Enforcing Robust Declassification and Qualified Robustness

Andrew C. Myers
Department of Computer Science
Cornell University
andru@cs.cornell.edu

Andrei Sabelfeld[*]
Department of Computer Science
Chalmers University of Technology
andrei@cs.chalmers.se

Steve Zdancewic
Department of Computer and Information Science
University of Pennsylvania
stevez@cis.upenn.edu

Abstract

Noninterference requires that there is no information flow from sensitive to public data in a given system. However, many systems release sensitive information as part of their intended function and therefore violate noninterference. To control information flow while permitting information release, some systems have a downgrading or declassification mechanism, but this creates the danger that it may cause unintentional information release. This paper shows that a robustness property can be used to characterize programs in which declassification mechanisms cannot be controlled by attackers to release more information than intended. It describes a simple way to provably enforce this robustness property through a type-based compile-time program analysis. The paper also presents a generalization of robustness that supports upgrading (endorsing) data integrity.

1 Introduction

Information flow controls have many appealing properties as a security enforcement mechanism. Unlike access controls, they track the propagation of information and prevent sensitive information from being released, regardless of how that information is transformed by the system. Dually, information flow controls may be used to enforce

[*]This work was partly done while the author was at Cornell University.

data integrity. One common formal underpinning of these mechanisms is the *noninterference* security property [19], which imposes an end-to-end requirement on the behavior of the system: sensitive data may not affect public data. Recent work on language-based enforcement of confidentiality (e.g., [48, 1, 20, 44, 46, 2, 41, 35, 42, 52, 3, 36]) have used various flavors of noninterference as the definition of security. However, in practice noninterference is too strong; real systems leak some amount of sensitive information as part of their proper functioning.

One way to accommodate information release is to allow explicit declassification or downgrading of sensitive information (e.g., [17, 31, 35, 8]). These mechanisms are inherently unsafe and create the possibility that a downgrading channel may release information in a way that was not intended by the system designer.

Given that noninterference is not satisfied, we would like to know that the information release occurs as intended, in accordance with some kind of security policy. However, it seems difficult in general to express these policies precisely and even more difficult to show that systems satisfy them [43]. Therefore a reasonable strategy is instead to identify and enforce important *aspects* of the intended security policy rather than trying to express and enforce the entire policy.

A recent example of this approach is *robust declassification*, a security property introduced by Zdancewic and Myers [51] in the context of a state transition system. Robustness addresses an important issue for security analysis: the possibility that an attacker can affect some part of the system. If a system contains declassification, it is possible that an attacker can cause the declassification mechanism to release more information than was intended. A system whose declassification is robust may release information, but it gives attackers no control over what information is released or whether information is released. Although this paper is about robust declassification, robustness has been explored for other aspects of information security, such as information erasure [7].

Robustness is an important property for systems that may be attacked; for example, it is particularly important for distributed systems containing untrusted host machines, but is also useful for systems that simply process some untrusted input that might affect later information release. However, robustness is not intended to be a complete answer to the question of whether information release is secure. Additional useful ways to reason about information release include delimited information release [40], intransitive noninterference [38, 34, 37, 28], quantitative information flow [14, 30, 27, 9, 10], and complexity-bounded information flow [47, 22, 23]. However, among these only robustness directly addresses the important problem of attackers who exploit declassification.

This paper generalizes previous work on robust declassification in several ways. First, the paper shows how to express it in a language-based setting: specifically, for a simple imperative programming language. Second, it generalizes robust declassification so that untrusted code and data are explicitly part of the system rather than having them appear only when there is an *active* attack. Third, it introduces a relaxed security guarantee called *qualified robustness* that gives untrusted code and data a limited ability to affect information release.

The key technical result of the paper is a demonstration that both robustness and qualified robustness can be enforced by compile-time program analyses based on simple security type systems. Using lightweight annotations, these type systems track data confidentiality and integrity, in a manner similar to earlier work by Zdancewic [50]. However, this paper takes the new step of proving that all well-typed programs satisfy robustness.

The rest of the paper is structured as follows. Section 2 introduces robustness informally and shows how it adds value as a method for reasoning about information release in some simple program fragments. Section 3 presents basic assumptions and models used for this work, including a simple imperative language with an explicit declassification construct that downgrades confidentiality levels. Section 4 presents a generalized robustness condition in this language-based setting. Section 5 presents a security type system that enforces robustness in the imperative language. Section 6 presents more detailed examples and shows how the robust declassification condition gives insight into program security. Section 7 generalizes the robust declassification condition to allow untrusted code limited control over information release, and shows that useful code examples satisfy this limited robustness property. Section 8 discusses related work, and Section 9 concludes.

2 Robustness

The essential idea of robustness is that although systems may release some information, attackers should not be able to affect what information is released or whether information is released. Regardless of what attack is launched against the system, the same information should be released. This implies that a *passive* attacker who merely observes system execution learns no more than an *active* attacker who both changes and observes system execution. In a system that has robust declassification, the problem of understanding whether all information release is intentional in the presence of the attacker is reduced to the problem of understanding whether information flows are as intended when the attacker does nothing. It is not necessary to reason about all possible attacks.

Consider the following simple program, which releases information from a secret location z_H to a publicly readable variable y_L if the boolean variable x_L is *true*, and otherwise has no effect:

$$\begin{array}{l}\texttt{if}\ x_L\ \texttt{then}\ y_L := \texttt{declassify}(z_H)\\ \qquad\quad \texttt{else}\ \texttt{skip}\end{array}$$

The subscripts H and L are used to indicate secret (“high”) and public (“low”) information respectively. The special operator `declassify` is simply an identity function that explicitly releases information from high to low.

Clearly this program is intended to release information and therefore violates noninterference. Although noninterference is violated, robustness allows us to gain more understanding of the security of this program. Suppose that the attacker is able to change the value of x_L. In that case, the attacker is able to affect whether information is released; the declassification is not robust. Suppose instead that the attacker can affect the value of z_H, perhaps by causing some other secret information to be copied into it. In that case the attacker can cause different information to be released than was intended. Conversely, if the attacker is unable to affect the values of these two variables, the declassification is robust because these are the only variables that affect what information is released from high to low.

It is clear from this example that the robustness of a system is contingent on the power an active attacker has to affect the execution of the system. A natural way to describe this power is by ascribing *integrity* labels to the information manipulated. High-integrity information and code are trusted and assumed not to be affected by the attacker; all low-integrity information and code are untrusted and assumed to be under the control of the attacker. Low-integrity variables may be changed by the attacker; low-integrity code may be replaced by the attacker with different code.

This is a very general model of the system. The attacker may in fact be an ordinary user, in which case robustness means that program users cannot cause unintended information release, perhaps by providing unexpected inputs. Alternatively, as in the work on secure program partitioning [53, 54], the system might be a distributed program in which some program code runs on untrusted hosts and is assumed to be controlled by a malicious attacker who will try to exploit declassification. (In fact, robustness was inspired by the work on secure program partitioning.)

A recent survey on declassification [43] categorizes approaches to information release by *what* information is declassified, by *whom* it is released, and *where* and *when* in the system declassification occurs. We view robustness as operating primarily on the *who* dimension. Robustness controls on *whose* behalf declassification occurs because it prevents untrusted attackers from affecting declassification. The security type system we suggest in Section 5 enforces these restrictions by controlling *where* information release can occur, which helps express which part of the system can release *what* information and on *whose* behalf. This aspect is provided by localizing information release to declassification statements, whose occurrence is regulated by the type system.

Other mechanisms that control information release, such as the decentralized label model [32], also control who can declassify information, using an access control mechanism. Access control mechanisms can prevent the attacker from declassifying information directly, but robustness is a stronger notion of security because it prevents the attacker even from *influencing* declassification.

Requiring that the attacker cannot affect information release turns out to be too restrictive for many systems of interest; some systems satisfy their security requirements yet do not have robust declassification. Section 7 explores a relaxation of robust declassification that addresses this restrictiveness. An *endorsement* operation is added that explicitly says untrusted data may be treated as trusted, *upgrading* its integrity level. This leads to a relaxation of the robustness property that makes it less restrictive.

3 Language and attacker model

3.1 Security lattice

We assume that the security levels form a *security lattice* $\mathcal{L}$. The ordering specifies the relationship between different security levels. To enable reasoning about both confidentiality and integrity, the security lattice $\mathcal{L}$ is a product of *confidentiality* and *integrity*

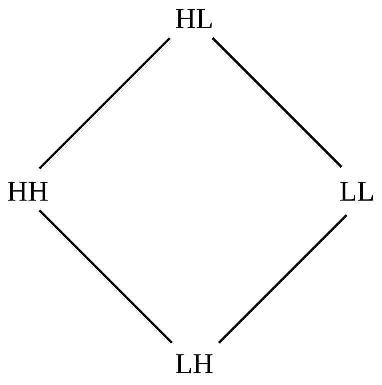

Figure 1: Security lattice $\mathcal{L}_{LH}$.

lattices, $\mathcal{L}_C$ and $\mathcal{L}_I$, with orderings $\sqsubseteq_C$ and $\sqsubseteq_I$, respectively. Suppose x and y are both elements in the confidentiality lattice (or both in the integrity lattice). If $x \sqsubseteq y$ then data at level x is no more confidential (no less trustworthy) than data at level y. An element ℓ of the product lattice is a pair $(C(\ell), I(\ell))$ (which we sometimes write as $C(\ell)I(\ell)$ for brevity), denoting the confidentiality part of ℓ by $C(\ell)$ and the integrity part by $I(\ell)$. The ordering on $\mathcal{L}$, $\mathcal{L}_C$, and $\mathcal{L}_I$ corresponds to the restrictions on how data at a given security level can be used. The use of high-confidentiality data is more restricted than that of low-confidentiality data, which helps prevent information leaks. Dually, the use of low-integrity data is more restricted than that of high-integrity data, which helps prevent information corruption.

An example $\mathcal{L}_{LH}$ of a security lattice is displayed in Figure 1. This lattice is a product of a simple confidentiality lattice (with elements L and H of low and high confidentiality so that $L \sqsubseteq_C H$) and a dual integrity lattice (with elements L and H of low and high integrity so that $H \sqsubseteq_I L$). At the bottom of the lattice is the level LH for data that may be used arbitrarily. This data has the lowest confidentiality and highest integrity level. At the top of the lattice is the data that is most restrictive in usage. This data has the highest confidentiality and lowest integrity level.

3.2 Attacker model

In all these scenarios, the power of the attacker is described by a lattice element A, where the confidentiality level $C(A)$ is the confidentiality of data the attacker is expected to be able to read, and the integrity level $I(A)$ is the integrity of data or code that the attacker is expected to be able to affect. Thus, the robustness of a system is with respect to the attacker parameters $(C(A), I(A))$. As far as a given attacker is concerned, the four-point lattice $\mathcal{L}_{LH}$ captures the relevant features of the general lattice $\mathcal{L}$. Let us define high- and low-confidentiality areas of $\mathcal{L}$ by $H_C = \{\ell \mid C(\ell) \not\sqsubseteq C(A)\}$ and $L_C = \{\ell \mid C(\ell) \sqsubseteq C(A)\}$, respectively. Similarly, we define low- and high-integrity areas by $L_I = \{\ell \mid I(A) \sqsubseteq I(\ell)\}$ and $H_I = \{\ell \mid I(A) \not\sqsubseteq I(\ell)\}$, respectively. The four

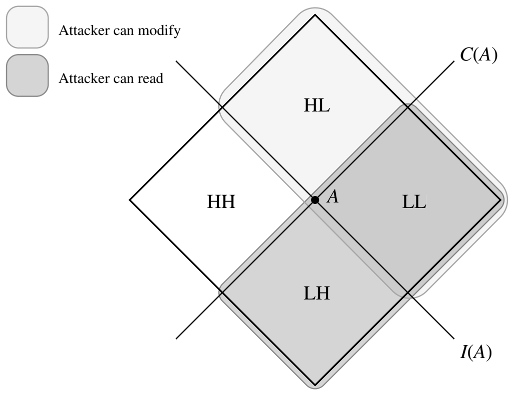

Figure 2: Attacker's view of a general lattice.

key areas of lattice $\mathcal{L}$ correspond exactly to the four points of lattice $\mathcal{L}_{LH}$:

$$LH \sim L_C \cap H_I \qquad HH \sim H_C \cap H_I$$
$$LL \sim L_C \cap L_I \qquad HL \sim H_C \cap L_I$$

This correspondence is illustrated in Figure 2. From the attacker's point of view, area LH describes data that is visible but cannot be modified; area HH describes data that is not visible and cannot be modified; area LL describes data that is both visible and can be modified; and, finally, area HL describes data that is not visible but can be modified by the attacker. Because of this correspondence between $\mathcal{L}_{LH}$ and $\mathcal{L}$, results obtained for the lattice $\mathcal{L}_{LH}$ generalize naturally to the full lattice $\mathcal{L}$.

3.3 Language

This paper uses a simple sequential language consisting of expressions and commands. It is similar to several other security-typed imperative languages (e.g., [48, 2]), and its semantics are largely standard (cf. [49]).

Definition 1. *The language syntax is defined by the following grammar:*

$$e ::= \mathit{val} \mid v \mid e_1 \;\texttt{op}\; e_2 \mid \texttt{declassify}(e, \ell)$$
$$\begin{aligned} c ::= {} & \texttt{skip} \mid v := e \mid c_1; c_2 \\ & \mid \texttt{if}\; e \;\texttt{then}\; c_1 \;\texttt{else}\; c_2 \mid \texttt{while}\; e \;\texttt{do}\; c \end{aligned}$$

where val *ranges over values* $\mathit{Val} = \{\mathit{false}, \mathit{true}, 0, 1, \dots\}$*,* v *ranges over variables* Var*,* op *ranges over arithmetic and boolean operations on expressions, and* ℓ *ranges over the* security levels*.*

The *security environment* $\Gamma : \mathit{Var} \to \mathcal{L}$ describes the type of each program variable as a security level. The security lattice and security environment together constitute a

$$\begin{array}{rcll}
\langle M, v := e\rangle & \xrightarrow{v} & \langle M[v \mapsto M(e)], \mathtt{skip}\rangle & \\
\langle M, \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\rangle & \xrightarrow{\cdot} & \langle M, c_1\rangle & (\text{if} M(b) = \textit{true}) \\
\langle M, \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\rangle & \xrightarrow{\cdot} & \langle M, c_2\rangle & (\text{if} M(b) = \textit{false}) \\
\langle M, \mathtt{while}\ b\ \mathtt{do}\ c\rangle & \xrightarrow{\cdot} & \langle M, c; \mathtt{while}\ b\ \mathtt{do}\ c\rangle & (\text{if} M(b) = \textit{true}) \\
\langle M, \mathtt{while}\ b\ \mathtt{do}\ c\rangle & \xrightarrow{\cdot} & \langle M, \mathtt{skip}\rangle & (\text{if} M(b) = \textit{false}) \\
\langle M, \mathtt{skip}; c\rangle & \xrightarrow{\cdot} & \langle M, c\rangle & \\
\langle M, c_1; c_2\rangle & \xrightarrow{\alpha} & \langle M', c_1'; c_2\rangle & (\text{if} \langle M, c_1\rangle \xrightarrow{\alpha} \langle M', c_1'\rangle)
\end{array}$$

Figure 3: Operational semantics.

security policy, specifying that information flow from a variable v_1 to a variable v_2 is allowed only if $\Gamma(v_1) \sqsubseteq \Gamma(v_2)$. This means that the confidentiality (integrity) of v_2 must be at least (at most) as high as that of v_1. In the rest of this paper, we assume that we are given a fixed, arbitrary security environment, Γ, which is an implicit parameter of the concepts defined below.

The only non-standard language expression is the construct $\mathtt{declassify}(e, \ell)$, which *declassifies* the security level of the expression e (which is simply the join of the security levels of the variables used in e) to the level $\ell \in \mathcal{L}$. Operationally, the result of $\mathtt{declassify}(e, \ell)$ is the same as that of e regardless of ℓ. The intention is that declassification is used for controlling the security level of information without affecting the execution of the program.

The evaluation semantics are defined in terms of small-step transitions between configurations. A configuration $\langle M, c\rangle$ consists of a memory M (which is a finite mapping $M : \mathit{Var} \to \mathit{Val}$ from variables to values) and a command c. A transition from configuration $\langle M, c\rangle$ to configuration $\langle M', c'\rangle$ is denoted by $\langle M, c\rangle \xrightarrow{\alpha} \langle M', c'\rangle$. Configurations of the form $\langle M, \mathtt{skip}\rangle$ are *terminal*. The complete operational semantics are shown in Figure 3, where we write $M(e)$ to denote the result of evaluating expression e in memory M. We assume that operations used in expressions are total—expressions always terminate (while command configurations might diverge).

In the operational semantics, α is an *event* that is either $\cdot$, indicating that no assignment has taken place during this transition step, or a variable v, indicating that the variable has been updated. We extend the label ordering to events by defining $\ell \sqsubseteq \alpha$ if $\alpha = \cdot$ or $\alpha = v$ and $\ell \sqsubseteq \Gamma(v)$. Similarly, $\alpha \sqsubseteq \ell$ iff $\alpha = v$ and $\Gamma(v) \sqsubseteq \ell$.

The *trace* $\mathit{Tr}(\langle M, c\rangle)$ of the execution of configuration $\langle M, c\rangle$ is the sequence

$$\langle M, c\rangle \xrightarrow{\alpha_0} \langle M_1, c_1\rangle \xrightarrow{\alpha_1} \langle M_2, c_2\rangle \xrightarrow{\alpha_2} \langle M_3, c_3\rangle \ldots$$

We write $\longrightarrow$ for the relation obtained by erasing the event annotation from $\xrightarrow{\alpha}$. As usual, $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$. Configuration $\langle M, c\rangle$ *terminates* in M' if $\langle M, c\rangle \longrightarrow^* \langle M', \mathtt{skip}\rangle$, which is denoted by $\langle M, c\rangle \Downarrow M'$ or, simply, $\langle M, c\rangle \Downarrow$ when M' is unimportant. If there is an infinitely long sequence of transitions from the initial configuration $\langle M, c\rangle$ then that configuration *diverges*.

4 Formalizing robustness

Let us define the view of the memory at level ℓ. The idea is that the observer at level ℓ may only distinguish data whose security level is at or below ℓ. Formally, memories M_1 and M_2 are indistinguishable at a level ℓ (written $M_1 =_\ell M_2$) if $\forall v.\, \Gamma(v) \sqsubseteq \ell \implies M_1(v) = M_2(v)$. The *restriction* $M|_\ell$ of memory M to the security level ℓ is defined by restricting the mapping to variables whose security level is at or below ℓ. In the following, we make use of this simple proposition:

Proposition 1. *If $\langle M, c\rangle \xrightarrow{\alpha} \langle M', c'\rangle$ and $\alpha \not\sqsubseteq \ell$ then $M =_\ell M'$.*

Proof. By induction on the structure of c. The only interesting cases are for assignment and sequencing. For assignment, $v := e$, the result follows from the observation that $\Gamma(v) \not\sqsubseteq \ell$ and the definition of $=_\ell$, which implies that $M =_\ell M[v \mapsto M(e)]$. The case for sequential composition follows immediately from the inductive hypothesis. $\square$

Because computation steps can be observed at level ℓ only if they update variables of level ℓ or below, and because we are not concerned with timing channels, we identify traces up to *high-stuttering* with respect to a security level ℓ.

Definition 2. *The ℓ-*projection *of the trace t, written $t|_\ell$, where*

$$t = \langle M_0, c_0\rangle \xrightarrow{\alpha_0} \langle M_1, c_1\rangle \xrightarrow{\alpha_1} \dots \langle M_i, c_i\rangle \xrightarrow{\alpha_i} \dots$$

is the sequence of (ℓ-restrictions of) memories

$$m = M_0|_\ell, M_{i_1}|_\ell, M_{i_2}|_\ell, \dots$$

Such that $0 < i_1 < i_2 < \dots$ and for every transition $\langle M_j, c_j\rangle \xrightarrow{\alpha_j} \langle M_{j+1}, c_{j+1}\rangle$ in t, if $\alpha_j \sqsubseteq \ell$ then $j + 1 = i_k$ for some k, and for every i_k in m, there is a j such that $j + 1 = i_k$.

Let p_1 and p_2 be trace projections. Their concatenation $p_1 \cdot p_2$ must take into account divergence and stuttering. If p_1 is infinite, then $p_1 \cdot p_2 = p_1$. Otherwise, we have $p_1 = M_1|_\ell, M_2|_\ell, \dots, M_n|_\ell$ and $p_2 = M_{n+1}|_\ell, M_{n+2}|_\ell, \dots$ If $M_n|_\ell = M_{n+1}|_\ell$ then

$$p_1 \cdot p_2 = M_1|_\ell, M_2|_\ell, \dots, M_n|_\ell, M_{n+2}|_\ell, \dots$$

otherwise

$$p_1 \cdot p_2 = M_1|_\ell, M_2|_\ell, \dots, M_n|_\ell, M_{n+1}|_\ell, M_{n+2}|_\ell, \dots$$

We need a simple proposition that relates sequential composition of commands with the projections of their traces:

Proposition 2. *If $\langle M, c_1\rangle \Downarrow M'$ then for all ℓ and c_2:*

$$Tr(\langle M, c_1; c_2\rangle)|_\ell = (Tr(\langle M, c_1\rangle)|_\ell) \cdot (Tr(\langle M', c_2\rangle|_\ell)$$

Proof. Straightforward from the definitions of the operational semantics and ℓ-projection of traces. $\square$

Definition 3. *Given traces t_1 and t_2 and a security level ℓ, we say that t_1 and t_2 are* high-stutter equivalent up to ℓ, *written $t_1 \sim_\ell t_2$ if $t_1|_\ell = t_2|_\ell$.*

Intuitively, $t_1 \sim_\ell t_2$ means that t_1 and t_2 look the same to an observer at level ℓ after all successive transitions involving memories related by $=_\ell$ have been collapsed together, provided that the transition between them did not arise from an update. For example, consider $t_1 = Tr(\langle M_1, l := h\rangle)$ and $t_2 = Tr(\langle M_2, l := h\rangle)$ where $\Gamma(l) = \ell$, $\Gamma(h) \not\sqsubseteq \ell$, $\forall x \in \mathit{Var}.\, M_1(x) = 0$, and $M_2 = M_1[h \mapsto 1]$. Although $M_1 =_\ell M_2$, we have $Tr(\langle M_1, l := h\rangle) \not\sim_\ell Tr(\langle M_2, l := h\rangle)$ because the effect of the assignment $l := h$ is distinguishable at the level ℓ:

$$Tr(\langle M_1, l := h\rangle)|_\ell \;=\; (M_1|_\ell, M_1|_\ell) \;\neq\; (M_2|_\ell, M_2[l \mapsto 1]|_\ell) \;=\; Tr(\langle M_2, l := h\rangle)|_\ell$$

On the other hand, if

$$c = (\texttt{if}\ h = 0\ \texttt{then}\ h := h + l\ \texttt{else skip}); l := 2$$

then we have $Tr(\langle M_1, c\rangle) \sim_\ell Tr(\langle M_2, c\rangle)$: because $M_1 =_\ell M_2$, and the assignment to l updates the low parts of the memories with the same value:

$$Tr(\langle M_1, c\rangle)|_\ell \;=\; (M_1|_\ell, M_1[l \mapsto 2]|_\ell) \quad = \quad (M_2|_\ell, M_2[l \mapsto 2]|_\ell) \;=\; Tr(\langle M_2, c\rangle)|_\ell$$

Definition 4. *Two traces t_1 and t_2 are* indistinguishable up to ℓ, *written $t_1 \approx_\ell t_2$, if whenever both t_1 and t_2 terminate then $t_1 \sim_\ell t_2$.*

We lift indistinguishability from memories and traces to configurations by the following definition:

Definition 5. *Two configurations $\langle M_1, c_1\rangle$ and $\langle M_2, c_2\rangle$ are* weakly indistinguishable up to ℓ *(written $\langle M_1, c_1\rangle \approx_\ell \langle M_2, c_2\rangle$) if $Tr(\langle M_1, c_1\rangle) \approx_\ell Tr(\langle M_2, c_2\rangle)$. We say that two configurations are* strongly indistinguishable up to ℓ *(written $\langle M_1, c_1\rangle \approxeq_\ell \langle M_2, c_2\rangle$) if $\langle M_1, c_1\rangle \Downarrow$, $\langle M_2, c_2\rangle \Downarrow$, and $\langle M_1, c_1\rangle \approx_\ell \langle M_2, c_2\rangle$.*

Note that weak indistinguishability is timing- and termination-insensitive because it deems a diverging trace indistinguishable from any other trace; although strong indistinguishability is timing-insensitive, it requires the termination of both configurations so that the traces remain related throughout their entire execution. Because of the termination insensitivity, weak indistinguishability is not transitive, but transitivity is not needed in the subsequent development. (Note that the transitivity of strong indistinguishability follows from the transitivity of equality on values.)

For example, configurations $\langle M_1, c\rangle$ and $\langle M_2, c\rangle$ so that $\Gamma(l) = \ell$, $\Gamma(h) \not\sqsubseteq \ell$, $\forall x \in \mathit{Var}.\, M_1(x) = 0$, $M_2 = M_1[h \mapsto 1]$, and $c = (\texttt{while}\ h \neq 0\ \texttt{do}\ h := h + 1); l := 2$ are weakly indistinguishable up to ℓ ($\langle M_1, c\rangle \approx_\ell \langle M_2, c\rangle$) because $\langle M_2, c\rangle$ diverges. For the same reason, these configurations are not strongly indistinguishable up to ℓ ($\langle M_1, c\rangle \not\approxeq_\ell \langle M_2, c\rangle$). Strong indistinguishability up to ℓ holds if instead of the memory M_2 we take $M_3 = M_1[h \mapsto -1]$. Clearly, we have both $\langle M_1, c\rangle \approx_\ell \langle M_3, c\rangle$ and $\langle M_1, c\rangle \approxeq_\ell \langle M_3, c\rangle$. Consider $d = (\texttt{while}\ h \neq 0\ \texttt{do}\ h := h + 1); l := h$. Despite the fact that $\langle M_1, d\rangle \approx_\ell \langle M_2, d\rangle$ and $\langle M_2, d\rangle \approx_\ell \langle M_3, d\rangle$ (thanks to the divergence of$\langle M_2, d\rangle$), we have $\langle M_1, d\rangle \not\approx_\ell \langle M_3, d\rangle$, which illustrates the intransitivity of weak indistinguishability,

Noninterference says that if two memories are indistinguishable at a certain level, then the executions of a given program on these two memories are also (at least weakly) indistinguishable at that level:

Definition 6 (Noninterference). *A command c satisfies* noninterference *if*

$$\forall \ell, M_1, M_2.\, M_1 =_\ell M_2 \Longrightarrow \langle M_1, c\rangle \approx_\ell \langle M_2, c\rangle$$

We assume that the attacker has the ability to read and write some data manipulated by the program. Recall that robustness means an active attacker, who both observes and modifies part of the system state, should not learn more sensitive information than a passive attacker, who merely observes visible data. The power of the attacker to observe and modify system state can be described by a single point A in a security lattice. A passive A-attacker may read data at or below $C(A)$ (i.e., at or below $(C(A), \top_I)$ in the product lattice); an active A-attacker may also modify data at or above $I(A)$ (i.e., at or above $(\bot_C, I(A))$ in the product lattice).

In general, an active attacker may change system behavior by injecting new code into the program. However, accesses by the attacker must satisfy certain conditions on what data can be read and modified. Code satisfying these conditions is considered a *fair attack* because the code should not be able to violate confidentiality and integrity directly; rather, a fair attack attempts exploiting insecure information flow in the trusted code.

Definition 7. *A command a is a* fair attack *(at some level $\ell \in LL$) if it is formed according to the following grammar, where expressions e and b do not contain* `declassify`*:*

$$\begin{array}{lcll}
a & ::= & \texttt{skip} & \\
 & | & v := e & (\forall x \in \mathit{Vars}(e).\, \Gamma(x) = \ell = \Gamma(v)) \\
 & | & a_1; a_2 & \\
 & | & \texttt{if}\ b\ \texttt{then}\ a_1\ \texttt{else}\ a_2 & (\forall x \in \mathit{Vars}(b).\, \Gamma(x) = \ell) \\
 & | & \texttt{while}\ b\ \texttt{do}\ a & (\forall x \in \mathit{Vars}(b).\, \Gamma(x) = \ell)
\end{array}$$

For simplicity, each attack operates at a single security level ℓ. This does not restrict the attacker because the attacker is already assumed to completely control the LL region of the lattice (recall from Section 3 that the LL region is relative to the attacker's point A in the lattice), and ℓ is an adequate representative of this region. Anything that the attacker can read has lower confidentiality than ℓ, and anything the attacker can write to has lower integrity. So if attacker code used other levels in LL, the code could be translated to code that only used ℓ. In fact, our results are proved for a strictly more general class of attacks that we call A-attacks (defined in Section 5), which subsume fair attacks and may operate across different security levels.

Attacker-controlled low-integrity computation may be interspersed with trusted high-integrity code. To distinguish the two, the high-integrity code is represented as a program in which some statements are missing, replaced by holes ($\bullet$). The idea is that the holes are places where the attacker can insert arbitrary low-integrity code. There may be multiple holes in the high-integrity code, represented by the notation $\vec{\bullet}$. The high-integrity computation is then a *high-integrity context* $c[\vec{\bullet}]$ in which the holes can be replaced by a vector of attacker code fragments $\vec{a}$ to obtain a complete program $c[\vec{a}]$. An attacker is thus a vector of such code fragments. A passive attacker is an attack vector that fills all holes with the low-integrity code from the original program. An active attacker is any attack vector that fills some hole in a way that changes the original program behavior.

Although the assumption that attackers are constrained to interpolating sequential code may seem artificial, it is a reasonable assumption to make both in a single-machine setting, where the attacker's code can be statically checked before it is run, and in a distributed setting where the attacker has complete power to change the untrusted code, but where that code is limited in its ability to affect the machines on which trusted code is run [53].

High-integrity contexts are defined formally as follows:

Definition 8. *High-integrity contexts, or commands with holes,* $c[\vec{\bullet}]$ *are defined by extending the command grammar from Definition 1 with:*

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

Using this definition, robust declassification can be translated into the language-based setting. Robust declassification holds if for all $\vec{a}$, whenever program $c[\vec{a}]$ cannot distinguish the behaviors of the program on some memories, then any change of the attacker's code to any other attack $\vec{a'}$ still cannot distinguish the behaviors of the program on these memories. In other words, the attacker's observations about $c[\vec{a'}]$ may not reveal any secrets apart from what the attacker already knows from observations about $c[\vec{a}]$. This is formally expressed in the following definition (where we assume that F is the set of fair-attack vectors).

Definition 9 (Robustness). *Command* $c[\vec{\bullet}]$ *has* robustness *with respect to fair attacks at level* A *if*

$$\forall M_1, M_2, \vec{a} \in F, \vec{a'} \in F.\, \langle M_1, c[\vec{a}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c[\vec{a}]\rangle \implies$$
$$\langle M_1, c[\vec{a'}]\rangle \approx_{(C(A),\top_I)} \langle M_2, c[\vec{a'}]\rangle$$

As noted, the attacker can observe data below the lattice point $(C(A), \top_I)$. This level is used in the relations $\approxeq_{(C(A),\top_I)}$ and $\approx_{(C(A),\top_I)}$, requiring equality for assignments to low-confidentiality variables. Observe that $\langle M_1, c\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c\rangle$ implies that $M_1 =_{(C(A),\top_I)} M_2$ by Definition 5. Note that attacks in the vectors $\vec{a}$ and $\vec{a'}$ may operate at different levels (as long as these levels are in the LL region).

The definition of robustness uses both strong and weak indistinguishability, which is needed to deal properly with nontermination. Because we are ignoring timing and termination channels, information is only really leaked if configurations are not weakly indistinguishable. However, the premise of the condition is based on strong indistinguishability because a sufficiently incompetent attacker may insert nonterminating code and thus make fewer observations than even a passive attacker. We are not concerned with such attackers.

Note that the robustness definition quantifies over both passive and active attacks. This is because neither passive or active attacker behavior is known a priori. The vector of `skip` commands is an example of a possible attack. Importantly, the robustness definition also guards against other attacks (which might affect what critical fragments of the target program are reachable). For example, under lattice $\mathcal{L}_{LH}$ and attacker at LL, consider the following program (here and in the rest of the paper the subscript of a variable indicates its security level in Γ):

$$\begin{array}{l} x_{LL} := 1; [\bullet]; \texttt{while}\ x_{LL} > 0\ \texttt{do skip};\\ \texttt{if}\ x_{LL} = 0\ \texttt{then}\ y_{LH} := \texttt{declassify}(z_{HH}, LH)\\ \qquad\qquad\qquad \texttt{else skip} \end{array}$$

This program would be robust if a in Definition 9 were fixed to be the `skip` command (as $c[a]$ would always diverge). However, the attacker may tamper with the declassification mechanism in the program because whether declassification code is reachable depends on the attacker-controlled variable x_{LL}. This is indeed captured by Definition 9, which deems the program as non-robust (take $a = x_{LL} := -1$ and $a' = x_{LL} := 0$).

The robustness definition ensures that the attacker's actions cannot lead the declassification mechanism to increase the attacker's observations about secrets. Note that robustness is really a property of a high-integrity program context rather than of an entire program. A full program $c[\vec{a}]$ is robust if its high-integrity part $c[\vec{\bullet}]$ is itself robust. Because the low-integrity code $\vec{a}$ is assumed to be under the control of the attacker, the security property is insensitive to it.

For example, under lattice $\mathcal{L}_{LH}$ and attacker at LL, consider programs:

$$[\bullet]; x_{LH} := \texttt{declassify}(y_{HH}, LH)$$

and

$$\begin{array}{l} [\bullet];\ \texttt{if}\ x_{LH}\ \texttt{then}\ y_{LH} := \texttt{declassify}(z_{HH}, LH)\\ \qquad\qquad\qquad \texttt{else skip} \end{array}$$

No matter what (terminating attack) fills the hole, these programs are rejected by noninterference although their declassification operations are intended. On the other hand, these programs satisfy robustness because the attacker may not either influence what is declassified (by assigning to y_{HH} in the former program) or manipulate the control flow leading to declassification (by assigning to x_{LH} in the latter program). Indeed, no fair attack filling the hole may assign to either y_{HH} or x_{LH}. Note that the latter program is similar to the example from Section 2, except with integrity annotations. However, a similar program where x is low-integrity is properly rejected:

$$\begin{array}{l} [\bullet];\ \texttt{if}\ x_{LL}\ \texttt{then}\ y_{LL} := \texttt{declassify}(z_{HH}, LH)\\ \qquad\qquad\qquad \texttt{else skip} \end{array}$$

To see a successful attack, take $a = x_{LL} := 0$ and $a' = x_{LL} := 1$ and memories different in z_{HH}.

5 Security type system for robustness

$$\frac{}{\Gamma \vdash \mathit{val} : \ell} \qquad \frac{\Gamma(v) = \ell}{\Gamma \vdash v : \ell}$$

$$\frac{\Gamma \vdash e : \ell \quad \Gamma \vdash e' : \ell}{\Gamma \vdash e \ \texttt{op}\ e' : \ell} \qquad \frac{\Gamma \vdash e : \ell \quad \ell \sqsubseteq \ell'}{\Gamma \vdash e : \ell'}$$

$$\frac{}{\Gamma, pc \vdash \texttt{skip}} \qquad \frac{\Gamma \vdash e : \ell \quad \ell \sqcup pc \sqsubseteq \Gamma(v)}{\Gamma, pc \vdash v := e}$$

$$\frac{\Gamma, pc \vdash c_1 \quad \Gamma, pc \vdash c_2}{\Gamma, pc \vdash c_1; c_2} \qquad \frac{\Gamma \vdash e : \ell \quad \Gamma, \ell \sqcup pc \vdash c_1 \quad \Gamma, \ell \sqcup pc \vdash c_2}{\Gamma, pc \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2}$$

$$\frac{\Gamma \vdash e : \ell \quad \Gamma, \ell \sqcup pc \vdash c}{\Gamma, pc \vdash \texttt{while}\ e\ \texttt{do}\ c} \qquad \frac{\Gamma, pc \vdash c \quad pc' \sqsubseteq pc}{\Gamma, pc' \vdash c}$$

$$\frac{\begin{array}{cc}\Gamma \vdash e : \ell' & \ell \sqcup pc \sqsubseteq \Gamma(v) \\ I(\ell) = I(\ell') & pc, \ell' \in H_I\end{array}}{\Gamma, pc \vdash v := \texttt{declassify}(e, \ell)}$$

Figure 4: Typing rules.

Figure 4 gives typing rules for the simple sequential language. These are security typing rules because they impose conditions on the security level components of types. As we show later in this section, any program that is well-typed according to these rules also satisfies the robustness property. Expressions and commands are typed with respect to a typing context that comprises both a security environment Γ and a program-counter security level pc. The program-counter security level tracks what information has affected control flow up to the current program point. For example, if pc is high-confidentiality at some program point, then an attacker might learn secrets from the fact that execution reached that point. If pc is low-integrity, the attacker might be able to affect whether control reaches that point.

We write $\Gamma \vdash e : \ell$ to mean that an expression e has type ℓ under an environment Γ and a security level pc. For commands, we write $\Gamma, pc \vdash c$ if command c is well-typed under Γ and pc.

The typing rules control the information flow due to assignments and control flow in a largely standard fashion (cf. [48]). However, the key rule governing uses of declassification is non-standard, though similar to that proposed by Zdancewic [50] (we discuss the relation at the end of this section). This rule states that only high-integrity data is allowed to be declassified and that declassification might only occur at a high-integrity program point (pc). The effect of this rule can be visualized by considering the lattice depicted in Figure 5. The figure includes an arrow corresponding to a declassification from security level ℓ to level ℓ'. If the area of possible flow origins (below ℓ) is

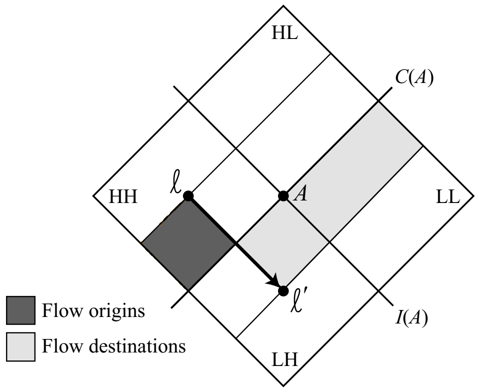

Figure 5: Effects of declassification.

within the high-integrity area of the lattice, the attacker (who can only affect the low-integrity area of the lattice) cannot influence uses of the declassification mechanism and therefore cannot exploit it.

Using the type system, we define *A-attacks*, programs controlled by the attacker at level A, which subsume fair attacks. We prove that well-typed programs are robust with respect to A-attacks (or simply "attacks" from here on) and therefore with respect to fair attacks.

Definition 10. *A command a is an A-*attack *if $\Gamma, (\bot_C, I(A)) \vdash a$ and* `declassify` *does not occur in a.*

Under lattice $\mathcal{L}_{LH}$ and $A = LL$, examples of attacks are programs `skip` (a harmless attack), $x_{LL} := y_{LL}$, and `while` x_{HL} `do skip`. On the other hand, programs $x_{HH} := y_{LH}$ and $x_{LH} :=$ `declassify`(y_{LH}, LH) are not attacks because they manipulate high-integrity data. Note that programs like $x_{LL} :=$ `declassify`(y_{HL}, LL) and `if` x_{LL} `then` $y_{LL}:=$`declassify`(z_{HH}, LH) `else skip` are not valid attacks because `declassify` may not be used in attacks. This is consistent with the discipline enforced by the type system that the attacker may not control declassification. All fair attacks are A-attacks, but the reverse is not true. For example, the program $x_{HL} := y_{HL}$ is an A-attack but is not a fair attack because it reads from a confidential variable. Recall the partition of data according to the confidentiality (H_C and L_C) and integrity (L_I and H_I) levels from Section 3.2. The following propositions provide some useful (and straightforward to prove) properties of attacks.

Proposition 3. *A fair attack a at level $\ell \in LL$ is also an A-attack.*

Proof. The proof is by induction on the structure of the attack a. The `declassify` expression cannot appear in a by construction, and to show that $\Gamma, (\bot_C, I(\ell)) \vdash a$, we rely on an auxiliary proposition for expressions that shows that if $\forall x \in \mathit{Vars}(e).\Gamma(x) = \ell$ then $\Gamma \vdash e : \ell$. □

Proposition 4. *An A-attack a (i) does not have occurrences of assignments to high-integrity variables (such v that $\Gamma(v) \in H_I$); and (ii) satisfies noninterference.*

Proof. Part (i) follows by induction on the derivation that $\Gamma, (\bot_C, I(A)) \vdash a$. To see why, observe that the pc label in the typing judgments only increases as the derivation moves from the root toward the leaves (this is seen in the typing rule for conditional statements in which the branches are checked under the higher label $\ell \sqcup pc$). Therefore, when type checking any assignment statement $v := e$, it must be the case that $(\bot_C, I(A)) \sqsubseteq pc$ and the typing rule for assignment implies that $pc \sqsubseteq \Gamma(v)$. By transitivity, it follows that $(\bot_C, I(A)) \sqsubseteq \Gamma(v)$, and we have $I(A) \sqsubseteq I(\Gamma(v))$. But from the definition of $H_I = \{\ell \mid I(A) \not\sqsubseteq I(\ell)\}$, so $\Gamma(v) \notin H_I$.

Part (ii) follows directly from Theorem 1 below. □

The type system can be used to enforce two interesting properties: noninterference (if `declassify` is not used) and robust declassification (even if it is).

Theorem 1. *If $\Gamma, pc \vdash c$ and* `declassify` *does not occur in c, then c satisfies noninterference.*

Because the declassification-free part of the security type system is largely standard, this result is proved by induction on the evaluation of c, similar to the proof of Volpano et al. [48]. Note that although the noninterference condition by Volpano et al. is weaker (the attacker can only observe the low part of the final state), their noninterference proof is based on a stronger condition. In particular, this condition happens to guarantee for executions that start in memories that agree on low data, that assignments to low variables affect the memories in the same way (see the UPDATE case in the proof of Theorem 6.8 of [48]). Hence, most of their proof can be simply reused for the present theorem. One simple proposition needed in the proof is the following:

Proposition 5. *If $\Gamma \vdash e : \ell$ where e contains no* `declassify` *expressions and $M_1 =_{\ell'} M_2$ and $\ell \sqsubseteq \ell'$ then $M_1(e) = M_2(e)$.*

Proof. By induction on the structure of e. The only interesting case is when e is a variable, but that follows immediately from the definition of memory equivalence at ℓ' and the assumption that $\ell \sqsubseteq \ell'$. □

The interesting question, however, is what the type system guarantees when declassification is used. Observing that declassification affects only confidentiality, we prove that the integrity part of the noninterference property is preserved in the presence of declassification:

Theorem 2. *If $\Gamma, pc \vdash c$ then for all integrity levels I we have*

$$\forall M_1, M_2.\, M_1 =_{(\top_C, I)} M_2 \Longrightarrow \langle M_1, c\rangle \approx_{(\top_C, I)} \langle M_2, c\rangle$$

Proof. This theorem follows essentially in the same way as Theorem 1, the only differing case is when the program contains a `declassify` expression, so it suffices to show that noninterference holds for the program $v :=$ `declassify`(e, ℓ).

Let $M_1 =_{(\top_C, I)} M_2$ be given. From the typing rule for `declassify`, we obtain that $\ell \sqcup pc \sqsubseteq \Gamma(v)$ and $\Gamma \vdash e : \ell'$ and $I(\ell) = I(\ell')$ and $\ell, \ell' \in H_I$. Also from the typing rules, we know that e itself contains no declassifications. From the definition of memory equivalence, it suffices to show that if $\Gamma(v) \sqsubseteq (\top_C, I)$ then $M_1(e) = M_2(e)$.

Observe that $\ell \sqsubseteq \Gamma(v)$ and, since $I(\ell) = I(\ell')$, we have $\ell' \sqsubseteq (\top_C, I)$ and result follows by Proposition 5.

$\square$

As for the confidentiality part, we show the key result of this paper: typable programs satisfy robustness and, thus, the attacker may not manipulate the declassification mechanism to leak more information than intended.

For robustness, it is important that holes not be placed at program points where the program counter pc is high-confidentiality, because then an attacker would be able to directly observe implicit flows [15] to the hole and could therefore cause information to leak even without a `declassify`. This restriction is achieved by defining a suitable typing rule for holes, permitting program contexts $c[\vec{\bullet}]$ to be type-checked:

$$\frac{pc \in L_C}{\Gamma, pc \vdash \bullet}$$

The main theorem of the paper can now be stated:

Theorem 3. *If $\Gamma, pc \vdash c[\vec{\bullet}]$ then $c[\vec{\bullet}]$ satisfies robustness.*

Before proving the theorem, we present a few helpful propositions. One such proposition says that if a sequential composition of well-typed commands may not distinguish two low-equivalent memories (through terminating execution), then the first command of the composition may not distinguish between the memories (which implies that it terminates in some low-equivalent intermediate memories). Further, the second command may not distinguish between these intermediate memories. This property is achieved because of the trace-level granularity of the security condition: the indistinguishability of configurations requires the indistinguishability of traces (up to high-stuttering).

Proposition 6. *If $\Gamma, pc \vdash c$ and $pc \not\sqsubseteq \ell$ then $Tr(\langle M, c\rangle)|_\ell = M|_\ell$.*

Proof. Observe that the typing rules require that all variables v assigned to in the body of c satisfy $pc \sqsubseteq \Gamma(v)$. Therefore $\Gamma(v) \not\sqsubseteq \ell$ and the result follows directly from the definition of ℓ-projection of traces. $\square$

Proposition 7. *Let $\ell = (C(A), \top_I)$. If $\Gamma, pc \vdash c_1; c_2$ and $\langle M_1, c_1; c_2\rangle \approxeq_\ell \langle M_2, c_1; c_2\rangle$ then $\langle M_1, c_1\rangle \approxeq_\ell \langle M_2, c_1\rangle$. Further, we have $\langle M_1, c_1\rangle \Downarrow N_1$ and $\langle M_2, c_1\rangle \Downarrow N_2$ for some N_1 and N_2 so that $\langle N_1, c_2\rangle \approxeq_\ell \langle N_2, c_2\rangle$.*

Proof. Let $t_1 = Tr(\langle M_1, c_1; c_2\rangle)$ and $t_2 = Tr(\langle M_2, c_1; c_2\rangle)$. We proceed by induction on the structure of c_1. We show the cases for `skip`, assignment, and sequential composition, and `if`. The loop case follows similarly to the case for conditionals.

- If $c_1 = \texttt{skip}$ then from the definition of the operational semantics we have that:

$$t_1 = \langle M_1, \texttt{skip}; c_2\rangle \overset{\cdot}{\longrightarrow} Tr(\langle M_1, c_2\rangle)$$
$$t_2 = \langle M_2, \texttt{skip}; c_2\rangle \overset{\cdot}{\longrightarrow} Tr(\langle M_2, c_2\rangle)$$

Clearly, $\mathit{Tr}(\langle M_1, \texttt{skip}\rangle) = \langle M_1, \texttt{skip}\rangle \approx_\ell \langle M_2, \texttt{skip}\rangle = \mathit{Tr}(\langle M_2, \texttt{skip}\rangle)$, since $M_1|_\ell = M_2|_\ell$. Moreover, $\langle M_1, \texttt{skip}\rangle \Downarrow M_1$ and $\langle M_2, \texttt{skip}\rangle \Downarrow M_2$ and, since $t_1|_\ell = \mathit{Tr}(\langle M_1, c_2\rangle)|_\ell$ and $t_2|_\ell = \mathit{Tr}(\langle M_2, c_2\rangle)|_\ell$, it follows that $\langle M_1, c_2\rangle \approxeq_\ell \langle M_2, c_2\rangle$.

- If $c_1 = v := e$, we let $M'_i = M_i[v \mapsto M_i(e)]$ (for $i \in \{1, 2\}$) and from the operational semantics we obtain that $\langle M_1, v := e\rangle \Downarrow M'_1$ and $\langle M_2, v := e\rangle \Downarrow M'_2$. We choose $N_1 = M'_1$ and $N_2 = M'_2$ observe that

$$t_1 = \langle M_1, v := e; c_2\rangle \xrightarrow{v} \mathit{Tr}(\langle M'_1, \texttt{skip}; c_2\rangle)$$
$$t_2 = \langle M_2, v := e; c_2\rangle \xrightarrow{v} \mathit{Tr}(\langle M'_2, \texttt{skip}; c_2\rangle)$$

If $\Gamma(v) \not\sqsubseteq \ell$, we have $\langle M_1, c_1\rangle \approxeq_\ell \langle M_2, c_1\rangle$ because

$$\mathit{Tr}(\langle M_1, v := e\rangle)|_\ell = M_1|_\ell = M_2|_\ell = \mathit{Tr}(\langle M_1, v := e\rangle)$$

Note that $M_i|_\ell = M'_i|_\ell$, and it follows that $\mathit{Tr}(\langle M'_1, c_2\rangle)|_\ell = t_1|_\ell = t_2|_\ell = \mathit{Tr}(\langle M'_2, c_2\rangle)|_\ell$. The other case is that $\Gamma(v) \sqsubseteq \ell$. This implies that the assignment step is ℓ-observable, so the ℓ projections must look like

$$t_1|_\ell = M_1|_\ell, M'_1|_\ell, \ldots = M_2|_\ell, M'_2|_\ell, \ldots = t_2|_\ell$$

Consequently, $\mathit{Tr}(\langle M_1, c_1\rangle)|_\ell = M_1|_\ell, M'_1|_\ell = M_2|_\ell, M'_2|_\ell = \mathit{Tr}(\langle M_2, c_1\rangle)$, and $\mathit{Tr}(\langle M'_1, c_2\rangle)|_\ell = \mathit{Tr}(\langle M'_2, c_2\rangle)|_\ell$ as required.

- If $c_1 = (d_1; d_2)$ then by inversion of the typing rule for sequences, we obtain $\Gamma, pc \vdash d_1$ and $\Gamma, pc \vdash d_2$. Therefore, $\Gamma, pc \vdash d_2; c_2$. Note that since $\mathit{Tr}(\langle M_1, (d_1; d_2); c_2\rangle) = \mathit{Tr}(\langle M_1, d_1; (d_2; c_2)\rangle)$ (and similarly for M_2) we must have $\langle M_1, d_1; (d_2; c_2)\rangle \approxeq_\ell \langle M_2, d_1; (d_2; c_2)\rangle$. One application of the induction hypothesis applied to the command $d_1; (d_2; c_2)$ yields the following: $\langle M_1, d_1\rangle \approxeq_\ell \langle M_2, d_1\rangle$ and there exist N'_1 and N'_2 such that $\langle M_1, d_1\rangle \Downarrow N'_1$ and $\langle M_2, d_1\rangle \Downarrow N'_2$ and, moreover, $\langle N'_1, d_2; c_2\rangle \approxeq_\ell \langle N'_2, d_2; c_2\rangle$. A second use of the induction hypothesis applied to the command $d_2; c_2$ yields that $\langle N'_1, d_2\rangle \approxeq_\ell \langle N'_2, d_2\rangle$ and there exist N_1 and N_2 such that $\langle N'_1, d_2\rangle \Downarrow N_1$ and $\langle N'_2, d_2\rangle \Downarrow N_2$ and, moreover, $\langle N_1, c_2\rangle \approxeq_\ell \langle N_2, c_2\rangle$. To complete the theorem, we observe that $\mathit{Tr}(\langle M_1, d_1; d_2\rangle) = \mathit{Tr}(\langle M_1, d_1\rangle) \xrightarrow{\cdot} \mathit{Tr}(\langle M_1, d_2\rangle)$ (and similarly for M_2). This yields that $\langle M_1, d_1; d_2\rangle \Downarrow N_1$ and $\langle M_2, d_1; d_2\rangle \Downarrow N_2$, and, moreover,

$$\begin{array}{rcl} \mathit{Tr}(\langle M_1, d_1; d_2\rangle)|_\ell &=& \mathit{Tr}(\langle M_1, d_1\rangle)|_\ell \cdot \mathit{Tr}(\langle N'_1, d_2\rangle)|_\ell \\ &=& \mathit{Tr}(\langle M_2, d_1\rangle)|_\ell \cdot \mathit{Tr}(\langle N'_2, d_2\rangle)|_\ell \\ &=& \mathit{Tr}(\langle M_2, d_1; d_2\rangle)|_\ell \end{array}$$

So $\langle M_1, d_1; d_2\rangle \approxeq_\ell \langle M_2, d_1; d_2\rangle$.

- If $c_1 = \texttt{if}\ e\ \texttt{then}\ d_1\ \texttt{else}\ d_2$ then $\Gamma \vdash e : \ell'$ and $\Gamma, (\ell \sqcup pc) \vdash d_i$ (for $i \in \{1, 2\}$). If $\ell' \sqsubseteq \ell$ then by Proposition 5 we have that $M_1(e) = M_2(e)$. Therefore both configurations take the same branch, and the result follows by application of the induction hypothesis. The case for $\ell' \not\sqsubseteq \ell$ follows from Proposition 6 and transitivity of $\approxeq_\ell$. □

The following proposition relates the executions of two well-typed programs formed by filling a target program with two different attacks. The proposition says that, assuming the set of high-integrity variables is not empty, if the memories in the initial configurations agree on high-integrity data and both configurations terminate then they also agree on high-integrity data at the end of computation. This is a form of noninterference of low-integrity code with high-integrity values.

Proposition 8. *If $H_I \neq \emptyset, \Gamma, pc \vdash c[\vec{\bullet}]$, $\langle M, c[\vec{a}]\rangle \Downarrow N$, and $\langle M', c[\vec{a'}]\rangle \Downarrow N'$ for some M' where $M(v) = M'(v)$ for all v such that $\Gamma(v) \in H_I$ and for some attacks $\vec{a}$ and $\vec{a'}$ then $N(v) = N'(v)$ for all v such that $\Gamma(v) \in H_I$.*

Proof. Induction on the structure of c. If $c[\vec{\bullet}]$ is `skip` then $M = N$ and $M' = N'$, which is a vacuous case. If $c[\vec{\bullet}]$ has the form $v := e$ (where e might contain declassification) then, as in the previous case, the command has no holes. If $\Gamma(v) \notin H_I$ then the high-integrity parts of the memories M and M' are not affected by the assignment. If $\Gamma(v) \in H_I$ then the type system guarantees that e might only depend on high-integrity data (if $\Gamma \vdash e : \ell$ then $\ell \sqsubseteq \Gamma(v)$), which implies that $N(v) = N'(v)$ for all v such that $\Gamma(v) \in H_I$. If $c[\vec{\bullet}] = [\bullet]$ then by Proposition 4 there are no assignments to high-integrity variables in either a or a', which gives the desired relation on the memories. Structural cases on c (where appropriate, we assume that $\vec{a}$ is split into two vectors $\vec{a_1}$ and $\vec{a_2}$):

$c_1[\vec{a_1}]; c_2[\vec{a_2}]$ Clearly, $\langle M, c_1[\vec{a_1}]\rangle \Downarrow N_1$ and $\langle M', c_1[\vec{a'_1}]\rangle \Downarrow N'_1$ for some N_1 and N'_1. By the induction hypothesis $N_1(v) = N'_1(v)$ for all high-integrity v. On the other hand, $\langle N_1, c_2[\vec{a_2}]\rangle \Downarrow N$ and $\langle N'_1, c_2[\vec{a'_2}]\rangle \Downarrow N'$. Applying the induction hypothesis again we receive $N(v) = N'(v)$ for all v such that $\Gamma(v) \in H_I$.

`if` b `then` $c_1[\vec{a_1}]$ `else` $c_2[\vec{a_2}]$ If $\exists v \in \mathit{Vars}(b).\, \Gamma(v) \notin H_I$ then there is a low-integrity variable that occurs in b. Therefore, the type system guarantees that there are no assignments to high-integrity variables in the branches $c_1[\vec{a_1}]$ and $c_2[\vec{a_2}]$; therefore the equality of the high parts of the memory is preserved through the entire command. If $\forall v \in \mathit{Vars}(b).\, \Gamma(v) \in H_I$ then b evaluates to the same value in both M_1 and M_2. Hence, this case follows from the induction hypothesis for $c_1[\vec{a_1}]$ (if this value is *true*) or for $c_2[\vec{a_2}]$ (if this value is *false*).

`while` b `do` $c_1[\vec{a_1}]$ If $\exists v \in \mathit{Vars}(b).\, \Gamma(v) \notin H_I$ then, as in the previous case, there is a low-integrity variable that occurs in b. Therefore, the type system guarantees that there are no assignments to high-integrity variables in the body $c_1[\vec{a_1}]$; therefore the equality of the high parts of the memory is preserved through the entire command. If $\forall v \in \mathit{Vars}(b).\, \Gamma(v) \in H_I$ then b evaluates to the same value in both M_1 and M_2. If b evaluates to *false* then this case reduces to `skip`. If b evaluates to *true* then this case reduces to a (finitely) nested sequential composition. We observe that b evolves under `while` b `do` $c_1[\vec{a}]$ and `while` b `do` $c_1[\vec{a'}]$ in the same way (due to the induction hypothesis). This guarantees that memories N and N' in which these loops (synchronously) terminate are equal in all high-integrity variables. □

In order to prove Theorem 3, we show a stronger property of typable commands. Suppose we have a typable program context filled with an attack and two memories forming configurations with this command. The next proposition states that whenever the terminating behaviors of the configurations are indistinguishable for the attacker then no alteration of the attacker-controlled part of the initial memory or alteration of the attacker-controlled code may make the behaviors distinguishable for the attacker. The key idea is that because declassification is not allowed in a low-integrity context (pc), the type system ensures that changes in low-integrity values may not reflect on low-confidentiality behavior of the computation.

Proposition 9. *If* $\Gamma, pc \vdash c[\vec{\bullet}]$ *and* $\langle M_1, c[\vec{a}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c[\vec{a}]\rangle$ *for some* M_1 *and* M_2 *then for any attack* $\vec{a'}$*, values* $val_1, \dots, val_n$*, and variables* $v_1, \dots, v_n$ *so that* $\forall i.\, \Gamma(v_i) \in L_I$ *we have* $\langle M_1', c[\vec{a'}]\rangle \approx_{(C(A),\top_I)} \langle M_2', c[a']\rangle$ *where* $M_1' = M_1[v_1 \mapsto val_1, \dots, v_n \mapsto val_n]$ *and* $M_2' = M_2[v_1 \mapsto val_1, \dots, v_n \mapsto val_n]$*.*

Proof. If $H_I = \emptyset$ then declassification is disallowed by the typing rules, and the proposition follows from Theorem 1. In the rest of the proof we assume $H_I \neq \emptyset$, and proceed by induction on the structure of $c[\vec{\bullet}]$. Suppose that for some $c[\vec{a}]$ and memories M_1 and M_2 we have $\langle M_1, c[\vec{a}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c[\vec{a}]\rangle$ (which, in particular, implies $M_1 =_{(C(A),\top_I)} M_2$). We need to show $\langle M_1', c[\vec{a'}]\rangle \approx_{(C(A),\top_I)} \langle M_2', c[\vec{a'}]\rangle$ for all $\vec{a'}$. Structural cases on $c[\vec{a}]$ (where appropriate, we assume that $\vec{a}$ is split into two vectors $\vec{a_1}$ and $\vec{a_2}$):

[$\bullet$] This case is straightforward because by Proposition 4 attack a' satisfies noninterference.

`skip` The proposition follows from the fact that $M_1' =_{(C(A),\top_I)} M_2'$, which is obvious.

$v := e$ The command has no holes, implying $c[\vec{a}] = c[\vec{a'}]$. If $\Gamma(v) \in H_C$ then the case is similar to `skip` because the assignment does not affect any part of the memory that is visible to the attacker. If $\Gamma(v) \in L_C$ then we have two sub-cases. If e is a declassification expression, then it does not depend on low-integrity variables, and hence low-confidentiality indistinguishability is preserved. If e is not a declassification expression, then the type system guarantees that $\Gamma \vdash e : \ell'$ for such ℓ' that $\ell' \sqsubseteq \ell$, which implies that e may depend only on low-confidentiality data. Whether the data that is used in e comes from M_1/M_2 or M_1'/M_2' does not matter because, in all cases, e evaluates to the same value in all of them. Hence, the assignment preserves low-confidentiality indistinguishability.

$c_1[\vec{a_1}]; c_2[\vec{a_2}]$ In this case we have $\langle M_1, c_1[\vec{a_1}]; c_2[\vec{a_2}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c_1[\vec{a_1}]; c_2[\vec{a_2}]\rangle$. By Proposition 7 we infer $\langle M_1, c_1[\vec{a_1}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c_1[\vec{a_1}]\rangle$. By the induction hypothesis we obtain $\langle M_1', c_1[\vec{a_1'}]\rangle \approx_{(C(A),\top_I)} \langle M_2', c_1[\vec{a_1'}]\rangle$. If one, say $\langle M_1', c_1[\vec{a_1'}]\rangle$, of the configurations diverges then $\langle M_1', c[\vec{a'}]\rangle$ also diverges. Since weak indistinguishability relates divergent traces to any trace, conclude $\langle M_1', c[\vec{a'}]\rangle \approx_{(C(A),\top_I)} \langle M_2', c[\vec{a'}]\rangle$. If both configurations $\langle M_1', c_1[\vec{a_1'}]\rangle$ and $\langle M_2', c_1[\vec{a_1'}]\rangle$ terminate, we have $\langle M_1', c_1[\vec{a_1'}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2', c_1[\vec{a_1'}]\rangle$. Thus, there exist some N_1' and N_2' so that $\langle M_1', c_1[\vec{a_1'}]\rangle \Downarrow N_1'$, $\langle M_2', c_1[\vec{a_1'}]\rangle \Downarrow N_2'$, and $N_1' =_{(C(A),\top_I)} N_2'$.

Because $\langle M_1, c_1[\vec{a_1}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c_1[\vec{a_1}]\rangle$, there exist some N_1 and N_2 such that $\langle M_1, c_1[\vec{a_1}]\rangle \Downarrow N_1$ and $\langle M_2, c_1[\vec{a_1}]\rangle \Downarrow N_2$. Applying Proposition 8 twice, we have $N'_1(v) = N_1(v)$ and $N'_2(v) = N_2(v)$ for all high-integrity variables v. Because $\langle M_1, c[\vec{a}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c[\vec{a}]\rangle$, by the last part of Proposition 7 we have $\langle N_1, c_2[\vec{a_2}]\rangle \approxeq_{(C(A),\top_I)} \langle N_2, c_2[\vec{a_2}]\rangle$. The induction hypothesis allows us to change memories N_1 and N_2 into N'_1 and N'_2, respectively, and attack $\vec{a_2}$ into $\vec{a'_2}$, yielding $\langle N'_1, c_2[\vec{a'_2}]\rangle \approx_{(C(A),\top_I)} \langle N'_2, c_2[\vec{a'_2}]\rangle$. Connecting the low-assignment traces for c_1 and c_2, $\langle M'_1, c_1[\vec{a'_1}]; c_2[\vec{a'_2}]\rangle \approx_{(C(A),\top_I)} \langle M'_2, c_1[\vec{a'_1}]; c_2[\vec{a'_2}]\rangle$.

`if` b `then` $c_1[\vec{a_1}]$ `else` $c_2[\vec{a_2}]$ If $\exists v \in \mathit{Vars}(b).\, \Gamma(v) \notin L_C$, i.e., a high-confidentiality variable occurs in b, then we observe that there are no assignments to low-confidentiality variables in the program, which is a vacuous case. If $\exists v \in \mathit{Vars}(b).\, \Gamma(v) \notin H_I$ then no declassification occurs in $c[\vec{a'}]$ by the definition of the type system and attacks. Consequently, by Theorem 1, $c[\vec{a'}]$ satisfies noninterference, which completes the proof.

If $\forall v \in \mathit{Varsb}.\, \Gamma(v) \in L_C \cap H_I$ then b evaluates to the same value, say *true*, under all of M_1, M_2, M'_1, and M'_2, i.e., the execution of the conditional reduces to the same branch in both memories. We have $\langle M_1, c[\vec{a}]\rangle \longrightarrow \langle M_1, c_1[\vec{a_1}]\rangle$ and $\langle M_2, c[\vec{a}]\rangle \longrightarrow \langle M_2, c_1[\vec{a_1}]\rangle$ as well as $\langle M'_1, c[\vec{a'}]\rangle \longrightarrow \langle M'_1, c_1[\vec{a'_1}]\rangle$ and $\langle M'_2, c[\vec{a'}]\rangle \longrightarrow \langle M'_2, c_1[\vec{a'_1}]\rangle$. Because $\langle M_1, c[\vec{a}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c[\vec{a}]\rangle$ we have $\langle M_1, c_1[\vec{a_1}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c_1[\vec{a_1}]\rangle$. By the induction hypothesis, we have $\langle M'_1, c_1[\vec{a'_1}]\rangle \approx_{(C(A),\top_I)} \langle M'_2, c_1[\vec{a'_1}]\rangle$. This implies $\langle M'_1, c[\vec{a'}]\rangle \approx_{(C(A),\top_I)} \langle M'_2, c[\vec{a'}]\rangle$.

`while` b `do` $c_1[\vec{a}]$ The cases when $\exists v \in \mathit{Vars}(b).\, \Gamma(v) \notin L_C$ and $\exists v \in \mathit{Vars}(b).\, \Gamma(v) \notin H_I$ are handled in the same way as for conditionals. The remaining case is $\forall v \in \mathit{Vars}(b).\, \Gamma(v) \in L_C \cap H_I$. Expression b evaluates to the same value under all of M_1, M_2, M'_1, and M'_2. If this value is *false* then both $\langle M'_1, c[\vec{a'}]\rangle$ and $\langle M'_2, c[\vec{a'}]\rangle$ terminate in one step with no change to the memories M'_1 and M'_2, yielding $\langle M'_1, c[\vec{a'}]\rangle \approx_{(C(A),\top_I)} \langle M'_2, c[\vec{a'}]\rangle$.

If, on the other hand, the value of b is *true* then $\langle M_1, \texttt{while}\ b\ \texttt{do}\ c_1[\vec{a}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, \texttt{while}\ b\ \texttt{do}\ c_1[\vec{a}]\rangle$ ensures that $\langle M_1, c_1[\vec{a}]; \texttt{while}\ b\ \texttt{do}\ c_1[\vec{a}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c_1[\vec{a}]; \texttt{while}\ b\ \texttt{do}\ c_1[\vec{a}]\rangle$. By Proposition 7, we have $\langle M_1, c_1[\vec{a}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c_1[\vec{a}]\rangle$. By the induction hypothesis, $\langle M'_1, c_1[\vec{a'}]\rangle \approx_{(C(A),\top_I)} \langle M'_2, c_1[\vec{a'}]\rangle$. If either $\langle M'_1, c_1[\vec{a'}]\rangle$ or $\langle M'_2, c_1[\vec{a'}]\rangle$ diverges then the top-level loop also diverges under M'_1 (or M'_2), implying $\langle M'_1, c[\vec{a'}]\rangle \approx_{(C(A),\top_I)} \langle M'_2, c[\vec{a'}]\rangle$. If both configurations terminate, then there exist some N'_1 and N'_2 so that $\langle M'_1, c_1[\vec{a'}]\rangle \Downarrow N'_1$, $\langle M'_2, c_1[\vec{a'}]\rangle \Downarrow N'_2$, and $N'_1 =_{(C(A),\top_I)} N'_2$. Note that the value of b is the same under N'_1 and N'_2. If this value is *false* then the proof is finished. Otherwise, we need to further unwind the loop, as follows.

Applying Proposition 8 twice, we infer $\langle M_1, c_1[\vec{a}]\rangle \Downarrow N_1$ and $\langle M_2, c_1[\vec{a}]\rangle \Downarrow N_2$ for some N_1 and N_2 so that $N'_1(v) = N_1(v)$ and $N'_2(v) = N_2(v)$ for all high-integrity variables v. This, in particular, implies that b evaluates to *true* in both N_1 and N_2.

Because $\langle M_1, c[\vec{a}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c[\vec{a}]\rangle$, we apply Proposition 7 to receive $\langle M_1, c_1[\vec{a}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c_1[\vec{a}]\rangle$ and thus $\langle N_1, c[\vec{a}]\rangle \approxeq_{(C(A),\top_I)} \langle N_2, c[\vec{a}]\rangle$ (because $c[\vec{a}]$ corresponds to unwinding the loop).

Note that we have mimicked a $c[\vec{a}]$ iteration by a $c[\vec{a'}]$ iteration (with the possibility that the latter might diverge because of an internal loop caused by low-integrity computation). During this iteration we have preserved the invariant that the executions for both respective pairs M_1, M_2 and M_1', M_2' give indistinguishable low-assignment traces (for each $c_1[\vec{a}]$ and $c_1[\vec{a'}]$ respectively), and low-confidentiality high-integrity data in the final states of all traces is the same regardless of the memory (M_1, M_2, M_1' or M_2') and the command ($c_1[\vec{a}]$ or $c_1[\vec{a'}]$). By finitely repeating this construction (with the possibility of finishing the proof at each step because of an internal loop of $c_1[\vec{a'}]$), we reach the state when b evaluates to *false*, which corresponds to the termination of the top-level loop for both $c_1[\vec{a}]$ and $c_1[\vec{a'}]$. We receive $\langle M_1', c[\vec{a'}]\rangle \approx_{(C(A),\top_I)} \langle M_2', c[\vec{a'}]\rangle$ by concatenating low-assignment traces from each iteration. $\square$

Theorem 3 is a simple corollary of the above proposition:

Proof (Theorem 3). By Proposition 9, setting $M_1' = M_1$ and $M_2' = M_2$. $\square$

It is worth clarifying the relation of this type system to that defined by Zdancewic [50]. While both require high pc integrity in the typing rule for `declassify`, the present system also requires high integrity of the expression to be declassified. The purpose of the latter requirement is illustrated by the following example:

$$[\bullet]; \texttt{if}\ x_{HL}\ \texttt{then}\ y_{HL} := z_{HL}\ \texttt{else}\ y_{HL} := v_{HL};$$
$$w_{LL} := \texttt{declassify}(y_{HL}, LL)$$

This program is allowed by the typing rules presented by Zdancewic [50]. However, the program clearly violates the definition of robustness presented here. By requiring high integrity of the declassified expression, the type system in Figure 4 rejects the program.

6 Password checking example

This section applies robust declassification to a program that performs password checking, illustrating how the type system gives security types to password-checking routines and prevents attacks.

Password checking in general releases information about passwords when attempts are made to log on. This is true even when the login attempt is unsuccessful, because the user learns that the password is *not* the password tried. A password checker must therefore declassify the result of password checking in order to report it to the user. A danger is that an attacker might exploit this login procedure by encoding some other sensitive data as a password.

We consider UNIX-style password checking where the system database stores *images* (e.g., secure hashes) of password-salt pairs. The salt is a publicly readable string stored in the database for each user id, as an impediment to dictionary attacks. For a successful login, the user is required to provide a query such that the hash of the string and the salt for that user matches the image from the database.

Below are typed expressions/programs for computing the hash, matching the user input to the password image from the database, and updating the password. Arrows in the types for expressions indicate that under the types of the arguments on the left from the arrow, the type of the result is on the right from the arrow. The expression $\mathtt{hash}(pwd, salt)$ concatenates the password pwd with the salt $salt$ and applies the one-way hash function `buildHash` to the concatenation (the latter is denoted by $||$). The result is declassified to the level C_{salt} (where $\Gamma(salt) \in L_C$). The command $\mathtt{match}(pwdI, salt, pwd, hashR, matchR)$ checks whether the password image $pwdI$ matches the hash of the password pwd with the salt $salt$. It stores the result in the variable $matchR$. We assume that C_v and I_v denote the confidentiality $C(\Gamma(v))$ and integrity $I(\Gamma(v))$ of the variable v, respectively.

$$\begin{aligned}
&\Gamma, pc \vdash \mathtt{hash}(pwd, salt) : \\
&\qquad C_{pwd} I_{pwd} \times C_{salt} I_{salt} \to C_{salt} I \\
&\qquad = \mathtt{declassify}(\mathtt{buildHash}(pwd||salt), C_{salt} I) \\
&\Gamma, pc \vdash \mathtt{match}(pwdI, salt, pwd, hashR, matchR) \\
&\qquad = \ hashR := \mathtt{hash}(pwd, salt); \\
&\qquad\quad\ matchR := (pwdI == hashR)
\end{aligned}$$

where $C_{matchR} = C_{pwdI} \sqcup C_{salt}$, $I_{matchR} = I_{pwdI} \sqcup I$, $I = I_{pwd} \sqcup I_{salt}$; and $I(A) \not\sqsubseteq I, I(A) \not\sqsubseteq I(pc)$ (both I and $I(pc)$ have high integrity). As before, basic security types are written in the form CI (e.g., LH) where C is the confidentiality level and I is the integrity level. Let us assume the lattice $\mathcal{L}_{LH}$ from Figure 1 and $A = LL$. Instantiating the typings (and omitting the environment Γ) for these functions shows that they capture the desired intuition:

The users apply hash to a password and salt:
$LH \vdash \mathtt{hash}(pwd, salt) : HH \times LH \to LH$

The users match a password to a password image:
$LH \vdash \mathtt{match}(pwdI, salt, pwd, hashR, matchR) : LH \times LH \times HH \times LH \times LH$

Consider an attack that exploits declassification in `hash` and `match` in order to leak information about whether x_{HH} ($\Gamma(x_{HH}) = HH$) equals y_{LL} ($\Gamma(y_{LL}) = LL$):

$$\begin{aligned}
&[\bullet]; \mathtt{match}(\mathtt{hash}(x_{HH}, 0), 0, y_{LL}, hashR, matchR); \\
&\mathtt{if}\ matchR\ \mathtt{then}\ z_{LL} := 1\ \mathtt{else}\ z_{LL} := 0
\end{aligned}$$

This attack is rejected by the type system because low-integrity data y_{LL} is fed to `match`. Indeed, this attack compromises robustness. For example, take M_1 and M_2 such that $M_1(x_{HH}) = 2$ and $M_2(x_{HH}) = 3$; $a = y_{LL} := 0$; and $a' = y_{LL} := 2$.

We have $\langle M_1, c[a]\rangle \approx_{(C(A),\top_I)} \langle M_2, c[a]\rangle$ (the `else` branch is taken regardless of x_{HH}) but $\langle M_1, c[a']\rangle \not\approx_{(C(A),\top_I)} \langle M_2, c[a']\rangle$ (which branch of the conditional is taken depends on the outcome of the `match`).

As a side note, this laundering attack is not defended against in many approaches that are agnostic about the origin (or integrity) of data. For example, a typical intransitive noninterference model accepts the attack as a secure program. Clearly, robust declassification and intransitive noninterference capture different aspects of safe downgrading.

The process of updating passwords can also be modeled as a typable program that satisfies robustness. We might define a procedure `update` to which the users must provide their old password in order to update to a new password:

$$\begin{aligned}
\Gamma, pc \vdash\ & \mathtt{update}(pwdI, salt, oldP, newP, hashR, matchR) \\
&= \mathtt{match}(pwdI, salt, oldP, hashR, matchR); \\
&\quad \mathtt{if}\ matchR \\
&\qquad \mathtt{then}\ pwdI := \mathtt{hash}(newP, salt) \\
&\qquad \mathtt{else\ skip}
\end{aligned}$$

where $C_{salt}(I_{salt} \sqcup I_{oldP} \sqcup I_{newP}) \sqsubseteq C_{pwdI} I_{pwdI}$ and $I_{salt}, I_{oldP}, I_{newP}, I(pc)$ have high integrity. In order for this code to be well-typed, both the old password $oldP$ and the new password $newP$ must be high-integrity variables; otherwise, `hash` would attempt to declassify low-integrity information $newP$ (with the decision to declassify dependent on low-integrity information $oldP$), which the type system prevents. Thus, an attacker is prevented from using the password system to launder information. Instantiating this typing to the simple lattice $\mathcal{L}_{LH}$ and $A = LL$ is as follows:

The users modify a password:
$LH \vdash \mathtt{update}(pwdI, salt, oldP, newP, hR, mR) : LH \times LH \times HH \times HH \times LH \times LH$

7 Endorsement and qualified robustness

Sometimes it makes sense to give untrusted code the ability to affect what information is released by a program. For example, consider an application that allows untrusted users to select and purchase information. The information provider does not care which information is selected, assuming that payment is forthcoming. This application is abstractly described by the following code:

$$\begin{aligned}
[\bullet]; \mathtt{if}\ x_{LL} = 1\ \mathtt{then}\ & z_{LH} := \mathtt{declassify}(y_{HH}, LH) \\
\mathtt{else}\ & z_{LH} := \mathtt{declassify}(y'_{HH}, LH)
\end{aligned}$$

There are two pieces of confidential information available, y_{HH} and y'_{HH}. The purchaser computes the choice in low-integrity code $\bullet$, which sets the variable x_{LL}. The user expects to receive output on z_{LH}. This code obviously violates robust declassification because the "attacks" $x_{LL} := 1$ and $x_{LL} := 2$ release different information, yet the program can reasonably be considered secure.

7.1 Characterizing qualified robustness

To address this shortcoming, we generalize robust declassification to a *qualified robustness* property in which untrusted code is given a limited ability to affect information release. This ability is marked explicitly in the code by the use of a new construct, $\texttt{endorse}(e, \ell)$. This *endorsement* operation has the same result as the expression e but *upgrades* the integrity of the result, indicating that although this value might be affected by untrusted code, the real security policy is insensitive to the value.

Suppose that the program contains endorsements of some expressions. We wish to qualify the robust declassification property to make it insensitive to how these expressions evaluate. To do this we consider the behavior of the program under an alternate semantics for `endorse` expressions, in which the `endorse` expression evaluates to a nondeterministically chosen new value val:

$$\langle M, \texttt{endorse}(e, \ell)\rangle \longrightarrow val$$

Interpreting the `endorse` statement in this way makes the evaluation semantics nondeterministic, so it is necessary to modify the definitions of configuration indistinguishability to reflect the fact that a given configuration may have multiple traces.

Let $T^{\Downarrow}$ stand for $\{t \in T \mid t \Downarrow\}$, the set of terminating traces from trace set T. Two trace sets T_1 and T_2 (generated by some two configurations) are *indistinguishable up to* ℓ (written $T_1 \approx_\ell T_2$) if whenever $T_1^{\Downarrow} \neq \emptyset$ and $T_2^{\Downarrow} \neq \emptyset$ then $\forall t_1 \in T_1^{\Downarrow}. \exists t_2 \in T_2^{\Downarrow}. t_1 \approx_\ell t_2 \quad \& \quad \forall t_2 \in T_2^{\Downarrow}. \exists t_1 \in T_1^{\Downarrow}. t_1 \approx_\ell t_2$. In other words, two trace sets are indistinguishable up to ℓ if either one of them contains diverging traces only or for any terminating trace from T_1 we can find a terminating trace from T_2 so that they are indistinguishable up to ℓ, and vice versa. We can now lift Definition 5 to multiple-trace semantics:

Definition 11. *Two configurations $\langle M_1, c_1\rangle$ and $\langle M_2, c_2\rangle$ are* weakly indistinguishable up to ℓ *(written $\langle M_1, c_1\rangle \approx_\ell \langle M_2, c_2\rangle$) if $Tr(\langle M_1, c_1\rangle) \approx_\ell Tr(\langle M_2, c_2\rangle)$. We say that two configurations are* strongly indistinguishable up to ℓ *(written $\langle M_1, c_1\rangle \approxeq_\ell \langle M_2, c_2\rangle$) if $\langle M_1, c_1\rangle \approx_\ell \langle M_2, c_2\rangle$ and both $\langle M_1, c_1\rangle$ and $\langle M_2, c_2\rangle$ always terminate.*

Using this notation, the robust declassification property can be qualified to express the idea that the attacker's effect on endorsed expressions does not matter:

Definition 12 (Qualified robustness). *Command $c[\vec{\bullet}]$ has* qualified robustness *with respect to fair attacks if*

$$\forall M_1, M_2, \vec{a} \in F, \vec{a'} \in F. \langle M_1, c[\vec{a}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c[\vec{a}]\rangle \implies \\ \langle M_1, c[\vec{a'}]\rangle \approx_{(C(A),\top_I)} \langle M_2, c[\vec{a'}]\rangle$$

Note the similarity of qualified robustness to the original robustness property from Definition 9. In fact, the difference is entirely contained in the generalized indistinguishability relations $\approxeq_{(C(A),\top_I)}$ and $\approx_{(C(A),\top_I)}$.

One may wonder why, if confidentiality and integrity are dual properties (cf. [5]), the treatment of `declassify` and `endorse` are not more symmetric. We argue that there are at least two explanations:

- The first explanation is that robustness is fundamentally about regulating access to the potentially dangerous operations like `declassify` and `endorse`, and, because both of them are "privileged" operations, the robustness constraints should be the *same*, not *dual*, for them.
- Another explanation is that treating integrity as a strict dual to confidentiality leads to a very weak notion of integrity (see, for example, the discussion by Li et al. [24, 26]), so we should not be surprised if treating them as duals leads to somewhat unsatisfactory results. The use of nondeterministic semantics for `endorse` but not `declassify` compensates for the differences between them.

It is fair to say that understanding the complete relation between declassification and endorsement is still unsettled, and poses an interesting problem for future work.

7.2 Enforcing qualified robustness

The use of `endorse` is governed by the following typing rule; in addition, attacker code may not use `endorse`:

$$\frac{\Gamma \vdash e : \ell' \quad \ell \sqcup pc \sqsubseteq \Gamma(v) \quad C(\ell) = C(\ell') \quad \ell' \in L_I \quad \ell \in H_I}{\Gamma, pc \vdash v := \texttt{endorse}(e, \ell)}$$

Adding this rule to the type system has no impact on confidentiality when no `declassify` occurs in a program. To be more precise, we have the following theorem:

Theorem 4. *If $\Gamma, pc \vdash c$ and no* `declassify` *occurs in c then for all confidentiality levels C we have*

$$\forall M_1, M_2.\, M_1 =_{(C,\top_I)} M_2 \Longrightarrow \langle M_1, c\rangle \approx_{(C,\top_I)} \langle M_2, c\rangle$$

Proof. We assume $Tr(\langle M_1, c\rangle)^{\Downarrow} \neq \emptyset$ and $Tr(\langle M_2, c\rangle)^{\Downarrow} \neq \emptyset$. Induction on the structure of c.

`skip` $Tr(\langle M_1, c\rangle)|_{(C,\top_I)} = M_1|_{(C,\top_I)} = M_2|_{(C,\top_I)} = Tr(\langle M_2, c\rangle)_{(C,\top_I)}$.

$v := e$ If $e = \texttt{endorse}(e', \ell)$ for some e' and ℓ then v can result in any value. Clearly, $\langle M_1, c\rangle \approx_{(C,\top_I)} \langle M_2, c\rangle$ regardless of the confidentiality level of v. If e is not an endorsement expression, then the proof is the same as in Theorem 1.

$c_1; c_2$ Suppose $t_1 \in Tr(\langle M_1, c_1; c_2\rangle)^{\Downarrow}$. There exist t_{11}, t_{12}, and N_1 such that $t_1 = t_{11} \cdot t_{12}$ and $t_1 \Downarrow N_1$. Clearly, $Tr(\langle M_1, c_1\rangle)^{\Downarrow} \neq \emptyset$ and $Tr(\langle M_2, c_1\rangle)^{\Downarrow} \neq \emptyset$. By the induction hypothesis, there exists $t_{21} \in Tr(\langle M_2, c_1\rangle)^{\Downarrow}$ so that $t_{11} \approx_{(C,\top_I)} t_{21}$. Suppose $t_{21} \Downarrow N_2$ for some N_2. We know $N_1 =_{(C,\top_I)} N_2$ by the induction hypothesis. This allows us to apply the induction hypothesis to c_2. If $Tr(\langle N_1, c_2\rangle)^{\Downarrow} = \emptyset$ or $Tr(\langle N_2, c_2\rangle)^{\Downarrow} = \emptyset$ then we receive a contradiction to the initial assumption about c. Hence, we receive that there exists $t_{22} \in Tr(\langle N_2, c_2\rangle)^{\Downarrow}$ such that $t_{12} \approx_{(C,\top_I)} t_{22}$. Connecting the traces we receive $t_1 \approx_{(C,\top_I)} t_{21} \cdot t_{22}$ where $t_{21} \cdot t_{22} \in Tr(\langle M_2, c_1; c_2\rangle)^{\Downarrow}$, which finishes the proof.

`if` b `then` $c_1[\vec{a_1}]$ `else` $c_2[\vec{a_2}]$ As in Theorem 1.

`while` b `do` $c_1[\vec{a_1}]$ As in Theorem 1. □

The interesting question is what security assurance is guaranteed in the presence of both `declassify` and `endorse`. The rule above rejects possible misuses of the endorsement mechanism leading to undesired declassification, as illustrated by the following example:

```
[•]; if $x_{LL}$ then $y_{LH}$ := endorse($z_{LL}, LH$)
                 else skip;
        if $y_{LH}$ then $v_{LH}$ := declassify($w_{HH}, LH$)
                 else skip
```

In this example, the attacker has control over x_{LL} which, in turn, controls whether the variable z_{LL} is endorsed for assignment to y_{LH}. It is through the compromise of y_{LH} that the attacker might cause the declassification of w_{HH}. This program does not satisfy qualified robustness (take $M_1(w_{HH}) = 2, M_2(w_{HH}) = 3, M_1(y_{LH}) = M_2(y_{LH}) = 0, M_1(z_{LL}) = M_2(z_{LL}) = 1, a = x_{LL} := 0$ and $a' = x_{LL} := 1$ to receive $\langle M_1, c[a]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c[a]\rangle$ but $\langle M_1, c[a']\rangle \not\approxeq_{(C(A),\top_I)} \langle M_2, c[a']\rangle$) and is rightfully rejected by the type system (`endorse` fails to type check under a low-integrity pc). In general, we prove that all typable programs (using the extended type system that includes the rule for `endorse`) must satisfy qualified robustness:

Theorem 5. *If $\Gamma, pc \vdash c[\vec{\bullet}]$ then $c[\vec{\bullet}]$ satisfies qualified robust declassification.*

As in Section 5, we prove the theorem with respect to a larger class of attacks, namely A-attacks. Recall that A-attacks subsume fair attacks by Proposition 3. In order to prove the theorem we lift the proof technique of Theorem 3 to a possibilistic setting by reasoning about the existence of individual traces that originate from a given configuration and possess desired properties. In parentheses, we provide references to the respective propositions and definitions for the non-qualified version of robustness.

Proposition 10 (4). *An A-attack (i) does not have occurrences of assignments to high-integrity variables (such v that $\Gamma(v) \in H_I$); and (ii) satisfies (possibilistic) noninterference.*

Proof. The proof is a straightforward adaptation of the proof of Proposition 4. □

Proposition 11 (7). *Let $\ell = (C(A), \top_I)$. If $\Gamma, pc \vdash c_1; c_2$ and $\langle M_1, c_1; c_2\rangle \approxeq_\ell \langle M_2, c_1; c_2\rangle$ then $\langle M_1, c_1\rangle \approxeq_\ell \langle M_2, c_1\rangle$. Moreover, for $t_1 \in Tr(\langle M_1, c_1\rangle)$ such that t_1 terminates in N_1 and $t_2 \in Tr(\langle M_2, c_1\rangle)$ such that t_2 terminates in N_2 if $t_1 \approx_\ell t_2$ then $\langle N_1, c_2\rangle \approxeq_\ell \langle N_2, c_2\rangle$.*

Proof. The proof follows by induction on the structure of c_1, using mostly the same structure as Proposition 7. Note that since the operational semantics is deterministic except in the case of `endorse`, most of the cases for this proof follow directly from the prior proposition. Nondeterminism plays an interesting role only for the assignment and sequencing operations, so we show only those cases here. From the assumptions that the configurations are related, we obtain $M_1 =_\ell M_2$.

- If $c_1 = \texttt{endorse}(e)$, then we have $T_1 = Tr(\langle M_1, x := \texttt{endorse}(e)\rangle) =$

  $$\begin{array}{l} T_1 = \{\langle M_1, x := \texttt{endorse}(e)\rangle \xrightarrow{x} \langle M_1[x \mapsto v], \texttt{skip}\rangle \mid v \in Val\} \\ T_2 = \{\langle M_2, x := \texttt{endorse}(e)\rangle \xrightarrow{x} \langle M_2[x \mapsto v], \texttt{skip}\rangle \mid v \in Val\} \end{array}$$

  If $\Gamma(x) \not\sqsubseteq \ell$ then the first part of the proposition follows because every trace in T_1 has ℓ-projection M_1 and every trace in T_2 has ℓ-projection M_2. If $\Gamma(x) \sqsubseteq \ell$ then for every $t_1 \in T_1$ it is easy to find related trace in T_2 by choosing the same nondeterministic choice. The second part of the theorem follows by similar reasoning, by noting that if there were no possible way for c_1 to evaluate via related traces then it would be impossible to extend the traces in T_1 and T_2 with a trace for c_2, which contradicts the assumption.

- If $c_1 = d_1; d_2$, then from the typing rule we obtain that $\Gamma; pc \vdash d_1$ and $\Gamma; pc \vdash d_2$. For the first part of the proposition, we use the inductive hypothesis plus the associativity of trace composition (as in the deterministic analog of this proof) to conclude that $\langle M_1, d_1\rangle \approxeq_\ell \langle M_2, d_1\rangle$ and, consequently for any $t_1 \in Tr(\langle M_1, d_1\rangle)$ there exists a trace $t_2 \in Tr(\langle M_2, d_1\rangle)$ such that $t_1 \approx_\ell t_2$ (and vice versa). Furthermore, t_1 terminates in some state N'_1 and t_2 terminates in some ℓ-equivalent state N'_2. A second application of the inductive hypothesis yields that $\langle N'_1, d_2\rangle \approxeq_\ell \langle N'_2, d_2\rangle$ so that for all $u_1 \in Tr(\langle N'_1, d_2\rangle)$ there is a $u_2 \in Tr(\langle N'_2, d_2\rangle)$ where $u_1 \approx_\ell u_2$. Let $r_1 \in Tr(\langle M_1, d_1; d_2\rangle)$ be given. It is enough to show that there exists an $r_2 \in Tr(\langle M_2, d_1; d_2\rangle)$ such that $r_1 \approx_\ell r_2$. But $r_1 = t_1 \xrightarrow{\cdot} u_1$ for some t_1 and u_1 and we can construct r_2 by appending the corresponding t_2 and u_2 provided by the inductive hypothesis. The second part of the proof follows from the fact that we assume that $c_1; c_2$ is everywhere terminating and reasoning similar to the case above that says that it is not possible to extend two inequivalent traces to obtain equivalent ones.

□

Proposition 12 (8). *If $H_I \neq \emptyset$ and $\Gamma, pc \vdash c[\vec{\bullet}]$ and $\langle M, c[\vec{a}]\rangle$ always terminates (for some attack $\vec{a}$) then for any attack $\vec{a'}$, values $val_1, \ldots, val_n$, variables $v_1, \ldots, v_n$ where $\forall i. \Gamma(v_i) \in L_I$, and $t' \in Tr(\langle M[v_1 \mapsto val_1, \ldots, v_n \mapsto val_n], c[\vec{a'}]\rangle)$ where t' terminates, there exists $t \in Tr(\langle M, c[\vec{a}]\rangle)$ where $t \sim_\ell t'$ for all such ℓ that $\ell \in H_I$.*

Proof. The proof closely follows the one of Proposition 8. Suppose $M' = M[v_1 \mapsto val_1, \ldots, v_n \mapsto val_n]$. Induction on the structure of c. If $c[\vec{\bullet}]$ is `skip` then there is only one possible $t \in Tr(\langle M, c[\vec{a}]\rangle)$ and $t \sim_\ell t'$ because $t|_\ell = M|_\ell = M'|_\ell = t'|_\ell$ for all such ℓ that $\ell \in H_I$. If $c[\vec{\bullet}]$ has the form $v := e$ (where e might contain declassification or endorsement) then, as in the previous case, the command has no holes. Suppose $N = M[v \mapsto M(e)]$ and $N' = M'[v \mapsto M'(e)]$. If $\Gamma(v) \notin H_I$ then the high-integrity parts of the memories M and M' are not affected by the assignment, and the case is similar to `skip`. If $\Gamma(v) \in H_I$ and e is not an endorsement expression, then the type system guarantees that e might only depend on high-integrity data (if $\Gamma \vdash e : \ell$ then $\ell \sqsubseteq \Gamma(v)$), which implies that $N(w) = N'(w)$ for all w such that $\Gamma(w) \in H_I$. If $\Gamma(v) \in H_I$ and e is an endorsement expression, then any value can be assigned to

v as an outcome of assignment under both M and M'. In this case, t' has the form $M', M'[v \mapsto val]$ for some val. Clearly, $M|_\ell, M[v \mapsto val]|_\ell$ is a possible ℓ-projection of trace $Tr(\langle M, c[\vec{a}]\rangle)$ and $(M|_\ell, M[v \mapsto val]|_\ell) = (M'|_\ell, M'[v \mapsto val]|_\ell)$ for all ℓ such that $\Gamma(\ell) \in H_I$. If $c[\vec{\bullet}] = [\bullet]$ then by Proposition 4 there are no assignments to high-integrity variables in either a or a', which gives the desired relation on the traces. Structural cases on c (where appropriate, we assume that $\vec{a}$ is split into two vectors $\vec{a_1}$ and $\vec{a_2}$):

$c_1[\vec{a_1}]; c_2[\vec{a_2}]$ Assume $t' = t'_1 \cdot t'_2$, $t'_1 \Downarrow N'$ (for some N'), $t'_1 \in Tr(\langle M', c_1[\vec{a'_1}]\rangle)$, and $t'_2 \in Tr(\langle N', c_2[\vec{a'_2}]\rangle)$. By the induction hypothesis there exists t_1 such that $t_1 \sim_\ell t'_1$ (for all ℓ such that $\Gamma(\ell) \in H_I$) and $t_1 \Downarrow N$ (for some N). This implies $N(v) = N'(v)$ for all v that $\Gamma(v) \in H_I$, and we can apply the induction hypothesis to c_2. By the induction hypothesis, there exists t_2 with the initial memory N such that $t_2 \sim_\ell t'_2$ (for all ℓ such that $\Gamma(\ell) \in H_I$). Connecting the traces, we receive $t_1|_\ell \cdot t_2|_\ell = t'|_\ell$ (for all ℓ such that $\Gamma(\ell) \in H_I$).

`if` b `then` $c_1[\vec{a_1}]$ `else` $c_2[\vec{a_2}]$ As in Proposition 8.

`while` b `do` $c_1[\vec{a_1}]$ As in Proposition 8. □

Proposition 13 (9). *If $\Gamma, pc \vdash c[\vec{\bullet}]$ and $\langle M_1, c[\vec{a}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c[\vec{a}]\rangle$ for some M_1 and M_2 then for any attack $\vec{a'}$, values $val_1, \ldots, val_n$, and variables $v_1, \ldots, v_n$ where $\forall i.\, \Gamma(v_i) \in L_I$ we have $\langle M'_1, c[\vec{a'}]\rangle \approx_{(C(A),\top_I)} \langle M'_2, c[a']\rangle$ where $M'_1 = M_1[v_1 \mapsto val_1, \ldots, v_n \mapsto val_n]$ and $M'_2 = M_2[v_1 \mapsto val_1, \ldots, v_n \mapsto val_n]$.*

Proof. If $H_I = \emptyset$ then declassification is disallowed by the typing rules, and the proposition follows from Theorem 4. In the rest of the proof we assume $H_I \neq \emptyset$. Suppose that for some $c[\vec{a}]$ and memories M_1 and M_2 we have $\langle M_1, c[\vec{a}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c[\vec{a}]\rangle$ (which, in particular, implies $M_1 =_{(C(A),\top_I)} M_2$). We need to show $\langle M'_1, c[\vec{a'}]\rangle \approx_{(C(A),\top_I)} \langle M'_2, c[\vec{a'}]\rangle$ for all $\vec{a'}$.

The proof is by structural induction on $c[\vec{\bullet}]$. We only show the sequential composition case as the rest of the cases can be reconstructed straightforwardly from the proof of Theorem 3. Assume $c[\vec{\bullet}] = c_1[\vec{\bullet_1}]; c_2[\vec{\bullet_2}]$ where the vector $\vec{\bullet}$ is split into two vectors $\vec{\bullet_1}$ and $\vec{\bullet_2}$.

A premise of the proposition is $\langle M_1, c_1[\vec{a_1}]; c_2[\vec{a_2}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c_1[\vec{a_1}]; c_2[\vec{a_2}]\rangle$. We need to show $\langle M'_1, c_1[\vec{a'_1}]; c_2[\vec{a'_2}]\rangle \approx_{(C(A),\top_I)} \langle M'_2, c_1[\vec{a'_1}]; c_2[\vec{a'_2}]\rangle$, i.e., by Definition 11, whenever $Tr(\langle M'_1, c_1[\vec{a'_1}]; c_2[\vec{a'_2}]\rangle)^{\Downarrow} \neq \emptyset$ and $Tr(\langle M'_2, c_1[\vec{a'_1}]; c_2[\vec{a'_2}]\rangle)^{\Downarrow} \neq \emptyset$ then $\forall t'_1 \in Tr(\langle M'_1, c_1[\vec{a'_1}]; c_2[\vec{a'_2}]\rangle)^{\Downarrow}.\, \exists t'_2 \in Tr(\langle M'_2, c_1[\vec{a'_1}]; c_2[\vec{a'_2}]\rangle)^{\Downarrow}.\, t_1 \approx_{(C(A),\top_I)} t_2$ along with the symmetric condition where M'_1 and M'_2 are swapped (which is proved analogously). In the rest of the proof, we assume $Tr(\langle M'_1, c_1[\vec{a'_1}]; c_2[\vec{a'_2}]\rangle)^{\Downarrow} \neq \emptyset$ and $Tr(\langle M'_2, c_1[\vec{a'_1}]; c_2[\vec{a'_2}]\rangle)^{\Downarrow} \neq \emptyset$. Obviously, this implies $Tr(\langle M'_1, c_1[\vec{a'_1}]\rangle)^{\Downarrow} \neq \emptyset$ and $Tr(\langle M'_2, c_1[\vec{a'_1}]\rangle)^{\Downarrow} \neq \emptyset$.

Suppose $t'_1 = t'_{11} \cdot t'_{12}$ where $t'_{11} \in Tr(\langle M'_1, c_1[\vec{a'_1}]\rangle)^{\Downarrow}$ and t'_{11} terminates in some state N'_1, and $t'_{12} \in Tr(\langle N'_1, c_2[\vec{a'_2}]\rangle)^{\Downarrow}$. By Proposition 11, $\langle M_1, c_1[\vec{a_1}]\rangle \approxeq_{(C(A),\top_I)} \langle M_2, c_1[\vec{a_1}]\rangle$. By the induction hypothesis we get $\langle M'_1, c_1[\vec{a'_1}]\rangle \approx_{(C(A),\top_I)} \langle M'_2, c_1[\vec{a'_1}]\rangle$.

In particular, $\exists t'_{21} \in Tr(\langle M'_2, c_1[\vec{a'_1}]\rangle)^{\Downarrow}. \, t'_{11} \approx_{(C(A),\top_I)} t'_{21}$. This means that there exists some N'_2 so that $\langle M'_1, c_1[\vec{a'_1}]\rangle \Downarrow N'_2$, corresponding to trace t'_{21}, and $N'_1 =_{(C(A),\top_I)} N'_2$.

By applying Proposition 12 twice, there exist $u_1 \in Tr(\langle M_1, c[\vec{a}]\rangle)$ and $u_2 \in Tr(\langle M_2, c[\vec{a}]\rangle)$ where $t'_{11} \sim_\ell u_1$ and $t'_{21} \sim_\ell u_2$ for all such ℓ that $\ell \in H_I$. Note that this implies that nondeterminism due to endorsement is resolved in the same way in both u_1 and u_2. We assume $\langle M_1, c_1[\vec{a_1}]\rangle \Downarrow N_1$, corresponding to trace u_1 and $\langle M_2, c_1[\vec{a_1}]\rangle \Downarrow N_2$, corresponding to trace u_2, for some N_1 and N_2. Because $t'_{11} \sim_\ell u_1$, $t'_{21} \sim_\ell u_2$, and $t'_{11} \sim_\ell t'_{21}$, we conclude $N_1 =_\ell N_2$ for all $\ell \in LH$.

The type system guarantees that assignments to variables in LL in $c_1[\vec{a_1}]$ may only depend on variables in LH and LL. This leads to $N_1 =_\ell N_2$ for all $\ell \in LL$. Thus, we have $N_1 =_{(C(A),\top_I)} N_2$ and therefore $\langle N_1, c_2[\vec{a_2}]\rangle \approxeq_{(C(A),\top_I)} \langle N_2, c_2[\vec{a_2}]\rangle$ by Proposition 11. By the induction hypothesis we have $\langle N'_1, c_2[\vec{a'_2}]\rangle \approx_{(C(A),\top_I)} \langle N'_2, c_2[\vec{a'_2}]\rangle$. By construction, $Tr(\langle N'_2, c_2[\vec{a'_2}]\rangle)^{\Downarrow} \neq \emptyset$. Connecting the traces for c_1 and c_2, we construct $t'_2 \in Tr(\langle M'_2, c_1[\vec{a'_1}]; c_2[\vec{a'_2}]\rangle)$ such that $t'_1 \approx_{(C(A),\top_I)} t'_2$. This implies $\langle M'_1, c_1[\vec{a'_1}]; c_2[\vec{a'_2}]\rangle \approx_{(C(A),\top_I)} \langle M'_2, c_1[\vec{a'_1}]; c_2[\vec{a'_2}]\rangle$. □

Theorem 5 is a simple corollary of the above proposition:

Proof (Theorem 5). By Proposition 13, setting $M'_1 = M_1$ and $M'_2 = M_2$. □

Below we consider two examples of typable and, thus, secure programs that involve both declassification and endorsement.

7.3 Password update example revisited

The first example is a variant of the password update code in which the requirement that the old and new passwords have high integrity is explicitly lifted (the assumption, in this case, is that checking the old password provides sufficient integrity assurance). Under the simple lattice $\mathcal{L}_{LH}$:

```
$LH \vdash$ update($pwdI, salt, oldP, newP, hashR, matchR$)
      $=$ $oldH :=$ endorse($oldP, LH$);
         $newH :=$ endorse($newP, LH$);
         match($pwdI, salt, oldH, hashR, matchR$);
         if $matchR$
            then $pwdI =$ hash($newH, salt$)
            else skip
```

which enables the following typing for password update:

The users modify a password:
$LH \vdash \texttt{update}(pwdI, salt, oldP, newP, hR, mR) : LH \times LH \times HL \times HL \times LH \times LH$

Under this typing, the above variant of `update` satisfies qualified robustness by Theorem 5.

7.4 Battleship game example

The second example is based on the game of Battleship, an example used by Zheng et al. [54]. Initially, two players place ships on their grid boards in secret. During the game they try to destroy each other's ships by firing shots at locations of the opponent's grid. On each move, the player making a shot learns whether it hit a ship or not. The game ends when all squares containing a player's ships are hit. It is critical to the security of a battleship implementation that information is disclosed one location at a time. Because the locations are initially secret, this disclosure must happen through declassification. However, a malicious opponent should not be able to hijack the control over the declassification mechanism to cause additional leaks about the secret state of the board. On the other hand, the opponent does have some control over what is disclosed because the opponent picks the grid location to hit. To allow the opponent to affect declassification in this way, `endorse` can be used to express the idea that any move by the opponent is acceptable.

Without loss of generality, let us consider the game from the viewpoint of one player only. The security classes can again be modeled by the simple lattice $\mathcal{L}_{LH}$ with $A = LL$. Consider the following core fragment of the main battleship program loop:

```
while $not\_done$ do
    $[\bullet_1]$;
    $m_2' :=$ endorse$(m_2, LH)$;
    $s_1 := apply(s_1, m_2')$;
    $m_1' := get\_move(s_1)$;
    $m_1 :=$ declassify$(m_1', LH)$;
    $not\_done :=$ declassify$(not\_final(s_1), LH)$;
    $[\bullet_2]$
```

We suppose that s_1 stores the first player's state (the secret grid and the current knowledge about the opponent) where $\Gamma(s_1) \in HH$. While the game is not finished the program gets a move from the opponent, computed in $[\bullet_1]$ and stored in m_2 where $\Gamma(m_2) \in LL$. In order to authorize the opponent to decide what location of s_1 to disclose, the move m_2 is endorsed in the assignment to m_2' where $\Gamma(m_2') \in LH$. The state s_1 is updated by a function $apply$. Then the first player's move m_1' (where $\Gamma(m_1') \in HH$) is computed using the current state. This move includes information about the location to be disclosed to the attacker. Hence, it is declassified to variable m_1 (where $\Gamma(m_1) \in LH$) before the actual disclosure, which takes place in $[\bullet_2]$. The information whether the game is finished (which determines when to leave the main loop) is public: $not\_done \in LH$. Hence, when updating not_done, the value of $not\_final(s_1)$ is downgraded to LH.

Clearly, this program is typable. Hence, from Theorem 5 we know that no more secret information is revealed than intended.

7.5 Observations on the semantic treatment of endorsement

The "scrambling" of the value of an endorsed expression allows for lifting the integrity of the expression. While such scrambling is based on a known approach to ignoring the difference between (high-confidentiality) values [21], it is worth highlighting some effects of scrambling when using it for ignoring the difference between low-integrity values in the presence of endorsement. We illustrate these effects by two examples.

Consider the following command $c[\bullet]$:

$$\begin{array}{l} [\bullet]; \\ y_{LL} := y_{LL} \;\texttt{mod}\; 2; \\ x_{LH} := \texttt{endorse}(y_{LL}, LH); \\ \texttt{if}\; x_{LH} > 42 \;\texttt{then}\; v_{LH} := \texttt{declassify}(w_{HH}, LH) \;\texttt{else skip}; \\ \texttt{if}\; z_{LL} \;\texttt{then}\; v_{LH} := \texttt{declassify}(w_{HH}, LH) \;\texttt{else skip} \end{array}$$

Intuitively, this program is insecure because the decision whether w_{HH} is leaked into v_{LH} is in the hands of the attacker. The `then` branch in the first `if` command is not reachable when the program is run. However, according to the scrambling semantics, x_{LH} might very well be greater than 42 after the endorsement. Hence, the first occurrence of $v_{LH} := \texttt{declassify}(w_{HH}, LH)$ is reachable under the scrambling semantics. The occurrence masks the real leak—the second occurrence of $v_{LH} := \texttt{declassify}(w_{HH}, LH)$—that the attacker has control over. To see that the program satisfies qualified robustness, observe that for initial memories M_1 and M_2 that are different in w_{HH} we have $\langle M_1, c[a]\rangle \not\approx_{(C(A),\top_I)} \langle M_2, c[a]\rangle$ for all attacks a because of the first occurrence of $v_{LH} := \texttt{declassify}(w_{HH}, LH)$.

Clearly, the scrambling treatment of endorsement may change the reachability of program commands. This example highlights that reachability can be affected in a way that makes intuitively insecure programs secure. Note that endorsement as scrambling merely reproduces arbitrary changes the attacker can make to the low-integrity memory before the endorse. If these changes are indeed arbitrary (which can be ensured by placing holes for low-integrity code immediately before occurrences of `endorse`) then reachability is not changed by the scrambling treatment of endorse, which allows for recovering from this undesirable effect. In any case, note that the above program is rightfully rejected by the type system because the second `if` command fails to type-check.

Consider another example command:

$$\begin{array}{l} [\bullet]; \\ x_{HH} := \texttt{endorse}(y_{HL}, HH); \\ v_{HH} := \texttt{endorse}(w_{HL}, HH); \\ \texttt{if}\; c_{LL} \;\texttt{then}\; z_{LH} := \texttt{declassify}(x_{HH}, LH) \;\texttt{else}\; z_{LH} := \texttt{declassify}(v_{HH}, LH) \end{array}$$

This program is intuitively insecure because the attacker controls what is leaked by choosing c_{LL}. However, the values of y_{HL} and w_{HL} are "forgotten" (and equalized) before they are written into x_{HH} and v_{HH}, respectively, because of the scrambling semantics of endorsement. Qualified robustness classifies the program as secure. Indeed, any outcome for z_{LH} is possible independently of the initial memories and attacks inserted in the hole. Again, the above program is rightfully rejected by the type system because the `if` command fails to typecheck.

The above examples illustrate that qualified robustness is in some cases more permissive than desired. An end-to-end strengthening of qualified robustness that is capable of rejecting these corner cases is an intriguing topic for future work.

8 Related work

Protecting confidential information in computer systems is an important problem that has been studied from many angles. This work has focused on language-based security, which has its roots in Cohen and Denning's work [11, 13, 15]. See the survey by Sabelfeld and Myers [39] for an overview of the language-based approach to information flow. With respect to declassification, the research most directly related to the topic of this paper is surveyed in Sabelfeld and Sands' work [43], which has a detailed comparison of many downgrading models. Below, we discuss the connections most relevant to this work.

Related to this paper is Myers' and Liskov's *decentralized label model* [32], which provides a rich policy language that includes a notion of *ownership* of the policy. Downgrading a principal's policy requires its authority. The decentralized label model has been implemented in the Jif compiler [33]. Work by Zdancewic and Myers [51, 50] introduced notions of robustness and an enforcement mechanism for robustness, as discussed in the introduction. The major contribution of this work is that it connects a (new) semantic security condition for robustness directly to a type-based enforcement mechanism; this connection has not been previously established.

The key observation behind the robustness approach is that to understand whether information release is secure, it is necessary to take into account the integrity of the decision to release information. Other work on controlled information release has not taken into account integrity, and as a result, does not address systems containing untrustworthy components.

Another approach to ensuring that only the intended information is released is to write a specification restricting which information should be released. Giambiagi's and Dam's work on *admissible flows* [12, 18] is an example of the specification approach. Their security condition requires that the implementation reveal no more information than the specification of a protocol. *Relaxed noninterference* [25] is a lighter-weight method for specifying what information should be released. It is a strict generalization of pure noninterference that gives an extensional definition of exactly what information may be declassified. *Delimited information release* [40] allows these specifications to be given explicitly, via "escape hatches". These escape hatches are represented by expressions that might legitimately leak sensitive information. Delimited release guarantees that the program may leak no more information than the escape hatch expressions alone.

The specification approach has value because it says directly what may be released. However, in the presence of an attacker, even the specification may contain subtle vulnerabilities that require careful analysis; we expect robustness to be a useful tool for analyzing such specifications because it captures an important property that specifications should ordinarily satisfy.

Despite their importance, general downgrading mechanisms and their related security policies are not yet thoroughly understood. *Partial* information flow policies [11, 21, 42] weaken noninterference by partitioning the domain of confidential information into subdomains such that noninterference is required only within each subdomain. *Quantitative* information flow policies [14, 27, 9] restrict the information-theoretic quantity of downgraded information. Policies for *complexity-theoretic* information flow [22, 23] aim to prevent complexity-bound attackers from laundering information through programs that declassify the result of encryption. *Approximate noninterference* [16] relaxes noninterference by allowing confidential processes to be (in a probabilistic sense) approximately similar for the attacker.

Intransitive noninterference policies [38, 34, 37, 28] alter noninterference so that the interference relation is intransitive. Certain information flows are designated as downward and must pass through trusted system components. The language-based work by Bevier et al. on *controlled interference* [4] similarly allows policies for information released to a set of *agents*. Bossi et al. [6] offer a framework where downgrading is characterized by a generalization of unwinding relations. Mantel and Sands [29] consider the problem of specifying and enforcing intransitive noninterference in a multi-threaded language-based setting. Such policies are attractive, but the concept of robustness in this paper is largely orthogonal to intransitive noninterference (cf. the discussion on the laundering attack in Section 6), suggesting that it may be profitable to combine the two approaches.

Volpano and Smith [47] consider a restricted form of declassification, in the form of a built in $\texttt{match}_h(l)$ operation, intended to model the password example. They require h to be an unmodifiable constant when introducing $\texttt{match}_h(l)$, but this means that password may not be updated. Volpano's subsequent work [45] models one-way functions by primitives $f(h)$ and a `match`-like $f(h) = f(r)$ (where h and r correspond to the password and user query, respectively), which are used in a hash-based password checking. However, the assumption is that one-way functions may not be applied to modifiable secrets. Both studies argue that one could do updates in an independent program that satisfies noninterference. However, in general this opens up possibilities for laundering attacks. The `match`, $f(h)$, and $f(h) = f(r)$ primitives are less general than declassification.

The alternate semantics for `endorse` that are used to define the qualified robustness are inspired by the "havoc" semantics that Joshi and Leino used to model confidentiality [21].

9 Conclusions

This paper presents a language-based robustness property that characterizes an important aspect of security policies for information release: that information release mechanisms cannot be exploited to release more information than intended. The language-based security condition generalizes the earlier robustness condition of Zdancewic and Myers [51] by expressing the property in a language-based setting: specifically, for a simple imperative programming language. Second, untrusted code and data are explicitly part of the system rather than an aspect that appears only when there is an active attacker. This removes an artificial modeling limitation of the earlier robustness condition. Third, a generalized security condition called *qualified robustness* is introduced that grants untrusted code a limited ability to affect information release.

The key contribution of the paper is a demonstration that robustness can be enforced by a compile-time program analysis based on a simple type system. A type system is given that tracks data confidentiality and integrity in the imperative programming language, similarly to the type system defined in [50]; this paper takes the new step of proving that all well-typed programs satisfy a language-based robustness condition. In addition, the analysis is generalized to accommodate untrusted code that is explicitly permitted to have a limited effect over information release.

Robust declassification appears to be a useful property for describing a variety of systems. The work was especially motivated by the work on Jif/split, a system that transforms programs to run securely on a distributed system [53, 54]. Jif/split automatically splits a sequential program into fragments that it assigns to hosts with sufficient trust levels. This system maps naturally onto the formal framework described here; holes correspond to low-integrity computation that can be run on untrusted host machines. In general, being an A-attack (cf. Definition 10) is required for a program to be placed on an A-trusted host. Thus, the results of this paper are a promising step toward the goal of establishing the robustness of the Jif/split transformation for the full Jif/split language.

Much further work is possible in this area. The security model in this paper assumes, as is common, a termination-insensitive and timing-insensitive attacker. However, we anticipate no major difficulties in adapting the robustness model and the security type system to enforce robust declassification for termination-sensitive or timing-sensitive attacks. These are worthwhile directions for future investigations.

Although we have argued that the sequential programming model is reasonable, and certainly a reasonable starting point, considering the impact of concurrency and concurrent attackers would be an important generalization. Combining robust declassification with other security properties related to downgrading (such as intransitive noninterference or relaxed noninterference) would also be of interest.

Acknowledgments

Thanks are due to David Naumann, David Sands, Stephen Chong, Daniel Hedin, Fred Schneider, and the anonymous reviewers for their useful feedback.

This work was supported by the Department of the Navy, Office of Naval Research, under ONR Grant N00014-01-1-0968. Any opinions, findings, conclusions, or recommendations contained in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research. Myers was supported by the National Science Foundation under Grant Nos. CNS-0430161, CNS-0208642 and CNS-0133302, and by an Alfred P. Sloan Research Fellowship. Zdancewic was supported by the National Science Foundation under Grant Nos. CNS-0346939 and CCR-0311204.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, Jan. 1999.

[2] J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, Jan. 2000.

[3] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proc. IEEE Computer Security Foundations Workshop*, pages 253–267, June 2002.

[4] W. R. Bevier, R. M. Cohen, and W. D. Young. Connection policies and controlled interference. In *Proc. IEEE Computer Security Foundations Workshop*, pages 167–176, June 1995.

[5] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, Apr. 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[6] A. Bossi, C. Piazza, and S. Rossi. Modelling downgrading in information flow security. In *Proc. IEEE Computer Security Foundations Workshop*, pages 187–201, June 2004.

[7] S. Chong and A. C. Myers. Language-based information erasure. In *Proc. IEEE Computer Security Foundations Workshop*, pages 241–254, June 2005.

[8] T. Chothia, D. Duggan, and J. Vitek. Type-based distributed access control. In *Proc. IEEE Computer Security Foundations Workshop*, pages 170–186, 2003.

[9] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. In *Proc. Quantitative Aspects of Programming Languages*, volume 59 of *ENTCS*. Elsevier, 2002.

[10] M. R. Clarkson, A. C. Myers, and F. B. Schneider. Belief in information flow. In *Proc. IEEE Computer Security Foundations Workshop*, June 2005.

[11] E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

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

[13] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.

[14] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, MA, 1982.

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

[16] A. Di Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *Proc. IEEE Computer Security Foundations Workshop*, pages 1–17, June 2002.

[17] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 130–140, May 1997.

[18] P. Giambiagi and M. Dam. On the secure implementation of security protocols. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 144–158. Springer-Verlag, Apr. 2003.

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

[20] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, Jan. 1998.

[21] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.

[22] P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. European Symp. on Programming*, volume 2028 of *LNCS*, pages 77–91. Springer-Verlag, Apr. 2001.

[23] P. Laud. Handling encryption in an analysis for secure information flow. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 159–173. Springer-Verlag, Apr. 2003.

[24] P. Li, Y. Mao, and S. Zdancewic. Information integrity policies. In *Proceedings of the Workshop on Formal Aspects in Security & Trust (FAST)*, Sept. 2003.

[25] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 2005.

[26] P. Li and S. Zdancewic. Unifying confidentiality and integrity in downgrading policies. In *Proc. of the LICS'05 Affiliated Workshop on Foundations of Computer Security (FCS)*, July 2005.

[27] G. Lowe. Quantifying information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 18–31, June 2002.

[28] H. Mantel. Information flow control and applications—Bridging a gap. In *Proc. Formal Methods Europe*, volume 2021 of *LNCS*, pages 153–172. Springer-Verlag, Mar. 2001.

[29] H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. In *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 129–145. Springer-Verlag, Nov. 2004.

[30] J. K. Millen. Covert channel capacity. In *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA, 1987.

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

[32] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[33] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001–2003.

[34] S. Pinsky. Absorbing covers and intransitive non-interference. In *Proc. IEEE Symp. on Security and Privacy*, pages 102–113, May 1995.

[35] F. Pottier and S. Conchon. Information flow inference for free. In *Proc. ACM International Conference on Functional Programming*, pages 46–57, Sept. 2000.

[36] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, Jan. 2002.

[37] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proc. IEEE Computer Security Foundations Workshop*, pages 228–238, June 1999.

[38] J. M. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, 1992.

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

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

[41] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.

[42] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher Order and Symbolic Computation*, 14(1):59–91, Mar. 2001.

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

[44] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, Jan. 1998.

[45] D. Volpano. Secure introduction of one-way functions. In *Proc. IEEE Computer Security Foundations Workshop*, pages 246–254, July 2000.

[46] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3):231–253, Nov. 1999.

[47] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 268–276, Jan. 2000.

[48] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[49] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, 1993.

[50] S. Zdancewic. A type system for robust declassification. In *Proc. Mathematical Foundations of Programming Semantics*, ENTCS. Elsevier, Mar. 2003.

[51] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.

[52] S. Zdancewic and A. C. Myers. Secure information flow and CPS. In *Proc. European Symp. on Programming*, volume 2028 of *LNCS*, pages 46–61. Springer-Verlag, Apr. 2001.

[53] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3):283–328, Aug. 2002.

[54] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 236–250, May 2003.