Enforcing Robust Declassification

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
Dept. 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 perform intentional release of sensitive information as part of their correct functioning and therefore violate noninterference. To control information flow while permitting intentional information release, some systems have a downgrading or declassification mechanism. A major danger of such a mechanism is 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 exploited 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 some appealing properties as a security enforcement mechanism. Unlike access controls, they track the propagation of information and prevent sensitive information from being released publicly, regardless of how information is transformed by the system. Dually, information flow controls may be used to enforce data integrity. One common formal underpinning of these mechanisms is the *noninterference* security property [16], which imposes an end-to-end requirement on the behavior of the system: sensitive data cannot affect public data. 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 downgrading or declassification of sensitive information (e.g., [13, 24, 5]). These mechanisms are inherently unsafe and there is the possibility that a downgrading channel that is part of a larger system may be exploited to release information in a way that was not intended.

Given that noninterference is not satisfied, we would like to know that the information release occurs in accordance with some presumably more flexible security policy. However, it seems to be difficult in general to express these policies precisely and even more difficult to show that systems satisfy them. 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 defined by Zdancewic and Myers [43]. The intuition is that although the system may release information, an attacker should have no control over what information is released. More generally, in a system that is separated into untrusted and trusted components, the untrusted components should not be able to affect information release. Zdancewic and Myers captured this idea formally in the context of a state transition system, but offered no practical way to analyze whether a program satisfied robust declassification.

This paper generalizes the previous work on robustness in three ways. First, it shows how to express the property in a language-based setting; specifically, for a simple imperative programming language. Second, it generalizes the

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

In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, Pacific Grove, CA, June 28–30, 2004.

property so that—unlike the earlier robustness property—untrusted code and data are explicitly part of the system rather than appearing only when there is an active attacker. Third, it introduces a security guarantee called *qualified robustness* that provides untrusted code with 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 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 by Zdancewic [42]. This paper also takes the new step of proving that all well-typed programs satisfy the language-based robustness condition it defines.

The rest of the paper is structured as follows. Section 2 presents some of the basic assumptions and models used for this work, including a simple imperative language with an explicit declassification construct that downgrades confidentiality levels. Section 3 presents and generalizes the robustness condition in this language-based setting, and gives some motivating code fragments that are used as running examples. Section 4 presents a security type system for the imperative language. This type system tracks both the confidentiality and integrity of data and imposes integrity requirements on declassification operations. It also ensures that any well-typed program satisfies the robust declassification condition. Section 5 presents more detailed examples and shows how the robust declassification condition gives insight into program security. Section 6 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 7 discusses related work, and Section 8 concludes.

2. Language and attacker model

2.1. 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 lattices*, $\mathcal{L}_C$ and $\mathcal{L}_I$, with orderings $\sqsubseteq_C$ and $\sqsubseteq_I$, respectively. If $C \sqsubseteq_C C'$ ($I \sqsubseteq_I I'$) then data at level C (I) is no more confidential (no less trustworthy) than data at level C' (I'). 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), where we denote the confidentiality and integrity parts of ℓ by $C(\ell)$ and $I(\ell)$, respectively. 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.

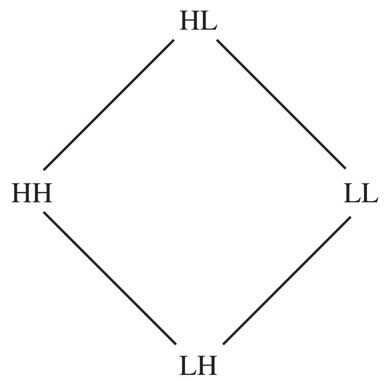

Figure 1. Security lattice $\mathcal{L}_{LH}$.

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.

2.2. Attacker model

The goal of this paper is to characterize programs in which untrusted components cannot improperly affect what information is released. These untrusted components are assumed to be under the control of some attacker. This is a very general model of the system. This attacker may in fact be an ordinary user, in which case the goal is to understand whether program users can cause unintended information release, perhaps by providing unexpected inputs. Alternatively, as in the work on secure program partitioning [45, 46], the system might be a distributed program in which some of the program code runs on untrusted hosts and is assumed to be controlled by a malicious attacker.

In all these scenarios, the attacker is described by a confidentiality level C_A representing the confidentiality of data the attacker is expected to be able to read, and an integrity level I_A defining the integrity of data 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 $\mathcal{L}$.

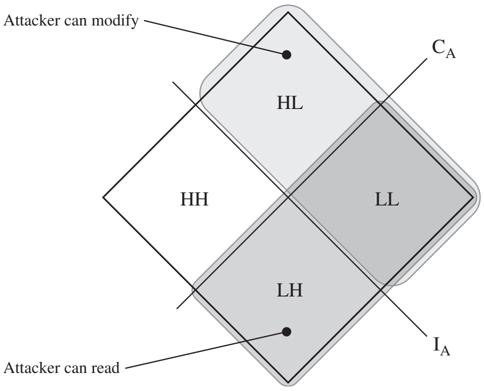

Figure 2. Attacker's view of a general lattice.

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

$$\begin{array}{cc} LH \sim L_C \cap H_I & HH \sim H_C \cap H_I \\ LL \sim L_C \cap L_I & HL \sim H_C \cap L_I \end{array}$$

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 $\mathcal{L}$.

2.3. 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., [40, 2]), and its semantics are largely standard (cf. [41]).

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

$$e ::= val \mid v \mid e_1 \ \mathtt{op}\ e_2 \mid \mathtt{declassify}(e, \ell)$$

$$\begin{array}{rl} c ::= & \mathtt{skip} \mid v := e \mid c_1; c_2 \\ & \mid \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \mid \mathtt{while}\ e\ \mathtt{do}\ c \end{array}$$

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

The *security environment* $\Gamma : Var \to \mathcal{L}$ describes the type of each program variable as a security level. The security lattice and security environment together constitute a *security policy*, specifying that information flow from a variable v_1 to a variable v_2 is allowed only if $\Gamma(v_1) \sqsubseteq \Gamma(v_2)$.

The only non-standard language expression is the construct $\mathtt{declassify}(e, \ell)$, which *declassifies* the security level of the expression 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 $M : Var \to Val$ from variables to values) and a command (or expression) c. A transition from configuration $\langle M, c\rangle$ to configuration $\langle M', c'\rangle$ is denoted by $\langle M, c\rangle \longrightarrow \langle M', c'\rangle$. A transition from configuration $\langle M, c\rangle$ to a terminating configuration with memory M' is denoted by $\langle M, c\rangle \longrightarrow M'$. As usual, $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$. Configuration $\langle M, c\rangle$ *terminates* in M' if $\langle M, c\rangle \longrightarrow^* M'$, 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*, written $\langle M, c\rangle \Uparrow$. We assume that operations used in expressions are total, and, hence, expression configurations always terminate (while command configurations might diverge). The *trace* $Tr(\langle M, c\rangle)$ of the execution of configuration $\langle M, c\rangle$ is the sequence $[M, M', M'', \dots]$ of memories extracted from the sequence of configurations $\langle M, c\rangle \longrightarrow \langle M', c'\rangle \longrightarrow \langle M'', c''\rangle \longrightarrow \dots$. Similarly to configurations, a trace t terminates (in M), written $t \Downarrow$ ($t \Downarrow M$) when t is finite (and the last memory in t is M); t diverges, written $t \Uparrow$, if t is infinite.

3. Robustness condition

A common way of specifying confidentiality is as noninterference [16], a security property that says that inputs of high confidentiality do not affect outputs of lower confidentiality. Recent work on language-based security (e.g., [40, 1, 17, 36, 38, 2, 34, 28, 35, 44, 3, 29]) has used various definitions of noninterference as the definition of security. However, noninterference cannot characterize the security of a program designed to declassify confidential information as part of its proper functioning. Therefore we propose a security condition that captures important aspects of the information release policy. This security condition is based on robust declassification [43], which intuitively states that declassification may not be abused by the attacker to gain more knowledge about secrets than intended. In Section 6, we also consider how *endorsement* (a dual primitive that upgrades the *integrity* of data) affects the security characterization.

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. Define the *projection* $t|_\ell$ of trace t to the security level ℓ by the trace consisting of the sequence of memories restricted to variables at or below ℓ. Formally, $[M_1, \ldots, M_n, \ldots]|_\ell = [M_1|_\ell, \ldots, M_n|_\ell \ldots]$. Because computation steps can be observed only if they make changes to the observable part of memory, we identify traces up to *high-stuttering* with respect to a security level ℓ. Traces t_1 and t_2 for configurations $\langle M_1, c_1\rangle$ and $\langle M_2, c_2\rangle$ are related ($t_1 \sim_\ell t_2$) if $M_1 =_\ell M_2$ and the subsequences (of t_1 and t_2) of memories resulting from ℓ-observable assignments in c_1 and c_2 are ℓ-indistinguishable. 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 2. *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 allows one trace to end prematurely; strong indistinguishability requires the termination of both configurations so that the traces remain related throughout their entire execution.

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 3 (Noninterference). *A command c satisfies* noninterference *under Γ if*

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

Because noninterference flatly rejects dependencies at any security level, it is overly restrictive for many systems. (However, it is still useful for reasoning about fragments of a larger program.) As described by Zdancewic and Myers [43], robust declassification ensures that declassification cannot be abused by the attacker. More precisely, a system is secure if an active attacker (who can observe and modify a part of the system state) may not learn more sensitive information than a passive attacker (who can merely observe visible data). Here we model both kinds of attackers relative to a 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) whereas an active A-attacker may modify data at or above I_A (i.e., at or above $(\bot_C, I_A)$ in the product lattice). In general, an attacker may run any program satisfying a combination of conditions on what data can be read and modified. We call such programs *fair attacks*.

Definition 4. *A command a is a* fair attack *if it is formed according to the following grammar (for some $\ell \in LL$):*

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

Attacker-controlled low-integrity computation may be interspersed with 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 *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 attack is thus a vector of such code fragments.

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 [45].

High-integrity contexts are defined formally as follows:

Definition 5. *High-integrity contexts, or commands with holes, $c[\vec{\bullet}]$ are defined by extending the command grammar from Definition 1 with:*

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

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 $c[a']$ may not reveal any secrets apart from what the attacker already knows from observations about $c[a]$. This is formally expressed in the following definition.

Definition 6 (Robustness). *Command* $c[\vec{\bullet}]$ *has* robustness *with respect to fair attacks if*

$$\forall M_1, M_2, \vec{a}, \vec{a'}.\ \langle M_1, c[\vec{a}]\rangle \cong_A \langle M_2, c[\vec{a}]\rangle \implies \langle M_1, c[\vec{a'}]\rangle \approx_A \langle M_2, c[\vec{a'}]\rangle$$

As noted, the attacker can observe data below the lattice point $(C_A, \top_I)$. This level is used for the relations $\cong_A$ and $\approx_A$, requiring equality for the low-confidentiality parts of memories and configurations, respectively. Note that $\langle M_1, c\rangle \cong_A \langle M_2, c\rangle$ implies that $M_1 =_A M_2$ by Definition 2.

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 who inserts `skip` into every hole. 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):

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

This program would be robust if a in Definition 6 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 6, 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.

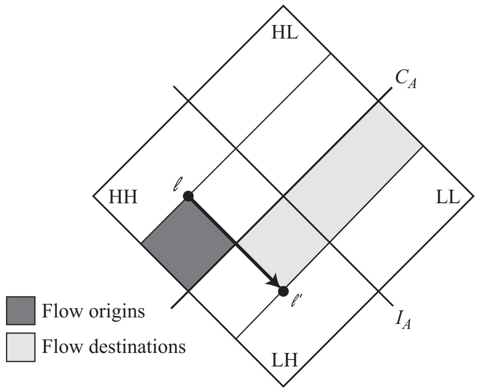

Figure 3. Effects of declassification.

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

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

and

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

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 have robustness because the attacker may not influence what is declassified (by assigning to y_{HH} in the former program) or by manipulating 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}. However, the program

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

is rejected because the attacker might affect what is declassified or when it is declassified, by controlling the decision variable x_{LL}.

4. Security type system for robustness

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 type. As we show later in this section, any program that is well-typed according to these rules also satisfies the robustness property. We write $\Gamma, pc \vdash e : \ell$ to mean that an expression e has type ℓ under an environment Γ and a context pc. For commands, we write $\Gamma, pc \vdash c$ if command c is well-typed under an environment Γ and a context pc.

$$\frac{}{\Gamma, pc \vdash val : \ell}$$

$$\frac{\Gamma(v) = \ell}{\Gamma, pc \vdash v : \ell}$$

$$\frac{\Gamma, pc \vdash e : \ell \quad \Gamma, pc \vdash e' : \ell}{\Gamma, pc \vdash e \;\mathtt{op}\; e' : \ell}$$

$$\frac{\Gamma, pc \vdash e : \ell \quad pc' \sqsubseteq pc \quad \ell \sqsubseteq \ell'}{\Gamma, pc' \vdash e : \ell'}$$

$$\frac{}{\Gamma, pc \vdash \mathtt{skip}}$$

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

$$\frac{\Gamma, pc \vdash c_1 \quad \Gamma, pc \vdash c_2}{\Gamma, pc \vdash c_1; c_2}$$

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

$$\frac{\Gamma, pc \vdash e : \ell \quad \Gamma, \ell \sqcup pc \vdash c}{\Gamma, pc \vdash \mathtt{while}\; e \;\mathtt{do}\; c}$$

$$\frac{\Gamma, pc \vdash c \quad pc' \sqsubseteq pc}{\Gamma, pc' \vdash c}$$

$$\frac{\begin{array}{c}\Gamma, pc \vdash e : \ell' \qquad \ell \sqcup pc \sqsubseteq \Gamma(v) \\ I(\ell) = I(\ell') \quad I(pc), I(\ell') \in H_I\end{array}}{\Gamma, pc \vdash v := \mathtt{declassify}(e, \ell)}$$

Figure 4. Typing rules.

The typing rules control the information flow due to assignments and control flow in a largely standard fashion (cf. [40]). However, the key rule governing uses of declassification is non-standard, though similar to that proposed by Zdancewic [42] (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 in a high-integrity context (*pc*). The effect of this rule can be visualized by considering the lattice depicted in Figure 3. The figure includes an arrow corresponding to a declassification from security level ℓ to level ℓ'. Restricting the area of possible flow origins (below ℓ) to the high-integrity area of the lattice prevents the attacker (who controls the low-integrity area of the lattice) from compromising the declassification mechanism.

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 fair attacks.

Definition 7. *A command a is an A-*attack *under Γ 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 $x_{LL} := y_{LL}$, `while` x_{HL} `do skip`, and (a harmless attack) `skip`. On the other hand, programs $x_{HH} := y_{LH}$ and $x_{LH} :=$ `declassify`(y_{LH}, LH) are not attacks as they manipulate high-integrity data. Note that programs $x_{LL} :=$ `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. Recall the partition of data according to the confidentiality (H_C and L_C) and integrity (L_I and H_I) levels from Section 2.2. The following propositions provide some useful (and straightforward to prove) properties of attacks.

Proposition 1. *A fair attack is also an A-attack.*

Proposition 2. *An A-attack under Γ (i) does not have occurrences of assignments to high-integrity variables (such v that $\Gamma(v) \in H_I$); and (ii) satisfies noninterference under Γ.*

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.*

This result is proved with a straightforward induction on the evaluation of *c* [40].

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*

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

As for the confidentiality part, we show the key result of this paper: typable programs satisfy robust declassification 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 into high-confidentiality environments. This is achieved by defining a suitable typing rule for holes:

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

This rule allows program contexts $c[\vec{\bullet}]$ to be type-checked. The robustness result is:

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

The proof is found in Appendix A. It is a straightforward induction on the structure of $c[\vec{\bullet}]$.

It is worth clarifying the relation of the type system to that defined by Zdancewic [42]. While both type systems 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 [42]. 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 ensures that the program above is rejected.

5. 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. The 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 the *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 a protection against dictionary attacks. For a successful login, the user is required to provide a query such that the hash of the string and salt 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 $\texttt{hash}(pwd, salt)$ 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 $C_{salt} \in L_C$). The command $\texttt{match}(pwdI, salt, pwd, hashR, matchR)$ 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.

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

where $C_{matchR} = C_{pwdI} \sqcup C_{salt}$, $I_{matchR} = I_{pwdI} \sqcup I$, $I = I_{pwd} \sqcup I_{salt}$; and $I, I(pc) \in H_I$. 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:
$LH \vdash \texttt{hash}(pwd, salt) : HH \times LH \to LH$

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

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} ($\Gamma(y_{LL}) = LL$):

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

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 $\langle M_1, c[a]\rangle \approxeq_A \langle M_2, c[a]\rangle$ (the `else` branch is taken regardless of x_{HH}) but $\langle M_1, c[a']\rangle \not\approxeq_A \langle M_2, c[a']\rangle$ (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:

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

where $C_{salt}(I_{salt} \sqcup I_{oldP} \sqcup I_{newP}) \sqsubseteq C_{pwdI} I_{pwdI}$ and $I_{salt}, I_{oldP}, I_{newP}, I(pc) \in H_I$. 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:
$LH \vdash \texttt{update}(pwdI, salt, oldP, newP, hR, mR) : LH \times LH \times HH \times HH \times LH \times LH$

6. 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:

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

There are two pieces of 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.

6.1. 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.

Two trace sets T_1 and T_2 are *indistinguishable up to* ℓ (written $T_1 \approx_\ell T_2$) if $\forall t_1 \in T_1.\, \exists t_2 \in T_2.\, t_1 \approx_\ell t_2$ & $\forall t_2 \in T_2.\, \exists t_1 \in T_1.\, t_1 \approx_\ell t_2$, i.e., for any trace from T_1 we can find a trace from T_2 so that if both terminate than they are indistinguishable up to ℓ and vice versa. We can now lift Definition 2 to multiple-trace semantics:

Definition 8. *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 9 (Qualified robustness). *Command $c[\vec{\bullet}]$ has* qualified robustness *with respect to fair attacks if*

$$\forall M_1, M_2, \vec{a}, \vec{a'}.\, \langle M_1, c[\vec{a}]\rangle \approxeq_A \langle M_2, c[\vec{a}]\rangle \implies \langle M_1, c[\vec{a'}]\rangle \approx_A \langle M_2, c[\vec{a'}]\rangle$$

Note the similarity of qualified robustness to the original robustness property from Definition 6. In fact, the difference is entirely contained in the generalized indistinguishability relations $\approxeq_A$ and $\approx_A$.

6.2. Enforcing qualified robustness

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

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

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*

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

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:

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

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 $\langle M_1, c[a]\rangle \simeq_A \langle M_2, c[a]\rangle$ but $\langle M_1, c[a']\rangle \not\approx_A \langle M_2, c[a']\rangle$) 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.*

A proof is sketched in Appendix B. Below we consider two examples of typable and, thus, secure programs that involve both declassification and endorsement.

6.3. 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 $\mathcal{L}_{LH}$:

$$\begin{array}{l} LH \vdash \texttt{update}(pwdI, salt, oldP, newP, hashR, matchR) \\ \quad = \ oldH := \texttt{endorse}(oldP, LH); \\ \qquad newH := \texttt{endorse}(newP, LH); \\ \qquad \texttt{match}(pwdI, salt, oldH, hashR, matchR); \\ \qquad \texttt{if}\ matchR \\ \qquad\quad \texttt{then}\ pwdI = \texttt{hash}(newH, salt) \\ \qquad\quad \texttt{else skip} \end{array}$$

which enables the following typing for password update:

The users modify a password:

$$\begin{array}{l} LH \ \vdash \ \texttt{update}(pwdI, salt, oldP, newP, hR, mR) \ : \\ LH \times LH \times HL \times HL \times LH \times LH \end{array}$$

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

6.4. Battleship game example

The second example is based on the game of Battleship, an example used by Zheng et al. [46]. 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 the 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:

$$\begin{array}{l} \texttt{while}\ not\_done\ \texttt{do} \\ \quad [\bullet_1]; \\ \quad m_2' := \texttt{endorse}(m_2, LH); \\ \quad s_1 := apply(s_1, m_2'); \\ \quad m_1' := get\_move(s_1); \\ \quad m_1 := \texttt{declassify}(m_1', LH); \\ \quad not\_done := \texttt{declassify}(not\_final(s_1), LH); \\ \quad [\bullet_2] \end{array}$$

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. 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 [7, 9, 11]. See the recent survey by Sabelfeld and Myers [32] for an overview of the language-based approach.

Related to this paper is Myers' and Liskov's work on the *decentralized label model* [25], which provides a rich policy language that includes a notion of *ownership* of the policy. Downgrading a principal's policy requires their authority. The decentralized label model has been implemented in the Jif compiler [26]. Work by Zdancewic and Myers [43, 42] also has similar goals to the work presented here, as discussed in the introduction. The major contribution of this work is that it connects a semantic security condition for robustness directly to a type-based enforcement mechanism; this connection has not been previously established.

Giambiagi's and Dam's work on *admissible flows* [8, 15] takes a similar approach to ours. Their security condition requires that the implementation reveal no more information than the specification of a protocol. This is appealing but the intended leaks are explicit in the syntax of the specification. In our approach, this is not necessary as robustness is expressed purely in terms of semantics.

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 [18]. They are also similar to some aspects of the generalization of noninterference proposed by Giacobazzi and Mastroeni [14], based on abstract interpretation; in particular, the abstraction that causes "deceptive" flows to be ignored.

Despite their importance, general downgrading mechanisms and their related security policies are not yet thoroughly understood. *Partial* information flow policies [7, 18, 35] weaken noninterference by partitioning the domain of confidential information into subdomains such that noninterference is required only within each subdomain. *Quantitative* information flow policies [10, 21, 6] restrict the information-theoretic quantity of downgraded information. *Complexity-theoretic* information flow policies [19, 20] facilitate preventing complexity-bound attackers from laundering information through programs that declassify the result of encryption. *Approximate noninterference* [12] relaxes noninterference by allowing confidential processes to be (in a probabilistic sense) approximately similar for the attacker.

Intransitive noninterference policies [31, 27, 30, 22] 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*. Mantel and Sands [23] 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 5), suggesting that it may be profitable to combine the two approaches.

Volpano and Smith [39] consider a restricted form of declassification, in the form of a built in $\mathtt{match}_h(l)$ operation, intended to model the password example. They require h to be an unmodifiable constant when introducing $\mathtt{match}_h(l)$, but this means that password may no be updated. Volpano's subsequent work [37] models one-way functions by primitives $f(h)$ and a `match`-like $f(h) = f(h)$ (where h and r correspond to the password and user query, respectively), which are used in a hash-based password checking. The assumption is however, 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 $f(h) = f(h)$ primitives are less general than declassification.

Recently, Sabelfeld and Myers have developed a model for *delimited information release* [33]. Delimited release allows a program to release information 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.

8. 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 [43] 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 [42]; 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 [45, 46]. 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 7) 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.

The security model in this paper assumes, as is common, a termination-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 attacks. This is a worthwhile direction for future investigations.

Much further work is possible in this area. 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) would also be of interest.

Acknowledgments

Thanks are due to David Naumann, David Sands, and Stephen Chong 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. This work was also supported by the National Science Foundation under Grant Nos. 0208642 and 0133302, and by an Alfred P. Sloan Research Fellowship.

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

[6] 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.

[7] 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.

[8] 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.

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

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

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

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

[13] 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.

[14] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 186–197, Jan. 2004.

[15] 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.

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

[17] 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.

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

[19] 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.

[20] 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.

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

[22] 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.

[23] H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. Draft, July 2003.

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

[25] 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.

[26] 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.

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

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

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

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

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

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

[33] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, LNCS. Springer-Verlag, 2004. To appear.

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

[35] 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.

[36] 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.

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

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

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

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

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

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

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

[44] 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.

[45] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. ACM Symp. on Operating System Principles*, pages 1–14, Oct. 2001.

[46] 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.

Appendix A

This appendix presents the proof of the main robustness result of the paper. If a command c is well-typed under Γ then it is robust with respect to the attacker-controlled code. The robustness theorem says that the attacker-controlled code may not increase the attacker's observations about the system. 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 due to the trace-level granularity of the security condition: the indistinguishability of configurations requires the indistinguishability of traces (up to high-stuttering).

Proposition 3. *If $\Gamma, pc \vdash c_1; c_2$ and $\langle M_1, c_1; c_2\rangle \cong_A \langle M_2, c_1; c_2\rangle$ then $\langle M_1, c_1\rangle \cong_A \langle M_2, c_1\rangle$. 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 $\langle N_1, c_2\rangle \cong_A \langle N_2, c_2\rangle$.*

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 if for some memory both programs terminate then they agree on high-integrity data (if the latter exist) at the end of computation. This is a form of noninterference of low-integrity code with high-integrity values.

Proposition 4. *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 attacks $\vec{a}$ and $\vec{a'}$ then $N(v) = N'(v)$ for all v such that $\Gamma(v) \in H_I$.*

Suppose we have a typable command 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 memory may make the behaviors distinguishable for the attacker. The key idea is that because declassification is not allowed in a low-integrity context, no change of a low-integrity value at the beginning of computation may reflect on low-confidentiality behavior of the computation.

Proposition 5. *If $\Gamma, pc \vdash c$ and $\langle M_1, c\rangle \cong_A \langle M_2, c\rangle$ for some M_1 and M_2 then for any value val and variable v so that $\Gamma(v) \in L_I$ we have $\langle M_1', c\rangle \approx_A \langle M_2', c\rangle$ where $M_1' = M_1[v \mapsto val]$ and $M_2' = M_2[v \mapsto val]$.*

We are now ready to prove Theorem 3.

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

Proof. If $H_I = \emptyset$ then declassification is disallowed by the typing rules, and the theorem follows form Theorem 1. In the rest of the proof we assume $H_I \neq \emptyset$. Induction on the structure of $c[\vec{\bullet}]$. Suppose that for some $c[\vec{a}]$ and memories M_1 and M_2 we have $\langle M_1, c[\vec{a}]\rangle \cong_A \langle M_2, c[\vec{a}]\rangle$ (which, in particular, implies $M_1 =_A M_2$). We need to show $\langle M_1, c[\vec{a'}]\rangle \approx_A \langle M_2, c[\vec{a'}]\rangle$ for all $\vec{a'}$. If $c[\vec{\bullet}]$ has the form `skip` or $v := e$ then the command has no holes, implying $c[\vec{a}] = c[\vec{a'}]$, which is a vacuous case. Note that this case covers (trusted) assignments with `declassify` in the right-hand side. The case $c[\vec{\bullet}] = [\bullet]$ is straightforward because by Proposition 2 attack a' satisfies noninterference. Structural cases on $c[\vec{a}]$ (where appropriate, we assume that $\vec{a}$ is split into two vectors $\vec{a_1}$ and $\vec{a_2}$):

$c_1[\vec{a_1}]; c_2[\vec{a_2}]$ We have $\langle M_1, c_1[\vec{a_1}]; c_2[\vec{a_2}]\rangle \cong_A \langle M_2, c_1[\vec{a_1}]; c_2[\vec{a_2}]\rangle$. By Proposition 3 we infer $\langle M_1, c_1[\vec{a_1}]\rangle \cong_A \langle M_2, c_1[\vec{a_1}]\rangle$. By the induction hypothesis we obtain $\langle M_1, c_1[\vec{a_1'}]\rangle \approx_A \langle M_2, c_1[\vec{a_1'}]\rangle$. If one, say $\langle M_1, c_1[\vec{a_1'}]\rangle$, of the configurations diverges then $\langle M_1, c[\vec{a'}]\rangle \approx_A \langle M_2, c[\vec{a'}]\rangle$ because potential computation of $c_2[\vec{a_2'}]$ in the second configuration may not change the relation. 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 $\langle M_2, c_1[\vec{a_1'}]\rangle \approx_A \langle M_1, c_1[\vec{a_1'}]\rangle$. Thus, there exist some M_1' and M_2' so that $\langle M_1, c_1[\vec{a_1'}]\rangle \Downarrow M_1'$, $\langle M_1, c_1[\vec{a_1'}]\rangle \Downarrow M_2'$, and $M_1' =_A M_2'$.

Because $\langle M_1, c_1[\vec{a_1}]\rangle \cong_A \langle M_2, c_1[\vec{a_1}]\rangle$, we have $\langle M_1, c_1[\vec{a_1}]\rangle \Downarrow N_1$ and $\langle M_2, c_1[\vec{a_1}]\rangle \Downarrow N_2$ for some N_1 and N_2. Applying Proposition 4 twice, we have $M_1'(v) = N_1(v)$ and $M_2'(v) = N_2(v)$ for all high-integrity variables v. Because $\langle M_1, c[\vec{a}]\rangle \cong_A \langle M_2, c[\vec{a}]\rangle$ and $N_1 =_A N_2$, by Proposition 3 we have $\langle N_1, c_2[\vec{a_2}]\rangle \cong_A \langle N_2, c_2[\vec{a_2}]\rangle$. The application of Proposition 5 yields $\langle M_1', c_2[\vec{a_2}]\rangle \approx_A \langle M_2', c_2[\vec{a_2}]\rangle$ By the induction hypothesis we have $\langle M_1', c_2[\vec{a_2'}]\rangle \approx_A \langle M_2', c_2[\vec{a_2'}]\rangle$. Connecting the traces for c_1 and c_2, we receive $\langle M_1, c_1[\vec{a_1'}]; c_2[\vec{a_2'}]\rangle \approx_A \langle M_2, c_1[\vec{a_1'}]; c_2[\vec{a_2'}]\rangle$.

`if` b `then` $c_1[\vec{a_1}]$ `else` $c_2[\vec{a_2}]$ If $Vars(b) \subseteq L_C$ then b evaluates to the same value, say *true*, under both 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 $\langle M_2, c[\vec{a'}]\rangle \longrightarrow \langle M_2, c_1[\vec{a_1'}]\rangle$. As $\langle M_1, c[\vec{a}]\rangle \cong_A \langle M_2, c[\vec{a}]\rangle$ we have $\langle M_1, c_1[\vec{a_1}]\rangle \cong_A \langle M_2, c_1[\vec{a_1}]\rangle$. By the induction hypothesis $\langle M_1, c_1[\vec{a_1'}]\rangle \approx_A \langle M_2, c_1[\vec{a_1'}]\rangle$. This implies $\langle M_1, c[\vec{a'}]\rangle \approx_A \langle M_2, c[\vec{a'}]\rangle$.

If $Vars(b) \not\subseteq L_C$, i.e., a high-confidentiality variable occurs in b, then we observe that there are no holes in the program, implying $c[\vec{a}] = c[\vec{a'}]$, which is a vacuous case.

`while` b `do` $c_1[\vec{a}]$ The case when $Vars(b) \not\subseteq L_C$ is handled in the same way as for conditionals. If $Vars(b) \not\subseteq H_I$ then no declassification may occur in `while` b `do` $c_1[\vec{a'}]$ by the definition of the type system and attacks. By

Theorem 1, the proof is $\texttt{while}\ b\ \texttt{do}\ c_1[\vec{a'}]$ satisfies noninterference, which completes the proof. The remaining case is $Vars(b) \subseteq L_C \cap H_I$. Expression b evaluates to the same value under both 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 $\langle M_1, c[\vec{a'}]\rangle \approx_A \langle M_2, c[\vec{a'}]\rangle$.

If, on the other hand, the value of b is *true* then for $\langle M_1, \texttt{while}\ b\ \texttt{do}\ c_1[\vec{a}]\rangle \approxeq_A \langle M_2, \texttt{while}\ b\ \texttt{do}\ c_1[\vec{a}]\rangle$ it is necessary that $\langle M_1, c_1[\vec{a}]\rangle \approxeq_A \langle M_2, c_1[\vec{a}]\rangle$. By the induction hypothesis, we have $\langle M_1, c_1[\vec{a'}]\rangle \approx_A \langle M_2, c_1[\vec{a'}]\rangle$. 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 $\langle M_1, c[\vec{a'}]\rangle \approx_A \langle M_2, c[\vec{a'}]\rangle$. If both configurations terminate, then there exist some M_1' and M_2' so that $\langle M_1, c_1[\vec{a'}]\rangle \Downarrow M_1'$, $\langle M_2, c_1[\vec{a'}]\rangle \Downarrow M_2'$, and $M_1' =_A M_2'$. Note that the value of b is the same under M_1 and M_2. If this value is *false* then the proof is finished. Otherwise, we need to further unwind the loop.

Applying Proposition 4 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 $M_1'(v) = N_1(v)$ and $M_2'(v) = N_2(v)$ for all high-integrity variables v. This, in particular, implies that b evaluates to *true* in both N_1 and N_2.

Because $\langle M_1, c[\vec{a}]\rangle \approxeq_A \langle M_2, c[\vec{a}]\rangle$ and $N_1 =_A N_2$, by Proposition 3 we have $\langle M_1, c_1[\vec{a}]\rangle \approxeq_A \langle M_2, c_1[\vec{a}]\rangle$ and therefore $\langle N_1, c[\vec{a}]\rangle \approxeq_A \langle N_2, c[\vec{a}]\rangle$ (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 due to an internal loop caused by low-integrity computation). During this iteration we have preserved the invariant that the executions for both M_1 and M_2 give low-confidentiality indistinguishable traces (for each $c_1[\vec{a}]$ and $c_1[\vec{a'}]$), and low-confidentiality high-integrity data in the final states of all traces is the same regardless of the memory (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 due to 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 $c_1[\vec{a'}]$. That $\langle M_1, c[\vec{a'}]\rangle \approx_A \langle M_2, c[\vec{a'}]\rangle$ we receive by concatenating low-assignment traces from each iteration. ☐

Appendix B

This appendix extends the proof of robustness to show that the type system with the rule for `endorse` guarantees qualified robustness. The proof structure is as in Appendix A. We lift the proof technique 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 6 (2). *An A-attack under Γ (i) does not have occurrences of assignments to high-integrity variables (such v that $\Gamma(v) \in H_I$); and (ii) satisfies (possibilistic) noninterference under Γ.*

Proposition 7 (3). *If $\Gamma, pc \vdash c_1; c_2$ and $\langle M_1, c_1; c_2\rangle \approxeq_A \langle M_2, c_1; c_2\rangle$ then $\langle M_1, c_1\rangle \approxeq_A \langle M_2, c_1\rangle$. If $t_1 \in Tr(\langle M_1, c_1\rangle)$ (assuming t_1 terminates in N_1) and $t_2 \in Tr(\langle M_2, c_1\rangle)$ (assuming t_2 terminates in N_2) so that $t_1 \approx_A t_2$ then $\langle N_1, c_2\rangle \approxeq_A \langle N_2, c_2\rangle$.*

Proposition 8 (4). *If $H_I \neq \emptyset$ and $\Gamma, pc \vdash c[\vec{\bullet}]$ for attacks $\vec{a}$ and $\vec{a'}$ so that and $\langle M, c[\vec{a}]\rangle$ always terminates then for any $t' \in Tr(\langle M, c[\vec{a'}]\rangle)$ so that t' terminates there exists $t \in Tr(\langle M, c[\vec{a}]\rangle)$ so that $t \sim_\ell t'$ for all such ℓ that $\ell \in H_I$.*

Proposition 9 (5). *If $\Gamma, pc \vdash c$ and $\langle M_1, c\rangle \approxeq_A \langle M_2, c\rangle$ for some M_1 and M_2 then for any value val and variable v so that $\Gamma(v) \in L_I$ we have $\langle M_1', c\rangle \approx_A \langle M_2', c\rangle$ where $M_1' = M_1[v \mapsto val]$ and $M_2' = M_2[v \mapsto val]$.*

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

Proof. If $H_I = \emptyset$ then declassification is disallowed by the typing rules, and the theorem follows form Theorem 4. In the rest of the proof we assume $H_I \neq \emptyset$. Induction on the structure of $c[\vec{\bullet}]$. Suppose that for some $c[\vec{a}]$ and memories M_1 and M_2 we have $\langle M_1, c[\vec{a}]\rangle \approxeq_A \langle M_2, c[\vec{a}]\rangle$ (which, in particular, implies $M_1 =_A M_2$). We need to show $\langle M_1, c[\vec{a'}]\rangle \approx_A \langle M_2, c[\vec{a'}]\rangle$ for all $\vec{a'}$. If $c[\vec{\bullet}]$ has the form `skip` or $v := e$ then the command has no holes, implying $c[\vec{a}] = c[\vec{a'}]$, which is a vacuous case because $\approxeq_A \subseteq \approx_A$. Note that this case covers (trusted) assignments with `declassify` and `endorse` in the right-hand side. The case $c[\vec{\bullet}] = [\bullet]$ is straightforward because by Proposition 6 attack a' satisfies noninterference.

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

The premise of the theorem states that $\langle M_1, c_1[\vec{a_1}]; c_2[\vec{a_2}]\rangle \approxeq_A \langle M_2, c_1[\vec{a_1}]; c_2[\vec{a_2}]\rangle$. We need to show $\langle M_1, c_1[\vec{a_1'}]; c_2[\vec{a_2'}]\rangle \approx_A \langle M_2, c_1[\vec{a_1'}]; c_2[\vec{a_2'}]\rangle$, i.e., by unfolding Definition 8, $\forall t_1 \in Tr(\langle M_1, c_1[\vec{a_1'}]; c_2[\vec{a_2'}]\rangle).\ \exists t_2 \in Tr(\langle M_2, c_1[\vec{a_1'}]; c_2[\vec{a_2'}]\rangle).\ t_1 \approx_A t_2$ along with the symmetric condition where M_1 and M_2 are swapped (which is proved analogously). We assume that t_1 terminates (otherwise the case is vacuous).

Suppose $t_1 = t_1' t_1''$ where $t_1' \in Tr(\langle M_1, c_1[\vec{a_1'}]\rangle)$, $\langle M_1, c_1[\vec{a_1'}]\rangle$ terminates with t_1' in some state M_1', and $t_1'' \in Tr(\langle M_1', c_2[\vec{a_2'}]\rangle)$. By Proposition 7 we deduce $\langle M_1, c_1[\vec{a_1}]\rangle \cong_A \langle M_2, c_1[\vec{a_1}]\rangle$. By the induction hypothesis we obtain $\langle M_1, c_1[\vec{a_1'}]\rangle \approx_A \langle M_2, c_1[\vec{a_1'}]\rangle$. In particular, $\exists t_2' \in Tr(\langle M_2, c_1[\vec{a_1'}]\rangle).\, t_1' \approx_A t_2'$.

If t_2' diverges, then we have found the necessary t_2 (it is simply t_2') to finish the proof. In the remaining case both traces t_1' and t_2' terminate and $t_1' \sim_A t_2'$. This means that there exist some M_1' and M_2' so that $\langle M_1, c_1[\vec{a_1'}]\rangle \Downarrow M_1'$, corresponding to trace t_1', $\langle M_1, c_1[\vec{a_1'}]\rangle \Downarrow M_2'$, corresponding to trace t_2', and $M_1' =_A M_2'$.

Applying Proposition 8 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 $t_1' \sim_\ell u_1'$ and $t_2' \sim_\ell u_2'$ for all such ℓ that $\ell \in H_I$. This leads to $u_1' \sim_A u_2'$. 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.

As $u_1' \approx_A u_2'$ and $\langle M_1, c[\vec{a}]\rangle \cong_A \langle M_2, c[\vec{a}]\rangle$ we have $\langle N_1, c_2[\vec{a_2}]\rangle \cong_A \langle N_2, c_2[\vec{a_2}]\rangle$ by Proposition 7. The application of Proposition 9 yields $\langle M_1', c_2[\vec{a_2}]\rangle \approx_A \langle M_2', c_2[\vec{a_2}]\rangle$ By the induction hypothesis we have $\langle M_1', c_2[\vec{a_2'}]\rangle \approx_A \langle M_2', c_2[\vec{a_2'}]\rangle$. 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 $t_1 \approx_A t_2$, implying $\langle M_1, c_1[\vec{a_1'}]; c_2[\vec{a_2'}]\rangle \approx_A \langle M_2, c_1[\vec{a_1'}]; c_2[\vec{a_2'}]\rangle$. $\square$