A Model for Delimited Information Release

Andrei Sabelfeld*[1] and Andrew C. Myers[2]

[1] Department of Computer Science, Chalmers University of Technology, 412 96 Gothenburg, Sweden, andrei@cs.chalmers.se
[2] Department of Computer Science, Cornell University, Ithaca, NY 14853, USA andru@cs.cornell.edu

Abstract. Much work on security-typed languages lacks a satisfactory account of intentional information release. In the context of confidentiality, a typical security guarantee provided by security type systems is *noninterference*, which allows no information flow from secret inputs to public outputs. However, many intuitively secure programs do allow some release, or *declassification,* of secret information (e.g., password checking, information purchase, and spreadsheet computation). Noninterference fails to recognize such programs as secure. In this respect, many security type systems enforcing noninterference are impractical. On the other side of the spectrum are type systems designed to accommodate some information leakage. However, there is often little or no guarantee about *what* is actually being leaked. As a consequence, such type systems are vulnerable to *laundering attacks*, which exploit declassification mechanisms to reveal more secret data than intended. To bridge this gap, this paper introduces a new security property, *delimited release*, an end-to-end guarantee that declassification cannot be exploited to construct laundering attacks. In addition, a security type system is given that straightforwardly and provably enforces delimited release.

Keywords: Computer security, confidentiality, information flow, noninterference, security-type systems, security policies, declassification.

1 Introduction

A long-standing problem in computer security is how to verifiably protect the confidentiality of sensitive information in practical computing systems. One of the most vexing difficulties is that realistic computing systems do release some confidential information as part of their intended function. The challenge is how to differentiate between proper and improper release of confidential information.

For example, it is possible to learn a small amount of information about a user's password by attempting to log in; the attacker likely learns that the password is *not* the one guessed. How can this secure system be distinguished from an insecure system that directly reports the entire password to the attacker? This paper proposes *delimited release*, a new definition of security that helps make the distinction.

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

To protect confidentiality within a computing system, it is important to control how information flows so that sensitive information is not transmitted inappropriately to system outputs. One way to control these flows is to associate a *security level* with information in the system, and to prevent higher-level (more confidential) information from affecting lower-level (less confidential) information. Recently there has been much work embodying this approach in a language-based setting [37], where the system to be validated is a program and the security levels are types in that program [47, 19, 27, 2, 42, 45, 4, 38, 33, 39, 51, 5, 34]. A program written in this sort of *security-typed language* is considered secure only if it is well-typed, which rules out, for example, assignments from high-level variables to low-level variables.

This kind of static checking tends to be very restrictive, preventing practical programming. Typically these languages are intended to enforce some version of the *noninterference* [16] security property, which prevents low-level information from depending on high-level information. Yet many practical programs, such as the password checker mentioned above, do release information. Another example is aggregating data from a large database (such as an employee database) to compute a less confidential result (such as the average salary). And sometimes confidential information is released as part of a transaction or agreed-upon protocol, such as when information is purchased. All of these programs violate noninterference and would be rejected by the type systems of most current security-typed languages.

Assuming that the confidentiality of data is expressed as a security level, some additional mechanism is needed in order to express programs in which there is an intentional release of information. Some security-typed languages (e.g., [27, 33, 13]) have therefore added a *declassification* mechanism that coerces the security level of information downwards. Declassification serves as an escape hatch from the rigid restrictions of security type systems, but it (intentionally) violates noninterference.

A question that has not been addressed satisfactorily by earlier work is what security guarantees can be offered in the presence of declassification. Delimited release is such a guarantee. Like noninterference, it has the attractive property that it defines security in terms of the program semantics rather than in terms of non-standard mechanisms. Thus, it controls the *end-to-end* [40] behavior of the program: it is an *extensional* security property [26].

In the rest of the paper, we present an imperative language (Section 2), formally define delimited release security (Section 3), give a security type system that provably enforces delimited release (Section 4), discuss a password-checking example (Section 5), sketch related work (Section 6), and conclude (Section 7).

2 A security-typed language

To illustrate the security model, we consider a simple sequential language, consisting of expressions and commands. The language is similar to several other

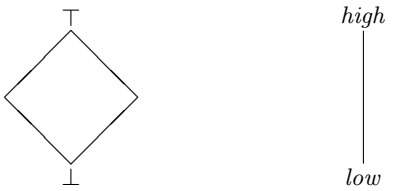

Fig. 1. A general security lattice $\mathcal{L}$ and the lattice $\mathcal{L}_{LH}$

security-typed imperative languages (e.g., [47, 4]), and its semantics are largely standard (cf. [48]).

The language syntax is defined by the following grammar:

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

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

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 security levels.

We assume that the security levels of data are elements of a *security lattice* $\mathcal{L}$. The ordering specifies the relationship between different security levels. If $\ell_1 \sqsubseteq \ell_2$ then information at level ℓ_1 is also visible at level ℓ_2. However, if $\ell_1 \not\sqsubseteq \ell_2$ then information at level ℓ_1 is invisible at level ℓ_2. The join operation ($\sqcup$) of $\mathcal{L}$ is useful, for example, for computing an upper bound on the security level of an expression that combines sub-expressions at different security levels. An example is the security lattice $\mathcal{L}_{LH}$ with two elements *high* and *low* representing high and low confidentiality levels, respectively, with the ordering $\mathit{low} \sqsubseteq \mathit{high}$. A general security lattice $\mathcal{L}$ with a top element $\top$ and bottom element $\bot$ is depicted in Figure 1, along with the lattice $\mathcal{L}_{LH}$.

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 *security policy*, which specifies that information flow from a variable v_1 to a variable v_2 is allowed only if $\Gamma(v_1) \sqsubseteq \Gamma(v_2)$. For simplicity, we assume a fixed Γ in the upcoming formal definitions.

The only language expression that is not standard is $\texttt{declassify}(e, \ell)$, a construct for declassifying the security level of the expression e to the level $\ell \in \mathcal{L}$. We require that `declassify` expressions are not nested. At the semantic level, $\texttt{declassify}(e, \ell)$ is equivalent to e regardless ℓ. The intention is that declassification is used for controlling the security level of information without affecting the execution of the program.

The semantics are defined in terms of 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* (or *expression*) c. If c is a command (resp. expression) then we sometimes refer to $\langle M, c\rangle$ as *command configuration* (resp. *expression configuration*). A transition from configuration

$\langle M_1, c_1\rangle$ to configuration $\langle M_2, c_2\rangle$ is denoted by $\langle M_1, c_1\rangle \longrightarrow \langle M_2, c_2\rangle$. 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. We assume that operations used in expressions are total, and, hence, expression configurations always terminate (denoted by $\langle M, e\rangle \Downarrow \mathit{val}$).

3 A delimiting model for confidentiality

The usual way of defining confidentiality is as *noninterference* [16], a security property stating that inputs of high confidentiality do not affect outputs of lower confidentiality. Various definitions of noninterference have been used by much recent work on language-based security (e.g., [47, 2, 19, 42, 45, 4, 38, 33, 39, 51, 5, 34]). However, noninterference cannot characterize the security of a program that is designed to release some confidential information as part of its proper functioning. We propose a new confidentiality characterization that delimits information release and precludes laundering attacks.

3.1 Noninterference

Noninterference is defined as follows for programs written in the language of Section 2: if two input memories are indistinguishable for an attacker at a security level ℓ then the behavior of the program on these memories is also indistinguishable at ℓ. Formally, two memories M_1 and M_2 are indistinguishable $M_1 =_\ell M_2$ at level ℓ if $\forall v.\, \Gamma(v) \sqsubseteq \ell \implies M_1(v) = M_2(v)$ (we assume a fixed Γ; hence the notation $M_1 =_\ell M_2$ is not parameterized by security environments). The behavior of two program configurations $\langle M_1, c_1\rangle$ and $\langle M_2, c_2\rangle$ is indistinguishable at ℓ (written $\langle M_1, c_1\rangle \approx_\ell \langle M_2, c_2\rangle$) if whenever $\langle M_1, c_1\rangle \Downarrow M_1'$ and $\langle M_2, c_2\rangle \Downarrow M_2'$ for some M_1' and M_2' then $M_1' =_\ell M_2'$. The behavior of two expression configurations $\langle M_1, e_1\rangle$ and $\langle M_2, e_2\rangle$ is indistinguishable (written $\langle M_1, e_1\rangle \approx \langle M_2, e_2\rangle$) if $\langle M_1, e_1\rangle \Downarrow \mathit{val}$ and $\langle M_2, e_2\rangle \Downarrow \mathit{val}$ for some val. We are now ready to formulate the noninterference security condition.

Definition 1 (Noninterference) *Command c satisfies* noninterference *if for all security levels ℓ we have*

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

While noninterference is a useful dependency-based security specification, it is over-restrictive for programs with declassification. For example, suppose we need to intentionally release the parity of a secret variable h in such a way that no other information about h is leaked. The program performing such a release is below:

$$l := \texttt{declassify}(\texttt{parity}(h), \mathit{low}) \qquad \text{(Par)}$$

where $\Gamma(h) = \mathit{high}$ and $\Gamma(l) = \mathit{low}$ under the lattice $\mathcal{L}_{LH}$. In the above sense, this program is intuitively secure. However, noninterference flatly rejects the program because l does depend on h.

3.2 Escape hatches and delimited release

In the example above, we want to express the requirement that only explicitly declassified data but no further information is released. Therefore, the specification of security must be relative to the expressions that appear under `declassify` operators. These expressions can be viewed as a part of the security policy, specifying the "escape hatches" for information release. To make this security policy explicit, one could require all escape hatch expressions to be declared in a separate interface. Because this is unimportant for the technical development, we omit this requirement. The new security specification *delimits* information release by only allowing release through escape hatch expressions:

Definition 2 (Delimited release) *Suppose the command c contains within it exactly n declassify expressions* $\texttt{declassify}(e_1, \ell_1), \ldots, \texttt{declassify}(e_n, \ell_n)$. *Command c is* secure *if for all security levels ℓ we have*

$$\forall M_1, M_2.\, (M_1 =_\ell M_2 \;\;\&\;\; \forall i \in \{i \mid \ell_i \sqsubseteq \ell\}\,.\,\langle M_1, e_i\rangle \approx \langle M_2, e_i\rangle) \implies \langle M_1, c\rangle \approx_\ell \langle M_2, c\rangle$$

Intuitively, this definition says that for all ℓ, M_1 and M_2 so that $M_1 =_\ell M_2$, if there is an information leak through one of the escape hatches $e_1, \ldots, e_n$ observable at level ℓ, i.e., $\exists i \in \{i \mid \ell_i \sqsubseteq \ell\}\,.\,\langle M_1, e_i\rangle \not\approx \langle M_2, e_i\rangle$, then this leak is allowed, i.e., no further conditions are imposed. However, if the difference between M_1 and M_2 is invisible at ℓ through all escape hatches, i.e., $\forall i \in \{i \mid \ell_i \sqsubseteq \ell\}\,.\,\langle M_1, e_i\rangle \approx \langle M_2, e_i\rangle$, then this difference must be invisible at ℓ through the entire execution of c, i.e., $\langle M_1, c\rangle \approx_\ell \langle M_2, c\rangle$.

One way of interpreting the delimited release definition is that a given program is secure as long as updates to variables that are later declassified occur in a way that does not increase the information visible by the attacker through the escape hatches. If no variables used in declassification are updated before the actual declassification, delimited release reduces to noninterference. This observation leads to a simple way of automatically enforcing delimited release, reported in Section 4.

It is instructive to compare delimited release to noninterference. Clearly, noninterference is stronger than delimited release:

Proposition 1 *If program c satisfies noninterference then c is secure.*

Furthermore, for a program without `declassify` primitives the two security properties coincide.

Proposition 2 *If* `declassify` *primitives do not occur in a secure program c then c satisfies noninterference.*

3.3 Examples

The security provided by delimited release can be understood from some simple examples: averaging salaries, an electronic wallet, and password checking.

Example 1 (Average salary). Suppose variables $h_1, \ldots, h_n$ store the salaries of n employees. The average salary computation is intended to intentionally release the average but no other information about $h_1, \ldots, h_n$ to a public variable *avg*:

$$avg := \texttt{declassify}((h_1 + \cdots + h_n)/n, low) \qquad \text{(Avg)}$$

We assume lattice $\mathcal{L}_{LH}$ so that $\forall i.\, \Gamma(h_i) = high$ and $\Gamma(avg) = low$. Clearly the program does not satisfy noninterference as there is a dependency from $h_1, \ldots, h_n$ to *avg*. However, the nature of information flow from high to low is limited. Although the low-level observer learns the average of the secret inputs, it is not possible to learn more information about them. For example, swapping the values of h_i and h_j is not visible at the low level. Allowing these limited flows, the program is accepted as secure by the delimited release definition.

On the other hand, consider a laundering attack on program Avg that leaks the salary of employee i to *avg*.

$$\begin{array}{c} h_1 := h_i; \ldots h_n := h_i; \\ avg := \texttt{declassify}((h_1 + \cdots + h_n)/n, low) \end{array} \qquad \text{(Avg-Attack)}$$

This program does not satisfy delimited release. To see this, take $i = 1$, $M_1(h_1) = M_2(h_2) = 2$, $M_2(h_1) = M_1(h_2) = 3$, and $M_1(v) = M_2(v) = 0$ for all variables v different from h_1 and h_2. For $\ell = low$ we have $M_1 =_\ell M_2$ and $\langle M_1, (h_1 + \cdots + h_n)/n\rangle \approx \langle M_2, (h_1 + \cdots + h_n)/n\rangle$ because both expression configurations evaluate to $5/n$. But $\langle M_1, \text{Avg-Attack}\rangle \not\approx_\ell \langle M_2, \text{Avg-Attack}\rangle$ because the final value of the public variable *avg* is 2 and 3, respectively, which violates Definition 2. Therefore, the laundering attack is rejected as insecure.

Example 2 (Electronic wallet). Consider an electronic shopping scenario. Suppose h stores the (secret) amount of money in a customer's electronic wallet, l stores the (public) amount of money spent during the current session, and k stores the cost of the item to be purchased. The following code fragment checks if the amount of money in the wallet is sufficient and, if so, transfers the amount k of money from the customer's wallet to the spent-so-far variable l:

$$\texttt{if declassify}(h \geq k, low)\ \texttt{then}\ (h := h - k; l := l + k)\ \texttt{else skip} \qquad \text{(Wallet)}$$

We assume lattice $\mathcal{L}_{LH}$ so that $\Gamma(h) = high$ and $\Gamma(k) = \Gamma(l) = low$. As with program Avg, this program fails to satisfy noninterference but does satisfy delimited release. Below is an attack that abuses the declassification primitive and

leaks[3] the secret variable h bit-by-bit to l (assuming h is an n-bit integer):

$$
\begin{array}{l}
l := 0; \\
\texttt{while } (n \geq 0) \texttt{ do} \\
\quad k := 2^{n-1}; \\
\quad \texttt{if declassify}(h \geq k, \mathit{low}) \\
\qquad \texttt{then } (h := h - k; l := l + k) \texttt{ else skip}; \\
\quad n := n - 1
\end{array}
\qquad \text{(Wallet-Attack)}
$$

where $\Gamma(n) = \mathit{low}$. This is a laundering attack whose effect is magnified by the loop. It is not difficult to see that the attack is indeed rejected by the delimited release model.

3.4 Features and extensions

An interesting feature of the delimited release is that it forces the programmer to be explicit about what information is being released. This is because the security policy, in the form of expressions under `declassify`, is simply a part of program text. For example, consider the following program:

$$
\begin{array}{l}
h := \texttt{parity}(h); \\
\texttt{if declassify}(h = 1, \mathit{low}) \texttt{ then } (l := 1; h := 1) \texttt{ else } (l := 0; h := 0)
\end{array}
$$

where $\Gamma(h) = \mathit{high}$ and $\Gamma(l) = \mathit{low}$ under the lattice $\mathcal{L}_{LH}$. According to the security policy $h = 1$, whether or not the initial value of h was 1 is the information intended for declassification. Instead, however, the low-level observer learns the parity of the initial value of h. This is a laundering attack, which is rejected by the delimited release model. To produce a semantically equivalent secure version of the program above, the programmer may rewrite it to the following program:

$$
\texttt{if declassify}(\texttt{parity}(h), \mathit{low}) \texttt{ then } (l := 1; h := 1) \texttt{ else } (l := 0; h := 0)
$$

This is indeed a secure program according to Definition 2. That the parity of h is subject to intentional information release is now more evident from the security policy $\texttt{parity}(h)$.

The desire to automate the above transformation leads to extensions based on *security ordering* for programs. A program c_2 is at least as secure as a program c_1 (written $c_1 \sqsubseteq_{sec} c_2$) if for all security levels $\ell \in \mathcal{L}$ and memories M_1 and M_2 we have whenever $M_1 =_\ell M_2$ and $\langle M_1, c_1\rangle \approx_\ell \langle M_2, c_1\rangle$ then $\langle M_1, c_2\rangle \approx_\ell \langle M_2, c_2\rangle$. The intuition is that c_2 leaks no more secret information than c_1, hence the security ordering. It is straightforward to see that security ordering $\sqsubseteq_{sec}$ is a preorder (reflexive and transitive). In the partial order, formed by equivalence classes of $\sqsubseteq_{sec}$, programs satisfying noninterference belong to the top-security class (in the sense that the other programs are strictly less secure). A decidable

[3] Furthermore, this attack compromises the *integrity* of the customer's wallet variable. However, this is orthogonal to the confidentiality issues dealt with in this paper.

approximation of this security ordering for driving security-enhancing program transformation is an attractive direction for future work.

The delimited release model has some limitations in describing security policies for information release, because all of the declassified expressions are assumed to be released. The model does not permit the expression of a security policy in which information should be released only under certain conditions. For example, consider a program that leaks either h_1 or h_2, but not both:

$$\texttt{if } l \texttt{ then } l := \texttt{declassify}(h_1, \mathit{low}) \texttt{ else } l := \texttt{declassify}(h_2, \mathit{low})$$

where $\Gamma(h_1) = \Gamma(h_2) = \mathit{high}$ and $\Gamma(l) = \mathit{low}$ under the lattice $\mathcal{L}_{LH}$. This program is secure according to Definition 2. Both h_1 and h_2 appear under `declassify`, which, according to Definition 2, means that the program might leak the values of both. The requirement that only one of the two variables is released cannot be expressed using Definition 2. However, delimited release can be enhanced with *disjunctive policies* for representing finer policies as, e.g., "either h_1 or h_2 can be released but not both." Moreover, delimited release can be integrated with techniques based on *who* controls information release, such as *robust declassification* [50, 49, 30]. This integration can help specify whether the decision on which of h_1 and h_2 can be released may or may not be in the hands of an attacker. A remark on the combination of delimited release and robust declassification follows in Section 7.

4 A security type system for delimited release

This section presents a type system that statically enforces security. The typing rules are displayed in Figure 2. The general form of typing for an expression is $\Gamma \vdash e : \ell, D$ meaning that an expression e has type ℓ and effect D under an environment Γ. Typing for commands has the form $\Gamma, pc \vdash c : U, D$ meaning that a command c is typable with effects U and D under an environment Γ and a *program counter* pc. Program counters range over security levels; they help track information flow due to control flow. A program counter records a lower bound on the security level of variables assigned in a given program. The type system guarantees that if there is branching (as in `if` and `while` commands) on data at level ℓ then the branches must be typable under a program counter at least at ℓ, preventing leaks via assignments in the branches.

Besides tracking information flow through assignments and control flow (in the spirit of [11, 47]), the type system collects information about what variables are used under declassification (which is recorded in the effect D of an expression or a command) and what variables are updated by commands (which is recorded in the effect U of a command). For example, the D effect of a $\texttt{declassify}(e, \ell)$ expression is the set of variables appearing in e, written as $\mathit{Vars}(e)$. The key restriction guaranteed by the type system is that variables used under declassification may not be updated prior to declassification. This restriction is enforced in the rules for the sequential composition and the while loop. The overall programming discipline enforced by the type system ensures that typable programs are secure, which is formalized by the following theorem.

$$\Gamma \vdash val : \ell, \emptyset$$

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

$$\frac{\Gamma \vdash e : \ell, D_1 \quad \Gamma \vdash e' : \ell, D_2}{\Gamma \vdash e \ \texttt{op} \ e' : \ell, D_1 \cup D_2}$$

$$\frac{\Gamma \vdash e : \ell, D}{\Gamma \vdash \texttt{declassify}(e, \ell') : \ell', \mathit{Vars}(e)}$$

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

$$\Gamma, pc \vdash \texttt{skip} : \emptyset, \emptyset$$

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

$$\frac{\Gamma, pc \vdash c_1 : U_1, D_1 \quad \Gamma, pc \vdash c_2 : U_2, D_2 \quad U_1 \cap D_2 = \emptyset}{\Gamma, pc \vdash c_1; c_2 : U_1 \cup U_2, D_1 \cup D_2}$$

$$\frac{\Gamma \vdash e : \ell, D \quad \Gamma, \ell \sqcup pc \vdash c_1 : U_1, D_1 \quad \Gamma, \ell \sqcup pc \vdash c_2 : U_2, D_2}{\Gamma, pc \vdash \texttt{if} \ e \ \texttt{then} \ c_1 \ \texttt{else} \ c_2 : U_1 \cup U_2, D \cup D_1 \cup D_2}$$

$$\frac{\Gamma \vdash e : \ell, D \quad \Gamma, \ell \sqcup pc \vdash c : U_1, D_1 \quad U_1 \cap (D \cup D_1) = \emptyset}{\Gamma, pc \vdash \texttt{while} \ e \ \texttt{do} \ c : U_1, D \cup D_1}$$

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

Fig. 2. Typing rules

Theorem 1. *$\Gamma, pc \vdash c : U, D \Longrightarrow c$ is secure.*

A proof by induction on the typing derivation is sketched in the appendix.

Note that the type system is more restrictive than necessary to enforce the security condition. For example, consider the program

$$h := \texttt{parity}(h); l := \texttt{declassify}(h, low)$$

where $\Gamma(h) = high$ and $\Gamma(l) = low$ under the lattice $\mathcal{L}_{LH}$. Although h is updated prior to declassification, the entire program actually leaks only the parity of the initial value of h, which is less information than the complete initial value of h leaked by the declassifying assignment alone. Indeed, according to Definition 2 the program is secure; however, it is rejected by the type system. Devising a more permissive type system for enforcing the delimited release security condition is a worthwhile topic for future work.

5 A password-checking example

This section applies the delimited release model to password checking, illustrating how the type system gives security types to password-checking routines and also prevents laundering attacks.

We consider UNIX-style password checking where the system database stores the *images* (or the hashes) of password-salt pairs. *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. We use arrows in types for expressions to 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 *low*.

$$\begin{aligned}\Gamma \vdash \texttt{hash}(pwd, salt) &: \ell_{pwd} \times \ell_{salt} \rightarrow low \\ &= \texttt{declassify}(\texttt{buildHash}(pwd||salt), low)\end{aligned}$$

The expression $\texttt{match}(pwdImg, salt, query)$ checks if the password image *pwdImg* is equal to the hash of the user query *query* with the salt *salt*.

$$\begin{aligned}\Gamma \vdash \texttt{match}(pwdImg, salt, query) &: \ell_{pwdImg} \times \ell_{salt} \times \ell_{query} \rightarrow \ell_{pwdImg} \sqcup low \\ &= (pwdImg = \texttt{hash}(query, salt))\end{aligned}$$

Notice that the expression is typable only if the security level of the result is no less confidential than both the security level of *pwdImg* and *low*. The program

$\texttt{update}(pwdImg, salt, oldPwd, newPwd)$ updates the old password hash $pwdImg$ by querying the old password $oldPwd$, matching its hash to $pwdImg$ and (if matched) updating the hashed password with the hash of $newPwd$.

$$\begin{aligned}\Gamma, \ell_{pwdImg} \vdash \texttt{update}(pwdImg, salt, oldPwd, newPwd)\ (low \sqsubseteq \ell_{pwdImg})\\ = \texttt{if match}(pwdImg, salt, oldPwd)\\ \qquad \texttt{then } pwdImg = \texttt{hash}(newPwd, salt)\\ \qquad \texttt{else skip}\end{aligned}$$

Let us instantiate the typings above for the lattice $\mathcal{L}_{LH}$ and show that they capture the desired intuition.

- The honest user applying hash to a password and salt:
  $\Gamma \vdash \texttt{hash}(pwd, salt) : high \times low \to low$.
- The attacker hashing a password with the honest user's public salt:
  $\Gamma \vdash \texttt{hash}(pwd, salt) : low \times low \to low$.
- The honest user matching a password:
  $\Gamma \vdash \texttt{match}(pwdImg, salt, query) : low \times low \times high \to low$.
- The attacker attempting to guess a password by matching it to a legitimate password image and salt:
  $\Gamma \vdash \texttt{match}(pwdImg, salt, query) : low \times low \times low \to low$.
- The honest user modifying a password:
  $\Gamma, low \vdash \texttt{update}(pwdImg, salt, oldPwd, newPwd) : low \times low \times high \times high$.
- The attacker attempting to modify the honest user's password:
  $\Gamma, low \vdash \texttt{update}(pwdImg, salt, oldPwd, newPwd) : low \times low \times low \times low$.

These are all typable and hence secure programs. The rationale for considering these programs secure is that to succeed the attacker needs to guess the password, which is unlikely given a large password space and little prior knowledge.

That the programs are typable guarantees that the password-checking mechanism is not vulnerable to laundering attacks. For example, consider an attack that, similarly to Wallet-Attack, launders bit-by-bit the secret variable h to l (assuming h is an n-bit integer) via the declassification mechanism that is built in the hash expression.

$$\begin{aligned}&l := 0;\\ &\texttt{while } (n \geq 0) \texttt{ do}\\ &\quad k := 2^{n-1};\\ &\quad \texttt{if hash}(\texttt{sign}(h - k + 1), 0) = \texttt{hash}(1, 0)\\ &\qquad \texttt{then } (h := h - k; l := l + k) \texttt{ else skip};\\ &\quad n := n - 1\end{aligned}$$

where $\Gamma(k) = \Gamma(l) = \Gamma(n) = low, \Gamma(h) = high$ and $\texttt{sign}$ returns $1, -1$ or 0 if the argument is positive, negative, or 0, respectively. That this attack might indeed leak h in a bit-by-bit fashion is easy to see because the inequality $h \geq k$ holds if and only if the inequality $h - k + 1 > 0$ holds, which, for a sensible hashing algorithm, is likely to be equivalent to $\texttt{hash}(\texttt{sign}(h - k + 1), 0) = \texttt{hash}(1, 0)$.

Clearly, the program above is insecure according to Definition 2. Notice that the program is rightfully rejected by the type system. This is because variable h both occurs under declassification and is updated in the body of the loop.

Furthermore, observe that programs Avg and Wallet are typable whereas attacks Avg-Attack and Wallet-Attack are rejected by the type system.

6 Related work

Policies for intentional information release have been an active area of research. Cohen's *selective (in)dependence* [8] security definition can be viewed as a precursor for our work. Cohen's definition is based on partitioning the secret input domain into subdomains requiring noninterference when secret variables are restricted to each subdomain. For example, program Par, revealing the parity of h to l, satisfies selective independence with respect to the partitioning of the domain of integers for h into odd and even numbers. However, the security policy of specifying *what* can be leaked relies on a semantic specification of subdomains. Recent incarnations of selective independence based on *abstract variables* [20], *equivalence relations* [39], and *abstract noninterference* [14] also need to be specified at the semantic level. In contrast, our escape hatches facilitate a syntactic way of specifying selective independence: two values are in the same subdomain if and only if the results of evaluating the expression under each `declassify` primitive on these two values are the same values. Moreover, the escape hatches provide the flexibility to condition the release of information on public values (cf. program Wallet), which cannot be represented by the original definition of selective independence. Finally, the syntactic escape-hatch policy mechanism leads us to a security type system that enforces security, whereas there appears no automatic enforcement mechanisms for (any variation of) selective independence.

Further related work is grouped into categories of *how* information is released, *how much* information is released, and *relative to what* information is released. For a more detailed overview of this area we refer to a recent survey [37].

How? A common approach to relaxing noninterference to account for intentional information release is based on *intransitive noninterference* [36, 32, 35, 24], which originated from early work on *conditional noninterference* [16, 17]. Mantel and Sands have recently addressed intransitive noninterference in a language-based setting [25]. Intransitive flows in the context of declassification-sensitive unwinding have been explored by Bossi et al. [6]. Intransitive noninterference accommodates policies where information might flow intransitively, e.g., from level ℓ_1 to ℓ_2 and from ℓ_2 to ℓ_3 but not from ℓ_1 to ℓ_3 directly. The goal is that information may only be declassified if it passes through a special declassifier security level. The assurance provided by this approach is that portions of computation between declassification actions are in a certain sense secure. However, no guarantees are given for the entire computation. Myers and Liskov's *decentralized model* [28, 29] offers security labels in which *selective declassification* [33] is permitted on the basis of a static analysis of process authority and relationships between

principals. Security labels have additional structure that describes the entities capable of performing declassification. While the above policies help express *how* information is released, they fail to account for *what* has been released. In particular, neither intransitive noninterference nor selective declassification directly prevents laundering attacks.

How much? A *quantitative* approach to information flow gives bounds on *how much* information may be released. For instance, this is useful for measuring how much information about the password is revealed on a login attempt during password checking. Based on Shannon's information theory [41], early ideas of quantitative security go back to Denning's work [10] which, however, does not provide automated tools for estimating the bandwidth. Clark et al. [7] propose syntax-directed inference rules for computing estimates on information flow resulted from `if` statements in an imperative language. Recent line of work by Di Pierro et al. [12] suggests *approximate noninterference*, which can be thought of as noninterference modulo probabilistically specified "noise" In a process-algebra setting, Lowe's quantitative definition of information flow [23] measures the capacity of information flow channels. However, tracking the quantity of information through program construct appears to be a daunting task. To date, there appears no static analysis with reasonably permissive rules for `while` loops.

Relative to what? In the rest of this section, we discuss models for information release relative to the attacker's power to observe and affect declassification. Volpano and Smith [46] have proposed a type system that allows password-matching operations and with the security assurance that (i) no well-typed program can leak secrets in polynomial (in the length of the secret) time, and (ii) secret leaks are only possible with a negligible probability. In subsequent work [44], Volpano proves that leaking passwords in a system where passwords are stored as images of a one-way function is not easier than breaking the one-way function. Both of these studies are, however, tailored to the password-checking scenario. Abadi gives a type system [1] in which declassification is connected to uses of encryption, for a calculus of cryptographic protocols, the spi calculus [3]. Secret keys and their usage are hidden by the security definition, allowing the result of encryption to be considered publicly visible. Sumii and Pierce [43] employ relational parametricity techniques for reasoning about cryptographic protocols involving encryption. Laud's complexity-theoretic security definition [21, 22] is also specific to declassification by encryption. This security definition ensures that a polynomial-time (in the length of the secret) adversary in an imperative language is not able to leak secrets by abusing the encryption-based declassification mechanism.

The idea underlying Dam and Giambiagi's *admissibility* [9, 15] is that the implementation of a specification satisfies admissibility if there are no other information flows than those described in a confidentiality policy of the specification. The relativity of information release here is with respect to information release in the specification.

Finally, Zdancewic and Myers have proposed a security condition called *robust declassification* [50], which captures the idea that an attacker may not learn more information than intended. The key idea is that attacker-controlled computation is not allowed to increase observations about secrets by causing misuse of the declassification mechanism. Robust declassification ensures that an active attacker (who can affect system behavior) cannot learn anything more than a passive attacker (who may only observe the system's behavior). Zdancewic has proposed a type system [49] intended to enforce robust declassification. Recently, Myers et al. [30] have generalized robust declassification as an enforceable end-to-end security property and introduced *qualified robustness* that provides untrusted code with a limited ability to affect information release.

7 Conclusion

We have presented a security model for intentional information release. Because this model delimits information flow by explicit policies, we are able to capture *what* information is released as opposed to *how* it is released. This approach enables us to track laundering attacks that are often undetectable by other models of information flow. Much work on information flow relies on *compartmentalization* (creating special security levels, or compartments, for data to restrict information flow) to fence against laundering attacks. However, as we have seen from the average-salary, electronic-wallet, and password-checking examples, compartmentalization is not a panacea. Our model can be viewed as another line of defense that prevents attacks missed by compartmentalization.

The delimited release model is in some ways orthogonal to robust declassification [50]; the former controls what may be declassified, and the latter ensures that the attacker cannot control decisions about when it is declassified. A synthesis of these security definitions in a language-based setting would further improve assurance that information is being released properly. Delimited release also opens up possibilities for using security-typed languages such as Jif [27, 31] to write components of larger systems written in more conventional languages such as Java [18]. Delimited release security could guarantee that security-critical Jif code wrapped into Java programs would not disclose more information than is released by the Jif code alone.

Acknowledgment Thanks are due to Fabio Martinelli, David Sands, Eijiro Sumii, and Steve Zdancewic for helpful comments.

This research was supported by the Department of the Navy, Office of Naval Research, 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.

References

1. M. Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786, September 1999.

2. 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, January 1999.
3. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Information and Computation*, 148(1):1–70, January 1999.
4. J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, January 2000.
5. 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.
6. A. Bossi, C. Piazza, and S. Rossi. Modelling downgrading in information flow security. In *Proc. IEEE Computer Security Foundations Workshop*, June 2004. To appear.
7. 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.
8. 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.
9. 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.
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. D. Duggan. Cryptographic types. In *Proc. IEEE Computer Security Foundations Workshop*, pages 238–252, June 2002.
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, January 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, April 2003.
16. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.
17. J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symp. on Security and Privacy*, pages 75–86, April 1984.
18. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, August 1996.
19. 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, January 1998.
20. R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.
21. 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, April 2001.

22. 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, April 2003.
23. G. Lowe. Quantifying information flow. In *Proc. IEEE Computer Security Foundations Workshop*, pages 18–31, June 2002.
24. H. Mantel. Information flow control and applications—Bridging a gap. In *Proc. Formal Methods Europe*, volume 2021 of *LNCS*, pages 153–172. Springer-Verlag, March 2001.
25. H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. Draft, July 2003.
26. J. McLean. The specification and modeling of computer security. *Computer*, 23(1):9–16, January 1990.
27. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.
28. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, October 1997.
29. A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symp. on Security and Privacy*, pages 186–197, May 1998.
30. A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, June 2004. To appear.
31. 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.
32. S. Pinsky. Absorbing covers and intransitive non-interference. In *Proc. IEEE Symp. on Security and Privacy*, pages 102–113, May 1995.
33. F. Pottier and S. Conchon. Information flow inference for free. In *Proc. ACM International Conference on Functional Programming*, pages 46–57, September 2000.
34. F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, January 2002.
35. A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proc. IEEE Computer Security Foundations Workshop*, pages 228–238, June 1999.
36. J. M. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, 1992.
37. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
38. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.
39. A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher Order and Symbolic Computation*, 14(1):59–91, March 2001.
40. J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.
41. C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, 1963.
42. 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, January 1998.
43. E. Sumii and B. Pierce. Logical relations for encryption. In *Proc. IEEE Computer Security Foundations Workshop*, pages 256–269, June 2001.

44. D. Volpano. Secure introduction of one-way functions. In *Proc. IEEE Computer Security Foundations Workshop*, pages 246–254, July 2000.
45. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3):231–253, November 1999.
46. D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 268–276, January 2000.
47. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.
48. G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, 1993.
49. S. Zdancewic. A type system for robust declassification. In *Proc. Mathematical Foundations of Programming Semantics*, ENTCS. Elsevier, March 2003.
50. S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.
51. 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, April 2001.

Appendix

This appendix presents a proof of Theorem 1.

Theorem 1. *$\Gamma, pc \vdash c : U, D \Longrightarrow c$ is secure.*

Proof. We sketch a proof by induction on the typing derivation for c. With the exception of the straightforward case for the subsumption rule (in the bottom of Figure 2), the induction is on the structure of c. Suppose c contains exactly n declassify expressions $\texttt{declassify}(e_1, \ell_1), \ldots, \texttt{declassify}(e_n, \ell_n)$. Suppose that for some security level ℓ and memories M_1 and M_2 where $M_1 =_\ell M_2$, we have $\forall i \in \{i \mid \ell_i \sqsubseteq \ell\} \,.\, \langle M_1, e_i\rangle \approx \langle M_2, e_i\rangle$. We need to show $\langle M_1, c\rangle \approx_\ell \langle M_2, c\rangle$. We assume that $\langle M_1, c\rangle \Downarrow M_1'$ and $\langle M_2, c\rangle \Downarrow M_2'$ for some M_1' and M_2' (the relation is obvious if one of the configurations diverges). It remains to be shown that $M_1' =_\ell M_2'$.

`skip` Straightforward, because $M_1' = M_1 =_\ell M_2 = M_2'$.

$v := e$ Clearly, $M_1' =_\ell M_1 =_\ell M_2 =_\ell M_2'$ in case $\Gamma(v) \not\sqsubseteq \ell$. In case $\Gamma(v) \sqsubseteq \ell$, the typing rule for assignments ensures that the security levels of variables occurring in e—outside `declassify` primitives—are below ℓ. Hence the values of these variables are the same in both M_1 and M_2. That the values of expressions under `declassify` primitives are the same in both M_1 and M_2 is guaranteed by the assumption $\forall i \in \{i \mid \ell_i \sqsubseteq \ell\} \,.\, \langle M_1, e_i\rangle \approx \langle M_2, e_i\rangle$. (Note that declassification to level ℓ_j for some j so that $\ell_j \not\sqsubseteq \ell$ is not allowed by the typing rule for assignments.) To sum up, the values of variables outside declassification primitives are the same in both M_1 and M_2 and so are the values of expressions under `declassify`. Clearly, the application of `op` operations to the subexpressions gives the same results for both M_1 and M_2. Therefore, $M_1' =_\ell M_2'$.

$c_1; c_2$ By the typing rule for sequential composition, both c_1 and c_2 must be typable. The set $I = \{i \mid \ell_i \sqsubseteq \ell \ \&\ \texttt{declassify}(e_i, \ell_i) \text{ occurs in } c\}$ can be viewed as the union of $I_1 = \{i \mid \ell_i \sqsubseteq \ell \ \&\ \texttt{declassify}(e_i, \ell_i) \text{ occurs in } c_1\}$ and $I_2 = \{i \mid \ell_i \sqsubseteq \ell \ \&\ \texttt{declassify}(e_i, \ell_i) \text{ occurs in } c_2\}$. As $\langle M_1, c_1; c_2\rangle \Downarrow M_1'$ and $\langle M_2, c_1; c_2\rangle \Downarrow M_2'$ there are M_1'' and M_2'' so that $\langle M_1, c_1\rangle \Downarrow M_1''$ and $\langle M_2, c_1\rangle \Downarrow M_2''$. Because $I_1 \subseteq I$, we can apply the induction hypothesis to c_1. We receive $M_1'' =_\ell M_2''$. In order to show $M_1' =_\ell M_2'$ we would like apply the induction hypothesis to c_2. However, this requires that we demonstrate $\forall i \in I_2 . \langle M_1'', e_i\rangle \approx \langle M_2'', e_i\rangle$, which is different from what we know ($\forall i \in I_2 . \langle M_1, e_i\rangle \approx \langle M_2, e_i\rangle$). But because the effect system ensures that no variable used in $\{e_i\}_{i\in I_2}$ is updated by c_1, we infer that for any variable v such that v occurs in $\{e_i\}_{i\in I_2}$ we have $M_1(v) = M_1''(v)$ and $M_2(v) = M_2''(v)$. This assures $\forall i \in I_2 . \langle M_1'', e_i\rangle \approx \langle M_2'', e_i\rangle$, and hence, by the induction hypothesis, $M_1' =_\ell M_2'$.

`if` e `then` c_1 `else` c_2 Suppose $\Gamma \vdash e : \ell', D'$ for some ℓ' and D'. In case $\ell' \not\sqsubseteq \ell$, the *pc*-based mechanism of the type system ensures that only variables at or above ℓ' may be assigned to in c_1 and c_2. Thus, there are no assignments to variables at ℓ or below in either c_1 or c_2. Hence the memories M_1 and M_2 are unaffected below ℓ throughout the execution, which results in $M_1' =_\ell M_2'$.
If $\ell' \sqsubseteq \ell$, then, because, $\forall i \in \{i \mid \ell_i \sqsubseteq \ell\} . \langle M_1, e_i\rangle \approx \langle M_2, e_i\rangle$, including all occurrences of e_i under `declassify` in e, we have $\langle M_1, e\rangle \Downarrow \mathit{val}$ and $\langle M_2, e\rangle \Downarrow \mathit{val}$ for some val. Hence, for both M_1 and M_2, the computation will take the same branch, i.e., c_1, if $\mathit{val} = \mathit{true}$, or c_2 otherwise. That $M_1' =_\ell M_2'$ follows by the application of the induction hypothesis to c_1 or c_2, respectively.

`while` e `do` d Suppose $\Gamma \vdash e : \ell', D'$ for some ℓ' and D'. Case $\ell' \not\sqsubseteq \ell$ is resolved in the same fashion as for `if`. If $\ell' \sqsubseteq \ell$, then

$$\forall i \in \{i \mid \ell_i \sqsubseteq \ell\} . \langle M_1, e_i\rangle \approx \langle M_2, e_i\rangle$$

which includes all occurrences of e_i under `declassify` in e. Therefore, we have $\langle M_1, e\rangle \Downarrow \mathit{val}$ and $\langle M_2, e\rangle \Downarrow \mathit{val}$ for some val. Hence, for both M_1 and M_2, either the computation proceeds with d, if $\mathit{val} = \mathit{true}$, or terminates otherwise. Because the `while` loop terminates for both M_1 and M_2, the computation can be represented as a sequential composition of a series of d commands. This case reduces to consequently applying the sequential composition case. Note that e keeps evaluating to the same value under both M_1 and M_2 after each iteration (the effect system ensures that no variable used under `declassify` is updated by d). Hence, when e becomes *false*, the loop terminates after executing the same number of d commands for M_1 and M_2. This implies $M_1' =_\ell M_2'$. □