A Semantic Framework for Declassification and Endorsement

Aslan Askarov   Andrew Myers

Department of Computer Science, Cornell University

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 described in terms of the power that declassification and endorsement give the attacker. The new framework specifies how attacker-controlled code affects program execution and what the attacker is able to learn from observable effects of this code. 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 [17] and at the operating-system level. The language-based approach holds the appeal that the security property of noninterference [11], can be provably enforced using a type system [19]. 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* [14] and *endorsement* [16, 21], 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: for example, the various applications built with Jif [12, 15], including games [4], a voting system [10], and web applications [8].

A continuing challenge is to understand what security is obtained when code uses downgrading. The contribution of this paper is providing a more precise and satisfactory answer to this question. Much prior work on declassification is usefully summarized by

Sands and Sabelfeld [18]. However, there is comparatively little work on characterizing the security of declassification in the presence of endorsement. Because confidentiality and integrity are not independent, it is important to understand how endorsement weakens confidentiality.

To see an interaction between endorsement and confidentiality, 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`, that means 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 [20, 13, 9] 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 [15]. However, the security conditions for robustness are not satisfactory. First, they largely ignore the possibility of endorsement, with the exception of *qualified robustness* [13], which works by giving the `endorse` operation a somewhat ad-hoc, nondeterministic semantics. Second, prior conditions only characterize information security for programs that terminate. 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.

The main contribution of this paper is a general, language-based semantic framework for expressing information flow security. This semantically captures the ability of the attacker to influence knowledge. The robust interaction of integrity and confidentiality can then be captured cleanly as a constraint on attacker control. 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.
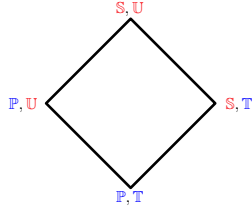
Fig. 1: Information flow lattice

$$e ::= n \mid x \mid e \; op \; e$$
$$c ::= \texttt{skip} \mid x := e \mid c; c$$
$$\mid \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } e \texttt{ do } c$$

Fig. 2: Syntax of the language

We show that the progress-insensitive variants of these improved security conditions can be 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 [8], 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.

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 power. Additional examples are presented in Section 9, related work is discussed in Section 10, and Section 11 concludes.

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 information flow ordering to elements on this lattice: $\ell_1 \sqsubseteq \ell_2$ if the ordering holds between the corresponding components. As standard, we define *join* $\ell_1 \sqcup \ell_2$ as the least upper bound of ℓ_1 and ℓ_2, and *meet* $\ell_1 \sqcap \ell_2$ as the greatest upper bound of ℓ_1 and ℓ_2.

Language and semantics We consider a simple imperative language with syntax presented on Figure 2. The semantics of the language is fairly standard. 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. For a single transition, we write $\langle c, m \rangle \longrightarrow_t \langle c', m' \rangle$, where c and m are the initial command and memory, c' and m' are the resulting command and memory. The transitions defined by the semantics are fully standard, and are described in detail in the associated technical report. 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). We write $\langle c, m \rangle \longrightarrow^*_{\vec{t}}$ 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 $t_1 \ldots t_k \ldots$, and a *prefix* of $\vec{t}$ up to the i-th event is denoted as $\vec{t}_i$. If a transition does not affect memory, its event is *empty*, which is either written as ϵ or is omitted, e.g.: $\langle c, m \rangle \longrightarrow \langle c', m' \rangle$.

Finally, we assume that the *security environment* Γ maps program variables to their security levels. Given a memory m we write $m_{\mathbb{P}}$ for the public part of the memory.

2.1 Attacker knowledge

This section provides background on the attacker-centric model for information flow security [2]. 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 set of low (or public) events. Low events correspond to observations that an attacker can make during a run of the program. We assume that 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.

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 ℓ, and a sequence of low events is denoted as $\vec{\ell}$. We overload the notation for semantic transitions, writing $\langle c, m \rangle \longrightarrow^*_{\vec{\ell}}$ if only low events produced from configuration $\langle c, m \rangle$ are relevant, that is there is a trace $\vec{t}$ such that $\langle c, m \rangle \longrightarrow^*_{\vec{t}} \wedge \vec{t}_{\mathbb{P}} = \vec{\ell}$. Low events are the key element in the definition of *attacker knowledge* [2].

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

Attacker knowledge gives a handle on what information 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* [1, 3] and *divergence knowledge* [1].

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

Progress knowledge represents the information the attacker obtains by seeing public events $\vec{\ell}$ followed by one more public event. Progress knowledge and attacker knowledge are related as follows: given a program c, memory m and a sequence of low events $\ell_1 \ldots \ell_n$ 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, consider program $l := 0; \mathtt{while}\ h = 0\ \mathtt{do\ skip}; l := h$ with initial memory $m(h) = 7$. This program produces a sequence of two low events $(l, 0)(l, 7)$. The knowledge after the first event $k(c, m_\mathbb{P}, (l, 0))$ is a set of all possible memories. 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 $k(c, m_\mathbb{P}, (l, 0)(l, 7))$ is a set of memories where $h = 7$.

Using attacker knowledge, one can express many confidentiality policies [6, 3, 7]. For example, a strong notion of *progress-sensitive noninterference* [11] 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* (in [1] it is called *termination-insensitive*), which allows leakage of information, but only via termination channels. This is expressed by requiring equivalence of 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 $l := 0; \mathtt{while}\ h = 0\ \mathtt{do\ skip}; l := 1$, the knowledge inclusion between the two events is strict: $k(c, m_\mathbb{P}, (l, 0)) \supset k(c, m_\mathbb{P}, (l, 0)(l, 1))$. Therefore, the example does not satisfy progress-sensitive noninterference. On the other hand, the low event that follows the $\mathtt{while}$ loop does not reveal more information than the knowledge about the existence of that event. Formally, $k_\rightarrow(c, m_\mathbb{P}, (l, 0)) = k(c, m_\mathbb{P}, (l, 0)(l, 1))$, 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 $\Uparrow$. For attackers that may observe program divergence $\Uparrow$, we define knowledge on the sequence of low events that includes divergence (we write $\langle c, m \rangle \Uparrow$ to mean configuration $\langle c, m \rangle$ diverges):

Definition 3 (Divergence knowledge)

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

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*

$$\langle c, m \rangle \longrightarrow^*_{\vec{\ell}} \langle c', m' \rangle \longrightarrow^*_r$$

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

For example, in the program $low := 1; low' := h$, the second assignment is both a progress-sensitive and a progress-insensitive release event. 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 security of program in the presence of active attacks, we introduce a formal model of the attacker. Our formalization follows the one in [13], 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.

We extend the syntax to allow execution of attacker-controlled code:

$$c[\vec{\bullet}] ::= \ldots \mid [\bullet]$$

We limit attacks that can be substituted into the holes to so-called *fair attacks* — attacks that do not read confidential information and do not modify trusted variables.

Definition 5 (Fair attacks) *An attack is a vector of commands $\vec{a}$ that are substituted in place of holes in $c[\vec{\bullet}]$. Fair attacks are defined by the following grammar where for all variables y in e we have $\Gamma(y) \sqsubseteq (\mathbb{P}, \mathbb{U})$ and for variable x in assignments we have $(\mathbb{P}, \mathbb{U}) \sqsubseteq \Gamma(x)$.*

$$a ::= \texttt{skip} \mid x := e \mid a; a \mid \texttt{if } e \texttt{ then } a \texttt{ else } a \mid \texttt{while } e \texttt{ do } a$$

3.1 Examples of attacker influence

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 is executed 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 $[(u, 0)](low, 0)$, while the second trace results in low events $[(u, 10)](low, 1)$. This also means that the knowledge about the secret is different in each trace. We have

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

Clearly, in this program the attacker has 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 this, consider attack $a_3 = u := 5$. This

attack results in low events $[(u, 5)](low, 0)$, that do the same assignment to low as a_1 does. Attacker knowledge, however, is different from that obtained by a_1:

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

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 $a = \texttt{while true do skip}$, 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 $[\bullet]; \texttt{while } u = 0 \texttt{ do skip}; low := h$. In this program, any attack that sets u to 0 prevents the assignment from happening.

Consider another example $[\bullet]; \texttt{while } u < h' \texttt{ do skip}; low := 1$. As in the previous example, the value of u may change the reachability of $low := 1$. However, this is not a pure availability attack, because (assuming the attacker can observe divergence) diverging before the last assignment gives the attacker additional information about secrets, 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. Here, 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:

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

We compare $\vec{a}$ and $\vec{b}$ based on how they change attacker knowledge along their respective traces. First, if knowledge is preserved along one of the traces, say $\vec{t}$, it must be preserved along $\vec{q}$ 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.

Visually, this requirement is described by the two diagrams in Figure 3. 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 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.

7

Definition 6 (Knowledge segmentation) *Given a program c, memory m, and a trace $\vec{t}$, a sequence of indexes $p_1, \ldots p_N$ such that $p_1 < p_2 \cdots < p_N$ and $\vec{t}_\mathbb{P} = \ell_{1 \ldots p_1} \ell_{p_1+1 \ldots p_2} \ldots \ell_{p_{N-1}+1 \ldots p_N}$ 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* $\mathrm{Seg}(c, m, \vec{t}, p_1 \ldots p_N)$.
- progress-insensitive knowledge segmentation *of size N if*
  $\forall j \leq N, \forall i \,.\, p_{j-1} + 1 \leq i < p_j \,.\, k_\rightarrow(c, m_\mathbb{P}, \vec{\ell}_i) = k(c, m_\mathbb{P}, \vec{\ell}_{i+1})$, *denoted by* $\mathrm{Seg}_\rightarrow(c, m, \vec{t}, p_1 \ldots p_N)$.

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.

Definition 7 (Similar attacks and traces $\sim^{c[\vec{\bullet}],m}$) *Given a program $c[\vec{\bullet}]$, 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 $p_1 \ldots p_N$ and $p'_1 \ldots p'_N$ (for some N) such that $\mathrm{Seg}(c[\vec{a}], m, \vec{t}, p_1 \ldots p_N)$, $\mathrm{Seg}(c[\vec{b}], m, \vec{q}, p'_1 \ldots p'_N)$, 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 $\mathrm{Seg}_\rightarrow$. If two attack–trace pairs are similar, we write $(\vec{a}, \vec{t}) \sim^{c[\vec{\bullet}],m} (\vec{b}, \vec{q})$ (for progress-insensitive similarity, $(\vec{a}, \vec{t}) \sim_\rightarrow^{c[\vec{\bullet}],m} (\vec{b}, \vec{q})$).
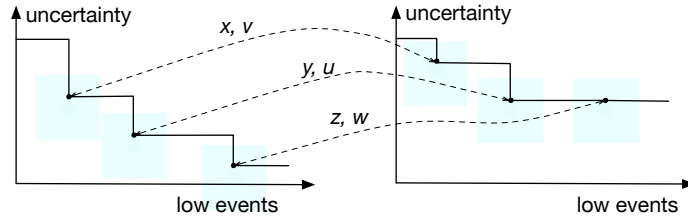

Fig. 3: Similar attacks and traces

The construction of Definitions 6 and 7 can be exemplified by program $[\bullet]$; `if` u `then (` `while` $h \leq 100$ `do skip)` `else skip`; $low_1 := 0$; $low_2 := h > 100$. 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 8 (Attacker control (progress-sensitive))
$$R(c[\vec{\bullet}], m, \vec{a}, \vec{t}) \triangleq \{\vec{b} \mid \exists \vec{q} \,.\, (\vec{a}, \vec{t}) \sim^{c[\vec{\bullet}],m} (\vec{b}, \vec{q})\}$$

To illustrate how attacker control changes, consider example program $[\bullet]; low := u < h; low' := h$ where u is an untrusted variable and h is a secret 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 a 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 9 (Attacker control (progress-insensitive))

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

Consider program $[\bullet]; \texttt{while } u < h \texttt{ do skip}; low := 1$. 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 Next, we define *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 10 (Release control (progress-sensitive))

$$\begin{aligned}
R^\triangleright(c[\vec{\bullet}], m, \vec{a}, \vec{t}) \triangleq \{\vec{b} \mid \exists \vec{q} . (\vec{a}, \vec{t}) \sim^{c[\vec{\bullet}], m} (\vec{b}, \vec{q}) \wedge \\
(\exists \vec{r'} . k(c[\vec{b}], m_\mathbb{P}, \vec{q}_\mathbb{P}) \supset k(c[\vec{b}], m_\mathbb{P}, \vec{q}_\mathbb{P}\vec{r'}_\mathbb{P}) \\
\vee k(c[\vec{b}], m_\mathbb{P}, \vec{q}_\mathbb{P}) \supset k(c[\vec{b}], m_\mathbb{P}, \vec{q}_\mathbb{P} \Uparrow) \\
\vee \langle c[\vec{b}], m \rangle \Downarrow)\}
\end{aligned}$$

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 additional clause (on the third line) that uses divergence knowledge to reflect that.

Figure 4a 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 top 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 $\vec{tr}$, where $\vec{r}$ contains a release event, we have the following relation between release control and attacker control:

$$R(c[\bullet], m, \vec{a}, \vec{t}) \supseteq R^\triangleright(c[\bullet], m, \vec{a}, \vec{t}) \supseteq R(c[\bullet], m, \vec{a}, \vec{tr})$$

Note the white gaps and the gray release control above the dotted lines on Figure 4a. The white gaps correspond to difference $R(c[\bullet], m, \vec{a}, \vec{t}) \setminus R^\triangleright(c[\bullet], m, \vec{a}, \vec{t})$. This is a set of attacks that do not produce further release events and 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 $R^\triangleright(c[\bullet], m, \vec{a}, \vec{t}) \setminus R(c[\bullet], m, \vec{a}, \vec{tr})$. In particular, when this set is non-empty, the attacker can launch attacks corresponding to each of the last three lines of Definition 10:

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 12 and 13. Before moving on to these definitions, we introduce the progress-insensitive variant of release control.

Definition 11 (Release control (progress-insensitive))

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

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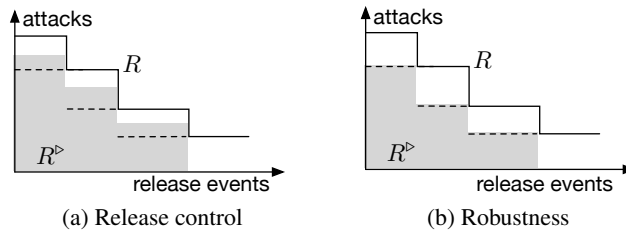

(a) Release control   (b) Robustness

Fig. 4: Release control and robustness

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 12 (Progress-sensitive robustness) *Program $c[\vec{\bullet}]$ satisfies* progress-sensitive robustness *if for all memories m and attacks $\vec{a}$, s.t. $\langle c[\vec{a}], m \rangle \longrightarrow^*_{\vec{a}} \langle c', m' \rangle \longrightarrow^*_{\vec{r}}$, 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*

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

Note that set inclusion in the above definition could be replaced with strict equality, but we use $\subseteq$ for compatibility with future definitions. Figure 4b 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 $[\bullet]; low := u < h$, and memory such that $m(h) = 7$. This program is rejected by Definition 12. 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 $R(c[\vec{\bullet}], m, a, [(u, 5)](low, 1))$ is a set of all attacks where $u < 7$, which is smaller than $R^\triangleright$. Therefore this program does not satisfy the condition.

Program $[\bullet]; low := h; low' := u < h$ 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 $[\bullet];$ if $u > 0$ then $low := h$ else skip is rejected. Consider memory $m(h) = 7$, and attack $a = u := 1$ that leads to low trace $[(u, 1)], (low, 7)$. The attacker control for this attack and trace is a 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 $[\bullet];$ while $u > 0$ do skip$; low := h$ 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 12.

Program $[\bullet];$ while $u > h$ do skip$; low := 1$ 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 12, but uses progress-insensitive variants of release events, control, and release control. As a result, program $[\bullet];$ while $u > h$ do skip$; low := 1$ is accepted: attacker control is all attacks.

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

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

5   Endorsement

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

11

Syntax and semantics   We extend the language syntax with endorsement:

$$c[\vec{\bullet}] ::= \ldots \mid x := \texttt{endorse}_\eta(e)$$

We assume that every endorsement in the program source has a unique *endorsement label* η. Semantically, endorsements produce *endorsement events* which record the label of the endorsement statement together with the value that is endorsed. Whenever the endorsement label is unimportant we omit it from the examples.

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

Note that $endorse(\eta_i, v)$ events need not mention variable name x since that information is implied by the unique label η_i.

Irrelevant attacks   Given a trace $\vec{t}$, we introduce *irrelevant attacks* $\Phi(\vec{t})$ as 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 all endorsements but the last one. We define this set as follows.

Definition 14 (Irrelevant traces) *Given a trace $\vec{t}$, where endorsements are marked as* $endorse(\eta_j, v_j)$, *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*

$$\phi_i(\vec{t}) = \{\vec{t'} \mid \vec{t'} = \ldots endorse(\eta_{i-1}, v_{i-1}) \ldots endorse(\eta_i, v_i') \ldots\} \ s.t. \ v_i \neq v_i'$$

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

Definition 15 (Irrelevant attacks) $\Phi(c[\vec{\bullet}], m, \vec{t}) \triangleq \{\vec{a} \mid \langle c[\vec{a}], m \rangle \longrightarrow^*_{\vec{t'}} \land \vec{t'} \in \phi(\vec{t})\}$

Security   The security conditions for robustness can be adjusted now to accommodate endorsements that happen along the trace. The idea is to exclude irrelevant attacks from the left-hand side of Definitions 12 and 13. This security condition, which has both progress-sensitive and progress-insensitive versions, expresses roughly the same idea as *qualified robustness* [13], but in a more natural and direct way.

Definition 16 (Progress-sensitive robustness with endorsements) *Program $c[\vec{\bullet}]$ satisfies* progress-sensitive robustness with endorsement *if for all memories m and attacks $\vec{a}$, such that $\langle c[\vec{a}], m \rangle \longrightarrow^*_{\vec{t}} \langle c', m' \rangle \longrightarrow^*_{\vec{r}}$, 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*

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

We refer to the set $R^\rhd(c[\vec{\bullet}], m, \vec{a}, \vec{t}) \setminus \Phi(c[\vec{\bullet}], m, \vec{t}\vec{r})$ as a set of *relevant attacks*.

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

- Release control $R^{\triangleright}$ is a set of all attacks that reach the low assignment.
- Irrelevant traces $\phi([\bullet]; low := \mathtt{endorse}_{\eta_1}(u < h), m, [(u, 1)]endorse(\eta_1, 0))$ is a set of traces that end in endorsement event $endorse(\eta_1, v)$ such that $v \neq 0$. Thus, irrelevant attacks $\Phi([\bullet]; low := \mathtt{endorse}_{\eta_1}(u < h), m, [(u, 1)]endorse(\eta_1, 0))$ must consist of attacks that reach the low assignment and set u to values $u \geq 7$.
- The left-hand side of Definition 16 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 16 holds and the program is accepted.

Program $[\bullet]; low := \mathtt{endorse}_{\eta_1}(u); low' := u < h''$ 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 $[\bullet]; low := \mathtt{endorse}_{\eta_1}(u < h); low' := u < h''$ 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 $[\bullet]; \mathtt{if}\ u > 0\ \mathtt{then}\ h' := \mathtt{endorse}(u)\ \mathtt{else}\ \mathtt{skip}; low := h' < h$ is rejected. Assume initial memory where $m(h) = m(h') = 7$. Consider attack a_1 that sets $u := 1$ and consider trace $\vec{t}_1$ which it gives. This trace endorses u in the $\mathtt{then}$ branch, overwrites the value of h' with 1, and produces a release event $(low, 1)$. Consider another attack a_2 which 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 $\vec{t}_2$.

We can also define robustness with endorsement in a progress-insensitive way:

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

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

6   Enforcement

This section shows the enforcement of 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). Figure 5 displays type rules for expressions and commands. This type system is based on the one of [13] and is similar to many standard security type systems.

Declassification We extend the language with a language construct for *declassification* of expressions $\mathtt{declassify}(e)$. 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. While declassification has no additional semantics and can be inferred automatically, its use 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 ℓ 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.

$$\Gamma \vdash n : \ell, \emptyset \qquad \Gamma \vdash x : \Gamma(x), \emptyset \qquad \frac{\Gamma \vdash e_1 : \ell_1, D_1 \qquad \Gamma \vdash e_2 : \ell_2, D_2}{\Gamma \vdash e_1 \ op \ e_2 : \ell_1 \sqcup \ell_2, D_1 \cup D_2}$$

$$\frac{\Gamma \vdash e : \ell, D}{\Gamma \vdash \mathtt{declassify}(e) : \ell \sqcap (\mathbb{P}, \mathbb{U}), vars(e)} \qquad \Gamma, pc \vdash \mathtt{skip} \qquad \frac{\Gamma, pc \vdash c_1 \qquad \Gamma, pc \vdash c_2}{\Gamma, pc \vdash c_1; c_2}$$

$$\frac{\Gamma \vdash e : \ell, D \qquad \ell \sqcup pc \sqsubseteq \Gamma(x) \qquad \forall y \in D . \Gamma(y) \sqsubseteq (\mathbb{S}, \mathbb{T}) \qquad D \neq \emptyset \implies pc \sqsubseteq (\mathbb{P}, \mathbb{T})}{\Gamma, pc \vdash x := e}$$

$$\frac{\Gamma \vdash e : \ell, \emptyset \qquad \Gamma, pc \sqcup \ell \vdash c_1 \qquad \Gamma, pc \sqcup \ell \vdash c_2}{\Gamma, pc \vdash \mathtt{if} \ e \ \mathtt{then} \ c_1 \ \mathtt{else} \ c_2} \qquad \frac{\Gamma \vdash e : \ell, \emptyset \qquad \Gamma, pc \sqcup \ell \vdash c}{\Gamma, pc \vdash \mathtt{while} \ e \ \mathtt{do} \ c}$$

$$\frac{pc \sqsubseteq (\mathbb{P}, \mathbb{U})}{\Gamma, pc \vdash \bullet} \qquad \frac{pc \sqsubseteq (\mathbb{S}, \mathbb{T}) \qquad pc \sqsubseteq \Gamma(x) \qquad \Gamma \vdash e : \ell \qquad \ell \sqcap (\mathbb{S}, \mathbb{T}) \sqsubseteq \Gamma(x)}{\Gamma, pc \vdash x := \mathtt{endorse}(e)}$$

Fig. 5: Type system: expressions and commands

Typing of commands Type rule for commands have 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 expression contains declassification ($D \neq \emptyset$). It requires all variables that can be declassified have high integrity, and bounds the pc-label by $(\mathbb{P}, \mathbb{T})$. This 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.

Typing rule for endorsement requires that pc-label is trusted: $pc \sqsubseteq (\mathbb{S}, \mathbb{T})$. 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 ℓ and $(\mathbb{S}, \mathbb{T})$ boosts integrity, but keeps the confidentiality level of ℓ.

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`.

Soundness   Soundness of the type system is stated by the following proposition.

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

The proof of this and following propositions are given in the associated technical report.

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 with checked endorsements and with unchecked endorsements are the same.

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:
$$c[\vec{\bullet}] ::= \ldots \mid \texttt{endorse}_\eta(x) \texttt{ if } e \texttt{ then } c \texttt{ else } c$$
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 η. 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 η, the value of variable that can potentially be endorsed v, and a result of the check b, which can be either 0 or 1.

$$\frac{m(e) \downarrow v \qquad v \neq 0}{\langle \texttt{endorse}_\eta(x) \texttt{ if } e \texttt{ then } c_1 \texttt{ else } c_2, m \rangle \overset{checked(\eta, m(x), 1)}{\longrightarrow} \langle c_1, m \rangle}$$

$$\frac{m(e) \downarrow v \qquad v = 0}{\langle \texttt{endorse}_\eta(x) \texttt{ if } e \texttt{ then } c_1 \texttt{ else } c_2, m \rangle \overset{checked(\eta, m(x), 0)}{\longrightarrow} \langle c_2, m \rangle}$$

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 18 (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*

$$\psi_i(\vec{t}) = \{\vec{t'} \mid \vec{t'} = \ldots checked(\eta_{i-1}, v_{i-1}, b_{i-1}) \ldots checked(\eta_i, v'_i, b'_i) \ldots\}$$
$$such\ that\ (b_i + b'_i \geq 1) \wedge (v_i \neq v'_i)$$

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

Definition 19 (Irrelevant attacks) $\Psi(c[\vec{\bullet}], m, \vec{t}) \triangleq \{\vec{a} \mid \langle c[\vec{a}], m \rangle \longrightarrow^*_{\vec{t'}} \wedge\ \vec{t'} \in \psi(\vec{t})\}$

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

Definition 20 (Progress-sensitive robustness with checked endorsement) *Program $c[\vec{\bullet}]$ satisfies* progress-sensitive robustness with checked endorsement *if for all memories m and attacks $\vec{a}$, such that $\langle c[\vec{a}], m \rangle \longrightarrow^*_{\vec{a}} \langle c', m' \rangle \longrightarrow^*_{\vec{r}}$, 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*

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

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 $[\bullet]; endorse_{\eta_1}(u)$ if $u = u'$ then $low := u < h$ 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 20 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 $\vec{t}_1 = [(u, 5)(u', 0)] checked(\eta_1, 5, 0)$ and $\vec{t}_2 = [(u, 5)(u', 5)] checked(\eta_1, 5, 1)$. Because endorsement in the second trace succeeds, this trace also continues with a low event $(low, 1)$. Following Definition 18 we have that $t_1 \notin \psi(\vec{t}_2(low, 1))$, implying $a_1 \notin \Psi(c[\vec{\bullet}], m, \vec{t}_2(low, 1))$. Therefore, $a_1 \in R^{\triangleright}(c[\vec{\bullet}], m, \vec{a}_2, \vec{t}_2) \setminus \Psi(c[\vec{\bullet}], m, \vec{t}_2(low, 1))$. On the other hand, $a_1 \notin R(c[\vec{\bullet}], m, \vec{a}_2, \vec{t}_2(low, 1))$ because a_1 can produce no low events corresponding to $(low, 1)$.

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

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

This is a semantically faithful encoding: the condition is checked as early as possible.

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

$$\frac{\begin{array}{ccc} \Gamma' \triangleq \Gamma[x_i \mapsto x_i \sqcap (\mathbb{S}, \mathbb{T})] & \Gamma' \vdash e : \ell', D' & pc' \triangleq pc \sqcup \ell' \\ pc' \sqsubseteq (\mathbb{S}, \mathbb{T}) & \Gamma', pc' \vdash c_1 & \Gamma, pc' \vdash c_2 \end{array}}{\Gamma, pc \vdash \texttt{endorse}(x_1, \ldots, x_n) \texttt{ if } e \texttt{ then } c_1 \texttt{ else } c_2}$$

The expression e is type-checked in an environment Γ' in which endorsed variables $x_1, \ldots x_n$ have trusted integrity; its label ℓ' 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 Γ', since the x_i are trusted there.

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

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 $t_1 \ldots t_n$ and replace all occurrences of $x_1 \ldots x_n$ within c_1 with the temporary ones. All other commands are translated to themselves.

$$\llbracket\texttt{endorse}(x_1, \ldots x_n) \texttt{ if } e \texttt{ then } c_1 \texttt{ else } c_2\rrbracket \implies t_0 := \texttt{endorse}_{\eta_0}(e); \texttt{if } t_0$$
$$\texttt{then } t_1 := \texttt{endorse}_{\eta_1}(x_1); \ldots t_n := \texttt{endorse}_{\eta_n}(x_n); \llbracket c_1[t_i/x_i]\rrbracket \texttt{ else } \llbracket c_2\rrbracket$$

The following proposition relates the security of the original 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-(in)sensitive robustness for checked endorsements if and only if $\llbracket c[\vec{\bullet}]\rrbracket$ satisfies progress-(in)sensitive robustness for direct endorsements.*

For non-typed programs, the relation does not hold. For instance, a program like $[\bullet]; \texttt{endorse}(u) \texttt{ if } u = u' \texttt{ then } c_1 \texttt{ else } c_2$ satisfies Defn. 20, but not Defn. 16.

8 Attacker power

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 power*. Similarly to low events, we define *trusted events* as assignments to trusted variables and termination. Given a trace $\vec{t}$, we denote the trusted events in the trace as $\vec{t}_{\mathbb{T}}$. We use notation $t_\star$ for a single trusted event, and $\vec{t}_\star$ for a sequence of trusted events.

Definition 21 (Attacker power) *Given a program $c[\vec{\bullet}]$, memory m, and trusted events $\vec{t}_\star$, define $p(c[\vec{\bullet}], m, \vec{t}_\star)$ to be a set of attacks $\vec{a}$ which match trusted events $\vec{t}_\star$:*

$$p(c[\vec{\bullet}], m, \vec{t}_\star) \triangleq \{\vec{a} \mid \langle c[\vec{a}], m \rangle \longrightarrow^*_{\vec{t'}} \wedge \vec{t}_\star = \vec{t'}_{\mathbb{T}}\}$$

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

Intuitively, a smaller set for attacker power means that the attacker has greater power to influence trusted events. Similarly to progress knowledge, we define *progress power*, 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 22 (Progress power) *Given a program $c[\vec{\bullet}]$, memory m, and sequence of trusted $\vec{t}_\star$, define progress power $p_{\rightarrow}(c[\vec{\bullet}], m, \vec{t}_\star)$ as*

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

Definition 23 (Progress-insensitive integrity robustness with endorsements) *A program $c[\vec{\bullet}]$ satisfies progress-insensitive robustness for integrity if for all memories m, and for all traces $\vec{tt}_\star$ where $t_\star$ is a trusted event, we have*

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

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 $t := \mathtt{endorse}(u_2)$, but accepting $t := \mathtt{endorse}(u)$. Moreover, a connection between checked and direct endorsements, analogous to Proposition 2, holds for integrity robustness too.

9 Examples

Password update   Figure 6 shows code for updating a password. The attacker controls variables $\mathtt{guess}$ of level $(\mathbb{P}, \mathbb{U})$ and $\mathtt{new\_password}$ of level $(\mathbb{S}, \mathbb{U})$. The variable $\mathtt{password}$ has level $(\mathbb{S}, \mathbb{T})$ and variables $\mathtt{nfailed}$ and $\mathtt{ok}$ have level $(\mathbb{P}, \mathbb{T})$. The declassification on line 3 uses the untrusted variable $\mathtt{guess}$. This variable, however, is listed in the $\mathtt{endorse}$ clause on line 2; therefore, the declassification is accepted. The initially untrusted variable $\mathtt{new\_password}$ has to be endorsed to update the password on line 5. The example also shows how other trusted variables—$\mathtt{nfailed}$ and $\mathtt{ok}$—can be updated in the $\mathtt{then}$ and $\mathtt{else}$ branches.

```
 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;
```

Fig. 6: 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
```

Fig. 7: Accessing embargoed information

Data sanitization   Figure 7 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 [13, 9], 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 [18], 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 [5] derive sufficient conditions for robustness using weakest precondition semantics. These conditions are not precise enough for distinguishing the examples above, and, moreover, do not support endorsement.

Prior work on robustness semantics defines termination-insensitive security conditions [13, 5]. 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 [13] 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 framework [8] 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* [2]. Attacker knowledge is used for expressing confidentiality policies in recent work [6, 1, 3, 7]. 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 idea is to characterize the power 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, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. ESORICS 2008*, pages 333–348, October 2008.
2. 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.
3. A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.
4. Aslan Askarov and Andrei 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.
5. 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.
6. A. Banerjee, D. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. IEEE Symp. on Security and Privacy*, pages 339–353, May 2008.
7. 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.
8. 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.
9. 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.
10. 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.
11. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.
12. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.
13. A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. Computer Security*, 14(2):157–196, May 2006.
14. Andrew C. Myers and Barbara 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.
15. Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow. Software release, `http://www.cs.cornell.edu/jif`, July 2006.
16. P. Ørbæk and J. Palsberg. Trust in the λ-calculus. *J. Functional Programming*, 7(6):557–591, 1997.
17. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
18. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, 2009. To appear.
19. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.
20. Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.
21. Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.