A Logical Account of Secure Declassification (extended abstract) April 10, 2006

Anindya Banerjee
Kansas State University
ab@cis.ksu.edu

David A. Naumann
Stevens Institute of Technology
naumann@cs.stevens.edu

Abstract

Declassification is a vital ingredient for practical use of secure systems. Strong noninterference as a security policy does not account for declassification but is attractive as a baseline security policy because it provides an end-to-end account of security. Several recent efforts to formulate an end-to-end policy for declassification seem inconclusive and have focused on apparently different aspects, e.g., what values are involved, where in the code declassification occurs, when declassification happens and who (which principal) releases information. We argue that key security goals addressed by the proposed notions can be expressed using assertions with predicates and auxiliary state such as event history. The development is carried out in a modest extension of a recently developed Hoare-like logic for noninterference that provides for local reasoning about the heap.

1 Introduction

This paper is concerned with the specification and enforcement of secure information flow policies for imperative and sequential object-oriented programs. In its simplest form, a confidentiality policy labels certain variables as being secret, with the interpretation that the final value of a non-secret variable is not influenced by the initial values of any secrets. This has a precise semantic interpretation via *non-interference*: a program satisfies the policy if every pair of computations, from a pair of initial states differing only in secrets, leads to final states with identical non-secrets. Noninterference generalizes to a lattice of security levels and dualizes to integrity, i.e., that trusted outputs are not influenced by untrusted inputs. Such a policy is useful to the extent that the labeling is consistent with the actual deployment of the program—i.e., attackers are indeed prevented from directly reading secret variables. The property is not directly observable—it involves two runs of the program—but can potentially be enforced by using static analyses proved to ensure noninterference. Such enforcement is useful to the extent that a program is only executed if it is accepted by the analysis, e.g., an attacker may offer a Java applet but a user may insist on bytecode verification.

Enforcement by program analysis is useful for preventing Trojan horses and bugs; it does not address covert channels such as power consumption, which may or may not be a threat, depending on the system context. A more ubiquitous shortcoming is that, in practice, security policies are more subtle than noninterference owing to the need for declassification of secrets.

This paper argues for the use of program logic to specify security policies involving declassification as well as to verify compliance. The technical treatment is tailored to sequential programs in a language like Java but it should be evident that the ideas are pertinent to concurrent programs and other programming languages.

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 [33, 22, 28, 2, 24, 3, 5, 25]. By typing variable h as secret and l as low security, an evident rule disallows direct assignment of $l := h$ and additional constraints prevent implicit flows as in $\mathbf{if}\ h\ \mathbf{then}\ l := true$. Typechecking can be adapted to allow different level assignments at different points in the code but the flow-insensitivity which makes checking so fast also makes it reject many secure programs.

An alternative enforcement approach is to formulate security as a verification problem and use program logic [11, 9, 10]. Noninterference can be described by viewing one of the paired computations as acting on a renamed copy, say h', l', of the variables. For the two variables h, l, the equation $l = l'$ can be used as precondition and postcondition over state space h, h', l, l' to express the noninterference property as a pre-post specification $\{l = l'\}\ \_\ \{l = l'\}$ interpreted with respect to two runs. This idea can be realized in terms of a "relational Hoare logic" [6, 34, 1] or by embedding in standard program logic by composing the program with a renamed copy of itself [4, 32, 13]. (The latter technique was developed for reasoning about data abstraction in the 1970's [12, 26, 21, 16] and recently extended to heap structure [23].) In this paper we build on the work of Amtoft et al. [1] which addresses the key challenge for reasoning about object oriented programs—mutable data structure in the heap. Their logic appears much like conventional Hoare logic, but a triple $\{\varphi\}S\{\psi\}$ is interpreted with respect to state pairs and two executions of S. Assertions can include *independences* of the form $l \ltimes$ which basically means $l = l'$. Independences can also involve *region* expressions which

abstract the heap.

A number of proposals have been made to extend enforcement techniques, especially type-based ones, to encompass declassification. Unfortunately, no compelling semantic property has emerged to provide an end-to-end meaning for policies with declassification. Extant proposals seem fragmented and offer complicated analyses for limited and sometimes obscure properties [30]. Several proposals which address the more difficult forms of declassification boil down to a "resetting" semantics in which an intransitive noninterference condition connects the initial state to the point where a declassification takes place, and then again imposes noninterference between that point and the final state (or next declassification). Some proposals offer policies that say declassification happens only under certain conditions or under the control of certain agents. Yet, for lack of cogent semantics of declassification, the very well executed attempt by Sabelfeld and Sands [30] to make sense of the literature is only partly able to ground its informal principles in precise terms.

In this paper we borrow intuitions from existing proposals but formalize them as pre-post specifications in a relational Hoare logic. Without aiming to be exhaustive, we survey declassification examples from the recent literature and argue for decomposing the program, rather than the computations, and for specifying the components using independences together with state predicates. The approach addresses several dimensions of declassification [30] including what information is released, where in the code it can occur, when can it occur and under whose authority.

We extend the logic of Amtoft et al. [1] with richer assertions involving both state predicates and independences. We also revise the assertion language and proof rules, in particular making region disjointness and containment explicit in the assertion language, so that specifications are more transparent. (The presentation in [1] is tailored to serve as the specification of a flow-sensitive static analysis, rather than for direct use as a proof system.) We envision several applications of the logic. One is for proving security of programs; to this end it is attractive that the logic provides *heap-local reasoning* and a frame rule similar to that in Separation Logic [27], though using region predicates rather than separating conjunction. Another kind of application would be as basis for more specialized analyses: a type system could be proved sound by translating typing derivations to proofs in the logic, or the soundness could be embodied on a per-program basis for use in proof-carrying code.

Many of the recent proposals embed the security specification as part of the program, e.g., via a special declassification construct. In our approach, policy is expressed by pre-post specifications; for policies tied to specific program points, the tie is made by attaching specifications to designated subprograms. Triples in Hoare logic are intended to compose into proofs of complete programs; by contrast, our specifications for policy indicate an exemption from the baseline security policy (but can also impose ordinary 1-state preconditions under which the exempt code is allowed to execute).

This approach encompasses a wide range of policies in straightforward way and with clear semantics. Perhaps the main contribution is to do so while avoiding a number of anomalies found in previous proposals [30]. Whereas a number of works on declassification only pertain to a simple model of state, our logic encompasses the heap, which is relevant both as an information channel and as site of conditions on which declassification policy may depend.

Outline. Section 2 considers a range of declassification scenarios and describes how pre-post specifications can be deployed to express policies. The following sections formalize the logic and soundness. Section 7 puts our work in perspective.

2 Examples

Password checking. Here there are two principals, $User$ and $System$ with the latter password protected. Access is granted if $User$'s keyboard entry matches the password and denied otherwise. In either case, the result of comparison of the guess and password is revealed. Password checking must guarantee absence of laundering: no other information held secret by $System$ can be divulged. There are two policies of interest. The first is absence of laundering before declassification. Using $secret$, pwd and $guess$ as variables with the obvious meanings, and with the intention that $secret$ must not be laundered, the pre-post specification $\{guess\ltimes \wedge pwd\ltimes\}\ \_\ \{guess\ltimes \wedge pwd\ltimes\}$ applies to the code that obtains the user's guess. It says that for any two runs, if the initial values of $guess$ and pwd are identical then their final values are also identical. This is exactly the noninterference property associated with the labeling $secret : High$, $pwd : Low$, and $guess : Low$, as explained in Section 1.

The following code snippet, due to Chong and Myers [8] satisfies the specification:

$$secret := \text{`}A0uv'; pwd := \text{`}v0\,Y7Xa'; guess := getInput();$$

whereas $guess := secret$ does not.

The second policy pertains to the code that checks $guess$ and reveals the result, which should reveal the value of the comparison but nothing more. For any two runs, if the value of $guess = pwd$ is the same initially $result\ltimes$ should hold finally. This policy is expressed by

$$\{(guess = pwd)\ltimes\}\ \_\ \{result\ltimes\}$$

The code $result := (guess = pwd)$ satisfies the above specification. On the other hand, $result := secret$ does

not. The code $result := secret; result := (guess = pwd)$, which would be rejected by most type systems, can also be proved to meet the specification.

In the terminology of [30], the partial release of $(guess = pwd)$ but not pwd is a "what" policy. We are also expressing a policy about "where" in the code a release is permitted, by expressing our policy for the seqential code $getWithoutLaunder; compareAndRelease$ as two specifications, $\{\varphi_1\}\ getWithoutLaunder\ \{\varphi_2\}$ and $\{\varphi_3\}\ compareAndRelease\ \{\varphi_4\}$, that *do not* compose (i.e., our φ_2 does not imply φ_3).

The individual triples have a conventional, extensional meaning, and our proof system resembles Hoare logic. But this use of two triples to specify an intransitive noninterference policy should not be confused with ordinary intermediate assertions to express proof outlines.

Electronic Wallet. Consider the following, due to Sabelfeld and Myers [29], where h is the only secret.

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

The declassification policy is again partial release: it is ok to reveal $h \geq k$, but nothing more about h must be revealed. Our specification is

$$(1) \qquad \{(h \geq k)\ltimes \wedge l\ltimes \wedge k\ltimes\} \_ \{l\ltimes \wedge k\ltimes\}$$

For the code above, this is easy to verify: k is not modified, so since $k\ltimes$ holds before, it holds after; and $l\ltimes$ is established because it holds in the precondition along with $k\ltimes$.

Sabelfeld and Myers [29] propose explicit $declassify$ expressions in code to serve as "escape hatches", marking the expressions whose (initial) values are to be released. For example, the code above would contain $declassify(h \geq k)$.

Consider now a variation of the code, also due to Sabelfeld and Myers, where the secret variable h is an n-bit integer. Assume n is public as are l and k.

$l := 0;$
while $n > 0$ **do**
  $k := 2^{n-1};$
  if $h \geq k$ **then** $h := h - k; l := l + k$ **else skip**;
  $n := n - 1$

The code does not satisfy the specification (1), even if $n\ltimes$ is added to the precondition.

Suppose we decide to release the least significant 2 bits of h in addition to the information $h \geq k$. That is, the specification is

$$\{(h \geq k)\ltimes \wedge (h\%4)\ltimes \wedge n\ltimes \wedge l\ltimes \wedge k\ltimes\} \_ \{l\ltimes\}$$

The code above does not meet this specification. But it can be proved to satisfy the specification with $0 \leq n \leq 2$ added to the precondition. This kind of reasoning goes beyond reasoning in most type systems including [29]. The type system also rejects the following secure program: $h := h\%2; l := declassify(h\%2)$. The reason is that the program falls afoul of the restriction that variables used under declassification may not be updated prior to declassification. Our logic, on the other hand, can prove it satisfies $\{h\%2\ltimes\} \_ \{l\ltimes\}$ using some arithmetic reasoning.

Sealed auctions. Consider two principals $Alice$ and Bob taking part in an auction. The protocol is as follows: $Alice$ and Bob place their bids; the system determines the higher bid and reveals the value of the high bid and the identity of the high bidder. The first policy says that neither bid influences the other. The second policy says that only the high bid and bidder are made public. For instance, if $Alice$ is the winner, the value of Bob's bid remains secret. Note that the two policies address two separate concerns: the first, absence of cheating, is up to the point just before the outcome is determined. The second concern, absence of laundering, applies from that point through the action of publicizing the result. Surely these are clearly identifiable code fragments.

The first concern is addressed by conjoining two specifications. The code for placing bids must satisfy the conjunction of the specifications $\{bBid\ltimes\} \_ \{bBid\ltimes\}$ (guarding $Alice$'s bid) and $\{aBid\ltimes\} \_ \{aBid\ltimes\}$ (guarding Bob's bid), where $bBid$ and $aBid$ are the bids placed by Bob (resp. $Alice$). [1]

The code $aBid := 100; bBid := 150$ satisfies the first specification, whereas the code $aBid := 100; bBid := aBid + 1$ does not: Bob has placed his bid after getting information about $Alice$'s.

The next stage of the auction determines the winning bid and the winner and reveals both pieces of information. Here is our specification:[2]

$$\{(max(aBid, bBid))\ltimes \wedge (aBid \geq bBid)\ltimes\} \_ \{result\ltimes\}$$

The code $result := (max(aBid, bBid), (aBid \geq bBid))$ satisfies the above specification. Here is one way in which malicious code could try to release a losing bid, in this case, Bob's.

$winBid := max(aBid, bBid);$
$aliceWins := aBid \geq bBid;$
if $aliceWins$ **then** $winBid := bBid$ **else** $winBid := aBid;$
$result := (winBid, aliceWins)$

[1] Apropos security type systems, this is akin to type checking assuming $aBid : H, bBib : L$ and then type checking again with assumption $aBid : L, bBib : H$; the underlying security lattice is the 4-point diamond lattice with $Alice$ and Bob incomparable. Note how just one typing could be used with incomparable types, whereas in the logic-based approach separate policies must be verified for each principal.

[2] It allows a leak to $aBid$ or $bBid$, which could be disallowed by a "modifies" clause or explicit postconditions.

Now $result\ltimes$ can no longer be established as postcondition, because $winBid\ltimes$ cannot.

Intermezzo. It is possible for partial release to be conditional [29], using conditional expressions, e.g., to release a secret only if adequate payment is made. This can also be done in a specification, using precondition $(\mathbf{if}\ pay > thresh\ \mathbf{then}\ secret\ \mathbf{else}\ 0)\ltimes$ and postcondition $result\ltimes$. The meaning of $(\mathbf{if}\ pay > thresh\ \mathbf{then}\ secret\ \mathbf{else}\ 0)\ltimes$ is that for any two runs of a program the value of the conditional is the same. Note that it is possible that different branches of the conditional get executed in the two runs and yet yield the same value, provided $secret = 0$. An alternative specification uses disjunction in the precondition $pay \leq thresh \ \dot\vee\ secret\ltimes$. This is satisfied in a pair of initial states if either $pay < thresh$ is false in both or the value of $secret$ is the same in both. So the resulting specification differs in meaning from the one using the conditional expression only when $secret = 0$.

Sabelfeld and Myers also suggest that conditional release can be extended to disjunctive policies, but do not propose a syntax. Our conjecture is that this is because their syntax is too closely tied to the program (the policy is determined by the conjunction of all the *declassify* statements). It is perhaps better to spell out the policy in specifications, which are sets of Hoare triples (interpreted conjunctively). A disjunctive policy is easily expressed: to allow either h_1 or h_2 but not both to be released, the precondition is $h_1\ltimes \ \dot\vee\ h_2\ltimes$ and the postcondition $result\ltimes$.

As an example of disjunctive policies, consider a card game in which the player reveals one card from her hand in which all cards are initially held secret; the policy is that in a single round of the game, the player chooses to reveal either the first card, or the second card, etc.

A pre-post specification imposes no constraint on computations from states falsifying the precondition. Our use of specifications is not for composition into proof outlines, where a precondition imposes an obligation on the environment, it does make sense to include preconditions on one copy of the state that does play that role. For example, a declassifier method may require to be invoked in a process with some specific authority.

Access control for information flow. Stack inspection is an access control mechanism present in Java [15] and the .NET CLR. Each class C has a set of permissions statically granted to it by the virtual machine, often based on code origin, but initially not enabled. We assume there is a ghost variable Q, in terms of which we reason using the "eager" description of the mechanism [15, 31]. So Q is the set of currently enabled permissions. The command $\mathbf{enable}\ p\ \mathbf{in}\ S$ checks whether permission p is statically authorized for the class implementing the method in execution, and if so p is added to Q for the duration of S. The boolean expression $\mathbf{check}\ p$ checks whether p is in Q. In [2], the authors study the pattern wherein high security information is released by a certain method only when the caller has permission, in this case permission $stat$.

$\mathbf{class}\ Kern\ \mathbf{extends}\ Object\{$//static perms: $stat$, sys
  $\mathbf{String}\ Hinfo; \mathbf{String}\ Linfo;$
  $\mathbf{String}\ getHinfo()\{$
    $\mathbf{if\ check}\ sys\ \mathbf{then}\ result := Hinfo\ \mathbf{else\ abort}\}$
  $\mathbf{String}\ getStatus()\{$
    $\mathbf{if\ check}\ stat$
      $\mathbf{then\ enable}\ sys\ \mathbf{in}\ result := \mathbf{self}.getHinfo()$
      $\mathbf{else}\ result := Linfo\}\}$

When $getStatus$ is called, permission $stat$ is checked. If it is in Q then sys gets enabled and $Hinfo$ is obtained via a call to $getHinfo$. On the other hand, if $stat$ is not in Q then $Linfo$ is released. The policy for $getStatus$, that $Hinfo$ is only released to callers with permission $stat$, is expressed in [2] by giving $getStatus$ two types. It can be expressed by a single specification, in which we also need to deal with the heap.

$$\{stat \notin Q \wedge self \rightsquigarrow L \wedge \mathbf{self}\ltimes \wedge L.Linfo\ltimes\}\ \_\ \{result\ltimes\}$$

Assume that the heap is divided into several (possibly overlapping) regions and the actual object, say o, bound to **self** appears in region L of the heap. This is written as the assertion, $\mathbf{self} \rightsquigarrow L$. Roughly, we use the notation $L.Linfo$ to denote all objects, $o.Linfo$ where $o \in L$.

The fields $Linfo$ and $Hinfo$ could be model fields [17] defined by a representation function based on some heap objects.

Release in multiple steps. Chong and Myers [8] use types to express declassifications that can happen only after several conditions have been true in succession. They do not give an example with multiple steps so we sketch our own. The pattern is similar to that of password checking, except that there are three phases in succession.

Consider a patient record that includes medical information that only doctors and the patient are permitted to access (according to the baseline policy), say their HIV status. There is also a doctor's log and a nurse's log. A doctor is allowed to release the HIV status to nurses (say, assign it to a field labeled with the "nurse" level), provided a log entry is made with authenticating evidence for the doctor. A nurse is allowed to release the HIV status (perhaps in sanitized form—a partial release) to insurance representatives, provided a doctor has already logged a release to nurses and moreover the nurse makes a log entry with authentication of the nurse and insurance rep. These actions will correspond to clear segments of code (e.g., a certain method that does the action and is invoked in a GUI listener). The policies

$$
\begin{array}{lcl}
T & ::= & \mathbf{int} \mid C \quad \text{where } C \in \mathit{ClassName} \\
CL & ::= & \mathbf{class}\ C\ \{\ \overline{T}\ \overline{f}; \overline{M}\ \} \quad \text{where } f \in \mathit{FieldName} \\
M & ::= & T\ m(U\ u)\ \{S\} \quad \text{method declaration} \\
S & ::= & x := E \mid x.f := y \\
 & \mid & x := \mathbf{new}\ C \mid x := y.f \\
 & \mid & x := y.m(z) \mid S\ ;\ S \\
 & \mid & \mathbf{if}\ x\ \mathbf{then}\ S\ \mathbf{else}\ S \mid \mathbf{while}\ x\ \mathbf{do}\ S \\
E & ::= & x \mid c \mid \mathbf{null} \mid E\ \mathsf{op}\ E \mid k(E)
\end{array}
$$

Figure 1. BNF of language

can be formulated as two specifications that can both be imposed on any methods that fail a type-check for the baseline policy. The logs need not be present as such; ghost variables could be used to record the relevant event history in preconditions.

3 Language

The technical contribution of this paper is a logic, for an object-oriented programming language, that is sound and sufficiently expressive to specify and verify policies like those in the preceding section—and more practical examples where declassification is governed by nontrivial predicates on program state. This section sketches the language and its semantics, which is taken from [1].

The language is essentially sequential Java, omitting exceptions. The grammar is in Figure 1. As in separation logic, programs are desugared so that expressions do not depend on the heap; this is why there are several forms of assignment including $x := y.f$. To save space we omit consideration of method calls in the rest of this extended abstract; their semantics, specifications, and proof rules can be treated as in [1]. This lets us focus entirely at the level of commands in the rest of the paper.

A *state* consists of a *store* assigning current values to local variables (including parameters of the method body in which the command occurs, among which is the target object **self**) together with a heap. A *heap* is a mapping, defined on a finite set of currently-allocated object references (drawn from a countable set Ref); it maps each such reference to a record of the object's current field values. So $h\ o\ f$ is the value of field f of object o. We use the term *reference* for addresses, to emphasize that they are an abstract type—no pointer arithmetic. The term "location" is used later, to mean assignment target in modifies specifications.

Commands are given a relational, partial-correctness semantics [1], in which $(s,h)[\![S]\!](s',h')$ means that from the initial state with store s and heap h, command S can terminate in state (s',h'). Note that the domain of s' is the same as that of s, that is, the local variables. The semantics does not model garbage collection, so $\mathit{dom}(h)$ is always a subset of $\mathit{dom}(h')$. Without formalization, we assume S is well typed.

By contrast with the language, the logic presented in subsequent sections is significantly changed from [1]. The abstract domain of regions (called locations in [1]) has been dropped and the assertion language augmented with region identifiers and atomic predicates for disjointness and inclusion of regions. Hoare triples are augmented with a context for region invariants and the proof rules are revised accordingly. Creation effects and second level disjunction have been added but these also appear in the technical report [1]. An atomic predicate is added for field access (since expressions E do not depend on the heap) and this is used to strengthen the rules for assignment.

4 Assertions

Two kinds of formulas are used in assertions. Ordinary first order formulae over program variables, ranged over by θ in the grammar below, are called *1-predicates*. Formulas involving independences, ranged over by φ, are called *2-predicates*. The latter are given a two-state semantics and this is our main interest. The embedding of 1-predicates as 2-predicates is interpreted as follows: θ is true in a pair of states if it is true in both of them.

The grammar of 1-predicates is as follows. Just as x ranges over a countable set (elements of which are called variables), we let L range over a countable set, elements of which are called *regions*, more properly: region identifiers. (The L is mnemonic for "abstract locations", the term used in [1]). In the semantics, regions are interpreted as sets of references. In addition to regions we add pseudo-region $\bot$, which is always mapped to the singleton of null, and pseudo-region **int** which is mapped to the set of integers (this facilitates a uniform treatment of fields and variables of primitive or pointer type).

$$
\begin{array}{lcl}
LI & ::= & L \mid \mathbf{int} \mid \bot \quad \text{regions and psuedo regions} \\
\theta & ::= & E = E \mid x = y.f \mid \ldots \quad \text{atomic predicates} \\
 & \mid & \theta \wedge \theta \mid \theta \vee \theta \mid \neg\theta \\
 & \mid & x \rightsquigarrow LI \mid L.f \rightsquigarrow LI \mid LI \diamond LI \mid LI \leq LI
\end{array}
$$

The atomic predicate $x \rightsquigarrow L$ means that the current value of x is a reference abstracted by L; $L_1 \diamond L_2$ means that these two regions represent disjoint sets of references; $L_1 \leq L_2$ means that the references abstracted by L_1 are also abstracted by L_2. Since expressions E do not depend on the heap, we need an atomic formula $x = y.f$ to access fields (cf. the points-to predicate in separation logic).

For brevity in this extended abstract we omit quantifiers; quantification over heap references requires a little care since for practical use there needs to be a way to restrict to allocated references [17].

The grammar of 2-predicates is as follows.

$$\varphi ::= \theta \mid E\ltimes \mid L.f\ltimes \mid \varphi \wedge \varphi \mid \varphi \mathbin{\dot\vee} \varphi$$

The meaning of $a\ltimes$ is that the *two* current states in question, say (s,h) and (s_1,h_1), agree on the value of a. Logical connectives are available at both levels, and in our concrete syntax it is ambiguous whether $\theta_1 \wedge \theta_2$ is a conjunction at the level of 1-predicates that is embedded as a 2-predicate, or is a conjunction of 2-predicates which happen both to be 2-predicates. The ambiguity is harmless since the meaning is the same in each case. Disjunction is slightly trickier, as hinted by the $pay > thresh$ example in Section 2. If we embed the 1-predicate $\theta_1 \vee \theta_2$ then the interpretation is that $\theta_1 \vee \theta_2$ must be true in both of the two states—which allows that θ_1 is true in one and θ_2 in the other. If instead we embed θ_1 and θ_2 separately, their disjunction as 2-predicates is written $\theta_1 \mathbin{\dot\vee} \theta_2$ and has a different interpretation in a pair of states: either θ_1 is true in both states or θ_2 is.

For use in the proof rule for conditionals, we need to extract the independence content, if you will, from 2-predicates. For any φ, obtain $\mathcal{I}(\varphi)$ from φ by replacing each subformula θ (i.e., a 1-predicate) by $true$.

Semantics of 1-predicates. To give a precise meaning to 1-predicates, we use extraction relations η between values and regions. An *extraction relation* must satisfy

- $v\,\eta\,\bot$ iff $v = nil$
- $v\,\eta\,\mathbf{int}$ iff v is an integer
- $v\,\eta\,L$ implies v is a reference (neither nil nor an integer)

We say that η *is over* h if $o\,\eta\,L$ implies $o \in dom(h)$ for all references o.

The semantics of a 1-predicate θ is given as a satisfaction relation, written $(s,h) \models_\eta \theta$ and defined for all (s,h) and all extraction relations η over h. The definition is in Figure 2.

A 1-predicate θ is called *valid* just if for any (s,h) and any η over h we have $(s,h) \models_\eta \theta$. We refrain from giving proof rules but note that the semantics validates classical logic. Here are some valid predicates:

$x = y.f \Rightarrow \neg(y \rightsquigarrow \bot)$
$L \diamond \bot$ and $L \diamond \mathbf{int}$
$L \diamond L_1 \Rightarrow L \not\leq L_1 \wedge L_1 \not\leq L$
$L \diamond L_1 \iff L_1 \diamond L$
$x \rightsquigarrow \bot \vee x \rightsquigarrow \mathbf{int} \Rightarrow \neg(x \rightsquigarrow L)$
$x \rightsquigarrow L \Rightarrow \neg(x \rightsquigarrow \bot) \wedge \neg(x \rightsquigarrow \mathbf{int})$
$x \rightsquigarrow L \wedge L \leq L_1 \Rightarrow x \rightsquigarrow L_1$
$L \diamond L_1 \wedge L_2 \leq L_1 \Rightarrow L \diamond L_2$
and similarly for $L.f \rightsquigarrow \ldots$.

Semantics of 2-predicates. To cater for renaming of references and differing allocation behavior under high guards, formalization of noninterference for heaps involves manipulation of bijections as explained in [2]. Let β range over bijections from a subset of Ref to a subset of Ref. That is, if $o\,\beta\,o_1$ and $o\,\beta\,o_2$ then $o_1 = o_2$, but for some o there might not be any o_1 such that $o\,\beta\,o_1$; and symmetrically. Every β is silently lifted to a relation on all values, by taking it to be the identity on nil and on integers.

We say that β *is over* $h\&h_1$ if $o\,\beta\,o_1$ implies $o \in dom(h)$ and $o_1 \in dom(h_1)$ (Throughout the paper, o and o_1 range over references only.) For extraction relations η over h and η_1 over h_1 we say η, η_1 are *compatible with* β just if $o\,\beta\,o_1$ implies $(o\,\eta\,L$ iff $o_1\,\eta_1\,L)$. That is, references o and o_1 related by β are abstracted to the same region.

The two-state semantics of assertion φ is written $(s,h)\&(s_1,h_1) \models_{\beta,\eta,\eta_1} \varphi$ and defined for any β over $h\&h_1$ and η, η_1 compatible with β. The definition is in Figure 3. Validity is defined in the usual way. Here are some valid implications:

$x = c \Rightarrow x\ltimes$ where c is an integer literal
$x = y \wedge y\ltimes \Rightarrow x\ltimes$
$x = k(y) \wedge y\ltimes \Rightarrow x\ltimes$ where k is an arithmetic function
$z_1\ltimes \wedge \ldots \wedge z_n\ltimes \Rightarrow E\ltimes$ where $FV(E) \subseteq \{z_1 \ldots z_n\}$

We refrain from giving a proof system for 2-predicates.

5 Program judgements

Judgements in the program logic have the form

$$\Delta \vdash \{\varphi\}\ S\ \{\varphi'\}\ [X]$$

Here X is a set of *locations* λ which include program variables and fields of regions, which are susceptible to update, and also regions to which newly created objects may be added: $\lambda ::= x \mid L.f \mid L$. The idea is that if y can be updated by S then y is in X; if $o.f$ can be updated then o is abstracted by some $L.f$ in X; and if o can be created then it is abstracted in the final state by some L in X. We call X the *effect set*; it is like the standard "modifies clause" but with the addition of creation effects.

The *region context* Δ ranges over sets of disjointness and containment invariants of the form $L \diamond L_1$ and $L \leq L_1$. The disjointnesses, in particular, are used to discharge an antecedent of the same form in the frame rule, as we will explain in due course. That rule also involves the effect set X. We begin by laying the groundwork for its semantics.

Recall that 2-predicates are interpreted in contexts of the form $(s,h)\,\&\,(s_1,h_1) \models_{\beta,\eta,\eta_1} \ldots$ where η is over h, η_1 over h_1, etc. We need a notion of extension, for such contexts, to the final state, which has heaps h', h_1' with $dom(h') \supseteq dom(h)$ and $dom(h_1') \supseteq dom(h_1)$. In this situation, we say η' over h' *extends* η iff for all $o \in dom(h)$ and all L we have $o\,\eta\,L$ iff $o\,\eta'\,L$. That is, η' does not change the region of a pre-existing location. For β' over $h'\&h_1'$ we

$$
\begin{array}{lll}
(s,h) \models_\eta E = E_1 & \text{iff} & [\![E]\!]s = [\![E_1]\!]s \\
(s,h) \models_\eta x = y.f & \text{iff} & s(y) \in dom(h) \text{ and } s(x) = h\,(s(y))\,f \\
(s,h) \models_\eta x \rightsquigarrow LI & \text{iff} & s(x)\,\eta\,LI \\
(s,h) \models_\eta L.f \rightsquigarrow LI & \text{iff} & \forall o \cdot o\,\eta\,L \Rightarrow (hof)\,\eta\,LI \\
(s,h) \models_\eta LI_1 \diamond LI_2 & \text{iff} & \{o \mid o\,\eta\,LI_1\} \cap \{o \mid o\,\eta\,LI_2\} = \emptyset \\
(s,h) \models_\eta LI_1 \leq LI_2 & \text{iff} & \forall o \cdot o\,\eta\,LI_1 \Rightarrow o\,\eta\,LI_2 \\
(s,h) \models_\eta \neg\theta & \text{iff} & \text{not } (s,h) \models_\eta \theta \\
(s,h) \models_\eta \theta_1 \wedge \theta_2 & \text{iff} & (s,h) \models_\eta \theta_1 \text{ and } (s,h) \models_\eta \theta_2 \\
(s,h) \models_\eta \theta_1 \vee \theta_2 & \text{iff} & (s,h) \models_\eta \theta_1 \text{ or } (s,h) \models_\eta \theta_2
\end{array}
$$

Figure 2. Semantics of selected 1-predicates.

$$
\begin{array}{lll}
(s,h)\,\&\,(s_1,h_1) \models_{\beta,\eta,\eta_1} \theta & \text{iff} & (s,h) \models_\eta \theta \text{ and } (s_1,h_1) \models_{\eta_1} \theta \\
(s,h)\,\&\,(s_1,h_1) \models_{\beta,\eta,\eta_1} E{\ltimes} & \text{iff} & ([\![E]\!]s)\,\beta\,([\![E]\!]s_1) \\
(s,h)\,\&\,(s_1,h_1) \models_{\beta,\eta,\eta_1} L.f{\ltimes} & \text{iff} & o\,\beta\,o_1 \text{ and } o\,\eta\,L \text{ imply } (h\,o\,f)\,\beta\,(h_1\,o_1\,f) \text{ for all } o, o_1 \\
(s,h)\,\&\,(s_1,h_1) \models_{\beta,\eta,\eta_1} \varphi_1 \wedge \varphi_2 & \text{iff} & (s,h)\,\&\,(s_1,h_1) \models_{\beta,\eta,\eta_1} \varphi_1 \text{ and } (s,h)\,\&\,(s_1,h_1) \models_{\beta,\eta,\eta_1} \varphi_2 \\
(s,h)\,\&\,(s_1,h_1) \models_{\beta,\eta,\eta_1} \varphi_1 \dot{\vee} \varphi_2 & \text{iff} & (s,h)\,\&\,(s_1,h_1) \models_{\beta,\eta,\eta_1} \varphi_1 \text{ or } (s,h)\,\&\,(s_1,h_1) \models_{\beta,\eta,\eta_1} \varphi_2
\end{array}
$$

Figure 3. Semantics of 2-predicates.

say β' *extends* β iff

$$\beta = \{(o,o_1) \mid (o,o_1) \in \beta' \wedge (o \in dom(h) \vee o_1 \in dom(h_1))\}$$

This is stronger than $\beta' \supseteq \beta$. It says that if $o\beta' o_1$ and o is in the initial heap h then o_1 is in the initial heap h_1 (and symmetrically).

To express that effect X is an over-approximation of the variables and object fields modified in an execution, we introduce some notation. Suppose h' *extends* h in the sense that $dom(h') \supseteq dom(h)$. For η over h' we define

$$(s,h) \rightarrow (s',h') \models_\eta X$$

iff (a) for every $y \in dom(s)$ we have $s(y) = s'(y)$ or $y \in X$; (b) for every f and every $o \in dom(h)$, if $h\,o\,f \neq h'\,o\,f$ then there is some $L.f$ in X such that $o\eta L$; and (c) for every o in $dom(h')$ but not in $dom(h)$ there is some L in X with $o\eta L$.

Disjointness and a frame property. We define several judgements involving disjointness consequences of a region context Δ. First, we write $\Delta \vdash L \diamond L_1$ just if $L \diamond L_1$ is a valid consequence of Δ and the same for $L \leq L_1$. Now $\Delta \vdash \lambda \diamond X$ is defined as follows:

- $\Delta \vdash L.f \diamond X$ iff $\Delta \vdash L \diamond L_1$ for all $L_1.f$ in X
- $\Delta \vdash y \diamond X$ iff $y \notin X$
- $\Delta \vdash L \diamond X$ iff $L \notin X$

This in turn is used to define disjointness for 2-predicates, which is used in the frame rule. Define $\Delta \vdash \varphi \diamond X$ by structural induction on φ. Here are the cases for atomic formulas. Note: $y \rightsquigarrow LI \diamond X$ is parsed as $(y \rightsquigarrow LI) \diamond X$.

$$\frac{\Delta \vdash y \diamond X}{\Delta \vdash y \rightsquigarrow LI \diamond X} \qquad \frac{\Delta \vdash L.f \diamond X}{\Delta \vdash L.f \rightsquigarrow LI \diamond X}$$

$$\frac{\Delta \vdash y \diamond X}{\Delta \vdash y{\ltimes} \diamond X} \qquad \frac{\Delta \vdash L.f \diamond X}{\Delta \vdash L.f{\ltimes} \diamond X}$$

$$\frac{\Delta \vdash y \diamond X \qquad \Delta \vdash y_1 \diamond X}{\Delta \vdash (y = y_1) \diamond X}$$

The last case is representative for other primitive conditions; each free variable must be disjoint from X. The rules for compound formulas just distribute, for example:

$$\frac{\Delta \vdash \varphi \diamond X \qquad \Delta \vdash \varphi_1 \diamond X}{\Delta \vdash (\varphi \wedge \varphi_1) \diamond X}$$

The key lemma says that if $\varphi \diamond X$ then φ is not falsified by updates to locations in X. This underlies the frame rule.

For disjointness formulas $L \diamond L_1$, the satisfaction relation $(s,h) \models_\eta L \diamond L_1$ depends only on η, not the state. So we define $\eta \models \Delta$ iff for each $L \diamond L_1$ in Δ there is some (s,h) with $(s,h) \models_\eta L \diamond L_1$.

Semantics of program judgements. We define $\Delta \models \{\varphi\}\ S\ \{\varphi'\}\ [X]$ iff the following holds for all $s, h, s_1, h_1, s', \ldots, \eta, \eta_1, \beta$ over $h\&h_1$. If

- $(s,h)\&(s_1,h_1) \models_{\beta,\eta,\eta_1} \varphi$
- $(s,h)[\![S]\!](s',h')$ and $(s_1,h_1)[\![S]\!](s_1',h_1')$
- $\eta \models \Delta$ and $\eta_1 \models \Delta$

then there exist β', η', η_1' such that

- $(s',h')\&(s_1',h_1') \models_{\beta',\eta',\eta_1'} \varphi'$
- β', η', η_1' extend β, η, η_1
- $(s,h) \rightarrow (s',h') \models_{\eta'} X$ and $(s_1,h_1) \rightarrow (s_1',h_1') \models_{\eta_1'} X$
- $\eta' \models \Delta$ and $\eta_1' \models \Delta$

6 Proof Rules

We begin with the small rules for primitive commands. These are "small" in the sense of mentioning only the relevant regions and also using a minimal region context. We abuse notation and omit the braces when writing enumerated effect sets. For assignment of a pure integer expression, one rule is

$$\emptyset \vdash \{E{\ltimes} \wedge \theta[E/x]\}\ x := E\ \{x \rightsquigarrow \mathbf{int} \wedge x{\ltimes} \wedge \theta\}\ [x]$$

There is a second rule: just omit the independences from both precondition and postcondition. For assignments $x := \mathbf{null}$ and $x := z$ the rules are similar. Here is the first of two rules for field access, the second being obtained by omitting the independences:

$$\begin{array}{l}\emptyset \vdash \{y \rightsquigarrow L \wedge L.f \rightsquigarrow LI \wedge y{\ltimes} \wedge L.f{\ltimes}\} \\ \quad x := y.f \\ \quad \{x \rightsquigarrow LI \wedge x{\ltimes} \wedge x = y.f\}\ [x]\end{array}$$

The first of two for field update:

$$\begin{array}{l}\emptyset \vdash \{x \rightsquigarrow L \wedge y \rightsquigarrow LI \wedge L.f \rightsquigarrow LI \wedge x{\ltimes} \wedge y{\ltimes} \wedge L.f{\ltimes}\} \\ \quad x.f := y \\ \quad \{L.f \rightsquigarrow LI \wedge L.f{\ltimes} \wedge y = x.f\}\ [L.f]\end{array}$$

The first of two for New:

$$\emptyset \vdash \{true\}\ x := \mathbf{new}\ C\ \{x \rightsquigarrow L \wedge x{\ltimes} \wedge z = x.f\}\ [x, L]$$

Note that the region L chosen for the postcondition is also in the effect set. Postcondition $z = x.f$ can be instantiated with any z and any field of the type of x (though we omit typing issues in this extended abstract); it expresses that x is allocated.

Compound commands. A single rule for sequence:

$$\frac{\Delta \vdash \{\varphi_0\}\ S_1\ \{\varphi_1\}\ [X_1] \qquad \Delta \vdash \{\varphi_1\}\ S_2\ \{\varphi\}\ [X_2]}{\Delta \vdash \{\varphi_0\}\ S_1\,;S_2\ \{\varphi\}\ [X_1 \cup X_2]}$$

For If there are two rules. The first is for "low guard" [28]:

$$\frac{\begin{array}{c}\Delta \vdash \{\varphi_0 \wedge x > 0\}\ S_1\ \{\varphi\}\ [X] \\ \Delta \vdash \{\varphi_0 \wedge x \leq 0\}\ S_2\ \{\varphi\}\ [X] \qquad \Delta \models \varphi_0 \Rightarrow x{\ltimes}\end{array}}{\Delta \vdash \{\varphi_0\}\ \mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \{\varphi\}\ [X]}$$

The second is more elaborate, to achieve the effect of not writing low under a high guard (see [1] for explanation):

$$\frac{\begin{array}{c}\Delta \vdash \{\varphi_0 \wedge x > 0\}\ S_1\ \{\varphi\}\ [X] \\ \Delta \vdash \{\varphi_0 \wedge x \leq 0\}\ S_2\ \{\varphi\}\ [X] \qquad \varphi \text{ contains no } \dot{\vee} \\ \Delta \vdash \mathcal{I}(\varphi) \diamond X \qquad \Delta \models \varphi_0 \Rightarrow \mathcal{I}(\varphi)\end{array}}{\Delta \vdash \{\varphi_0\}\ \mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \{\varphi\}\ [X]}$$

Structural rules. First we introduce a form of implication for effect sets. Say that $\Delta \vdash X \blacktriangleright X'$ iff $(s,h) \rightarrow (s',h') \models_\eta X$ and $\eta \models \Delta$ imply $(s,h) \rightarrow (s',h') \models_\eta X'$ for all s, s', h, h', η. Here is a sufficient condition, expressed syntactically: If

- $x \in X$ implies $x \in X'$
- $L \in X$ implies there exists $L' \in X'$ with $\Delta \vdash L \leq L'$
- $L.f \in X$ implies there exists L' such that $L'.f \in X'$ and $\Delta \vdash L \leq L'$

then $\Delta \vdash X \blacktriangleright X'$. In particular, if $X \subseteq X'$ then $X \blacktriangleright X'$.

The rule of consequence uses valid implications as well as the implication for effect sets.

$$\frac{\begin{array}{c}\Delta \vdash \{\varphi\}\ S\ \{\varphi'\}\ [X] \\ \Delta \models \varphi_1 \Rightarrow \varphi \qquad \Delta \models \varphi' \Rightarrow \varphi_1' \qquad \Delta \vdash X \blacktriangleright X_1\end{array}}{\Delta \vdash \{\varphi_1\}\ S\ \{\varphi_1'\}\ [X_1]}$$

The frame rule uses the disjointness judgement.

$$\frac{\Delta \vdash \{\varphi\}\ S\ \{\varphi'\}\ [X] \qquad \Delta \vdash \varphi_1 \diamond X}{\Delta \vdash \{\varphi \wedge \varphi_1\}\ S\ \{\varphi' \wedge \varphi_1\}\ [X]}$$

There are two rules of disjunction which look like Hoare's rules, one for $\vee$ and one for $\dot{\vee}$. Hoare's rule of conjunction, however, is unsound for the reasons discussed in [1]. The small rules together with frame and consequence appear to suffice for reasoning where one might expect to use conjunction.

The remaining rules manipulate the region context. First, new disjointnesses can be introduced. A side condition is needed since Δ plays a role as both pre- and post-condition.

$$\frac{\Delta \vdash \{\varphi\}\ S\ \{\varphi'\}\ [X] \qquad (\text{either } L \notin X \text{ or } L_1 \notin X)}{\Delta, L \diamond L_1 \vdash \{\varphi\}\ S\ \{\varphi'\}\ [X]}$$

Region invariants can be moved to specifications: letting RI stand for either $L \diamond L_1$ or $L \leq L_1$, the rule is

$$\frac{\Delta, RI \vdash \{\varphi\}\ S\ \{\varphi'\}\ [X]}{\Delta \vdash \{RI \wedge \varphi\}\ S\ \{RI \wedge \varphi'\}\ [X]}$$

Soundness and completeness. Amtoft et al. [1] prove soundness of their logic. In large part our adaptations make explicit in the region context the disjointness assumptions embodied in the fixed lattice and disjointness operator used in their work. So much of the soundness argument can be adapted to our setting. We are convinced of the following result but have not completed a rigorous proof.

Theorem 6.1 (soundness) *If* $\Delta \vdash \{\varphi\}\ S\ \{\varphi'\}\ [X]$ *then* $\Delta \models \{\varphi\}\ S\ \{\varphi'\}\ [X]$.

A natural formulation of completeness would be relative to completeness of the proof system for assertions, indeed, we have built that in to the program logic by referring to validity in the rule of consequence and elsewhere. From a practical point of view, our logic inherits some inexpressiveness; it lacks quantifiers and inductive definitions or reachability. These should be straightforward to add. But few completeness results are known for object-oriented programs and at this stage it is probably more fruitful to explore practical utility.

Semantic consistency Sabelfeld and Sands [30] define the principle of semantic consistency: "The (in)security of a program is invariant under semantics-preserving transformations of declassification-free subprograms." This holds in our approach, owing to the use of program specifications with extensional semantics.

Lemma 6.2 *If* $[\![S]\!] = [\![S_1]\!]$ *and* $\Delta \models \{\varphi\}\ S\ \{\varphi'\}\ [X]$ *then* $\Delta \models \{\varphi\}\ S_1\ \{\varphi'\}\ [X]$.

7 Discussion

In recent language-based work on declassification, policy is expressed in part by labeling variables and fields with security types. One approach uses security levels for the security types, with the usual interpretation, together with some special program construct to designate where and/or how the default policy is overridden [29, 20]. Another approach uses richer security types that designate, e.g., what may be released [18] or conditions under which release is allowed [8].

Our approach is to begin by labeling the program's interfaces using lattice levels as usual. Many methods in a program are likely to satisfy noninterference with respect to these levels and even to be accepted by a security type checker. For variables[3] subject to a policy allowing declassification, this policy is expressed by specifications for the method bodies that can access the variables. Instead of typechecking them, their specification is proved.

For policies concerning *what* is released, what is needed is a precondition $E\ltimes$ where E is an expression for the value to be released. Such a policy may typically be imposed as a pre-post specification for a method. (Note that this avoids the problem that, if the declassification policy is written as an annotation on E as it occurs in code, there is a potential discrepancy between the value of E at time of release and its initial value to which the policy refers [29].)

For policies concerning *where* in the code—in particular, via *which channels*—downgrading occurs, we impose specifications on the relevant subprograms, e.g., of the form $\{\varphi_1\}\ getBids\ \{\varphi_2\}$ and $\{\varphi_3\}\ announceWinner\ \{\varphi_4\}$.[4] Though the notation is different, it does not seem so different in practical terms from what appears in specialized calculi such as [20, 29, 19]. Practical use of our approach would involve marking the code segments subject to declassification specifications and including preconditions to be imposed in the ordinary way on callers/context.

In some sense, the proposal closest to ours is that of Chong and Myers [8] who address "where" and "when" policies by labeling variables with security types that refer to a sequence of conditions and levels through which downgrading is allowed to pass.[5] This seems more general than what we have described, in that it is not coupled to the implementation. But one could imagine requiring that any code with access to the variable in question is subject to one or more pre-post specifications. Note also that Chong and Myers propose to enforce their policies using a type system augmented with some means to reason about "conditions". The means most well understood and supported by tools is assertion-based verification. Typical conditions might refer to log entries or other data already present in the program state. To refer to event occurrences, history variables would be used. Use of such ghost variables/fields depends on properly annotating the program with ghost updates of course, e.g., to correctly track what events have occurred, but this is standard verification methodology.

The "who" dimension of policies is also important, e.g., an official may be authorized to release confidential records in a financial database to law enforcement officials exhibiting a subpoena [19]. The fact that authentication checks have been made is likely to be encoded in the program state and if necessary it too can be modeled with ghost variables. There is some similarity with Chong and Myers' password example where release depends on a condition that means

[3] And fields. Object references, like other values, are not directly assigned a security level; their security is in terms of the variables from which they can be reached and the levels of their fields [2].

[4] Though these subprograms occur as a sequence $getBids; announceWinner$ in the auction program, our particular φ_2 does not imply φ_3 and thus these specifications do not compose to prove $\{\varphi_1\}\ getBids; announceWinner\ \{\varphi_4\}$. Indeed, the latter triple is neither valid nor a desired property of the auction.

[5] It is not clear to the authors the extent to which declassification levels per se are an artifice in treatments of declassification. The real-world policies seem to involve declassification being controlled by suitable authorities and via specific channels and operations, rather than intrinsically meaningful security levels.

"only trusted code is running". The stack inspection mechanism is also intended to be used to check for such conditions. As in many such examples, the idea is that certain programs have been validated by various means, with respect to policies that are not formalized as part of the declassification policy per se.

One shortcoming of logic-based approaches is that there does not seem to be a direct way to handle incomparable levels in a security lattice. In the auction example, instead of having incomparable levels for Alice's and Bob's bids, thereby specifying absence of flow in either direction, we need two Hoare triples. (We refrain from formalizing conjunction of triples; the straightforward generalization can be found in many places [26, 17].) But it is not difficult to imagine syntactic sugar; indeed, one use of our approach is as a unifying foundation for lightweight type-oriented notations.

An advantage of logic-based approaches is that in some cases, verification can be fully automated, e.g., if the specification consists only of independences and regions [1]. One can fall back on interactive verification, or at least programmer-supplied invariants etc, in the harder cases.

Other related work The survey of Sabelfeld and Myers [28] is slightly dated but still an excellent survey of language based information flow. For declassification, the analysis by Sabelfeld and Sands [30] touches on many aspects of and works on declassification beyond what we mentioned in earlier sections. They identify a number of informal principles for declassification. Our approach seems to live up to them all. For example, their principle of semantic consistency was discussed at the end of Section 6. On may freely transform $getBids$ or $announceWinner$, though the sequence $getBids$; $announceWinner$ can't be transformed in a way that loses the semicolon to which policy is attached.

An interesting recent proposal uses security types of the form $\overline{\sigma} \Rightarrow k$ where k is a security level and $\overline{\sigma}$ is a list of "flow locks" [7]. Essentially, a flow lock is a boolean ghost variable and lists are interpreted conjunctively. The two program constructs, open and close, used to manipulate locks are essentially assignments of true and false to the lock. If x is labeled with k and $\sigma \Rightarrow k'$ then x is treated as having level k but also level k' just when lock σ is true. The paper sketches how "where", "when", and "who" policies can be expressed, much in the way we use assertions like $\neg\theta \dot{\vee} x\ltimes$. The semantics, which is only sketched in the paper, is a form of resetting. Thus the use of locks for these policies is very much in accord with our proposal. It seems attractive to have a simple notion of type associated with ghosts, but in practice one would want to specify the connection between locks and other events, evidence, etc, and this would likely appear similar to specifications in our style. The restricted use of ghosts together with types offers the potential for automated checking. On the other hand, as the paper shows through a number of examples, the mechanism interacts with features of the language (an ML-like language with references) in ways that seem inscrutable.

The Jif tool offers, for sequential Java, rich information policies including declassification and much more [22].

Benton [6] gives a relational Hoare logic, that is, a logic similar to ours but in which runs of two different programs are compared. The logic is shown to encompass noninterference as a special case, although the main focus is on reasoning about program transformations based on static analyses. Only simple imperative programs are considered, with no heap. Yang [34] adapts separation logic to a similar logic for imperative programs acting on the heap. Giacobazzi and Mastroeni [14] use abstract interpretation techniques to provide a setting for noninterference that encompasses declassification. The connection between their work and our logic remains to be explored.

Acknowledgements: To Isabella Mastroeni, Alejandro Russo and Andrei Sabelfeld for early stage discussions.

References

[1] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *POPL*, 2006. Extended version available as KSU CIS-TR-2005-1.

[2] 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.

[3] G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In *VMCAI*, 2004.

[4] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW*, 2004.

[5] G. Barthe, D. A. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for java. In *27th IEEE Symposium on Security and Privacy*, May 2006. To appear.

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

[7] N. Broberg and D. Sands. Flow locks. In *European Symposium on Programming (ESOP)*, 2006.

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

[9] E. S. Cohen. Information transmission in sequential programs. In A. K. J. Richard A. DeMillo, David P. Dobkin and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[10] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *SPC*, 2005.

[11] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[12] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[13] G. Dufay, A. Felty, and S. Matwin. Privacy-sensitive information flow with JML. In *CADE*, 2005.
[14] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *POPL*, 2004.
[15] L. Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.
[16] D. Gries. Data refinement and the tranform. In M. Broy, editor, *Program Design Calculi*. Springer, 1993. International Summer School at Marktoberdorf.
[17] 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.
[18] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *POPL*, 2005.
[19] H. Mantel and D. Sands. Controlled declassification based on intransitive noninterference. In *APLAS*, 2004.
[20] A. A. Matos and G. Boudol. On declassification and the non-disclosure policy. In *CSFW*, pages 226–240, 2005.
[21] C. Morgan. Auxiliary variables in data refinement. *Inf. Process. Lett.*, 29(6):293–296, 1988.
[22] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, 1999.
[23] D. A. Naumann. From coupling relations to mated invariants for secure information flow and data abstraction. 2005.
[24] D. A. Naumann. Verifying a secure information flow analyzer. In *TPHOLS*, 2005.
[25] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, Jan. 2003.
[26] J. C. Reynolds. *The Craft of Programming*. Prentice-Hall, 1981.
[27] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press, 2002.
[28] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
[29] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *ISSS*, 2004.
[30] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW*, 2005.
[31] J. Smans, B. Jacobs, and F. Piessens. Static verification of code access security policy compliance of .NET applications. In *.NET Technologies*, 2004.
[32] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS*, 2005.
[33] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
[34] H. Yang. Relational separation logic. *Theoretical Comput. Sci.*, 2004. To appear.