}(e, H), m, E, \hat{L}, \hat{H}, hist\rangle \xrightarrow{nop}\\ \langle stop, m, E, \hat{L}, [H_{\texttt{I}}, v : H_{\texttt{O}}], hist\rangle\end{array}}$$

Fig. 4. Extended command semantics

Semantics for outputs is similar to assignments. We evaluate an expression and send the evaluated value over to the corresponding channel. If the channel is low, we also produce a low-observable output event $(\texttt{O}, v)$.

The semantics for the rest of the commands can be adapted from Figure 8 in a straightforward way as E, $\hat{L}$, $\hat{H}$, and $hist$ parts of the configurations are left intact.

Enforcement: monitoring Figures 5 and 6 present modular extensions to the termination-insensitive and termination-sensitive monitors. Monitor configurations contain just one extra component: input context label ct, which records if there has been an input in a high context. We let $ct = L$ initially. Monitoring outputs in both monitors is similar to monitoring assignments. In the termination-insensitive monitor, the only apparent difference is syntactic: instead of variable names we use channel names; and in the termination-sensitive monitor we do not modify the set of updated variables.

The termination-insensitive monitor disallows low input in a high context, similarly to the assignment rule. This rule also modifies the reference memory of the monitor, which allows declassifications of expressions that refer to the values for variables from most recent input. If the input happens in a high context, the monitor updates the context input label with H.

We extend function $lev(c)$, computing the lower bound on side effects of c: it returns L if there are assignments to low variables or input/output operations

$$\frac{lev(st) \sqsubseteq lev(x)}{\langle i, st, ct\rangle \xrightarrow{in(x,v)} \langle i[x \mapsto v'], st, lev(st) \sqcup ct\rangle}$$

$$\frac{lev(e) \sqsubseteq lev(ch) \qquad lev(st) \sqsubseteq lev(ch)}{\langle i, st, ct\rangle \xrightarrow{out(ch,e)} \langle i, st, ct\rangle}$$

$$\frac{m(e) = i(e) \qquad lev(st) \sqsubseteq lev(x) \qquad ct \sqsubseteq lev(x)}{\langle i, st, ct\rangle \xrightarrow{d(x,e,m)} \langle i, st, ct\rangle}$$

Fig. 5. Extended termination-insensitive monitor

$$\frac{lev(st) = L \Longrightarrow U' = U \backslash \{x\} \quad lev(st) = H \Longrightarrow U' = U}{\langle st, U, ct\rangle \xrightarrow{in(x,v)} \langle st, U', lev(st) \sqcup ct\rangle}$$

$$\frac{lev(st) = L \Longrightarrow lev(e) \sqsubseteq lev(ch)}{\langle st, U, ct\rangle \xrightarrow{out(ch,e)} \langle st, U, ct\rangle}$$

$$\frac{vars(e) \cap U = \emptyset \qquad lev(ct) \sqsubseteq lev(x)}{\langle st, U, ct\rangle \xrightarrow{d(x,e,m)} \langle st, U \cup \{x\}, ct\rangle}$$

$$\frac{\begin{array}{c} lev(e) \sqsubseteq lev(c) \qquad lev(e) = L \Longrightarrow U' = \emptyset \\ lev(e) = H \Longrightarrow noeval(c) \wedge noloop(c) \wedge U' = upd(c) \\ \wedge ct' = inputs(c) \end{array}}{\langle st, U, ct\rangle \xrightarrow{b(e,c)} \langle lev(e) : st, U \cup U', ct \sqcup ct'\rangle}$$

Fig. 6. Extended termination-sensitive monitor

on the low channel in c, and returns H otherwise. We also extend function $upd(c)$, computing the set of variables that are updated in c, to take inputs into account: if there is an input into a variable in c, this variable is included in $upd(c)$. The termination-sensitive monitor disallows low input in a high context thanks to the rule for conditionals, which demands $lev(c) = H$ in high contexts. We let $inputs(c)$ return H if c contains input statements, and L otherwise. The monitor features some *flow-sensitivity*: an input in a low context removes the variable from the set U of variables that have been assigned to. (It is not safe in a high context because the fact that input has occurred carries some information about the context.)

Both monitors disallow declassification if the level of the input context label ct is H. This is necessary because inputs, unlike branch/loop guards are not lexically-bounded in their impact. This is not too restrictive for the examples in Section 5. More liberal treatments of input in high context are possible at the price of complicating the monitors.

In the monitored runs of the program:

$h' := h; \texttt{input}(h, H); l := \texttt{declassify}(h); l' := h'$

the termination-sensitive monitor always stops the execution of this program before the last assignment, thus preventing leakage of the initial value of h.

Given an initial environment with $H_{\mathtt{I}} = 1, 2, \ldots$ and $i(h') \neq 0$ and the program:

$\texttt{input}(h, H); l_1 := \texttt{declassify}(h);$
$\texttt{if}\ h'\ \texttt{then input}(h, H)\ \texttt{else skip};$
$l_2 := \texttt{declassify}(h)$

both monitors accept the first declassification, but stop the program execution before the second one. This program satisfies neither of the security conditions: observing the value of ℓ_2 refines knowledge about h' which does not appear in any of the escape hatches. Section 5 presents further examples on the differences between the monitors.

Soundness The security of the extended monitors is assured by these soundness results:

Proposition 3. *Given a program c, initial memory i, communication environments* $\hat{L}, \hat{H}$ *and a sequence* $\vec{\ell}$ *of low events produced by* $\langle c, i, \emptyset, \hat{L}, \hat{H}, \epsilon\rangle$ *while monitored by monitor* $\langle i, \epsilon, L\rangle$*, we have that c satisfies termination-insensitive security with respect to* $\vec{\ell}$*,* i_L*, and* $\hat{L}$*, that is,* $\widehat{TISec}(c, i_L, \hat{L}, \vec{\ell})$*.*

The proof can be found in Appendix E.

Proposition 4. *Given a program c, initial memory i, communication environments* $\hat{L}, \hat{H}$*, and a sequence* $\vec{\ell}$ *of low events produced by* $\langle c, i, \emptyset, \hat{L}, \hat{H}, \epsilon\rangle$ *while monitored by monitor* $\langle \epsilon, \emptyset, L\rangle$*, we have that c satisfies termination-sensitive security with respect to* $\vec{\ell}$*,* i_L*, and* $\hat{L}$*, that is,* $\widehat{TSec}(c, i_L, \hat{L}, \vec{\ell})$*.*

The proof can be found in Appendix E.

Procedure declarations The enforcement mechanism of this section can be extended to accommodate procedure declarations. One extension is to require that declassification of formal procedure parameters is allowed if the parameters are declared read-only. This allows evaluating escape hatches that involve formal procedure arguments with respect to the reference memory by substituting the actual argument expression for the formal argument variable.

5. Examples

We have implemented the monitors from Sections 3 and 4 for the language of the respective sections. This section reports on experiments with the implementations that illustrate the difference between the monitors and give a flavor of expressiveness that our model provides. In addition, Appendix A works out the auction bidding and dynamic code evaluation examples that we outline in the introduction.

Differences between monitors Recall the average program, which illustrates the precision of the progress-insensitive monitor:

$t := h'; h' := h; h := t; l := \texttt{declassify}(avg(h, h'))$

The progress-insensitive monitor accepts runs of this program because at the time of declassification the escape hatch—the average of h and h'—evaluates to the same value in both current and initial memories. On the other hand, the termination-sensitive monitor stops the execution at the declassification because the latter involves updated variables.

Another example illustrating precision of the progress-insensitive monitor is $\texttt{if}\ h\ \texttt{then}\ \texttt{eval}(\text{"}\texttt{input}(h', H)\text{"})\ \texttt{else skip}$. The progress-insensitive enforcement accepts this program. Since one of the branches contains `eval`, the program is stopped by the termination-sensitive monitor before executing any of the branches.

If instead of the input from a high channel the argument of the `eval` above is a command with a low side effect, e.g., $\texttt{input}(l, L)$, then, provided the value of h is non-zero, the progress-insensitive monitor stops the execution before executing $\texttt{input}(l, L)$.

Password checking Consider a password-checking example. We use a high string variable `password` and assume the rest of the variables are low.

```
1  input(password, H);
2  i  := 0; ok := 0;
3  while i < 3 {
4    input (guess, L);
5    ok:=declassify (password == guess);
6    if ok then {i:=3} else {i:=i+1}
7  };
8  output(ok)
```

The password is read from high input on line 1. In the loop body, which executes at most three times, line 4 obtains the user's guess. Line 5 declassifies the result of the match, which is returned to the user on the low channel (line 8). This program is accepted by both monitors, preventing unintended leakage of the password, but allowing declassification of the match after new input.

It this example we could also read user guess and password into byte arrays and provide an escape hatch that would compare corresponding elements of these arrays.

6. Related work

Monitoring Volpano [38] considers a monitor that only checks explicit flows. Implicit flows are allowed, and no support for declassification policies or dynamic code evaluation is provided. Monitors by Venkatakrishnan et al. [36], Le Guernic et al. [21], [20], and Shroff et al. [32] are in the spirit of our work. However, they address languages without dynamic code evaluation and lack formal support for declassification. The baseline policy for their soundness proofs is "batch-job" noninterference, which does not scale to languages with output [3]. Stopped execution is the only possible deviation from the original semantics in this paper. But we are not critically dependent on this choice. For example, Le Guernic et al. [21], [20] are a bit more liberal: their monitor may suppress or rewrite execution events. We can similarly introduce rewriting and/or suppressing of execution events to win in permissiveness (e.g., allowing evaluation of high strings with suppressed low events) but to lose in the semantic transparency of the monitor. Practical aspects of this trade-off are to be explored. Yu et al. [40] take a description of a runtime monitor for JavaScript as an input and implements this monitor by instrumenting target code. Our monitored semantics can be a starting point for such an implementation. On the other hand, our main results are the semantic properties guaranteed by the monitored executions, whereas Yu et al. report no such results.

Declassification Much progress has been recently made on policies along the *dimensions of declassification* [30] that correspond to *what* information is released, *where* in the systems is released, *when* and by *whom*. Combining the dimensions remains an open challenge [30]. We discuss approaches that, similarly to ours, address both the *what* and *where* dimensions. Our approach subsumes both gradual release [4] and localized delimited release [5] policies (see Appendix D for details). Mantel and Reinhard [22] suggest an approach for expressing both *what* and *where* of declassification in a timing-sensitive setting, which involves hard restrictions on programs to ensure secrets do not affect the execution time. Banerjee et al. [6] use gradual release as a starting point for combining the *what* and *where* of declassification. However, their treatment of *what* differs from ours: their policies are defined with respect to the *current*, not initial state. The difference in this view can be seen in the treatment of program $h := h'; l := \texttt{declassify}(h)$, which we interpret as laundering (see Section 2) but they interpret as secure since the current value of h is intended to be leaked. Extending our framework to reason about the confidentiality with respect to the current state is an intriguing topic. The combination of *what* and *where* by Barthe et al. [7] has similarities to the work by Banerjee et al. [6] in that additional mechanisms need to be placed to prevent information laundering. Neither of these policies considers termination-insensitivity. Similarly to noninterference enforcement, none of the approaches to enforcing declassification handles code generation.

On declassification and untrusted code Note that for scenarios of untrusted code (as in the low-level part of work by Barthe et al. [7] and in applying our work—with or without `eval`—to a client-side setting), it is important to separate the code from the declassification policy. This is especially critical for the *where* aspects of declassification because the attacker should not be able to introduce declassifications in untrusted code. For untrusted code, a simple solution can be to only allow declassification upon method return (cf. [34]) and only as long as the method matches a client-specified signature. Ideas from admissibility[13] can also be used to ensure that declassifications in untrusted code follow a trusted protocol for declassification.

Secure information flow for web security On the other side of the spectrum, there are approaches that target realistic languages but lacking soundness guarantees. A promising line of work by Chong et al. on Sif [10] and SWIFT [9] accommodates secure application programming by compilation. Programmers develop a web application in generalized versions of Jif [25] (an extension of Java with information-flow tracking), which, after security type checking, is compiled into a web application. The source language supports declassification (based on the decentralized label model [24]), but not dynamic code evaluation. No soundness guarantees are provided by this approach. Fable [35] is a framework by Swamy et al. that enforces a wide range of security policies and is implemented as part of the LINKS web-programming language [12]. The framework can statically enforce traditional termination-insensitive noninterference in the presence of references [3] and basic declassification policies [18], but supports neither dynamic code evaluation nor observable side-effects such as outputs. Several web programming languages, such as Perl, PHP, and Ruby, support a *taint* mode, which is an information-flow tracking mechanism for integrity. The taint mode treats input data as untrusted and propagates the taint labels along the computation so that tainted data cannot directly affect sensitive operations. However, this mode does not track implicit flows. Information-flow control as combinations of tainting and static analysis have been suggested by, e.g., Huang et al. [19], Vogt et al. [37] in the context of web applications, and by Chandra and Franz [8] for JVM. However, while promising evidence for scalability of information-flow control, these approaches lack soundness arguments and declassification support. The lack of soundness itself is sometimes the price (unsound aspects of [37] are discussed in [27]). McCamant and Ernst [23] present a tool that computes a quantitative bound on the amount of information a program leaks during a run. This approach provides a limited form of the *what* dimension of declassification, where the release policy boils down to a number of secret bits that an attacker may learn.

7. Conclusion

We have presented a framework for rich release policies and showed how to tightly enforce these policies by hybrids of monitoring and on-the-fly static analysis for a dynamic language with communication primitives. The main contributions of the paper are: (i) a declassification framework that improves and generalizes previous approaches to the *what* and *where* dimensions of declassification; (ii) a sound information-flow enforcement mechanism for a language with dynamic code evaluation; (iii) support for both termination-sensitive and insensitive policies; and (iv) support for communication primitives.

Our results are a step toward bridging the gap between formal approaches that lack rich policies and language features and practical approaches that lack soundness. These results open up new directions, which we are pursuing in our current and future work. The framework can be extended to integrity policies, which is particularly interesting for e.g., mashup security and SQL-injection prevention policies. The *what* and *where* aspects of declassification have dual counterparts in endorsement for integrity [31].

We have enhanced the termination-insensitive monitor to handle the interaction dynamic tree structures, such as those available via the browsers' document object model (DOM) API [27]. We investigate references, dynamic objects, exceptions, and asynchronous communication via `XMLHttpRequest` requests. Each feature corresponds to its own channel for leaks. Our approach is to focus on the most easily exploitable ones (like implicit-flow channel in this paper) first. As a practical study, we plan to extend our prototype to fully-fledged JavaScript and use the case study by Vogt et al. [37] as a starting point. Vogt et al. [37] extend the Firefox browser with a monitor for JavaScript to prevent flow of sensitive information (such as cookies, user inputs, etc.) to the attacker. However, their experiments show that it is often desirable for JavaScript code to leak some information outside the domain of origin: they identify 30 domains such as google-analytics.com that should be allowed *some* leaks. Their solution is to white-list these domains, and therefore allow *any* leaks to these domains, opening up possibilities for laundering. With our approach, these domains can be integrated into a policy of Internet domains as security levels (e.g., in a flat security lattice) with declassification specifications of exactly what can be leaked to which security level, avoiding information laundering.

References

[1] Auction Sniper. http://www.auctionsniper.com.

[2] Google Maps API. http://code.google.com/apis/maps.

[3] A. Askarov and S. Hunt and A. Sabelfeld and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. European Symp. on Research in Computer Security*, pages 333–348, October 2008.

[4] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 207–221, May 2007.

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

[6] A. Banerjee, D. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. IEEE Symp. on Security and Privacy*, pages 339–353, May 2008.

[7] G. Barthe, S. Cavadini, and T. Rezk. Tractable enforcement of declassification policies. In *Proc. IEEE Computer Security Foundations Symposium*, June 2008.

[8] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Proc. Annual Computer Security Applications Conference*, pages 463–475, December 2007.

[9] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. ACM Symp. on Operating System Principles*, pages 31–44, October 2007.

[10] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proc. USENIX Security Symposium*, pages 1–16, August 2007.

[11] D. Clark and S. Hunt. Noninterference for deterministic interactive programs. In *Workshop on Formal Aspects in Security and Trust (FAST'08)*, October 2008.

[12] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links web-programming language. Software release. Located at http://groups.inf.ed.ac.uk/links/, 2006–2008.

[13] M. Dam and P. Giambiagi. Confidentiality for mobile code: The case of a simple payment protocol. In *Proc. IEEE Computer Security Foundations Workshop*, pages 233–244, July 2000.

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

[15] C. Dima, C. Enea, and R. Gramatovici. Nondeterministic nointerference and deducible information flow. Technical Report 2006-01, University of Paris 12, LACL, 2006.

[16] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.

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

[18] B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification:: high-level policy for a security-typed language. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 65–74, June 2006.

[19] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proc. International Conference on World Wide Web*, pages 40–52, May 2004.

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

[21] G. Le Guernic, Anindya Banerjee, Thomas Jensen, and David Schmidt. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference (ASIAN'06)*, volume 4435 of *LNCS*. Springer-Verlag, 2006.

[22] H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In *Proc. European Symp. on Programming*, volume 4421 of *LNCS*, pages 141–156. Springer-Verlag, March 2007.

[23] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proc. ACM SIGPLAN Conference on Programming language Design and Implementation*, pages 193–205, 2008.

[24] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, October 1997.

[25] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001–2006.

[26] K. O'Neill, M. Clarkson, and S. Chong. Information-flow security for interactive programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 190–201, July 2006.

[27] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic dom tree structures, February 2009.

[28] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[29] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, October 2004.

[30] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.

[31] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, 2007. To appear.

[32] P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proc. IEEE Computer Security Foundations Symposium*, pages 203–217, July 2007.

[33] V. Simonet. The Flow Caml system. Software release. Located at http://cristal.inria.fr/~simonet/soft/flowcaml/, July 2003.

[34] S. Smith and M. Thober. Refactoring programs to secure information flows. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 75–84, June 2006.

[35] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 369–383, May 2008.

[36] V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *Proc. International Conference on Information and Communications Security*, pages 332–351. Springer-Verlag, December 2006.

[37] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proc. Network and Distributed System Security Symposium*, February 2007.

[38] D. Volpano. Safety versus secrecy. In *Proc. Symp. on Static Analysis*, volume 1694 of *LNCS*, pages 303–311. Springer-Verlag, September 1999.

[39] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. *Proc. IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.

[40] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 237–249. ACM, 2007.

Appendix A.
Programming examples

Auction bidding The program implements the server-side scenario from Section 1: a third party service that offers incremental bidding at an auction site on behalf of the user. We assume a variable of type `int H: bid` where `bid` is the maximum bid provided by the user. The rest of the variables are low.

```
 1  input (bid, H);
 2  won:=0; proceed := 1;
 3  while proceed {
 4   input (status, L );
 5   if (status == 1) then { // we won
 6     won := 1;  proceed := 0
 7   } else {//get updated bid from auction
 8     input (current, L);
 9     // read new bid from the user
10     input (bid, H);
11     proceed:=declassify(current<bid);
12     if proceed then  {
13      current := current + 1;
14      output (current, L)
15    } else {}
16   }
17  };
18  output(won, H);
19  if won {output (current, H)} else {}
```

We start by asking the user's maximum bid on the high channel. The input on line 4 reads the status of the auction from the auction site value 1 encodes that the auction is over and won. Otherwise, there must have been a new bid placed by someone else which we read on line 8. The new value of the user's bid is read on line 10 from the high channel. Next, we declassify if the user's bid is higher than the current public bid. The result of the declassification is stored in a low variable `proceed` that controls the execution of the main loop. If the user's bid is indeed higher, the program increments the current by a minimum value on line 13 and outputs this values on the public channel. We finish by notifying the user with the results of the auction on lines 18–19. This program is accepted by both progress-insensitive and termination-sensitive monitors.

Dynamic code evaluation This program implements the Google Maps API client-side scenario from Section 1. The map is used to show a user-specified location on the client's web-page. Every time the user enters a new location the browser loads new location-specific code from Google's server (a pattern actually used by Google Maps):

```
while 1 {
 // get location from a high channel
 input (user_location, H);
 // make the location public
 ploc := declassify (user_location);
 output(ploc, L);
 // Get new code that redraws the map
 input (code, L);
 eval  (code)       // run the code
}
```

We assume that the received code may not contain declassification policies (which can be enforced by a simple syntactic check) and let all client data be secret. The only declassification is on line 5 which releases the `user_location` variable before passing it in the request for new code (lines 6–8). This code is then evaluated (line 9). Again, the program is accepted by both progress-insensitive and termination-sensitive monitors.

$$
\begin{aligned}
e &::= n \mid s \mid x \mid e \mathrel{op} e\\
c &::= \mathtt{skip} \mid x := e \mid x := \mathtt{declassify}(e) \mid c; c\\
&\quad \mid \mathtt{if}\ e\ \mathtt{then}\ c\ \mathtt{else}\ c \mid \mathtt{while}\ e\ \mathtt{do}\ c \mid \mathtt{eval}(e)
\end{aligned}
$$

Fig. 7. Syntax

$$
\textsc{Skip}\quad \langle \mathtt{skip}, m, E\rangle \xrightarrow{nop} \langle stop, m, E\rangle
$$

$$
\textsc{Asgn-Low}\quad \frac{m(e) = v \qquad lev(x) = L}{\langle x := e, m, E\rangle \xrightarrow{a(x,e)}_{(x,v)} \langle stop, m[x \mapsto v], E\rangle}
$$

$$
\textsc{Asgn-High}\quad \frac{m(e) = v \qquad lev(x) = H}{\langle x := e, m, E\rangle \xrightarrow{a(x,e)} \langle stop, m[x \mapsto v], E\rangle}
$$

$$
\textsc{Declassify}\quad \frac{m(e) = v}{\langle x := \mathtt{declassify}(e), m, E\rangle \xrightarrow{d(x,e,m)}_{(x,v)} \langle stop, m[x \mapsto v], E \cup \{e\}\rangle}
$$

$$
\textsc{Seq-1}\quad \frac{\langle c_1, m, E\rangle \xrightarrow{\beta}_{\alpha} \langle stop, m', E'\rangle}{\langle c_1; c_2, m, E\rangle \xrightarrow{\beta}_{\alpha} \langle c_2, m', E'\rangle}
$$

$$
\textsc{Seq-2}\quad \frac{\langle c_1, m, E\rangle \xrightarrow{\beta}_{\alpha} \langle c_1', m', E'\rangle}{\langle c_1; c_2, m, E\rangle \xrightarrow{\beta}_{\alpha} \langle c_1'; c_2, m', E'\rangle}
$$

$$
\textsc{If-1}\quad \frac{m(e) = n \qquad n \neq 0}{\langle \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, m, E\rangle \xrightarrow{b(e,c_1;c_2)} \langle c_1; end, m, E\rangle}
$$

$$
\textsc{If-2}\quad \frac{m(e) = 0}{\langle \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, m, E\rangle \xrightarrow{b(e,c_1;c_2)} \langle c_2; end, m, E\rangle}
$$

$$
\textsc{While-1}\quad \frac{m(e) = n \qquad n \neq 0}{\langle \mathtt{while}\ e\ \mathtt{do}\ c, m, E\rangle \xrightarrow{we(e)} \langle c; end; \mathtt{while}\ e\ \mathtt{do}\ c, m, E\rangle}
$$

$$
\textsc{While-2}\quad \frac{m(e) = 0}{\langle \mathtt{while}\ e\ \mathtt{do}\ c, m, E\rangle \xrightarrow{we(e)} \langle end, m, E\rangle}
$$

$$
\textsc{Eval}\quad \frac{m(e) = s \qquad parse(s) = c}{\langle \mathtt{eval}(e), m, E\rangle \xrightarrow{we(e)} \langle c; end, m, E\rangle}
$$

$$
\textsc{End}\quad \langle end, m, E\rangle \xrightarrow{f} \langle stop, m, E\rangle
$$

Fig. 8. Command semantics

Appendix B. Language

Figure 7 presents the syntax and Figure 8 presents the semantics of the simple language from Section 3.

Appendix C. Multiple channels

The development from Section 4 can be easily generalized to a setting of more than two channels. Assume channels are identified by channel names $ch \in \mathit{ChId}$. We tag low input and output events with channel identities:

$$\ell ::= \ldots \mid (\mathtt{I}_{ch}, x, v) \mid (\mathtt{O}_{ch}, v)$$

and define input and output environments $\hat{L}$ and $\hat{H}$ as mappings from channel names to pairs of input and output channels:

$$\hat{L} = \{ch \mapsto [L_{\mathtt{I}}, L_{\mathtt{O}}]\}_{ch \in \mathit{ChId}} \qquad\qquad \hat{H} = \{ch \mapsto [H_{\mathtt{I}}, H_{\mathtt{O}}]\}_{ch \in \mathit{ChId}}$$

Semantic rules from Figure 4 and enforcement rules from Figures 5 and 6 are then modified to operate on extended environments and produce tagged low events.

Appendix D.
Relation to gradual and localized delimited release

We demonstrate that our framework subsumes two definitions from the literature: *gradual release* [4] and *localized delimited release* [5]. One major improvement over both definitions is the treatment of termination-insensitivity. While these definitions let the attacker observe intermediate states, they simply ignore diverging runs (as is common in "batch-job" models). This is not satisfactory because accepted programs are allowed to leak the entire secret provided they enter an infinite loop [6], [3]. Instead of ignoring diverging runs, definition *TISec* provides insensitivity to divergence at any given step. We now discuss further relation to the gradual and localized delimited release properties.

D.1. Relation to gradual release

Gradual release is a knowledge-based condition, which only addresses the *where* of declassification. It ensures that refinements of knowledge are only allowed at declassification points. It leaves unspecified *what* can be leaked by each declassification.

As we point out above, the definition of gradual release operates on terminating traces. In order to establish formal relation, we need to constrain *TISec* to terminating traces. For this we use the notion of *initial knowledge* [4], which corresponds to all initial memories from which we can reach termination:

$$k(c, i_L) = \{m \mid i_L = m_L \ \wedge \ \langle m, c, \emptyset \rangle \longrightarrow^{*}_{\vec{\ell}\downarrow} \langle c', m', E' \rangle\}$$

Using the initial knowledge, we can define batch-job style termination-insensitive knowledge $k_{\downarrow}(c, i_L, \vec{\ell})$, i.e., the initial memories whose low projection is i_L and that can generate low-event sequence $\vec{\ell}$ as a part of a terminating trace of program c:

$$k_{\downarrow}(c, i_L, \vec{\ell}) = k(c, i_L, \vec{\ell}) \cap k(c, i_L)$$

The gradual release definition permits changes in the knowledge only at declassification points. In our notation:

Definition 5. (GRADUAL RELEASE). *A command c satisfies* gradual release *if for all m_L and all low-event sequences $\vec{\ell_n} = \ell_1 \ldots \ell_n$ that are generated by c from memories whose low projection is m_L where $\ell_{r_1}, \ldots, \ell_{r_m}$ are all declassification events, we have for all $i \in \{1, \ldots, n\}$:*

$$(\forall j \,.\, r_j \neq i) \implies k_{\downarrow}(c, m_L, \vec{\ell}_{i-1}) = k_{\downarrow}(c, m_L, \vec{\ell_i})$$

For the purpose of comparison with gradual release, we use a ↓-variant of policy $p_{\downarrow}(c, i_L, E) = p(i_L, E) \cap k(c, i_L)$ and let $TSec_{\downarrow}$ correspond to a variant of Definition 1 where the policy and knowledge are replaced with their ↓-versions.

While gradual release demands that the knowledge can only be refined at declassification points, *TISec* is more liberal in that the knowledge can be refined even after declassification events. For example, program:

$$h' := h; h := 0; l := \texttt{declassify}(h); l := h'$$

is rejected by gradual release because the attacker gains knowledge at the last assignment, which is not a declassification event. Runs of the program are accepted by *TISec* because the knowledge about h (which is gained at the last assignment) is allowed by prior declassification.

On the other hand, *TISec* controls what is released (which gradual release is agnostic about). Recall the laundering attack program:

$$h := h'; l := \texttt{declassify}(h)$$

It is accepted by gradual release because the attacker's knowledge changes at a declassification event. But, as we discussed earlier, *TISec* rejects laundering by this program because the program allows gaining knowledge about h', and not about h from the declassification policy.

The following theorem establishes a formal relation. In short, if we force a declassification event to actually release the information it declares it releases, then *TISec* implies gradual release. In terms of the diagram in Figure 1, this corresponds to a gray area that fills the entire space under the solid line.

Proposition 5. (RELATION TO GRADUAL RELEASE). *Obtain $T(c)$ from c by replacing declassification $l :=$ $\texttt{declassify}(e)$ with $l' := \texttt{declassify}(e_g); l := \texttt{declassify}(e)$ where l' is a fresh variable and e_g is obtained from e by replacing each variable x with its "ghost" version x_g. Then, if for all low-event sequences $\vec{\ell}$ that start in a memory m and are generated by eventually terminating traces, where variables agree with their ghost versions ($\forall x.\, m(x) = m_g(x)$), we have $TSec_{\downarrow}(T(c), m_L, \vec{\ell})$ then $T(c)$ satisfies gradual release.*

Proof: We pick a $m \in k_{\downarrow}(T(c), m_L)$. Consider a trace started in m producing a sequence of low events $\vec{\ell}_{n\downarrow}$

$$\langle T(c), m, \emptyset\rangle \longrightarrow^*_{\vec{\ell}_{t-1}} \langle d_{t-1}, m_{t-1}, E_{t-1}\rangle \longrightarrow_{\ell_t} \langle d_t, m_t, E_t\rangle \longrightarrow^*_{\vec{\ell}_{t+1\dots n\downarrow}} \langle d_n, m_n, E_n\rangle$$

where ℓ_t is not a declassification event. Observe that the transformation T guarantees that the policy set is fully included into the knowledge: $\forall t\ 1 \leq t \leq n.\ p_{\downarrow}(m, E_t) \supseteq k_{\downarrow}(T(c), m_L, \vec{\ell_t})$. On the other hand, $TSec_{\downarrow}(T(c), m_L, \vec{\ell})$ implies that $\forall t. 1 \leq t \leq n\ .\ p_{\downarrow}(m, E_t) \subseteq k_{\downarrow}(T(c), m_L, \vec{\ell_t})$. Therefore, we obtain that $\forall t. 1 \leq t \leq n\ .\ p_{\downarrow}(m, E_t) = k_{\downarrow}(T(c), m_L, \vec{\ell_t})$.

In particular, $p_{\downarrow}(m, E_{t-1}) = k_{\downarrow}(T(c), m_L, \vec{\ell}_{t-1})$ and $p_{\downarrow}(m, E_t) = k_{\downarrow}(T(c), m_L, \vec{\ell_t})$. If ℓ_t is not a declassification event then $E_{t-1} = E_t$, and hence $k_{\downarrow}(T(c), m_L, \vec{\ell}_{t-1}) = k_{\downarrow}(T(c), m_L, \vec{\ell_t})$, which implies that $T(c)$ satisfies gradual release.

□

D.2. Relation to localized delimited release

The localized delimited release definition addresses both *what* is declassified and *where* it is declassified. Not only *TISec* provides an adequate treatment of termination-insensitivity (as discussed above), but also breaks away from unnecessary conservativeness of localized delimited release.

Memories m_1 and m_2 are *low-equivalent*, written $m_1 =_L m_2$, if they agree on the low variables. We recall the definition of localized delimited release [5] adapted to the semantics with low events. First we define an indistinguishability on configurations via a bisimulation relation:

Definition 6. (LOW BISIMULATION). *Given memories i_1 and i_2 a symmetric relation R_{i_1,i_2} on configurations is an* i_1, i_2-low bisimulation *if, for all $c_1, c_2, m_1, m_2, E_1, E_2, \vec{\ell_1}$, and $\vec{\ell_2}$, we have (i) both $\langle c_1, m_1, E_1\rangle$ and $\langle c_2, m_2, E_2\rangle$ terminate and (ii) $\langle c_1, m_1, E_1\rangle\ R_{i_1,i_2}\ \langle c_2, m_2, E_2\rangle$ then*

1) *$i_1\ I(E_1)\ i_2$ if and only if $i_1\ I(E_2)\ i_2$, and*
2) *if $i_1\ I(E_1)\ i_2$ then (i) $m_1 =_L m_2$ and (ii) if $\langle c_1, m_1, E_1\rangle \longrightarrow_{\alpha} \langle c_1', m_1', E_1'\rangle$ then $\langle c_2, m_2, E_2\rangle \longrightarrow^*_{\alpha} \langle c_2', m_2', E_2'\rangle$ and also $\langle c_1', m_1', E_1'\rangle\ R_{i_1,i_2}\ \langle c_2', m_2', E_2'\rangle$ for some c_2', m_2', and E_2'.*

Two configurations cfg_1 and cfg_2 are i_1, i_2-low-bisimilar *(written $cfg_1 \sim_{i_1,i_2} cfg_2$) if there exists an i_1, i_2-low bisimulation that relates them.*

Localized delimited release for a given program demands indistinguishability of any two initial configurations for the program with low-equivalent memories:

Definition 7. (LOCALIZED DELIMITED RELEASE). *A command c satisfies localized delimited release with low events if for all m_1 and m_2 such that $m_1 =_L m_2$ we have $\langle c, m_1, \emptyset\rangle \sim_{m_1,m_2} \langle c, m_2, \emptyset\rangle$.*

We state the main formal result of this section:

Proposition 6. (RELATION TO LOCALIZED DELIMITED RELEASE). *If c satisfies localized delimited release then for all low-memories i_L and low-event sequences $\vec{\ell}$ generated by eventually terminating traces, we have $TSec_{\downarrow}(c, i_L, \vec{\ell})$.*

Proof: Consider a terminating trace t of a program c_0 starting with an initial memory i and producing a sequence of low events $\vec{\ell}$:

$$\langle i, c_0, \emptyset\rangle \longrightarrow^*_{\vec{\ell}} \langle m', c', E\rangle$$

We know that c_0 satisfies localized delimited release and we need to show that $p_{\downarrow}(i, E) \subseteq k_{\downarrow}(c_0, i_L, \vec{\ell})$. For this we pick $m' \in p_{\downarrow}(i, E)$ and we aim to prove that $m' \in k_{\downarrow}(c_0, i_L, \vec{\ell})$. Since by definition $p_{\downarrow}(i, E)$ contains only memories that lead to terminating configurations, we know that there exists a terminating trace

$$\langle c_0, m', \emptyset\rangle \longrightarrow^*_{\vec{\ell}''} \langle c'', m'', E''\rangle.$$

We have to show that $\vec{\ell}$ can be a subsequence of $\vec{\ell}''$, i.e. it can be produced by a subtrace of the above trace. To show that such subtrace exists we proceed by induction on the length of $\vec{\ell}$. We also prove and use that pairs of configurations that yield low events of the same length (up to the length of $\vec{\ell}$) when started in the memories i and m' respectively are related by i, m'-low bisimulation.

- Base case. $\vec{\ell} = \epsilon$. In this case $\langle c_0, m', \emptyset\rangle \xrightarrow{0}_{\epsilon} \langle c_0, m', \emptyset\rangle$. Moreover, since c_0 satisfies Localized Delimited Release, we have that $\langle c_0, i, \emptyset\rangle \sim_{i,m'} \langle c_0, m', \emptyset\rangle$.
- Induction step. $\vec{\ell} = \vec{\ell}\alpha$. We assume the induction hypothesis for $\vec{\ell}$:

$$\langle c_0, i, \emptyset\rangle \xrightarrow{*}_{\vec{\ell}} \langle c^1, m^1, E^1\rangle$$

$$\langle c_0, m', \emptyset\rangle \xrightarrow{*}_{\vec{\ell}} \langle c^2, m^2, E^2\rangle$$

and $\langle c^1, m^1, E^1\rangle \sim_{i,m'} \langle c^2, m^2, E^2\rangle$. The latter implies that $i\ (E_1)\ '$ iff $i\ I(E_2)\ m'$. We also know that since $m' \in p_{\downarrow}(i, E)$ it holds that $i\ I(E)\ m'$, and since $E_1 \subseteq E$ it holds that $i\ I(E^1)\ m'$. Then by item *2(ii)* of Definition 6 when

$$\langle c^1, m^1, E^1\rangle \longrightarrow_{\alpha} \langle c^3, m^3, E^3\rangle$$

then

$$\langle c^2, m^2, E^2\rangle \xrightarrow{*}_{\alpha} \langle c^4, m^4, E^4\rangle$$

such that $\langle c^3, m^3, E^3\rangle \sim_{i,m'} \langle c^4, m^4, E^4\rangle$.

□

Because localized delimited release also tracks both what and where of declassification, the examples from Section D.1 behave in the same way for *TISec* and localized delimited release. However, we show that localized delimited release is more restrictive than *TISec*. Recall that localized delimited release implies $TSec_{\downarrow}(c, i_L, \vec{\ell})$ for all low-memories i_L and low-event sequences $\vec{\ell}$ generated by eventually terminating traces by Proposition 6. The converse of this proposition does not hold. The reason is an unnecessary conservativeness of localized delimited release, which we illustrate in the following example:

$$h' := 0; \texttt{if}\ h\ \texttt{then}\ l := \texttt{declassify}(h')\ \texttt{else}\ l := 0$$

This program is intuitively secure because the final value of l is always 0. However, this program is rejected by localized delimited release because it insists that the indistinguishability of the initial memory by the escape-hatch set is invariant (condition 1 in Definition 6). This anomaly breaks the *monotonicity of release* [30] principle for localized delimited release. However, although the policy can be modified, the knowledge is always unchanged. Thus, the set inclusion of *TISec* holds for all runs of the program. This shows that the converse of Proposition 6 does not hold.

Appendix E.
Proofs for Extended Language

The following two sections present proofs of termination-insensitive and termination-sensitive security. Since the extended language presented in Section 4 fully subsumes the simple language of Section 2 we limit the presentation to extended language only. We state the key lemmas and show proofs of propositions 3 and 4.

The proof of Proposition 3 is given in Appendix F and the proof of Proposition 4 is given in Appendix G.

We refer to termination-insensitive monitor as ι and to termination-sensitive monitor as σ. We assume that attacker knows monitor used for enforcement, and parameterize our definition of knowledge with it. We use notation $k^{\mu}(c, i_L, \vec{\ell})$ to be explicit that transitions that generate the low events are monitored. We use notation $k(c, i_L, \vec{\ell})$ when the monitored semantics is obvious from the context.

E.1. Input histories revisited

In this paragraph we give formal definition of input histories and indistinguishability relation informally defined in Section 4. For this, we introduce function $memupd(m, \hat{L}, \hat{H}, hist)$ of four arguments that defines an update of memory m from the input streams $\hat{L}$ and $\hat{H}$, using input recorded in the input history $hist$. As one can see from the definition of this function, shown on Figure 9, memory update replays input history recorded in $hist$.

Definition 8 (Indistinguishability of initial high environments)**.**

$$(m_1, \hat{H}_1)\ I(E, \hat{L}, hist)\ (m_2, \hat{H}_2) \Leftrightarrow \forall (e, r) \in E\ .\ m_1^{hist[r]}(e) = m_2^{hist[r]}(e)$$

where

- $hist = (ch_n, x_n) : (ch_{n-1}, x_{n-1}) : \ldots : (ch_1, x_1)$,
- $r \leq n$,
- $hist[r] = (ch_r, x_r) : (ch_{r-1}, x_{r-1}) : \ldots : (ch_1, x_1)$
- $m_j^{hist[r]} = memupd(m, \hat{L}, \hat{H}_j, hist[r])$ *for* $j = 1, 2$

Definition 9 (Low-consistency of input histories)**.**

- *We say that an input history* $hist$ *contains no explicit flows if none of the entries in* $hist$ *are of the form* (x, L) *such that* $lev(x) = H$.
- *Low-projection of an input history* $hist$ *is a sub-sequence of it denoted as* $hist_L$ *obtained by stripping off entries of the form* (x, H) *from* $hist$.
- *Two input histories* $hist_1$ *and* $hist_2$ *are low-consistent if none of them contains explicit flows and* $hist_{1L} = hist_{2L}$.

Definition 10 (Consistent monitor configurations)**.**

1) *Two termination-insensitive monitor configurations* $\langle i_1, st_1, ct_1 \rangle$ *and* $\langle i_2, st_2, ct_2 \rangle$ *are consistent whenever* $i_{1L} = i_{2L}$ *and* $st_1 = st_2$.
2) *Two termination-sensitive monitor configurations* $\langle st_1, U_1, ct_1 \rangle$ *and* $\langle st_2, U_2, ct_2 \rangle$ *are consistent whenever* $U_1 = U_2$ *and* $st_1 = st_2$.

Appendix F. Proofs for Extended Language: Termination-Insensitive Enforcement

F.1. Monitor consistency

Lemma 1 (Monitor configuration through high context)**.** *Given a trace*

$$(\langle c, m, \hat{L}, \hat{H}, E, hist \rangle, \langle i, st, ct \rangle) \longrightarrow^* (\langle c', m', \hat{L}, \hat{H}', E, hist' \rangle, \langle i', st', ct \rangle)$$

where

1) *the context stack of the first configuration has a form* $st = lev_n \ldots lev_{k+1} : H : lev_{k-1} \ldots lev_1$ *with* lev_n *on the top of the stack and* lev_1 *on the bottom of it.*
2) *the execution of this trace does not consume stack entries below and including* $lev_k = H$

then the starting and ending reference memories are low-equal: $i_L = i'_L$.

Proof: By induction on c. □

$$memupd(m, \hat{L}, \hat{H}, hist) = m \text{ if } hist = \epsilon$$

$$memupd(m, v : \hat{L}, \hat{H}, hist : (L, x)) = memupd(m[x \mapsto v], \hat{L}, \hat{H}, hist)$$

$$memupd(m, \hat{L}, v : \hat{H}, hist : (H, x)) = memupd(m[x \mapsto v], \hat{L}, \hat{H}, hist)$$

Fig. 9. Memory update

Lemma 2 (Relationship between reference memory and input history)**.** *Given a trace*

$$(\langle d, i, \hat{L}, \hat{H}, \emptyset, \epsilon\rangle, \langle i, st, L\rangle) \longrightarrow^*_{\vec{\ell}} (\langle c', m', \hat{L}', \hat{H}', E', hist'\rangle, \langle i', st', ct'\rangle)$$
$$\longrightarrow^*_{\vec{\ell}'} (\langle c'', m'', \hat{L}'', \hat{H}'', E'', hist''\rangle, \langle i'', st'', ct''\rangle)$$

where we have that

- $i' = memupd(i, \hat{L}, \hat{H}, hist')$

then also

- $i'' = memupd(i, \hat{L}, \hat{H}, hist'')$

Proof: By induction on the length of the trace starting from c'. □

F.2. Sequential composition

This section contains a series of lemmas exploring properties of sequential composition.

Lemma 3. *Given ι_1 and ι_2 that are*

1) *two consistent termination-insensitive monitor configurations such that*
2) *both ι_1 and ι_2 are faithful with respect to the* same *source program configuration and two final configurations $\langle c_a; c_b, m_1, \hat{L}_1, \hat{H}_1, E, hist_1\rangle$ and $\langle c_a; c_b, m_2, \hat{L}_2, \hat{H}_2, E, hist_2\rangle$ respectively, where*
3) $m_{1L} = m_{2L}, hist_{1L} = hist_{2L}$
4) *and we know of two transition sequences*

$$(\langle c_a; c_b, m_1, \hat{L}, \hat{H}_1, E, hist_1\rangle, \iota_1) \longrightarrow^* (\langle c_b, m_1', \hat{L}, \hat{H}_1', E, hist_1'\rangle, \iota_1') \longrightarrow^*_{\ell_1} (\langle c'', m_1'', \hat{L}_1'', \hat{H}_1'', E_1'', hist_1''\rangle, \iota_1'')$$

and

$$(\langle c_a; c_b, m_2, \hat{L}, \hat{H}_2, E, hist_2\rangle, \iota_2) \longrightarrow^*_{\ell_2} (\langle d'', m_2'', \hat{L}_2'', \hat{H}_2'', E_2'', hist_2''\rangle, \iota_2'')$$

then

1) *for the second trace we have that*

$$(\langle c_a; c_b, m_2, \hat{L}, \hat{H}_2, E, hist_2\rangle, \iota_2) \longrightarrow^* (\langle c_b, m_2', \hat{L}, \hat{H}_2', E, hist_2'\rangle, \iota_2') \longrightarrow^*_{\ell_2} (\langle d'', m_2'', \hat{L}_2'', \hat{H}_2'', E_2'', hist_2''\rangle, \iota_2'')$$

2) $m'_{1L} = m'_{2L}, hist'_{1L} = hist'_{2L}$
3) *ι_1' and ι_2' are consistent*

Proof: By induction on the structure of c_a using Lemma 1.

□

Lemma 4. *Given ι_1 and ι_2 that are*

1) *two consistent termination-insensitive monitor configurations such that*
2) *both ι_1 and ι_2 are faithful with respect to the* same *source program configuration and two final configurations $\langle c_a; c_b, m_1, \hat{L}_1, \hat{H}_1, E, hist_1\rangle$ and $\langle c_a; c_b, m_2, \hat{L}_2, \hat{H}_2, E, hist_2\rangle$ respectively, where*
3) $m_{1L} = m_{2L}, hist_{1L} = hist_{2L}$
4) *and we know of two transition sequences*

$$(\langle c_a; c_b, m_1, \hat{L}, \hat{H}_1, E, hist_1\rangle, \iota_1) \longrightarrow^* (\langle c_a'; c_b, m_1', \hat{L}, \hat{H}_1', E, hist_1'\rangle, \iota_1') \longrightarrow^*_{\ell_1} (\langle c'', m_1'', \hat{L}_1'', \hat{H}_1'', E_1'', hist_1''\rangle, \iota_1'')$$

and

$$(\langle c_a; c_b, m_2, \hat{L}, \hat{H}_2, E, hist_2\rangle, \iota_2) \longrightarrow^*_{\ell_2} (\langle d'', m_2'', \hat{L}_2'', \hat{H}_2'', E_2'', hist_2''\rangle, \iota_2'')$$

then

1) *for the second trace we have that*

$$(\langle c_a; c_b, m_2, \hat{L}, \hat{H}_2, E, hist_2\rangle, \iota_2) \longrightarrow^* (\langle c_a'; c_b, m_2', \hat{L}, \hat{H}_2', E, hist_2'\rangle, \iota_2') \longrightarrow^*_{\ell_2} (\langle d'', m_2'', \hat{L}_2'', \hat{H}_2'', E_2'', hist_2''\rangle, \iota_2'')$$

2) $m'_{1L} = m'_{2L}, hist'_{1L} = hist'_{2L}$
3) *ι_1' and ι_2' are consistent*

4) $c'' = d''$, $\hat{L}''_1 = \hat{L}''_2$, $E''_1 = E''_2, hist''_{1L} = hist''_{2L}$
5) ι''_1 *and* ι''_2 *are consistent*
6) *if* ℓ_1 *is not a declassification event then*
   - $m''_{1L} = m''_{2L}$ *and* $\ell_1 = \ell_2$
7) *if* ℓ_1 *is a declassification event then*
   - ℓ_2 *is a declassification event as well.*

Proof: By induction on the structure of c_a using Lemma 1.

□

F.3. Advancement Lemma

Lemma 5. *Given* ι_1 *and* ι_2 *that are*

1) *two consistent termination-insensitive monitor configurations such that*
2) *both* ι_1 *and* ι_2 *are faithful with respect to the* same *source program configuration and two final configurations* $\langle c, m_1, \hat{L}_1, \hat{H}_1, E, hist_1\rangle$ *and* $\langle c, m_2, \hat{L}_2, \hat{H}_2, E, hist_2\rangle$ *respectively, where*
3) $m_{1L} = m_{2L}, hist_{1L} = hist_{2L}$
4) *and we know of two transition sequences*

$$(\langle c, m_1, \hat{L}, \hat{H}_1, E, hist_1\rangle, \iota_1)\longrightarrow^*(\langle c', m'_1, \hat{L}, \hat{H}'_1, E, hist'_1\rangle, \iota'_1)\longrightarrow^*_{\ell_1}(\langle c'', m''_1, \hat{L}''_1, \hat{H}''_1, E''_1, hist''_1\rangle, \iota''_1)$$

and

$$(\langle c, m_2, \hat{L}, \hat{H}_2, E, hist_2\rangle, \iota_2)\longrightarrow^*_{\ell_2}(\langle d'', m''_2, \hat{L}''_2, \hat{H}''_2, E''_2, hist''_2\rangle, \iota''_2)$$

then

1) *for the second trace we have that*

$$(\langle c, m_2, \hat{L}, \hat{H}_2, E, hist_2\rangle, \iota_2)\longrightarrow^*(\langle c', m'_2, \hat{L}, \hat{H}'_2, E, hist'_2\rangle, \iota'_2)\longrightarrow^*_{\ell_2}(\langle d'', m''_2, \hat{L}''_2, \hat{H}''_2, E''_2, hist''_2\rangle, \iota''_2)$$

2) $m'_{1L} = m'_{2L}, hist'_{1L} = hist'_{2L}$
3) ι'_1 *and* ι'_2 *are consistent*
4) $c'' = d''$, $\hat{L}''_1 = \hat{L}''_2$, $E''_1 = E''_2, hist''_{1L} = hist''_{2L}$
5) ι''_1 *and* ι''_2 *are consistent*
6) *if* ℓ_1 *is not a declassification event then*
   - $m''_{1L} = m''_{2L}$ *and* $\ell_1 = \ell_2$
7) *if* ℓ_1 *is a declassification event then*
   - ℓ_2 *is a declassification event as well.*

Proof: By induction on c using sequential composition Lemmas 3 and 4. □

F.4. Backbone Lemma

Lemma 6 (Backbone Lemma)**.** *Given a program* c *and an enforced trace started from configuration* $(\langle c, i, \hat{L}, \hat{H}, \emptyset, \epsilon\rangle, \langle i, \epsilon, L\rangle)$ *which produces a sequence of low events* $\vec{\ell_n}$*:*

$$(\langle c, i, \hat{L}, \hat{H}, \emptyset, \epsilon\rangle, \langle i, \epsilon, L\rangle)\longrightarrow^*(\langle c_1, i_1, \hat{L}, \hat{H}_1, \emptyset, hist_1\rangle, \iota_1)\longrightarrow_{\ell_1}(\langle c'_1, i'_1, \hat{L}_1, \hat{H}_1, E_1, hist'_1\rangle, \iota'_1)\longrightarrow^*_{\ell_2\ldots\ell_{j-1}}$$
$$(\langle c_j, i_j, \hat{L}_{j-1}, \hat{H}_j, E_{j-1}, hist_j\rangle, \iota_j)\longrightarrow_{\ell_j}(\langle c'_j, i'_j, \hat{L}_j, \hat{H}'_j, E_j, hist'_j\rangle, \iota'_j)\longrightarrow^*_{\ell_{j+1}\ldots\ell_{n-1}}$$
$$(\langle c_n, i_n, \hat{L}_{n-1}, \hat{H}_n, E_{n-1}, hist_n\rangle, \iota_n)\longrightarrow_{\ell_n}(\langle c'_n, i'_n, \hat{L}_n, \hat{H}'_n, E_n, hist'_n\rangle, \iota'_n)$$

and an initial high environment $(m, \hat{S}) \in k^\iota(c, i_L, \hat{L}, \vec{\ell_n})$*, there exist* $m_1, m'_1, \hat{S}_1, \hat{S}'_1, s_1, s'_1, \kappa_1, \kappa'_1, \ldots m_n, m'_n, \hat{S}_n, \hat{S}'_n, s_n, s'_n, \kappa'_n, \kappa'_n$ *such that* $\forall\ j\ .\ 1 \leq j \leq n$ *we have*

1) $m_{j\,L} = i_{j\,L}, m'_{j\,L} = i'_{j\,L}$*, and*
2) $hist_j, s_j$ *and* $hist'_j, s'_j$ *are pairwise consistent input histories,*
3) ι_j, κ_j *and* ι'_j, κ'_j *are pairwise consistent monitor configurations,*

4) *for the trace starting with m and $\hat{S}$ we have*

$$\begin{aligned}(\langle c, m, \hat{L}, \hat{S}, \emptyset, \epsilon\rangle, \langle m, \epsilon, L\rangle) \longrightarrow^{*} (\langle c_1, m_1, \hat{L}, \hat{S}_1, \emptyset, s_1\rangle, \kappa_1) \longrightarrow_{\ell_1} (\langle c_1', m_1', \hat{L}_1, \hat{S}_1', E_1, s_1'\rangle, \kappa_1) \longrightarrow^{*}{}_{\ell_2 \ldots \ell_{j-1}} \\ (\langle c_j, m_j, \hat{L}_{j-1}, \hat{S}_j, E_{j-1}, s_j\rangle, \kappa_j) \longrightarrow_{\ell_j} (\langle c_j', m_j', \hat{L}_j, \hat{S}_j', E_j, s_j'\rangle, \kappa_j') \longrightarrow^{*}{}_{\ell_{j+1} \ldots \ell_{n-1}} \\ (\langle c_n, m_n, \hat{L}_{n-1}, \hat{S}_n, E_{n-1}, s_n\rangle, \kappa_n) \longrightarrow_{\ell_n} (\langle c_n', m_n', \hat{L}_n, \hat{S}_n', E_n, s_n'\rangle, \kappa_n')\end{aligned}$$

Proof: By induction on n using Advancement Lemma 5.

□

F.5. Termination-insensitive security

Restatement of Proposition 3. Given a program c, initial memory i, communication environments $\hat{L}, \hat{H}$ and a sequence $\vec{\ell}$ of low events produced by $\langle c, i, \emptyset, \hat{L}, \hat{H}, \epsilon\rangle$ while monitored by monitor $\langle i, \epsilon, L\rangle$, we have that c satisfies termination-insensitive security with respect to $\vec{\ell}$, i_L, and $\hat{L}$, that is, $\widehat{TISec}(c, i_L, \hat{L}, \vec{\ell})$.

Proof: By induction on the length of the sequence of low events $\vec{\ell}$.

- Base case. $\vec{\ell} = \epsilon$. In this case we have n = 0, and, therefore, the main expression of the security definition trivially holds.
- Induction step. We assume the proposition holds for events of length up to $n-1$ and need to prove that if

$$\langle c, i, \hat{L}, \hat{H}, \emptyset, hist\rangle \longrightarrow^{*}_{\vec{\ell}_n} \langle c_n', i_n', \hat{L}_n, \hat{H}_n, E_n, hist_n\rangle$$

is a trace enforced by ι then $\forall m \in k^{\iota}(c, i_L, \hat{L}, \vec{\ell}_n)$.

$$p(m, \hat{L}, \hat{H}, E_n, hist_n) \cap \bigcup_{\ell'} k^{\iota}(c, m_L, \hat{L}, \vec{\ell}_{n-1}\ell') \subseteq k^{\iota}(c, m_L, \hat{L}, \vec{\ell}_n).$$

We consider two cases for the low event ℓ_n

1) ℓ_n is not a declassification event.
In this case we apply Backbone Lemma 6 followed by Advancement Lemma 5 which gives us that all possible ℓ' in the term corresponding to progress knowledge must be equal to ℓ_n. This allows us to state that

$$\bigcup_{\ell'} k^{\iota}(c, m_L, \hat{L}, \vec{\ell}_{n-1}\ell') \subseteq k^{\iota}(c, m_L, \vec{\ell}_n)$$

We can intersect the left hand side of this expression with $p(m, \hat{L}, \hat{H}, E_n, hist_n)$ obtaining the desired inequality.

2) ℓ_n is a declassification event.
We assume contrary, that is, there exists an initial high environment $(m, \hat{S}) \in k^{\iota}(c, m_L, \hat{L}, \vec{\ell}_n)$ such that

$$p(m, \hat{L}, \hat{H}, E_n, hist_n) \cap \bigcup_{\ell'} k^{\iota}(c, m_L, \hat{L}, \vec{\ell}_{n-1}\ell') \supset k^{\iota}(c, m_L, \hat{L}, \vec{\ell}_n).$$

In other words there are $(m^\star, \hat{S}^\star)$ such that $(m^\star, \hat{S}^\star) \in p(m, \hat{L}, \hat{H}, E_n, hist_n)$ and $(m^\star, \hat{S}^\star) \in k^{\iota}(c, m_L, \hat{L}, \vec{\ell}_{n-1}\ell')$ for $\ell' \neq \ell_n$, but $(m^\star, \hat{S}^\star) \notin k^{\iota}(c, m_L, \hat{L}, \vec{\ell}_n)$.
Next, $(m^\star, \hat{S}^\star) \in p(m, \hat{L}, \hat{H}, E_n, hist_n) \implies m^\star =_L m \wedge (m^\star, \hat{S}^\star) I(E_n, \hat{L}, hist_n)(m, \hat{H})$. That is, $memupd(m^\star, \hat{L}, \hat{S}^\star, hist_n) = memupd(m^\star, \hat{L}, \hat{H}^\star, hist_n)$. Since $(m, \hat{H}) \in k^{\iota}(c, m_L, \hat{L}, \vec{\ell}_n)$ then, using Lemma 2, $m_n(e) = memupd(m, \hat{L}, \hat{H}, hist_n)(e)$. Similarly, since $(m^\star, \hat{S}^\star) \in k^{\iota}(c, m_L, \hat{L}, \vec{\ell}_{n-1}\ell')$ then we have that $m_n^\star(e) = memupd(m^\star, \hat{L}, \hat{S}^\star, hist_n^\star)(e)$. Due to our assumption that $\ell' \neq \ell_n$, we also have that $m_n^\star(e) \neq m_n(e)$. Also since both declassifications are enforced it should be that the input context level ct and $ct^\star$ must be equal to L and as the result of that no inputs have taken place in high context, that is $hist^\star = hist$. Taking all of the above facts into account we can derive that

$$\begin{aligned} m_n(e) = memupd(m, \hat{L}, \hat{H}, hist_n)(e) = memupd(m, \hat{L}, \hat{S}, hist_n)(e) \\ = memupd(m, \hat{L}, \hat{S}, hist_n^\star)(e) = m_n^\star(e) \neq m_n(e)\end{aligned}$$

which brings us to contradiction.

□

Appendix G. Proofs for Extended Language: Termination-Sensitive Enforcement

For termination-sensitive enforcement we show a set of lemmas similar to the one shown in the previous section.

G.1. Monitor consistency

Lemma 7 (Monitor configuration through high context)**.** *Given a trace*

$$(\langle c, m, \hat{L}, \hat{H}, E, hist\rangle, \langle st, U, ct\rangle) \longrightarrow^* (\langle c', m', \hat{L}, \hat{H}', E, hist'\rangle, \langle st', U', ct\rangle)$$

where

1) *the context stack of the first configuration has a form $st = lev_n \dots lev_{k+1} : H : lev_{k-1} \dots lev_1$ with lev_n on the top of the stack and lev_1 on the bottom of it.*
2) *the execution of this trace does not consume stack entries below and including $lev_k = H$*

then the starting and ending sets of updated memories are equal: $U = U'$.

Proof: By induction on c. □

G.2. Sequential composition

Lemma 8. *Given σ_1 and σ_2 that are*

1) *two consistent termination-sensitive monitor configurations such that*
2) *both σ_1 and σ_2 are faithful with respect to the* same *source program configuration and two final configurations $\langle c_a; c_b, m_1, \hat{L}_1, \hat{H}_1, E, hist_1\rangle$ and $\langle c_a; c_b, m_2, \hat{L}_2, \hat{H}_2, E, hist_2\rangle$ respectively, where*
3) $m_{1L} = m_{2L}, hist_{1L} = hist_{2L}$
4) *and*

$$(\langle c_a; c_b, m_1, \hat{L}, \hat{H}_1, E, hist_1\rangle, \sigma_1) \longrightarrow^* (\langle c_b, m_1', \hat{L}, \hat{H}_1', E, hist_1'\rangle, \sigma_1') \longrightarrow^*_{\ell_1} (\langle c'', m_1'', \hat{L}_1'', \hat{H}_1'', E_1'', hist_1''\rangle, \sigma_1'')$$

then

1) $(\langle c_a; c_b, m_2, \hat{L}, \hat{H}_2, E, hist_2\rangle, \sigma_2) \longrightarrow^* (\langle c_b, m_2', \hat{L}, \hat{H}_2', E, hist_2'\rangle, \sigma_2') \longrightarrow^*_{\ell_2} (\langle d'', m_2'', \hat{L}_2'', \hat{H}_2'', E_2'', hist_2''\rangle, \sigma_2'')$
2) $m'_{1L} = m'_{2L}, hist'_{1L} = hist'_{2L}$
3) *σ_1' and σ_2' are consistent*

Proof: By induction on the structure of c_a using Lemma 7. □

Lemma 9. *Given σ_1 and σ_2 that are*

1) *two consistent termination-sensitive monitor configurations such that*
2) *both σ_1 and σ_2 are faithful with respect to the* same *source program configuration and two final configurations $\langle c_a; c_b, m_1, \hat{L}_1, \hat{H}_1, E, hist_1\rangle$ and $\langle c_a; c_b, m_2, \hat{L}_2, \hat{H}_2, E, hist_2\rangle$ respectively, where*
3) $m_{1L} = m_{2L}, hist_{1L} = hist_{2L}$
4) *and*

$$(\langle c_a; c_b, m_1, \hat{L}, \hat{H}_1, E, hist_1\rangle, \sigma_1) \longrightarrow^* (\langle c_a'; c_b, m_1', \hat{L}, \hat{H}_1', E, hist_1'\rangle, \sigma_1') \longrightarrow^*_{\ell_1} (\langle c'', m_1'', \hat{L}_1'', \hat{H}_1'', E_1'', hist_1''\rangle, \sigma_1'')$$

then

1) $(\langle c_a; c_b, m_2, \hat{L}, \hat{H}_2, E, hist_2\rangle, \sigma_2) \longrightarrow^* (\langle c_a'; c_b, m_2', \hat{L}, \hat{H}_2', E, hist_2'\rangle, \sigma_2') \longrightarrow^*_{\ell_2} (\langle d'', m_2'', \hat{L}_2'', \hat{H}_2'', E_2'', hist_2''\rangle, \sigma_2'')$
2) $m'_{1L} = m'_{2L}, hist'_{1L} = hist'_{2L}$
3) *σ_1' and σ_2' are consistent*
4) $c'' = d''$, $\hat{L}_1'' = \hat{L}_2''$, $E_1'' = E_2''$, $hist''_{1L} = hist''_{2L}$
5) *σ_1'' and σ_2'' are consistent*
6) *if ℓ_1 is not a declassification event then*
   - *$m''_{1L} = m''_{2L}$ and $\ell_1 = \ell_2$*
7) *if ℓ_1 is a declassification event then*
   - *ℓ_2 is a declassification event as well.*

Proof: By induction on the structure of c_a using Lemma 7. □

G.3. Advancement Lemma

Lemma 10. *Given σ_1 and σ_2 that are*

1) *two consistent termination-sensitive monitor configurations such that*
2) *both σ_1 and σ_2 are faithful with respect to the* same *source program configuration and two final configurations $\langle c, m_1, \hat{L}_1, \hat{H}_1, E, hist_1\rangle$ and $\langle c, m_2, \hat{L}_2, \hat{H}_2, E, hist_2\rangle$ respectively, where*
3) $m_{1L} = m_{2L}, hist_{1L} = hist_{2L}$
4) *and*

$$(\langle c, m_1, \hat{L}, \hat{H}_1, E, hist_1\rangle, \sigma_1) \longrightarrow^{*} (\langle c', m_1', \hat{L}, \hat{H}_1', E, hist_1'\rangle, \sigma_1') \longrightarrow^{*}_{\ell_1} (\langle c'', m_1'', \hat{L}_1'', \hat{H}_1'', E_1'', hist_1''\rangle, \sigma_1'')$$

then

1) $(\langle c, m_2, \hat{L}, \hat{H}_2, E, hist_2\rangle, \sigma_2) \longrightarrow^{*} (\langle c', m_2', \hat{L}, \hat{H}_2', E, hist_2'\rangle, \sigma_2') \longrightarrow^{*}_{\ell_2} (\langle d'', m_2'', \hat{L}_2'', \hat{H}_2'', E_2'', hist_2''\rangle, \sigma_2'')$
2) $m'_{1L} = m'_{2L}, hist'_{1L} = hist'_{2L}$
3) *σ_1' and σ_2' are consistent*
4) $c'' = d''$, $\hat{L}_1'' = \hat{L}_2''$, $E_1'' = E_2''$, $hist''_{1L} = hist''_{2L}$
5) *σ_1'' and σ_2'' are consistent*
6) *if ℓ_1 is not a declassification event then*
   - *$m''_{1L} = m''_{2L}$ and $\ell_1 = \ell_2$*
7) *if ℓ_1 is a declassification event then*
   - *ℓ_2 is a declassification event as well.*

Proof: By induction on c using sequential composition Lemmas 8 and 9. □

G.4. Backbone Lemma

Lemma 11 (Backbone Lemma)**.** *Given a program c and an enforced trace started from configuration $(\langle c, i, \hat{L}, \hat{H}, \emptyset, \epsilon\rangle, \langle \epsilon, \emptyset, L\rangle)$ which produces a sequence of low events $\vec{\ell_n}$:*

$$\begin{aligned}(\langle c, i, \hat{L}, \hat{H}, \emptyset, \epsilon\rangle, \langle \epsilon, \emptyset, L\rangle) \longrightarrow^{*} (\langle c_1, i_1, \hat{L}, \hat{H}_1, \emptyset, hist_1\rangle, \sigma_1) \longrightarrow_{\ell_1} (\langle c_1', i_1', \hat{L}_1, \hat{H}_1, E_1, hist_1'\rangle, \sigma_1') \longrightarrow^{*}_{\ell_2 \ldots \ell_{j-1}} \\ (\langle c_j, i_j, \hat{L}_{j-1}, \hat{H}_j, E_{j-1}, hist_j\rangle, \sigma_j) \longrightarrow_{\ell_j} (\langle c_j', i_j', \hat{L}_j, \hat{H}_j', E_j, hist_j'\rangle, \sigma_j') \longrightarrow^{*}_{\ell_{j+1} \ldots \ell_{n-1}} \\ (\langle c_n, i_n, \hat{L}_{n-1}, \hat{H}_n, E_{n-1}, hist_n\rangle, \sigma_n) \longrightarrow_{\ell_n} (\langle c_n', i_n', \hat{L}_n, \hat{H}_n', E_n, hist_n'\rangle, \sigma_n')\end{aligned}$$

and an initial high environment $(m, \hat{S}) \in k^{\sigma}(c, i_L, \hat{L}, \vec{\ell_n})$, there exist $m_1, m_1', \hat{S}_1, \hat{S}_1', s_1, s_1', \kappa_1, \kappa_1', \ldots m_n, m_n', \hat{S}_n, \hat{S}_n', s_n, s_n', \kappa_n', \kappa_n'$ such that $\forall\ j\ .\ 1 \leq j \leq n$ we have

1) *$m_{jL} = i_{jL}, m'_{jL} = i'_{jL}$, and*
2) *$hist_j, s_j$ and $hist_j', s_j'$ are pairwise consistent input histories,*
3) *σ_j, κ_j and σ_j', κ_j' are pairwise consistent monitor configurations,*
4) *for the trace starting with m and $\hat{S}$ we have*

$$\begin{aligned}(\langle c, m, \hat{L}, \hat{S}, \emptyset, \epsilon\rangle, \langle m, \epsilon, L\rangle) \longrightarrow^{*} (\langle c_1, m_1, \hat{L}, \hat{S}_1, \emptyset, s_1\rangle, \kappa_1) \longrightarrow_{\ell_1} (\langle c_1', m_1', \hat{L}_1, \hat{S}_1', E_1, s_1'\rangle, \kappa_1) \longrightarrow^{*}_{\ell_2 \ldots \ell_{j-1}} \\ (\langle c_j, m_j, \hat{L}_{j-1}, \hat{S}_j, E_{j-1}, s_j\rangle, \kappa_j) \longrightarrow_{\ell_j} (\langle c_j', m_j', \hat{L}_j, \hat{S}_j', E_j, s_j'\rangle, \kappa_j') \longrightarrow^{*}_{\ell_{j+1} \ldots \ell_{n-1}} \\ (\langle c_n, m_n, \hat{L}_{n-1}, \hat{S}_n, E_{n-1}, s_n\rangle, \kappa_n) \longrightarrow_{\ell_n} (\langle c_n', m_n', \hat{L}_n, \hat{S}_n', E_n, s_n'\rangle, \kappa_n')\end{aligned}$$

Proof: By induction on n using Advancement Lemma 10. □

G.5. Termination-sensitive security

Restatement of Proposition 4. Given a program c, initial memory i, communication environments $\hat{L}, \hat{H}$, and a sequence $\vec{\ell}$ of low events produced by $\langle c, i, \emptyset, \hat{L}, \hat{H}, \epsilon\rangle$ while monitored by monitor $\langle \epsilon, \emptyset, L\rangle$, we have that c satisfies termination-sensitive security with respect to $\vec{\ell}$, i_L, and $\hat{L}$, that is, $\widehat{TSec}(c, i_L, \hat{L}, \vec{\ell})$.

Proof: By induction on the length of the sequence of low events $\vec{\ell}$.

- Base case. $\vec{\ell} = \epsilon$. In this case we have n = 0, and, therefore, the main expression of the security definition trivially holds.
- Induction step. We assume the proposition holds for events of length up to $n-1$ and need to prove that if

  $$\langle c, i, \hat{L}, \hat{H}, \emptyset, hist\rangle \longrightarrow^{*}_{\vec{\ell}_n} \langle c'_n, i'_n, \hat{L}_n, \hat{H}_n, E_n, hist_n\rangle$$

  is a trace enforced by σ then $\forall m \in k^{\sigma}(c, i_L, \hat{L}, \vec{\ell}_n)$.

  $$p(m, \hat{L}, \hat{H}, E_n, hist_n) \subseteq k^{\sigma}(c, m_L, \hat{L}, \vec{\ell}_n).$$

  We consider two cases for the low event ℓ_n

  1) ℓ_n is not a declassification event.
     In this case we apply Backbone Lemma 11 followed by Advancement Lemma 10 which gives us

     $$k^{\sigma}(c, m_L, \hat{L}, \vec{\ell}_{n-1}) \subseteq k^{\sigma}(c, m_L, \vec{\ell}_n)$$

     We can intersect the left hand side of this expression with $p(m, \hat{L}, \hat{H}, E_n, hist_n)$ obtaining the desired inequality. Since no declassifications take place between ℓ_{n-1} and ℓ_n we have that $E_{n-1} = E_n$. Consequently, no escape hatches in E_n refer to the events in input history after those in $hist_{n-1}$ Together with the induction hypothesis this gives us

     $$p(m, \hat{L}, \hat{H}, E_n, hist_n) = p(m, \hat{L}, \hat{H}, E_{n-1}, hist_{n-1}) \subseteq k^{\sigma}(c, m_L, \hat{L}, \vec{\ell}_{n-1}) \subseteq k^{\sigma}(c, m_L, \vec{\ell}_n)$$

  2) ℓ_n is a declassification event.
     We assume contrary, that is, there exists an initial high environment $(m, \hat{S}) \in k^{\sigma}(c, m_L, \hat{L}, \vec{\ell}_n)$ such that

     $$p(m, \hat{L}, \hat{H}, E_n, hist_n) \supset k^{\sigma}(c, m_L, \hat{L}, \vec{\ell}_n).$$

     In other words there are $(m^{\star}, \hat{S}^{\star})$ such that $(m^{\star}, \hat{S}^{\star}) \in p(m, \hat{L}, \hat{H}, E_n, hist_n)$ but $(m^{\star}, \hat{S}^{\star}) \notin k^{\sigma}(c, m_L, \hat{L}, \vec{\ell}_n)$.

     Next, $(m^{\star}, \hat{S}^{\star}) \in p(m, \hat{L}, \hat{H}, E_n, hist_n) \implies m^{\star} =_L m \wedge (m^{\star}, \hat{S}^{\star}) I(E_n, \hat{L}, hist_n)(m, \hat{H})$. That is, $memupd(m^{\star}, \hat{L}, \hat{S}^{\star}, hist_n) = memupd(m^{\star}, \hat{L}, \hat{H}^{\star}, hist_n)$. Since $(m, \hat{H}) \in k^{\sigma}(c, m_L, \hat{L}, \vec{\ell}_n)$ then, using Lemma 2, $m_n(e) = memupd(m, \hat{L}, \hat{H}, hist_n)$.

     We also have that $m^{\star} \in p(m, \hat{L}, \hat{H}, E_n, hist_n) \implies m^{\star} \in p(m, \hat{L}, \hat{H}, E_{n-1}, hist_{n-1})$ and by induction hypothesis this gives us that $m^{\star} \in k^{\sigma}(c, m_L, \hat{L}, \vec{\ell}_{n-1})$. This allows us to apply Backbone Lemma 11 followed by Advancement Lemma 10 that gives us that the trace starting with $(m^{\star}, \hat{S}^{\star})$ should also produce a declassification event $\ell^{\star}_n$. But since $(m^{\star}, \hat{S}^{\star})$ is not part of the knowledge set for $(m, \hat{H})$ after ℓ_n, we have also that

     $$m^{\star} \notin k^{\sigma}(c, m_L, \hat{L}, \vec{\ell}_n) \Longrightarrow m^{\star}_n(e) \neq m_n(e)$$

     On the other hand, because both ℓ_n and $\ell^{\star}_n$ are produced while being enforced by the termination-sensitive monitor, it should be that $vars(e) \cap U_n = \emptyset$, $vars(e) \cap U^{\star}_n = \emptyset$, and $ct_n = ct^{\star}_n = L$. Together these facts give us that the current value of variables appearing in expression e is the same as it was at their last input. Moreover, all inputs so far have happened in low context. That is, we can say that $m_n = memupd(m, \hat{L}, \hat{H}, hist_n)(e)$ and $m^{\star}_n(e) = memupd(m^{\star}, \hat{L}, \hat{S}^{\star}, hist^{\star}_n)(e)$, and moreover $hist = hist^{\star}$. Putting all these facts together we have that

     $$\begin{aligned} m_n(e) = memupd(m, \hat{L}, \hat{H}, hist_n)(e) &= memupd(m, \hat{L}, \hat{S}, hist_n)(e) \\ &= memupd(m, \hat{L}, \hat{S}, hist^{\star}_n)(e) = m^{\star}_n(e) \neq m_n(e) \end{aligned}$$

     which brings us to contradiction.

□