Declassification predicates for controlled information release

Graeme Smith[0000−0003−1019−4761]

School of Information Technology and Electrical Engineering,
The University of Queensland, Australia

Abstract. Declassification refers to the controlled release of sensitive information by a program. It is well recognised that security analyses, formal or otherwise, need to take declassification into account to be practically usable. This paper introduces the concept of *declassification predicates* which enable a programmer to define precisely *what* can be declassified in a given program, and *where* in the program it can be released. We show how declassification predicates can be added to an existing information flow logic, and how the extended logic can be implemented within the Dafny program verifier.

1 Introduction

Information flow analyses track the flow of data through a program, and can hence detect when sensitive data flows to program locations which are considered to be accessible by an attacker. Such analyses range from simple security type systems [15, 12] to more advanced logics which use predicates to represent the program state and the potentially dynamic security classifications of program variables [11, 16]. These analysis techniques typically equate security with the notion of *non-interference* [5]. Non-interference holds when sensitive data does not influence the data held in program locations considered to be accessible to an attacker.

While non-interference provides a highly intuitive notion of information flow security, it is too strict to be used with many programs in practice. Consider, for example, a simple password checking program. In such a program, a user's password would be regarded as sensitive data. But this sensitive data influences what an attacker sees (success or failure) when they enter a guess for the password. Other examples include a program in which employees' salaries are sensitive, but their average may be released for statistical purposes, or a program which determines (and hence releases) whether a financial transaction can proceed although the balance of the account from which the money is being transferred is sensitive. In all cases, the information which is released, and hence accessible to an attacker, is influenced by the sensitive data.

This realisation has led to a range of proposals for weakening non-interference to allow *controlled* declassification of sensitive information. Sabelfeld and Sands [14] identify four dimensions of declassification related to how the release of

information is controlled: *what* information can be released; *where* in the program the information can be released; *who* can release the information; and *when* the information can be released. While approaches considering the who and when dimensions have been developed, e.g., Zdancewic and Myers consider the who dimension with their notion of *robust declassification* [17], there has been a significant focus on the what and where dimensions.

To capture the what dimension of declassification, Sabelfeld and Myers [13] introduce the concept of *delimited release*. It requires the programmer to specify a number of expressions in a program, referred to as *escape hatches*, whose values can be released. An escape hatch is specified in the program via a declassify annotation. When such an escape hatch is specified, other occurrences of the expression, either before or after the annotated expression, are also declassified. Hence, the notion is concerned solely with *what* information can be released and not *where* in the program it can be released.

In contrast, Mantel and Sands define a notion of security based on *intransitive non-interference* [8] which considers solely the where dimension. Declassification is allowed at any annotated program step, but only at annotated program steps. A similar approach is provided by Mantel and Reinhard [7] along with two notions of security to capture the what dimension of declassification.

To capture both the what and where dimensions in a single definition of security, Askarov and Sabelfeld introduce *localized delimited release* [3]. As with delimited release, the programmer specifies the expressions which can be released via a declassify annotation. However, only the annotated occurrence of a particular expression, and those following it, are declassified. The annotation therefore marks the point in the program from which the expression becomes declassified.

In this paper we present a new notion of security, *qualified release*, which combines the what and where dimensions, and demonstrate that it is more general, i.e., applicable to more programs, than localized delimited release. The key to our definition is a concept we call *declassification predicates*. These are predicates that must be true at the point in the program where declassification occurs. They can be used to describe a range of values that an escape-hatch expression can take at the point where its value is released. We begin in Section 2 with an overview of the earlier work described above. Then, in Section 3, we demonstrate how declassification predicates can overcome the shortcomings of existing approaches and define the notion of qualified release. In Section 4, we show how qualified release can be enforced by encoding declassification predicates in an existing information flow logic from Winter et al. [16]. In Section 5, we present an implementation of the logic with declassification predicates in the Dafny program verification tool [6]. We conclude in Section 6.

2 Exisiting approaches

2.1 Non-interference

In information flow analyses in which declassification is not considered, the standard definition of security is *non-interference* [5]. Given a lattice of security clas-

sifications, non-interference states that data classified at a level lower than or equal to a given level ℓ cannot be influenced by data at levels higher than ℓ. When ℓ is the highest level of data accessible to an attacker, this means the attacker cannot deduce anything about the data classified higher than level ℓ. We refer to data classified at a security level higher than ℓ as *high data* and all other data as *low data*. Similarly, a variable that is able to hold high or low data is referred to as a *high variable*, and a variable that is only able to hold low data as a *low variable*.

Non-interference can be formally defined in terms of comparing two runs of the program under consideration. For concurrent programs, if the runs start with the same low data then at each point in the program the low data should remain the same [9].[1] Let a program configuration $\langle c, m\rangle$ denote a program c and memory, i.e., mapping of variables to values, m, and let $\langle c, m\rangle \longrightarrow \langle c', m'\rangle$ and $\langle c, m\rangle \longrightarrow^n \langle c', m'\rangle$ denote that $\langle c, m\rangle$ reaches configuration $\langle c', m'\rangle$ in one or $n \geq 1$ program steps, respectively. Let $m_1 =_\ell m_2$ denote that memories m_1 and m_2 are indistinguishable at security levels ℓ and below. This relation between memories is referred to as *low equivalence*.

Definition 1. Non-interference for program c
Program c is secure if the following holds.

$$\begin{array}{l}\forall\, m_1, m_2, n, c', m_1' \cdot \\ \quad m_1 =_\ell m_2 \wedge \langle c, m_1\rangle \longrightarrow^n \langle c', m_1'\rangle \Rightarrow \\ \quad\quad \exists\, m_2', c'' \cdot \langle c, m_2\rangle \longrightarrow^n \langle c'', m_2'\rangle \wedge m_1' =_\ell m_2'\end{array}$$

This can equally be defined in terms of a bisimulation over configurations. Here we use an intuitive definition based on that for sequential programs. The premise of the implication in the above definition does not require the high data to be the same in m_1 and m_2. Hence, if high data which differs between m_1 and m_2 is declassified in c and then assigned to a low variable, the condition fails. Non-interference is, therefore, too strong to use in the presence of declassification.

2.2 Delimited release

Delimited release [13] is a weakening of non-interference that allows identified expressions, referred to as *escape hatches*, to be declassified in a program. Escape hatches are identified as annotated expressions of the form $\mathsf{declassify}(e, d)$ where e is the expression to be declassified, i.e., e is the escape hatch, and d the level to which it is to be declassified. It is a requirement that d is not higher than ℓ, the security level accessible by an attacker.

Under delimited release, the condition $m_1' =_\ell m_2'$ in the consequent of Definition 1 need only hold when all escape hatches have the same value in the initial memories m_1 and m_2. Let $m_1\, I(E)\, m_2$ denote that all expressions in set E are indistinguishable on memories m_1 and m_2.

[1] When concurrency is not a consideration, this definition (as well as the others in this section) can be weakened to only consider the point where the program terminates.

Definition 2. Delimited release of escape hatches E on program c
Program c is secure if the following holds.

$$\forall\, m_1, m_2, n, c', m_1' \cdot \\ m_1 =_\ell m_2 \wedge \langle c, m_1\rangle \longrightarrow^n \langle c', m_1'\rangle \Rightarrow \\ \exists\, m_2', c'' \cdot \langle c, m_2\rangle \longrightarrow^n \langle c'', m_2'\rangle \wedge (m_1\, I(E)\, m_2 \Rightarrow m_1' =_\ell m_2')$$

This definition obviously equates to non-interference when there are no declassified expressions in a program, i.e., when $E = \varnothing$. To illustrate how it captures the what dimension of declassification, consider the following examples adapted from [13].

Example 1. Average salary

A program uses high variables $h_1, \ldots, h_n$ to store the salaries of employees in an organisation. While these salaries are sensitive and have a security level higher than ℓ (the level accessible by a potential attacker), the average can be released for statistical purposes. The average is stored in a low variable avg.

$$avg := \mathsf{declassify}((h_1 + \ldots + h_n)/n, \ell)$$

Given that the annotation of the expression does not change its value, this program does not satisfy non-interference since the final value of avg is influenced directly via the values of $h_1, \ldots, h_n$. However, it does satisfy delimited release which only requires avg to be the same after two runs of the program if those runs start from memories in which the average of the salaries is the same.

Consider, however, extending the program as follows.

$$h_2 := h_1;\ \ldots h_n := h_1;\ avg := \mathsf{declassify}((h_1 + \ldots + h_n)/n, \ell)$$

In this case, the program releases not the average of the initial salaries, but the initial value of h_1. This program does not satisfy delimited release. For example, if one run starts with $h_1 = v$ and $h_2 = u$ and the other with $h_1 = u$ and $h_2 = v$, for some values u and v, then the runs will start from memories in which the average of the salaries are the same. Yet the values of avg won't agree after the runs; it will be v for the first run and u for the second.

It is in this way that delimited release controls *what* is released; in this case the average of the initial salaries. ◇

Example 2. Electronic wallet

An electronic wallet indicates whether a transaction of amount k can proceed despite the balance h being sensitive and having a security level higher than ℓ (the level accessible by a potential attacker). The amount of a successful transaction is removed from h and added to a low variable amt.

$$\mathsf{if}\ \mathsf{declassify}(h \geq k, \ell)\ \mathsf{then}\ h := h - k;\ amt := amt + k\ \mathsf{else}\ skip$$

Again this program does not satisfy non-interference (since amt is influenced by h), but does satisfy delimited release (since amt will have the same final value

for two runs of a program in which it has the same initial value and same value for the released expression $h \geq k$ in the initial state). When extended as follows, however, the program does not satisfy delimited release.

$n := 0;$
$\mathsf{while}(n < N)$
    $\mathsf{if\ declassify}(h \geq k, \ell)\ \mathsf{then}\ h := h - k;\ amt := amt + k\ \mathsf{else}\ skip$
    $n := n + 1$

Rather than releasing the expression $h \geq k$ (as specified), this program releases how many transactions of amount k between 0 and an arbitrary value N can occur. It does not satisfy delimited release since the expression $h \geq k$ will be the same for an initial memory with $h = 2 * k$ and another with $h = 4 * k$, but the value released through amt will be different when $N \geq 2$. ◇

2.3 Localized delimited release

Consider the following program where out_1 and out_2 have security level ℓ, and h has a security level higher than ℓ.

$out_1 := h;\ out_2 := \mathsf{declassify}(h, \ell)$

This satisfies delimited release even though it seems it is releasing the value of h early. Definition 2 only requires two runs to agree on out_1 when they agree on the value of h (identified as an escape hatch expression in the second instruction). That is, the definition is concerned with *what* can be released, not *where*.

To weaken non-interference for the where dimension of declassification, Mantel and Sands [8] modify the bisimulation underlying non-interference. Instructions which declassify an expression only need to maintain low equivalence of memories when the expression is equivalent in both memories. All other instructions need to always maintain low equivalence. Hence, the above program would be regarded as insecure since the first instruction does not maintain low equivalence for memories with different values of h. However, the extended versions of Examples 1 and 2 would be regarded as secure (since only the where, and not the what, dimension is considered by this definition). This is also true of other approaches which focus on the where dimension, including those which represent the knowledge of the attacker [2, 4].

Localized delimited release [3] combines the idea of escape hatches from delimited release with a similar modification of the bisimulation underlying non-interference. Let $\langle c, m, E\rangle$ denote a program configuration with an additional element E denoting the set of escape hatches encountered so far in the program.

Definition 3. Localized delimited release on program c

Program c is secure if for all initial states i_1 and i_2 such that $i_1 =_\ell i_2$, there exists a bisimulation R_{i_1,i_2} such that $\langle c, i_1, \varnothing\rangle\ R_{i_1,i_2}\ \langle c, i_2, \varnothing\rangle$ and for all programs c_1

and c_2, and memories m_1 and m_2, the following holds.

$$
\begin{array}{l}
\langle c_1, m_1, E_1\rangle \, R_{i_1,i_2} \, \langle c_2, m_2, E_2\rangle \Rightarrow \\
\quad 1.\ (i_1 \, I(E_1) \, i_2 \Leftrightarrow i_1 \, I(E_2) \, i_2) \wedge \\
\quad 2.\ (i_1 \, I(E_1) \, i_2 \Rightarrow \\
\qquad \text{(i)}\ m_1 =_\ell m_2 \wedge \\
\qquad \text{(ii)}\, \forall\, c_1', m_1', E_1' \cdot \\
\qquad\qquad (\langle c_1, m_1, E_1\rangle \longrightarrow \langle c_1', m_1', E_1'\rangle \Rightarrow \\
\qquad\qquad\quad \exists\, c_2', m_2', E_2' \cdot \\
\qquad\qquad\qquad \langle c_2, m_2, E_2\rangle \longrightarrow \langle c_2', m_2', E_2'\rangle \wedge \\
\qquad\qquad\qquad \langle c_1', m_1', E_1'\rangle \, R_{i_1,i_2} \, \langle c_2', m_2', E_2'\rangle))
\end{array}
$$

The configurations related by R_{i_1,i_2} correspond to those reached by two runs of the program c which start from any low-equivalent states. Note that the escape hatches encountered during two runs of a program may differ due to branching on high data. (Any branch taken based on low data will be the same for two runs whose states are low-equivalent.) The definition of R_{i_1,i_2} requires that the escape hatches of the first run are indistinguishable in the initial states of the runs, i_1 and i_2, precisely when the escape hatches of the second run are indistinguishable in i_1 and i_2 (requirement 1). Furthermore, when they are indistinguishable in the initial states, the current program states are low-equivalent (requirement 2(i)), and any step of the first run can be matched by a step of the second run to reach configurations again related by R_{i_1,i_2} (requirement 2(ii)).

Localized delimited release agrees with delimited release for the programs in Examples 1 and 2. However, the program $out_1 := h;\ out_2 := \mathsf{declassify}(h, \ell)$ is insecure under localized delimited release.

3 Declassification predicates

Sabelfeld and Sands [13] define a security type system for ensuring delimited release. As well as tracking information flow, the type system keeps track of which variables have been updated and which have been declassified. The key constraint placed on secure programs is that escape-hatch variables must not be updated before they are declassified. The same type system is shown to also enforce localized delimited release by Askarov and Sabelfeld [3].

It is easy to see that this type system will find the extended programs in Examples 1 and 2 insecure (as required). Both extended programs change high variables before declassifying them. Relying on such a simple check, however, limits how declassification can be used. For example, consider the following program in which employees are awarded an annual bonus (which is between 0 and 10% of their salary) before the average of their salaries are released. $b_1, \ldots, b_n$ are the bonuses provided as inputs to the program.

$$h_1 := h_1 + b_1;\ \ldots h_n := h_n + b_n;\ avg := \mathsf{declassify}((h_1 + \ldots + h_n)/n, \ell)$$

This program illustrates a reasonable release of information, but would be regarded as insecure by the type system.

Similarly, consider an electronic wallet which can indicate whether two transactions of amount k can proceed.

$n := 0;$
$\mathsf{while}(n < 2)$
    $\mathsf{if\ declassify}(h \geq k, \ell)\ \mathsf{then}\ h := h - k;\ amt := amt + k\ \mathsf{else}\ skip$
    $n := n + 1$

Again this is a reasonable release of information, but cannot be shown to be secure using the type system.

In both examples above, we want to declassify the identified expression in the current state of the program. The type system, however, requires the current values of escape-hatch variables to be equal to their initial state values. As the examples of Section 2 show, this is key to capturing *what* is declassified. Modifying Definitions 2 and 3 to refer to the current states in place of the initial states of the runs would result in them capturing the where dimension of declassification, and no longer the what dimension. As a consequence, the extensions of both Example 1 and 2 would be regarded as secure.

To allow such a modification and not lose the what dimension, we propose introducing constraints at the points of declassification. These constraints, which we call *declassification predicates*, provide a specification of precisely what can be released with respect to the initial state of the program. For the salaries example above, we want to release a value between the average of the initial values (corresponding to the case where all employees get a minimal bonus of 0) and the average of the values obtained by adding 10% to each of the initial values (corresponding to the case where all employees get the maximal bonus). The declassification predicate is written as a function $P_i(e)$ where e is the expression being declassified and i is the initial state of the run in which the predicate is being evaluated. Let $[\![v]\!]^i$ denote the value of variable v in state i. The declassification predicate for the salaries program is

$$P_i(e) \mathrel{\widehat{=}} ([\![h_1]\!]^i + \ldots + [\![h_n]\!]^i)/n \leq e \leq ([\![h_1]\!]^i + [\![h_1]\!]^i/10 + \ldots + [\![h_n]\!]^i + [\![h_n]\!]^i/10)/n$$

Provided the inputs $b_1, \ldots, b_n$ are between 0 and 10% of the salaries $h_1, \ldots, h_n$ respectively, this predicate will hold at the point of declassification. Hence, the above program will be regarded as secure. However, the predicate will not hold at the point of declassification of the extended program of Example 1, making that program insecure (as desired).

For the electronic wallet example, we want to release the information about whether $[\![h]\!]^i \geq k$ and then, after a successful transaction, whether $[\![h]\!]^i - k \geq k$. A suitable declassification predicate is

$$P_i(e) \mathrel{\widehat{=}} ([\![h]\!]^i \geq k \Leftrightarrow e) \vee ([\![h]\!]^i - k \geq k \Leftrightarrow e)$$

In general, there may be more than one declassification predicate associated with a program, and each declassified expression must be annotated with the

predicate which needs to hold. When a declassification predicate does not hold, the expression is interpreted with its usual, i.e., non-declassified, security classification.

To support declassification predicates, we define a notion of security which we call *qualified release*. Qualified release differs from localized delimited release since it assumes that an escape-hatch expression is only released at the instruction where it is annotated as being declassified (similarly, to the approach of Mantel and Sands [8]). In contrast, with localized delimited release once an escape-hatch expression is released, it remains released for the rest of the program. For example, the program $out_1 := \mathsf{declassify}(h, \ell);\ out_2 := h$ is secure under localized delimited release when h is a high variable, and out_1 and out_2 are low.

Hence, we modify requirement 1 and the premise of requirement 2 of Definition 3 so that the escape-hatch expressions are compared in the current states of the runs, m_1 and m_2, rather than the initial states, i_1 and i_2. That is, the conditions on low-equivalence of future states (requirements 2(i) and 2(ii)) need only hold when all escape hatches have the same value in the state at which they are released. Furthermore, an annotated expression is only considered to be an escape hatch when the associated declassification predicate holds.

Qualified release is defined below where we let $\varepsilon_i(c, m)$ denote the set of expressions e where

- the first instruction in c has an expression $\mathsf{declassify}(e, d)$ such that d is not higher than ℓ, the security level accessible by an attacker (the requirement from Section 2), and
- m satisfies the associated declassification predicate $P_i(e)$ where i is the initial state of the run that led to the configuration $\langle c, m\rangle$.

Definition 4. Qualified release on program c

Program c is secure if for all initial states i_1 and i_2 such that $i_1 =_\ell i_2$, there exists a bisimulation R_{i_1,i_2} such that $\langle c, i_1\rangle\, R_{i_1,i_2}\, \langle c, i_2\rangle$ and for all programs c_1 and c_2, and memories m_1 and m_2, the following holds.

$$
\begin{array}{l}
\langle c_1, m_1\rangle\, R_{i_1,i_2}\, \langle c_2, m_2\rangle \Rightarrow \\
\quad 1.\ (m_1\, I(\varepsilon_{i_1}(c_1, m_1))\, m_2 \Leftrightarrow m_1\, I(\varepsilon_{i_2}(c_2, m_2))\, m_2) \wedge \\
\quad 2.\ m_1 =_\ell m_2 \wedge \\
\quad 3.\ (m_1\, I(\varepsilon_{i_1}(c_1, m_1))\, m_2 \Rightarrow \\
\qquad\quad \forall\, c_1', m_1' \cdot \\
\qquad\qquad (\langle c_1, m_1\rangle \longrightarrow \langle c_1', m_1'\rangle \Rightarrow \\
\qquad\qquad\quad \exists\, c_2', m_2' \cdot \langle c_2, m_2\rangle \longrightarrow \langle c_2', m_2'\rangle \wedge \langle c_1', m_1'\rangle\, R_{i_1,i_2}\, \langle c_2', m_2'\rangle))
\end{array}
$$

For requirement 1 and the premise of requirement 3, the escape-hatch expressions are compared in the current states of the runs, m_1 and m_2, rather than the initial states, i_1 and i_2, as in Definition 3. This allows the variables in the escape-hatch expressions to be modified before declassification.

Like localized delimited release, when there is no declassification in a program qualified release is equivalent to non-interference. It requires that the states

reached by a pair of runs, after any given number of steps from low-equivalent states, are low-equivalent.

In programs with declassification, the where dimension of declassification is captured by only enforcing low-equivalence when the escape-hatch expressions in the *current* states of the runs are indistinguishable. Note that when the two runs have identical values for their high variables initially, all escape-hatch expressions will be indistinguishable allowing the bisimulation to traverse the entire program.

The what dimension of declassification is enforced by the declassification predicates which must hold for each associated expression to be in $\varepsilon_{i_1}(c_1, m_1)$, and hence be declassified.

4 Enforcing qualified release

Due to the need to be able to evaluate arbitrary predicates $P_i(e)$ for qualified release, it cannot be enforced by a simple security type system (such as that designed for delimited release in [13]). A program logic which keeps track of the current state is required. In this section, we show how it can be enforced in the program logic developed by Winter et al. [16]. For presentation purposes, we focus on a subset of this logic supporting static security classifications of variables in single-threaded programs. Our approach, however, is equally applicable to the full logic for dynamic security classifications that change as the program executes (often referred to as *value-dependent* security classifications [11]) and multi-threaded programs.

In this subset of the logic, the function $\mathcal{L}$ maps each program variable to its (static) security classification. In a secure program, each program variable x is only allowed to hold data whose security level is lower than or equal to $\mathcal{L}(x)$. The security level of the data held in variable x, when it can be determined from the program, is captured in an auxiliary variable Γ_x. The security level of an expression e in a program which is so far judged to be secure is calculated from the security level of each variable x in e as follows.

$$\Gamma_E(e) \mathrel{\widehat{=}} \bigsqcup_{x \in vars(e)} (\Gamma_x \sqcap \mathcal{L}(x))$$

Note that $\sqcup$ denotes the join operator (or least upper bound) of the security lattice and $\sqcap$ denotes the meet operator (or greatest lower bound). Hence, if the security level of x can be determined, since it will be less than or equal to $\mathcal{L}(x)$, the expression $\Gamma_x \sqcap \mathcal{L}(x)$ equates to Γ_x. If it cannot be determined, the expression equates to $\mathcal{L}(x)$, accounting for all possible values of Γ_x. The security level of expression e is the highest of the values for each of its variables.

The logic is then expressed in terms of a weakest precondition predicate transformer *wpif* over a simple language comprising skips, assignments, conditionals (if-then-else) and loops (while).[2] Starting from the last line of code and the postcondition *true*, the predicate transformer introduces proof obligations

[2] For capturing concurrent algorithms, the simple language also supports atomic compare-and-swap (CAS) instructions which are not considered in this paper.

associated with information flow which are then transformed successively over the rest of the code to a predicate which must hold in the code's initial state. That is, a program c starting in an initial state satisfying S_0 is secure when $S_0 \Rightarrow \mathit{wpif}(c, \mathit{true})$.

A skip instruction has no effect.

$$\mathit{wpif}(\mathsf{skip}, Q) \mathrel{\widehat{=}} Q$$

For an assignment instruction, the predicate transformer is defined as follows where $\mathit{def}(e)$ is true when e is defined, i.e., won't throw an exception.

$$\mathit{wpif}(x := e, Q) \mathrel{\widehat{=}} \Gamma_E(e) \sqsubseteq \mathcal{L}(x) \wedge \mathit{def}(e) \wedge Q[x \leftarrow e, \Gamma_x \leftarrow \Gamma_E(e)]$$

The first conjunct is a check that the expression e's security level is lower than or equal to that which x is allowed to hold. The second conjunct ensures e is defined. This is needed since throwing of exceptions can reveal information about sensitive data. For example, a divide-by-zero exception can reveal that a sensitive variable holds value 0. The final conjunct updates the values of x and Γ_x in the postcondition Q as in the standard weakest precondition transformer for a pair of assignments.

For a conditional instruction, we require that the branching condition is defined and has a security level lower than or equal to ℓ, the highest security level of data which a potential attacker can access. This is not common to all information flow logics, but is used in Winter et al. [16] to prohibit information being leaked when an attacker can observe the timing of the chosen branch (and hence deduce the value of the branching condition [1, 10]).

$$\begin{aligned}&\mathit{wpif}(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, Q) \mathrel{\widehat{=}}\\ &\qquad \Gamma_E(b) \sqsubseteq \ell \wedge \mathit{def}(b) \wedge (b \Rightarrow \mathit{wpif}(c_1, Q)) \wedge (\neg\, b \Rightarrow \mathit{wpif}(c_2, Q))\end{aligned}$$

The final two conjuncts are the standard weakest precondition transformer for conditionals (with *wpif* replacing *wp*).

For while loops, the security level of the guard similarly needs to be defined and low for each iteration of the loop, i.e., whenever the loop's invariant is true. For a program with variables $x_1, \ldots, x_n$ we have

$$\begin{aligned}&\mathit{wpif}(\mathsf{while}(b)\ c, Q) \mathrel{\widehat{=}}\\ &\qquad (\forall\, x_1, \ldots, x_n \cdot \mathit{Inv} \Rightarrow \mathit{def}(b) \wedge \Gamma_E(b) \sqsubseteq \ell) \wedge\\ &\qquad \mathit{Inv} \wedge (\forall\, x_1, \ldots, x_n \cdot (\mathit{Inv} \wedge b \Rightarrow \mathit{wpif}(c, \mathit{Inv})) \wedge (\mathit{Inv} \wedge \neg\, b \Rightarrow Q))\end{aligned}$$

where the final two conjuncts are the standard weakest precondition transformer for partial correctness of loops (with *wpif* replacing *wp*). Note that since guards do not contain sensitive data (as ensured by the first conjunct), whether or not a loop terminates does not leak information.

To add declassification predicates to the logic, we introduce program annotations of the form $\mathsf{declassify}_P(...)$ where P is the associated declassification predicate with initial values of variables replaced by auxiliary variables, $v_1^0, \ldots, v_n^0$,

i.e., additional variables which do not affect the program execution. These auxiliary variables are initialised to the corresponding program variables to determine if a program c is secure. That is, we check $S_0 \Rightarrow \mathit{wpif}(c', \mathit{true})$ where c' is $v_1^0 := v_1;\ \ldots;\ v_n^0 := v_n;\ c$.

We assume that when an expression is declassified, the declassification is applied to the entire expression. That is, the syntax added to include annotations is $x := \mathsf{declassify}_P(e, \ell)$ for assignments, and $\mathsf{if}\ \mathsf{declassify}_P(b, \ell)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ and $\mathsf{while}(\mathsf{declassify}_P(b, \ell))\ c$ for conditionals and loops.

The logic is then extended with the following rules, where d is the level to which information is being declassified and ℓ is the level of access of an attacker.

$$\begin{aligned}
&\mathit{wpif}(x := \mathsf{declassify}_P(e, d), Q) \mathrel{\widehat{=}} \\
&\qquad d \sqsubseteq \mathcal{L}(x) \wedge P(e) \wedge \mathit{def}(e) \wedge Q[x \leftarrow e, \Gamma_x \leftarrow d)] \\
&\mathit{wpif}(\mathsf{if}\ \mathsf{declassify}_P(b, d)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, Q) \mathrel{\widehat{=}} \\
&\qquad d \sqsubseteq \ell \wedge P(b) \wedge \mathit{def}(b) \wedge (b \Rightarrow \mathit{wpif}(c_1, Q)) \wedge (\neg\, b \Rightarrow \mathit{wpif}(c_2, Q)) \\
&\mathit{wpif}(\mathsf{while}(\mathsf{declassify}_P(b, d))\ c, Q) \mathrel{\widehat{=}} \\
&\qquad (\forall\, x_1, \ldots, x_n \cdot \mathit{Inv} \Rightarrow \mathit{def}(b) \wedge P(b)) \wedge d \sqsubseteq \ell \wedge \\
&\qquad \mathit{Inv} \wedge (\forall\, x_1, \ldots, x_n \cdot (\mathit{Inv} \wedge b \Rightarrow \mathit{wpif}(c, \mathit{Inv})) \wedge (\mathit{Inv} \wedge \neg\, b \Rightarrow Q))
\end{aligned}$$

Each of the rules checks that the declassification predicate holds at the point of declassification. Hence, they will judge a program which does not satisfy a declassification predicate as insecure. This is stricter than necessary since an expression can be used with its usual classification when the predicate does not hold. However, the rules provide a simple extension to the logic which suffices in most situations. A proof of soundness of the extended logic with respect to Definition 4 is provided in Appendix A. To provide an intuitive understanding of the new rules, we revisit the suggested extensions to Examples 1 and 2 of Section 3.

Example 3. Average salary with bonuses

Let $b_1, \ldots, b_n$ be between 0 and 10% of the salaries $h_1, \ldots, h_n$ respectively. Let $P(e) \mathrel{\widehat{=}} (h_1^0 + \ldots + h_n^0)/n \leq e \leq (h_1^0 + h_1^0/10 + \ldots + h_n^0 + h_n^0/10)/n$ and c be the following program (where $n > 0$ and avg has security level ℓ).

$$h_1 := h_1 + b_1;\ \ldots h_n := h_n + b_n;\ \mathit{avg} := \mathsf{declassify}_P((h_1 + \ldots + h_n)/n, \ell)$$

The program is secure since the proof obligation $P((h_1 + \ldots + h_n)/n)$ introduced by *wpif* over the final instruction, is transformed to

$$\begin{aligned}
(h_1^0 + \ldots + h_n^0)/n &\leq (h_1 + b_1 + \ldots + h_n + b_n)/n) \\
&\leq (h_1^0 + h_1^0/10 + \ldots + h_n^0 + h_n^0/10)/n
\end{aligned}$$

by the preceding instructions. It is further transformed to

$$\begin{aligned}
(h_1 + \ldots + h_n)/n &\leq (h_1 + b_1 + \ldots + h_n + b_n)/n) \\
&\leq (h_1 + h_1/10 + \ldots + h_n + h_n/10)/n
\end{aligned}$$

by the initialisation of the auxiliary variables $h_1^0 := h_1;\ \ldots;\ h_n^0 := h_n$, which then evaluates to true given the constraints on $b_1, \ldots, b_n$. $\diamond$

Example 4. Electronic wallet, two withdrawals

Let $P(e) \mathrel{\widehat{=}} (h^0 \geq k \Leftrightarrow e) \vee (h^0 - k \geq k \Leftrightarrow e)$ and c be the following program (where $k \geq 0$ and amt has security level ℓ).

$n := 0;$
while$(n < 2)$
  if $\mathsf{declassify}_{\mathsf{P}}(h \geq k, \ell)$ then $h := h - k;\ amt := amt + k$ else *skip*
  $n := n + 1$

Since the final instruction of this program is a loop, we need a loop invariant to apply the logic. It is easy to see from the loop body, that when $n = 0$, $h = h^0$ and when $n = 1$, either $h = h^0$ when the if condition evaluated to false on the first loop iteration, or $h = h^0 - k$ when the if condition evaluated to true. Hence we choose the following as the invariant.

$(n = 0 \Rightarrow h = h^0) \wedge$
$(n = 1 \Rightarrow (h^0 \geq k \Rightarrow h = h^0 - k) \wedge (h^0 < k \Rightarrow h = h^0)$

The while loop introduces this invariant and the proof obligation $P(h \geq k)$ which is $(h^0 \geq k \Leftrightarrow h \geq k) \vee (h^0 - k \geq k \Leftrightarrow h \geq k)$. The former is transformed to $h = h^0$ by the first instruction (since n is replaced by 0) while the latter (which does not refer to n) is not changed. They are then both transformed to true by the initialisation of the auxiliary variable $h^0 := h$. ◇

5 Dafny encoding

Being based on weakest precondition calculations, the *wpif*-based information flow logic is readily encoded in a program verification tool such as Dafny [6]. The lattice of security levels can be encoded as an enumerated type along with functions defining when two levels are ordered, and for returning the meet and join of any two levels. For example, a standard diamond lattice where level A is higher than levels B and C which are in turn higher than level D (but B and C are not ordered) can be encoded in Dafny as follows.

```
datatype SL = A | B | C | D

function order(l1:SL, l2:SL):bool {
    l1 == A || l1 == l2 || l2 == D
}

function join(l1:SL, l2:SL):SL {
    if order(l1,l2) then l1 else if order(l2,l1) then l2 else A
}

function meet(l1:SL, l2:SL):SL {
    if order(l1,l2) then l2 else if order(l2,l1) then l1 else D
}
```

Note that $order(l_1, l_2)$ is true precisely when $l_2 \sqsubseteq l_1$. To check the encoding is indeed a lattice, the following lemmas, which can be proved automatically by Dafny, are added.

```
lemma partialorder(l1:SL, l2:SL, l3:SL)
    ensures order(l1,l1)
    ensures order(l1,l2) && order(l2,l1) ==> l1 == l2
    ensures order(l1,l2) && order(l2,l3) ==> order(l1,l3)

lemma joinLemma(l1:SL,l2:SL,l3:SL)
    ensures order(join(l1,l2),l1) && order(join(l1,l2),l2)
    ensures order(l3,l1) && order(l3,l2) ==> order(l3,join(l1,l2))

lemma meetLemma(l1:SL,l2:SL,l3:SL)
    ensures order(l1,meet(l1,l2)) && order(l2,meet(l1,l2))
    ensures order(l1,l3) && order(l2,l3) ==> order(meet(l1,l2),l3)
```

Then for each program variable x, *ghost* variables (which are used for specification purposes and are not part of the compiled program) can be added for $\mathcal{L}(x)$, Γ_x and the initial value of x. These allow the security checks (including declassification predicates) added by the *wpif* rules to be encoded as program assertions. For example, consider Example 3 with salaries $h1$, $h2$ and $h3$ and their associated bonuses at security level C, and an attacker able to access data at security level D. This can be encoded within a Dafny class as follows (for readability the full encoding for $h2$ and $h3$ is elided, but follows that for $h1$).

```
var h1:int, h2:int, h3:int, avg: int;   // salaries
ghost var h01:int, h02:int, h03:int, avg0:int;   // initial values
ghost const L_h1 := C; ...; const L_avg := D;
ghost var Gamma_h1: SL, ..., Gamma_avg: SL;

predicate P(e:int)
    reads this   // allows read access to the variables declared above
{
    (h01+h02+h03)/3 <= e <= (h01+h01/10+h02+h02/10+h03+h03/10)/3
}

method average(b1:int, b2:int, b3:int)
     requires 0 <= b1 <= h1/10 && ...
     modifies this   // allows changes to the variables declared above
{
     h01 := h1; h02 := h2; h03 := h3;   // set initial values
     assert order(L_h1,join(meet(Gamma_h1,L_h1),C)) && ...
     h1 := h1 + b1; Gamma_h1 := join(meet(Gamma_h1,L_h1),C); ...;
     assert order(L_avg,D) && P((h1+h2+h3)/3);
     avg := (h1+h2+h3)/3; Gamma_avg := D;   // declassify to level D
}
```

The program of Example 3 is captured by the method *average*. The requires clause of the method ensures the bonuses are within 10% of the associated salaries. The assertions capture the checks required by the *wpif* rules for

the instructions which follow them (note that Dafny automatically checks for definedness of expressions).

For example, consider the final line of code which represents the assignment $avg := \mathsf{declass}_P((h1 + h2 + h3)/3)$. Applying the *wpif* rule for assignments involving declassification, we would get $D \sqsubseteq \mathcal{L}(avg) \wedge P((h1 + h2 + h3)/3) \wedge def((h1 + h2 + h3)/3) \wedge Q[avg \leftarrow (h1 + h2 + h3)/3, \Gamma_{avg} \leftarrow D]$. The final two conjuncts are routinely checked by Dafny (given that we have explicitly included the update to Γ_{avg}). The other conjuncts are included in the assertion and are hence also checked. Dafny can prove each assertion in this example automatically, proving that the program is secure. Similarly, other examples in this paper can be automatically proven to be secure or insecure as required. Importantly, the insertion of the assertions into the code, as well as the declaration and updates of ghost variables (apart from the values of $\mathcal{L}$ which must be provided) can be readily automated to reduce the programmer's burden.

6 Conclusion

In this paper, we have introduced declassification predicates and an associated notion of security called qualified release. Declassification predicates enable program developers to precisely specify *what* sensitive information can be released by a program, and *where* in the program it can be released. We have shown how they can be incorporated into an information flow logic for checking program security, and checked using the Dafny program verifier. Future research includes investigating the application of declassification predicates to a wide range of scenarios where information release is required, and their use in larger, real-world programs.

Acknowledgements Thanks to Kirsten Winter for her feedback on this paper.

References

1. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations. In: Holz, T., Savage, S. (eds.) 25th USENIX Security Symposium, USENIX Security 16. pp. 53–70. USENIX Association (2016)
2. Askarov, A., Sabelfeld, A.: Gradual release: Unifying declassification, encryption and key release policies. In: 2007 IEEE Symposium on Security and Privacy (S&P 2007). pp. 207–221. IEEE Computer Society (2007). https://doi.org/10.1109/SP.2007.22
3. Askarov, A., Sabelfeld, A.: Localized delimited release: combining the what and where dimensions of information release. In: Hicks, M.W. (ed.) Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS 2007. pp. 53–60. ACM (2007). https://doi.org/10.1145/1255329.1255339
4. Chudnov, A., Naumann, D.A.: Assuming you know: Epistemic semantics of relational annotations for expressive flow policies. In: 31st IEEE Computer Security Foundations Symposium, CSF 2018. pp. 189–203. IEEE Computer Society (2018). https://doi.org/10.1109/CSF.2018.00021

5. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, 1982. pp. 11–20. IEEE Computer Society (1982). https://doi.org/10.1109/SP.1982.10014
6. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4“¨20
7. Mantel, H., Reinhard, A.: Controlling the what and where of declassification in language-based security. In: Nicola, R.D. (ed.) Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007. Lecture Notes in Computer Science, vol. 4421, pp. 141–156. Springer (2007). https://doi.org/10.1007/978-3-540-71316-6“¨11
8. Mantel, H., Sands, D.: Controlled declassification based on intransitive noninterference. In: Chin, W. (ed.) Programming Languages and Systems: Second Asian Symposium, APLAS 2004. Lecture Notes in Computer Science, vol. 3302, pp. 129–145. Springer (2004). https://doi.org/10.1007/978-3-540-30477-7“¨9
9. Mantel, H., Sands, D., Sudbrock, H.: Assumptions and guarantees for compositional noninterference. In: Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011. pp. 218–232. IEEE Computer Society (2011). https://doi.org/10.1109/CSF.2011.22
10. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.A.: The program counter security model: Automatic detection and removal of control-flow side channel attacks. In: Won, D., Kim, S. (eds.) Information Security and Cryptology - ICISC 2005, 8th International Conference. Lecture Notes in Computer Science, vol. 3935, pp. 156–168. Springer (2005). https://doi.org/10.1007/11734727“¨14
11. Murray, T.C., Sison, R., Engelhardt, K.: Covern: A logic for compositional verification of information flow control. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018. pp. 16–30. IEEE (2018). https://doi.org/10.1109/EuroSP.2018.00010
12. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003). https://doi.org/10.1109/JSAC.2002.806121
13. Sabelfeld, A., Myers, A.C.: A model for delimited information release. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003. Lecture Notes in Computer Science, vol. 3233, pp. 174–191. Springer (2003). https://doi.org/10.1007/978-3-540-37621-7“¨9
14. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. J. Comput. Secur. **17**(5), 517–548 (2009). https://doi.org/10.3233/JCS-2009-0352
15. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2/3), 167–188 (1996). https://doi.org/10.3233/JCS-1996-42-304
16. Winter, K., Coughlin, N., Smith, G.: Backwards-directed information flow analysis for concurrent programs. In: 34th IEEE Computer Security Foundations Symposium, CSF 2021. pp. 1–16. IEEE (2021). https://doi.org/10.1109/CSF51468.2021.00017
17. Zdancewic, S., Myers, A.C.: Robust declassification. In: 14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001. pp. 15–23. IEEE Computer Society (2001). https://doi.org/10.1109/CSFW.2001.930133

A Soundness

Theorem 1. *Given a program c with variables $v_1, \ldots, v_n$ and set of initial states S_0, if $S_0 \Rightarrow wpif(v_1^0 := v_1;\ \ldots;\ v_n^0 := v_n;\ c, true)$, where $v_1^0, \ldots, v_n^0$ are fresh, then qualified release (Definition 4) holds for all initial states $i_1, i_2 \in S_0$.*

Proof
Let m_1 and m_2 be two program states such that $m_1 =_\ell m_2$. Following Definition 4, we need to prove that there exists a relation R_{m_1,m_2} such that $\langle c, m_1\rangle$ $R_{m_1,m_2}\langle c, m_2\rangle$ for each program c considered secure under *wpif*. The proof is by induction over the instructions of the simple programming language whose operational semantics with respect to an initial state i is given below.

Let stop denote the program with no instructions, and $\varepsilon_i(\mathsf{stop}, m) = \varnothing$.

$\langle \mathsf{skip}, m\rangle \longrightarrow \langle \mathsf{stop}, m\rangle \qquad \varepsilon_i(\mathsf{skip}, m) = \varnothing$

$\langle x := e, m\rangle \longrightarrow \langle \mathsf{stop}, m[x \mapsto e']\rangle$ when $e = \mathsf{declassify}_P(e', d)$
$\langle x := e, m\rangle \longrightarrow \langle \mathsf{stop}, m[x \mapsto e]\rangle$ otherwise
$\varepsilon_i(x := e, m) = \{e'\}$ when $e = \mathsf{declassify}_P(e', d)$ and $d \sqsubseteq \ell$ and m satisfies $P(e')$
$\varepsilon_i(x := e, m) = \varnothing$ otherwise

$\langle c_1;\ c2, m\rangle \longrightarrow \langle c_1';\ c2, m'\rangle$ when $\langle c_1, m\rangle \longrightarrow \langle c_1', m'\rangle$
$\langle c_1;\ c2, m\rangle \longrightarrow \langle c2, m'\rangle$ when $\langle c_1, m\rangle \longrightarrow \langle \mathsf{stop}, m'\rangle$

$\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, m\rangle \longrightarrow \langle c_1, m\rangle$ when b is true
$\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, m\rangle \longrightarrow \langle c_2, m\rangle$ when b is false
$\varepsilon_i(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, m) = \{b'\}$ when $b = \mathsf{declassify}_P(b', d)$ and $d \sqsubseteq \ell$ and m satisfies $P(b')$
$\varepsilon_i(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, m) = \varnothing$ otherwise

$\langle \mathsf{while}(b)\ c, m\rangle \longrightarrow \langle c;\ \mathsf{while}(b)\ c, m\rangle$ when b is true
$\langle \mathsf{while}(b)\ c, m\rangle \longrightarrow \langle \mathsf{stop}, m\rangle$ when b is false
$\varepsilon_i(\mathsf{while}(b)\ c, m) = \{b'\}$ when $b = \mathsf{declassify}_P(b', d)$ and $d \sqsubseteq \ell$ and m satisfies $P(b')$
$\varepsilon_i(\mathsf{while}(b)\ c, m) = \varnothing$ otherwise

skip According to the operational semantics, the skip instruction changes neither the state m nor introduces escape-hatch expressions, and results in the program stop. Therefore, choosing $R_{m_1,m_2} = \{(\langle \mathsf{skip}, m_1\rangle, \langle \mathsf{skip}, m_2\rangle), (\langle \mathsf{stop}, m_1\rangle, \langle \mathsf{stop}, m_2\rangle)\}$ will satisfy Definition 4. For both configuration pairs, requirement 1 of Definition 4 holds trivially and requirement 2 holds due to $m_1 =_\ell m_2$ holding in the starting state. Requirement 3 holds for the second pair due to there being no further program steps and, since it holds for the second pair, also holds for the first pair.

$x := e$ An assignment updates the state m and results in the program stop. If the assignment has a declassification annotation then (according to the

wpif rule) the program will only be secure when the associated predicate is true in any state that may hold immediately before the assignment. Assume this is the case, and hence the declassification predicate is true in m_1 and m_2. Given this, any escape hatches introduced only depend on e and the level of declassification d (see the operational semantics). Hence, requirement 1 of Definition 4 trivially holds. Consider the relation $R_{m_1,m_2} = \{(\langle x := e, m_1\rangle, \langle x := e, m_2\rangle), (\langle \mathsf{stop}, m_1'\rangle, \langle \mathsf{stop}, m_2'\rangle)\}$, where m_1' and m_2' are derived from m_1 and m_2, respectively, by updating the value of x.

For the first configuration pair, if the released expressions are distinguishable on m_1 and m_2 then there is nothing further to prove. If they are indistinguishable then, since the assignments will replace x by an expression which is in the released expressions and at a level $d \sqsubseteq \ell$ (as required by the *wpif* rule), requirement 2 (which holds for the first pair) is preserved. Requirement 3 also holds for the second pair (as argued for skip) and hence holds for the first pair.

If the assignment does not have a declassification annotation then no escape hatches will be introduced and requirement 1 trivially holds. The value of x will be replaced by e for both initial states. The *wpif* rule for assignment only holds when $\Gamma_E(e) \sqsubseteq \mathcal{L}(x)$. Hence, if x is low so is e, and low equivalence of states (requirement 2) is preserved. Requirement 3 is also satisfied by the first pair due to the second pair satisfying all requirements.

$c_1;\ c_2$ By the induction hypothesis, there exists a relation $R^1_{m_1,m_2}$ such that $\langle c_1, m_1\rangle R^1_{m_1,m_2}\langle c_1, m_2\rangle$ and $R^1_{m_1,m_2}$ satisfies Definition 4. Let $\langle c_1, m_1\rangle \rightarrow^n \langle c_1', m_1'\rangle$ and $\langle c_1, m_2\rangle \rightarrow^n \langle c_1', m_2'\rangle$, for some n. Note that both ending configurations have the same program, c_1', since the logic does not allow branching on high data.

If $\neg\ (m_1'\ I(\varepsilon_i(c_1', m_1'))\ m_2')$ then there are no further requirements to prove and the following relation satisfies Definition 4 for $c_1;\ c_2$.

$$\begin{aligned} R_{m_1,m_2} = \{(\langle c_1^1;\ c_2, m_1'\rangle, \langle c_1^2;\ c_2, m_2'\rangle) \mid \\ \exists\, n \cdot \langle c_1, m_1\rangle \longrightarrow^n \langle c_1^1, m_1'\rangle \wedge \langle c_1, m_2\rangle \longrightarrow^n \langle c_1^2, m_2'\rangle \wedge \\ \langle c_1^1, m_1'\rangle R^1_{m_1,m_2}\langle c_1^2, m_2'\rangle\} \end{aligned}$$

If $m_1'\ I(\varepsilon_i(c_1', m_1'))\ m_2'$ and $\langle c_1', m_1'\rangle \rightarrow \langle \mathsf{stop}, m_1''\rangle$ and $\langle c_1', m_2'\rangle \rightarrow \langle \mathsf{stop}, m_2''\rangle$, i.e., c_1' terminates after its next instruction, then $m_1'' =_\ell m_2''$ by Definition 4. By the induction hypothesis, there exists a relation $R^2_{m_1'',m_2''}$ satisfying Definition 4 where $\langle c_2, m_1''\rangle R^2_{m_1'',m_2''}\langle c_2, m_2''\rangle$. Consider then the following relation.

$$\begin{aligned} R_{m_1,m_2} = \{(\langle c_1^1;\ c_2, m_1'\rangle, \langle c_1^2;\ c_2, m_2'\rangle) \mid \\ \exists\, n \cdot \langle c_1, m_1\rangle \longrightarrow^n \langle c_1^1, m_1'\rangle \wedge \langle c_1, m_2\rangle \longrightarrow^n \langle c_1^2, m_2'\rangle \wedge \\ \langle c_1^1, m_1'\rangle R^1_{m_1,m_2}\langle c_1^2, m_2'\rangle\} \\ \cup \{(\langle c_2^1, m_1'''\rangle, \langle c_2^2, m_2'''\rangle) \mid \\ \exists\, n \cdot \langle c_2, m_1''\rangle \longrightarrow^n \langle c_2^1, m_1'''\rangle \wedge \langle c_2, m_2''\rangle \longrightarrow^n \langle c_2^2, m_2'''\rangle \wedge \\ \langle c_2^1, m_1'''\rangle R^2_{m_1'',m_2''}\langle c_2^2, m_2'''\rangle\} \end{aligned}$$

This clearly relates $\langle c_1;\ c_2, m_1\rangle$ and $\langle c_1;\ c_2, m_2\rangle$. We now show that it also satisfies Definition 4 by considering two cases: a sequence of steps in program c_1, and a sequence of steps in program c_2.

First case: $\langle c_1, m_1\rangle \longrightarrow^n \langle c_1^1, m_1'\rangle$ and $\langle c_1, m_2\rangle \longrightarrow^n \langle c_1^2, m_2'\rangle$ such that $\langle c_1^1, m_1'\rangle R^1_{m_1,m_2}\langle c_1^2, m_2'\rangle$. The latter implies requirements 1 and 2 for R_{m_1,m_2}. If c_1^1 has not terminated then requirement 3 follows from requirement 3 of $R^1_{m_1,m_2}$. If c_1^1 has terminated, i.e., $m_1' = m_1''$ and $m_2' = m_2''$ (since the absence of branching on high data means the runs will terminate together), then from the operational semantics for sequential composition we know that $\langle c_1;\ c_2, m_1\rangle \longrightarrow^n \langle c_2, m_1''\rangle$ and $\langle c_1;\ c_2, m_2\rangle \longrightarrow^n \langle c_2, m_2''\rangle$. And since $\langle c_2, m_1''\rangle$ and $\langle c_2, m_2''\rangle$ are related by $R^2_{m_1'',m_2''}$, they are also related by R_{m_1,m_2} and we have requirement 3.

Second case: $\langle c_2, m_1''\rangle \longrightarrow^n \langle c_2^1, m_1'''\rangle$ and $\langle c_2, m_2''\rangle \longrightarrow^n \langle c_2^2, m_2'''\rangle$ such that $\langle c_2^1, m_1'''\rangle R^2_{m_1,m_2}\langle c_2^2, m_2'''\rangle$. As before, the latter implies requirements 1 and 2 for R_{m_1,m_2}. In this case, requirement 3 of R_{m_1,m_2} also follows from requirement 3 of $R^2_{m_1'',m_2''}$.

$\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ By the induction hypothesis, there exists relations $R^1_{m_1,m_2}$ and $R^2_{m_1,m_2}$ satisfying Definition 4 such that $\langle c_1, m_1\rangle R^1_{m_1,m_2}\langle c_1, m_2\rangle$, and $\langle c_2, m_1\rangle R^2_{m_1,m_2}\langle c_2, m_2\rangle$. Consider the first configuration pair of the following relation.

$$R_{m_1,m_2} = \{(\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, m_1\rangle, \langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, m_2\rangle)\} \cup R^1_{m_1,m_2} \cup R^2_{m_1,m_2}$$

If the guard has a declassification annotation then, following the proof for assignment, requirements 1 and 2 trivially hold, and requirement 3 follows from the requirements holding for subsequent configurations. These hold due to the fact that the *wpif* rule for conditionals requires $\Gamma_E(b) \sqsubseteq \ell$ or, when b is declassified to security level d, $d \sqsubseteq \ell$. Hence, when any released expressions are indistinguishable on m_1 and m_2, the choice of branch c_1 or c_2 will be the same for the low-equivalent states m_1 and m_2 (according to the operational semantics). Therefore, all resulting configurations will be related by either $R^1_{m_1,m_2}$ or $R^2_{m1,m2}$.

$\mathsf{while}(b)\ c$ The *wpif* rule for loops requires a loop invariant which holds in both m_1 and m_2 to be provided. Let M be the set of all memories satisfying the loop invariant and consider the following relation. By the induction hypothesis, for each $m_1', m_2' \in M$ such that $m_1' =_\ell m_2'$ there exists a relation $R^1_{m_1,m_2}$ such that $\langle c, m_1'\rangle R^1_{m_1',m_2'}\langle c, m_2'\rangle$ and $R^1_{m_1',m_2'}$ satisfies Definition 4.

$$R_{m_1,m_2} = \bigcup_{m_1',m_2'\in M} \{(\langle \mathsf{while}(b)\ c, m_1'\rangle, \langle \mathsf{while}(b)\ c, m_2'\rangle), \langle \mathsf{stop}, m_1'\rangle, \langle \mathsf{stop}, m_2'\rangle\} \cup R^1_{m_1',m_2'}$$

Given $m_1, m_2 \in M$, the proof follows the proof for conditionals with the following changes based on the operational semantics: (i) when b is false, the subsequent program is stop and the state is unchanged, and (ii) when b is true, the subsequent program is $c;\ \mathsf{while}(b)\ c$. Note that in the latter case, since the loop invariant will be true after c terminates, requirement 3 will hold due to the distributed union over all states satisfying the invariant in the definition of R_{m_1,m_2}. □