Expressive Declassification Policies and Modular Static Enforcement

Anindya Banerjee
Kansas State University and Microsoft Research
Manhattan, KS and Redmond, WA, USA
Email: ab@cis.ksu.edu

David A. Naumann Stan Rosenberg
Stevens Institute of Technology
Hoboken, NJ, USA
Email: {naumann, srosenbe}@cs.stevens.edu

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, static enforcement is provided by combining type-checking with program verification techniques applied to the small subprograms that carry out declassifications. Enforcement is proved sound for a simple programming language and the extension to object-oriented programs is described.*

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 access policies can express 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. Our attack model ignores covert channels.

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

The first author was partially supported by NSF awards CNS-0627748, CCR-0296182, ITR-0326577.
The second and third authors were partially supported by NSF awards CNS-0627338, CRI-0708330, CCF-0429894.

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 [29], 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 [11] who proposed confidentiality policies that include conditions under which declassification is allowed. The *second contribution* is an end-to-end semantic property extending the *gradual release* property recently introduced by Askarov and Sabelfeld [4]. Ours encompasses conditioned policies like those of Chong and Myers [11] but with stronger security guarantees.[1] Briefly, the security property pertains to attackers that know the code, have unbounded deductive power, and see intermediate low states. The property says that low observers' 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 that combines the simple type-checking of [4] with verification of program assertions and relational program verification [1], [8]. Relational verification is applied only to the small subprograms that carry out declassifications. Type checking is compositional. Static verification for assertions can be modular at the granularity of procedures: a procedure body is checked using only the specifications for procedures it calls (e.g., [7]). We prove soundness of the enforcement régime, relative to a sound verifier and typechecker.

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 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 the following benefits of the approach:

(a) Policies are expressed using two commonplace means: ordinary labeling of variables with security levels, together with ordinary program assertions, 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.

(b) 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.

(c) Our security property constrains the flow of information even *after* one or more release actions (unlike [11] but like [4]). It is defined in terms of a standard, un-instrumented program semantics. It reduces to termination-sensitive noninterference in the absence of declassification and accords with the other prudent principles of Sabelfeld and Sands [29].

(d) In many systems, sensitive data is manipulated in pointer-based shared data structures rather than named program variables; this poses challenges for security labeling and for static analysis, due to aliasing etc. Our use of program logic fits with the solution of Amtoft et al [1], [2], which provides effective, modular verification for object-oriented programs. We can specify and verify the secure release of an unbounded data structure by a single pointer assignment, rather than by cloning as in previous work [3].

Our enforcement régime integrates, in a straightforward way, three existing technologies: security typing (e.g. [37]), relational verification (e.g. [1], [8]), and ordinary program verification for assertions (implemented in many tools). Our main theorem says that the régime 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 (e.g., [6]) and logics (e.g., [1], [22]) 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, security property, and enforcement régime. Sect. 3 formalizes a simple programming language with declass commands. Sect. 4 defines our policies (dubbed *flowspecs*). Sect. 5 defines our end-to-end security property in terms of the program semantics. Sect. 6 addressess enforcement by type checking together with verification conditions. Sect. 7 gives the

1. Also stronger than [4] where the attacker learns nothing from divergent computations —which is dubious, given that the attacker can see intermediate steps.

main technical result, that statically checked programs are secure. Sect. 8 sketches the extension to programs using shared, dynamically allocated objects. Sect. 9 covers related work and Sect. 10 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 [37]. 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 $\mathbf{if}\ h\ \mathbf{then}\ l := true$.

An alternative approach for enforcing noninterference is to formulate security as a verification problem and use program logic [13], [15]. A basic form of noninterference says 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" [8]. We focus on the logic of Amtoft et al. [1] which addresses a key challenge for reasoning: mutable data structures 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. The interpretation is with respect to *two* executions of C, one from each of the initial pair of states. Pre- and postconditions can include predicates of a special form, which we call an *agreement* and write as $\mathbf{A}(e)$. The meaning is that the two considered states agree on the value of expression e. Agreements can also involve *region* expressions which abstract the heap, as we discuss later. 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 no-read-up and no-write-down constraints.

In a precondition, an agreement expresses what is considered low-visible. So the logical formulation of noninterference can be used in a natural way to describe the *delimited release* policies of Sabelfeld and Myers [28]. They consider this example, akin to an electronic wallet scenario, with l, k : L and h : H:

$$\mathbf{if}\ h \geq k\ \mathbf{then}\ h := h - k; l := l + k \tag{1}$$

To express the policy that it is fine to reveal whether $h \geq k$, but nothing more about h, our specification takes the following form, which we call a *flowspec*:

$$\mathbf{flow\ pre}\ P \& \varphi\ \mathbf{mod}\ l \tag{2}$$

In general, the precondition combines the two-state predicate φ with an ordinary state ordinary state predicate P that is to hold in both initial states. For this example, the state predicate P is simply $true$, and the agreement φ is $\mathbf{A}(h \geq k) \wedge \mathbf{A}(l) \wedge \mathbf{A}(k)$. 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. That is, the flowtriple $\{\varphi\}\ C\ \{\mathbf{A}(l)\}$ is valid. Moreover, the only low variable that is updated is l.

In the analysis by Sabelfeld and Sands [29], delimited release specifies *what* is released. To specify *where* in the system release is allowed, we follow previous work in explicitly marking commands that are allowed to declassify. To specify *when* release may happen, the flowspecs attached to our declassification commands include a state predicate P as in (2). We now proceed to an example that clarifies the utility of P.

Release after multiple events. Our leading example below 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 just record saved test info, whereas 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.

The example is illustrative; we ignore other 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} diag; String{H} notes; }
```

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

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

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

```
Object release(Database db, int patID,
   Bookkeeper b, InsRep r)
// precondition: 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.diag := pr.diag;
    return ir;
  } else  return new Msg("unavailable");
}
```

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 to `ir.diag`. We designate the assignment as a declassification, exempt from typechecking but subject to a flowspec of the form (2) where the state predicate P in the precondition is:

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

The agreement part of the precondition, i.e., φ in (2), is $\mathbf{A}(pr.diag)$ —this expresses "what" is released. The presence of flowspecs indicate the program points "where" release occurs. Predicate P expresses "when" the release happens. The conditions represent a sequence of requisite events: $db.recent(pr)$ expresses that pr is the most recent patient record; $sys.auth(b, \text{'book'})$ says that b is authenticated by sys as bookkeeper; etc.

Our enforcement régime accepts the program. One ingredient of enforcement is that the rest of the program typechecks, essentially by the rules in [4] which disallow declassification inside a high conditional (in addition to the usual rules that prevent direct and implicit flows). Another ingredient is that the flowtriple

$$\{P \& \varphi\}\ ir.diag := pr.diag\ \{\mathbf{A}(ir.diag)\}$$

is valid; indeed, the stronger flowtriple $\{\varphi\}\ ir.diag := pr.diag\ \{\mathbf{A}(ir.diag)\}$ can be proved in the logic of [1] or using self-composition with an off-the-shelf automated verifier [26] (both deal with mutable heap objects). The last ingredient is that P is a valid pre-assertion, i.e., it holds on all paths to the declassification, as we now argue in detail. The conjunct $pr.committed$ holds owing to the guard condition of the **if**. The conjuncts $sys.auth(b, \text{'book'})$ and $sys.auth(r, \text{'rep'})$ are preconditions to the method — its calls must therefore be verified for these conditions. Recency should be ensured by 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 $append$.

Conditioned gradual release. By itself, checking of security-labelled types should enforce noninterference. (A practical checker of this kind is Jif [24]; others have been formally validated [25], [32].) But we exempt declassification commands from type checking! Instead, each declassification is required to 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 [4]. An observer at level L sees each *low action* —assignment to a low variable, declassification step, or termination. Gradual release says that the observer gains no knowledge about the initial value of secret variables except from declassifications. We require in addition that what is learned about initial secrets 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). Moreover, 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, conditioned gradual release amounts to noninterference: knowledge remains constant through every step of a computation.

Fine points. To achieve modular enforcement using off-the-shelf tools, we define our security property to interpret flowspec preconditions in terms of the *current value* of expressions e that occur in agreements $\mathbf{A}(e)$. This poses a risk of laundering: If $h0$ and $h1$ are high variables, a policy with precondition $\mathbf{A}(h0)$ would appear to allow the release of $h0$, but declassification of $h0$ subsequent to an assignment $h0 := h1$ would

actually release $h1$. The solution to this known problem is to disallow reassignment of high variables prior to their use in declassifications [28].

It is not really practical to disallow all updates of high variables prior to their use in declassifications. For example, one would like to make multiple uses of an electronic wallet, each time decreasing the balance, though of course if the user allows many such transactions the entire balance could be revealed. What is needed is means to designate sessions or transactions, so a declassification policy can refer to the initial value of a high variable *within a session*, and disallow its update prior to release. So too, the security property would refer to secrets at session start, not the system's initial state. Our insistence to use ordinary assertions rather than exotic syntax may pay off here, since session boundaries are typically embodied in program control and data state, and the notion of "session" may be policy-specific. For example, a session for a login password is bracketed by uses of the *passwd* program. Thorough treatment of this issue is left to future work.

Another feature of our technique is that a flowspec could allow modification of several low variables (or heap locations) in a single declassification. However, in order to adhere to the principle of non-occlusion,[2] a declassification step should be atomic as viewed by the low observer (e.g., by use of locks). To avoid distraction, our formalization achieves atomicity by restricting declassification to single assignments.[3]

Finally, the semantic formalization of our security property effectively treats a flowspec precondition $P\&\varphi$ as licensing the release of not only what is explicitly mentioned in φ but also any high information in the control state. To ensure that the policy $P\&\varphi$ is meaningful in isolation, we choose to disallow any declassification in the context of a high branch condition. This we do in the enforcement régime.[4]

3. Programming language

This section formalizes the simple imperative language over integer variables, augmented with the so-called *declass* command. To focus on the key ideas in a comprehensible way, we refrain from considering pointers, procedures, or other language features. But a number of technicalities are needed to formalize the security property (Sect. 5).

2. Adding declassification cannot make an insecure program secure [29].

3. This is already quite expressive in conjunction with data structures (Sect. 8) or encryption keys [4], but it precludes wrapping example (1) inside a declassification.

4. As it happens the security condition is well-defined even for programs, such as example (6), with declassification under high guard. Little would be gained by complicating the security property on this account.

$$C, B, M ::= \mathbf{declass}^{\,\iota}\,\langle x := e\rangle \mid x := e \mid \mathbf{skip} \mid C; C \mid \mathbf{if}\ e\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{while}\ e\ \mathbf{do}\ C$$
$$e ::= x \mid 0 \mid 1 \mid \ldots \mid e + e \mid e \leq e \mid \ldots$$

Figure 1. Grammar of commands and expressions; ι ranges over declass identifiers.

$$\langle\, \mathbf{stop},\ s\,\rangle \to \langle\, \mathbf{stop},\ \checkmark\,\rangle \qquad \langle\, \mathbf{skip},\ s\,\rangle \to \langle\, \mathbf{stop},\ s\,\rangle$$

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

$$\langle\, \mathbf{declass}^{\,\iota}\,\langle x := e\rangle,\ s\,\rangle \to \langle\, \mathbf{stop},\ s[x := [\![e]\!](s)]\,\rangle$$

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

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

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

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

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

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

Figure 2. Semantics; s, s' range over non-$\checkmark$ states.

The command "$\mathbf{declass}^{\,\iota}\,\langle x := e\rangle$" behaves as $x := e$. Its syntax includes an identifier, ι, used later to refer to an associated flowspec. A well-formed program has a different identifier ι for each declassification.

Figure 2 defines the semantics. We write $[\![e]\!](s)$ for the value of expression e in state s. 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, $\to$, over *configurations* of the form $\langle\, C,\ s\,\rangle$ where s is a state and C is either a command or $\mathbf{stop}$. The latter triggers an observable step to the *improper state*, $\checkmark$, for termination.

Every command can be written in the form $C_0; C_1$ or else 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 (Fig. 2). 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 $state(\langle\ C,\ t\ \rangle) = t$.

An initial configuration $\langle\ C,\ s\ \rangle$ determines a unique finite or infinite *run*, that is, the maximal sequence of configurations given by the transition relation, starting with $\langle\ C,\ s\ \rangle$. 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., the step to ✓). The other transitions, e.g., those for **if** and **while**, never change the state. It is convenient to work with a notion of trace which extracts from a pre-run the series of states resulting from actions.

Let M be a fixed command, the main program. We define several notions based on runs of M, leaving M implicit in the notation. For any pre-run S of M, let $trace(S)$ be the sequence of states starting with $state(S_0)$ and thereafter including every state that results from an action. For any state s, let $Traces(s) = \{trace(S) \mid S \text{ is a pre-run from } s\}$. Define $TRACES = \cup s \cdot Traces(s)$. Note that declassification steps are not marked as such in traces.

We say S is a *generating pre-run for* σ, if $trace(S) = \sigma$. 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 the initial state, 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 $last(\sigma)$ for $len(\sigma) - 1$ and abbreviate $\sigma_{last(\sigma)}$ as σ_{last}.

4. Policy specification

In this section we formalize our notion of policy specification. This is intended as a foundation for concrete policy languages, so we begin with some discussion of desiderata for policy specifications.

A "where" policy [29] designates where in the code declassification is allowed. This goes against common wisdom that policies should be separate from implementations. If the intention of a "where" policy is to restrict declassification to some program components that have been subject to security audits, or the code is in high-integrity storage, then "where in the code" is at the granularity of, say, a load module. Arguably that is somewhat separate from the implementation. On the other hand, extant "where" policy formulations are fine grained, e.g., individual assignments are marked as declassifiers (e.g., [4], [24]). This is fragile, as implementations often change, and it raises the question of an independent meaning for the policy.

Judging by the examples in the literature, the rationale for fine-grained "where" policies is pragmatic: the specifier may choose as declassifiers some assignments that appear to conform to some (informal) "what" or "when" policy, based on the specifier's understanding of the code. In this paper we provide direct means to specify "what" and "when" policies. So we formulate policy in a way that caters for separating it from the code, yet still enables fine-grained "where" specification if desired.

Definition 4.1 (policy): A *baseline security policy* for a program M is a mapping, Γ, from the variables of M to the security levels $\{\mathrm{L}, \mathrm{H}\}$. A *declassification policy* is a set, Φ, of flowspecs, of the form $\mathbf{flow\ pre}\ P\&\varphi\ \mathbf{mod}\ x$ where

- φ is a conjunction of agreements, $\mathbf{A}(e_0)\wedge\mathbf{A}(e_1)\wedge\ldots\mathbf{A}(e_k)$ where the e_i are expressions;
- P is a formula over the program variables;
- the "modifiable" variable x is L according to the baseline policy Γ.

For the language of Sect. 3, P can be a first-order formula with atomic predicates for integer arithmetic. Technically, all we need is that the semantics is two-valued, i.e., the satisfaction relation $s \models P$ means that P is true in state s and otherwise P is false. For the richer programming language discussed in Sect. 8, formulas would be as in JML and similar specification languages [7], [19], and the "modifies" clause could designate a heap location or region.

Agreement formulas $P\&\varphi$ are interpreted in a pair of (non-✓) states. For φ of the form $\mathbf{A}(e_0)\wedge\ldots\mathbf{A}(e_k)$, define $(s,t) \models P\&\varphi$ iff

$$(3) \qquad \begin{cases} s \models P \text{ and } t \models P \text{ and} \\ [\![e_i]\!](s) = [\![e_i]\!](t) \text{ for } 0 \le i \le k \end{cases}$$

As explained in Sect. 2 and formalized in Sect. 5, we interpret the policy to mean that information flows in accord with the baseline policy Γ, except that each declass may have additional flows if justified by some flowspec in Φ. One might say that the declass commands in the code are expressing the "where" part of the policy. To see why we choose not to refer to them as part of the policy, let us reconsider the issue of separating code from policy.

Labeling of variables is somewhat separate from the code that acts on them. (Only external interfaces need be labelled; the rest, e.g., local variables, fields, methods, can be inferred [18], [24], [31], [33].) One can imagine "what" 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]. Several works explore type labeling for declassification (e.g., [10], [11], [20]).

So long as P and φ refer only to global variables and to x, one can read **flow pre** $P\&\varphi$ **mod** x as a schematic specification of "what" and "when" policy, taking x to be a placeholder for any variable. Instead of assuming that the code has marked declass commands, we could let the type-checker add a declass for each assignment that violates the baseline policy and is not in the scope of a high branch. (The latter is needed in order to maintain the interpretation of a flowspec precondition as completely specifying what is released —recall the last "fine point" in Sect. 2.)

Many interesting policies can be expressed using schematic flowspecs. For the example in Sect. 2, we can take ir and pr to be schematic variables so that the policy is applicable to any assignment of a `PatientRecord`'s diagnosis to an `InsRecord`. The preconditions refer to fields of these objects and to global data structures (the log and the authentication system).

To cater for schematic use of flowspecs, our formalization of policy (Def. 4.1) does not map declass identifiers to flowspecs in Φ. Instead, the security property (Def. 5.5) and the enforcement régime (Def. 6.2) both require that such mappings exist. In practical use of the schematic approach, the mapping would be created when the type-checker marks the code with declass commands.

However, it is straightforward to adapt our formulations (Defs. 5.5 and 6.2) to consider the mapping to be part of policy, in order to fully capture fine-grained "where" policies with associated "what" and "when" policies.[5]

5. The end-to-end security property

In this section, let M be a fixed program, with policy Γ, Φ as in the previous section. To lighten various notations we suppress their dependence on M, Γ, and Φ; for example, σ and τ range over the set $TRACES$ of M. This section defines the semantics of the policy.

5. In fact this can be encoded in the present formalulation. Suppose we wish to associate the flowspec **flow pre** $P\&\varphi$ **mod** x with a single declass, say $\mathbf{declass}^{\,\iota}\,\langle x := e\rangle$. Add to the program a fresh variable v initialized to 0. Replace the declass by the sequence $v := 1; \mathbf{declass}^{\,\iota}\,\langle x := e\rangle; v := 0$. Revise the flowspec precondition to be $(P \wedge v = 1)\&\varphi$. The revised flowspec licenses no declassifications other than the one labelled ι.

Low observations. The gradual release paper [4] defines knowledge directly in terms of low observations, i.e., sequences of the low-visible parts of states. Our definition is formulated in terms of traces of complete states, since these 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 of indistinguishability and observed knowledge are somewhat involved and are carefully designed to facilitate proof of the soundness theorem, in a way that can be extended to richer languages.

As in other works [27], [35], our notion of indistinguishability is defined in terms of a purging function to eliminate timing channels from the model. Assignment to a high variable is called a *high action*; the other actions —termination and low assignments including declassifications— are low actions. For any pre-run S, let $p\text{-}trace(S)$ be the same as $trace(S)$ except omitting states that result from high actions, i.e., assignments to variables x with $\Gamma(x) = \text{H}$. Define $purge(\sigma)$ to be the p-trace determined by a generating pre-run for σ. Note that all generating pre-runs for σ yield the same p-trace.

Define $lowvis(s)$ be the restriction of a (proper) state s to its low variables (according to Γ), and define $lowvis(\checkmark) = \checkmark$. Two traces are considered indistinguishable if there is a one-to-one correspondence between the states resulting from low actions and moreover corresponding states are low-equivalent.

Definition 5.1 (indistinguishable ($\sim$)): Define $\sigma \sim \tau$ if and only if $lowvis(purge(\sigma)) = lowvis(purge(\tau))$, where we map $lowvis$ over each state in the sequence.

Indistinguishability for singleton traces is the same as low equivalence, i.e., $s \sim t$ iff $lowvis(s) = lowvis(t)$, because there is no stuttering to remove. Note that if $\sigma \sim \tau$ then $\sigma_{last(\sigma)} = \checkmark$ iff $\tau_{last(\tau)} = \checkmark$.

Definition 5.2 (observed knowledge): Define $\mathcal{K}(\sigma)$ by $\mathcal{K}(\sigma) = \{s \mid \exists \tau \in Traces(s) \cdot \sigma \sim \tau\}$.

An observer, seeing the low part of σ, knows that the initial state is one of the elements of $\mathcal{K}(\sigma)$ but is ignorant of which it is. The condition $\tau \in Traces(s)$ reflects a feature of the attacker model, namely that the low observer knows the complete text of program M and its semantics.

Proposition 5.3: Knowledge is monotonic: $\mathcal{K}(\sigma\, t) \subseteq \mathcal{K}(\sigma)$ for any σ and t such that $\sigma\, t \in TRACES$. Here t may be either $\checkmark$ or a proper state.

Revelation. The connection between flowspecs and traces rests on the following notion of *revealed knowledge*, 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 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 $\mathbf{declass}^{\,\iota}\,\langle B\rangle$. Informally, $\mathcal{R}(\sigma,\, P\&\varphi,\, \iota)$ represents the set of initial states s from which there is a trace $\tau \in \mathit{Traces}(s)$ 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 unwise, 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.

Definition 5.4 (revealed knowledge, $\mathcal{R}$): For state predicate P, agreement formula φ, declass identifier ι, and $\sigma \in \mathit{TRACES}$, define $\mathcal{R}(\sigma, P\&\varphi, \iota)$ to be the set

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

Because each declass has a unique identifier, ι determines the body B of $\mathbf{declass}^{\,\iota}\,\langle B\rangle$. The condition "$S$ is a pre-run" reflects that the attacker knows the program text.

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 5.5 (CGR, conditioned gradual release): The program M under consideration satisfies *conditioned gradual release* for policy (Γ, Φ), 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 $\langle\, C,\ t\,\rangle \rightarrow \langle\, D,\ u\,\rangle$ then

1) if the active command in C is $\mathbf{stop}$ or an assignment to some variable x with $\Gamma(x) = \textsc{l}$ then $\mathcal{K}(\sigma\, u) \supseteq \mathcal{K}(\sigma)$
2) if the active command in C is $\mathbf{declass}^{\,\iota}\,\langle x := e\rangle$ then there is some $(\mathbf{flow\ pre}\ P\&\varphi\ \mathbf{mod}\ x)$ in Φ such that (a) $t \models P$ and (b) $\mathcal{R}(\sigma, P\&\varphi, \iota) \subseteq \mathcal{K}(\sigma\, u)$

Note that conditions are only imposed on termination steps, low assignments, and declassifications. In a declassification step (item 2), u and t are identical on all low variables except possibly x (by semantics). Item 2(a) expresses that release only happens under designated conditions. Item 2(b) captures the delimited release constraint. (A minor variant, for which we miss practical motivation, would also allow a declass step under item 1.)

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 2(b) 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. For brevity, let us write "$\mathbf{declass}\ \varphi\ \langle C\rangle\ \mathbf{A}(x)$" to abbreviate $\mathbf{declass}^{\,\iota}\,\langle C\rangle$ tied to an evident flowspec, $\mathbf{flow\ pre}\ \mathit{true}\&\varphi\ \mathbf{mod}\ x$. Let variables $l, l0, \ldots$ be low and $h, h0, \ldots$ high.

The first example (from [4]) shows how knowledge increases over time. The program satisfies CGR. (We have worked out the other programs in [4, Sect. 2] and our results conform to theirs).

$$\begin{array}{l}\mathbf{declass}\ \mathbf{A}(h \neq 0)\ \langle l := (h \neq 0)\rangle\ \mathbf{A}(l);\\ \mathbf{if}\ l\ \mathbf{then}\ \mathbf{declass}\ \mathbf{A}(h1)\ \langle l1 := h1\rangle\ \mathbf{A}(l1)\end{array}$$

The next example also satisfies CGR.

$$(4)\quad \begin{array}{l}\mathbf{declass}\ \mathbf{A}(h \geq 0)\ \langle l0 := (h \geq 0)\rangle\ \mathbf{A}(l0);\\ \mathbf{declass}\ \mathbf{A}(h \leq 0)\ \langle l1 := (h \leq 0)\rangle\ \mathbf{A}(l1)\end{array}$$

Next, we consider a program without declassification but with looping. This program violates CGR but is secure according to gradual release [4] which is said to be termination-insensitive.

$$(5)\quad \begin{array}{l} l := \mathbf{true};\\ \mathbf{if}\ h\ \mathbf{then}\ l := \mathbf{false}\ \mathbf{else}\ C\end{array}$$

where C is $\mathbf{while\ true\ do\ skip}$.

Upon observing termination, the attacker learns that h is true, but this is already known initially according to [4] in which divergent runs are discarded by fiat. (See their Def. 2 and their $k_{\downarrow}$.)

Our enforcement régime disallows declass in high contexts, e.g., this version of the wallet example.

$$(6)\ \mathbf{if}\ h \geq k\ \mathbf{then}\ h := h - k; \mathbf{declass}\ \langle l := l + k\rangle$$

It satisfies CGR for the intended flowspec

$$\mathbf{flow\ pre}\ \mathbf{A}(h \geq k)\ \mathbf{mod}\ l$$

but it also satisfies CGR for $\mathbf{flow\ pre}\ \mathit{true}\ \mathbf{mod}\ l$ — recall the "fine points" in Sect. 2. An improved version uses $t : \textsc{h}$ as follows:

$$(7)\quad \begin{array}{l}\mathbf{if}\ h \geq k\ \mathbf{then}\ t := k\ \mathbf{else}\ t := 0\ \mathbf{fi};\\ \mathbf{declass}\ \langle l := l + t\rangle;\ h := h - t\end{array}$$

This satisfies CGR for $\mathbf{flow\ pre}\ \mathbf{A}(h \geq k)\ \mathbf{mod}\ l$ but does not satisfy it for the trivial policy, $\mathbf{flow\ pre}\ \mathit{true}\ \mathbf{mod}\ l$, that we would like to read as allowing nothing about h to be released.

The state predicate part of a flowspec precondition is treated as an ordinary program assertion but also affects the relational (agreement) part of the specification, e.g., $\mathbf{declass}\ \langle l := l + h \rangle$ satisfies $\mathbf{flow\ pre}\ h{=}0\,\&\,\mathbf{A}(l)\ \mathbf{mod}\ l$ but not $\mathbf{flow\ pre}\ \mathbf{A}(l)\ \mathbf{mod}\ l$.

Noninterference. One of the prudent principles [29] is that the security property should reduce to noninterference in the absence of declassifications. The program M under consideration is called *noninterferent* 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.

Lemma 5.6 (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 5.7: If C has no declassifications, then it is noninterferent iff it satisfies CGR.

6. Enforcement régime

This section formalizes the static security checks, for a fixed main command M together with policy Γ, Φ.

One part of enforcement is checking validity of flowtriples (using the semantics of formulas, see (3)).

Definition 6.1 (valid): Say $\{P\&\varphi\}\ C\ \{\varphi'\}$ is *valid* iff for all states s, t, if $(s, t) \models P\&\varphi$ and $\langle\, C,\ s\,\rangle \rightarrow^* \langle\, \mathbf{stop},\ s'\,\rangle$ and $\langle\, C,\ t\,\rangle \rightarrow^* \langle\, \mathbf{stop},\ t'\,\rangle$ where s', t' are non-✓ states, then $(s', t') \models \varphi'$.

Validity is defined in the sense of partial correctness, but for our purposes we are only concerned with flowtriples for assignments, which always terminate. A provably sound logic for checking validity of flowtriples, for the simple language here, is that of Benton [8]. In fact our only triples are simple assignments, for which self-composition provides automatic verification [34] (in fact it provides automatic verification in the case of assignments of fresh objects, or field updates, as well [26]).

Another part of enforcement is type checking. Figure 3 gives straightforward typing rules to enforce the baseline policy Γ. These enforce the usual no-read-up and no-write-down conditions [37] but also disallow declassification under high branching conditions and disallow assignments to high variables mentioned in policies. We write Pvars for the set of x such that

$$\Gamma \vdash \mathbf{stop} : \mathrm{L} \qquad \Gamma \vdash \mathbf{skip} : \mathrm{H}$$

$$\frac{\Gamma \vdash e : \lambda \qquad \lambda \leq \Gamma(x) \qquad x \notin \mathit{Pvars}}{\Gamma \vdash x := e : \Gamma(x)}$$

$$\frac{\Gamma(x) = \mathrm{L}}{\Gamma \vdash \mathbf{declass}^{\,\iota}\,\langle x := e\rangle : \mathrm{L}} \qquad \frac{\Gamma \vdash C_0 : \lambda_0 \qquad \Gamma \vdash C_1 : \lambda_1}{\Gamma \vdash C_0; C_1 : \lambda_0 \sqcap \lambda_1}$$

$$\frac{\Gamma \vdash e : \mathrm{L} \qquad \Gamma \vdash C : \mathrm{L}}{\Gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ C : \mathrm{L}}$$

$$\frac{\Gamma \vdash C_0 : \lambda_0 \quad \Gamma \vdash C_1 : \lambda_1 \quad \Gamma \vdash e : \lambda \quad \lambda \leq \lambda_0 \sqcap \lambda_1}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ C_0\ \mathbf{else}\ C_1 : \lambda_0 \sqcap \lambda_1}$$

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

$\Gamma(x) = \mathrm{H}$ and x occurs in the precondition of some flowspec in Φ. (We could as well adopt the effect system of [28] to prevent prior updates of high variables mentioned in flowspecs, which would give the minor satisfaction of allowing example (7). But we prefer the more complete approach advocated under "fine points" in Sect. 2.) For expressions, the notation $\Gamma \vdash e : \lambda$ just means the highest level of a variable in e is λ. The rules define a judgement $\Gamma \vdash C : \lambda$ that says command C is secure and writes no variable of level below λ. We must also prevent unbounded computations with no observable steps (recall example (5)). We choose a simple but restrictive way [36] for simplicity: high loops are not allowed. Boudol [9] investigates more sophisticated type systems for termination and current program verification technology can automate termination checking in many cases [14].

Definition 6.2 (statically secure): We say M is *statically secure* provided there exists some mapping, *fspecs*, from declass identifiers to sets of flowspecs, such that $\mathit{fspecs}(\iota) \subseteq \Phi$ for all ι and moreover the following three conditions hold:

1) (typechecking) $\Gamma \vdash M : \lambda$ for some level λ.
2) (valid pre-assertion) For each $\mathbf{declass}^{\,\iota}\,\langle x := e\rangle$ in M, suppose the elements of $\mathit{fspecs}(\iota)$ are $\mathbf{flow\ pre}\ P_i\&\varphi_i\ \mathbf{mod}\ y_i$ for i from 0 to k. Then it is valid to assert $P_0 \vee \ldots \vee P_k$ immediately before $\mathbf{declass}^{\,\iota}\,\langle x := e\rangle$ in M.
3) (relational correctness) For each $\mathbf{declass}^{\,\iota}\,\langle x := e\rangle$ and each $(\mathbf{flow\ pre}\ P_i\&\varphi_i\ \mathbf{mod}\ y_i)$ in $\mathit{fspecs}(\iota)$, we have that y_i is x and moreover $\{P_i\&\varphi_i\}\ x := e\ \{\mathbf{A}(x)\}$ is valid.

Here is a more precise statement of item 2. Let $PP = P_0 \vee \ldots \vee P_k$. Let $\mathcal{C}[-]$ be the context in which $\mathbf{declass}^{\,\iota}\,\langle x := e\rangle$ occurs. That is, the main

program M has the form $\mathcal{C}[\mathbf{declass}^{\iota}\langle x := e\rangle]$. Then $\mathcal{C}[\mathbf{assert}\ PP; \mathbf{declass}^{\iota}\langle x := e\rangle]$ is a valid program annotation in the sense of partial correctness.

In many examples, item 2 holds trivially, but in general it may involve arbitrary assertions, e.g., isolation of a data structure, the state of an authentication system, etc. But any verification system or method that applies to Floyd-Hoare partial-correctness assertions may be used. For our main result, we simply assume that valid pre-assertion and relational correctness are checked soundly.

What we need about typing is the following.

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

Type checking and assertion checking potentially involve the whole program, but in both cases each subprogram can be checked separately, using types or specifications for other program units. The relational correctness checks are modular in a stronger sense: they apply only to declass commands.

7. Soundness

This section is devoted to proving the following.

Theorem 7.1: Suppose M is statically secure (Def. 6.2). Then it has the conditioned gradual release property (Def. 5.5).

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 and low-observable distinctions on traces. But to show soundness of the enforcement régime we need a finer analysis in terms of pre-runs.

For traces σ, τ with generating pre-runs S and T, indistinguishability can be characterized using a notion which resembles the instantiation of a simulation, matching up the low parts of S and T.

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

Definition 7.2 (correspondence): Suppose S and T are pre-runs and let $dom(S)$ be the set of indices, $\{0, \ldots, last(S)\}$, of S. A *correspondence from S to T* is a relation $Q \subseteq dom(S) \times dom(T)$ such that $0\ Q\ 0$ and for all i, j with $i\ Q\ j$ the following hold:

1) (state agreement) $state(S_i) \sim state(T_j)$
2) (level agreement) $actc(S_i) : \text{L}$ iff $actc(T_j) : \text{L}$
3) (code agreement L) $code(S_i) = code(T_j)$ if $actc(S_i) : \text{L}$
4) (code agreement H) $Lcont(code(S_i)) = Lcont(code(T_j))$ if $actc(S_i) : \text{H}$
5) (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 $i \leq i'$
6) (completeness) for every $i \in dom(S)$ there is some j with $i\ Q\ j$, and symmetrically

Lemma 7.3 (correspondence for $\sim$): Suppose the main program M typechecks. If σ, τ are generated by S, T and there is a correspondence from S to T then $\sigma \sim \tau$.

The converse is more difficult and holds less generally. Unlike the soundness Theorem, the correspondence Lemma 7.5 below does not involve liveness and the proof would go through for a type system that allows high loops.

Definition 7.4 (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 7.5 (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.

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 7.5: though similar to a proof of noninterference, it 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 of states generates a noninterferent pair of traces. At declassification steps, indistinguishability is given rather than proved. On the other hand, the proof of Theorem 7.1 has to construct the extension of a noninterferent pair of traces.

To prove the Theorem, suppose that M is statically secure, trace σ is generated by a pre-run S, and $S_{last} = \langle\ C,\ t\ \rangle$. And suppose $\langle\ C,\ t\ \rangle \rightarrow \langle\ D,\ u\ \rangle$. Items 1

and 2 in Def. 5.5 give two cases to check; in any other case there is nothing to prove. The proof for item 1 is intricate so we do item 2 first.

Case 2: $actc(\langle\, C,\ t\,\rangle)$ is $\mathbf{declass}^{\,\iota}\,\langle B\rangle$, where B is $x\ :=\ e$. We must show there is some $(\mathbf{flow\ pre}\ P\&\varphi\ \mathbf{mod}\ x)$ in $fspecs(\iota)$ such that $t \models P$ and $\mathcal{K}(\sigma\, u) \supseteq \mathcal{R}(\sigma, P\&\varphi, \iota)$. By valid pre-assertion (Def. 6.2(2) there is at least one flowspec in $fspecs(\iota)$, 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 $(\mathbf{flow\ pre}\ P\&\varphi\ \mathbf{mod}\ x)$ in $fspecs(\iota)$ 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 $\mathbf{declass}^{\,\iota}\,\langle B\rangle$. Because it is an assignment, B terminates from $state(T_{last})$ in some state, say q. By relational correctness for B with respect to this flowspec (Def 6.2(3)), from $(\sigma_{last}, state(T_{last})) \models P\&\varphi$ we get that $(u, q) \models \mathbf{A}(x)$ and hence from $\sigma_{last} \sim T_{last}$ and the fact that the modified variables agree 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 1: $actc(\langle\, C,\ t\,\rangle)$ is an assignment to a low variable or $C = \mathbf{stop}$. 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 $trim(\sigma) = \sigma[0..ii] \sim \sigma \sim \tau$. 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 7.5 is applicable; it yields correspondence Q from $S[0..i]$ to T. Let $j := last(T)$.

To complete the proof, we need to extend τ with a state that corresponds to u. This will be constructed using a program working on variables i, j, ii, 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 $0 \le j = last(T)$

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

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 $\langle\, C,\ t\,\rangle \rightarrow \langle\, D,\ u\,\rangle$. The main loop extends T to match S; following the loop we match the step to $\langle\, D,\ u\,\rangle$.

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

- if $actc(S_i)$: L (by the preceding, it is not an action) then $code(T_j) = code(S_i)$ and we let A, p be given by $T_j \rightarrow \langle\, A,\ p\,\rangle$ in
  $T, Q, i, j := T\langle\, A,\ p\,\rangle,\ Q \cup \{(i{+}1, j{+}1)\},\ i{+}1, j{+}1$
  Note that $code(S_j) \neq \mathbf{stop}$ since we are given $\langle\, C,\ t\,\rangle \rightarrow \langle\, D,\ u\,\rangle$, so the successor to T_j exists.
- else if $actc(S_i)$: H and $actc(S_{j+1})$: 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 $actc(S_i)$: H and $actc(S_{j+1})$: L then we have to extend T to catch up, i.e., do its next high steps. First set A, p according to $T_j \rightarrow \langle\, A,\ p\,\rangle$ and set $T := T\langle\, A,\ p\,\rangle$, following which we have $last(T) = j{+}1$. Then do
  `while` $actc(T_{j+1})$: 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 $T_j \rightarrow \langle\, A,\ p\,\rangle$ `od`;
  $Q, i, j := Q \cup \{(i{+}1, j{+}1)\},\ i{+}1, j{+}1$
  Following this code the invariants J0–J3 are restored. This inner loop terminates because, by typing Lemma 6.3(d), 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 $actc(C)$: 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 $\langle\, C,\ t'\,\rangle \rightarrow \langle\, D,\ u'\,\rangle$ 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 $\sigma\, u \sim \tau\, u'$ and thus $\tau\, u'$ witnesses that r is in $\mathcal{K}(\sigma\, u)$.

8. Object-oriented programs

The formal results are given for a rudimentary programming language but a key feature of both the security property and the enforcement régime is that they scale to richer languages. We sketch here the extension to dynamically allocated mutable objects, as found in Java-like languages such as Jif [24] in which security labels are associated with object fields as in our leading example (Sect. 2). Owing to the memory safety provided by strong typing and the absence of pointer arithmetic, it has been possible to develop

security type systems that provably enforce noninterference despite subtleties such as low and high aliases to objects with mutable fields and encompassing other language features such as inheritance and dynamically dispatched method calls [6], [33].

The relational Hoare logic of Amtoft et al [1] also pertains to object-oriented programs. It can express finer-grained policies in which a certain field may be treated as high for objects in one region of the heap while low in another. This presents something of a mismatch when it comes to combining typing with logic for declassification, but this can be overcome and in fact an important benefit achieved: declassification of an entire data structure in constant time and space.

Consider first an assignment $l := h$ where l, h have type $Node$, and class $Node$ has low fields $item : int$ and $nxt : Node$. Following the assignment, the low observer may see not only the value of l but also all the items reached from l, which may have been previously unreachable. Thus naïve use of our enforcement régime would be unsound for $\mathbf{declass}\ \langle l := h \rangle$ and a flowspec precondition $\mathbf{A}(h)$. Following Amtoft et al (but using our notation), a valid flowspec precondition would look like $R = h.nxt^* \& \mathbf{A}(h) \wedge \mathbf{A}(R.nxt) \wedge \mathbf{A}(R.item)$. The state predicate $R = h.nxt^*$ says that R is the region containing all the nodes of the list. The agreements say that in the two compared states, the nxt and $item$ fields agree (modulo a suitable renaming of object addresses, since allocations may differ in two program runs).

Because the underlying notion of low-indistinguishability for heaps is the same in the cited logic and type system, it is not difficult to adapt CGR to the richer language. Object allocation is treated as a form of assignment, and field update is also treated as an observable action. The enforcement régime can be carried over as well, but with an added requirement on flowspecs that they cover the reachable locations as illustrated above.

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 the doctors' notes. We use an alternate version of $PatientRecord$ given by:

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

Note that now the levels of $diag$ and $notes$ are type parameters (level polymorphism like this is available in Jif and similar security type systems [31], [33]). Patient records are linked by the $next$ field, which itself is low —for clarity we have made that label explicit. (Recall from Sect. 2 that the unmarked default is L.) We are assuming that $diag$ and $notes$ contain secret values, so a sensible declaration and (noninterferent) assignment to obtain the list of records is

```
root := db.lookupAll();
```

where `root: PatientRecord<H,H>{L}`. To obtain a list of records in which the diagnoses have been declassified, it would be possible to clone the list, iteratively performing a declassification like that in Sect. 2. In fact, such cloning is done extensively in Jif case studies [3], in order to avoid laundering attacks whereby fields are updated with high info subsequent to their declassification, via high aliases into the data structure.

If in fact there are no exploitable aliases, cloning is costly and unnecessary. We would like to release the entire list of patient records by a single assignment:

```
newRoot := root;
```

where `newRoot: PatientRecord<L, H>{L}`. This will be rejected by the typechecker owing to the types of $root$ versus $newRoot$. But it can be made a declassification, subject to a flowspec like this:

$$\begin{array}{l}\mathbf{flow\ pre}\ ISOL(root)\ \&\\ \mathbf{A}(root) \wedge \mathbf{A}(root.next^*.diag)\\ \mathbf{mod}\ newRoot,\ newRoot.next^*.diag\end{array}$$

The agreement precondition says that what is released is the $diag$ field of all the records. The state predicate $ISOL(root)$ is intended to say that the records are reachable *only* via $root$. This property is well studied in the literature on ownership for heap encapsulation [12]. For present purposes, we need transferrable ownership, which can be expressed and enforced by certain type systems and program verifiers [7], [23].

The modifies clause reflects that giving type `PatientRecord<L, H>{L}` to $newRoot$ effectively changes the type of all the records (owing to the declared type of $next$). Thus the relational correctness condition for static security, Def. 6.2(3), will specify agreement on all the reachable $diag$ fields —and this will follow from the agreement precondition.

In summary, the extension of our enforcement régime to declassification of data structures adds a requirement on flowspecs, namely the isolation precondition which must then be proved as a valid preassertion.

9. Related work

Sabelfeld and Sands [29] systematically analyze many recent proposals for declassification, noting shortcomings and anomalies which motivate the "prudent principles" we address in Sect. 10. Another notable work is by van der Meyden [35]; 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 [27]. Like ours, these works also distinguish low from high events and purge the latter as a way to remove timing leaks from consideration. But we address finer-grained policies and also enforcement for a concrete programming language.

Our work builds very directly on the gradual release paper [4], more specifically on the semantic property introduced in the first part of the paper. Our formalization is quite different, in part because we correct an evident weakness in the attacker model: by fiat, their attacker observes no steps if the computation is not going to terminate. (Of course for terminating computations their low events include termination as well as low writes.) Askarov and Sabelfeld [4] extend 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 [4] there are brief comments on combining gradual release with delimited release [28] but no hints as to how this would be done. Gradual release is proved to be enforced by a standard type system including the constraint that declass commands are low [4]. This is in accord with our result, since we use essentially the same type system and one can take every flowspec to have agreements for all secrets read by the declass.

Any logic or verification system can be used to discharge the "valid pre-assertion" proof obligation, Def. 6.2(2), —e.g., tools like ESC/Java and Spec# [7] which can reason about pointer isolation (see Sect. 8). To verify relational correctness, Def. 6.2(3), Benton's [8] relational logic suffices for the simple imperative language of Sect. 3 and is implementable by self-composition [26]. Motivated by the less conservative analysis provided by logics, as opposed to usual type systems, Amtoft et al [1] develop a relational logic for object-based programs, 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 benefit is the ability to express fine-grained flow policies as we have proposed here.

Our use of state predicates in release policy is inspired by Chong and Myers [11], 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 that label variables and 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 can express past events using specification-only history variables (indeed, relevant history is often already available in the program state). Their security property is rather weak, as pointed out in [29]: 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 [10] conditions the level on a boolean ghost variable subject to updates in program annotations which thereby express where in the code declassification is allowed. This is subsumed by our proposal.

As discussed in more detail elsewhere [4], [21], [29], several interesting proposals treat "where" declassification policies 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. Pre-assertions can sensibly be combined with any means to specify where in the code declassification is permitted, perhaps even achieving an end-to-end property like CGR.

Askarov and Sabelfeld [5] give a combination of "what" and "where" policies, dubbed localized delimited release, different from ours. The idea is to instrument the semantics to track expressions that have been declassified. 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 the 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. Allowing such tardy release might be difficult to reconcile with "when" policies like the accurate audit log in our Sect. 2. Localized delimited release is said to adhere to the prudent principles and is shown to be enforced by the type system for delimited release [28], with the additional restriction against declassification under high branch conditions. It could be interesting

to adapt the work to use more semantic reasoning about equivalence of expressions, and to incorporate assertions in policies. It does not seem obvious how to adapt the instrumented semantics to features of richer languages, such as heap objects.

10. Discussion

We extend the gradual release security property [4], which uses knowledge to describe information flow, with state conditions and agreements. Conditioned gradual release 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. Our enforcement régime combines simple type-based rules with program verification. To prove soundness, we devised an apparently novel technique: Owing to declassification it does not seem possible to define a notion of simulation (or unwinding conditions) of the usual sort; 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 [33] for Jif-style level-polymorphic typing. Heap data structures are essential for many applications.

Zdancewic [38] 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, the currently-enabled permissions in Java stack inspection can be tracked in a ghost variable [30] so flowspecs can express what is released given various privileges (cf. [6]). (Perhaps schematic flowspecs as in Sect. 4.) Zdancewic's second challenge is to "escape the confines of pure noninterference"; he mentions both declassification and conservativity of flow-insensitive static analysis. The third challenge is to manage complex policies. We conjecture that such policies should mostly be expressed using ordinary program specifications including state-based descriptions of sophisticated access controls.

Sabelfeld and Sands [29] suggest informal principles, with which our proposal seems to be in accord. *Semantic consistency* says that replacement of a "declass free" subprogram by a semantically equivalent one does not affect security. 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, even $\mathbf{skip}$ for $l := l$, so one must take the principle, and theoretical models like ours, with a grain of salt. The principle of *conservativity* amounts to our Proposition 5.7. This is problematic for [11] because their notion of security is not purely semantic. The principle of *monotonicity of release* says that adding a declassification cannot make a secure program insecure. This presupposes a treatment of declassification in which there is an explicit construct that can be "added" to a program. Unlike Jif and similar notations, our declass construct is distinct from policy specifications; if we wrap $l := l$ in a declass, we had better also add a baseline flowspec, $\mathbf{flow\ pre}\ \mathbf{A}(l)\ \mathbf{mod}\ l$, or CGR is violated. The principle of *non-occlusion* says that adding declassification cannot make an insecure program secure. Our proposal satisfies the principle, since the semantics of an assignment is not altered by marking it as a declass. A natural extension of our work is to add atomic blocks to the language (c.f. [16]) which would embody a more realistic attacker model for many purposes. Declassification of atomic blocks would not risk occlusion.

Acknowledgements. The exposition in this version of the paper is improved thanks to feedback from Aslan Askarov, Paul Karger, Andrei Sabelfeld, and anonymous referees. We also thank the organizers and participants of the Dagstuhl Seminar on Mobility, Ubiquity and Security, held in February 2007.

References

[1] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *POPL*, pages 91–102, 2006.

[2] T. Amtoft and A. Banerjee. Verification condition generation for conditional information flow. In *FMSE*, 2007.

[3] A. Askarov and A. Sabelfeld. Secure implementation of cryptographic protocols: A case study of mutual distrust. In *ESORICS*, pages 197–221, 2005.

[4] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *IEEE Symp. Security and Privacy*, pages 207–221, 2007.

[5] A. Askarov and A. Sabelfeld. Localized delimited release: combining the what and where dimensions of information release. In *PLAS*, pages 53–60, 2007.

[6] 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.

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

[8] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, pages 14–25, 2004.

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

[10] N. Broberg and D. Sands. Flow locks. In *ESOP*, pages 180–196, 2006.

[11] S. Chong and A. C. Myers. Security policies for downgrading. In *ACM CCS*, pages 198–209, 2004.

[12] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, 2002.

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

[14] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.

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

[16] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, 2003.

[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] K. Hristova, T. Rothamel, Y. A. Liu, and S. D. Stoller. Efficient type inference for secure information flow. In *PLAS*, pages 85–94, 2006.

[19] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In *FMCO*. 2003.

[20] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *POPL*, 2005.

[21] H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In *ESOP*, pages 141–156, 2007.

[22] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[23] P. Müller and A. Rudich. Ownership transfer in universe types. In *OOPSLA*, pages 461–478, 2007.

[24] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.

[25] D. A. Naumann. Verifying a secure information flow analyzer. In *TPHOLS*, pages 211–226, 2005.

[26] D. A. Naumann. From coupling relations to mated invariants for secure information flow and data abstraction. In *ESORICS*, 2006.

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

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

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

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

[31] S. F. Smith and M. Thober. Improving usability of information flow security in java. In *PLAS*, pages 11–20, 2007.

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

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

[34] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS*, pages 352–367, 2005.

[35] R. van der Meyden. What, indeed, is intransitive noninterference? In *ESORICS*, pages 235–250, 2007.

[36] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *CSFW*, pages 156–169, 1997.

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

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