Expressive Declassification Policies and Modular Static Enforcement

Anindya Banerjee David A. Naumann Stan Rosenberg

Stevens Institute of Technology, CS Report 2007-04

August 23, 2007; revised September 10, 2007

Abstract: This paper provides a way to specify expressive declassification policies, in particular, *when*, *what*, and *where* policies that include conditions under which downgrading is allowed. Secondly, an end-to-end semantic property is introduced, based on a model that allows observations of intermediate low states as well as termination. An attacker's knowledge only increases at explicit declassification steps, and within limits set by policy. Thirdly, practical means of enforcement is provided, by combining type-checking with program verification techniques applied to the small subprograms that carry out declassifications. Enforcement is proved sound.

This material is based upon work supported by the National Science Foundation under grants CCF-0429894, CNS-0627338, and CNS-0627748.

Expressive Declassification Policies and Modular Static Enforcement

Anindya Banerjee
Kansas State University, Manhattan, KS, USA
ab@cis.ksu.edu

David A. Naumann and Stan Rosenberg
Stevens Institute of Tech., Hoboken, NJ, USA
[naumann|srosenbe]@cs.stevens.edu

September 10, 2007

Abstract

This paper provides a way to specify expressive declassification policies, in particular, *when*, *what*, and *where* policies that include conditions under which downgrading is allowed. Secondly, an end-to-end semantic property is introduced, based on a model that allows observations of intermediate low states as well as termination. An attacker's knowledge only increases at explicit declassification steps, and within limits set by policy. Thirdly, practical means of enforcement is provided, by combining type-checking with program verification techniques applied to the small subprograms that carry out declassifications. Enforcement is proved sound.

1 Introduction

Protection of information confidentiality and integrity in computer systems has long been approached in three ways. *Cryptography* provides mechanisms that can be used to hide information in data that is openly accessible and to authenticate information in data that is susceptible to tampering. But it is usually impractical to process data in encrypted form. *Access controls* regulate actions by which data is manipulated. Access control mechanisms can often be implemented efficiently and have evident connections with security goals ranging from low level process separation to high level application-specifics. But confidentiality and integrity requirements often encompass indirect manipulation of information in addition to direct access. *Information flow controls* address the manipulation of information once data has been accessed and resides in memory in plaintext form. Research on information flow attempts to *specify* a full range of the confidentiality and integrity goals. It also seeks ways to *check* system designs and implementations for conformance with flow policies, to complement and complete the assurance provided by access control and cryptography.

This paper advances the state of the art in information flow control by specifying a confidentiality property that gives a strong guarantee akin to noninterference while allowing constrained downgrading of secrets under conditions that are explicit in policy specifications. It advances the state of the art in enforcement by combining type checking with localized formal verification in a practical way that provably enforces the security property.

Consider the canonical case where data channels are labelled with one of the levels *high* (secret) or *low*. Information flow control rests on the *mandatory access assumption*: a principal has direct access to a high channel only if the principal is authorized for high channels. (All can read the code being executed, but none can alter it, in this paper.) For confidentiality, noninterference says low observations reveal nothing about high inputs. But notions of observation range from input-output behavior at the level of abstraction of source code to covert channels like battery power and real time. Absence of flows via covert channels is difficult to achieve, much less to verify, and in many scenarios weaker attack models are suitable [21]. Data and control dependencies can be tracked dynamically, by label passing, but there is cost in performance, label creep, and risk of runtime security exceptions.

Static verification of information flow properties is attractive, especially for high assurance of system infrastructure and for integrating components (e.g., web services) from disparate sources. Though long studied, static verification is used only rarely, in part due to high potential cost during software development. Also, security goals must be precisely formalized —and noninterference is too strong to admit certain intended flows, e.g., in password checking, data aggregation, encryption, release of secrets upon successful protocol completion or financial transaction, etc. As discussed in a recent survey [24], it has proved quite difficult to find adequate refinements of noninterference, even to cater for very limited forms of declassification.

The *first contribution* of this paper is a way to specify a wide range of declassification policies, by novel use of existing forms of security typing and program specification, inspired by Chong and Myers [12] who proposed confidentiality policies that include conditions under which declassification is allowed. The *second contribution* is an end-to-end semantic property extended from one recently introduced by Askarov and Sabelfeld [2]. Ours encompasses conditioned policies in the spirit of [12] but with stronger security guarantees.[1] By observing intermediate low states and termination, an attacker's knowledge only increases at explicit declassification steps, which reveal limited information and happen only under specified conditions. The *third contribution* is a practical means of enforcement: by simple type-checking combined with relational program verification techniques [9, 1] applied only to the small subprograms that carry out declassifications. We prove soundness of the enforcement regime.

It is not our intention to propose a concrete policy language. The ideas are formalized here using only a simple programming language, leaving aside issues such as combining confidentiality with integrity, separating policy from code (but see Sect. 4), and tradeoffs

[1] Also stronger than [2] where the attacker learns nothing from divergent computations —which is dubious, given that the attacker can see intermediate steps.

between what is encoded in the lattice of security levels versus what is encoded in state-dependent policy. The exposition and technical development are designed to support a rigorous soundness proof and to highlight important benefits of the approach.

- Policies are expressed using two commonplace means: ordinary labeling of variables with security levels, together with ordinary program assertions, slightly extended with *agreement predicates* $\mathbf{A}(e)$ used to say that some function e of the secrets may be released. This should facilitate integration into existing system development processes and tool chains, as well as integration with existing access mechanisms and cryptographic techniques.

- Policies can express a range of reasons for declassification to be allowed, encompassing what may be released, where in the system, when and under whose authority. The connection with application requirements can be clear because policies refer directly to program data structures or auxiliary state that tracks event history. State dependence caters for some forms of dynamic policy updates, while admitting a cogent semantics to support high assurance.

- Our security property constrains the flow of information even following one or more release actions (vs. [12]). We believe it can capture many important application requirements and also specify the strong properties of security kernels [16] that are the basis for achieving higher level goals. It reduces to termination-sensitive noninterference in the absence of declassification and accords with other prudent principles [24].

- In many systems, sensitive data is manipulated in pointer-based shared data structures rather than named program variables; this poses a challenge for security labeling and for static analysis, due to aliasing. Our use of program logic fits with the solution of Amtoft et al [1], which provides effective, modular verification for typical cases.

A virtue of the enforcement regime is that it integrates, in a straightforward way, three existing technologies: security typing (as provided by [30]), relational verification (as provided by [9]), and ordinary program verification for assertions (implemented in many tools). Our main theorem says that the regime enforces the security property. It relies on existing soundness results for the type system and logics. For practical application, one needs logics and security type system for a richer programming language. For the sequential core of Java, suitable type systems [7, 27] and logics [1] also exist and have been proved sound.

Outline. Sect. 2 illustrates by examples the rich declassification policies of interest. It informally introduces our policy notation, dubbed *flowspecs*, and summarizes the security property and enforcement regime. Sect. 3 formalizes a simple programming language with declass commands. Sect. 4 defines flowspec policies. Sect. 5 defines our end-to-end security property. Sect. 6 addressess enforcement by type checking together with

verification conditions. Sect. 7 gives the main technical result, that statically checked programs are secure. (Some details are in the appendix.) Sect. 8 covers related work and Sect. 9 discusses future challenges.

2 Synopsis

A number of works provide techniques for enforcement of noninterference for imperative and object-oriented programs. One approach treats security labels as non-standard types [30]. By typing variable h as secret (H) and l as low (L), an evident rule disallows direct assignment of $l := h$ and additional constraints prevent implicit flows as in **if** h **then** $l := \mathit{true}$.

An alternative approach for enforcing noninterference is to formulate security as a verification problem and use program logic [15, 14]. The basic idea of noninterference is that if two initial states agree on the non-secret variables (thereby representing uncertainty about the secrets), and there are two runs of the program from those states, the resulting pair of final states agrees on low variables. The idea can be realized in terms of a "relational Hoare logic" [9]. We focus on the logic of Amtoft et al. [1] which addresses a key challenge for reasoning: mutable data structure in the heap. A triple $\{\varphi\}C\{\psi\}$, termed *flowtriple* in the sequel, involves pre- and post-conditions φ, ψ on pairs of program states. Consequently, the interpretation is with respect to *two* executions of C, one from each of the initial pair. Pre- and postconditions can include predicates of a special form, which we call an *agreement* and write as $\mathbf{A}(l)$: the meaning is that the two considered states agree on the value of l. Agreements can also involve *region* expressions which abstract the heap. The problem of showing that a command C is noninterferent for some low variables $l_0, \ldots, l_n$ reduces to showing the validity of the triple $\{\varphi\}\ C\ \{\varphi\}$ where φ is $\mathbf{A}(l_0) \wedge \ldots \wedge \mathbf{A}(l_n)$. Compositional proof rules provide flow-sensitive reasoning and incorporate the usual rules that disallow direct writes of high values into low variables/fields and that disallow low writes in branches of conditionals. Our use of flowtriples will typically be for a single assignment command.

Specification of Delimited Release policies. In a precondition, an agreement expresses what is considered visible. So the logical formulation of noninterference can be used in a natural way to describe *delimited release* policies of Sabelfeld and Myers [23], who consider this example akin to an electronic wallet scenario (with $l, k : \mathrm{L}$ and $h : \mathrm{H}$):

$$\textbf{if } h \geq k \textbf{ then } h := h - k; l := l + k \textbf{ else skip}$$

To express the policy that it is fine to reveal whether $h \geq k$, but nothing more about h, our specification (termed a *flowspec* in the sequel) is

$$\textbf{flow } \iota \textbf{ pre } P\&\varphi \textbf{ post } \varphi' \tag{1}$$

where ι is an identifier (to link with the declassification code). The precondition is $P\&\varphi$ where P is the state predicate *true*, and φ is the agreement, $\mathbf{A}(h \geq k) \wedge \mathbf{A}(l) \wedge \mathbf{A}(k)$; and,

φ' is the postcondition, $\mathbf{A}(l) \wedge \mathbf{A}(k)$. The meaning of the flowspec is this. A command C satisfies the specification provided that if it is run twice, from initial states that agree on l, on k, and on the value of expression $h \geq k$ —but not necessarily on the value of h— the final states agree on l and on k. In this particular example P is *true* but it need not be, as demonstrated later. Also, when clear from context, we will elide the label of the flowspec.

Release after multiple events. The next example has declassification conditioned upon multiple events. Consider a patient's medical record that contains fields with mixed data, some secret and some public. A bookkeeper needs to release the patient's information to an insurance representative subject to the following policy.

- The patient's diagnosis is released, but not the doctor's notes (both are normally secret).
- The version of the record to be released should be the most recent one.
- The record should be in "committed" state. The database contains some versions that record saved test info; a committed record reflects a doctor's firm diagnosis.
- Preceding release, an audit log entry is made, including the patient ID and record version, as well as the IDs of the bookkeeper and insurance rep.
- At the time of release, both bookkeeper and representative should be users with valid credentials to act in their respective roles.[2]

The example is illustrative, but in this paper, we do not consider issues such as integrity (e.g. of the audit logs) or roles; nor do we restrict to representatives of the patient's insurance company.

Let security level L be associated with information for which at least the insurance company is permitted access, and H be associated with private patient information and clinic-internal information. The clinic's database contains records of this form:

```
class PatientRecord {
  int id;   boolean committed;  int vsn;
  String{H} diagnosis;  String{H} notes;  }
```

A similar record is provided to insurance representatives. Note that L fields are unmarked.

```
class InsRecord {
  int id;  String diagnosis; }
```

Before formalizing the policy we give a conforming implementation (in Java-like syntax).

[2]We want to enforce this policy in the code, but one might also include in the log entry a timestamp, to facilitate *post hoc* correlation with the authentication logs to check that the bookkeeper and representative were indeed authorized.

```
Object release(Database db, int patID,
               Bookkeeper b, InsRep r)
  pre: sys.auth(b,"book") && sys.auth(r,"rep")
{
  InsRecord ir := new InsRecord();
  PatientRecord pr := db.lookup(patID);
  if (pr != null && pr.committed) {
    log.append(b.id, r.id,
               pr.id, pr.vsn, "release");
    ir.id := pr.id;
    ir.diagnosis := pr.diagnosis;
    return ir;
  } else { return new Msg("not available"); }
}
```

Note that the parameters and local variables are all L (unmarked). Only certain fields of patient records are marked H. With this labeling, the program typechecks except for the assignment[3]

```
ir.diagnosis := pr.diagnosis;
```

Our approach turns this assignment into a declassification that we would explicitly exempt from typechecking. The declassification itself must satisfy its flowspec(s). We must also check that the flowspec and the declassification are *healthy* (Def. 4.1): any low write, in this case `ir.diagnosis`, is reflected in the postcondition of the flowspec, e.g., as $\mathbf{A}(ir.diagnosis)$.

The flowspec to be attached to this assignment is of the form (1) where the state predicate P in the precondition is:

$$\begin{array}{l} pr.committed \wedge db.recent(pr) \\ \wedge sys.auth(b, \text{'book'}) \wedge sys.auth(r, \text{'rep'}) \\ \wedge log.contains(b.id, r.id, pr.id, pr.vsn, \text{'release'}) \end{array}$$

and φ is $\mathbf{A}(pr.diagnosis)$ and φ' is $\mathbf{A}(ir.diagnosis)$. Predicate P makes explicit that the release of $pr.diagnosis$ happens after multiple conditions have been satisfied. Informally, the multiple conditions represent a sequence of events that must transpire before release can happen. Predicate $db.recent(pr)$ expresses that pr is the most recent patient record; $sys.auth(b, \text{'book'})$ says that b is authenticated by sys as bookkeeper, etc. The precondition $\mathbf{A}(pr.diagnosis)$ allows partial release of the patient record. The other preconditions express the conditions under which this release is permitted.

This policy is healthy: the flowspec is not inside a high conditional; its postcondition covers the modified locations; the state predicate in its precondition covers the possible states just before the assignment to `ir.diagnosis`, as we now argue in detail. The conjunct $pr.committed$ holds owing to the guard condition of the **if**. The conjuncts

[3]Systems like Jif do some amount of inference for labels of internal variables etc. In that case, this example would still be rejected but it might not be as clear which sub-program is offensive.

$sys.auth(b, \text{'book'})$ and $sys.auth(r, \text{'rep'})$ are preconditions to the method —its calls must therefore be verified for these conditions. Recency should be a consequence of the specification of *lookup*. (This would get more complicated if we considered concurrent access to the database; the policy is perhaps too strong on this point.) Presence of the log entry is ensured by the call to *append*.

Declassifying a data structure. Suppose as a result of a disaster relief plan we would like to release all patient records together with their respective diagnoses, but not necessarily the doctors' notes. We use an alternate version of *PatientRecord* given by:

```
class PatientRecord<alpha, beta> {
  int id;   boolean committed;  int vsn;
  String{alpha} diagnosis;  String{beta} notes;
  PatientRecord<alpha, beta> next; }
```

Note that now the levels of *diagnosis* and *notes* are type parameters, and can be instantiated based on context [27]. Patient records are linked by the *next* field. One could envisage a method which retrieves all patient records from the database by returning a *PatientRecord* object that serves as a root for iterating over all records. Since we are assuming that *diagnosis* and *notes* contain secret values, the returned root must be of type $PatientRecord\langle \text{H}, \text{H}\rangle$ and the label on `root` itself must be *low* (recall unmarked means low) to facilitate iterating over the list:

```
PatientRecord<H, H> root := db.lookupAll();
```

At this point we would like to declassify the entire list of patient records so that the diagnosis of each patient is no longer secret while the doctors' notes remain secret. The release is achieved by a single assignment:

```
PatientRecord<L, H> newRoot := root;
```

This assignment would be rejected by the typechecker owing to level L in the type of `newRoot`. However, we would like to turn this assignment into a declassification together with the following flowspec:

$$\begin{array}{l}\textbf{pre } root \in R \wedge R.next \subseteq R \;\&\; \mathbf{A}(root) \wedge \mathbf{A}(R.diagnosis)\\ \textbf{post } \mathbf{A}(newRoot) \wedge \mathbf{A}(newRoot.next^*.diagnosis)\end{array}$$

This says that, for any two runs, (a) if in each run *root* is in some region R of the heap (i.e., set of object references) and all nodes reachable from *root* by *next* are in R; and, (b) *root* values agree and all *diagnosis* values agree for the two runs; then, the records reachable from *newRoot* agree on *diagnosis* values. Consequently the lists are indistinguishable. This kind of reasoning is possible in the logic of [1], which introduced agreements for heap regions. (We take liberties with their notation, however, for expository purposes.)

It is a known problem [3] that such a program is susceptible to a laundering attack since there may be high aliases to *newRoot* (e.g. *root* is one such alias). Therefore any subsequent updates to *diagnosis* must be allowed only if they are made through a high

reference not in R. The problem can be avoided by cloning the entire date structure, as in [3]. Alternatively, the requisite isolation can be enforced using region specifications or with the aid of pointer confinement regimes like ownership types [13, 6]. Detailed investigation is beyond the scope of this paper.

Conditioned gradual release. By itself, checking of security-labelled types should enforce noninterference. (A practical checker of this kind is Jif [19]; others have been formally validated [26, 20].) But we exempt declassification commands from type checking! Instead, each declassification should form a valid flowtriple for its associated flowspec; and this only says something about the declassification code in isolation. To put the two together we propose an enrichment of the gradual release property [2]. An observer at level L sees each *low action* —assignment to a low variable, declassification step, or termination. The low observer should gain no knowledge about secrets except from declassifications, and then what is learned is only what is allowed by the associated flowspec precondition. That is, they learn no more than they would know if told the current value of each e for which $\mathbf{A}(e)$ is in the precondition (together with what is known from any previous releases). Finally, a declassification step must not be taken except from a state that satisfies the state predicate part of the flowspec precondition. The formal definition is in Sect. 5.

In the absence of declassification, conditional gradual release amounts to noninterference: knowledge remains constant through every step of a computation.

Assignments to high variables. This is a small subtlety in our security property. A declassification policy refers to the *current value* of expressions e that occur in agreements $\mathbf{A}(e)$ of the policy's precondition. This poses a risk of laundering: If $h0$ and $h1$ are high variables, a policy using precondition $\mathbf{A}(h0)$ would appear to allow the release of $h0$, but if the policy is attached to the program at a point in the program following assignment $h0 := h1$, then in fact what is released is initial value of $h1$ (if the latter has never been reassigned). The problem has been noted before and a solution is to disallow reassignment of high variables used in declassifications [23]. (This becomes less straightforward for programs acting on heap objects; then regions or some other form of separation would be needed.) The point is that some high variables correspond to inputs, to which policies refer. The other high variables are used to compute intermediate results and outputs; confidentiality policies need not refer to them. For reasons of parsimony, we refrain from formalizing this distinction: It happens that, even without imposing any restriction on assignment to high variables, our security property and related notions are well defined and main results are provable.

In terms of the informal principles of [24], this design decision means that the formalized version of our proposal is susceptible to laundering attacks. Nonetheless, the gap between formal specifications and informal requirements is often tricky. In the case of information flow there are analyses to check whether a program releases more than a flowspec explicitly permits [5].

$$
\begin{array}{lcl}
C, B, M & ::= & \textbf{declass}\ \iota\,\langle C\rangle \mid x := e \mid C; C \mid \textbf{skip} \mid \\
 & & \textbf{if}\ e\ \textbf{then}\ C\ \textbf{else}\ C \mid \textbf{while}\ e\ \textbf{do}\ C \\
e & ::= & x \mid 0 \mid 1 \mid \ldots \mid e + e \mid e \leq e \mid \ldots
\end{array}
$$

Figure 1: Commands; ι ranges over flowspec identifiers.

Atomicity. On one hand, intermediate steps must be observed to give meaning to policies that refer to conditions during computation. On the other hand, our attackers cannot see or count high steps, as we aim for practical enforcement without need to consider timing. We consider a declassification to be an atomic step, in order to use results and implementations of relational logic. As illustrated by the examples, and mirrored by key release in [2] and declassify expressions in Jif [19], it is often easy to identify a single action that can serve as the point of declassification.

We formalize a more general notion, that an arbitrary command can be designated as a declassifier. This admits mis-use, however. Making declassification atomic can hide leaks that would be visible at intermediate steps. For example, consider the command

$$l := h0;\ l1 := h0;\ l1 := 0;\ l := l + h1$$

It this is treated as atomic, it releases $h0 + h1$ only, but without atomicity it also reveals $h0$ at the moment following the second assignment. To strictly adhere to the principle of *non-occlusion* [24], we could add atomic blocks to the language and treat declass as applying only to such blocks.

3 Programming language

To focus on the key ideas in a comprehensible way, we refrain from considering pointers, procedures, or other language features. The formalization uses the simple imperative language over integer variables, augmented with the declassification command.

The command "**declass** $\iota\,\langle C\rangle$" executes C atomically; its syntax includes a *flowspec identifier*, ι, used to associate the command with its policy. A well-formed program has no nested declass commands and each declass has a unique identifier ι. For given ι there may be multiple flowspecs labelled ι.[4]

A *state* s is a mapping from variables to values, and we write $s[x := n]$ for updates. The semantics is given as a deterministic transition relation, $\rightarrow$, over *configurations* of the form $\langle\ C,\ s\ \rangle$ where s is a state and C is either a command or **stop**. The latter triggers an observable step to the *improper state*, ✓, for termination.

Figure 2 defines the semantics. We write $[\![e]\!](s)$ for the value of expression e in state s. The semantics of declass refers to the transitive closure $\rightarrow^+$ of the transition relation. If $\langle\ C,\ s\ \rangle$ diverges then $\langle\ \textbf{declass}\ \iota\,\langle C\rangle,\ s\ \rangle$ has no successor; no other configurations are

[4]The uniqueness condition is a technical convenience; it loses no generality because one can simply copy shared flowspecs.

$$\langle\, \mathbf{stop},\ s\,\rangle \to \langle\, \mathbf{stop},\ \checkmark\,\rangle \qquad \langle\, \mathbf{skip},\ s\,\rangle \to \langle\, \mathbf{stop},\ s\,\rangle \qquad \langle\, x := e,\ s\,\rangle \to \langle\, \mathbf{stop},\ s[x := [\![e]\!](s)]\,\rangle$$

$$\frac{\langle\, C_0,\ s\,\rangle \to \langle\, C_0',\ s'\,\rangle \qquad C_0' \neq \mathbf{stop}}{\langle\, C_0; C_1,\ s\,\rangle \to \langle\, C_0'; C_1,\ s'\,\rangle} \qquad \frac{\langle\, C_0,\ s\,\rangle \to \langle\, \mathbf{stop},\ s'\,\rangle}{\langle\, C_0; C_1,\ s\,\rangle \to \langle\, C_1,\ s'\,\rangle}$$

$$\frac{[\![e]\!](s) \neq 0}{\langle\, \mathbf{if}\ e\ \mathbf{then}\ C_0\ \mathbf{else}\ C_1,\ s\,\rangle \to \langle\, C_0,\ s\,\rangle} \qquad \frac{[\![e]\!](s) = 0}{\langle\, \mathbf{if}\ e\ \mathbf{then}\ C_0\ \mathbf{else}\ C_1,\ s\,\rangle \to \langle\, C_1,\ s\,\rangle}$$

$$\frac{[\![e]\!](s) \neq 0}{\langle\, \mathbf{while}\ e\ \mathbf{do}\ C,\ s\,\rangle \to \langle\, C; \mathbf{while}\ e\ \mathbf{do}\ C,\ s\,\rangle} \qquad \frac{[\![e]\!](s) = 0}{\langle\, \mathbf{while}\ e\ \mathbf{do}\ C,\ s\,\rangle \to \langle\, \mathbf{stop},\ s\,\rangle}$$

$$\frac{\langle\, C,\ s\,\rangle \to^+ \langle\, \mathbf{stop},\ s'\,\rangle}{\langle\, \mathbf{declass}\ \iota\, \langle C\rangle,\ s\,\rangle \to \langle\, \mathbf{stop},\ s'\,\rangle}$$

Figure 2: Semantics; here s, s' range over non-✓ states.

stuck except the terminated one, $\langle\, \mathbf{stop},\ \checkmark\,\rangle$. There is no need to model divergent declass more sensibly, because the proof obligations (Sect. 6) ensure that C always terminates.

Every command can be written in the form $C_0; C_1$ or C_0, where C_0 is not a sequence, and then we call C_0 the *active command*. The active command is the one that gets replaced in a transition step. Define $actc(\langle\, C,\ s\,\rangle)$ to be the active command of C, and define $actc(\langle\, \mathbf{stop},\ t\,\rangle) = \mathbf{stop}$. Define $code(\langle\, C,\ t\,\rangle) = C$ and $state(\langle\, C,\ t\,\rangle) = t$.

4 Policy specification

This section formalizes policies as we use them in the paper, but also discusses the practical scenario where policy is separated from the code.

The *baseline security policy* for a program M is a mapping Γ from the variables[5] of M to the security levels $\{\text{L}, \text{H}\}$. Policy specifications for declass commands, which are exempt from the baseline policy, are given as *flowspecs* which take the form $\mathbf{flow}\ \iota\ \mathbf{pre}\ P\&\varphi\ \mathbf{post}\ \varphi'$. Here φ, φ' are agreement formulas, i.e., conjunctions of the form $\mathbf{A}(e_0) \wedge \mathbf{A}(e_1) \wedge \ldots \mathbf{A}(e_k)$ where the e_i are expressions which may contain low and high variables. In this paper we do not give a syntax and semantics for state predicates P. We assume it is two-valued, so $s \models P$ means that P is true in state s and otherwise it is false.

Of course $P\&\varphi$ is interpreted in a pair of (non-✓) states. Define $s, t \models P\&\varphi$ iff $[\![e_i]\!](s) = [\![e_i]\!](t)$ for $0 \leq i \leq k$, and moreover $s \models P$ and $t \models P$.

[5]For a language with heap objects, object fields should also be labelled and class types can be parameterized on levels [19, 27] but, in keeping with the semantics of [1] pointer values, like primitive values, are not labelled.

Healthiness conditions. For the streamlined formalization in this paper, the healthiness conditions involve both the flowspec and the declassification to which it is attached.

Definition 4.1 (healthy policy) A program and its policy are *healthy* provided (a) There is *postcondition coverage:* If the command associated with **flow** ι **pre** $P\&\varphi$ **post** φ' is **declass** $\iota\,\langle B\rangle$ then for every low variable x written by B, φ' contains $\mathbf{A}(x)$. (b) There is *flowspec coverage:* there is at least one flowspec for every flowspec identifier that occurs in a declass command.

There should be no declass commands in high conditionals and every declass command should be correct in the sense that it establishes the agreements φ' from states that satisfy $P\&\varphi$ —but this pertains to the enforcement of policy, the topic of Sect. 6.

In the presence of pointers, an additional healthiness condition would ensure that the region indirectly released via a root pointer is suitably confined, as per the laundering example mentioned in Sect. 2.

As mentioned in Sect. 2, an important policy guideline is to avoid updates of high variables for which there are flowspecs. The point is that knowledge is defined in terms of initial values of secrets, whereas an agreement $\mathbf{A}(e)$ in a flowspec is interpreted at the point of declassification. Of course some high variables serve as high outputs or for intermediate computation. In a full programming language it would be natural (and easy) to designate secret inputs, to which flowspecs might refer, and disallow updates to them. High updates to declassified variables are disallowed in the delimited release type system [23] but we refrain from adding this complication since our results hold regardless.

Separating policy from the system to which it applies. Labeling of variables is somewhat separate from the code that acts on them (and in practice only external interfaces need be labelled, the rest can be inferred). One can imagine partial release policies being expressed using an augmented labeling that designates levels for certain "escape hatch" expressions, overriding the level given by usual typing rules; e.g., $h \geq k$ could be declared low despite the join of its variable levels being high. This is explored by Hicks et al [17]. Similarly, we believe that many richer policies can be expressed using schematic flowspecs. For the example in Sect. 2, the assignment $ir.diagnosis := pr.diagnosis$ makes sense in any context where ir and pr are declared. Moreover, the pre- and post-conditions refer to fields of these objects and to global data structures (the log and the authentication system). So it can be read as a schematic policy, applicable to any assignments from fields of `PatientRecord` objects to `InsRecord` objects (in any context where the globals are visible). Such a schema could be used to automatically exempt matching subprograms from type-checking and at the same time impose proof obligations on each match. Experimental investigation is needed and is beyond the scope of this paper.

5 The end-to-end security property

Let M be a fixed program throughout this section, and Γ the baseline security policy, i.e., assignment of levels to the variables of M. To lighten various notations we suppress their dependence on M and Γ. Also given is a set of flowspecs. This section defines the semantics of the policy.

Whereas the gradual release paper [2] defines knowledge in terms of observations, i.e., sequences of the low-visible parts of states, our definition is in terms of traces of complete states which are needed to interpret flowspec preconditions.

What a low observer knows about the initial state after observing the visible part of some trace σ is that it could be any state that yields a trace τ low-indistinguishable from σ. The precise definitions are somewhat involved and are carefully designed to facilitate proof of the soundness theorem (in a way that can be extended to richer languages).

Runs, actions, and traces. An initial configuration $\langle\, M,\ s \,\rangle$ determines a unique finite or infinite *run*, that is, the maximal sequence of configurations given by the transition relation. We use the term *pre-run* for a finite, non-empty prefix of a run.

An *action* is a transition step for an assignment, declassification, or termination (i.e., to ✓). The other transitions, e.g., those for **if** and **while**, never change the state. Assignment to a high variable is a *high action*; the others are low actions. It is convenient to work with a notion of trace which extracts from a pre-run the series of states resulting from actions (both low and high). Later we also purge states resulting from high assignments.

Definition 5.1 (traces of M, generating pre-run) For any pre-run S, let $\mathit{trace}(S)$ be the sequence of states starting with $\mathit{state}(S_0)$ and thereafter including every state that results from an action. For any state s, let

$$\mathit{Traces}(s) = \{\mathit{trace}(S) \mid S \text{ is a pre-run from } s\}$$

We say S is a *generating pre-run for* σ, if $\mathit{trace}(S) = \sigma$. Define $\mathit{TRACES} = \cup s \cdot \mathit{Traces}(s)$.

A trace σ can have more than one generating pre-run. The *minimal generating pre-run* for σ is just the shortest one, i.e., with no unnecessary steps at its tail. (It is unique since the run is determined by σ_0.)

We do not distinguish between a state s and the singleton trace consisting of s. Juxtaposition is used to express catenation, e.g., $\sigma\, s$ is the trace consisting of σ followed by s. Also σ_i is the ith element, counting from zero. We write $\mathit{last}(\sigma)$ for $\mathit{len}(\sigma) - 1$ and abbreviate $\sigma_{\mathit{last}(\sigma)}$ as σ_{last} when confusion seems unlikely.

Low observations. To eliminate timing flows, we use an alternate form of trace that omits states resulting from assignments to high variables. This is only used to define indistinguishability; usually we work with unpurged traces.

Definition 5.2 (purged traces, *purge*) For any pre-run S, let *p-trace*(S) be the same as *trace*(S) except omitting states that result from assignments to variables[6] x with $\Gamma(x) =$ H. Define *purge*(σ) to be the p-trace determined by a generating pre-run for σ.

(All generating pre-runs for σ yield the same p-trace.)

Two traces are considered indistinguishable if there is a one-to-one correspondence between the states resulting from actions and moreover corresponding states are low-equivalent.

Let *lowvis*(s) be the restriction of a (proper) state to its low variables (according to Γ), and define *lowvis*$(\checkmark) = \checkmark$.

Definition 5.3 (indistinguishable ($\sim$)) Let σ and τ be traces. Define $\sigma \sim \tau$ iff *lowvis*$(purge(\sigma)) =$ *lowvis*$(purge(\tau))$, where we map *lowvis* over each state in the sequence.

Indistinguishability $s \sim t$ for singleton traces s, t is the same as low equivalence, *lowvis*$(s) =$ *lowvis*(t), because there is no stuttering to remove. Some authors write $s =_L t$ for *lowvis*$(s) =$ *lowvis*(t). Note that if $\sigma \sim \tau$, we have $\sigma_{last(\sigma)} = \checkmark$ iff $\tau_{last(\tau)} = \checkmark$.

Definition 5.4 (observed knowledge) For $\sigma \in TRACES$, define $\mathcal{K}(\sigma)$ by $\mathcal{K}(\sigma) = \{s \mid \exists \tau \in Traces(s) \cdot \sigma \sim \tau\}$.

Despite the name, this represents the low observer's uncertainty or ignorance about the initial state.

Proposition 5.5 Knowledge is monotonic: $\mathcal{K}(\sigma\, t) \subseteq \mathcal{K}(\sigma)$ for any σ and state t such that $\sigma\, t \in TRACES$.[7]

Revelation. The connection between flowspecs and traces rests on the following notion, which is used to express a bound on the knowledge that can be gained by observing a declassification step. The bound is expressed in terms of a flowspec precondition $P\&\varphi$, using a special notion of knowledge, written $\mathcal{R}(\sigma,\ P\&\varphi,\ \iota)$, which will be used only when σ is a trace leading up to an execution of **declass** $\iota\, \langle B \rangle$. Intuitively, $\mathcal{R}(\sigma,\ P\&\varphi,\ \iota)$ represents the set of initial states s from which there is a trace $\tau \in Traces(s)$ with $\sigma \sim \tau$ and moreover $(\sigma_{last}, \tau_{last}) \models P\&\varphi$. The idea is that τ is also poised to do a declassification, from a state that matches σ_{last} in terms of the flowspec precondition. But formalizing $\mathcal{R}$ in these exact terms would be unsound, because it would admit the possibility that τ does not reflect the full run up to the point of declassification, and subsequent high steps could falsify P or the relation φ before the declassification happens.[8]

[6] In the extension of our work to programs using heap objects, a field update is considered as a high assignment just if the field label is high, regardless of the variables and fields by which the object is reached.

[7] The condition $\sigma\, t \in TRACES$ ensures that σ does not end in the improper state, $\checkmark$.

[8] If we restrict P to depend only on low variables, and φ to have agreements only for designated high variables that are never reassigned, then the intuitive description is indeed an adequate formalization.

Definition 5.6 (revealed knowledge, $\mathcal{R}$) For state predicate P, agreement formula φ, flowspec identifier ι, and $\sigma \in \mathit{TRACES}$, define $\mathcal{R}(\sigma, P\&\varphi, \iota)$ to be the set

$$\{s \mid \exists S \cdot \begin{array}{l} S \text{ is a pre-run from } s \text{ with } \sigma \sim \mathit{trace}(S) \\ \text{and } \mathit{actc}(S_{last}) \text{ is } \mathbf{declass}\ \iota\,\langle B\rangle \\ \text{and } (\sigma_{last}, \mathit{state}(S_{last})) \models P\&\varphi \ \} \end{array}$$

By uniqueness of flowspec identifiers, ι determines the body B of $\mathbf{declass}\ \iota\,\langle B\rangle$.

A straightforward consequence of the definitions is that

$$\mathcal{R}(\sigma, P\&\varphi, \iota) \subseteq \mathcal{K}(\sigma)$$

for any $\sigma, P, \varphi, \iota$. Our security property says that in a step that extends trace σ to $\sigma\, u$, if there is a gain of knowledge, i.e., a strict inclusion $\mathcal{K}(\sigma\, u) \subset \mathcal{K}(\sigma)$, then $\mathcal{K}(\sigma\, u)$ is no smaller than $\mathcal{R}(\sigma, P\&\varphi, \iota)$.

Definition 5.7 (CGR, conditioned gradual release) The program M under consideration satisfies *conditioned gradual release* iff the following holds for all commands C, D, traces σ, and states s, t, u: For any pre-run that generates σ and ends with $\langle\, C,\ t\,\rangle$, if $\langle\, C,\ t\,\rangle \to \langle\, D,\ u\,\rangle$ then

1. if the active command in C is **stop** or an assignment to some variable x with $\Gamma(x) = \text{L}$ then $\mathcal{K}(\sigma\, u) \supseteq \mathcal{K}(\sigma)$
2. if the active command in C is $\mathbf{declass}\ \iota\,\langle B\rangle$ then there is some flowspec for ι with precondition $P\&\varphi$ such that
   (a) $t \models P$, and
   (b) $\mathcal{K}(\sigma\, u) \supseteq \mathcal{R}(\sigma, P\&\varphi, \iota)$

Here $t \models P$ expresses that release only happens under designated conditions. The partial release constraint is supposed to be captured by $\mathcal{K}(\sigma\, u) \supseteq \mathcal{R}(\sigma, P\&\varphi, \iota)$.

Owing to monotonicity of knowledge, the condition in item 1 is equivalent to $\mathcal{K}(\sigma\, u) = \mathcal{K}(\sigma)$. On the other hand, the inclusion in item 2b bounds the knowledge $\mathcal{K}(\sigma\, u)$ and is not an equality in general. Whereas $\mathcal{R}(\sigma, P\&\varphi, \iota)$ is what would be known if all information allowed by φ was revealed, $\mathcal{K}(\sigma\, u)$ is what is known upon observing $\sigma\, u$.

Examples. We now consider some example programs and check whether CGR holds by calculating observed and revealed knowledge. We write "$\mathbf{declass}\ \varphi\ \langle C\rangle\ \varphi'$" to abbreviate $\mathbf{declass}\ \iota\,\langle C\rangle$ where ι is a fresh label tied to the evident flowspec.

The point of the first program (from [2]) is to show that knowledge increases over time. The program satisfies CGR. (We have worked out the other programs in [2, Sect. 2] and our results conform to theirs).

(2)
$$\begin{array}{l} \mathbf{declass}\ \mathbf{A}(h \neq 0)\ \langle l := (h \neq 0)\rangle\ \mathbf{A}(l); \\ \mathbf{if}\ l\ \mathbf{then}\ C\ \mathbf{else\ skip} \end{array}$$

Let C be $\mathbf{declass}\ \mathbf{A}(h1)\ \langle l1 := h1\rangle\ \mathbf{A}(l1)$. Consider initial states where l and $l1$ are both *true* (the other cases are similar). We informally discuss calculations for initial state, v, where $v = [l : \mathit{true}, l1 : \mathit{true}, h1 : \mathit{false}, h : a]$, for some $a \neq 0$. The traces from v are $(v\ v\ v'\checkmark)$ and all non-empty prefixes thereof, where $v' = v[l1 := \mathit{false}]$. The traces that reflect declassification are $(v\ v)$ and $(v\ v\ v')$.

Clearly, v is in the observed knowledge of $(v\ v)$. It is easy to see that the state, $t = [l : \mathit{true}, l1 : \mathit{true}, h1 : \mathit{true}, h : b]$, for some $b \neq 0$ is also in the observed knowledge of $(v\ v)$. For trace $(v\ v\ v')$, however, the observed knowledge is the singleton set $\{v\}$. This represents an increase of knowledge over time as new declassifications take place. To check CGR for trace $(v\ v)$, note that the revealed knowledge from v and the flowspec precondition, $\mathbf{A}(h \neq 0)$ is again $\{t, v\}$ and this bound continues to be maintained by $\mathcal{K}(v\ v)$. Similarly, CGR is maintained for $(v\ v\ v')$ and for all other traces.

The next program is the sequence

$$\begin{array}{l}\mathbf{declass}\ \mathbf{A}(h \geq 0)\ \langle l0 := (h \geq 0)\rangle\ \mathbf{A}(l0);\\ \mathbf{declass}\ \mathbf{A}(h \leq 0)\ \langle l1 := (h \leq 0)\rangle\ \mathbf{A}(l1)\end{array} \tag{3}$$

The program satisfies CGR. Consider initial states where $l0$ and $l1$ are both *true* (the other cases are similar). We informally discuss calculations for initial state, u, where $u = [l0 : \mathit{true}, l1 : \mathit{true}, h : 0]$. The traces from u are $(u\ u\ u\checkmark)$ and all non-empty prefixes thereof. The traces that reflect declassification are $(u\ u)$ and $(u\ u\ u)$.

For observed knowledge, $\mathcal{K}(u\ u) = \{s, u\}$ where $s = [l0 : \mathit{true}, l1 : \mathit{true}, h : a]$ for some $a > 0$. At this stage, the attacker only knows the value of $h \geq 0$. However, $\mathcal{K}(u\ u\ u) = \{u\}$: from this trace and its final state u, the attacker learns that $h \geq 0$ and $h \leq 0$ hold simultaneously. That is, $h = 0$. The initial state, u, says as much.

Finally, we consider a program without declassification but with looping. This program is insecure but it would be considered secure by gradual release [2] which is termination-insensitive.

$$l := \mathbf{true};\ \mathbf{if}\ h\ \mathbf{then}\ C0\ \mathbf{else}\ C1 \tag{4}$$

Let $C0$ be $l := \mathbf{false}$ and $C1$ be $\mathbf{while\ true\ do\ skip}$. For reasons of symmetry, we will consider initial states where l is initialized to *true*. The possible initial states are $s = [l : \mathit{true}, h : \mathit{false}]$ and $t = [l : \mathit{true}, h : \mathit{true}]$. The traces from s are s and $(s\ s)$, while the traces from t are $(t\ t\ t'\checkmark)$ and all prefixes thereof. Note that the observed knowledge for traces s, $(s\ s)$ and $(t\ t)$ is $\{s, t\}$: from observing them the attacker is unsure of the initial value of h. However, $\mathcal{K}(t\ t\ t') = \{t\}$; this means that CGR does not hold as the knowledge has increased without declassifications. Indeed from $l : \mathit{false}$ in t', the attacker can find the initial value of h.

Noninterference. One of the prudent principles [24] is that the security property should reduce to noninterference in the absence of declassifications.

Definition 5.8 (noninterference) The program M under consideration satisfies *noninterference* iff $\mathcal{K}(\sigma) = \mathcal{K}(\sigma_0)$ for all $\sigma \in \mathit{TRACES}$.

That is, knowledge after σ is the same as knowledge after the singleton trace σ_0 consisting of the initial state. The notion is termination-sensitive, in the sense that liveness of one trace must be reflected by liveness of the other.

Lemma 5.9 (characterization of noninterference) M is noninterferent iff the following holds for all s, t, σ, C': If $s \sim t$ and $\sigma \in \mathit{Traces}(s)$ then there is some $\tau \in \mathit{Traces}(t)$ such that $\sigma \sim \tau$.

Proposition 5.10 If C has no declassifications, then it is noninterferent iff it satisfies CGR.

Proof: In one direction, CGR says that $K(\sigma_0) = K(\sigma_0\sigma_1) = K(\sigma_0\sigma_1\sigma_2) = \ldots = K(\sigma)$, whence noninterference. In the other direction, use monotonicity of knowledge. □

6 Enforcement regime

In this section we consider a fixed main command M and baseline policy Γ, together with a collection of flowspecs that satisfies the healthiness conditions (Def. 4.1). In Sect. 7 we show that M is secure in the sense of CGR provided that it typechecks and moreover the declass commands, the bodies of which are exempt from typechecking, are correct with respect to their flowspecs.

Correctness is defined in terms of validity of a *flowtriple* $\{P\&\varphi\}\ C\ \{\varphi'\}$, which means that two runs of C from a pair of states, both satisfying P and related by φ, both terminate and the final states are related by φ'.

Definition 6.1 (validity of flowtriple) Say $\{P\&\varphi\}\ C\ \{\varphi'\}$ is *valid* iff for all states s, t, if $s, t \models P\&\varphi$ there exist non-✓ states s', t' such that $\langle\, C,\ s\,\rangle \to^* \langle\, \mathbf{stop},\ s'\,\rangle$ and $\langle\, C,\ t\,\rangle \to^* \langle\, \mathbf{stop},\ t'\,\rangle$ and $s', t' \models \varphi'$.

Owing to the semantics of agreements, validity can be factored into two parts: the ordinary total correctness statement $\{P\}\ C\ \{\mathit{true}\}$ (i.e., termination of C from P) and the partial-correctness version of the relational part.

The typing rules are given in Figure 3 and discussed later.

Definition 6.2 (statically secure) We say M is *statically secure* provided the following three conditions hold.

typechecks: we have $\Gamma \vdash M : \lambda$ for some level λ.

precondition coverage: Let the flowspecs for ι be

$$\textbf{flow}\ \iota\ \textbf{pre}\ P_i\&\varphi_i\ \textbf{post}\ \varphi_i' \quad \text{for } i \text{ from } 0 \text{ to } k$$

$$\Gamma \vdash \mathbf{stop} : \mathrm{L} \qquad \Gamma \vdash \mathbf{skip} : \mathrm{H} \qquad \Gamma \vdash \mathbf{declass}\ \iota\,\langle C\rangle : \mathrm{L} \qquad \frac{\Gamma \vdash C_0 : \lambda_0 \qquad \Gamma \vdash C_1 : \lambda_1}{\Gamma \vdash C_0; C_1 : \mathit{min}(\lambda_0, \lambda_1)}$$

$$\frac{\Gamma \vdash e : \lambda \qquad \lambda \leq \Gamma(x)}{\Gamma \vdash x := e : \Gamma(x)} \qquad\qquad \frac{\Gamma \vdash e : \mathrm{L} \qquad \Gamma \vdash C : \mathrm{L}}{\Gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ C : \mathrm{L}}$$

$$\frac{\Gamma \vdash C_0 : \lambda_0 \quad \Gamma \vdash C_1 : \lambda_1 \quad \Gamma \vdash e : \lambda \quad \lambda \leq \mathit{min}(\lambda_0, \lambda_1)}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ C_0\ \mathbf{else}\ C_1 : \mathit{min}(\lambda_0, \lambda_1)}$$

Figure 3: Security typing rules; here λ ranges over H, L.

Let $PP = P_0 \vee \ldots \vee P_k$. Then it is valid to assert PP immediately before the associated declass command.[9] [10]

relationally correct: For each **declass** $\iota\,\langle C\rangle$ and each flowspec for ι, the flowtriple $\{P_i \& \varphi_i\}\ C\ \{\varphi_i'\}$ is valid.

Because declass commands are typically just an assignment, or a short segment of code, their relational correctness should be easily checked by an automated verifier.

In many examples, precondition coverage is trivial. In general, precondition coverage can be the hardest part of enforcement: it may involve arbitrary assertions, e.g., confinement of data pointer data structures, the state of an authentication system, etc. But any verification system or method that applies to Floyd-Hoare partial-correctness assertions may be used (or none at all, as might be reasonable in some practical situations).

Type checking. Figure 3 give typing rules for the given policy Γ. For expressions, $\Gamma \vdash e : \lambda$ just means the highest level of a variable in e is λ. The rules define a judgement $\Gamma \vdash C : \lambda$ for commands that is intended to mean that C is secure and writes no variable of level below λ.

We must prevent unbounded computations with no observable steps (recall (4)). We choose a simple but restrictive way [29] for simplicity: high loops are not allowed. Recent work by Boudol [10] investigates more sophisticated type systems for this, and current program verification technology can automate termination checking in many cases.

Note that a declass is not allowed in high contexts and its body is not subject to any typing constraint.

The soundness proof relies on a number of semantic properties ensured by the type system.

[9] The precondition can always include $\mathbf{A}(x)$ for every low variable in the program. But there is no need to impose this as a healthiness condition; if some low x is read by C but the precondition φ doesn't include $\mathbf{A}(x)$, the relational correctness verification condition for C will not be valid (so of course not provable).

[10] In detail: Let $\mathcal{C}[-]$ be the context in which **declass** $\iota\,\langle C\rangle$ occurs. That is, the main program M has the form $\mathcal{C}[\mathbf{declass}\ \iota\,\langle C\rangle]$. Then $\mathcal{C}[\mathbf{assert}\ PP; \mathbf{declass}\ \iota\,\langle C\rangle]$ is a valid program annotation in the sense of partial correctness.

Lemma 6.3 (typing) (a) If $\langle\, C,\ s\,\rangle \to \langle\, C',\ s'\,\rangle$ and C is typable then C' is typable, indeed, the assignment of types to constituent commands is maintained. (b) Suppose the active command in C is low and is not a declassification: If $\langle\, C,\ s\,\rangle \to \langle\, C',\ s'\,\rangle$ and $t \sim s$ then there exists t' with $\langle\, C,\ t\,\rangle \to \langle\, C',\ t'\,\rangle$ and $t' \sim s'$. (c) If the active command of C is high and $\langle\, C,\ s\,\rangle \to \langle\, C',\ s'\,\rangle$ then $s \sim s'$. (d) If $\Gamma \vdash C : \mathrm{H}$ then C always terminates and every constituent command of C is also a high.

By (a), in any configuration we can refer to a unique level for each assignment, declassification, **if**, and **while** (we never need to refer to the level of a sequence, only to its constituent commands).

Note that typable commands make progress, but low commands need not terminate.

7 Soundness

To connect the static analysis with CGR we need a simulation-style characterization of the situation where an additional observed state does not increase knowledge.

The main definitions (Section 5) work at the level of traces —i.e., global observations of states— and observable distinctions on traces. But to show soundness of the enforcement regime we need a finer analysis at the level of runs.

If the active command of C is typed high then the *L-continuation of* C, written $\mathit{Lcont}(C)$, is the command D such that $C = B; D$ where $B : \mathrm{H}$ and $D : \mathrm{L}$. (The active command may be all or part of B.) Here we allow that D may be empty.

For traces σ, τ with generating pre-runs S and T, indistinguishability will be characterized using a notion of correspondence. It resembles a simulation, or rather an instantiation of a simulation but only for a given pair of runs.

Definition 7.1 (correspondence) Suppose S and T are pre-runs. A *correspondence from S to T* is a relation $Q \subseteq \mathit{dom}(S) \times \mathit{dom}(T)$ such that $0\ Q\ 0$ and for all i, j with $i\ Q\ j$ the following hold:

- (state agreement) $\mathit{state}(S_i) \sim \mathit{state}(T_j)$
- (level agreement) $\mathit{actc}(S_i) : \mathrm{L}$ iff $\mathit{actc}(T_j) : \mathrm{L}$
- (L code agreement) $\mathit{code}(S_i) = \mathit{code}(T_j)$ if $\mathit{actc}(S_i) : \mathrm{L}$
- (H code agreement) $\mathit{Lcont}(\mathit{code}(S_i)) = \mathit{Lcont}(\mathit{code}(T_j))$ if $\mathit{actc}(S_i) : \mathrm{H}$
- (monotonicity) if $i\ Q\ j$, $i < i'$, and $i'\ Q\ j'$ then $j \leq j'$; and symmetrically: if $i\ Q\ j$, $j < j'$, and $i'\ Q\ j'$ then $i \leq i'$
- (completeness) for every $i \in \mathit{dom}(S)$ there is some j with $i\ Q\ j$, and symmetrically

Lemma 7.2 () Suppose the main program M typechecks. If σ, τ are generated by S, T and there is a correspondence from S to T then $\sigma \sim \tau$.

Proof: Each state in σ that comes from a low action in S has, by correspondence conditions (level agree), (code agree L), and (state agree), a matching low action in T. These are in the right order by (monotonicity). □

The converse is more difficult and holds less generally. Unlike the soundness theorem, the correspondence Lemma 7.4 does not involve liveness and the proof would go through for a type system that allows high loops.

Definition 7.3 (trim traces) Define $trim(\sigma)$ to be the shortest prefix of σ such that $trim(\sigma) \sim \sigma$. (In other words, remove the longest suffix consisting only of states from high assignments.) Say σ *is trim* if $trim(\sigma) = \sigma$.

Note that $\sigma \sim \tau$ iff $trim(\sigma) \sim trim(\tau)$.

Lemma 7.4 (correspondence) Suppose the main program M typechecks. Suppose σ and τ are trim and $\sigma \sim \tau$. Let S (resp. T) be the minimal generating pre-run for σ (resp. τ). Then there is a correspondence from S to T.

Theorem 7.5 Suppose M is statically secure (Def. 6.2). Then it has the conditioned gradual release property (Def. 5.7).

8 Related work

Sabelfeld and Sands [24] systematically analyze many recent proposals for declassification, noting shortcomings and anomalies which motivate the "prudent principles" we address in Sect. 9. Another notable work is by van der Meyden [28]; it improves on Rushby's influential analysis of declassification policy that requires interfering flows to go via channels labelled at some level intended to represent trusted sanitization code [22]. Like ours, these works also distinguish low from high events and purge the latter as a way to remove timing leaks from consideration.

Our work builds very directly on the gradual release paper [2], more specifically on the semantic property introduced in the first part of the paper. We found some complication in adapting it to termination-sensitivity, which confirms the wisdom of their choice to use a simpler attack model, given their ambitious aims in the second part of the paper. That part extends gradual release to programs using cryptographic primitives; in particular, declassification is an atomic action achieved by releasing a previously secret key —the data of interest having already been released but encrypted under that key. In Sect. 2 of [2] there are brief comments on combining gradual release with delimited release [23] but no hints as to how this would be done or shown sound. Gradual release is proved to be enforced by a standard type system but with declass commands typed low [2]; this is in accord with our result, taking every flowspec to have agreements for all secrets read by the declass.

Any logic or verification system can be used to discharge the precondition coverage proof obligation (Def. 6.2), e.g., tools like ESC/Java and Spec# [8] or for highest assurance a provably sound verifier. To verify relational correctness (Def. 6.2), Benton's [9]

relational logic suffices for the simple imperative language of Sect. 3. Motivated by the ability of such logics to handle flow sensitivity, Amtoft et al [1] develop a relational logic that encompasses heap objects, using regions to express agreements (there called "independences") for anonymous objects. Besides the ability to prevent illegal flows while allowing standard programming idioms (including low/high aliases to objects with both low and high fields), the other key benefit is the ability to express fine-grained flow policies as we have proposed here. For encapsulating regions to prevent leaks via high aliases into declassified data structures, existing pointer confinement techniques can be used [13, 6] but further development of region-based relational logic, or separation logic, might give the best integration for our application.

In this paper we confine attention to preconditions of the form $P\&\varphi$ for expository purposes. The logics [9, 1] allow boolean combinations with agreements. Sabelfeld and Myers [23] speculate that their notion of conditional release might be extended to disjunctive policies, e.g. either h_1 or h_2 but not both can be released (these could be cards to be revealed in a round of a game). In a previous version of this paper, we speculated that the policy could be written as **flow pre** $\mathbf{A}(h_1) \vee \mathbf{A}(h_2)$ **post** $\mathbf{A}(\mathit{result})$, but Andrei Sabelfeld pointed out that this rejects the program that chooses nondeterministically between $\mathit{result} := h_1$ and $\mathit{result} := h_2$. Indeed, the flowspec also rejects $\mathit{result} := h_1$ which also satisfies the informal policy, indicating that the policy is questionable in the game scenario.

Our use of state predicates in release policy is inspired by Chong and Myers [12], who formulate the idea in an elegant way, relative to an abstract notion of "conditions" and means for verifying them. Policy is expressed by fancy types on variables that designate a series of "conditions" following which the secret may be released. They do not give examples where it is a temporal series of events, though the security property caters for that. Our proposal is more definite (and so less general) in tying conditions to state predicates, which in practice can express past events —by means of history variables if need be, though relevant history is often already available in the program state. Their security property is rather weak, as pointed out in [24]: the program is noninterferent until the conditions have been true, after which there is no constraint on what might leak. Another proposal for state-dependent labels [11] conditions the level on a boolean, ghost variable subject to updates in program annotations which thereby express where in the code declassification is allowed.

As discussed in more detail elsewhere [2, 24], several interesting proposals treat "where" declassification using notions of bisimulation that "reset" the program state at each release, in a way that for sequential code does not correspond to feasible attacks. One of the most advanced is by Mantel and Reinhard [18]; they combine "what" with "where" policies for multithreaded programs and provide type-based enforcement.

[This new work should be mentioned in our intro too.] Finally, Askarov and Sabelfeld [4] give a different combination "what" and "where" policies, dubbed localized delimited release. The idea is to instrument the semantics to track expressions that have been declassified. These are used in the standard way to define indistinguishability. The security property is defined as a kind of bisimulation where indistinguishability is with respect to

the expressions that have been declassified "up to now". The property is termination insensitive and differs from gradual release in that, although release cannot happen unless a declass command executes, the actual change in knowledge may come later, as illustrated by this example

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

where nothing is learned at the declass step, but h is learned in the last step. Remarkably, they show that security can be enforced by the type system for delimited release [23], with the additional restriction against declassification under high branch conditions. It could be interesting to adapt localized delimited release to use more semantic reasoning about equivalence of expressions, and to incorporate assertions in policies. The authors argue that their proposal is in accord with all the principles

9 Discussion

By using knowledge to describe information flow, our extension of the gradual release property [2] is able to capture conditions under which secrets are released, the extent to which they are released, and the absence of flows except by explicit downgrading actions. Our policy specifications make simple use of static security labels and program assertions so that information policy can be tied directly with application requirements and access mechanisms. To prove that the associated enforcement regime is sound, we devised an apparently novel technique: Owing to declassifications it does not seem possible to define a notion of simulation (or unwinding conditions) of the usual sort, but in some sense our proof constructs a simulation instance for a given pair of runs. Working out the details led to revision of several obvious but wrong definitions. We believe that our proofs address the main complications and that the technique will extend to the more complicated notion of low-equivalence used for heaps in both [1] and [27] for Jif-style level-polymorphic typing. Applicability to programs acting on the heap is essential for most practical applications.

Zdancewic [31] poses three "challenges for information-flow security". The first is integration with existing infrastructure. Our results suggest the use of typechecking (for simple security-labeled types) together with modest use of program specification for subprograms that must be exempt because of declassification or because typechecking is too conservative. Our approach fits well with access control. For example, consider the permissions and stack inspection in Java. This is easily modeled using program assertions over a ghost variable *SecCtx* that tracks currently enabled permissions [25]. To specify that h may be released but only if permission ph is enabled, the declass code is subject to two flowspecs. One covers the situation where the permission is enabled, with precondition $ph \in \mathit{SecCtx} \& \mathbf{A}(h)$. The other precondition keeps h secret: $ph \notin \mathit{SecCtx}$ (and has no agreements). As often happens, precondition coverage is trivial in this case.

Zdancewic's second challenge is to "escape the confines of pure noninterference"; he mentions both declassification and conservativity of flow-insensitive static analysis, both of which we address. The third challenge is to manage complex policies. We suggest that such policies should mostly be expressed using ordinary program specifications including

state-based descriptions of sophisticated access controls.

Sabelfeld and Sands [24] propose informal principles with which our proposal is largely in accord. *Semantic consistency* says that replacement of a "declass free" subprogram by a semantically equivalent one does not affect security. Clearly, for an attacker model in which intermediate states are visible, the relevant notion of equivalence is trace equivalence; for this, our proposal is semantically consistent. Of course such fine-grained observations disallow many standard compiler optimizations.

The principle of *conservativity* amounts to our Proposition 5.10 (this is problemmatic for [12] because their notion of security is not purely semantic). *Non-occlusion* says that adding declass cannot make a secure program insecure. (This only pertains to treatments of declassification in which, like ours, there is an explicit construct that can be "added" to a program.) Technically, our proposal violates this principle, because declass commands are atomic and can thus hide intermediate leaks in examples like **declass** $\langle l := h; l := 0\rangle$. As mentioned in Sect. 2, this can be fixed by adding a construct for atomic blocks and treating declass as a separate construct that can only be applied to atomic blocks. For the examples we have seen, it would suffice to restrict declass to apply to single assignments, which are atomic.

Askarov and Sabelfeld argue that localized delimited release [4] is in accord with all of the informal principles, noting in particular the need to use the semantic equivalence that preserves intermediate steps.

In addition to the separation of policy from code (see Sect. 4), two other topics need to be investigated before our approach can be advocated for practical use or explored in comprehensive case studies. One is the extension to concurrency, for which the knowledge based formulation seems well suited, given that declassification seems natural as a transaction. The other topic is integration with integrity; though integrity is in some ways dual to confidentiality, it is crucial for confidentiality that our declassification conditions depend on data with sufficient integrity. Finally there is the issue of laundering via pointer aliasing.

Acknowledgements The exposition in this version of the paper is improved thanks to feedback from anonymous CCS'07 referees and from Aslan Askarov and Andrei Sabelfeld.

References

[1] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *POPL*, 2006.

[2] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *2007 IEEE Symposium on Security and Privacy*. To appear.

[3] A. Askarov and A. Sabelfeld. Secure implementation of cryptographic protocols: A case study of mutual distrust. In *ESORICS*, 2005.

[4] A. Askarov and A. Sabelfeld. Localized delimited release: combining the what and where dimensions of information release. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 53–60, 2007.

[5] A. Banerjee, R. Giacobazzi, and I. Mastroeni. What you lose is what you leak: Information leakage in declassification policies. In *MFPS*, 2007. To appear.

[6] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *J. ACM*, 2005.

[7] A. Banerjee and D. A. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005. Special issue on Language Based Security.

[8] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: an overview. In *CASSIS*, 2004.

[9] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, 2004.

[10] G. Boudol. On typing information flow. In *ICTAC*, 2005.

[11] N. Broberg and D. Sands. Flow locks. In *ESOP*, 2006.

[12] S. Chong and A. C. Myers. Security policies for downgrading. In *CCS*, 2004.

[13] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, 2002.

[14] E. S. Cohen. Information transmission in sequential programs. In R. DeMillo, D. Dobkin, A. Jones, and R. Lipton, editors, *Foundations of Secure Computation*, 1978.

[15] D. E. Denning. *Cryptography and Data Security*. 1982.

[16] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *CCS*, 2006.

[17] B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification: high-level policy for a security-typed language. In *PLAS*, pages 65–74, 2006.

[18] H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In *ESOP*, 2007.

[19] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, 1999.

[20] D. A. Naumann. Verifying a secure information flow analyzer. In *TPHOLS*, 2005.

[21] R.Sailer, T. Jaeger, E. Valdez, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn. Building a MAC-based security architecture for the Xen opensource hypervisor. Technical Report RC23629, IBM Research.

[22] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, SRI, Dec. 1992.

[23] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *ISSS*, 2004.

[24] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. *Journal of Computer Security*, 2007.

[25] J. Smans, B. Jacobs, and F. Piessens. Static verification of code access security policy compliance of .NET applications. *Journal of Object Technology*, 2006.

[26] M. Strecker. Formal analysis of an information flow type system for MicroJava (extended version). Technical report, Technische Universität München, July 2003.

[27] Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *SAS*, 2004.

[28] R. van der Meyden. What, indeed, is intransitive noninterference? In *ESORICS*, 2007. To appear.

[29] D. Volpano and G. Smith. In *CSFW*, 1997.

[30] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 1996.

[31] S. Zdancewic. Challenges for information-flow security. In *PLID*, 2004.

A Appendix

The CGR definition (and $\mathcal{R}$ predicate) only appear to constrain the most recent declassification. But the previous ones are taken into account: they affect visible variables, so an assumption $\sigma \sim \tau$ already expresses that we are considering initial states that agree on the released parts of the secrets. This is evident in the proof of Lemma 7.4: though similar to a proof of noninterference, is really quite different: it says an indistinguishable pair of traces looks like a "noninterferent pair of traces", rather than saying every indistinguishable initial pair generates a noninterferent pair. At declassification steps, indistinguishability is given rather proved (contra the proof of Theorem 7.5).

Proof of Lemma 7.4. By construction. The following program builds a correspondence in variable Q, using integer variables i, j, ii, jj.

Initially, $i = j = ii = jj = 0$ and $Q = \{(0, 0)\}$. The program maintains the following loop invariants:

I0 $0 \leq i \leq \mathit{last}(S)$ and $0 \leq j \leq \mathit{last}(T)$ and $0 \leq ii \leq \mathit{last}(\sigma)$ and $0 \leq jj \leq \mathit{last}(\tau)$

I1 $i\, Q\, j$

I2 $S[0..i]$ is a generating pre-run for $\sigma[0..ii]$ and $T[0..j]$ is a generating pre-run for $\tau[0..jj]$

I3 Q is a correspondence from $S[0..i]$ to $T[0..j]$

The main loop processes one element of S per iteration, which may involve advancing j in T by zero or more steps. It constructs a correspondence in which high steps of T are "deferred" until forced by an impending low step in S. Note that, owing to trimness of σ and τ, and minimality of S, T, the invariants imply that $i < \mathit{last}(S)$ iff $j < \mathit{last}(T)$ and thus upon termination we have $i = \mathit{last}(S)$ and $j = \mathit{last}(T)$. The program is

`While` $i < last(S)$ `do`

- if $actc(S_i) : \text{L}$ and is an action (assignment or declassification) then do
  $Q, i, ii, j, jj := Q \cup \{(i{+}1, j{+}1)\}, i{+}1, j{+}1, ii{+}1, jj{+}1$

  Before updating i, j we have $code(S_i) = code(T_j)$ by (code agree L), so $code(S_{i+1}) = code(T_{j+1})$; hence if the latter are high they have the same L-continuation. By $\sigma \sim \tau$ and I2, we get (state agreement) for S_{i+1} and T_{j+1} (note that we do not get this by typing, since the step could be a declassification).

- else if $actc(S_i) : \text{L}$ but is not an action then
  $Q, i, j := Q \cup \{(i{+}1, j{+}1)\}, i{+}1, j{+}1$

  Again, before updating i, j we have $code(S_i) = code(T_j)$ by (code agree L) and establish (code agree L/H) in the new pair. And the states in the new pair are indistinguishable since they were not changed.

- else if $actc(S_i) : \text{H}$ and is an assignment, and $actc(S_{i+1}) : \text{H}$, then by typing (Lemma 6.3) S_i assigns a high variable so we have $state(S_{i+1}) \sim state(S_i) \sim state(T_j)$. The L-continuations of $code(S_i)$, $code(S_{i+1})$, and $code(T_j)$ are the same (given that $actc(S_i)$ and its successor $code(S_{i+1})$ are high). This step appears in σ, so we do
  $Q, i, ii := Q \cup \{(i{+}1, j)\}, i{+}1, ii{+}1$

- else if $actc(S_i) : \text{H}$ but is not an assignment, and $actc(S_{i+1}) : \text{H}$, the situation is like in the preceding case except that a non-assignment step is not recorded in σ, so we do
  $Q, i := Q \cup \{(i{+}1, j)\}, i{+}1$

- otherwise we have $actc(S_i) : \text{H}$ and $actc(S_{i+1}) : \text{L}$. So S_{i+1} should correspond to the next low configuration in T. This exists, because S is minimal to generate σ, and σ is trim, so there must be a low assignment at S_{i+1} or later. Hence by $\sigma \sim \tau$ there is a corresponding assignment at T_{j+1} or at some later T_{j+k}. Intervening high configurations $T_{j+1}, \ldots, T_{j+k-1}$, if any, must also get put in Q for (completeness). Since each is typed high, its state is indistinguishable from T_j and has the same L-continuation. So we do
  `while` $actc(T_{j+1}) : \text{H}$ `do`
  `if` $actc(T_{j+1})$ is assignment then $jj := jj{+}1$ fi;
  $Q, j := Q \cup \{(i, j{+}1)\}, j{+}1$ `od`
  $Q, i, j := Q \cup \{(i{+}1, j{+}1)\}, i{+}1, j{+}1$
  The invariants are re-established following this loop (not during it), because $actc(S_i)$ is not branching: it is high and followed by a low command, which cannot happen thanks to typing.

The main loop terminates because every iteration increases i. Upon termination invariants $I3$ and $I0$ yield that Q is a correspondence from S to T.

It is straightforward to check that invariants I0 and I1 are maintained. For I2, note that when j is at an assignment, jj is incremented to record the state. (Note that $S[0..i]$ is not always minimal for $\sigma[0..ii]$, and this is not needed.)

I3 is the hard one. For (completeness), note that a pair is added to Q for every value of i and for every value of j. For (monotonicity) note that pairs are added in order with increasing i or j. About the other correspondence conditions, see the remarks made within the program above.

Proof of Theorem 7.5. Suppose M is statically secure and trace σ is generated by a pre-run, S, and that $S_{last} = \langle\, C,\ t\,\rangle$. And suppose $\langle\, C,\ t\,\rangle \rightarrow \langle\, D,\ u\,\rangle$. Items 1 and 2 in Def. 5.7 give two cases to check; in any other case there is nothing to prove. The proof for item 1 is intricate so we do the other case first.

Case $actc(\langle\, C,\ t\,\rangle)$ is **declass** $\iota\,\langle B\rangle$. We must show there is some flowspec for ι, with precondition $P\&\varphi$ such that $t \models P$ and $\mathcal{K}(\sigma\, u) \supseteq \mathcal{R}(\sigma, P\&\varphi, \iota)$. By precondition coverage (Def. 6.2) there is at least one flowspec for ι, and the disjunction of the state predicates of the flowspecs for ι is a valid pre-assertion for the statement, i.e., it holds in t. Choose any **flow** ι **pre** $P\&\varphi$ **post** φ' such that $t \models P$. Consider any $r \in \mathcal{R}(\sigma, P\&\varphi, \iota)$. We must show $r \in \mathcal{K}(\sigma\, u)$. By definition of $\mathcal{R}$ there is some pre-run T from r such that $\sigma \sim trace(T)$ and $(\sigma_{last}, state(T_{last})) \models P\&\varphi$ and $actc(T_{last})$ is **declass** $\iota\,\langle B\rangle$. By relational correctness (Def.6.2), B terminates from $state(T_{last})$ in some state, say q. Moreover, φ' includes agreements for all low variables assigned by B, by postcondition coverage (Def. 4.1). By relational correctness for B with respect to this flowspec (Def 6.2), from $(\sigma_{last}, state(T_{last})) \models P\&\varphi$ we get that $(u, q) \models \varphi'$ and hence from $\sigma_{last} \sim T_{last}$ and the fact that the modified variables agree according to φ' we get that $u \sim q$. For brevity, let $\tau = trace(T)$. Because declassifications get included in traces, $\tau\, q$ is a trace, indeed $\tau\, q \in Traces(r)$. From above we have $\sigma \sim \tau$ and $u \sim q$, whence $\sigma\, u \sim \tau\, q$. Thus r is in $\mathcal{K}(\sigma\, u)$.

Case $actc(\langle\, C,\ t\,\rangle)$ is an assignment to a low variable or $C =$ **stop**. Then CGR requires $\mathcal{K}(\sigma\, u) \supseteq \mathcal{K}(\sigma)$. Consider any $r \in \mathcal{K}(\sigma)$, to show $r \in \mathcal{K}(\sigma\, u)$.

By $r \in \mathcal{K}(\sigma)$ there is some $\tau \in Traces(r)$; choose one that is trim. Let ii be the unique integer such that $trim(\sigma) = \sigma[0..ii]$. Note that $\sigma[0..ii] \sim \sigma \sim \tau$. For the given pre-run S that generates σ, let i be such that $S[0..i]$ is the minimal pre-run that generates $\sigma[0..ii]$. Let T be the minimal pre-run that generates τ. Now we have a situation to which Lemma 7.4 is applicable; it yields correspondence Q from $S[0..i]$ to T. Let $j := last(T)$.

We wrote "$j := last(T)$" as an assignment, to foreshadow what comes next. To complete the proof, we need to extend τ with a state that corresponds to u — and this will be constructed using a program working on variables i, j, ii, jj, Q, T, initialized in the previous paragraph. Note that S is given whereas T needs to be extended far enough to reach a configuration that corresponds to $\langle\, C,\ t\,\rangle$, from which a step can be taken to match $\langle\, C,\ t\,\rangle \rightarrow \langle\, D,\ u\,\rangle$.

The main loop maintains the following invariants:

J0 $0 \le i \le last(S)$ and $0 \le ii \le last(\sigma)$ and $0 \le j = last(T)$ and $0 \le jj = last(\tau)$

J1 $i\,Q\,j$

J2 $S[0..i]$ is a generating pre-run for $\sigma[0..ii]$ and T is a generating pre-run for τ

J3 Q is a correspondence from $S[0..i]$ to T

These are very similar to the invariants I0-I3 in the proof of Lemma 7.4. But here we are constructing T and τ, so j and jj are the upper limits of T and τ (see J0 versus I0) and thus J2 and J3 refer to T since $T = T[0..j]$ is invariant.

Because $\sigma\, u$ is a trace and is generated by S, the steps from S_i to $S_{last(S)}$, i.e., to $\langle\, C,\ t\,\rangle$, are not low actions; the next low action is the step $\langle\, C,\ t\,\rangle \to \langle\, D,\ u\,\rangle$. The main loop extends T to match S; following the loop we match the step to $\langle\, D,\ u\,\rangle$. (The justification that the invariants are preserved is similar to that in the proof of the Lemma so we err on the side of brevity here.)

`while` $i < last(S)$ `do`

- if $actc(S_i) : \text{L}$ (by the preceding, it is not an action) then $code(T_j) = code(S_i)$ and we let A, p be given by $T_j \to \langle\, A,\ p\,\rangle$ in
  $T, Q, i, j := T\langle\, A,\ p\,\rangle,\ Q \cup \{(i{+}1, j{+}1)\},\ i{+}1, j{+}1$

  Note that $code(S_j) \neq \textbf{stop}$ since we are given $\langle\, C,\ t\,\rangle \to \langle\, D,\ u\,\rangle$, so the successor to T_j exists.

- else if $actc(S_i) : \text{H}$ and $actc(S_{j+1}) : \text{H}$ then by J1 and J3 we have $Lcont(code(S_i)) = Lcont(code(T_j))$ and being typed high the step from S_i does not write low. If $actc(S_i)$ is an assignment then do $ii := ii{+}1$, and in any case do
  $Q, i := Q \cup \{(i{+}1, j)\},\ i{+}1$

- else if $actc(S_i) : \text{H}$ and $actc(S_{j+1}) : \text{L}$ then we have to extend T to catch up, i.e., do its next high steps. First set A, p according to $T_j \to \langle\, A,\ p\,\rangle$ and set $T := T\langle\, A,\ p\,\rangle$, following which we have $last(T) = j{+}1$. Then do
  `while` $actc(T_{j+1}) : \text{H}$ `do`
  `if` $actc(T_{j+1})$ is an assignment (necessarily high)
  `then` $\tau := \tau\, state(T_{j+1})$ `fi`;
  $T, Q, j := T\langle\, A,\ p\,\rangle,\ Q \cup \{(i, j{+}1)\},\ j{+}1$
  where $T_j \to \langle\, A,\ p\,\rangle$ `od`;
  $Q, i, j := Q \cup \{(i{+}1, j{+}1)\},\ i{+}1, j{+}1$

  TBD Try to rewrite so inner loop maintains main invariants, since it's confusing to see T_{j+1}

  Following this code the invariants J0–J3 are restored. This inner loop terminates because, by typing Lemma 6.3, high code never diverges (see Section 6).

The outer loop terminates because every iteration increases i. Upon termination, $i = last(S)$. Because the step from $\langle\, C,\ t\,\rangle$ assigns low or terminates, we have $actc(C) : \text{L}$ by typing. Since $i Q j$, the invariants yield that $T_j = \langle\, C,\ t'\,\rangle$ for some t' such that $t \sim t'$. The next step is $\langle\, C,\ t'\,\rangle \to \langle\, D,\ u'\,\rangle$ for some u' with $u \sim u'$ (because by typing Lemma 6.3

the active command in C sends indistinguishable t, t' to indistinguishable u, u'). Hence $\sigma\ u \sim \tau\ q$ and we have the witness of $r \in \mathcal{K}(\sigma\ u)$.

Proof of Lemma 5.9. By mutual implication. Suppose $\mathcal{K}(\sigma) = \mathcal{K}(\sigma_0)$ for all σ. To prove the condition above, consider any s, t, σ, C' such that $s \sim t$ and $\sigma \in \mathit{Traces}(s)$. By definition of $\mathcal{K}$ we have $t \in \mathcal{K}(s)$ (writing s as a singleton trace), so since σ is a trace from s we have $t \in \mathcal{K}(\sigma_0)$. Now $t \in \mathcal{K}(\sigma_0)$ iff $t \in \mathcal{K}(\sigma)$ by assumption, so by definition of $\mathcal{K}(\sigma)$ there is some τ such that $\tau \sim \sigma$.

For the converse, it suffices to prove for all σ that $\mathcal{K}(\sigma_0) \subseteq \mathcal{K}(\sigma)$ since the reverse holds by monotonicity of knowledge. Consider any $t \in \mathcal{K}(\sigma_0)$. We must show $t \in \mathcal{K}(\sigma)$. By definition, $t \in \mathcal{K}(\sigma_0)$ means $\sigma_0 \sim t$ (a property of $\sim$ on singleton traces). Now in the condition above take $s := \sigma_0$. Since we have trace σ from σ_0 which is low-equivalent to t, there must be some $\tau \in \mathit{Traces}(t)$ with $\tau \sim \sigma$. Hence $t \in \mathcal{K}(\sigma)$.