

# ATTACKER CONTROL AND IMPACT FOR CONFIDENTIALITY AND INTEGRITY

ASLAN ASKAROV AND ANDREW C. MYERS

Department of Computer Science, Cornell University  
*e-mail address:* aslan@cs.cornell.edu

Department of Computer Science, Cornell University  
*e-mail address:* andru@cs.cornell.edu

---

**ABSTRACT.** Language-based information flow methods offer a principled way to enforce strong security properties, but enforcing noninterference is too inflexible for realistic applications. Security-typed languages have therefore introduced declassification mechanisms for relaxing confidentiality policies, and endorsement mechanisms for relaxing integrity policies. However, a continuing challenge has been to define what security is guaranteed when such mechanisms are used. This paper presents a new semantic framework for expressing security policies for declassification and endorsement in a language-based setting. The key insight is that security can be characterized in terms of the influence that declassification and endorsement allow to the attacker. The new framework introduces two notions of security to describe the influence of the attacker. Attacker control defines what the attacker is able to learn from observable effects of this code; attacker impact captures the attacker's influence on trusted locations. This approach yields novel security conditions for checked endorsements and robust integrity. The framework is flexible enough to recover and to improve on the previously introduced notions of robustness and qualified robustness. Further, the new security conditions can be soundly enforced by a security type system. The applicability and enforcement of the new policies is illustrated through various examples, including data sanitization and authentication.

## 1. Introduction

Many common security vulnerabilities can be seen as violations of either confidentiality or integrity. As a general way to prevent these information security vulnerabilities, information flow control has become a popular subject of study, both at the language level [23] and at the operating-system level (e.g., [14, 12, 30]). The language-based approach holds the appeal that the security property of noninterference [13], can be provably enforced using a type system [27]. In practice, however, noninterference is too rigid: many programs considered secure need to violate noninterference in limited ways.

Using language-based downgrading mechanisms such as *declassification* [17, 21] and *endorsement* [20, 29], programs can be written in which information is intentionally released, and in which untrusted information is intentionally used to affect trusted information or decisions. Declassification relaxes confidentiality policies, and endorsement relaxes integrity policies. Both endorsement and declassification have been essential for building realistic applications, such as various applications built with Jif [15, 18]: games [5], a voting system [11], and web applications [9].

A continuing challenge is to understand what security is obtained when code uses downgrading. This paper contributes a more precise and satisfactory answer to this question, particularly

clarifying how the use of endorsement weakens confidentiality. While much work has been done on declassification (usefully summarized by Sands and Sabelfeld [24]), there is comparatively little work on the interaction between confidentiality and endorsement.

To see such an interaction, consider the following notional code example, in which a service holds both old data (`old_data`) and new data (`new_data`), but the new data is not to be released until time `embargo_time`. The variable `new_data` is considered confidential, and must be declassified to be released:

---

```

if request_time >= embargo_time
  then return declassify(new_data)
  else return old_data

```

---

Because the requester is not trusted, the requester must be treated as a possible attacker. Suppose the requester has control over the variable `request_time`, which we can model by considering that variable to be low-integrity. Because the intended security policy depends on `request_time`, the attacker controls the policy that is being enforced, and can obtain the confidential new data earlier than intended. This example shows that the integrity of `request_time` affects the confidentiality of `new_data`. Therefore, the program should be considered secure only when the guard expression, `request_time >= embargo_time`, is high-integrity.

A different but reasonable security policy is that the requester may specify the request time as long as the request time is in the past. This policy could be enforced in a language with endorsement by first checking the low-integrity request time to ensure it is in the past; then, if the check succeeds, endorsing it to be high-integrity and proceeding with the information release. The explicit endorsement is justifiable because the attacker’s actions are permitted to affect the release of confidential information as long as adversarial inputs have been properly sanitized. This is a common pattern in servers that process possibly adversarial inputs.

*Robust declassification* has been introduced in prior work [28, 16, 10] as a semantic condition for secure interactions between integrity and confidentiality. The prior work also develops type systems for enforcing robust declassification, which are implemented as part of Jif [18]. However, prior security conditions for robustness are not satisfactory, for two reasons. First, these prior conditions characterize information security only for terminating programs. A program that does not terminate is automatically considered to satisfy robust declassification, even if it releases information improperly during execution. Therefore the security of programs that do not terminate, such as servers, cannot be described. A second and perhaps even more serious limitation is that prior security conditions largely ignore the possibility of endorsement, with the exception of *qualified robustness* [16]. Qualified robustness gives the endorse operation a somewhat ad-hoc, nondeterministic semantics, to reflect the attacker’s ability to choose the endorsed value. This approach operationally models what the attacker can do, but does not directly describe the attacker’s control over confidentiality. The introduction of nondeterminism also makes the security property possibilistic. However, possibilistic security properties have been criticized because they can weaken under refinement [22, 25].

The main contribution of this paper is a general, language-based semantic framework for expressing information flow security and semantically capturing the ability of the attacker to influence both the confidentiality and integrity of information. The key building blocks for this semantics are *attacker knowledge* [1] and its (novel) dual, *attacker impact*, which respectively describe what attackers can know and what they can affect. Building upon attacker knowledge, the interaction of confidentiality and integrity, which we term *attacker control*, can be characterized formally. The robust interaction of confidentiality and integrity can then be captured cleanly as a constraint on attacker control. Further, endorsement is naturally represented in this framework as a form of attacker control, and a more satisfactory version of qualified robustness can be defined. All these security

conditions can be formalized in both *progress-sensitive* and *progress-insensitive* variants, allowing us to describe the security of both terminating and nonterminating systems.

We show that the progress-insensitive variants of these improved security conditions are enforced soundly by a simple security type system. Recent versions of Jif have added a *checked endorsement* construct that is useful for expressing complex security policies [9], but whose semantics were not precisely defined; this paper gives semantics, typing rules and a semantic security condition for checked endorsement, and shows that checked endorsement can be translated faithfully into simple endorsement at both the language and the semantic level. Our type system can easily be adjusted to enforce the progress-sensitive variants of the security conditions, as has been shown in the literature [26, 19].

The rest of this paper is structured as follows. Section 2 shows how to define information security in terms of attacker knowledge. Section 3 introduces attacker control. Section 4 defines progress-sensitive and progress-insensitive robustness using the new framework. Section 5 extends this to improved definitions of robustness that allow endorsements, generalizing qualified robustness. A type system for enforcing these robustness conditions is presented in Section 6. The checked endorsement construct appears in Section 7, which introduces a new notion of robustness that allows checked endorsements, and shows that it can be understood in terms of robustness extended with simple endorsements. Section 8 introduces attacker impact. Additional examples are presented in Section 9, related work is discussed in Section 10, and Section 11 concludes.

This paper is an extended version of a previous paper by the same authors [4]. The significant changes include proofs of all the main theorems, a semantic rather than syntactic definition of fair attacks, and a renaming of “attacker power” to “attacker impact”.

## 2. Semantics

*Information flow levels.* We assume two security levels for confidentiality—*public* and *secret*—and two security levels for integrity—*trusted* and *untrusted*. These levels are denoted respectively  $\mathbb{P}$ ,  $\mathbb{S}$  and  $\mathbb{T}$ ,  $\mathbb{U}$ . We define information flow ordering  $\sqsubseteq$  between these two levels:  $\mathbb{P} \sqsubseteq \mathbb{S}$ , and  $\mathbb{T} \sqsubseteq \mathbb{U}$ . The four levels define a security lattice, as shown on Figure 1. Every point on this lattice has two security components: one for confidentiality, and one for integrity. We extend the information flow ordering to elements on this lattice:  $\ell_1 \sqsubseteq \ell_2$  if the ordering holds between the corresponding components. As is standard, we define *join*  $\ell_1 \sqcup \ell_2$  as the least upper bound of  $\ell_1$  and  $\ell_2$ , and *meet*  $\ell_1 \sqcap \ell_2$  as the greatest lower bound of  $\ell_1$  and  $\ell_2$ . All four lattice elements are meaningful; for example, it is possible for information to be both secret and untrusted when it depends on both secret and untrusted (i.e., attacker-controlled) values. This lattice is the simplest possible choice for exploring the topics of this paper; however, the results of this paper straightforwardly generalize to the richer security lattices used in other work on robustness [10].

*Language and semantics.* We consider a simple imperative language with syntax presented in Figure 2. The semantics of the language is fairly standard and is given in Figures 3 and 4. For expressions, we define big-step evaluation of the form  $\langle e, m \rangle \Downarrow v$ , where  $v$  is the result of evaluating expression  $e$  in memory  $m$ . For commands, we define a small-step operational semantics, in which a single transition is written as  $\langle c, m \rangle \xrightarrow{t} \langle c', m' \rangle$ , where  $c$  and  $m$  are the initial command and memory, and  $c'$  and  $m'$  are the resulting command and memory. The only unusual feature is the annotation  $t$  on each transition, which we call an *event*. Events record assignments: an assignment to variable  $x$  of value  $v$  is recorded by an event  $(x, v)$ . This corresponds to our attacker model, in

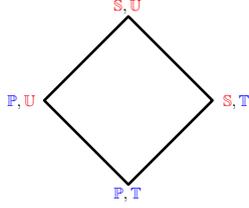


FIGURE 1. Information flow lattice

$$\begin{aligned}
 e &::= n \mid x \mid e \text{ op } e \\
 c &::= \text{skip} \mid x := e \mid c; c \\
 &\quad \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c
 \end{aligned}$$

FIGURE 2. Syntax of the language

$$\begin{array}{c}
 \langle n, m \rangle \downarrow n \qquad \langle x, m \rangle \downarrow m(x) \qquad \frac{\langle e_1, m \rangle \downarrow v_1 \quad \langle e_2, m \rangle \downarrow v_2 \quad v = v_1 \text{ op } v_2}{\langle e_1 \text{ op } e_2, m \rangle \downarrow v}
 \end{array}$$

FIGURE 3. Semantics of expressions

$$\begin{array}{c}
 \langle \text{skip}, m \rangle \longrightarrow \langle \text{stop}, m \rangle \qquad \frac{\langle e, m \rangle \downarrow v}{\langle x := e, m \rangle \longrightarrow_{(x,v)} \langle \text{stop}, m[x \mapsto v] \rangle} \\
 \\
 \frac{\langle c_1, m \rangle \longrightarrow_t \langle c'_1, m' \rangle}{\langle c_1; c_2, m \rangle \longrightarrow_t \langle c'_1; c_2, m' \rangle} \qquad \frac{\langle c_1, m \rangle \longrightarrow_t \langle \text{stop}, m' \rangle}{\langle c_1; c_2, m \rangle \longrightarrow_t \langle c_2, m' \rangle} \\
 \\
 \frac{\langle e, m \rangle \downarrow n \quad n \neq 0}{\langle \text{if } e \text{ then } c_1 \text{ else } c_2, m \rangle \longrightarrow \langle c_1, m \rangle} \qquad \frac{\langle e, m \rangle \downarrow n \quad n = 0}{\langle \text{if } e \text{ then } c_1 \text{ else } c_2, m \rangle \longrightarrow \langle c_2, m \rangle} \\
 \\
 \frac{\langle e, m \rangle \downarrow n \quad n \neq 0}{\langle \text{while } e \text{ do } c, m \rangle \longrightarrow \langle c; \text{while } e \text{ do } c, m \rangle} \qquad \frac{\langle e, m \rangle \downarrow n \quad n = 0}{\langle \text{while } e \text{ do } c, m \rangle \longrightarrow \langle \text{stop}, m \rangle}
 \end{array}$$

FIGURE 4. Semantics of commands

which the attacker may only observe assignments to public variables. We write  $\langle c, m \rangle \xrightarrow{*} \vec{t}$  to mean that trace  $\vec{t}$  is produced starting from  $\langle c, m \rangle$  using zero or more transitions. Each trace  $\vec{t}$  is composed of individual events  $t_1 \cdot t_2 \cdots t_k \cdots$ , and a *prefix* of  $\vec{t}$  up to the  $i$ -th event is denoted as  $\vec{t}_i$ ; we use the operator  $\cdot$  to denote the concatenation of two traces or events. If a transition does not affect memory, its event is *empty*, which is either written as  $\epsilon$  or is omitted, e.g.:  $\langle c, m \rangle \longrightarrow \langle c', m' \rangle$ .

Finally, we assume that the *security environment*  $\Gamma$  maps program variables to their security levels. Given a memory  $m$ , we write  $m_{\mathbb{P}}$  for the public part of the memory; similarly,  $m_{\mathbb{T}}$  is the trusted part of  $m$ . We write  $m =_{\mathbb{T}} m'$  when memories  $m$  and  $m'$  agree on their trusted parts, and  $m =_{\mathbb{P}} m'$  when  $m$  and  $m'$  agree on their public parts.

## 2.1. Attacker knowledge

This section provides background on the attacker-centric model for information flow security [1]. We recall definitions of attacker knowledge, progress knowledge, and divergence knowledge, and introduce progress-(in)sensitive *release events*.

*Low events.* Among the events that are generated during a trace, we distinguish a sequence of low (or public) events. Low events correspond to observations that an attacker can make during a run of the program. We assume that the attacker may observe individual assignments to public variables. Furthermore, if the program terminates, we assume that a termination event  $\Downarrow$  may also be observed by the attacker. If attacker can detect divergence of programs (cf. Definition 3) then divergence  $\Uparrow$  is also a low event.

Given a trace  $\vec{t}$ , low events in that trace are denoted as  $\vec{t}_{\mathbb{P}}$ . A single low event is often denoted as  $\ell$ , and a sequence of low events is denoted as  $\vec{\ell}$ . We overload the notation for semantic transitions, writing  $\langle c, m \rangle \xrightarrow{*}_{\vec{\ell}}$  if only low events produced from configuration  $\langle c, m \rangle$  are relevant; that is, there is a trace  $\vec{t}$  such that  $\langle c, m \rangle \xrightarrow{*}_{\vec{t}} \wedge \vec{t}_{\mathbb{P}} = \vec{\ell}$ . Low events are the key element in the definition of *attacker knowledge* [1].

The knowledge of the attacker is described by the set of initial memories compatible with low observations. Any reduction in this set means the attacker has learned something about secret parts of the initial memory.

**Definition 1** (Attacker knowledge). Given a sequence of low events  $\vec{\ell}$ , initial low memory  $m_{\mathbb{P}}$ , and program  $c$ , *attacker knowledge* is

$$k(c, m_{\mathbb{P}}, \vec{\ell}) \triangleq \{m' \mid m_{\mathbb{P}} = m'_{\mathbb{P}} \wedge \langle c, m' \rangle \xrightarrow{*}_{\vec{\ell}}\}$$

Attacker knowledge gives a handle on what information the attacker learns with every low event. The smaller the knowledge set, the more precise is the attacker's information about secrets. Knowledge is monotonic in the number of low events: as the program produces low events, the attacker may learn more about secrets.

Two extensions of attacker knowledge are useful: *progress knowledge* [3, 2] and *divergence knowledge* [3].

**Definition 2** (Progress knowledge). Given a sequence of low events  $\vec{\ell}$ , initial low memory  $m_{\mathbb{P}}$ , and a program  $c$ , define *progress knowledge*  $k_{\rightarrow}(c, m_{\mathbb{P}}, \vec{\ell})$  as

$$k_{\rightarrow}(c, m_{\mathbb{P}}, \vec{\ell}) \triangleq \{m' \mid m'_{\mathbb{P}} = m_{\mathbb{P}} \wedge \exists \ell' . \langle c, m' \rangle \xrightarrow{*}_{\vec{\ell}} \langle c', m'' \rangle \xrightarrow{*}_{\ell'}\}$$

Progress knowledge represents the information the attacker obtains by seeing public events  $\vec{\ell}$  followed by some other public event. Progress knowledge and attacker knowledge are related as follows: given a program  $c$ , memory  $m$  and a sequence of low events  $\ell_1 \cdots \ell_n$  obtained from  $\langle c, m \rangle$ , we have that for all  $i < n$ ,

$$k(c, m_{\mathbb{P}}, \vec{\ell}_i) \supseteq k_{\rightarrow}(c, m_{\mathbb{P}}, \vec{\ell}_i) \supseteq k(c, m_{\mathbb{P}}, \vec{\ell}_{i+1})$$

To illustrate this with an example, consider program  $l := 0; (\text{while } h = 0 \text{ do skip}); l := h$  with initial memory  $m(h) = 7$ . This program produces a sequence of two low events  $(l, 0) \cdot (l, 7)$ . The knowledge after the first event  $k(c, m_{\mathbb{P}}, (l, 0))$  is a set of all possible memories that agree with  $m$  on the public parts and can produce the low event  $(l, 0)$ . Note that no low events are possible after the first assignment unless  $h$  is non-zero. Progress knowledge reflects this:  $k_{\rightarrow}(c, m_{\mathbb{P}}, (l, 0))$  is a set of memories such that  $h \neq 0$ . Finally, the knowledge after two events  $k(c, m_{\mathbb{P}}, (l, 0) \cdot (l, 7))$  is a set of memories where  $h = 7$ .

Using attacker knowledge, one can express many confidentiality policies [7, 2, 8]. For example, a strong notion of *progress-sensitive noninterference* [13] can be expressed by demanding that knowledge between low events does not change:

$$k(c, m_{\mathbb{P}}, \vec{\ell}_i) = k(c, m_{\mathbb{P}}, \vec{\ell}_{i+1})$$

Progress knowledge enables expressing more permissive policies, such as *progress-insensitive noninterference*, which allows leakage of information, but only via termination channels (in [3] it is called *termination-insensitive*). This is expressed by requiring equivalence of the progress knowledge after seeing  $i$  events with the knowledge obtained after  $i + 1$ -th event:

$$k_{\rightarrow}(c, m_{\mathbb{P}}, \vec{\ell}_i) = k(c, m_{\mathbb{P}}, \vec{\ell}_{i+1})$$

In the example  $l := 0; (\text{while } h = 0 \text{ do skip}); l := 1$ , the knowledge inclusion between the two events is strict:  $k(c, m_{\mathbb{P}}, (l, 0)) \supset k(c, m_{\mathbb{P}}, (l, 0) \cdot (l, 1))$ . Therefore, the example does not satisfy progress-sensitive noninterference. On the other hand, the low event that follows the `while` loop does not reveal more information than the knowledge about the existence of that event. Formally,  $k_{\rightarrow}(c, m_{\mathbb{P}}, (l, 0)) = k(c, m_{\mathbb{P}}, (l, 0) \cdot (l, 1))$ , hence the program satisfies progress-insensitive noninterference.

These definitions also allow us to reason about knowledge changes along *parts of the traces*. We say that knowledge is preserved in a progress-(in)sensitive way along a part of a trace, assuming that the respective knowledge equality holds for the low events that correspond to that part.

Next, we extend possible observations to a divergence event  $\uparrow$  (we write  $\langle c, m \rangle \uparrow$  to mean configuration  $\langle c, m \rangle$  diverges). For attackers that can observe program divergence  $\uparrow$ , we define knowledge on the sequence of low events that includes divergence:

**Definition 3** (Divergence knowledge).

$$k(c, m_{\mathbb{P}}, \vec{\ell} \uparrow) \triangleq \{m' \mid m'_{\mathbb{P}} = m_{\mathbb{P}} \wedge \langle c, m' \rangle \xrightarrow{\vec{\ell}}^* \langle c'', m'' \rangle \wedge \langle c'', m'' \rangle \uparrow\}$$

Note that the above definition does not require divergence immediately after  $\vec{\ell}$  — it allows for more low events to be produced after  $\vec{\ell}$ . Divergence knowledge is used in Section 4.

Let us consider events at which knowledge preservation is broken. We call these events *release events*.

**Definition 4** (Release events). Given a program  $c$  and a memory  $m$ , such that

$$\langle c, m \rangle \xrightarrow{\vec{\ell}}^* \langle c', m' \rangle \xrightarrow{r}^*$$

- $r$  is a *progress-sensitive release event*, if  $k(c, m_{\mathbb{P}}, \vec{\ell}) \supset k(c, m_{\mathbb{P}}, \vec{\ell} \cdot r)$
- $r$  is a *progress-insensitive release event*, if  $k_{\rightarrow}(c, m_{\mathbb{P}}, \vec{\ell}) \supset k(c, m_{\mathbb{P}}, \vec{\ell} \cdot r)$

It is easy to validate that a progress-insensitive release event is also a progress-sensitive event. For example, in the program  $low := 1; low' := h$ , the second assignment is both a progress-sensitive and a progress-insensitive release event. The reverse is not true — in the program `while  $h = 0$  do skip;  $low := 1$`  the assignment to `low` is a progress-sensitive release event, but is not a progress-insensitive release event.

### 3. Attacks

To reason about program security in the presence of active attacks, we introduce a formal model of the attacker. Our formalization follows that in [16], where attacker-provided code can be injected into the program. This section provides examples of how attacker-injected code may affect attacker knowledge, followed by a semantic characterization of the attacker's influence on knowledge.

First, we extend the syntax to allow execution of attacker-controlled code:

$$c[\bullet] ::= \dots \mid [\bullet]$$

Next, we introduce notation  $\vec{t}$  to highlight that the trace  $\vec{t}$  is produced by attacker-injected code. The semantics of the language is extended accordingly.

$$\frac{\langle a, m \rangle \longrightarrow_t \langle a', m' \rangle}{\langle [a], m \rangle \longrightarrow_{[\vec{t}]} \langle [a'], m' \rangle} \quad \langle [stop], m \rangle \longrightarrow \langle stop, m \rangle$$

We limit attacks that can be substituted into holes to so-called *fair attacks*, which represent reasonable limitations on the impact of the attacker. Unlike earlier approaches, where fair attacks are defined syntactically [16, 10], we define them semantically. This allows us to include a larger set of attacks. To ensure that we include all syntactic attacks we make use of a reachability translation, explained below.

Roughly, we require a fair attack to not give new knowledge and to not modify trusted variables. A refinement of this idea is that an attack is fair if it gives new knowledge but only because the reachability of the attack depends on a secret. To capture this refinement, we define an auxiliary translation to make reachability of attacks explicit. We assume a trusted, public variable *reach* that does not appear in the source of  $c[\bullet]$ . Let operator  $T_{\rightsquigarrow}$  be a source-to-source transformation of  $c[\bullet]$  that makes reachability of attacks explicit.

**Definition 5** (Explicit reachability translation). Given a program  $c[\bullet]$ , define  $(T_{\rightsquigarrow}(c[\bullet]))$  as follows:

- $T_{\rightsquigarrow}([\bullet]) \implies \text{reach} := \text{reach} + 1; [\bullet]$
- $T_{\rightsquigarrow}(c_1; c_2) \implies T_{\rightsquigarrow}(c_1); T_{\rightsquigarrow}(c_2)$
- $T_{\rightsquigarrow}(\text{if } e \text{ then } c_1 \text{ else } c_2) \implies \text{if } e \text{ then } T_{\rightsquigarrow}(c_1) \text{ else } T_{\rightsquigarrow}(c_2)$
- $T_{\rightsquigarrow}(\text{while } e \text{ do } c) \implies \text{while } e \text{ do } T_{\rightsquigarrow}(c)$
- $T_{\rightsquigarrow}(c) \implies c$  for all other commands  $c$

The formal definition uses that any trace  $\vec{t}$  can be represented as a sequence of subtraces  $\vec{t}_1 \cdot [\vec{t}_2] \cdot \dots \cdot \vec{t}_{2*n-1} \cdot [\vec{t}_{2*n}]$ , where even-numbered subtraces correspond to the events produced by attacker-controlled code.

Given a trace  $\vec{t}$ , we denote the trusted events in the trace as  $\vec{t}_{\mathbb{T}}$ . We use notation  $t_*$  for a single trusted event, and  $\vec{t}_*$  for a sequence of trusted events.

**Definition 6** (Fair attack). Given a program  $c[\bullet]$ , such that  $T_{\rightsquigarrow}(c[\bullet]) \implies c_{\rightsquigarrow}[\bullet]$ , say that  $\vec{a}$  is a *fair attack* on  $c[\bullet]$  if for all memories  $m$ , such that  $\langle c_{\rightsquigarrow}[\vec{a}], m \rangle \xrightarrow{*} \vec{t}$  and  $\vec{t} = \vec{t}_1 \cdot [\vec{t}_2] \cdot \dots \cdot \vec{t}_{2*n-1} \cdot [\vec{t}_{2*n}]$ , i.e., there are  $2n$  intermediate configurations  $\langle c_j, m_j \rangle$ ,  $1 \leq j \leq 2n$ , for which

$$\langle c_{\rightsquigarrow}[\vec{a}], m \rangle \xrightarrow{*} \vec{t}_1 \langle c_1, m_1 \rangle \xrightarrow{*} [\vec{t}_2] \langle c_2, m_2 \rangle \xrightarrow{*} \vec{t}_3 \dots \xrightarrow{*} [\vec{t}_{2n}] \langle c_{2n}, m_{2n} \rangle \dots$$

then for all  $i$ ,  $1 \leq i \leq n$ , it holds that  $k(c_{\rightsquigarrow}[\vec{a}], m, \vec{t}_1 \dots \vec{t}_{2i-1}) = k(c_{\rightsquigarrow}[\vec{a}], m, \vec{t}_1 \dots [\vec{t}_{2i}])$  and  $\vec{t}_{2i*} = \epsilon$ .

For example, in the program `if  $h > 0$  then  $[\bullet]$  else skip` the attacks  $a_1 = [low := 1]$  and attack  $a_2 = [low := h > 0]$  are fair, but attack  $a_3 = [low := h]$  is not.

### 3.1. Examples of attacker influence

This section presents a few examples of attacker influence on knowledge. We also introduce pure availability attacks and progress attacks, to which we refer later in this section.

In the examples below, we use notation  $[(u, v)]$  when a low event  $(u, v)$  is generated by attacker-injected code.

Consider program  $[\bullet]; low := u > h;$  where  $h$  is a secret variable, and  $u$  is an untrusted public variable. The attacker's code executes before the low assignment and may change the value of  $u$ .

Consider memory  $m$ , where  $m(h) = 7$ , and the two attacks  $a_1 = u := 0$  and  $a_2 = u := 10$ . These attacks result in different values being assigned to variable  $low$ . The first trace results in low events  $[(u, 0)] \cdot (low, 0)$ , while the second trace results in low events  $[(u, 10)] \cdot (low, 1)$ . Therefore, the knowledge about the secret is different in each trace. We have

$$\begin{aligned} k(c[a_1], m_{\mathbb{P}}, [(u, 0)] \cdot (low, 0)) &= \{m' \mid m'(h) \geq 0\} \\ k(c[a_2], m_{\mathbb{P}}, [(u, 10)] \cdot (low, 1)) &= \{m' \mid m'(h) < 10\} \end{aligned}$$

Clearly, this program gives the attacker some control over what information about secrets he learns. Observe that it is not necessary for the last assignment to differ in order for the knowledge to be different. For example, consider attack  $a_3 = u := 5$ . This attack results in low events  $[(u, 5)] \cdot (low, 0)$ , which do the same assignment to  $low$  as  $a_1$  does. Attacker knowledge, however, is different from that obtained by  $a_1$ :

$$k(c[a_3], m_{\mathbb{P}}, [(u, 5)] \cdot (low, 0)) = \{m' \mid m'(h) \geq 5\}$$

Next, consider program  $[\bullet]; low := h$ . This program gives away knowledge about the value of  $h$  independently of untrusted variables. The only way for the attacker to influence what information he learns is to prevent that assignment from happening at all, which, as a result, will prevent him from learning that information. This can be done by an attack such as  $a = \text{while true do skip}$ , which makes the program diverge before the assignment is reached. We call attacks like this *pure availability attacks*. Another example of a pure availability attack is in the program  $[\bullet]; (\text{while } u = 0 \text{ do skip}); low := h$ . In this program, any attack that sets  $u$  to 0 prevents the assignment from happening.

Consider another example:  $[\bullet]; \text{while } u < h' \text{ do skip}; low := 1$ . As in the previous example, the value of  $u$  may change the reachability of  $low := 1$ . Assuming the attacker can observe divergence, this is not a pure availability attack, because diverging before the last assignment gives the attacker additional secret information, namely that  $u < h'$ . New information is also obtained if the attacker sees the low assignment. We name attacks like this *progress attacks*. In general, a progress attack is an attack that leads to program divergence in a way that observing that divergence (i.e., detecting there is no progress) gives new knowledge to the attacker.

### 3.2. Attacker control

We represent attacker control as a set of attacks that are similar in their influence on knowledge. Intuitively, if a program leaks no information to the attacker, the control corresponds to all possible attacks. In general, the more attacks are similar, the less influence the attacker has. Moreover, the control is a temporal property and depends on the trace that has been currently produced. The longer a trace is, the more influence an attack may have, and the smaller the control set is.

*Similar attacks.* The key element in the definition of control is specifying when two attacks are similar. Given a program  $c[\vec{\bullet}]$ , memory  $m$ , consider two attacks  $\vec{a}$  and  $\vec{b}$  that produce traces  $\vec{t}$  and  $\vec{q}$  respectively:

$$\langle c[\vec{a}], m \rangle \xrightarrow{*}_{\vec{t}} \quad \text{and} \quad \langle c[\vec{b}], m \rangle \xrightarrow{*}_{\vec{q}}$$

We compare  $\vec{a}$  and  $\vec{b}$  based on how they change attacker knowledge along their respective traces. First, if knowledge is preserved along a subtrace of one of the traces, say  $\vec{t}$ , it must be preserved along a subtrace of  $\vec{q}$  as well. Second, if at some point in  $\vec{t}$  there is a release event  $(x, v)$ , there must be a matching low event  $(x, v)$  in  $\vec{q}$ , and the attacks are similar along the rest of the traces.

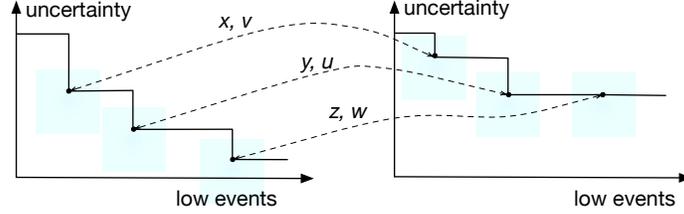


FIGURE 5. Similar attacks and traces

Visually, this requirement is described by the two diagrams in Figure 5. Each diagram shows the change of knowledge as more low events are produced. Here the  $x$ -axis corresponds to low events, and the  $y$ -axis reflects the attacker's uncertainty about initial secrets. Whenever one of the traces reaches a release event, depicted by vertical drops, there must be a corresponding low event in the other trace, such that the two events agree. This is depicted by the dashed lines between the two diagrams.

Formally, these requirements are stated using the following definitions.

**Definition 7** (Knowledge segmentation). Given a program  $c$ , memory  $m$ , and a trace  $\vec{t}$ , a sequence of indexes  $p_1 \dots p_N$  such that  $p_1 < p_2 < \dots < p_N$  and  $\vec{t}_{\mathbb{P}} = \ell_{1\dots p_1} \cdot \ell_{p_1+1\dots p_2} \dots \ell_{p_{N-1}+1\dots p_N}$  is called

- *progress-sensitive knowledge segmentation* of size  $N$ , if  $\forall j \leq N, \forall i . p_{j-1} + 1 \leq i < p_j . k(c, m_{\mathbb{P}}, \vec{\ell}_i) = k(c, m_{\mathbb{P}}, \vec{\ell}_{i+1})$ , denoted by  $\text{Seg}(c, m, \vec{t}, p_1 \dots p_N)$ .
- *progress-insensitive knowledge segmentation* of size  $N$  if  $\forall j \leq N, \forall i . p_{j-1} + 1 \leq i < p_j . k_{\rightarrow}(c, m_{\mathbb{P}}, \vec{\ell}_i) = k_{\rightarrow}(c, m_{\mathbb{P}}, \vec{\ell}_{i+1})$ , denoted by  $\text{Seg}_{\rightarrow}(c, m, \vec{t}, p_1 \dots p_N)$ .

Low events  $p_i + 1$  for  $1 \leq i < N$  are called *segmentation events*.

Note that given a trace, there can be more than one way to segment it, and for every trace consisting of  $n$  low events, this can be trivially achieved by a segmentation of size  $n$ . We use knowledge segmentation to define *attack similarity*:

**Definition 8** (Similar attacks and traces  $\sim^{c[\bullet], m}$ ). Given a program  $c[\bullet]$ , memory  $m$ , and two attacks  $\vec{a}$  and  $\vec{b}$  that produce traces  $\vec{t}$  and  $\vec{q}$ , define  $\vec{a}$  and  $\vec{b}$  as *similar* along  $\vec{t}$  and  $\vec{q}$  for the progress-sensitive attacker, if there are two segmentations  $p_1 \dots p_N$  and  $p'_1 \dots p'_N$  (for some  $N$ ) such that

- $\text{Seg}(c[\vec{a}], m, \vec{t}, p_1 \dots p_N)$ ,
- $\text{Seg}(c[\vec{b}], m, \vec{q}, p'_1 \dots p'_N)$ , and
- $\forall i . 1 \leq i < N . t_{\mathbb{P} p_i+1} = q_{\mathbb{P} p'_i+1}$ .

For the progress-insensitive attacker, the definition is similar except that it uses progress-insensitive segmentation  $\text{Seg}_{\rightarrow}$ . If two attack–trace pairs are similar, we write  $(\vec{a}, \vec{t}) \sim^{c[\bullet], m} (\vec{b}, \vec{q})$  (for progress-insensitive similarity,  $(\vec{a}, \vec{t}) \sim_{\rightarrow}^{c[\bullet], m} (\vec{b}, \vec{q})$ ).

The construction of Definitions 7 and 8 can be illustrated by program

```
[•]; if u then (while h ≤ 100 do skip) else skip; low1 := 0; low2 := h > 100
```

Consider memory with  $m(h) = 555$ , and two attacks  $a_1 = u := 1$ , and  $a_2 = u := 0$ . Both attacks reach the assignments to low variables. However, for  $a_2$  the assignment to  $low_2$  is a progress-insensitive release event, while for  $a_1$  the knowledge changes at an earlier assignment.

*Attacker control.* We define attacker control with respect to an attack  $\vec{a}$  and a trace  $\vec{t}$  as the set of attacks that are similar to the given attack in its influence on knowledge.

**Definition 9** (Attacker control (progress-sensitive)).

$$R(c[\bullet], m, \vec{a}, \vec{t}) \triangleq \{\vec{b} \mid \exists \vec{q}. (\vec{a}, \vec{t}) \sim^{c[\bullet], m} (\vec{b}, \vec{q})\}$$

To illustrate how attacker control changes, consider program  $[\bullet]$ ;  $low := u < h$ ;  $low' := h$  where  $u$  is an untrusted variable and  $h$  is a secret trusted variable. To understand attacker control of this program, we consider an initial memory  $m(h) = 7$  and attack  $a = u := 5$ . The low event  $(low, 1)$  in this trace is a release event. The attacker control is the set of all attacks that are similar to  $a$  and trace  $[(u := 5)]$ ,  $(low, 1)$  in its influence on knowledge. This corresponds to attacks that set  $u$  to values such that  $u < 7$ . The assignment to  $low'$  changes attacker knowledge as well, but the information that the attacker gets does not depend on the attack: any trace starting in  $m$  and reaching the second assignment produces the low event  $(low', 7)$ ; hence, the attacker control does not change at that event.

Consider the same example but with the two assignments swapped:  $[\bullet]$ ;  $low' := h$ ;  $low := u < h$ . The assignment to  $low'$  is a release event that the attacker cannot affect. Hence the control includes all attacks that reach this assignment. The result of the assignment to  $low$  depends on  $u$ . However, this result does not change attacker knowledge. Indeed, in this program, the second assignment is not a release event at all. Therefore, the attacker control is simply all attacks that reach the first assignment.

*Progress-insensitive control.* For progress-insensitive security, attacker control is defined similarly using the progress-insensitive comparison of attacks.

**Definition 10** (Attacker control (progress-insensitive)).

$$R_{\rightarrow}(c[\bullet], m, \vec{a}, \vec{t}) \triangleq \{\vec{b} \mid \exists \vec{q}. (\vec{a}, \vec{t}) \sim_{\rightarrow}^{c[\bullet], m} (\vec{b}, \vec{q})\}$$

Consider program  $[\bullet]$ ; **while**  $u < h$  **do** **skip**;  $low := 1$ . Here, any attack produces a trace that preserves progress-insensitive noninterference. If the loop is taken, the program produces no low events, hence, it gives no new knowledge to the attacker. If the loop is not taken, and the low assignment is reached, this assignment preserves attacker knowledge in a progress-insensitive way. Therefore, the attacker control is all attacks.

## 4. Robustness

*Release control.* This section defines *release control*  $R^{\triangleright}$ , which captures the attacker's influence on release events. Intuitively, release control expresses the extent to which an attacker can affect the *decision* to produce some release event.

**Definition 11** (Progress-sensitive release control).

$$\begin{aligned} R^{\triangleright}(c[\bullet], m, \vec{a}, \vec{t}) \triangleq & \{\vec{b} \mid \exists \vec{q}. (\vec{a}, \vec{t}) \sim^{c[\bullet], m} (\vec{b}, \vec{q}) \wedge \\ & (\exists r^{\vec{t}}. k(c[\vec{b}], m_{\mathbb{P}}, \vec{q}_{\mathbb{P}}) \supset k(c[\vec{b}], m_{\mathbb{P}}, \vec{q}_{\mathbb{P}} \cdot r^{\vec{t}}_{\mathbb{P}})) \\ & \vee k(c[\vec{b}], m_{\mathbb{P}}, \vec{q}_{\mathbb{P}}) \supset k(c[\vec{b}], m_{\mathbb{P}}, \vec{q}_{\mathbb{P}} \uparrow) \\ & \vee \langle c[\vec{b}], m \rangle \Downarrow\} \end{aligned}$$

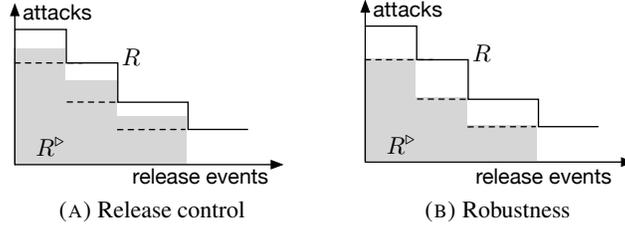


FIGURE 6. Release control and robustness

The definition for release control is based on the one for attacker control with the three additional clauses, explained below. These clauses restrict the set of attacks to those that either terminate or produce a release event. Because the progress-sensitive attacker can also learn new information by observing divergence, the definition contains an additional clause (on the third line) that uses divergence knowledge to reflect that.

Figure 6a depicts the relationship between release control and attacker control, where the  $x$ -axis corresponds to low events, and the  $y$ -axis corresponds to attacks. The solid line depicts attacker control  $R$ , where vertical lines correspond to release events. The gray area denotes release control  $R^\triangleright$ . In general, for a given attack  $\vec{a}$  and a corresponding trace  $\vec{t} \cdot \vec{r}$ , where  $\vec{r}$  contains a release event, we have the following relation between release control and attacker control:

$$R(c[\vec{\bullet}], m, \vec{a}, \vec{t}) \supseteq R^\triangleright(c[\vec{\bullet}], m, \vec{a}, \vec{t}) \supseteq R(c[\vec{\bullet}], m, \vec{a}, \vec{t} \cdot \vec{r}) \quad (4.1)$$

Note the white gaps and the gray release control above the dotted lines on Figure 6a. The white gaps correspond to difference  $R(c[\vec{\bullet}], m, \vec{a}, \vec{t}) \setminus R^\triangleright(c[\vec{\bullet}], m, \vec{a}, \vec{t})$ . This is a set of attacks that do not produce further release events and that diverge without giving any new information to the attacker—pure availability attacks. The gray zones above the dotted lines are more interesting. Every such zone corresponds to the difference  $R^\triangleright(c[\vec{\bullet}], m, \vec{a}, \vec{t}) \setminus R(c[\vec{\bullet}], m, \vec{a}, \vec{t} \cdot \vec{r})$ . In particular, when this set is non-empty, the attacker can launch attacks corresponding to each of the last three lines of Definition 11:

- (1) either trigger a different release event  $\vec{r}'$ , or
- (2) cause program to diverge in a way that also releases information, or
- (3) prevent a release event from happening in a way that leads to program termination

Absence of such attacks constitutes the basis for our security conditions in Definitions 13 and 14. Before moving on to these definitions, we introduce the progress-insensitive variant of release control.

**Definition 12** (Release control (progress-insensitive)).

$$R_{\rightarrow}^\triangleright(c[\vec{\bullet}], m, \vec{a}, \vec{t}) \triangleq \{ \vec{b} \mid \exists \vec{q}. (\vec{a}, \vec{t}) \sim_{c[\vec{\bullet}], m}^{c[\vec{\bullet}], m} (\vec{b}, \vec{q}) \wedge (\exists \vec{r}' . k_{\rightarrow}(c[\vec{b}], m_{\mathbb{P}}, \vec{q}_{\mathbb{P}}) \supset k(c[\vec{b}], m_{\mathbb{P}}, \vec{q}_{\mathbb{P}} \cdot \vec{r}'_{\mathbb{P}}) \vee \langle c[\vec{b}], m \rangle \Downarrow) \}$$

This definition uses the progress-insensitive variants of similar attacks and release events. It also does not account for knowledge obtained from divergence.

With the definition of release control at hand we can now define semantic conditions for robustness. The intuition is that all attacks leading to release events should lead to the same release event. Formally, this is defined as inclusion of release control into attacker control, where release control is computed on the prefix of the trace without a release event.

**Definition 13** (Progress-sensitive robustness). Program  $c[\vec{\bullet}]$  satisfies *progress-sensitive robustness* if for all memories  $m$ , attacks  $\vec{a}$ , and traces  $\vec{t}\vec{r}$ , such that  $\langle c[\vec{a}], m \rangle \xrightarrow{\vec{t}}^* \langle c', m' \rangle \xrightarrow{\vec{r}}$  and  $\vec{r}$  contains a release event, i.e.,  $k(c[\vec{a}], m_{\mathbb{P}}, \vec{t}_{\mathbb{P}}) \supset k(c[\vec{a}], m_{\mathbb{P}}, \vec{t}_{\mathbb{P}} \cdot \vec{r}_{\mathbb{P}})$ , we have

$$R^{\triangleright}(c[\vec{\bullet}], m, \vec{a}, \vec{t}) \subseteq R(c[\vec{\bullet}], m, \vec{a}, \vec{t} \cdot \vec{r})$$

Note that because of Equation 4.1, set inclusion in the above definition could be replaced with strict equality, but we use  $\subseteq$  for compatibility with future definitions. Figure 6b illustrates the relation between release control and attacker control for robust programs. Note how release control is bounded by the attacker control at the next release event.

*Examples.* We illustrate the definition of robustness with a few examples.

Consider program  $[\bullet]; \text{low} := u < h$ , and memory such that  $m(h) = 7$ . This program is rejected by Definition 13. To see this, pick an  $a = u := 5$ , and consider the part of the trace preceding the low assignment. Release control  $R^{\triangleright}(c[\vec{\bullet}], m, a, [(u, 5)])$  is all attacks that reach the assignment to *low*. On the other hand, the attacker control  $R(c[\vec{\bullet}], m, a, [(u, 5)] \cdot (\text{low}, 1))$  is the set of all attacks where  $u < 7$ , which is smaller than  $R^{\triangleright}$ . Therefore this program does not satisfy the condition.

Program  $[\bullet]; \text{low} := h; \text{low}' := u < h$  satisfies robustness. The only release event here corresponds to the first assignment. However, because the knowledge given by that assignment does not depend on untrusted variables, the release control includes all attacks that reach the assignment.

Program  $[\bullet]; \text{if } u > 0 \text{ then } \text{low} := h \text{ else skip}$  is rejected. Consider memory  $m(h) = 7$ , and attack  $a = u := 1$  that leads to low trace  $[(u, 1)] \cdot (\text{low}, 7)$ . The attacker control for this attack and trace is the set of all attacks such that  $u > 0$ . On the other hand, release control  $R^{\triangleright}(c[\vec{\bullet}], m, \vec{a}, [(u, 1)])$  is the set of all attacks that lead to termination, which includes attacks such that  $u \leq 0$ . Therefore, the release control corresponds to a bigger set than the attacker control.

Program  $[\bullet]; \text{while } u > 0 \text{ do skip}; \text{low} := h$  is accepted. Depending on the attacker-controlled variable the release event is reached. However, this is an example of availability attack, which is ignored by Definition 13.

Program  $[\bullet]; \text{while } u > h \text{ do skip}; \text{low} := 1$  is rejected. Any attack leading to the low assignment restricts the control to attacks such that  $u \leq h$ . However, release control includes attacks  $u > h$ , because the attacker learns information from divergence.

The definition of progress-insensitive robustness is similar to Definition 13, but uses progress-insensitive variants of release events, control, and release control. As a result, program  $[\bullet]; \text{while } u > h \text{ do skip}; \text{low} := 1$  is accepted: attacker control is all attacks.

**Definition 14** (Progress-insensitive robustness). Program  $c[\vec{\bullet}]$  satisfies *progress-insensitive robustness* if for all memories  $m$ , attacks  $\vec{a}$ , and traces  $\vec{t}\vec{r}$ , such that  $\langle c[\vec{a}], m \rangle \xrightarrow{\vec{t}}^* \langle c', m' \rangle \xrightarrow{\vec{r}}$  and  $\vec{r}$  contains a release event, i.e.,  $k_{\rightarrow}(c[\vec{a}], m_{\mathbb{P}}, \vec{t}_{\mathbb{P}}) \supset k(c[\vec{a}], m_{\mathbb{P}}, \vec{t}_{\mathbb{P}} \cdot \vec{r}_{\mathbb{P}})$ , we have

$$R_{\rightarrow}^{\triangleright}(c[\vec{\bullet}], m, \vec{a}, \vec{t}) \subseteq R_{\rightarrow}(c[\vec{\bullet}], m, \vec{a}, \vec{t} \cdot \vec{r})$$

## 5. Endorsement

This section extends the semantic policies for robustness in a way that allows *endorsing* attacker-provided values.

*Syntax and semantics.* We add endorsement to the language:

$$c[\bullet] ::= \dots \mid x := \text{endorse}_\eta(e)$$

We assume that every endorsement in the program source has a unique *endorsement label*  $\eta$ . Semantically, endorsements produce *endorsement events*, denoted by  $\text{endorse}(\eta, v)$ , which record the label of the endorsement statement  $\eta$  together with the value  $v$  that is endorsed.

$$\frac{\langle e, m \rangle \downarrow v}{\langle x := \text{endorse}_\eta(e), m \rangle \longrightarrow_{\text{endorse}(\eta, v)} \langle \text{stop}, m[x \mapsto v] \rangle}$$

Whenever the endorsement label is unimportant, we omit it from the examples. Note that  $\text{endorse}(\eta, v)$  events need not mention variable name  $x$  since that information is implied by the unique label  $\eta$ .

Consider example program  $[\bullet]; \text{low} := \text{endorse}_{\eta_1}(u < h)$ . This program does not satisfy Definition 13. The reasoning for this is exactly the same as for program  $[\bullet]; \text{low} := u < h$  from Section 4.

*Irrelevant attacks.* Endorsement of certain values gives attacker some control over the knowledge. The key technical element of this section is the notion of *irrelevant attacks*, which defines the set of attacks that are endorsed, and that are therefore *excluded* when comparing attacker control with release control. We define irrelevant attacks formally below, based on the trace that is produced by a program.

Given a program  $c[\bullet]$ , starting memory  $m$ , and a trace  $\vec{t}$ , irrelevant attacks, denoted here by  $\Phi(c[\bullet], m, \vec{t})$ , are the attacks that lead to the same sequence of endorsement events as in  $\vec{t}$ , until they necessarily disagree on one of the endorsements. Because the influence of these attacks is reflected at endorsement events, we exclude them from consideration when comparing with attacker control.

We start by defining *irrelevant traces*. Given a trace  $\vec{t}$ , irrelevant traces for  $\vec{t}$  are all traces  $\vec{t}'$  that agree with  $\vec{t}$  on some prefix of endorsement events until they necessarily disagree on some endorsement. We define this set as follows.

**Definition 15** (Irrelevant traces). Given a trace  $\vec{t}$ , where endorsements are marked as  $\text{endorse}(\eta_j, v_j)$ , define a set of irrelevant traces based on the number of endorsements in  $\vec{t}$  as  $\phi_i(\vec{t})$ :  $\phi_0(\vec{t}) = \emptyset$ , and

$$\phi_i(\vec{t}) = \{\vec{t}' \mid \vec{t}' = \vec{q} \cdot \text{endorse}(\eta_i, v'_i) \cdot \vec{q}'\} \text{ such that}$$

$\vec{q}$  is a prefix of  $\vec{t}'$  with  $i - 1$  events all of which agree with  $\text{endorse}$  events in  $\vec{t}$ , and

$$v_i \neq v'_i$$

Define  $\phi(\vec{t}) \triangleq \bigcup_i \phi_i(\vec{t})$  as a set of *irrelevant traces* w.r.t.  $\vec{t}$ .

With the definition of irrelevant traces at hand, we can define irrelevant attacks: irrelevant attacks are attacks that lead to irrelevant traces.

**Definition 16** (Irrelevant attacks). Given a program  $c[\bullet]$ , initial memory  $m$ , and a trace  $\vec{t}$ , such that  $\langle c[\bullet], m \rangle \xrightarrow{*} \vec{t}$ , define *irrelevant attacks*  $\Phi(c[\bullet], m, \vec{t})$  as

$$\Phi(c[\bullet], m, \vec{t}) \triangleq \{\vec{a} \mid \langle c[\vec{a}], m \rangle \xrightarrow{*} \vec{t} \wedge \vec{t}' \in \phi(\vec{t})\}$$

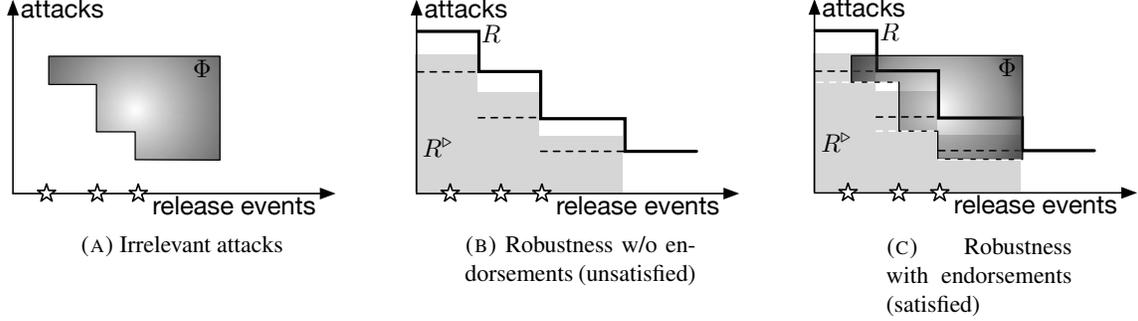


FIGURE 7. Irrelevant attacks and robustness with endorsements

*Security.* The security conditions for robustness can now be adjusted to accommodate endorsements that happen along traces. The idea is to exclude irrelevant attacks from the left-hand side of Definitions 13 and 14. This security condition, which has both progress-sensitive and progress-insensitive versions, expresses roughly the same idea as *qualified robustness* [16], but in a more natural and direct way.

**Definition 17** (Progress-sensitive robustness with endorsements). Program  $c[\bullet]$  satisfies *progress-sensitive robustness with endorsement* if for all memories  $m$ , attacks  $\vec{a}$ , and traces  $\vec{t}\vec{r}$ , such that  $\langle c[\vec{a}], m \rangle \xrightarrow{*_{\vec{t}}} \langle c', m' \rangle \xrightarrow{*_{\vec{r}}}$  and  $\vec{r}$  contains a release event, i.e.,  $k(c[\vec{a}], m_{\mathbb{P}}, \vec{t}_{\mathbb{P}}) \supset k(c[\vec{a}], m_{\mathbb{P}}, \vec{t}_{\mathbb{P}}\vec{r}_{\mathbb{P}})$ , we have

$$R^{\triangleright}(c[\bullet], m, \vec{a}, \vec{t}) \setminus \Phi(c[\bullet], m, \vec{t}\vec{r}) \subseteq R(c[\bullet], m, \vec{a}, \vec{t}\vec{r})$$

We refer to the set  $R^{\triangleright}(c[\bullet], m, \vec{a}, \vec{t}) \setminus \Phi(c[\bullet], m, \vec{t}\vec{r})$  as a set of *relevant attacks*. Figures 7a to 7c visualize irrelevant attacks and the semantic condition of Definition 17. Figure 7a shows the set of irrelevant attacks, depicted by the shaded gray area. This set increases at endorsement events marked by stars. Figure 7b shows an example trace where robustness is not satisfied — the gray area corresponding to release control  $R^{\triangleright}$  exceeds the attacker control (depicted by the solid line). Finally, in Figure 7c, we superimpose Figures 7a and 7b. This illustrates that when the set of irrelevant attacks is excluded from the release control (the area under white dashed lines), the program is accepted by robustness with endorsements.

*Examples.* Program  $[\bullet]; \text{low} := \text{endorse}_{\eta_1}(u < h)$  is accepted by Definition 17. Consider initial memory  $m(h) = 7$ , and an attack  $u := 1$ ; this produces a trace  $[(u, 1)]\text{endorse}(\eta_1, 1)$ . The endorsed assignment also produces a release event. We have that

- Release control  $R^{\triangleright}$  is the set of all attacks that reach the low assignment.
- Irrelevant traces  $\phi([(u, 1)]\text{endorse}(\eta_1, 1))$  is a set of traces that end in endorsement event  $\text{endorse}(\eta_1, v)$  such that  $v \neq 1$ . Thus, irrelevant attacks  $\Phi([\bullet]; \text{low} := \text{endorse}_{\eta_1}(u < h), m, [(u, 1)]\text{endorse}(\eta_1, 1))$  must consist of attacks that reach the low assignment and set  $u$  to values  $u \geq 7$ .
- The left-hand side of Definition 17 is therefore the set of attacks that reach the endorsement and set  $u$  to  $u < 7$ .
- As for the attacker control on the right-hand side, it consists of attacks that set  $u < 7$ . Hence, the set inclusion of Definition 17 holds and the program is accepted.

Program  $[\bullet]$ ;  $low := \text{endorse}_{\eta_1}(u)$ ;  $low' := u < h''$  is accepted. The endorsement in the first assignment implies that all relevant attacks must agree on the value of  $u$ , and, consequently, they agree on the value of  $u < h''$ , which gets assigned to  $low'$ . This also means that relevant attacks belong to the attacker control (which contains all attacks that agree on  $u < h''$ ).

Program  $[\bullet]$ ;  $low := \text{endorse}_{\eta_1}(u < h)$ ;  $low' := u < h''$  is rejected. Take initial memory such that  $m(h) \neq m(h')$ . The set of relevant attacks after the second assignment contains attacks that agree on  $u < h$  (due to the endorsement), but not necessarily on  $u < h''$ . The latter, however, is the requirement for the attacks that belong to the attacker control.

Program  $[\bullet]$ ;  $\text{if } u > 0 \text{ then } h' := \text{endorse}(u) \text{ else skip}$ ;  $low := h' < h$  is rejected. Assume initial memory where  $m(h) = m(h') = 7$ . Consider attack  $a_1$  that sets  $u := 1$  and consider the trace  $\vec{t}_1$  that it gives. This trace endorses  $u$  in the then branch, overwrites the value of  $h'$  with 1, and produces a release event  $(low, 1)$ . Consider another attack  $a_2$  that sets  $u := 0$ , and consider the corresponding trace  $\vec{t}_2$ . This trace contains release event  $(low, 0)$  without any endorsements. Now, attacker control  $R(c[\vec{\bullet}], m, a_2, \vec{t}_2)$  excludes  $a_1$ , because of the disagreement at the release event. At the same time,  $a_1$  is a relevant attack for  $a_2$ , because no endorsements happen along  $\vec{t}_2$ .

Consider program  $c[\vec{\bullet}]$ , which contains no endorsements. In this case, for all possible traces  $\vec{t}$ , we have that  $\phi(\vec{t}) = \phi_0(\vec{t}) = \emptyset$ . Therefore, by Definition 16 it must be that  $\Phi(c[\vec{\bullet}], m, \vec{t}) = \emptyset$  for all memories  $m$  and traces  $\vec{t}$ . This indicates that for programs without endorsements, progress-sensitive robustness with endorsements (Definition 17) conservatively reduces to the earlier definition of progress-sensitive robustness (Definition 13).

Progress-insensitive robustness with endorsement is defined similarly. The intuition for the definition remains the same, while we use progress-insensitive variants of progress control and control:

**Definition 18** (Progress-insensitive robustness with endorsement). Program  $c[\vec{\bullet}]$  satisfies *progress-insensitive robustness with endorsement* if for all memories  $m$ , attacks  $\vec{a}$ , and traces  $\vec{t}, \vec{r}$ , such that  $\langle c[\vec{a}], m \rangle \xrightarrow{\vec{t}} \langle c', m' \rangle \xrightarrow{\vec{r}}$ , and  $\vec{r}$  contains a release event, i.e.,  $k_{\rightarrow}(c[\vec{a}], m_{\mathbb{P}}, \vec{t}_{\mathbb{P}}) \supset k(c[\vec{a}], m_{\mathbb{P}}, \vec{t}_{\mathbb{P}} \cdot \vec{r}_{\mathbb{P}})$ , we have

$$R_{\rightarrow}^{\geq}(c[\vec{\bullet}], m, \vec{a}, \vec{t}) \setminus \Phi(c[\vec{\bullet}], m, \vec{t} \cdot \vec{r}) \subseteq R_{\rightarrow}(c[\vec{\bullet}], m, \vec{a}, \vec{t} \cdot \vec{r})$$

As a final note in this section, observe that because of the particular use of irrelevant attacks in Definitions 17 and 18 it is sufficient for us to define irrelevant traces so that they only match at the endorsement events. A slightly more generalized notion of irrelevance would require  $\vec{q}$  in Definition 15 to be similar to a prefix of  $\vec{t}'$ .

## 6. Enforcement

We now explore how to enforce robustness using a security type system. While this section focuses on progress-insensitive enforcement, it is possible to refine the type system to deal with progress sensitivity (modulo availability attacks) [26, 19]. Figures 8 and 9 display typing rules for expressions and commands. This type system is based on the one of [16] and is similar to many standard security type systems.

*Declassification.* We extend the language with a language construct for *declassification* of expressions  $\text{declassify}(e)$ . Whereas in earlier examples, we considered an assignment  $l := h$  to be secure if it did not violate robustness, we now require information flows from public to secret to be mediated by declassification. We note that declassification has no additional semantics and, in the

$$\begin{array}{c}
\Gamma \vdash n : \ell, \emptyset \quad \Gamma \vdash x : \Gamma(x), \emptyset \quad \frac{\Gamma \vdash e_1 : \ell_1, D_1 \quad \Gamma \vdash e_2 : \ell_2, D_2}{\Gamma \vdash e_1 \text{ op } e_2 : \ell_1 \sqcup \ell_2, D_1 \cup D_2} \\
\text{(T-DECL)} \\
\frac{\Gamma \vdash e : \ell, D}{\Gamma \vdash \text{declassify}(e) : \ell \sqcap (\mathbb{P}, \mathbb{U}), \text{vars}(e)}
\end{array}$$

FIGURE 8. Type system: expressions

$$\begin{array}{c}
\text{(T-SKIP)} \quad \Gamma, pc \vdash \text{skip} \quad \text{(T-SEQ)} \quad \frac{\Gamma, pc \vdash c_1 \quad \Gamma, pc \vdash c_2}{\Gamma, pc \vdash c_1; c_2} \\
\text{(T-ASGMT)} \quad \frac{\Gamma \vdash e : \ell, D \quad \ell \sqcup pc \sqsubseteq \Gamma(x) \quad \forall y \in D. \Gamma(y) \sqsubseteq (\mathbb{S}, \mathbb{T}) \quad D \neq \emptyset \implies pc \sqsubseteq (\mathbb{P}, \mathbb{T})}{\Gamma, pc \vdash x := e} \\
\text{(T-IF)} \quad \frac{\Gamma \vdash e : \ell, \emptyset \quad \Gamma, pc \sqcup \ell \vdash c_1 \quad \Gamma, pc \sqcup \ell \vdash c_2}{\Gamma, pc \vdash \text{if } e \text{ then } c_1 \text{ else } c_2} \quad \text{(T-WHILE)} \quad \frac{\Gamma \vdash e : \ell, \emptyset \quad \Gamma, pc \sqcup \ell \vdash c}{\Gamma, pc \vdash \text{while } e \text{ do } c} \\
\text{(T-HOLE)} \quad \frac{pc \sqsubseteq (\mathbb{P}, \mathbb{U})}{\Gamma, pc \vdash \bullet} \quad \text{(T-ENDORSE)} \quad \frac{pc \sqcup \Gamma(x) \sqsubseteq (\mathbb{S}, \mathbb{T}) \quad pc \sqsubseteq \Gamma(x) \quad \Gamma \vdash e : \ell, \emptyset \quad \ell \sqcap (\mathbb{S}, \mathbb{T}) \sqsubseteq \Gamma(x)}{\Gamma, pc \vdash x := \text{endorse}(e)}
\end{array}$$

FIGURE 9. Type system: commands

context of our simple language, can be inferred automatically. This may be achieved by placing declassifications in public assignments that appear in trusted code, i.e., in non- $\bullet$  parts of the program. Moreover, making declassification explicit has the following motivations:

- (1) On the enforcement level, the type system conveniently ensures that a non-progress release event may happen only at declassification. All other assignments preserve progress-insensitive knowledge.
- (2) Much of the related work on language-based declassification policies uses similar type systems. Showing our security policies can be enforced using such systems makes the results more general.

*Typing of expressions.* Type rules for expressions have form  $\Gamma \vdash e : \ell, D$  where  $\ell$  is the level of the expression, and  $D$  is a set of variables that may be declassified. The declassification is the most interesting rule among expressions. It downgrades the confidentiality level of the expression by returning  $\ell \sqcap (\mathbb{P}, \mathbb{U})$ , and counts all variables in  $e$  as declassified.

*Typing of commands.* The typing judgments for commands have the form  $\Gamma, pc \vdash c$ . The rules are standard for a security type system. We highlight typing of assignments, endorsement, and holes.

Assignments have two extra clauses for when the assigned expression contains a declassification ( $D \neq \emptyset$ ). The rule (T-ASGMT) requires all variables that can be declassified have high

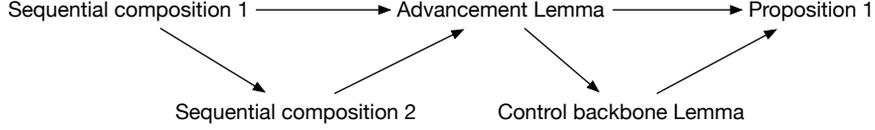


FIGURE 10. High-level structure of proof of Proposition 1

integrity. The rule also bounds the  $pc$ -label by  $(\mathbb{P}, \mathbb{T})$ , which enforces that no declassification happens in untrusted or secret contexts. These requirements guarantee that the information released by the declassification does not directly depend on the attacker-controlled variables.

The typing rule for endorsement (T-ENDORSE) requires that the  $pc$ -label is trusted and that the result of the endorsement is stored in a trusted variable:  $pc \sqcup \Gamma(x) \sqsubseteq (\mathbb{S}, \mathbb{T})$ . Note that requiring a trusted  $pc$ -label is crucial, while the restriction that  $x$  is trusted could easily be lifted, since trusted values may flow into untrusted variables. Because endorsed expressions preserve their confidentiality level, we also check that  $x$  has the right security level to store the result of the expression. This is done by demanding that  $\ell \sqcap (\mathbb{S}, \mathbb{T}) \sqsubseteq \Gamma(x)$ , where taking meet of  $\ell$  and  $(\mathbb{S}, \mathbb{T})$  boosts integrity, but keeps the confidentiality level of  $\ell$ .

The rule for holes forbids placing attacker-provided code in high confidentiality contexts. For simplicity, we disallow declassification in the guards of `if` and `while`.

### 6.1. Soundness

This section shows that the type system of Figures 8 and 9 is sound. We formulate top-level soundness in Proposition 1. The proof of Proposition 1 appears in the end of the section.

**Proposition 1.** If  $\Gamma, pc \vdash c[\bullet]$  then for all attacks  $\vec{a}$ , memories  $m$ , and traces  $\vec{t} \cdot \vec{r}$  produced by  $\langle c[\vec{a}], m \rangle$ , where  $k_{\rightarrow}(c[\vec{a}], m_{\mathbb{P}}, \vec{t}_{\mathbb{P}}) \supset k(c[\vec{a}], m_{\mathbb{P}}, \vec{t}_{\mathbb{P}} \cdot \vec{r}_{\mathbb{P}})$ , we have that

$$R_{\rightarrow}^{\supset}(c[\bullet], m, \vec{a}, \vec{t}) \setminus \Phi(c[\bullet], m, \vec{t} \cdot \vec{r}) \subseteq R_{\rightarrow}(c[\bullet], m, \vec{a}, \vec{t} \cdot \vec{r})$$

*Auxiliary definitions.* We introduce an auxiliary definition of progress-insensitive noninterference along a part of a trace, abbreviated PINI, which we will use in the proof of Proposition 1. Figure 10 shows the high-level structure of the proof. We define *declassification events* to be low events that involve declassifications. The central property of this proof — the control backbone lemma (Lemma 6) — captures the behavior of similar attacks and traces that are generated by well-typed commands. Together with the Advancement Lemma, it shows that declassification events soundly approximate release events. The proof of Proposition 1 follows directly from the Control Backbone and Advancement lemmas.

**Definition 19** (Progress-insensitive noninterference along a part of a trace). Given a program  $c$ , memory  $m$ , and two traces  $\vec{t}$  and  $\vec{t}^+$  such that  $\vec{t}^+$  is an extension of  $\vec{t}$ , we say that  $c$  satisfies *progress-insensitive noninterference* along the part of the trace from  $\vec{t}$  to  $\vec{t}^+$ , denoted by  $\text{PINI}(c, m, \vec{t}, \vec{t}^+)$  whenever for the low events in the corresponding traces  $\vec{\ell}_n \triangleq \vec{t}_{\mathbb{P}}^n$  and  $\vec{\ell}_N \triangleq \vec{t}_{\mathbb{P}}^N$ ,  $n \leq N$ , it holds that

$$\forall i. n < i < N. k_{\rightarrow}(c, m_{\mathbb{P}}, \vec{\ell}_i) \subseteq k(c, m_{\mathbb{P}}, \vec{\ell}_{i+1})$$

**Lemma 1** (Noninterference for no declassifications). Given a program  $c$  without declassifications such that  $\Gamma, pc \vdash c$  then for all memories  $m$  and possible low events  $\vec{\ell} \cdot \ell'$  such that

$$\langle c, m \rangle \xrightarrow{\vec{\ell}}^* \langle c', m' \rangle \xrightarrow{\ell'}^* \langle c'', m'' \rangle$$

it holds that  $k_{\rightarrow}(c, m, \vec{\ell}) \subseteq k(c, m, \vec{\ell} \cdot \ell')$ .

**Proof.** By induction on  $c$  (cf. [1]).  $\square$

**Lemma 2** (Noninterference for the tail of sequential composition). Assume a program  $c$  such that for all memories  $m$  and low events  $\ell$ , such that  $\langle c, m \rangle \xrightarrow{*}_{\ell} \langle c', m' \rangle$ , it holds that  $k_{\rightarrow}(c, m_{\mathbb{P}}, \epsilon) \subseteq k(c, m_{\mathbb{P}}, \ell)$ . Then for all programs  $c_0$ , initial memories  $i$ , and low events  $\vec{\ell}_0$ , such that

$$\langle c_0; c, i \rangle \xrightarrow{*}_{\vec{\ell}_0} \langle c, i' \rangle \xrightarrow{*}_{\ell'}$$

we have  $k_{\rightarrow}(c_0; c, i_{\mathbb{P}}, \vec{\ell}_0) \subseteq k(c_0; c, i_{\mathbb{P}}, \vec{\ell}_0 \cdot \ell')$ .

**Proof.** Assume the set inclusion from the statement of the lemma does not hold. By Definition 2, there must exist an initial memory  $m$ , such that  $m =_{\mathbb{P}} i$  and  $\langle c_0; c, m \rangle \xrightarrow{*}_{\vec{\ell}_0} \langle c, m' \rangle \xrightarrow{*}_{\ell'}$ , but  $\ell' \neq \ell''$ . Because  $m =_{\mathbb{P}} i$  and both traces produce  $\vec{\ell}_0$ , it must also be that  $m' =_{\mathbb{P}} i'$ . But this also implies that  $m' \notin k(c, i'_{\mathbb{P}}, \ell')$ , that is,  $k_{\rightarrow}(c, i'_{\mathbb{P}}, \epsilon) \not\subseteq k(c, i'_{\mathbb{P}}, \ell')$ , which contradicts the main assumption about  $c$ .  $\square$

The following two helper lemmas correspond to the sequential composition sub-cases of the Advancement Lemma. Lemma 3 captures the special case when the first command in the sequential composition  $c_1[\bullet]; c_2[\bullet]$  does not produce a declassification event, while Lemma 4 considers the general case when a declassification event may be produced by either of  $c_1[\bullet]$  or  $c_2[\bullet]$ .

**Lemma 3** (Sequential composition 1). Given

- program  $\vec{c}_0[\vec{\bullet}]$  such that  $\Gamma, pc \vdash c_0[\vec{\bullet}]$ ,
- initial memory  $m_0$ ,
- two initial attacks  $\vec{a}_0, \vec{b}_0$ ,
- two intermediate configurations  $\langle c_1[\vec{a}_1]; c_2[\vec{a}_2], m \rangle$  and  $\langle c_1[\vec{b}_1]; c_2[\vec{b}_2], s \rangle$  such that
- $\langle c_0[\vec{a}_0], m_0 \rangle \xrightarrow{*}_{\vec{t}} \langle c_1[\vec{a}_1]; c_2[\vec{a}_2], m \rangle \xrightarrow{*}_{\vec{t}_\alpha} \langle c_2[\vec{a}_2], m' \rangle \xrightarrow{*}_{\vec{t}_\beta r}$
- $\langle c_0[\vec{b}_0], m_0 \rangle \xrightarrow{*}_{\vec{q}'} \langle c_1[\vec{b}_1]; c_2[\vec{b}_2], s \rangle \xrightarrow{*}_{\vec{q}' r'}$
- $\text{PINI}(c_0[\vec{a}_0], m_0, \vec{t}', \vec{t}' \cdot \vec{t}_\alpha \cdot \vec{t}_\beta)$
- $\text{PINI}(c_0[\vec{b}_0], m_0, \vec{q}', \vec{q}' \cdot \vec{q}'')$
- $r$  and  $r'$  are declassification events
- $\vec{b}_0 \notin \Phi(c_0[\vec{\bullet}], m_0, \vec{t}' \cdot \vec{t}_\alpha \cdot \vec{t}_\beta \cdot r)$
- $\vec{t}'_\star = \vec{q}'_\star$
- $m =_{\mathbb{T}} s$

then  $\vec{q}'' = \vec{q}'_\alpha \cdot \vec{q}'_\beta$  such that

- $\langle c_1[\vec{a}_1]; c_2[\vec{a}_2], s \rangle \xrightarrow{*}_{\vec{q}'_\alpha} \langle c_2[\vec{a}_2], s' \rangle \xrightarrow{*}_{\vec{q}'_\beta r}$
- $\vec{t}'_{\alpha\star} = \vec{q}'_{\alpha\star}$
- $m' =_{\mathbb{T}} s'$

**Proof.** By induction on the structure of  $c_1[\vec{\bullet}]$ . Case `skip` is immediate. Consider the other cases.

- case `[•]`

In this case  $\vec{a}_1 = a_1$  and  $\vec{b}_1 = b_1$ . By assumption,  $a_1$  and  $b_1$  are fair attacks, which means that  $\vec{t}'_\alpha$  has no release events and no assignments to trusted variables. Similarly, because no low assignments can be produced when running  $\vec{b}_1$ , then by Definition 6 there must be  $s'$  and  $\vec{q}'_\alpha$  that would satisfy the demand of the lemma.

- case  $x := e$

We consider confidentiality and integrity properties separately.

**Confidentiality:** We show that even if a low event is possible, it is not a release event. We have two cases, based on the confidentiality level of  $x$ .

(1)  $\Gamma(x) = (\mathbb{P}, \_)$

A low event is generated by the low assignment. By Lemma 1 and Lemma 2 the assignment must not be a release event.

(2)  $\Gamma(x) = (\mathbb{S}, \_)$

In this case no low events are generated.

**Integrity:** Next, we show that the resulting memories agree on trusted values. The two cases are

(1)  $\Gamma(x) = (\_, \mathbb{T})$  In this case it must be that  $\Gamma(e) = (\_, \mathbb{T})$  and, hence,  $m(e) = s(e)$ . Therefore  $m' =_{\mathbb{T}} s'$ .

(2)  $\Gamma(x) = (\_, \mathbb{U})$  Assignment to  $x$  does not change how memories agree on trusted values.

- case  $x := \text{endorse}_{\eta}(e)$

We consider the confidentiality and integrity properties of this command separately.

**Confidentiality:** Similar to the case for assignment.

**Integrity:** We consider two cases.

(1)  $\Gamma(x) = (\_, \mathbb{T})$

In this case, the trace produces an event  $\text{endorse}(\eta, v)$ . We note that  $\vec{b}_0 \notin \Phi(c_0[\vec{\bullet}], m_0, \vec{t}' \cdot \vec{t}'_{\alpha} \cdot \vec{t}'_{\beta} \cdot r)$ . In particular, we have that  $\vec{q}' \cdot \vec{q}'' \cdot r' \notin \phi(\vec{t}' \cdot \vec{t}'_{\alpha} \cdot \vec{t}'_{\beta} \cdot r)$ . If we assume that the current command is the  $i$ -th endorsement in the trace, we have that  $\vec{q}' \cdot \vec{q}'' \cdot r' \notin \phi_i(\vec{t}' \cdot \vec{t}'_{\alpha} \cdot \vec{t}'_{\beta} \cdot r)$ . But we also know that  $\vec{t}'_{\star} =_{\mathbb{T}} \vec{q}'_{\star}$ . Because, by the rule (T-ENDORSE), the result of endorsement is assigned to trusted variables, this implies that both  $\vec{q}'$  and  $\vec{t}'$  must agree on the endorsed values. Therefore, the only possibility with which  $\vec{q}' \cdot \vec{q}'' \cdot r' \notin \phi_i(\vec{t}')$  is that  $\vec{q}''$  generates  $\text{endorse}(\eta, v)$  as well. This implies that value  $v$  is assigned to  $x$  in both cases, which guarantees that  $m' =_{\mathbb{T}} s'$ .

(2)  $\Gamma(x) = (\_, \mathbb{U})$  Not applicable by (T-ENDORSE).

- case  $c_{\alpha}; c_{\beta}$

By two applications of induction hypothesis: one to  $c_{\alpha}; (c_{\beta}; c_2[\vec{\bullet}])$  and the other one to  $c_{\beta}; c_2[\vec{\bullet}]$ .

- case **if**  $e$  **then**  $c_{\text{true}}$  **else**  $c_{\text{false}}$

We have the following cases based on the type of expression  $e$ .

(1)  $\Gamma(e) = (\_, \mathbb{T})$

In this case both branches are taking the same branch and we are done by induction hypothesis.

(2)  $\Gamma(e) = (\_, \mathbb{U})$

In this case neither of  $c_{\text{true}}$  or  $c_{\text{false}}$  contain declassifications or high integrity assignments. This guarantees that  $m' =_{\mathbb{T}} s'$ .

- case **while**  $e$  **do**  $c_{\text{loop}}$

Similar to sequential composition and conditionals.

□

**Lemma 4** (Sequential composition 2). Given

- program  $c_0[\vec{\bullet}]$  such that  $\Gamma, pc \vdash c_0[\vec{\bullet}]$
- initial memory  $m_0$
- two initial attacks  $\vec{a}_0, \vec{b}_0$

- two intermediate configurations  $\langle c_1[\vec{a}_1]; c_2[\vec{a}_2], m \rangle$  and  $\langle c_1[\vec{b}_1]; c_2[\vec{b}_2], s \rangle$  such that
- $\langle c_0[\vec{a}_0], m_0 \rangle \xrightarrow{\vec{t}} \langle c_1[\vec{a}_1]; c_2[\vec{a}_2], m \rangle \xrightarrow{\vec{t}'} \langle c_1[\vec{a}'_1]; c_2[\vec{a}_2], m' \rangle \xrightarrow{(x,v)} \langle c_1[\vec{a}''_1]; c_2[\vec{a}_2], m'' \rangle$
- $\langle c_0[\vec{b}_0], m_0 \rangle \xrightarrow{\vec{q}} \langle c_1[\vec{b}_1]; c_2[\vec{b}_2], s \rangle \xrightarrow{\vec{q}'} \langle d', s' \rangle \xrightarrow{(y,u)} \langle d'', s'' \rangle$
- $\text{PINI}(c_0[\vec{a}_0], m_0, \vec{t}', \vec{t}' \cdot \vec{t}'')$
- $\text{PINI}(c_0[\vec{b}_0], m_0, \vec{q}', \vec{q}' \cdot \vec{q}'')$
- $(x, v)$  and  $(y, u)$  are declassification events
- $\vec{b}_0 \notin \Phi(c_0[\vec{\bullet}], m_0, \vec{t}' \cdot \vec{t}'', (x, v))$
- $\vec{t}'_{\star} = \vec{q}'_{\star}$
- $m =_{\mathbb{T}} s$

then

- $\vec{a}'_1 = \vec{a}''_1$
- there is  $b'_1$  such that
- $d' = c_1[\vec{b}'_1]; c_2[\vec{b}_2]$
- $d'' = c_1[\vec{b}''_1]; c_2[\vec{b}_2]$
- $\vec{t}''_{\star} = \vec{q}''_{\star}$
- $m' =_{\mathbb{T}} s'$
- $m'' =_{\mathbb{T}} s''$
- $(x, v) = (y, u)$ .

**Proof.** By induction on the structure of  $c_1[\vec{\bullet}]$ . In the cases of  $[\vec{\bullet}]$ , `skip`, and  $x := \text{endorse}(e)$  no declassification events may be produced, so these cases are impossible.

- $x := e$ . When  $D = \emptyset$ , no declassification events may be produced. When  $D \neq \emptyset$ , a declassification event is produced by both traces. Also,  $\vec{t}'' = \vec{q}'' = \epsilon$ , and  $m' = m$  and  $s' = s$ . Because  $m =_{\mathbb{T}} s$  and  $\Gamma(e) = (\_, \mathbb{T})$  we have that both traces produces the same declassification event  $(x, v)$ , and therefore,  $m'' =_{\mathbb{T}} s''$ .
- case  $c_\alpha[a_\alpha]; c_\beta[a_\beta]$   
We have two cases depending on whether  $c_\alpha[a_\alpha]$  generates low events:
  - (1)  $\langle c_\alpha[a_\alpha]; (c_\beta[a_\beta]; c_2[a_2]), m \rangle \xrightarrow{\vec{\ell}_1 \dots \ell_N} \langle c_\beta[a_\beta]; c_2[a_2], m' \rangle$  In this case by Lemma 3 it must be that  $\langle c_\alpha[b_\alpha]; (c_\beta[b_\beta]; c_2[b_2]), m \rangle \xrightarrow{\vec{\ell}_1 \dots \ell'_N} \langle c_\beta[b_\beta]; c_2[b_2], s' \rangle$  such that  $m' =_{\mathbb{T}} s'$ . Then we can apply the induction hypothesis to  $c_\beta[\bullet]$ .
  - (2) In this case  $(x, v)$  is produced by  $c_\alpha[a_\alpha]$  and we are done by application of induction hypothesis to  $c_\alpha[\bullet]$ .
- case `if e then ctrue else cfalse`  
We have two cases:
  - (1)  $\Gamma(e) = (\_, \mathbb{T})$   
In this case both branches take the same command, and we are done by the induction hypothesis.
  - (2)  $\Gamma(e) = (\_, \mathbb{U})$ .  
Impossible, because declassification events are not allowed in untrusted integrity contexts.
- case `while e do cloop`

Similar to sequential composition and conditionals.

□

**Lemma 5** (Advancement). Given

- program  $c_0[\vec{\bullet}]$  such that  $\Gamma, pc \vdash c_0[\vec{\bullet}]$
- initial memory  $m_0$
- two initial attacks  $\vec{a}_0, \vec{b}_0$
- two intermediate configurations  $\langle c[\vec{a}], m \rangle$  and  $\langle c[\vec{b}], s \rangle$  such that
- $\langle c_0[\vec{a}_0], m_0 \rangle \xrightarrow{*} \vec{t}' \langle c[\vec{a}], m \rangle \xrightarrow{*} \vec{t}'' \langle c'[\vec{a}'], m' \rangle \xrightarrow{(x,v)} \langle c''[\vec{a}''], m'' \rangle$
- $\langle c_0[\vec{b}_0], m_0 \rangle \xrightarrow{*} \vec{q}' \langle c[\vec{b}], s \rangle \xrightarrow{*} \vec{q}'' \langle d', s' \rangle \xrightarrow{(y,u)} \langle d'', s'' \rangle$
- $\text{PINI}(c_0[\vec{a}_0], m_0, \vec{t}', \vec{t}' \cdot \vec{t}'')$
- $\text{PINI}(c_0[\vec{b}_0], m_0, \vec{q}', \vec{q}' \cdot \vec{q}'')$
- $(x, v)$  and  $(y, u)$  are declassification events
- $\vec{b} \notin \Phi(c_0[\vec{\bullet}], m_0, \vec{t}' \cdot \vec{t}'', (x, v))$
- $\vec{t}'_{\star} = \vec{q}'_{\star}$
- $m =_{\mathbb{T}} s$

then

- $\vec{a}' = \vec{a}''$
- there is  $b'$  such that
- $d' = c'[\vec{b}']$
- $d'' = c''[\vec{b}']$
- $\vec{t}''_{\star} = \vec{q}''_{\star}$
- $m' =_{\mathbb{T}} s'$
- $m'' =_{\mathbb{T}} s''$
- $(x, v) = (y, u)$ .

**Proof.** By induction on  $c[\vec{\bullet}]$ . In the cases of  $[\bullet]$ , `skip`, and  $x := \text{endorse}(e)$ , no declassification events may be produced, so these cases are impossible.

- $x := e$ . In case  $D = \emptyset$ , no declassification events may be produced. When  $D \neq \emptyset$ , a declassification event is produced by both traces. Also,  $\vec{t}'' = \vec{q}'' = \epsilon$ , and  $m' = m$  and  $s' = s$ . Because  $m =_{\mathbb{T}} s$  and  $\Gamma(e) = (\_, \mathbb{T})$  we have that both traces produces the same declassification event  $(x, v)$ , and therefore,  $m'' =_{\mathbb{T}} s''$ .
- case  $c_{\alpha}; c_{\beta}$   
By Lemma 4.
- case `if e then ctrue else cfalse`  
We have two cases:
  - (1)  $\Gamma(e) = (\_, \mathbb{T})$   
In this case both branches take the same command and we are done by induction hypothesis.
  - (2)  $\Gamma(e) = (\_, \mathbb{U})$ .  
Impossible, because declassification events are not allowed in untrusted integrity contexts.
- case `while e do cloop`  
Similar to sequential composition and conditionals.

□

**Lemma 6** (Control Backbone). Given  $\Gamma, pc \vdash c[\bullet]$ , memory  $m$ , an initial attack  $\vec{a}$  and a trace  $\vec{t}$ , such that

$$\langle c[\vec{a}], m \rangle \xrightarrow{*}_{\vec{t}_1} \langle c_1[\vec{a}_1], m_1 \rangle \xrightarrow{r_1} \langle c'_1[\vec{a}'_1], m'_1 \rangle \xrightarrow{*}_{\vec{t}_2 \dots r_{i-1}} \langle c'_{i-1}[\vec{a}'_{i-1}], m'_{i-1} \rangle \xrightarrow{*}_{\vec{t}_i} \boxed{\langle c_i[\vec{a}_i], m_i \rangle \xrightarrow{r_i} \langle c'_i[\vec{a}'_i], m'_i \rangle} \xrightarrow{*} \dots$$

where  $r_i$  are declassification events, then for all  $\vec{b}, \vec{q}$  such that  $(\vec{a}, \vec{t}) \sim_{\vec{c}[\bullet], m}^{c[\bullet], m} (\vec{b}, \vec{q})$  and  $\vec{b} \notin \Phi(c[\bullet], m, \vec{t})$ , it holds that the respective configurations (highlighted in boxes here) match at the declassification events, that is

$$\langle c[\vec{b}], m \rangle \xrightarrow{*}_{\vec{q}_1} \langle c_1[\vec{b}_1], s_1 \rangle \xrightarrow{r_1} \langle c'_1[\vec{b}'_1], s'_1 \rangle \xrightarrow{*}_{\vec{q}_2 \dots r_{i-1}} \langle c'_{i-1}[\vec{b}'_{i-1}], s'_{i-1} \rangle \xrightarrow{*}_{\vec{q}_i} \boxed{\langle c_i[\vec{b}_i], s_i \rangle \xrightarrow{r_i} \langle c'_i[\vec{b}'_i], s'_i \rangle} \xrightarrow{*} \dots$$

where  $i$  ranges over the number of declassification events in  $\vec{t}$ , and moreover

- $m_i =_{\mathbb{T}} s_i$  and  $m'_i =_{\mathbb{T}} s'_i$
- $\vec{q}_{i*} = \vec{t}_{i*}$

**Proof.** By induction on the number of declassification events. The base case, where  $n = 0$ , is immediate. For the inductive case, assume the proposition holds for the first  $n$  declassification events in  $\vec{t}$ , and apply Lemma 5. □

We conclude this section with the proof of Proposition 1.

**Proof of Proposition 1** Consider  $\vec{b} \in R_{\rightarrow}^{\mathbb{P}}(c[\bullet], m, \vec{a}, \vec{t}) \setminus \Phi(c[\bullet], m, \vec{t} \cdot \vec{r})$ . We want to show that  $\vec{b} \in R_{\rightarrow}(c[\bullet], m, \vec{a}, \vec{t} \cdot \vec{r})$ . Because  $\vec{b} \in R_{\rightarrow}^{\mathbb{P}}(c[\bullet], m, \vec{a}, \vec{t})$ , we have that  $\vec{b} \in \{\vec{b} \mid \exists \vec{q}. (\vec{a}, \vec{t}) \sim_{\vec{c}[\bullet], m}^{c[\bullet], m} (\vec{b}, \vec{q}) \wedge (\exists r'. k_{\rightarrow}(c[\vec{b}], m_{\mathbb{P}}, \vec{q}_{\mathbb{P}}) \supset k(c[\vec{b}], m_{\mathbb{P}}, \vec{q}_{\mathbb{P}} \cdot r'_{\mathbb{P}}) \vee \langle c[\vec{b}], m \rangle \Downarrow)\}$ . We consider the two cases

$$(1) \vec{b} \in \{\vec{b} \mid \exists \vec{q}. (\vec{a}, \vec{t}) \sim_{\vec{c}[\bullet], m}^{c[\bullet], m} (\vec{b}, \vec{q}) \wedge \exists r'. k_{\rightarrow}(c[\vec{b}], m_{\mathbb{P}}, \vec{q}_{\mathbb{P}}) \supset k(c[\vec{b}], m_{\mathbb{P}}, \vec{q}_{\mathbb{P}} \cdot r'_{\mathbb{P}})\}$$

By definition of  $\Phi(c[\bullet], m, \vec{t} \cdot \vec{r})$ , we have that  $\vec{b} \notin \Phi(c[\bullet], m, \vec{t} \cdot \vec{r}) \implies \vec{b} \notin \Phi(c[\bullet], m, \vec{t})$ . By the Control Backbone Lemma 6, we have that two traces agree on the declassification points up to the length of  $\vec{t}$ , and in particular there are  $\vec{t}_0, \vec{t}_1, \vec{q}_0, \vec{q}_1$  such that  $\vec{t} = \vec{t}_0 \cdot \vec{t}_1 \cdot \vec{r}$  and  $\vec{q} = \vec{q}_0 \cdot \vec{q}_1$  and that there are no release events along  $\vec{t}_1$  and  $\vec{q}_1$ , for which it holds that

$$\langle c[\vec{a}], m \rangle \xrightarrow{*}_{\vec{t}_0} \langle c'[\vec{a}'], m' \rangle \xrightarrow{*}_{\vec{t}_1 \vec{r}}$$

and

$$\langle c[\vec{b}], m \rangle \xrightarrow{*}_{\vec{q}_0} \langle c'[\vec{b}'], s' \rangle \xrightarrow{*}_{\vec{q}_1 \vec{r}}$$

where  $\vec{t}_* = \vec{q}_*$  and  $m' =_{\mathbb{T}} s'$ . By Advancement Lemma 5, we obtain that both traces must agree on  $\vec{r}$  and  $\vec{r}'$ . This is sufficient to extend the original partitioning of  $(\vec{a}, \vec{t})$  and  $(\vec{b}, \vec{q})$  to  $(\vec{a}, \vec{t} \cdot \vec{r})$  and  $(\vec{b}, \vec{q}_0 \cdot \vec{q}_1 \cdot \vec{r})$  such that  $(\vec{a}, \vec{t} \cdot \vec{r}) \sim_{\vec{c}[\bullet], m}^{c[\bullet], m} (\vec{b}, \vec{q}_0 \cdot \vec{q}_1 \cdot \vec{r})$ .

$$(2) \vec{b} \in \{\vec{b} \mid \exists \vec{q}. (\vec{a}, \vec{t}) \sim_{\vec{c}[\bullet], m}^{c[\bullet], m} (\vec{b}, \vec{q}) \wedge \langle c[\vec{b}], m \rangle \Downarrow\}$$

This case is impossible. By the Control Backbone Lemma 6 there must be two respective configurations  $\langle c'[\vec{a}'], m' \rangle$  and  $\langle c'[\vec{b}'], s' \rangle$  where  $m' =_{\mathbb{T}} s'$ , such that  $\langle c'[\vec{a}'], m' \rangle$  leads to a release event, but  $\langle c'[\vec{b}'], s' \rangle$  terminates without release events. By analysis of  $c'$ , similar to the Advancement Lemma, we conclude that none of the cases is possible. □

## 7. Checked endorsement

Realistic applications endorse attacker-provided data based on certain conditions. For instance, an SQL string that depends on user-provided input is executed if it passes sanitization, a new password is accepted if the user can provide an old one, and a secret key is accepted if nonces match. Because this is a recurring pattern in security-critical applications, we argue for language support in the form of *checked endorsements*.

This section extends the language with checked endorsements and derives both security conditions and a typing rule for them. Moreover, we show checked endorsements can be decomposed into a sequence of direct endorsements, and prove that for well-typed programs, the semantic conditions for robustness are the same with checked endorsements and with unchecked endorsements.

*Syntax and semantics.* In the scope of this section, we assume checked endorsements are the only endorsement mechanism in the language. We introduce a syntax for checked endorsements:

$$c[\vec{\bullet}] ::= \dots \mid \text{endorse}_{\eta}(x) \text{ if } e \text{ then } c \text{ else } c$$

The semantics of this command is that a variable  $x$  is endorsed if the expression  $e$  evaluates to true. If the check succeeds, the `then` branch is taken, and  $x$  is assumed to have high integrity there. If the check fails, the `else` branch is taken. As with direct endorsements, we assume checked endorsements in program text have unique labels  $\eta$ . These labels may be omitted from the examples, but they are explicit in the semantics.

*Endorsement events.* Checked endorsement events  $checked(\eta, v, b)$  record the unique label of the endorsement command  $\eta$ , the value of variable that can potentially be endorsed  $v$ , and a result of the check  $b$ , which can be either 0 or 1.

$$\frac{m(e) \downarrow v \quad v \neq 0}{\langle \text{endorse}_{\eta}(x) \text{ if } e \text{ then } c_1 \text{ else } c_2, m \rangle \xrightarrow{checked(\eta, m(x), 1)} \langle c_1, m \rangle}$$

$$\frac{m(e) \downarrow v \quad v = 0}{\langle \text{endorse}_{\eta}(x) \text{ if } e \text{ then } c_1 \text{ else } c_2, m \rangle \xrightarrow{checked(\eta, m(x), 0)} \langle c_2, m \rangle}$$

*Irrelevant attacks.* For checked endorsement we define a suitable notion of irrelevant attacks. The reasoning behind this is the following.

- (1) Both  $\vec{t}$  and  $\vec{t}'$  reach the same endorsement statement:  $\eta_i = \eta'_i$ .
- (2) At least one of them results in the positive endorsement:  $b_i + b'_i \geq 1$ . This ensures that if both traces do not take the branch then none of the attacks are ignored.
- (3) The endorsed values are different:  $v_i \neq v'_i$ . Otherwise, there should be no further difference in what the attacker can influence along the trace.

The following definitions formalize the above construction.

**Definition 20** (Irrelevant traces). Given a trace  $\vec{t}$ , where endorsements are labeled as  $checked(\eta_j, v_j, b_j)$ , define a set of irrelevant traces based on the number of checked endorsements in  $\vec{t}$  as  $\psi_i(\vec{t})$ . Then

$\psi_0(\vec{t}) = \emptyset$ , and

$\psi_i(\vec{t}) = \{\vec{t}' \mid \vec{t}' = \vec{q} \cdot \text{checked}(\eta_i, v'_i, b'_i) \cdot \vec{q}'\}$  such that

- $\vec{q}$  is a prefix of  $\vec{t}'$  with  $i - 1$  *checked* events, all of which agree with *checked* events in  $\vec{t}$ ,
- $(b_i + b'_i \geq 1) \wedge (v_i \neq v'_i)$ , and
- $\vec{q}'$  contains no *checked* events

Define  $\psi(\vec{t}) \triangleq \bigcup_i \psi_i(\vec{t})$  as a set of *irrelevant traces* w.r.t.  $\vec{t}$ .

**Definition 21** (Irrelevant attacks).  $\Psi(c[\bullet], m, \vec{t}) \triangleq \{\vec{a} \mid \langle c[\vec{a}], m \rangle \xrightarrow{*} \vec{a} \wedge \vec{t}' \in \psi(\vec{t})\}$

Using this definition, we can define security conditions for checked robustness.

**Definition 22** (Progress-sensitive robustness with checked endorsement). Program  $c[\bullet]$  satisfies *progress-sensitive robustness with checked endorsement* if for all memories  $m$  and attacks  $\vec{a}$ , such that  $\langle c[\vec{a}], m \rangle \xrightarrow{*} \vec{a} \langle c', m' \rangle \xrightarrow{*} \vec{r}$ , and  $\vec{r}$  contains a release event, i.e.,  $k(c[\vec{a}], m_{\mathbb{P}}, \vec{t}_{\mathbb{P}}) \supset k(c[\vec{a}], m_{\mathbb{P}}, \vec{t}_{\mathbb{P}} \vec{r}_{\mathbb{P}})$ , we have

$$R^{\triangleright}(c[\bullet], m, \vec{a}, \vec{t}) \setminus \Psi(c[\bullet], m, \vec{t} \cdot \vec{r}) \subseteq R(c[\bullet], m, \vec{a}, \vec{t} \cdot \vec{r})$$

The progress-insensitive version is defined similarly, using progress-insensitive definition for release events and progress-insensitive versions of control and release control.

*Example.* In program  $[\bullet]; \text{endorse}_{\eta_1}(u) \text{ if } u = u' \text{ then } \text{low} := u < h \text{ else skip}$ , the attacker can modify  $u$  and  $u'$ . This program is insecure because the unendorsed, attacker-controlled variable  $u'$  influences the decision to declassify. To see that Definition 22 rejects this program, consider running it in memory with  $m(h) = 7$ , and two attacks:  $a_1$ , where attacker sets  $u := 5; u' := 0$ , and  $a_2$ , where attacker sets  $u := 5; u' := 5$ . Denote the corresponding traces up to endorsement by  $\vec{t}_1$  and  $\vec{t}_2$ . We have  $\vec{t}_1 = [(u, 5)(u', 0)]\text{checked}(\eta_1, 5, 0)$  and  $\vec{t}_2 = [(u, 5)(u', 5)]\text{checked}(\eta_1, 5, 1)$ . Because endorsement in the second trace succeeds, this trace also continues with a low event  $(\text{low}, 1)$ . Following Definition 20 we have that  $t_1 \notin \psi(\vec{t}_2(\text{low}, 1))$ , implying  $a_1 \notin \Psi(c[\bullet], m, \vec{t}_2(\text{low}, 1))$ . Therefore,  $a_1 \in R^{\triangleright}(c[\bullet], m, \vec{a}_2, \vec{t}_2) \setminus \Psi(c[\bullet], m, \vec{t}_2(\text{low}, 1))$ . On the other hand,  $a_1 \notin R(c[\bullet], m, \vec{a}_2, \vec{t}_2(\text{low}, 1))$  because  $a_1$  can produce no low events corresponding to  $(\text{low}, 1)$ .

*Endorsing multiple variables.* The syntax for checked endorsements can be extended to multiple variables with the following syntactic sugar, where  $\eta_i$  is an endorsement label corresponding to variable  $x_i$ :

$$\text{endorse}(x_1, \dots, x_n) \text{ if } e \text{ then } c_1 \text{ else } c_2 \implies \text{endorse}_{\eta_1}(x_1) \text{ if } e \text{ then} \\ \text{endorse}_{\eta_2}(x_2) \text{ if true then } \dots c_1 \text{ else skip } \dots \text{ else } c_2$$

Note that in this encoding the condition is checked as early as possible; an alternative encoding here would check the condition in the end. While such encoding would have an advantage of type checking immediately, we believe that checking the condition as early as possible avoids spurious (albeit harmless in this simple context) endorsements of all but the last variable, and is therefore more faithful semantically.

*Typing checked endorsements.* To enforce programs with checked endorsements, we extend the type system with the following general rule:

$$\begin{array}{c}
 \text{(T-CHECKED)} \\
 \frac{\Gamma' \triangleq \Gamma[x_i \mapsto \Gamma(x_i)] \sqcap (\mathbb{S}, \mathbb{T}) \quad \Gamma' \vdash e : \ell', D' \quad pc' \triangleq pc \sqcup \ell' \quad pc' \sqsubseteq (\mathbb{S}, \mathbb{T}) \quad \Gamma', pc' \vdash c_1 \quad \Gamma, pc' \vdash c_2}{\Gamma, pc \vdash \text{endorse}(x_1, \dots, x_n) \text{ if } e \text{ then } c_1 \text{ else } c_2}
 \end{array}$$

The expression  $e$  is type-checked in an environment  $\Gamma'$  in which endorsed variables  $x_1, \dots, x_n$  have trusted integrity; its label  $\ell'$  is joined to form auxiliary  $pc$ -label  $pc'$ . The level of  $pc'$  must be trusted, ensuring that endorsements happen in a trusted context, and that no declassification in  $e$  depends on untrusted variables other than the  $x_i$  (this effectively subsumes the need to check individual variables in  $D'$ ). Each of the branches is type-checked with the program label set to  $pc'$ ; however, for  $c_1$  we use the auxiliary typing environment  $\Gamma'$ , since the  $x_i$  are trusted there.

Program  $[\bullet]; \text{endorse}(u) \text{ if } u = u' \text{ then } low := \text{declassify}(u < h) \text{ else skip}$  is rejected by this type system. Because variable  $u'$  is not endorsed, the auxiliary  $pc$ -label has untrusted integrity.

### 7.1. Relation to direct endorsements

Finally, for well-typed programs we can safely translate checked endorsements to direct endorsements using a translation in which a checked endorsement of  $n$  variables is translated to  $n + 1$  direct endorsements. First, we unconditionally endorse the result of the check. The rest of the endorsements happen in the `then` branch, before translation of  $c_1$ . We save the results of the endorsements in temporary variables  $t_1 \dots t_n$  and replace all occurrences of  $x_1 \dots x_n$  within  $c_1$  with the temporary ones (we assume that each  $t_i$  has the same confidentiality level as the corresponding original  $x_i$ , and  $t_0$  has the confidentiality level of the expression  $e$ ). All other commands are translated to themselves.

**Definition 23** (Labeled translation from checked endorsements to direct endorsements). Given a program  $c[\vec{\bullet}]$  that only uses checked endorsements, we define its labeled translation to direct endorsements  $\llbracket c[\vec{\bullet}] \rrbracket$  inductively:

- $\llbracket \text{endorse}_{\eta}(x_1, \dots, x_n) \text{ if } e \text{ then } c_1 \text{ else } c_2 \rrbracket \Longrightarrow t_0 := \text{endorse}_{\eta_0}(e); \text{ if } t_0 \text{ then } t_1 := \text{endorse}_{\eta_1}(x_1); \dots t_n := \text{endorse}_{\eta_n}(x_n); \llbracket c_1[t_i/x_i] \rrbracket \text{ else } \llbracket c_2 \rrbracket$
- $\llbracket c_1; c_2 \rrbracket \Longrightarrow \llbracket c_1 \rrbracket; \llbracket c_2 \rrbracket$
- $\llbracket \text{if } e \text{ then } c_1 \text{ else } c_2 \rrbracket \Longrightarrow \text{if } e \text{ then } \llbracket c_1 \rrbracket \text{ else } \llbracket c_2 \rrbracket$
- $\llbracket \text{while } e \text{ do } c \rrbracket \Longrightarrow \text{while } e \text{ do } \llbracket c \rrbracket$
- $\llbracket c \rrbracket \Longrightarrow c$ , for other commands  $c$ .

*Adequacy of translation for checked endorsements for well-typed programs.* Next we show adequacy of the labeled translation of Definition 23 for well-typed programs. Note that for non-typed programs this adequacy does not hold, as shown by an example in the end of the section.

Without loss of generality, we assume checked endorsements have only one variable ( $n = 1$  in the translation of checked endorsement in Definition 23). We adopt an indexing convention where checked endorsement with the label  $\eta_i$ , corresponds to two direct endorsements with the labels  $\eta_{2i-1}$  and  $\eta_{2i}$ . The following lemma establishes a connection between irrelevant attacks of the source and translated runs.

**Lemma 7** (Synchronized endorsements). Given a program  $c[\vec{\sigma}]$  that only uses checked endorsements, such that  $\Gamma, pc \vdash c[\vec{\sigma}]$ , memory  $m$ , and attack  $\vec{a}$ , such that

$$\langle c[\vec{a}], m \rangle \longrightarrow_{\vec{t}}^* \text{ and } \langle \llbracket c[\vec{a}] \rrbracket, m \rangle \longrightarrow_{\hat{\vec{t}}}^*$$

where

- $\vec{t} = \vec{t}' \cdot \text{checked}(\eta_i, v_i, b_i)$  and
- $\hat{\vec{t}} = \hat{\vec{t}}' \cdot \text{endorse}(\eta_{2i-1}, 0)$  or  $\hat{\vec{t}} = \hat{\vec{t}}' \cdot \text{endorse}(\eta_{2i-1}, 1) \cdot \text{endorse}(\eta_{2i}, v)$
- $k$  is a number of checked endorse events in  $\vec{t}$  and

we have that

- $R(c[\vec{\sigma}], m, \vec{a}, \vec{t}) = R(\llbracket c[\vec{\sigma}] \rrbracket, m, \vec{a}, \hat{\vec{t}})$ .
- $R_{\rightarrow}(c[\vec{\sigma}], m, \vec{a}, \vec{t}) = R_{\rightarrow}(\llbracket c[\vec{\sigma}] \rrbracket, m, \vec{a}, \hat{\vec{t}})$ .
- $\Phi(\llbracket c[\vec{\sigma}] \rrbracket, m, \hat{\vec{t}}) = \Psi(c[\vec{\sigma}], m, \vec{t})$

**Proof.** The first two items follow from the definition of the translation, because the translation does not generate new release events.

To prove the second item, we consider partitions of irrelevant traces generated by every  $k$ -th checked endorsement and the direct endorsement(s) that correspond to it. We proceed by induction on  $k$ . For the base case,  $k = 0$ , i.e., neither  $\vec{t}$  nor  $\hat{\vec{t}}$  contain endorsements, it holds that  $\Phi(\llbracket c[\vec{\sigma}] \rrbracket, m, \hat{\vec{t}}) = \Psi(c[\vec{\sigma}], m, \vec{t}) = \emptyset$ . For the inductive case, define a pair of auxiliary sets

$$\begin{aligned} F_k &\triangleq \Phi(\llbracket c[\vec{\sigma}] \rrbracket, m, \hat{\vec{t}}) \setminus \Phi(\llbracket c[\vec{\sigma}] \rrbracket, m, \hat{\vec{t}}') \\ P_k &\triangleq \Psi(c[\vec{\sigma}], m, \vec{t}) \setminus \Psi(c[\vec{\sigma}], m, \vec{t}') \end{aligned}$$

By the induction hypothesis,  $\Phi(c[\vec{\sigma}], m, \hat{\vec{t}}) = \Psi(c[\vec{\sigma}], m, \vec{t})$ . By Definitions 16 and 21, we know that  $\Phi(\llbracket c[\vec{\sigma}] \rrbracket, m, \hat{\vec{t}}) \supseteq \Phi(\llbracket c[\vec{\sigma}] \rrbracket, m, \hat{\vec{t}}')$  and  $\Psi(c[\vec{\sigma}], m, \vec{t}) \supseteq \Psi(c[\vec{\sigma}], m, \vec{t}')$ . Therefore, in order to prove that  $\Phi(\llbracket c[\vec{\sigma}] \rrbracket, m, \hat{\vec{t}}) = \Psi(c[\vec{\sigma}], m, \vec{t})$  it is sufficient to show that  $F_k = P_k$ . We consider each direction of equivalence separately.

- $F_k \supseteq P_k$ . Take an attack  $\vec{b} \in P_k$ . That is  $\langle c[\vec{b}], m \rangle$  produces a trace  $\vec{q}$  such that  $\vec{q}$  agrees on all checked endorsements with  $\vec{t}$  except the last one. There are three possible ways in which these endorsements may disagree:
  - (1) Trace  $\vec{t}$  contains  $\text{checked}(\eta_k, v_k, 1)$  and  $\vec{q}$  contains  $\text{checked}(\eta_k, v'_k, 1)$  such that  $v_k \neq v'_k$ . By the rules for the translation, it must be that the trace  $\hat{\vec{t}}$ , which is produced by configuration  $\langle \llbracket c[\vec{a}] \rrbracket, m \rangle$ , has two corresponding endorsement events  $\text{endorse}(\eta_{2k-1}, 1)$  and  $\text{endorse}(\eta_{2k}, v_k)$ . Similarly, the trace  $\vec{q}$ , produced by  $\langle \llbracket c[\vec{b}] \rrbracket, m \rangle$ , has two corresponding endorsement events  $\text{endorse}(\eta_{2k-1}, 1)$  and  $\text{endorse}(\eta_{2k}, v'_k)$ . Because  $v'_k \neq v_k$  we have that  $\hat{\vec{q}} \in \phi(\hat{\vec{t}})$ .
  - (2) Trace  $\vec{t}$  contains checked endorsement event  $\text{checked}(\eta_k, v_k, 1)$ , while trace  $\vec{q}$  contains event  $\text{checked}(\eta_k, v'_k, 0)$ . In this case, the trace  $\hat{\vec{t}}$  obtained from running  $\langle \llbracket c[\vec{a}] \rrbracket, m \rangle$  must contain two endorsement events  $\text{endorse}(\eta_{2k-1}, 1)$  and  $\text{endorse}(\eta_{2k}, v_k)$ , while the trace  $\hat{\vec{q}}$  corresponding to  $\langle \llbracket c[\vec{b}] \rrbracket, m \rangle$  contains one event  $\text{endorse}(\eta_{2k-1}, 0)$ . Therefore,  $\hat{\vec{q}} \in \phi(\hat{\vec{t}})$ .
  - (3) Trace  $\vec{t}$  contains checked endorsement event  $\text{checked}(\eta_k, v_k, 0)$ , while trace  $\vec{q}$  contains event  $\text{checked}(\eta_k, v'_k, 1)$ . This is similar to the previous case.

From  $\hat{q} \in \phi(\hat{t})$  it follows that  $\vec{b} \in F_k$ .

- $F_k \subseteq P_k$ . Take an attack  $\vec{b} \in F_k$ . There must be a trace  $\hat{q}$ , produced by  $\langle \llbracket c[\vec{b}] \rrbracket, m \rangle$ , such that  $\hat{q} \in \phi(\hat{t})$ . There are two ways this can happen:

- (1)  $\hat{q}$  and  $\hat{t}$  disagree at the translated endorsement event that has label  $\eta_{2k-1}$ . More precisely, one must have form  $endorse(\eta_{2k-1}, 1)$  and the other,  $endorse(\eta_{2k-1}, 0)$ . In the original run, this corresponds to two traces  $\vec{t}$  and  $\vec{q}$  such that  $\vec{t}$  contains the event  $checked(\eta_k, b_k, v_k)$  and  $\vec{q}$  contains the event  $checked(\eta_k, b'_k, v'_k)$ . We know that  $b_k = 1$  and  $b'_k = 0$ , and hence  $b_k + b'_k \leq 1$ . According to Definition 20, we need to show that  $v'_k \neq v_k$ . Assume this is not the case, and that  $v'_k = v_k$ . Then by rule (T-CHECKED), we have  $b_k = b'_k$ , which contradicts the earlier conclusion. Hence  $\vec{q} \in \psi(\vec{t})$ .
- (2) Alternatively,  $\hat{q}$  and  $\hat{t}$  disagree at the endorsement event that has label  $\eta_{2k}$ . This also means that they agree on the earlier endorsement, i.e., for the corresponding trace with checked endorsement, we can show that  $b_k = b'_k = 1$ , and  $v_k \neq v'_k$ . Therefore,  $\vec{q} \in \psi(\vec{t})$ .

From  $\vec{q} \in \psi(\vec{t})$  it follows that  $\vec{b} \in P_k$ .

Using Lemma 7 we can show the following Proposition, which relates the security of the source and translated programs.

**Proposition 2** (Relation of checked and direct endorsements). Given a program  $c[\vec{\bullet}]$  that only uses checked endorsements such that  $\Gamma, pc \vdash c[\vec{\bullet}]$ , then  $c[\vec{\bullet}]$  satisfies progress-insensitive robustness for checked endorsements if and only  $\llbracket c[\vec{\bullet}] \rrbracket$  satisfies progress-insensitive robustness for direct endorsements.

**Proof.** Note that our translation preserves typing: when  $\Gamma, pc \vdash c[\vec{\bullet}]$ , then  $\Gamma, pc \vdash \llbracket c[\vec{\bullet}] \rrbracket$ . Therefore, by Proposition 1 the translated program satisfies progress-insensitive robustness with endorsements. To show that the source program satisfies the progress-insensitive robustness with checked endorsements, we use Lemma 7 and note that the corresponding sets of irrelevant attacks and control between any two runs of the programs must be in sync.  $\square$

*Notes on the adequacy of the translation.* We observe two facts about the adequacy of this translation. First, for non-typed programs, the relation does not hold. For instance, a program like

$$[\bullet]; \text{endorse}(u) \text{ if } u = u' \text{ then } low := \text{declassify}(u < h) \text{ else skip}$$

does not satisfy Definition 22. However, translation of this program satisfies Definition 17.

Second, observe that omitting endorsement of the expression would lead to occlusion. Consider an alternative translation that endorses only the variables  $x_1, \dots, x_n$  but not the result of the whole expression. Using such a translation, a program

$$\text{if } u \cdot 0 > 0 \text{ then skip else skip; } trusted := x$$

is translated to

$$temp := x; \text{if } t \cdot 0 > 0 \text{ then skip else skip; } trusted := x$$

However, while the first program does not satisfy Definition 22, the second program is accepted by Definition 17.

## 8. Attacker impact

In prior work, robustness controls the attacker’s ability to cause information release. In the presence of endorsement, the attacker’s ability to influence trusted locations also becomes an important security issue. To capture this influence, we introduce an integrity dual to attacker knowledge, called *attacker impact*. Similarly to low events, we define *trusted events* as assignments to trusted variables and termination.

**Definition 24** (Attacker impact). Given a program  $c[\bullet]$ , memory  $m$ , and trusted events  $\vec{t}_*$ , define  $p(c[\bullet], m, \vec{t}_*)$  to be a set of attacks  $\vec{a}$  that match trusted events  $\vec{t}_*$ :

$$p(c[\bullet], m, \vec{t}_*) \triangleq \{\vec{a} \mid \langle c[\vec{a}], m \rangle \xrightarrow{*}_{\vec{t}} \wedge \vec{t}_* = \vec{t}_{\mathbb{T}}\}$$

Attacker impact is defined with respect to a given sequence of trusted events  $\vec{t}_*$ , starting in memory  $m$ , and program  $c[\bullet]$ . The impact is the set of all attacks that agree with  $\vec{t}_*$  in their footprint on trusted variables.

Intuitively, a smaller set for attacker impact means that the attacker has greater power to influence trusted events. Similarly to progress knowledge, we define *progress impact*, characterizing which attacks lead to one more trusted event. This then allows us to define robustness conditions for *integrity*, which have not previously been identified.

**Definition 25** (Progress impact). Given a program  $c[\bullet]$ , memory  $m$ , and sequence of trusted events  $\vec{t}_*$ , define progress impact  $p_{\rightarrow}(c[\bullet], m, \vec{t}_*)$  as

$$p_{\rightarrow}(c[\bullet], m, \vec{t}_*) \triangleq \{\vec{a} \mid \langle c[\vec{a}], m \rangle \xrightarrow{*}_{\vec{t}} \langle c', m' \rangle \wedge \vec{t}_* = \vec{t}'_{\mathbb{T}} \wedge \langle c', m' \rangle \xrightarrow{*}_{t''}\}$$

The intuition for the baseline robustness for integrity is that attacker should not influence trusted data. This is similar to noninterference for integrity (modulo availability attacks, which have not been explored in this context before). However unlike earlier work, we can easily extend the notion of integrity robustness to endorsements and checked endorsements.

**Definition 26** (Progress-insensitive integrity robustness with endorsements). A program  $c[\bullet]$  satisfies progress-insensitive robustness for integrity if for all memories  $m$ , and for all traces  $\vec{t} \cdot t_*$  where  $t_*$  is a trusted event, we have

$$p_{\rightarrow}(c[\bullet], m, \vec{t}_{\mathbb{T}}) \setminus \Phi(c[\bullet], m, \vec{t} \cdot t_*) \subseteq p(c[\bullet], m, \vec{t}_{\mathbb{T}} \cdot t_*)$$

Irrelevant attacks are defined precisely as in Section 5. We omit the corresponding definitions for programs without endorsements and with checked endorsements.

The type system of Section 6 also enforces integrity robustness with endorsements, rejecting insecure programs such as  $t := u$  and `if` ( $u_1$ ) `then`  $t := \text{endorse}(u_2)$ , but accepting  $t := \text{endorse}(u)$ . Moreover, a connection between checked and direct endorsements, analogous to Proposition 2, holds for integrity robustness too.

## 9. Examples

*Password update.* Figure 11 shows code for updating a password. The attacker controls variables `guess` of level  $(\mathbb{P}, \mathbb{U})$  and `new_password` of level  $(\mathbb{S}, \mathbb{U})$ . The variable `password` has level  $(\mathbb{S}, \mathbb{T})$  and variables `nfailed` and `ok` have level  $(\mathbb{P}, \mathbb{T})$ . The declassification on line 3 uses the untrusted variable `guess`. This variable, however, is listed in the `endorse` clause on line 2; therefore, the declassification is accepted. The initially untrusted variable `new_password` has to be endorsed to

```

1  [●]
2  endorse(guess, new_password)
3  if (declassify(guess==password))
4  then
5    password = new_password;
6    nfailed = 0;
7    ok = true;
8  else
9    nfailed = nfailed + 1;
10   ok = false;

```

FIGURE 11. Password update

```

1  [●]
2  endorse(req_time)
3  if (req_time <= now)
4  then
5    if (req_time >= embargo_time)
6      then return declassify(new_data)
7      else return old_data
8  else
9    return old_data

```

FIGURE 12. Accessing embargoed information

update the password on line 5. The example also shows how other trusted variables—`nfailed` and `ok`—can be updated in the `then` and `else` branches.

*Data sanitization.* Figure 12 shows an annotated version of the code from the introduction, in which some information (`new_data`) is not allowed to be released until time `embargo_time`. The attacker-controlled variable is `req_time` of level ( $\mathbb{P}, \mathbb{U}$ ), and `new_data` has level ( $\mathbb{S}, \mathbb{T}$ ). The checked endorse ensures that the attacker cannot violate the integrity of the test `req_time >= embargo_time`. (Variable `now` is high-integrity and contains the current time). Without the checked endorse, the release of `new_data` would not be permitted either semantically or by the type system.

## 10. Related work

Prior robustness definitions [16, 10], based on equivalence of low traces, do not differentiate programs such as  $[\bullet]; low := u < h; low' := h$  and  $[\bullet]; low' := h; low := u < h$ ; Per dimensions of information release [24], the new security conditions cover not only the “who” dimension, but are also sensitive to “where” information release happens. Also, the security condition of robustness with endorsement does not suffer from the occlusion problems of qualified robustness. Balliu and Mastroeni [6] derive sufficient conditions for robustness using weakest precondition semantics. These conditions are not precise enough to distinguish the examples above, and, moreover, do not support endorsement.

Prior work on robustness semantics defines termination-insensitive security conditions [16, 6]. Because the new framework is powerful enough to capture the security of programs with intermediate observable events, it can describe the robustness of nonterminating programs. Prior work on qualified robustness [16] uses a non-standard *scrambling* semantics in which qualified robustness unfortunately becomes a *possibilistic* condition, leading to anomalies such as reachability of dead code. The new framework avoids such artifacts because it uses a standard, deterministic semantics.

Checked endorsement was introduced informally in the Swift web application framework [9] as a convenient way to implement complex security policies. The current paper is the first to formalize and to study the properties of checked endorsement.

Our semantic framework is based on the definition of attacker knowledge, developed in prior work introducing *gradual release* [1]. Attacker knowledge is used for expressing confidentiality policies in recent work [7, 3, 2, 8]. However, none of this work considers integrity; applying attacker-centric reasoning to integrity policies is novel.

## 11. Conclusion

We have introduced a new knowledge-based framework for semantic security conditions for information security with declassification and endorsement. A key technical innovation is characterizing the impact and control of the attacker over information in terms of sets of similar attacks. Using this framework, we can express semantic conditions that more precisely characterize the security offered by a security type system, and derive a satisfactory account of new language features such as checked endorsement.

## References

- [1] 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.
- [2] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.
- [3] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, pages 333–348, October 2008.
- [4] A. Askarov and A. C. Myers. A semantic framework for declassification and endorsement. In *Proc. 19th European Symp. on Programming (ESOP'10)*, March 2010.
- [5] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proc. 10th European Symposium on Research in Computer Security (ESORICS)*, number 3679 in Lecture Notes in Computer Science. Springer-Verlag, September 2005.
- [6] M. Balliu and I. Mastroeni. A weakest precondition approach to active attacks analysis. In *PLAS '09: Proc. of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 59–71. ACM, 2009.
- [7] 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.
- [8] N. Broberg and D. Sands. Flow-sensitive semantics for dynamic information flow policies. In S. Chong and D. Naumann, editors, *ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS 2009)*, Dublin, June 15 2009. ACM.
- [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. SOSP 2007*, pages 31–44, October 2007.
- [10] S. Chong and A. C. Myers. Decentralized robustness. In *CSFW '06: Proc. of the 19th IEEE workshop on Computer Security Foundations*, pages 242–256, Washington, DC, USA, 2006. IEEE Computer Society.
- [11] M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: Toward a secure voting system. In *Proc. IEEE Symp. on Security and Privacy*, pages 354–368, May 2008.
- [12] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proc. 20th ACM Symp. on Operating System Principles (SOSP)*, October 2005.
- [13] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.
- [14] M. D. McIlroy and J. A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8):673–694, August 1992.
- [15] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.
- [16] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. Computer Security*, 14(2):157–196, May 2006.
- [17] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, Saint-Malo, France, 1997.
- [18] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif 3.0: Java information flow. Software release, <http://www.cs.cornell.edu/jif>, July 2006.
- [19] 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.
- [20] P. Ørnbæk and J. Palsberg. Trust in the  $\lambda$ -calculus. *J. Functional Programming*, 7(6):557–591, 1997.
- [21] F. Pottier and S. Conchon. Information flow inference for free. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 46–57, 2000.

- [22] A. W. Roscoe. Csp and determinism in security modeling. In *Proc. IEEE Symposium on Security and Privacy*, 1995.
- [23] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
- [24] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, 2009.
- [25] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 355–364, San Diego, California, January 1998.
- [26] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Proc. IEEE Computer Security Foundations Workshop*, pages 34–43, June 1998.
- [27] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.
- [28] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.
- [29] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.
- [30] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *Proc. 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 293–308, 2008.

## Acknowledgments

The authors would like to thank the anonymous reviewers for comments on a draft of this paper. We also thank Owen Arden, Stephen Chong, Michael Clarkson, Daniel Hedin, Andrei Sabelfeld, and Danfeng Zhang for useful discussions.

This work was supported by a grant from the Office of Naval Research (ONR N000140910652) and by two NSF grants (the TRUST center, 0424422; and 0964409). The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of any of the funding agencies or of the U.S. Government.