Localized Delimited Release:
Combining the What and Where Dimensions of Information Release

Aslan Askarov Andrei Sabelfeld
Department of Computer Science and Engineering, Chalmers University of Technology, 412 96 Göteborg, Sweden

Abstract

Information release (or declassification) policies are the key challenge for language-based information security. Although much progress has been made, different approaches to information release tend to address different aspects of information release. In a recent classification, these aspects are referred to as *what*, *who*, *where*, and *when* dimensions of declassification. In order to avoid information laundering, it is important to combine defense along the different dimensions. As a step in this direction, this paper presents a combination of *what* and *where* information release policies. Moreover, we show that a minor modification of a security type system from the literature (which was designed for treating the *what* dimension) in fact enforces the combination of *what* and *where* policies.

Categories and Subject Descriptors K.6.5 [*Management of Computing and Information Systems*]: Security and Protection

General Terms Security, Languages

Keywords Information flow, noninterference, downgrading, declassification, security policies.

1. Introduction

Information release (or declassification) policies are the key challenge for language-based information security [SM03, Zda04, SS05]. These policies ensure that legitimate information release is distinguished from information laundering (unintended leaks hidden by the system's release points). For example, a password checker should be able to legitimately leak whether the user's query matches the password. However, releasing the password itself regardless of the query would be an instance of laundering.

Much recent progress has been made in the area of information release policies. In a recent classification [SS05], these aspects are referred to as *what*, *who*, *where*, and *when* dimensions of declassification. However, different approaches to declassification tend to address different aspects of information release. We primarily focus on the *what* and *where* dimensions. Consider the following example:

$$l := \texttt{declassify}(h); c; l := h$$

where c is a command that does not update h. We assume that h is a secret (*high*) variable and l is a public (*low*) one. The declassification expression $\texttt{declassify}(h)$ means that the value of the high variable h should be declassified to low. Once the value of h is released at the beginning of the program, it can be freely used in subsequent assignments to l. Therefore, the above example is intuitively secure. Now consider a variation of the example (with the same restriction on c):

$$l := h; c; l := \texttt{declassify}(h)$$

This variation leaks the secret before it is legitimately declassified (which can be exploited if the value of l is output to the attacker before declassification). Nevertheless, this variation is accepted by security definitions (e.g., [Coh78, JL00, SS01, SM04, GM04, GM05]) that are based on the *what* dimension of information release. For these definitions, it is important that the value of h is leaked, but they ignore *where* in the program the leak occurs. Clearly, ignoring the *where* aspect is dangerous, when premature release is undesirable: for example, in an information-purchase scenario. To put it extremely: for batch-job programs, a policy based on the *what* dimension in isolation does not qualify for a declassification policy because it already assumes that secrets have been partially released at the program's start (and so the program does not actually declassify anything while running).

Consider another example, a simple password-checking fragment:

$$match := \texttt{declassify}(pwd == qry)$$

We assume that the variable pwd is high and $match$ is low (qry might be either high or low). The program checks whether the user query qry matches the password pwd and declassifies the boolean outcome of the match. Now consider a variation of the above example:

$$match := \texttt{declassify}(pwd)$$

By changing the argument to declassification, we are able to launder the entire password into the value of the $match$. Nevertheless, this variation is accepted by security definitions (e.g., [CM04, MS04, BS06, AS07]) that address the *where* dimension of information release. These definitions emphasize that leaks happen in a declassification-marked part of the code. However, these definitions are unable to distinguish leaks that reveal *something* about the password from leaks that reveal *everything* about the password. (A possibility to remedy this problem by connecting data to syntactic functions through which it can be released has been explored in the setting of relaxed noninterference [LZ05], although at a price of semantic consistency [SS05].)

The above examples indicate that in order to avoid information laundering, it is important to combine defense along the different dimensions.

PLAS'07 June 14, 2007, San Diego, California, USA.

As a step in this direction, this paper presents a combination of *what* and *where* information release policies. We capture the location of release by instrumenting the semantics with the set of released expressions and extending the definition of *delimited release* [SM04] with this information. The resulting security definition, *localized delimited release*, appears to satisfy the semantic consistency, conservativity, and non-occlusion principles of declassification [SS05]. Moreover, we show that a minor modification of a security type system from the literature [SM04], which was designed for enforcing the pure *what* delimited release policy in fact enforces the combination of *what* and *where* policies.

2. Language

To be concrete, we illustrate our approach on a simple imperative language whose expressions and commands are built according to Figure 1. The language contains a declassification expression $\mathtt{declassify}(e)$, which declassifies the value of e to low. For simplicity, but without loss of generality, we consider two levels of security: high and low.

The semantics of expressions are presented in Figure 2. Expression configurations have the form $\langle e, m, E\rangle$, where e is an expression, m is a memory (a mapping of variables to values), and E is a set of released expressions (*escape hatches*). Expression evaluation rules have the form $\langle e, m, E\rangle \downarrow \langle n, E'\rangle$ and perform total arithmetic computations while accumulating the set of released expressions. We define $m(e) = n$ whenever $\langle e, m, E\rangle \downarrow \langle n, E'\rangle$.

Command configurations have the form $\langle c, m, E\rangle$ where c is a command, m is a memory, and E is a set of released expressions. The command semantics are given in Figure 3. Transitions between command configurations have the form $\langle c, m, E\rangle \longrightarrow \langle c', m', E'\rangle$, which corresponds to a step of computation. A special case is a step to a designated command *stop* (which is not a part of the source language), $\langle c, m, E\rangle \longrightarrow \langle \mathit{stop}, m', E'\rangle$, which corresponds to a step that results in termination. The transition rules propagate the set of released expressions. As is standard, the relation $\longrightarrow^*$ denotes the reflexive and transitive closure of the relation $\longrightarrow$. We use notation $\langle c, m, E\rangle \Downarrow \langle m', E'\rangle$ to indicate the termination of $\langle c, m, E\rangle$ in $\langle m', E'\rangle$, i.e., whenever $\langle c, m, E\rangle \longrightarrow^* \langle \mathit{stop}, m', E'\rangle$. When the final configuration is unimportant, we simply write $\langle c, m, E\rangle \Downarrow$.

$$
\begin{aligned}
e &::= n \mid x \mid e\ \mathit{op}\ e \mid \mathtt{declassify}(e)\\
c &::= \mathtt{skip} \mid x := e \mid c; c\\
&\quad\mid \mathtt{if}\ e\ \mathtt{then}\ c\ \mathtt{else}\ c \mid \mathtt{while}\ e\ \mathtt{do}\ c
\end{aligned}
$$

Figure 1. Expression and command syntax

$$\langle n, m, E\rangle \downarrow \langle n, E\rangle \qquad \langle x, m, E\rangle \downarrow \langle m(x), E\rangle$$

$$\frac{\langle e_i, m, E\rangle \downarrow \langle n_i, E_i\rangle}{\langle e_1\ \mathit{op}\ e_2, m, E\rangle \downarrow \langle \mathit{op}(n_1, n_2), E_1 \cup E_2\rangle}$$

$$\frac{\langle e, m, E\rangle \downarrow \langle n, E'\rangle}{\langle \mathtt{declassify}(e), m, E\rangle \downarrow \langle n, E' \cup \{e\}\rangle}$$

Figure 2. Expression semantics

$$\langle \mathtt{skip}, m, E\rangle \longrightarrow \langle \mathit{stop}, m, E\rangle$$

$$\frac{\langle e, m, E\rangle \downarrow \langle n, E'\rangle}{\langle x := e, m, E\rangle \longrightarrow \langle \mathit{stop}, m[x \mapsto n], E'\rangle}$$

$$\frac{\langle c_1, m, E\rangle \longrightarrow \langle \mathit{stop}, m', E'\rangle}{\langle c_1; c_2, m, E\rangle \longrightarrow \langle c_2, m', E'\rangle}$$

$$\frac{\langle c_1, m, E\rangle \longrightarrow \langle c_1', m', E'\rangle}{\langle c_1; c_2, m, E\rangle \longrightarrow \langle c_1'; c_2, m', E'\rangle}$$

$$\frac{\langle e, m, E\rangle \downarrow \langle n, E'\rangle \qquad n \neq 0}{\langle \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, m, E\rangle \longrightarrow \langle c_1, m, E'\rangle}$$

$$\frac{\langle e, m, E\rangle \downarrow \langle 0, E'\rangle}{\langle \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, m, E\rangle \longrightarrow \langle c_2, m, E'\rangle}$$

$$\frac{\langle e, m, E\rangle \downarrow \langle n, E'\rangle \qquad n \neq 0}{\langle \mathtt{while}\ e\ \mathtt{do}\ c, m, E\rangle \longrightarrow \langle c; \mathtt{while}\ e\ \mathtt{do}\ c, m, E'\rangle}$$

$$\frac{\langle e, m, E\rangle \downarrow \langle 0, E'\rangle}{\langle \mathtt{while}\ e\ \mathtt{do}\ c, m, E\rangle \longrightarrow \langle \mathit{stop}, m, E'\rangle}$$

Figure 3. Command semantics

3. Security

The security definition builds on indistinguishability relations for memories and configurations that represent the attacker's view: related memories (or configurations) are indistinguishable by the attacker.

To begin with a simple indistinguishability relation, memories m_1 and m_2 are *low-equal* if they agree on low variables, i.e., $m_1(x) = m_2(x)$ for all low variables x.

A slightly more complex indistinguishability relation on memories is parameterized over the set of released expressions:

DEFINITION 1. *The indistinguishability relation on memories I induced by a set of expressions E is defined by $m_1\ I(E)\ m_2$ whenever $m_1(e) = m_2(e)$ for all $e \in E$.*

The intuition is that if a set of expressions E has been released, then the attacker may be able to distinguish more memories. For example, if the set is empty, then the relation $m_1\ I(\emptyset)\ m_2$ holds for all m_1 and m_2. If the high variable pwd has been released, then only memories that agree on pwd are indistinguishable, i.e., $m_1\ I(\{pwd\})\ m_2$ implies $m_1(pwd) = m_2(pwd)$. If the expression $pwd == qry$ has been released, then $m_1\ I(\{pwd == qry\})\ m_2$ implies that related memories must agree on the released expression, i.e., either $m_1(pwd) = m_1(qry)$ and $m_2(pwd) = m_2(qry)$ or $m_1(pwd) \neq m_1(qry)$ and $m_2(pwd) \neq m_2(qry)$. The larger the set of the released expressions, the more memories the attacker may distinguish and thus the smaller is the indistinguishability relation. The extreme case is the identity relation: if the set of released expressions E contains all program variables, then $m_1\ I(E)\ m_2$ if and only if $m_1 = m_2$. This means that the attacker has gathered full knowledge about all variables.

The following indistinguishability relation is on configurations. While the relation concerns full traces, its restrictions about information release are with respect to the initial memories, and hence the relation is parameterized over the initial memories:

DEFINITION 2 (Low bisimulation). *Given memories i_1 and i_2 a symmetric*[1] *relation R_{i_1,i_2} on configurations is an* i_1, i_2-low bisimulation *if, for all c_1, c_2, m_1, m_2, E_1, and E_2,*

[1] The up-front requirement on the symmetry of the relation justifies the liberty of being asymmetric under requirement $2(ii)$.

- $\langle c_1, m_1, E_1\rangle \Downarrow$, $\langle c_2, m_2, E_2\rangle \Downarrow$, *and*
- $\langle c_1, m_1, E_1\rangle \; R_{i_1,i_2} \; \langle c_2, m_2, E_2\rangle$

implies

1. $i_1 \; I(E_1) \; i_2$ *if and only if* $i_1 \; I(E_2) \; i_2$, *and*
2. *if* $i_1 \; I(E_1) \; i_2$ *then*
   (i) $m_1 =_L m_2$ *and*
   (ii) if $\langle c_1, m_1, E_1\rangle \longrightarrow \langle c_1', m_1', E_1'\rangle$ *then* $\langle c_2, m_2, E_2\rangle \longrightarrow^* \langle c_2', m_2', E_2'\rangle$ *and* $\langle c_1', m_1', E_1'\rangle \; R_{i_1,i_2} \; \langle c_2', m_2', E_2'\rangle$ *for some* c_2', m_2', *and* E_2'.

Two configurations cfg_1 *and* cfg_2 *are* i_1, i_2-low-bisimilar *(written* $cfg_1 \sim_{i_1,i_2} cfg_2$*) if there exists an* i_1, i_2*-low bisimulation that relates them.*

Note that an i_1, i_2-low bisimulation relates configurations whose released expression sets are compatible: i.e., the induced indistinguishability relations either both relate the initial memories i_1 and i_2 or both have them unrelated. If i_1 and i_2 are related, then the current memories m_1 and m_2 of these configurations must be low-equal. In addition, a step by one of the configurations is required to be simulated by zero or more steps of the other configuration so that the resulting configurations are related by the low-bisimulation relation.

The above definition is *termination-insensitive* (e.g., [SM03]) in the sense that diverging runs are ignored, an assumption that is commonly made. We are ready to present the *localized delimited release* security definition.

DEFINITION 3 (Localized delimited release). *A command c is secure if for all m_1 and m_2 such that $m_1 =_L m_2$ we have* $\langle c, m_1, \emptyset\rangle \sim_{m_1,m_2} \langle c, m_2, \emptyset\rangle$.

For a command to be secure, it is required that configurations that contain the command and start from some low-equal memories m_1 and m_2 with empty released expression sets are m_1, m_2-low-bisimilar.

Suppose that l is a low and $h, h_1, h_2, \dots$ are high variables. Consider the program:

$$l := h$$

Clearly, the set of released expressions is empty at all times. Recall that the relation $I(\emptyset)$ relates all memories. Hence, the low-equality of memories is imposed after executing the above command. Consider some memories m_1 and m_2 so that $m_1(l) = m_2(l) = 0$, $m_1(h) = 1$ and $m_2(h) = 2$. The configuration $\langle l := h, m_1, \emptyset\rangle$ terminates in one step in the memory m_1 where $m_1'(l) = 1$. The configuration $\langle l := h, m_2, \emptyset\rangle$ terminates in one step in the memory m_2', where $m_2'(l) = 2$. For simulating the former step under an m_1, m_2-low-bisimulation, it is required that either $m_1' =_L m_2$ or $m_1' =_L m_2'$, which is impossible. Hence, there is no m_1, m_2-low-bisimulation that relates $\langle l := h, m_1, \emptyset\rangle$ and $\langle l := h, m_2, \emptyset\rangle$; therefore, the program $l := h$ is insecure.

Consider the program:

$$l := \texttt{declassify}(h)$$

To see that this program legitimately declassifies h, take the relation $\{(\langle l := \texttt{declassify}(h), m_1, \emptyset\rangle, \langle l := \texttt{declassify}(h), m_2, \emptyset\rangle), (\langle stop, m_1', \{h\}\rangle, \langle stop, m_2', \{h\}\rangle)\}$, which is parameterized over m_1 and m_2 and where $\langle h, m_1, \emptyset\rangle \downarrow \langle n_1, \{h\}\rangle$, $\langle h, m_2, \emptyset\rangle \downarrow \langle n_2, \{h\}\rangle$, $m_1' = m_1[l \mapsto n_1]$, and $m_2' = m_2[l \mapsto n_2]$. If $m_1 =_L m_2$, then the above relation is an m_1, m_2-low-bisimulation. To see that a step by the configuration $\langle l := \texttt{declassify}(h), m_1, \emptyset\rangle$ is simulated by the configuration $\langle l := \texttt{declassify}(h), m_2, \emptyset\rangle$, it is sufficient to observe that $m_1 \; I(\{h\}) \; m_2$ only holds if $m_1(h) = m_2(h)$, in which case the required relation $m_1' =_L m_2'$ is vacuous. This shows how the release of h affects the requirement of the low-equality of the memories from the two related runs: after h is released, traces that disagree on the value of h at the point of release are automatically accepted by the definitions. On the other hand, traces that agree on the value of h are still required to be related, in order to avoid leaks through other variables.

Consider another example:

$$h_1 := h_2; l := \texttt{declassify}(h_1)$$

This program is insecure because it releases the value of h_2, which is masked as a release of the value of h_1. This is captured by the definition because starting from two initial memories m_1 and m_2 that agree on h_1 but disagree on h_2, the execution leads to final states that disagree on l. Because we have $m_1 \; I(\{h_1\}) \; m_2$, it is required that the final states must agree on l; hence, the program is rejected.

A similar effect is exhibited by the following program:

$$\texttt{if}\; h \;\texttt{then}\; l := \texttt{declassify}(h_1) \;\texttt{else skip}$$

Although the value of h_1 is declared as an escape hatch, whether h_1 has been declassified or not leaks some information about h. This insecurity is captured by the definition (consider initial states that agree on h_1 but disagree on h).

Consider a simple average salary example:

$$l := \texttt{declassify}((h_1 + \dots + h_n)/n)$$

The intention is to release the average salary out of the salaries stored in the variables $h_1 \dots h_n$, but no more information about the salaries. This program is secure because the differences in the value of the low outcome will only occur if there are differences in what is intended to be released (the average). On the other hand, the program:

$$h_2 := h_1; \dots; h_n := h_1; l := \texttt{declassify}((h_1 + \dots + h_n)/n)$$

which leaks the value of h_1 is rejected. The reason is that there is no m_1, m_2-low-bisimulation for the memories m_1 and m_2 that agree on all variables except for h_1 and h_2 but agree on the average. For example, $m_1(h_1) = m_2(h_2) = 0$, $m_1(h_2) = m_2(h_1) = 1$ and $m_1(x) = m_2(x)$ on all other variables x. Although $m_1 \; I(\{(h_1 + \dots + h_n)/n\}) \; m_2$, clearly the resulting low outcomes m_1' and m_2' are not low-equal ($m_1'(l) = 0$ and $m_2'(l) = 1$).

Yet another example is worth discussing:

$$h' := h; h := 0; l := \texttt{declassify}(h); h := h'; l := h$$

At the time of declassification, nothing is released. The actual release takes place at the end of the program. It is sometimes important that secrets are leaked only at the time of declassification. For example, this philosophy is adopted by our policy of *gradual release* [AS07], which rejects the program above. However, the rationale of localized delimited release, for a given piece of data, is to disallow data release *before* it is declassified but, on the other hand, allow release to take place any time *after* declassification. This justifies the acceptance of the above program by Definition 3.

4. Relation to delimited release

As mentioned before, our starting point for the definition is the delimited release policy [SM04]. Here, we recall this policy and explain how our definition improves it.

For the purpose of delimited release, consider the semantics that do not track the set of released expressions. Under these semantics, we have the following definition:

DEFINITION 4 (Delimited release). *Let the command c contain exactly n declassified expressions $e_1 \dots e_n$. Command c satisfies* delimited release *if whenever $m_1 =_L m_2$,* $\langle c, m_1\rangle \Downarrow m_1'$,

$\langle c, m_2\rangle \Downarrow m_2'$, *and for all* i *we have* $m_1(e_i) = m_2(e_i)$, *then* $m_1' =_L m_2'$.

A program satisfies the delimited release property if whenever escape hatch expressions cannot distinguish between two low-equal initial memories, then the whole program cannot distinguish between these memories.

While our definition is compatible with delimited release on the examples from the previous section, the benefit of tracking the set of released expression is illustrated on the following examples. The first one is from the introduction:

$$l := h; c; l := \texttt{declassify}(h)$$

where c is a command that does not update h. This example is accepted by delimited release because h is considered released even before the declassification statement is reached. However, localized delimited release rejects this program because when the set of released expressions is empty, then the memories are required to maintain low-equality as they pass through assignments. Clearly, this is not the case for the first assignment $l := h$. Another, in some sense more dangerous, example is:

$$h_2 := 0;$$
$$\texttt{if}\ h_1\ \texttt{then}\ l := \texttt{declassify}(h_1)\ \texttt{else}\ l := \texttt{declassify}(h_2)$$

This example is accepted by delimited release because *both* h_1 and h_2 are considered released *from the outset*. Note, however, that if the computation takes the `else` branch, then it never encounters a declassification of h_1. Nevertheless, the information about h_1 is leaked in the `else` branch.

The program above is rightfully rejected by our definition. Indeed, when starting with two memories m_1 and m_2 that agree on all variables (including h_2) except $m_1(h_1) \neq 0$ and $m_2(h_1) = 0$, then the indistinguishability relations $I(\{h_1\})$ and $I(\{h_2\})$ induced by the released expression sets of the two branches of the conditional clearly differ on m_1 and m_2: $m_1\ \neg I(\{h_1\})\ m_2$ but $m_1\ I(\{h_2\})\ m_2$.

The final example is an instance of occlusion (occlusion is discussed in more detail in the next section):

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

No program run passes through a declassification statement because it occurs in dead code. On the other hand, the insecure assignment $l := h$ is always reachable. Nevertheless, delimited release accepts this program as secure because h is declared as an escape hatch. As for the previous example, the new definition rightfully rejects the program because h never becomes a part of the set of released expressions.

These examples illustrate that localized delimited release strengthens the demands of delimited release by location sensitivity. The following theorem states that localized delimited release is a conservative extension of delimited release.

THEOREM 1. *If command c is secure, then c satisfies the delimited release security condition.*

Proof. By induction on the length of execution traces. The details are given in the appendix. □

This confirms the intuition that addressing both the *what* and *where* dimensions subsumes pure *what* definitions such as delimited release.

5. Relation to declassification principles

A classification of declassification [SS05] identifies four principles for declassification policies to serve as sanity checks for new definitions. Below we discuss the relation to all four principles.

The first principle is *semantic consistency*, which states that the (in)security of a given program should be preserved by semantics-preserving transformations of declassification-free code. This principle is satisfied by our definition because its restrictions are only in terms of declassification events: a modification of a declassification-free fragment of a program in a semantics preserving way (where semantic equivalence is defined as low bisimilarity that treats all variables as low) will not reflect on the (in)security of the program.

The second principle is *conservativity*, which states that the definition of security should be a conservative extension of noninterference: for programs without declassification, the security definition should be equivalent to noninterference. *Noninterference* [GM82] is a baseline policy that requires complete independence of low outputs from high inputs. This is the case for our security definition. In the absence of declassification expressions, the indistinguishability relation $I(\emptyset)$ relates all memories, which means that the low-equality of states is always a requirement on configurations that are related by low-bisimulation. Note, however, that our definition boils down to a fine-grained flavor of noninterference: starting with two low-equivalent states, any two terminating traces should have the same sequence of low memory updates. For example, this definition rejects the program $l := h; l := 0$, which would be accepted by a more permissive definition that only considers initial and final states. Our definition, however, has an advantage of being easily scalable to reasoning about intermediate inputs and outputs (the intermediate leak in the above program should not be output to the attacker).

The third principle is *monotonicity of release*, which states that adding declassification annotations to the code should not turn a secure program into an insecure one. A declassification annotation may only result in weakening the demands in the security condition. This principle is not supported by our definition. Consider the program:

$$h' := 0; \texttt{if}\ h\ \texttt{then}\ l := h'\ \texttt{else}\ l := 0$$

The program satisfies localized delimited release. However, adding a declassification annotation, as in:

$$h' := 0; \texttt{if}\ h\ \texttt{then}\ l := \texttt{declassify}(h')\ \texttt{else}\ l := 0$$

renders the program insecure. For two initial memories m_1 and m_2, where $m_1(h) = m_2(h) = 1$, $m_1(h') = m_2(h') = 0$ and $m_1(x) = m_2(x)$ on all other variables x, requirement 1 of Definition 2 is broken by any candidate for an m_1, m_2-low bisimulation. Indeed, for the final configurations: $m_1\ I(\emptyset)\ m_2$ but $m_1\ \neg I(\{h'\})\ m_2$. The principle fails because the definition treats declassifications with respect to initial states, ignoring the possibility that a declassification may become harmless in the current state (as declassifying 0 in the example above).

The fourth principle is *non-occlusion*, which states that the presence of declassifications should not mask other unrelated information leaks. As many other definitions along the *what* dimension of release, the security condition records exactly what differences in the memories that are allowed to leak to the attacker. Therefore, there is no possible occlusion as to what can be leaked to the attacker. As discussed in the previous section, this shows improvement over delimited release, which suffers from occlusion [SS07].

6. Type-based enforcement

Interestingly, a type system [SM04] that is designed for tracking delimited release is readily suitable for tracking the new definition. A minor modification of the type system turns out to be sound because it already keeps track of *where* data is released.

We display the typing rules in Figure 4. The only difference from the original type system is that we disallow declassification in a high context. This is needed in order to support requirement 1 of

$$\Gamma, pc \vdash n : \ell, \emptyset \qquad \frac{\Gamma(x) = \ell}{\Gamma, pc \vdash x : \ell, \emptyset}$$

$$\frac{\Gamma, pc \vdash e : \ell, D_1 \qquad \Gamma, pc \vdash e' : \ell, D_2}{\Gamma, pc \vdash e \; op \; e' : \ell, D_1 \cup D_2}$$

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

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

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

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

$$\frac{\begin{array}{c}\Gamma, pc \vdash e : \ell, D \\ \Gamma, \ell \sqcup pc \vdash c_1 : U_1, D_1 \qquad \Gamma, \ell \sqcup pc \vdash c_2 : U_2, D_2\end{array}}{\Gamma, pc \vdash \texttt{if}\; e \;\texttt{then}\; c_1 \;\texttt{else}\; c_2 : U_1 \cup U_2, D \cup D_1 \cup D_2}$$

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

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

Figure 4. Typing rules

Definition 2. As earlier, we assume a simple two-element security lattice where $low \sqsubseteq high$, $low \sqsubseteq low$, $high \sqsubseteq high$, and $high \not\sqsubseteq low$. Besides checking for explicit and implicit flows in a standard way [VSI96], the type system propagates two kinds of effects: sets U and D. The set U keeps track of variables that have been possibly updated by a command. The set D keeps track of the set of variables that have been possibly used in declassified expressions. The key constraint that the type system enforces is that variables that are involved in any declassified expression have never been updated prior to declassification. This guarantees that no new information has been introduced into these variables since the beginning of the execution. Keeping escape hatch expressions intact prior to their declassification ensures the soundness of the the type system with respect to delimited release [SM04].

As mentioned earlier, the type system is readily suitable for enforcing the new security definition. The fact that variables of declassified expressions are not updated before their declassification is a strong property: it guarantees that parts of memories critical to declassification are the same as in the initial memories. On the other hand, the low-bisimulation definition demands low-equality of the memories after declassification only if the memories before declassification are indistinguishable by escape hatches. Clearly, this demand is ensured by the type system because going through a declassification statement means adding the escape hatch expression in the set of released expressions in the semantics; indeed, the indistinguishability through this escape hatch through the initial and current memories is equivalent: the relation $i_1 \; I(E \cup \{e\}) \; i_2$ holds if and only if $m_1 \; I(E \cup \{e\}) \; m_2$ holds for memories m_1 and m_2 immediately before declassifying expression e. Clearly, if

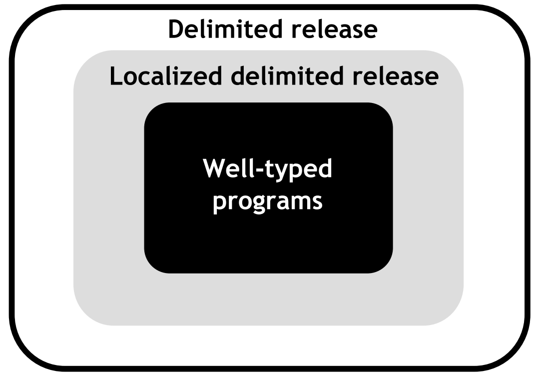

Figure 5. Localized delimited release in comparison

we have $m_1 \; I(E \cup \{e\}) \; m_2$ before declassifying e, then we obtain $m'_1 =_L m'_2$ for the respective memories m'_1 and m'_2 after the declassification step.

This leads us to the following soundness result:

THEOREM 2. *If command c is typable, then c satisfies localized delimited release.*

Proof. By induction on the type derivation. The details are given in the appendix. □

It is straightforward to see that the type system accepts the first two examples from Section 3 and, by soundness, rejects all insecure examples from Sections 3 and 4.

Figure 5 provides a pictorial representation of Theorems 1 and 2: programs with localized delimited release satisfy the basic delimited release policy (Theorem 1); and typable programs satisfy the localized delimited release policy (Theorem 2).

7. Related work

This paper combines two dimensions of information release: *what* and *where*, and so we concentrate on these two dimensions when discussing related work. For a comprehensive discussion of related work on declassification we refer to the classification of declassification [SS05, SS07]. Related work within the area of language-based information-flow security is discussed in a survey of the area [SM03].

A natural starting point along the *what* dimension is the delimited release policy [SM04]. Delimited release has its roots in *independence* [Coh78] policies, which have been generalized by approaches based on *abstract variables* [JL00], *partial equivalence relations* [ABHR99, SS01, Pro01], and *abstract noninterference* [GM04, GM05]. Delimited release is a natural starting point because it has an appealingly simple mechanism for extracting escape hatch expressions from the code, and because it comes with a type-based enforcement mechanism that turns out to handle the *where* dimension of information release.

As mentioned before, local policies for *relaxed noninterference* [LZ05] are related to combining the *what* and *where* of declassification. Under relaxed noninterference, subprograms are labeled with syntactic representations of functions that describe how an integer can be leaked. The syntactic nature of the definition, however, leads to the loss of semantic consistency: renaming functions might lead to changes in the security of the program.

Interestingly, there is no equally natural starting point along the *where* or *when* dimension (some *when* policies are capable of expressing code locality for release policies). A popular approach to policies along the *where* dimension is based on *intransitive nonin-*

terference [Rus92, Pin95, RS99, RG99, Mul00, Man01, BPR04]. Informally, intransitive noninterference requires noninterference between declassification events. In contrast to the presented condition, there are no guarantees for traces with declassifications in their entirety. Although there has been work (e.g., [CM04, MS04, BS06, AS07]) on mapping intransitive noninterference to a language-based setting, it does not fully address the *what* dimension and thus is vulnerable to attacks such as the password laundering attack from the introduction.

Most recently, and independently, Mantel and Reinhard [MR07] suggested three definitions ($WHAT_1$, $WHAT_2$, and $WHERE$) for controlling the *what* and *where* dimensions of declassification. However, they pursue somewhat different goals: compositional timing-sensitive security. Another difference is that their definitions consider the dimensions in separation: it is not possible to explicitly specify where a particular piece of data can be released. For a simple example, consider a scenario of releasing the average of salaries $h_1 \dots h_n$ and, some time later, possibly under some conditions, releasing the actual salaries. (Perhaps when a company goes bankrupt, the salaries have to be revealed to the authorities.) A legitimate program for these purposes is:

$l := \texttt{declassify}((h_1 + \ldots + h_n)/n);$
$\ldots$ // some code that does not assign to $h_1 \dots h_n$
$l_1 := \texttt{declassify}(h_1); \ldots ; l_n := \texttt{declassify}(h_n)$

This is a reasonable program, and it is accepted by most definitions under discussion, including localized delimited release. Consider the following occlusion attack:

$h'_2 := h_2; \ldots ; h'_n := h_n;$
$h_2 := h_1; \ldots ; h_n := h_1;$
$l := \texttt{declassify}((h_1 + \ldots + h_n)/n);$
$h_2 := h'_2; \ldots ; h_n := h'_n;$ $\quad //h'_1 \dots h'_n$ are not used further
$\ldots$ // some code that does not assign to $h_1 \dots h_n$
$l_1 := \texttt{declassify}(h_1); \ldots ; l_n := \texttt{declassify}(h_n)$

Clearly, the first declassification prematurely releases h_1. However, the attack is accepted by all of the $WHAT_1$, $WHAT_2$, and $WHERE$ definitions because of the independent treatment of the dimensions. On the other hand, this program is rejected by localized delimited release (similarly to the attack in the average example of Section 3) because escape hatch policies of localized delimited release are location-sensitive.

8. Conclusion

We have presented localized delimited release, a security characterization that combines the *what* and *where* dimensions of information release. Localized delimited release is a fully fledged combination of these dimensions. This combination offers protection from information laundering, which is reassured by the semantic consistency, conservativity, and non-occlusion principles of declassification [SS05].

Future work concerns including the *who* and *when* dimensions in the security definition and possibilities for parameterizing both the definition and enforcement mechanism in the degree of security along each dimension. Another direction is enforcement mechanisms that are more permissive than the type system discussed in the paper. In particular, we are interested in enforcing the security condition at the level of Java bytecode.

Acknowledgments

Thanks are due to David Naumann and David Sands for fruitful discussions on an earlier draft of this paper and to Heiko Mantel and Alexander Reinhard for helpful comments. This work was funded in part by the Swedish Research Council and in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies, under the IST-2005-015905 Mobius project.

References

[ABHR99] 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.

[AS07] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proc. IEEE Symp. on Security and Privacy*, May 2007.

[BPR04] A. Bossi, C. Piazza, and S. Rossi. Modelling downgrading in information flow security. In *Proc. IEEE Computer Security Foundations Workshop*, pages 187–201, June 2004.

[BS06] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Proc. European Symp. on Programming*, volume 3924 of *LNCS*, pages 180–196. Springer-Verlag, 2006.

[CM04] S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, October 2004.

[Coh78] 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.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

[GM04] 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.

[GM05] R. Giacobazzi and I. Mastroeni. Adjoining declassification and attack models by abstract interpretation. In *Proc. European Symp. on Programming*, volume 3444 of *LNCS*, pages 295–310. Springer-Verlag, April 2005.

[JL00] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.

[LZ05] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 158–170, January 2005.

[Man01] 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.

[MR07] H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In *Proc. European Symp. on Programming*, volume 4421 of *LNCS*, pages 141–156. Springer-Verlag, 2007.

[MS04] H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 129–145. Springer-Verlag, November 2004.

[Mul00] J. Mullins. Non-deterministic admissible interference. *J. of Universal Computer Science*, 6(11):1054–1070, 2000.

[Pin95] S. Pinsky. Absorbing covers and intransitive non-interference. In *Proc. IEEE Symp. on Security and Privacy*, pages 102–113, May 1995.

[Pro01] F. Prost. On the semantics of non-interference type-based analyses. In *JFLA'001, Journées Francophones des Langages Applicatifs*, January 2001.

[RG99] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proc. IEEE Computer Security Foundations Workshop*, pages 228–238, June 1999.

[RS99] P. Ryan and S. Schneider. Process algebra and non-interference. In *Proc. IEEE Computer Security Foundations Workshop*, pages 214–227, June 1999.

[Rus92] J. M. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, 1992.

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[SM04] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, October 2004.

[SS01] 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.

[SS05] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.

[SS07] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, 2007. To appear.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[Zda04] S. Zdancewic. Challenges for information-flow security. In *Proc. Programming Language Interference and Dependence (PLID)*, August 2004.

Appendix

This appendix presents the proofs of Theorems 1 and 2.

THEOREM 1. *If command c is secure, then c satisfies the delimited release security condition.*

Proof. Assume the command c contains exactly n declassified expressions $e_1 \dots e_n$ and assume for memories m_1 and m_2 that $m_1 =_L m_2$, $\langle c, m_1\rangle \Downarrow m_1'$, $\langle c, m_2\rangle \Downarrow m_2'$, and for all i we have $m_1(e_i) = m_2(e_i)$. In order to show that c satisfies delimited release, we need to prove $m_1' =_L m_2'$.

By the definition of localized delimited release, $m_1 =_L m_2$ implies $\langle c, m_1, \emptyset\rangle \sim_{m_1,m_2} \langle c, m_2, \emptyset\rangle$. Because both configurations terminate, there exist p and r such that we can write down the execution steps of $\langle c, m_1, \emptyset\rangle$ and $\langle c, m_2, \emptyset\rangle$ in our semantics as follows:

$$\begin{aligned}\langle c, m_1, \emptyset\rangle = \langle c_1^0, m_1^0, E_1^0\rangle \longrightarrow \langle c_1^1, m_1^1, E_1^1\rangle \longrightarrow \dots\\ \longrightarrow \langle c_1^p, m_1^p, E_1^p\rangle = \langle stop, m_1', E_1^p\rangle\end{aligned}$$

$$\begin{aligned}\langle c, m_2, \emptyset\rangle = \langle c_2^0, m_2^0, E_2^0\rangle \longrightarrow \langle c_2^1, m_2^1, E_2^1\rangle \longrightarrow \dots\\ \longrightarrow \langle c_2^r, m_2^r, E_2^r\rangle = \langle stop, m_2', E_1^r\rangle\end{aligned}$$

By bisimilarity, there exists a low bisimulation R_{m_1,m_2} such that $\langle c, m_1, \emptyset\rangle\ R_{m_1,m_2}\ \langle c, m_2, \emptyset\rangle$. We know that $\langle c, m_1, \emptyset\rangle \Downarrow$ and $\langle c, m_2, \emptyset\rangle \Downarrow$. By requirement $2(ii)$ of Definition 2, $\langle c, m_2, \emptyset\rangle \longrightarrow^* \langle c_2^q, m_2^q, E_2^q\rangle$ for some $q \in \{0 \dots r\}$ so that $\langle c_1^1, m_1^1, E_1^1\rangle\ R_{m_1,m_2}\ \langle c_2^q, m_2^q, E_2^q\rangle$. By simple induction on p, we obtain that there exists $t \in \{0 \dots r\}$ such that $\langle stop, m_1^p, E_1^p\rangle\ R_{m_1,m_2}\ \langle c_2^t, m_2^t, E_2^t\rangle$ and thus $\langle stop, m_1', E_1^p\rangle\ R_{m_1,m_2}\ \langle c_2^t, m_2^t, E_2^t\rangle$. Because R_{m_1,m_2} is symmetric, we have $\langle c_2^t, m_2^t, E_2^t\rangle\ R_{m_1,m_2}\ \langle stop, m_1', E_1^p\rangle$. We can now repeatedly apply requirement $2(ii)$ of Definition 2 to the execution of $\langle c_2^t, m_2^t, E_2^t\rangle$ to finally obtain $\langle stop, m_2^r, E_2^r\rangle\ R_{m_1,m_2}\ \langle stop, m_1^p, E_1^p\rangle$ and thus $\langle stop, m_1', E_1^p\rangle\ R_{m_1,m_2}\ \langle stop, m_2', E_2^r\rangle$. We have $m_1\ I(E_1^p)\ m_2$ because E_1^p is subsumed by the set of all escape hatch expressions ($E_1^p \subseteq \cup_{i=1\dots n}\{e_i\}$), and because m_1 and m_2 agree on all escape hatch expressions ($m_1(e_i) = m_2(e_i)$ for all i). Hence, by requirement $2(i)$ of Definition 2, we conclude $m_1' =_L m_2'$. □

THEOREM 2. *If command c is typable, then c satisfies localized delimited release.*

Proof. By induction on the type derivation. The case of the subsumption rule is straightforward. Suppose $m_1 =_L m_2$, $\langle c, m_1, \emptyset\rangle \Downarrow \langle m_1', E_1\rangle$, and $\langle c, m_2, \emptyset\rangle \Downarrow \langle m_2', E_2\rangle$. The rest of the rules are structural:

`skip` Relation $R_{m_1,m_2} = \{(\langle \texttt{skip}, m_1, \emptyset\rangle, \langle \texttt{skip}, m_2, \emptyset\rangle), (\langle stop, m_1, \emptyset\rangle, \langle stop, m_2, \emptyset\rangle)\}$ is an m_1, m_2-low bisimulation because the low-equality of m_1 and m_2 is preserved by a computation step.

$x := e$ Consider $R_{m_1,m_2} = \{(\langle x := e, m_1, \emptyset\rangle, \langle x := e, m_2, \emptyset\rangle), (\langle stop, m_1', E\rangle, \langle stop, m_2', E\rangle)\}$. We need to show that R_{m_1,m_2} is an m_1, m_2-low bisimulation. Clearly, $E_1 = E_2 = E$ for some E, which preserves requirement 1 after one step. If x is high, then $m_1' =_L m_2'$, which preserves requirement $2(i)$ after one step.

If x is low, we know that E is the (possibly empty) set of declassified expressions in expression e. Suppose $m_1\ I(E)\ m_2$, which means that all escape hatch expressions in e agree on the memories m_1 and m_2. Therefore, $m_1' =_L m_2'$, which preserves requirement $2(i)$ after one step.

$c_1; c_2$ Both c_1 and c_2 must be typable. Assume $\langle c_1, m_1, \emptyset\rangle \Downarrow \langle m_1'', E_1''\rangle$ and $\langle c_1, m_2, \emptyset\rangle \Downarrow \langle m_2'', E_2''\rangle$. By induction hypothesis, $\langle c_1, m_1, \emptyset\rangle \sim_{m_1,m_2} \langle c_1, m_2, \emptyset\rangle$. Hence, there is an m_1, m_2-low bisimulation $R^1_{m_1,m_2}$, where $\langle c_1, m_1, \emptyset\rangle R^1_{m_1,m_2} \langle c_1, m_2, \emptyset\rangle$.

If $\langle stop, m_1'', E_1''\rangle \neg R^1_{m_1,m_2} \langle stop, m_2'', E_2''\rangle$ or $m_1\ \neg I(E_1'')\ m_2$, then the following relation

$$\begin{aligned}R_{m_1,m_2} = \{(\langle c_1^1; c_2, m_1''', E_1'''\rangle, \langle c_1^2; c_2, m_2''', E_2'''\rangle) \mid\\ \langle c_1, m_1, \emptyset\rangle \longrightarrow^* \langle c_1^1, m_1''', E_1'''\rangle\ \&\\ \langle c_1, m_2, \emptyset\rangle \longrightarrow^* \langle c_1^2, m_2''', E_2'''\rangle\ \&\\ \langle c_1^1, m_1''', E_1'''\rangle\ R^1_{m_1,m_2}\ \langle c_1^2, m_2''', E_2'''\rangle\}\end{aligned}$$

is an m_1, m_2-low bisimulation for $c_1; c_2$.

If $\langle stop, m_1'', E_1''\rangle R^1_{m_1,m_2} \langle stop, m_2'', E_2''\rangle$ and $m_1\ I(E_1'')\ m_2$, then $m_1'' =_L m_2''$ by requirement $2(i)$ for $R^1_{m_1,m_2}$. By induction hypothesis, $\langle c_2, m_1'', \emptyset\rangle \sim_{m_1'',m_2''} \langle c_2, m_2'', \emptyset\rangle$, i.e., there is an m_1'', m_2''-low bisimulation $R^2_{m_1,m_2}$, where $\langle c_2, m_1'', \emptyset\rangle R^2_{m_1'',m_2''} \langle c_2, m_2'', \emptyset\rangle$. Consider relation

$$\begin{aligned}R_{m_1,m_2} = \{(\langle c_1^1; c_2, m_1''', E_1'''\rangle, \langle c_1^2; c_2, m_2''', E_2'''\rangle) \mid\\ \langle c_1, m_1, \emptyset\rangle \longrightarrow^* \langle c_1^1, m_1''', E_1'''\rangle\ \&\\ \langle c_1, m_2, \emptyset\rangle \longrightarrow^* \langle c_1^2, m_2''', E_2'''\rangle\ \&\\ \langle c_1^1, m_1''', E_1'''\rangle\ R^1_{m_1,m_2}\ \langle c_1^2, m_2''', E_2'''\rangle\}\\ \cup \{(\langle c_2^1, m_1''', E_1'' \cup E_1'''\rangle, \langle c_2^2, m_2''', E_2'' \cup E_2'''\rangle) \mid\\ \langle c_2, m_1'', \emptyset\rangle \longrightarrow^* \langle c_2^1, m_1''', E_1'''\rangle\ \&\\ \langle c_2, m_2'', \emptyset\rangle \longrightarrow^* \langle c_2^2, m_2''', E_2'''\rangle\ \&\\ \langle c_2^1, m_1''', E_1'''\rangle\ R^2_{m_1'',m_2''}\ \langle c_2^2, m_2''', E_2'''\rangle\}\end{aligned}$$

Clearly, $\langle c_1; c_2, m_1, \emptyset\rangle\ R_{m_1,m_2}\ \langle c_1; c_2, m_2, \emptyset\rangle$. To see that this relation is an m_1, m_2-low bisimulation, we assume that two terminating configurations cfg_1 and cfg_2 are related $cfg_1\ R_{m_1,m_2}\ cfg_2$ and show that requirements 1 and 2 are satisfied. We consider two cases.

In the first case, we obtain $cfg_1 = \langle c_1^1; c_2, m_1''', E_1''' \rangle$ and $cfg_2 = \langle c_1^2; c_2, m_2''', E_2''' \rangle$, where we have $\langle c_1, m_1, \emptyset \rangle \longrightarrow^* \langle c_1^1, m_1''', E_1''' \rangle$, $\langle c_1, m_2, \emptyset \rangle \longrightarrow^* \langle c_1^2, m_2''', E_2''' \rangle$, and $\langle c_1^1, m_1''', E_1''' \rangle \; R^1_{m_1,m_2} \; \langle c_1^2, m_2''', E_2''' \rangle$. The latter gives us requirements 1 and $2(i)$ for relation R_{m_1,m_2}.

Assume $cfg_1 \longrightarrow cfg_1'$. If c_1^1 has not terminated, then requirement $2(ii)$ follows from requirement $2(ii)$ of $R^1_{m_1,m_2}$. The interesting case is $cfg_1 = \langle c_1^1; c_2, m_1''', E_1''' \rangle \longrightarrow \langle c_2, m_1'', E_1'' \rangle$ and $\langle c_1^1, m_1''', E_1''' \rangle \longrightarrow \langle stop, m_1'', E_1'' \rangle$. In this case, we also know that $\langle stop, m_1'', E_1'' \rangle \; R^1_{m_1,m_2} \; \langle stop, m_2'', E_2'' \rangle$. Therefore, with $cfg_2 \longrightarrow^* cfg_2' = \langle c_2, m_2'', E_2'' \rangle$ and recalling that $\langle c_2, m_1'', \emptyset \rangle \; R^2_{m_1'',m_2''} \langle c_2, m_2'', \emptyset \rangle$ we note that $\langle c_2, m_1'', E_1'' \rangle$ and $\langle c_2, m_2'', E_2'' \rangle$ are indeed related by R_{m_1,m_2}, which gives us requirement $2(ii)$.

In the second case, we obtain $cfg_1 = \langle c_2^1, m_1''', E_1'' \cup E_1''' \rangle$ and $cfg_2 = \langle c_2^2, m_2''', E_2'' \cup E_2''' \rangle$, where we have $\langle c_2, m_1'', \emptyset \rangle \longrightarrow^* \langle c_2^1, m_1''', E_1''' \rangle$, $\langle c_2, m_2'', \emptyset \rangle \longrightarrow^* \langle c_2^2, m_2''', E_2''' \rangle$, and $\langle c_2^1, m_1''', E_1''' \rangle \; R^2_{m_1'',m_2''} \; \langle c_2^2, m_2''', E_2''' \rangle$.

The key observation of the proof is that $U_1 \cap D_2 = \emptyset$ from the typing rule for the sequential composition implies:

$$m_1 \; I(E_i''') \; m_2 \Leftrightarrow m_1'' \; I(E_i''') \; m_2'', \quad i = 1, 2$$

In other words, variables that appear in expressions that may be declassified in c_2 (namely, expressions from E_i'''), are never updated in c_1. Therefore, memories that agree on these expressions before starting c_1 just as well agree on them before c_2, and visa versa.

To show requirement 1 assume $m_1 \; I(E_1'' \cup E_1''') \; m_2$. We have $m_1 \; I(E_1'' \cup E_1''') \; m_2 \Leftrightarrow m_1 \; I(E_1'') \; m_2 \wedge m_1 \; I(E_1''') \; m_2$. By induction hypothesis $m_1 \; I(E_1'') \; m_2 \Leftrightarrow m_1 \; I(E_2'') \; m_2$. Also, by the above observation and the induction hypothesis $m_1 \; I(E_1''') \; m_2 \Leftrightarrow m_1'' \; I(E_1''') \; m_2'' \Leftrightarrow m_1'' \; I(E_2''') \; m_2'' \Leftrightarrow m_1 \; I(E_2''') \; m_2$. Hence, $m_1 \; I(E_2'' \cup E_2''') \; m_2$.

For requirement 2, assume $m_1 \; I(E_1'' \cup E_1''') \; m_2$. This implies $m_1 \; I(E_1''') \; m_2$. Applying induction hypothesis we obtain $2(i)$ from $2(i)$ for $R^2_{m_1'',m_2''}$. Finally, requirement $2(ii)$ follows from the construction of $R_{m_1'',m_2''}$.

`if` e `then` c_1 `else` c_2 From $\Gamma, pc \vdash e : \ell, D$ we obtain two cases. If $\ell \sqcup pc = high$, then since commands c_1 and c_2 are both typable in high context, no low updates or declassifications are allowed in these commands. This implies $D_1 = D_2 = \emptyset$. Consider the relation $R_{m_1,m_2} = \{(\langle c_1', m_1', E \rangle, \langle c_2', m_2', E \rangle \mid \langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m_i', \emptyset \rangle \longrightarrow^* \langle c_i', m_i', E \rangle\}$, where E is the (possibly empty) set of released expressions obtained by evaluating the guard e. This relation trivially satisfies requirement 1. Next, $m_1 \; I(E) \; m_2$ implies $m_1 =_L m_2$. Moreover, because no low updates are allowed in c_1 and c_2, low-equivalence of the updated memories is preserved along the execution, which, by transitivity, gives us the requirement $2(i)$. Finally, $2(ii)$ holds by construction of R_{m_1,m_2}.

If $\ell \sqcup pc = low$, then by induction hypothesis there exists a pair of relations $R^1_{m_1,m_2}$ and $R^2_{m_1,m_2}$ such that $\langle c_1, m_1, \emptyset \rangle \; R^1_{m_1,m_2} \; \langle c_1, m_2, \emptyset \rangle$, and $\langle c_2, m_1, \emptyset \rangle \; R^2_{m_1,m_2} \; \langle c_2, m_2, \emptyset \rangle$. Consider the relation $R_{m_1,m_2} = \{(\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m_1, \emptyset \rangle, \langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m_2, \emptyset \rangle)\} \cup_{j=1}^{2} \{(\langle c_j^1, m_1', E \cup E_1' \rangle, \langle c_j^2, m_2, E \cup E_2' \rangle) \mid \langle c_j^1, m_1', E_1' \rangle \; R^j_{m_1,m_2} \; \langle c_j^2, m_2', E_2' \rangle\}$, where E is the (possibly empty) set of released expressions obtained by evaluating the guard e. To ensure that this relation is an m_1, m_2-low bisimulation we show that it satisfies requirements 1 and 2.

Applying induction hypothesis, $m_1 \; I(E \cup E_1') \; m_2 \Leftrightarrow m_1 \, I(E) \, m_2 \wedge m_1 \, I(E_1') \, m_2 \Leftrightarrow m_1 \, I(E) \, m_2 \wedge m_1 \, I(E_2') \, m_2 \Leftrightarrow m_1 \; I(E \cup E_2') \; m_2$, which shows requirement 1.

For requirement 2, assume $m_1 \; I(E \cup E_1') \; m_2$, which implies $m_1 \; I(E) \; m_2$ and $m_1 \; I(E_1') \; m_2$. Then, $m_1 \; I(E) \; m_2$ implies that both runs agree on the value of the guard, and hence take the same branch, say c_1. We can then apply the induction hypothesis for that branch and $m_1 \; I(E_1') \; m_2$ gives us $m_1 =_L m_2$, which establishes $2(i)$. Similarly, $2(ii)$ follows from the induction hypothesis.

`while` e `do` c_1 Because only terminating runs are of interest, the proof is a combination of cases for the sequential composition and conditionals.

$\square$