JFP **15** (2): 131–177, 2005. © 2005 Cambridge University Press
DOI: 10.1017/S0956796804005453 Printed in the United Kingdom

Stack-based access control and secure information flow

ANINDYA BANERJEE[*]
Department of Computing and Information Sciences, Kansas State University, Manhattan, KS 66506, USA
(*e-mail:* ab@cis.ksu.edu)

DAVID A. NAUMANN[†]
Department of Computer Science, Stevens Institute of Technology, Hoboken, NJ 07030, USA
(*e-mail:* naumann@cs.stevens-tech.edu)

Abstract

Access control mechanisms are often used with the intent of enforcing confidentiality and integrity policies, but few rigorous connections have been made between information flow and runtime access control. The Java virtual machine and the .NET runtime system provide a dynamic access control mechanism in which permissions are granted to program units and a runtime mechanism checks permissions of code in the calling chain. We investigate a design pattern by which this mechanism can be used to achieve confidentiality and integrity goals: a single interface serves callers of more than one security level and dynamic access control prevents release of high information to low callers. Programs fitting this pattern would be rejected by previous flow analyses. We give a static analysis that admits them, using permission-dependent security types. The analysis is given for a class-based object-oriented language with features including inheritance, dynamic binding, dynamically allocated mutable objects, type casts and recursive types. The analysis is shown to ensure a noninterference property formalizing confidentiality and integrity.

Capsule Review

Tracking information flow is an appealing way to enforce strong security, but practical application of this approach encounters various difficulties. This article tackles the important practical problems of dealing with both object-oriented programming and access control. It defines a security-typed model of the Java programming language, including its access control mechanism of stack inspection. A security type system is given that controls information flow in this language, provably enforcing noninterference. While some prior work has explored type systems for information flow in the presence of objects and access control, this work is particularly useful because it obtains a noninterference result for a rich programming language with a realistic access control mechanism. This result requires novel proof techniques because of the possible presence of inheritance and recursive classes and methods. The type system that connects access control and information flow permits relatively precise

[*] Supported in part by NSF grants CCR-0209205 and NSF Career award CCR-0296182.
[†] Supported in part by the National Science Foundation under grants CCR-0208984 and CCR-ITR-0326540, the Office of Naval Research under grant N00014-01-1-0837, and the NJ Commission on Science and Technology.

characterization of information, which is possible because it allows the information flow labels of results to depend on permissions held by the program. Notably, this dependency relates information flow to negative permissions, which is needed for reasoning in the presence of stack inspection.

1 Introduction

Consider the following program, to be used by clerical staff to prepare a billing statement for a medical patient. It returns the catenation of the patient's name and address, italicizing the string if the patient is HIV positive.

```
String getPatientAddress(int patientID) { PatientRecord r := database[patientID];
                                          result := r.name ++ r.address;
                                          if r.hiv then result.font := "italic"   }
```

This would violate typical policies about privacy of sensitive medical information in patient records. Although the value r.hiv is not output directly, it is revealed to an observer who knows that not all outputs are italicized. The violation in this particular example is likely to be immediately apparent upon inspection, but that is only because the example is tiny. This paper is concerned with specification and formal justification of tools for automated static checking for information flow policies. The main novelty is rules to account for calls that may return high security information, but which use access checks to ensure that only low security information is returned unless the caller has been given access.

Besides being a goal in itself, confidentiality is often an ingredient in more complicated policies and for the mechanisms that enforce them. A lattice of confidentiality levels is given – for expository purposes it suffices to consider two levels with $L \leqslant H$. Input and output channels are labeled with these levels. The policy so expressed is that information visible on L output channels is not influenced by information on H input channels (Bell & LaPadula, 1973). This property is a form of dependency (Abadi *et al.*, 1999) and the absence of dependency of L on H can also be read as a form of integrity: Licensed data is not influenced by Hacked data. For simplicity we focus on confidentiality in the exposition but sometimes use the neutral term *flow policy*.

In this paper, the input/output channels are procedure arguments/results and state changes for heap objects. As we consider object-oriented programs in the sequel, we use the term *method* for procedure. Here is a labeling of the example.

$\mathbf{string}_L$ getPatientAddress($\mathbf{int}_L$ patientID)

An attractive way to specify and validate tools for checking that flow policy is enforced is to use decorated types within the program. A rule can easily be formulated to preclude, e.g. the command

```
result := r.name ++ r.address ++ r.hiv
```

if the hiv field is labeled H. The rule requires that the level of an assigned variable must be at least the level of the expression; and the level of an expression is the least upper bound of the levels of its constituents (Denning & Denning, 1977).

Labeling variables and imposing such a rule is sometimes called “access control” in the literature (Rushby, 1992). It is akin to, but more restrictive than, static visibility controls like private fields, for which the term “access modifier” is used. We refrain from using this terminology, as it invites confusion with runtime access control mechanisms.

The flow of information in the first example above is not by a direct assignment, but the control state can be treated as an implicit variable, the level of which is also tracked. This makes it feasible to give syntax-directed rules that prevent such flows as well (Denning & Denning, 1977).

To formalize the policy that no information flows from H to L, Goguen and Meseguer (1982) proposed a property called *noninterference* described in terms of a certain kind of simulation relation. Two program states are “equivalent for L”, or *L-indistinguishable*, if they agree for L variables and fields. The noninterference property is this: for any pair of initial states that are L-indistinguishable, the two corresponding runs of the program yield final states that are L-indistinguishable.

Volpano *et al.* (1996; 1997) formalized rules for static analysis of information flow using the techniques of type theory; their *security types* formalize the labeling discussed above. They proved that programs accepted by the typing rules have the noninterference property.

There has been considerable research exploring these ideas, but they have not seen much use in practice. There are several difficulties to be overcome for practical use. This paper makes progress in addressing two of the difficulties: the complexity of conventional languages and the essential use of runtime access control. A security typing system is given for an expressive, class-based language including runtime access control and the system is shown to ensure noninterference.

We proceed to discuss these and other difficulties and to outline our contribution in more detail. Sabelfeld & Myers (2003) give a comprehensive review of the literature and open challenges; closely related work is also discussed in section 9.

Excessive restriction. Noninterference, as commonly formulated, is too strong a property for some situations. For example, the message “password not valid” from a login program reveals some information about a password, but not much. For such situations, and for encryption, a quantitative measure seems appropriate. A potential shortcoming is that an associated static analysis could be hard to explain to programmers or to implement efficiently. Another practical issue is the need for selective declassification or downgrading. Work on these difficult problems is reviewed by Sabelfeld & Myers (2003). We address neither problem, but note that the strong noninterference property is appropriate for many situations. For example, encrypting a message reveals some information about the message to observers of the ciphertext (and it reveals the entire message to possessors of the deciphering key). It should reveal nothing about the passphrase for the user’s key management application, nor should it reveal information about the message through channels other than the ciphertext output.

Covert channels. While a straightforward data dependency analysis can detect flows of data from H to L, the noninterference property can be violated by more subtle flows. The first example above illustrates information propagation by control flow. In reactive and concurrent programs, intermediate states are visible and synchronization can be used, deliberately or accidentally, to leak information.

Memory allocation can also be an observable channel of information flow, if it is possible to detect that memory has been exhausted or to perform arithmetic operations on addresses. Even more subtly, timing and the consumption of power can reveal information – but such covert channels are tricky to exploit and in any event they do not occur at the level of abstraction of source or object code.

Termination is another covert channel, albeit one that can leak only one bit per program run. Termination-sensitive noninterference requires that, from two L-indistinguishable initial states, the corresponding runs of the program both terminate or both diverge. *Termination-insensitive* noninterference requires only that if both runs terminate then the resulting states are L-indistinguishable.

The standard rule to preclude leaks via termination requires all loop guards to be L. We are not aware of any work that deals with static analysis for termination-sensitive noninterference in the presence of general recursive procedures. One could approach the problem by determining which guards of conditionals can influence termination of such a procedure; but for sufficient flexibility in practice it could be better to augment typing rules with verification conditions for termination. This merits study in a simple language before it is combined with the features of a full object-oriented language.

A tool which checks source code with respect to a specified information flow policy can be useful for debugging and for guarding against deployment of faulty or malicious code, even if it does not handle all covert channels. We confine attention to checking termination-insensitive noninterference for source code in a sequential object-oriented language with unbounded memory, without pointer arithmetic, and with no restriction on recursion.

Trustworthiness of the checker. Covert channels such as power consumption subvert the assumptions on which high level abstractions are based. Such assumptions are also subverted by bugs in the implementation of compilers, runtime support, and the underlying operating systems and hardware. For a static checking tool used to enforce a confidentiality policy, trustworthiness depends on soundness of the rules and correctness of their implementation. Languages in widespread use are considerably more complicated than the simple ones for which information flow analysis has been studied – even ordinary type soundness for such languages has been problematic in practice and in theory.

The first of our contributions is to extend the security typing approach to a relatively rich class-based, imperative, object-oriented language. We give a self-contained proof of termination-insensitive noninterference for programs that follow our security typing rules. A compositional semantics is used and the main lemma is proved by induction on program structure.

Real programs use access control. Information flow checking has seen little use in practice, but various runtime access control mechanisms are widely used. These include operating system mechanisms at the level of processes, files, and devices, as well as application-specific mechanisms such as in database systems. Code-based access control has come into wide use, to cope with late-bound components and various forms of mobile code such as applets (Gong, 1999; LaMacchia *et al.*, 2003). Using object-oriented design patterns, these can involve fine-grained interactions, within a single address space, among programs having differing levels of trust.

Access control mechanisms directly enforce access policies, that is, they control actions including those which release information, but cannot fully control propagation of information. Technically, information flow is not a property of computations but of sets of computations (McLean, 1994; Volpano, 1999). Access mechanisms enforce *safety*: properties of computations that are finitely refutable (Erlingsson & Schneider, 1999). Nonetheless, many uses of access control are intended to enforce flow policy.

Our second major contribution is to refine the formulation of information flow policy by making it dependent on access permissions. For example, a method that prints information about a medical patient can be allowed to return H information for callers with specified permissions, while returning L information to others. Permissions may be absent due to misconfiguration, programming error, or because a single interface is intended to provide useful functionality to callers with differing access credentials.

We investigate permission-dependent information flow in the setting of the access control mechanism of Java (Gong, 1999) and the .NET Common Language Runtime (Gough, 2001), which aims to protect trusted system code, e.g. a browser, from untrusted mobile code. The principals that are granted permissions in an access policy are programs rather than, say, processes or users as in operating system security. The mechanism is sometimes called "stack inspection" in reference to one implementation. More fundamentally it is stack based in that access decisions depend on permissions of code in the calling chain. Put differently, the credentials on which access decisions are based, i.e. the security context, is passed as an implicit parameter in a method call (but is not updated as an effect of the method call).

A considerable amount of commercial software is being developed using stack-based access control, because it is part of the widely-deployed Java and .NET virtual machines. But it is open to question whether this is the best mechanism for its purposes; it is not well understood what security properties are achievable and what are good design patterns for using the mechanism (Wallach *et al.*, 2000; Fournet & Gordon, 2002; Abadi & Fournet, 2003). Our aim is neither to propose a new mechanism nor to argue in favor of this one. Rather, we make sense of a common pattern of usage and demonstrate that it can achieve a strong security goal under statically checkable conditions.

We formulate a type-based static analysis using security types that are dependent on permissions. The analysis tracks permissions and thereby distinguishes certain information flows that are guarded by permissions. We believe the technique can be adapted to other access mechanisms that are amenable to static analysis of access decisions. But this is left to future work.

Security typing does not include level polymorphism, in which the level of results depends on the level of inputs; this is orthogonal to flows that depend on permission so for simplicity it is omitted.

The static analysis is defined with respect to given access policy and given flow policy. The main result of the paper is that *safe programs*, i.e. those allowed by our security typing rules, are noninterfering in the sense of the given flow policy. Limited experience with examples suggests that permission-dependent typing may be a suitable basis for the design of practical interface specifications and tools.

Outline of the paper. In section 2 we give a streamlined description of the relevant features of the access control mechanism. We also review the use of labeled types to specify confidentiality in imperative (Volpano *et al.*, 1996) and object-oriented (Myers, 1999; Banerjee & Naumann, 2002c) programs.

Section 3 gives an example to illustrate the main idea of the paper: giving several information-flow types to a method, dependent on the permissions that may be enabled by callers.

Section 4 formalizes the programming language, giving typing rules and a compositional semantics which facilitates proof by structural induction on program syntax. At the cost of notation more complex than the minimum necessary to illustrate our main idea, we consider a sequential object-oriented language with pointers and mutable state, public fields, dynamic binding and inheritance, recursive classes, casts and type tests, and recursive methods. This shows that the ideas on which we build scale to a realistic language and lays a broader foundation for future work, e.g. permissions and protection domains as first-class objects as in Java and C♯.

Section 5 formalizes the static analysis using syntax-directed rules which admit the examples of section 3. Further examples are given in section 6. Section 7 proves basic results about the analysis, which are used in section 8 to prove the main result: the static analysis ensures noninterference. Section 9 discusses related and future work.

2 Background: access control and information flow

Access control by stack inspection. In the access control mechanism of Java (Gong, 1999) and the .NET Common Language Runtime (Gough, 2001), each class C has a set $Auth\,C$ of permissions statically associated with it; this comprises a local access control policy. A typical policy grants few permissions to code from remote sites and many to code residing on the local disk. The most interesting policies concern trusted remote sites: Code which has been cryptographically authenticated as originating at a trusted site may be granted particular permissions.

As an example of the use of permissions for integrity, a user program might have the permission p for changing passwords but not the permission w for directly writing the password file. There is an operation for checking whether a permission is authorized for the classes of all code with frames (activation records) on the current call stack. If this fails to be the case, a catchable exception is thrown. This mechanism has no intrinsic connection with particular data objects or events; it is up to the programmer to ensure that writes to the password file are guarded by checks of permission w.

To model the mechanism in a simplified language, we follow previous work (Fournet & Gordon, 2002) and refrain from modeling exceptions. Instead, we consider a construct, **test** p **then** S_1 **else** S_2, which checks for permission p, executing S_1 if the check succeeds and S_2 if it fails. A simple check can be written **test** p **then skip else abort**. To model the case where an exception is thrown and caught, the **else** branch can return some value that indicates to the caller that a check has failed. An alternative formulation is to treat **test** as a primitive boolean function, but our static analysis would still single out conditionals using **test** for special treatment.

The description above is over-simplified, in that tests do not simply depend on static permissions for all code on the stack. In keeping with the principle of least privilege, permissions must be explicitly enabled. In Java this is provided as an operation, doPrivileged, which uses a callback object with the effect of lexical scoping. We model it by a construct, **enable** p **in** S.

To describe these constructs more precisely, it is convenient to use the "eager" formulation (Gong, 1999; Wallach *et al.*, 2000) in which the current security context is maintained as an implicit parameter, Q, as opposed to lazily constructing the context from the stack when checks are performed. Like explicit parameters, Q can be updated in a method body. The effect of the command **enable** p **in** S is to add p to Q only if p is statically authorized for the class in which the command occurs; otherwise it has no effect. (It might be less misleading to use the term "try-to-enable" or "assert" but these have shortcomings too.) The **test** p **then** S_1 **else** S_2 simply branches on whether $p \in Q$. Finally, a method call $e.m(\ldots)$ passes to m the set $Q \cap Auth\, C$ where C is the class of the code for m. The method call has no effect on the caller's Q. Note that it is not the class, say D, of the target object e that determines permissions, nor the static type of e, but rather the class C of the dynamically dispatched code for m. By inheritance, D may be a proper subclass of C and have different static permissions.

Here are two example classes in our syntax. Assume that the local access policy, $Auth$, designates $Auth$ User $= \{\mathsf{p}\}$ and $Auth$ Sys $= \{\mathsf{p}, \mathsf{w}\}$. The objective is to disallow direct writing of the password file by code in class User. We assume that diskWrite is a low-level operation protected by some other mechanism, e.g. it is a private method or privileged operating system call.

```
class Sys {
  unit writepass(String x) {
    test w then diskWrite(x, "passfile") else abort}
  unit passwd(String x) {
    test p then enable w in writepass(x) else abort }}
class User {
  Sys s . . . /* field initialization omitted */
  unit use() { enable p in s.passwd("mypass") }
  unit try() { enable w in s.writepass("mypass") }}
```

In a main program with no permissions enabled, invoking method use gives it permission set ∅. Then use enables p and its permission set becomes {p}. This is implicitly passed to passwd, in which **test** p succeeds and permission w gets enabled. That is, the permission set in passwd is changed to {p, w}, and this set is passed to writepass. It checks w successfully and calls diskWrite.

Invocation of method try results in failure. Consider, for example, the case where try is invoked from a main program that has successfully enabled {p, w}. Because only p is statically enabled for class User, the initial permission set for try is {p}. For the same reason, **enable** w has no effect in try and so the permission set passsed to writepass is {p}. Thus the test in writepass fails and the integrity of the password file is maintained.

The **enable** construct allows a method to, in effect, temporarily grant a permission to methods in the call chain to it and thus perform sensitive operations on behalf of untrusted callers. It cannot, however, grant permissions to code that it invokes. Trusted code like a browser can invoke untrusted plug-ins without risk of giving them unintended permissions. In itself, the access mechanism does not prevent harm from misuse of results from calls to untrusted code, as emphasized by Abadi and Fournet (2003). In Section 6 we show how information flow analysis can help address this problem.

Checking information flow using security types. The idea developed by Volpano *et al.* (1996), based on earlier work (Denning, 1976; Denning & Denning, 1977; Ørbæk & Palsberg, 1997), is to label not only inputs and outputs but also variables and parameters by security types, for example replacing a variable declaration $x : T$ by $x : (T, \kappa)$ where κ is the security level. Syntax-directed typing rules specify conditions that ensure secure flow. Overt flows, like an assignment of an H-variable to an L-variable, are disallowed by the typing rules for assignment, argument passing, etc. To preclude covert flow via control flow, commands are given types $com\,\kappa$ with the meaning that all assigned variables have at least level κ. For a conditional, **if** e **then** S_1 **else** S_2, with e high, both S_1 and S_2 are required to have type $com\,H$.

In an object-oriented language, covert flow also happens via dynamically dispatched method call. Moreover, one must worry about the possibility of observing differing behavior of the allocator if objects allocated conditionally are accessible. In the sequel we use command types of the form $(com\,\kappa_1, \kappa_2)$ where κ_1 is a lower bound on level of assigned variables and κ_2 is a lower bound on the heap effect, i.e. assignments to object fields.

Annotated arrow types can be used for modular checking in the case of procedures or functions (Volpano & Smith, 1997; Abadi *et al.*, 1999). For example, Banerjee &

Naumann (2002c) use a notation like $(T, \kappa_1) \multimap \langle \kappa_2 \rangle \rightarrowtail (U, \kappa_3)$ for a method with one parameter of type T and result of type U. This annotated type is intended to express that if the argument is at most κ_1 then the heap effect is at least κ_2 and result level at most κ_3. A method body is checked with respect to its security type, which is used as an assumption for checking method calls.

The type of the target object, self, deserves special attention but we defer discussion until later, as the issue does not obtrude in section 3. In fact, even the heap effect can be dropped for the example discussed in section 3.

Type annotations are a perspicuous way to specify and reason about policy and static analysis. They may be useful in practice for specifying public interfaces, but it is important to keep in mind that policy is decided at deployment sites. An application programming interface might suggest suitable flow and access policies, but the policies imposed at deployment sites may differ. For this reason, among others, the automated inference of security types is useful (see section 9).

An access control mechanism may itself be a channel for covert flows. For the mechanism in this paper, the set of currently enabled permissions can be seen as an implicit variable which can be tested. But values of this implicit variable are manipulated in a very restricted way that reflects only control flow information. Our noninterference result confirms that straightforward security typing rules suffice to control the flows introduced by **test** and **enable**. What is more interesting is the use of **test** to achieve information flow goals.

3 Using access control for confidentiality

Sometimes a permission guards an action that is essential to a program's purpose and a failed permission check is seen as a catastrophic error in programming or security configuration. But many programs are intended to operate in more than one context. Gong (1999) gives the example of a Java program used both in a downloaded applet and in a locally installed application; it needs to create an output stream to store temporary results. To do so it attempts to open a local file, but if that results in a security exception the program attempts to open a network connection to its originating site. Because access policy is configured at deployment sites, the need for graceful error recovery is another reason it can be useful to specify the behavior of a program both when expected permissions are granted and when they are not.

For expository purposes we consider a toy system composed of components, some of which are from sources not fully trusted. Class Kern is a trusted system class and Comp1 is from a less trusted source. To flesh out the example, one could imagine that the system is used in a medical clinic. Class Comp1 could provide financial applications. Another class, say Comp2, could provide support for distributing a newsletter to patients. Personal financial information may be made available to Comp1 but not to Comp2; personal medical information should flow to neither of them. For this scenario, a realistic flow policy would involve at least a four-element lattice with incomparable levels *finance* and *newsletter* in addition to the maximum

H and minimum *L*. The code would also be far more complicated. But to sketch our ideas it suffices to consider *L* and *H* and a very simple program structure.

If the confidentiality goal is that information confidential to Kern is not leaked to Comp1, the security lattice might correspond to code sources and thus be correlated with permissions. But we do not want to presuppose a connection between information flow policy and the access control mechanism. Not all information manipulated by Kern is confidential.

For these examples we consider the set $Permissions = \{sys, stat, other\}$. The intention is that *sys* guards a method getHinfo of Kern that returns *H* information, and *stat* guards a method getStatus that can be used by trusted callers manipulating *H* information and also by untrusted ones manipulating *L*.

Suppose the access policy, *Auth*, is the following mapping from classnames to permission sets:

class	permissions
Comp1	*other*
Comp2	*stat*, *other*
Kern	*stat*, *sys*

Class Kern is as follows, where the intended flow policy is indicated in comments

```
class Kern extends Object {
  String Hinfo; // H
  String Linfo; // L
  String getHinfo() { // type () → H
    test sys then result := self.Hinfo else abort }
  String getStatus() { // type () → ?? (see below for discussion)
    test stat
    then enable sys in result := self.getHinfo()
    else result := self.Linfo }
. . . "other methods that manipulate Linfo and Hinfo"}
```

Class Comp1 has access to an instance of Kern. It has a method status returning the catenation of application-specific data v with the status from the kernel. This exemplifies the use of getStatus by untrusted callers.

```
class Comp1 extends Object {
  Kern k; // L
  String v; // L
  String status() { // type () → L
    result := self.v ++ k.getStatus() }
. . .}
```

Execution of method status proceeds as follows: To evaluate the catenation, invoke k.getStatus() which tests *stat*. The test fails, as *stat* has not been enabled, so getStatus returns k.Linfo. This is compatible with the policy that status has *L* output.

To flesh out the example, the reader is invited to contemplate a method of Kern for retrieving the record for a particular medical patient. Access control could be used to decide whether to include payment information (for a partly trusted medical application), name and address (for a partly trusted newsletter application), or nothing at all. The point is that Kern should not have to provide separate, redundant interfaces for the two components just to fit a type-based specification of flow policy.

For an information flow analysis to allow Comp1.status, it is necessary to take into account the **test** in getStatus and also the access policy for Comp1. Otherwise, a sound analysis of getStatus would say that it can return H which violates the flow policy for Comp1.status.

Comp1 could try to gain access to Hinfo as follows:

```
String status2() { // type () → L
  enable stat in result := self.v ++ k.getStatus() }
```

But because *stat* is not authorized for Comp1, the **enable** has no effect and the policy is not violated.

Code in classes Comp1 and Comp2 cannot successfully invoke getHinfo directly, because in *Auth* neither class is granted permission *sys*, without which method getHinfo aborts. An attempted **enable** *sys* does not help.

Our example access policy does, however, grant permission *stat* to Comp2. The flow policy also indicates a degree of trust in that method statusH is allowed to return H.

```
class Comp2 {
  Kern k;
  String statusH() { // type () → H
    enable stat in result := k.getStatus() }}
```

Method statusH succeeds in obtaining Hinfo: in k.getStatus(), permission *stat* is enabled and is authorized for Kern and for Comp2. This is consistent with the policy allowing H result from statusH; it would not be consistent with L result.

As indicated by "??" in class Kern, the question is how to type method getStatus so we can formulate a modular check that admits the valid examples while rejecting code (or access policy) that violates the information flow policy. In particular, all of the example code above should be allowed.

Volpano & Smith (1997), among others (Pottier & Simonet, 2003), consider procedure typings that are polymorphic in levels, to handle cases where level-α inputs yields level-α outputs. For example, the catenation operator ++ for strings could have type $\alpha \times \beta \to \alpha \sqcup \beta$. This does not handle our examples, because the result level for getStatus depends not on the level of inputs but on the enabled permissions. More to the point: although permissions can be seen as an implicit parameter, the dependence is not on information level but rather the *value* of that implicit parameter.

The justification of the examples hinges on reasoning about the behavior of the **test** in getStatus. This behavior depends on what permissions are enabled by the caller. This leads to our proposal: Methods are given types that depend on permissions authorized for the caller. More precisely, *types designate permissions* *that must not have been enabled by the caller*. For the moment we leave aside heap effects and a level for self. The meaning of a type $() \prec\!\langle P\rangle\!\succ \kappa$ is as follows: if invoked by a caller which cannot enable any of the permissions in P, the method returns a result of at most level κ. We annotate each method with one or more such typing, and check that the method body respects all of them.

Method getStatus is given types $()\prec\!\langle\emptyset\rangle\!\succ H$ and $()\prec\!\langle\{stat\}\rangle\!\succ L$. The call from Comp1.status can be typechecked with respect to $()\prec\!\langle\{stat\}\rangle\!\succ L$, because *stat* is not in *Auth*(Comp1). The typechecking rule does not allow the call from Comp2.statusH to be checked using $()\prec\!\langle\{stat\}\rangle\!\succ L$, because *stat* is authorized for Comp2. It can be checked using $()\prec\!\langle\emptyset\rangle\!\succ H$.

Consider this additional method for Comp2:

```
String statusH2() { result := k.getStatus() }
```

It can be given type $()\prec\!\langle\emptyset\rangle\!\succ H$ just like statusH. Although *stat* is not enabled by statusH2, it could be enabled by a caller of statusH2. So it is not sound to check the body of statusH2 using the type getStatus: $()\prec\!\langle\{stat\}\rangle\!\succ L$ unless we disallow such callers by typing statusH2 as $()\prec\!\langle\{stat\}\rangle\!\succ L$. That is, statusH2 does not have type $()\prec\!\langle\emptyset\rangle\!\succ L$.

We have discussed method types of particular interest, but others are also sound. For example, getStatus satisfies the types $()\prec\!\langle\{stat, sys\}\rangle\!\succ L$ and $()\prec\!\langle\{stat\}\rangle\!\succ H$ and getHinfo satisfies $()\prec\!\langle\{sys\}\rangle\!\succ L$. In the sequel we define a suitable notion of subtyping and show that these examples follow by considerations of subsumption. In fact security types form a complete lattice; for example, the greatest lower bound of $()\prec\!\langle\emptyset\rangle\!\succ H$ and $()\prec\!\langle\{stat\}\rangle\!\succ L$ is $()\prec\!\langle\emptyset\rangle\!\succ L$.

Note that the type $()\prec\!\langle\emptyset\rangle\!\succ L$ is not satisfied by getStatus and thus there is not a single type that expresses the desired properties of getStatus. The implication is that the most concise interface specifications are those that specify the security of a method using a set of types that are minimal with respect to subtyping. But for our purposes in this paper we can consider an arbitrary collection of types.

To deal with dynamic binding in a modular way, we require that an overriding declaration must be checked with the same set of typings as the method it overrides. The permissions involved need not be authorized for the class in which the declaration occurs. A subclass that overrides a method may have different permissions than its superclass. This is discussed further in section 5, where the security typing rules are defined using judgements $\Delta; P \vdash S$ that characterize the behavior of S under the assumption that permissions P are not initially enabled.

It is possible, and perhaps more natural at first glance, to formulate a static analysis using a typing arrow annotated with an upper bound on the caller's permissions, so for example getStatus would have a type $()\prec\!\langle P\rangle\!\succ L$ with $stat \notin P$, where the caller's permissions must be contained in P. But this is not modular; the caller may well have permissions, like *other*, not relevant to the callee.

On the other hand, one can envision a kind of dual to the design pattern we consider: A method may release H information just if a certain permission is absent. Our analysis does not handle this pattern and we are unaware of motivating examples; permission checks usually guard sensitive actions such as the release of

T	$::=$	$\mathbf{bool} \mid \mathbf{unit} \mid C$	data type, where C ranges over class names
CL	$::=$	$\mathbf{class}\ C\ \mathbf{extends}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\}$	class with public fields $\bar{f}$, public methods $\bar{M}$
M	$::=$	$T\ m(\bar{T}\ \bar{x})\ \{S\}$	method with result type T, parameters $\bar{T}$
S	$::=$	$x := e \mid \mathbf{if}\ e\ \mathbf{then}\ S\ \mathbf{else}\ S \mid S;\ S$	assign to variable; conditional; sequence
	$\mid$	$T\ x := e\ \mathbf{in}\ S \mid x := e.m(\bar{e})$	local variable block; method call
	$\mid$	$e.f := e \mid x := \mathbf{new}\ C$	assign to field; construct object
	$\mid$	$\mathbf{enable}\ P\ \mathbf{in}\ S$	enable permissions
	$\mid$	$\mathbf{test}\ P\ \mathbf{then}\ S\ \mathbf{else}\ S$	branch on permissions
e	$::=$	$x \mid \mathbf{null} \mid \mathbf{true} \mid \mathbf{false}$	variable, constant
	$\mid$	$e.f \mid e = e \mid e\ \mathbf{is}\ C \mid (C)\ e$	field access; equality test; type test; cast

Fig. 1. Grammar.

H information. A tempting idea is to downgrade or declassify information in the presence of certain permissions. But declassification violates noninterference and it is an open problem what is a good information flow property in the presence of declassification; for a recent attempt see Myers *et al.* (2004).

4 Language

This section formalizes the sequential class-based language for which our results are given. We assume given a finite set of *Permissions*. Finiteness is not essential, but it saves us from imposing explicit finiteness restrictions at various points in the syntax and semantics (or explaining why finiteness is not needed). The semantics is given with respect to a given function $Auth : ClassNames \to \mathscr{P}(Permissions)$ that specifies access policy. The aim of this section is to give an operationally transparent and mathematically convenient model of a conventional language. Information flow plays no role in the semantic definitions.

4.1 Syntax

The grammar is given by Figure 1. It is based on given sets of class names (with typical element C), field names (f), method names (m), and variable/parameter names x (including distinguished names "self" and "result" for the target object and return value). Identifiers like $\bar{T}$ with bars on top indicate finite lists, e.g. $\bar{T}\ \bar{f}$ stands for a list $\bar{f}$ of field names with corresponding types $\bar{T}$. We let P range over sets of permissions, without formalizing syntax for sets. We also assume there is a class Object with no fields or methods.

We omit loops but include unrestricted recursion. We omit super calls and constructor methods. It is straightforward to extend our work to super calls and to constructors that do not make method calls, using the semantics and proof technique of Banerjee & Naumann (2002a). We have not investigated constructors in the full generality found in conventional languages; there are difficulties even for type soundness, due to method invocations that leak partially initialized objects.

A complete program is given as a *class table*, CT, that associates each declared class name with its declaration. The typing rules make use of auxiliary notions that are defined in terms of CT, so the typing relation $\vdash$ depends on CT but this is elided in the notation. Because typing of each class is done in the context of the full table, methods can be recursive (mutually) and so can field types.

The subtyping relation $\leqslant$ on types is defined as follows. For base types, **bool** $\leqslant$ **bool** and **unit** $\leqslant$ **unit**. For classes C and D, we define $C \leqslant D$ iff either $C = D$ or the class declaration for C is **class** C **extends** B $\{\ldots\}$ for some $B \leqslant D$. The typing rules are syntax-directed: subsumption is built into the rules rather than appearing as a separate rule.

To define some auxiliary notations, which are implicitly dependent on CT, let

$$CT(C) = \textbf{class } C \textbf{ extends } D\ \{\bar{T}_1\ \bar{f};\ \bar{M}\}$$

and let M be in the list $\bar{M}$ of method declarations, with $M = T\ m(\bar{T}_2\ \bar{x})\{S\}$. We record the type by defining $mtype(m, C) = \bar{T}_2 \to T$ and let $pars(m, C) = \bar{x}$ record the parameter names. Let $super\, C = D$. For fields, we define $fields\, C = \bar{f} : \bar{T}_1 \cup fields\, D$ and assume $\bar{f}$ is disjoint from the names in $fields\, D$. The built-in class Object has no methods or fields. If m is inherited in C from B then $mtype(m, C)$ is defined to be $mtype(m, B)$, so that $mtype(m, C)$ is defined iff m is declared or inherited in C.

A class table is well formed if each of its method declarations is well formed according to the following rule.

$$\frac{\bar{x} : \bar{T}, \mathsf{self} : C, \mathsf{result} : T \vdash S \qquad mtype(m, super\, C) \text{ is undefined or equals } \bar{T} \to T \qquad pars(m, super\, C) \text{ is undefined or equals } \bar{x}}{C \vdash T\ m(\bar{T}\ \bar{x})\{S\}}$$

Figure 2 gives the other typing rules. A typing environment Γ is a finite function from variable names to types, written with the usual notation $x : T$. A judgement of the form $\Gamma \vdash e : T$ says that e has type T in the context of a method of class $\Gamma\mathsf{self}$, with parameters and local variables declared by Γ. A judgement $\Gamma \vdash S$ says that S is a command in the same context. Note that access policy has no influence on typing, though of course it does influence semantics.

4.2 Semantics

The state of a method in execution is comprised of a *heap* h, which is a finite partial function from locations to object states, and a *store* η, which assigns locations and primitive values to local variables and parameters. Every store of interest includes the distinguished variable self which points to the target object. A command denotes a function from initial state to either a final state or the error value $\perp$. States are self-contained in the sense that all locations in fields and in variables are in the domain of the heap.

For locations, we assume that a countable set Loc is given, along with a distinguished entity nil not in Loc. We treat object states as mappings from field names to values. To track the object's class, we assume given a function $loctype : Loc \to ClassNames$ such that there are infinitely many locations ℓ with $loctype\ \ell = C$, for each C. We assume that, like nil, the three primitive values it, $true$, and $false$ are not in Loc. The semantic definitions and results are given for an arbitrary allocator.

$$\Gamma \vdash x : \Gamma x \qquad\qquad \Gamma \vdash \mathbf{true} : \mathbf{bool} \qquad\qquad \Gamma \vdash \mathbf{null} : B$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 = e_2 : \mathbf{bool}} \qquad\qquad \frac{\Gamma \vdash e : C \qquad (f : T) \in \mathit{fields}\, C}{\Gamma \vdash e.f : T}$$

$$\frac{\Gamma \vdash e : D \qquad B \leqslant D}{\Gamma \vdash (B)\, e : B} \qquad\qquad \frac{\Gamma \vdash e : D \qquad B \leqslant D}{\Gamma \vdash e\ \mathbf{is}\ B : \mathbf{bool}}$$

$$\frac{\Gamma \vdash e : T \qquad T \leqslant \Gamma x \qquad x \neq \mathsf{self}}{\Gamma \vdash x := e}$$

$$\frac{\Gamma \vdash e_1 : C \qquad (f : T) \in \mathit{fields}\, C \qquad \Gamma \vdash e_2 : U \qquad U \leqslant T}{\Gamma \vdash e_1.f := e_2}$$

$$\frac{\Gamma \vdash e : D \qquad \mathit{mtype}(m, D) = \bar{T} \to T \qquad T \leqslant \Gamma x \qquad \Gamma \vdash \bar{e} : \bar{U} \qquad \bar{U} \leqslant \bar{T} \qquad x \neq \mathsf{self}}{\Gamma \vdash x := e.m(\bar{e})}$$

$$\frac{B \leqslant \Gamma x \qquad x \neq \mathsf{self}}{\Gamma \vdash x := \mathbf{new}\ B} \qquad\qquad \frac{\Gamma \vdash e : \mathbf{bool} \qquad \Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2}$$

$$\frac{\Gamma \vdash e : U \qquad U \leqslant T \qquad x \neq \mathsf{self} \qquad (\Gamma, x : T) \vdash S}{\Gamma \vdash T\ x := e\ \mathbf{in}\ S} \qquad\qquad \frac{\Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash S_1;\ S_2}$$

$$\frac{P \subseteq \mathit{Permissions} \qquad \Gamma \vdash S}{\Gamma \vdash \mathbf{enable}\ P\ \mathbf{in}\ S} \qquad\qquad \frac{P \subseteq \mathit{Permissions} \qquad \Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash \mathbf{test}\ P\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2}$$

Fig. 2. Typing rules for expressions and commands.

Definition 1
An *allocator* is a location-valued function *fresh* such that $\mathit{loctype}(\mathit{fresh}(C, h)) = C$ and $\mathit{fresh}(C, h) \notin \mathit{dom}\, h$, for all C, h. □

The semantics is isomorphic to one where addresses are untyped and each object has an immutable field tagging it with its type. One can take *Loc* to be the cartesian product $\mathbb{N} \times \mathit{Classnames}$ and define, e.g. $\mathit{fresh}(C, h) = (i, C)$ where i is the least natural not in $\mathit{dom}\, h$.

Methods are associated with classes, in a *method environment*, rather than with instances. For this reason the semantic domains are relatively simple; there are no recursive domain equations to be solved. For any data type T, the domain $[\![T]\!]$ is the set of values of type T. For any typing environment Γ, $[\![\Gamma]\!]$ is the set of stores assigning values of appropriate type to the variables in $\mathit{dom}\,\Gamma$. There are several other domains for which there is no corresponding notation in the syntax; for uniform notation, Figure 3 defines a collection of categories θ that includes T and Γ. Figure 4 defines the domain $[\![\theta]\!]$ for each category: $[\![\mathit{Heap}]\!]$ is the set of heaps, $[\![\mathit{state}\ C]\!]$ is the set of states of objects of class C, $[\![\mathit{perms}\ C]\!]$ is sets of permissions authorized for C, $[\![\mathit{MEnv}]\!]$ is the set of method environments, $[\![C, \bar{x}, \bar{T} \to T]\!]$ is the set of meanings for methods of class C with result T and parameters $\bar{x} : \bar{T}$. In a language like Java with garbage collection and without pointer arithmetic, dangling locations (those not in the domain of the heap) never occur in program states or

θ	$::=$	T	values of type T
	$\mid$	Γ	stores (map each variable $x \in dom\,\Gamma$ to value of type $\Gamma\,x$)
	$\mid$	$state\ C$	object states (map fields to values)
	$\mid$	$Heap$	maps locations to object states with no dangling locations
	$\mid$	$Heap \otimes \Gamma$	global states with no dangling locations
	$\mid$	$Heap \otimes T$	pairs (h, d) where value d is not a dangling location w.r.t. h
	$\mid$	$\theta_\bot$	lifting
	$\mid$	$perms\ C$	permission sets statically authorized for C
	$\mid$	$(C, \bar{x}, \bar{T} \to T)$	method of C with parameters $\bar{x} : \bar{T}$ and return type T
	$\mid$	$MEnv$	method environments

Fig. 3. Semantic categories and informal description of their denotation.

$$\begin{array}{lcl}
[\![\mathbf{bool}]\!] & = & \{true, false\} \\
[\![\mathbf{unit}]\!] & = & \{it\} \\
[\![C]\!] & = & \{nil\} \cup \{\ell \mid \ell \in Loc \wedge loctype\,\ell \leqslant C\} \\
[\![\Gamma]\!] & = & \{\eta \mid dom\,\eta = dom\,\Gamma \wedge \eta\,\mathsf{self} \neq nil \wedge \forall x \in dom\,\eta \bullet \eta\,x \in [\![\Gamma\,x]\!]\} \\
[\![state\ C]\!] & = & \{s \mid dom\,s = dom(fields\,C) \wedge \forall (f : T) \in fields\,C \bullet sf \in [\![T]\!]\} \\
[\![Heap]\!] & = & \{h \mid dom\,h \subseteq_{fin} Loc \wedge closed\,h \wedge \forall \ell \in dom\,h \bullet h\ell \in [\![state\,(loctype\,\ell)]\!]\} \\
 & & \text{where } closed\,h \text{ iff } rng\,s \cap Loc \subseteq dom\,h \text{ for all } s \in rng\,h \\
[\![Heap \otimes \Gamma]\!] & = & \{(h, \eta) \mid h \in [\![Heap]\!] \wedge \eta \in [\![\Gamma]\!] \wedge rng\,\eta \cap Loc \subseteq dom\,h\} \\
[\![Heap \otimes T]\!] & = & \{(h, d) \mid h \in [\![Heap]\!] \wedge d \in [\![T]\!] \wedge (d \in Loc \Rightarrow d \in dom\,h)\} \\
[\![\theta_\bot]\!] & = & [\![\theta]\!] \cup \bot \quad (\text{where } \bot \text{ is some fresh value not in } [\![\theta]\!]) \\
[\![perms\ C]\!] & = & \{P \mid P \subseteq Auth\ C\} \\
[\![C, \bar{x}, \bar{T} \to T]\!] & = & [\![Heap \otimes (\bar{x} : \bar{T}, \mathsf{self} : C)]\!] \to \mathscr{P}(Permissions) \to [\![(Heap \otimes T)_\bot]\!] \\
[\![MEnv]\!] & = & \{\mu \mid \ \forall C, m \bullet \mu Cm \text{ is defined iff } mtype(m, C) \text{ is defined,} \\
 & & \quad \text{and } \mu Cm \in [\![C, pars(m, C), mtype(m, C)]\!] \text{ if } \mu Cm \text{ defined}\}
\end{array}$$

Fig. 4. Semantic domains, for given policy *Auth*.

as expression values. Capturing this in the semantics is the purpose of the special cartesian products $Heap \otimes \Gamma$ and $Heap \otimes T$.

For expressions and commands, the semantics is defined only for derivable typing judgements. The meaning of an expression $\Gamma \vdash e : T$ is a function $[\![Heap \otimes \Gamma]\!] \to [\![T_\bot]\!]$ that takes a state $(h, \eta) \in [\![Heap \otimes \Gamma]\!]$ and returns either a value $d \in [\![T]\!]$, such that $(h, d) \in [\![Heap \otimes T]\!]$, or the improper value $\bot$ which represents errors. The errors are null dereferences and cast failure; the other expression constructs are strict in $\bot$. We have no need to define an explicit semantic category to designate the domain $[\![Heap \otimes \Gamma]\!] \to [\![T_\bot]\!]$.

The typing rules are syntax-directed, so it is easy to show that a given judgement has at most one derivation. Thus the semantics of a derivable expression typing can be defined by recursion on its derivation; see Mitchell (1996, section 4.5), for a textbook discussion. Figure 5 gives the definition.

The meaning of a command $\Gamma \vdash S$ is a function

$$[\![MEnv]\!] \to [\![Heap \otimes \Gamma]\!] \to [\![perms(\Gamma\,\mathsf{self})]\!] \to [\![(Heap \otimes \Gamma)_\bot]\!] \qquad (1)$$

$$
\begin{array}{lcl}
\llbracket \Gamma \vdash x : T \rrbracket(h,\eta) & = & \eta x \\
\llbracket \Gamma \vdash \mathbf{null} : B \rrbracket(h,\eta) & = & nil \\
\llbracket \Gamma \vdash \mathbf{true} : \mathbf{bool} \rrbracket(h,\eta) & = & true \\
\llbracket \Gamma \vdash \mathbf{false} : \mathbf{bool} \rrbracket(h,\eta) & = & false \\
\llbracket \Gamma \vdash e_1 = e_2 : \mathbf{bool} \rrbracket(h,\eta) & = & \mathsf{let}\ d_1 = \llbracket \Gamma \vdash e_1 : T_1 \rrbracket(h,\eta)\ \mathsf{in} \\
 & & \mathsf{let}\ d_2 = \llbracket \Gamma \vdash e_2 : T_2 \rrbracket(h,\eta)\ \mathsf{in}\ \mathsf{if}\ d_1 = d_2\ \mathsf{then}\ true\ \mathsf{else}\ false \\
\llbracket \Gamma \vdash e.f : T \rrbracket(h,\eta) & = & \mathsf{let}\ \ell = \llbracket \Gamma \vdash e : C \rrbracket(h,\eta)\ \mathsf{in}\ \mathsf{if}\ \ell = nil\ \mathsf{then}\ \bot\ \mathsf{else}\ h\,\ell\,f \\
\llbracket \Gamma \vdash (B)\ e : B \rrbracket(h,\eta) & = & \mathsf{let}\ \ell = \llbracket \Gamma \vdash e : D \rrbracket(h,\eta)\ \mathsf{in} \\
 & & \mathsf{if}\ \ell = nil \vee loctype\ \ell \leqslant B\ \mathsf{then}\ \ell\ \mathsf{else}\ \bot \\
\llbracket \Gamma \vdash e\ \mathbf{is}\ B : \mathbf{bool} \rrbracket(h,\eta) & = & \mathsf{let}\ \ell = \llbracket \Gamma \vdash e : D \rrbracket(h,\eta)\ \mathsf{in} \\
 & & \mathsf{if}\ \ell \neq nil \wedge loctype\ \ell \leqslant B\ \mathsf{then}\ true\ \mathsf{else}\ false
\end{array}
$$

Fig. 5. Semantics of expressions.

$$
\begin{array}{lcl}
\llbracket \Gamma \vdash x := e \rrbracket \mu(h,\eta) Q & = & \mathsf{let}\ d = \llbracket \Gamma \vdash e : T \rrbracket(h,\eta)\ \mathsf{in}\ (h, [\eta \mid x \mapsto d]) \\
\llbracket \Gamma \vdash e_1.f := e_2 \rrbracket \mu(h,\eta) Q & = & \mathsf{let}\ \ell = \llbracket \Gamma \vdash e_1 : C \rrbracket(h,\eta)\ \mathsf{in} \\
 & & \mathsf{if}\ \ell = nil\ \mathsf{then}\ \bot\ \mathsf{else} \\
 & & \mathsf{let}\ d = \llbracket \Gamma \vdash e_2 : U \rrbracket(h,\eta)\ \mathsf{in}\ ([h \mid \ell \mapsto [h\ell \mid f \mapsto d]], \eta) \\
\llbracket \Gamma \vdash x := \mathbf{new}\ B \rrbracket \mu(h,\eta) Q & = & \mathsf{let}\ \ell = fresh(B,h)\ \mathsf{in} \\
 & & \mathsf{let}\ h_1 = [h \mid \ell \mapsto [fields\,B \mapsto defaults]]\ \mathsf{in} \\
 & & (h_1, [\eta \mid x \mapsto \ell]) \\
\llbracket \Gamma \vdash x := e.m(\bar{e}) \rrbracket \mu(h,\eta) Q & = & \mathsf{let}\ \ell = \llbracket \Gamma \vdash e : D \rrbracket(h,\eta)\ \mathsf{in} \\
 & & \mathsf{if}\ \ell = nil\ \mathsf{then}\ \bot\ \mathsf{else} \\
 & & \mathsf{let}\ \bar{x} = pars(m,D)\ \mathsf{in} \\
 & & \mathsf{let}\ \bar{d} = \llbracket \Gamma \vdash \bar{e} : \bar{U} \rrbracket(h,\eta)\ \mathsf{in} \\
 & & \mathsf{let}\ \eta_1 = [\bar{x} \mapsto \bar{d}, \mathsf{self} \mapsto \ell]\ \mathsf{in} \\
 & & \mathsf{let}\ (h_0, d_0) = \mu(loctype\ \ell) m(h,\eta_1) Q\ \mathsf{in}\ (h_0, [\eta \mid x \mapsto d_0]) \\
\llbracket \Gamma \vdash S_1;\ S_2 \rrbracket \mu(h,\eta) Q & = & \mathsf{let}\ (h_1,\eta_1) = \llbracket \Gamma \vdash S_1 \rrbracket \mu(h,\eta) Q\ \mathsf{in}\ \llbracket \Gamma \vdash S_2 \rrbracket \mu(h_1,\eta_1) Q \\
\multicolumn{3}{l}{\llbracket \Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 \rrbracket \mu(h,\eta) Q} \\
 & = & \mathsf{let}\ b = \llbracket \Gamma \vdash e : \mathbf{bool} \rrbracket(h,\eta)\ \mathsf{in} \\
 & & \mathsf{if}\ b\ \mathsf{then}\ \llbracket \Gamma \vdash S_1 \rrbracket \mu(h,\eta) Q\ \mathsf{else}\ \llbracket \Gamma \vdash S_2 \rrbracket \mu(h,\eta) Q \\
\llbracket \Gamma \vdash T\ x := e\ \mathbf{in}\ S \rrbracket \mu(h,\eta) Q & = & \mathsf{let}\ d = \llbracket \Gamma \vdash e : U \rrbracket(h,\eta)\ \mathsf{in} \\
 & & \mathsf{let}\ \eta_1 = [\eta \mid x \mapsto d]\ \mathsf{in} \\
 & & \mathsf{let}\ (h_1,\eta_2) = \llbracket (\Gamma, x : T) \vdash S \rrbracket \mu(h,\eta_1) Q\ \mathsf{in}\ (h_1, (\eta_2 \restriction x)) \\
\llbracket \Gamma \vdash \mathbf{enable}\ P\ \mathbf{in}\ S \rrbracket \mu(h,\eta) Q & = & \llbracket \Gamma \vdash S \rrbracket \mu(h,\eta)(Q \cup (P \cap Auth(\Gamma\,\mathsf{self}))) \\
\multicolumn{3}{l}{\llbracket \Gamma \vdash \mathbf{test}\ P\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 \rrbracket \mu(h,\eta) Q} \\
 & = & \mathsf{if}\ P \subseteq Q\ \mathsf{then}\ \llbracket \Gamma \vdash S_1 \rrbracket \mu(h,\eta) Q\ \mathsf{else}\ \llbracket \Gamma \vdash S_2 \rrbracket \mu(h,\eta) Q
\end{array}
$$

Fig. 6. Semantics of commands, for given policy *Auth* and allocator *fresh*.

that takes a method environment μ (see below), a state (h,η), and the enabled permissions $Q \in \llbracket perms(\Gamma\,\mathsf{self}) \rrbracket$; it returns a state or $\bot$ which indicates divergence or error. The definition, in Figure 6, is again by recursion on the typing derivation. Note that the semantic domain (1) depends on $perms(\Gamma\,\mathsf{self})$. As with expressions, we refrain from introducing an explicit name for this domain.

To streamline the treatment of $\bot$ in the semantic definitions, we use a metalanguage construct, $\mathsf{let}\ d = E_1\ \mathsf{in}\ E_2$, with the following meaning: If the value of E_1 is $\bot$ then that is the value of the entire let expression; otherwise, its value is the value of E_2 with d bound to the value of E_1.

For functions of various kinds we write *dom* or *rng* for the domain or range. Function update or extension is written, e.g. $[\eta \mid x \mapsto d]$. In the semantics of local variables, we write $\upharpoonright$ for domain restriction: if x is in the domain of function η then $\eta \upharpoonright x$ is the function like η but without x in its domain.

A *method environment* μ maps each class name C and method name m (declared or inherited in C) to a meaning $\mu\, C\, m$ which is an element of

$$[\![Heap \otimes \Gamma]\!] \to \mathscr{P}(Permissions) \to [\![(Heap \otimes T)_\bot]\!]$$

where T is the return type and $\Gamma = \mathsf{self}:C, \bar{x}:\bar{T}$ is the parameter store, where $\bar{x} = pars(m, C)$. The result from a method, if not $\bot$, is a pair (h, d) with d in $[\![T]\!]$ such that, if d is a location then d is in the domain of the result heap h.

A class table denotes a method environment obtained as the least upper bound of a chain of approximations. For this purpose each semantic domain is given a partial ordering. The sets $[\![Heap]\!]$, $[\![\mathbf{bool}]\!]$, $[\![C]\!]$, $[\![state\ C]\!]$, $[\![perms\ C]\!]$, and $\mathscr{P}(Permissions)$ are ordered by equality, as is $[\![Heap \otimes \Gamma]\!]$. The ordering on $[\![(Heap \otimes \Gamma)_\bot]\!]$ is the *flat order*: $X \sqsubseteq Y$ iff $X = \bot \vee X = Y$. Thus elements of $[\![C, \bar{x}, \bar{T} \to T]\!]$ are curried functions from discrete domains to a flat one (all such functions are continuous). We order $[\![C, \bar{x}, \bar{T} \to T]\!]$ pointwise, and note that it has a least element (the constantly-$\bot$ function) and least upper bounds of ascending chains.

To keep the proofs self-contained, we need an elementary characterization of least upper bounds for method environments. To this end, note first that for any d_1, d_2 in $[\![C, \bar{x}, \bar{T} \to T]\!]$ we have the following, writing $\sqsubseteq$ for the pointwise order:

$$d_1 \sqsubseteq d_2 \iff \forall h, \eta, Q \bullet d_1(h, \eta)Q \neq \bot \Rightarrow d_1(h, \eta)Q = d_2(h, \eta)Q \tag{2}$$

The reason is that elements of $[\![C, \bar{x}, \bar{T} \to T]\!]$ are functions into flat cpo $[\![(Heap \otimes T)_\bot]\!]$. Now consider an ascending chain $d : \mathbb{N} \to [\![C, \bar{x}, \bar{T} \to T]\!]$. For any h, η, Q, there is some j such that $(lub\ d)(h, \eta)Q = d_j(h, \eta)Q$; this follows by the lub property from (2) with instantiation $d_1 := d_j$ and $d_2 := (lub\ d)$. Moreover, $(lub\ d)(h, \eta)Q = d_k(h, \eta)Q$ for all $k \geqslant j$.

To cope with mutually recursive definitions, the semantics is defined using a chain of method environments. The preceding ideas extend straightforwardly to yield the following.

Lemma 4.1 (*characterization of least upper bounds for method environments*)

Let μ be an ascending chain, $\mu : \mathbb{N} \to [\![MEnv]\!]$, of method environments for CT. For any C, m, h, η, Q, there is some j such that, for all $k \geqslant j$,

$$(lub\ \mu)\, C\, m\, (h, \eta)\, Q = \mu_k\, C\, m\, (h, \eta)\, Q$$

Proof
Follows from the fact, analogous to (2), that for any μ_1, μ_2 we have $\mu_1 \sqsubseteq \mu_2$ iff

$$\forall C, m, h, \eta, Q \bullet \mu_1 Cm(h,\eta)Q \neq \bot \;\Rightarrow\; \mu_1 Cm(h,\eta)Q = \mu_2 Cm(h,\eta)Q \tag{3}$$

□

Definition 2 (*semantics of complete program*)
The semantics of a class table CT is a method environment, written $[\![CT]\!]$, given as a least upper bound. Specifically, $[\![CT]\!] = \mathit{lub}\ \mu$ where the ascending chain $\mu \in \mathbb{N} \to [\![MEnv]\!]$ is defined as follows, using the semantics $[\![M]\!]$ defined later.[1]

$$\begin{aligned}
&\mu_0\, C\, m = \lambda(h,\eta) \bullet \lambda Q \bullet \bot \\
&\mu_{j+1}\, C\, m = [\![M]\!]\mu_j \quad \text{if } m \text{ is declared as } M \text{ in } C \\
&\mu_{j+1}\, C\, m = \mu_{j+1}\, B\, m \quad \text{if } m \text{ is inherited from } B \text{ in } C
\end{aligned}$$

To be very precise for an inherited method, if $mtype(m, C) = \bar{T} \to T$ then $\mu_{j+1}\, C\, m$ should apply to stores for $\bar{x} : \bar{T}, \mathsf{self} : C$ whereas $\mu_{j+1}\, B\, m$ applies to stores for $\bar{x} : \bar{T}, \mathsf{self} : B$. But the latter contains the former, as $C \leqslant B$ implies $[\![C]\!] \subseteq [\![B]\!]$. This does not obtrude in the sequel.

The interesting aspect of inheritance is that the permissions $\mathit{Auth}\, B$ are not required to have any relation to the permissions $\mathit{Auth}\, C$. Recall from section 2 that access control is defined in terms of the code on the stack, not the classes of objects for which the code is executing. Leaving aside dynamic binding, the semantics of method invocation could be defined by intersecting the current permissions with those authorized for the called method. To interpret dynamic binding, our semantics branches on the type of the target object, and the method environment provides a meaning for every method. In the case of an inherited method, the permissions "authorized for the called method" should be those of its defining class, not the class into which it is inherited. So we consider that a method meaning is defined for all permission sets. Intersection with the authorized permissions is done not in the semantics of method call but in the semantics of method declarations. This is why $[\![C,\ \bar{x},\ \bar{T} \to T]\!]$ is defined using $\mathscr{P}(\mathit{Permissions})$ rather than $[\![\mathit{perms}\ C]\!]$.

Definition 3 (*semantics of method declaration*)
For a declaration $M = T\ m(\bar{T}\ \bar{x})\{S\}$ in class C, define $[\![M]\!]$ by

$$\begin{aligned}
[\![M]\!]\mu(h,\eta)Q = \;& \mathsf{let}\ Q_1 = Q \cap \mathit{Auth}\ C\ \mathsf{in} \\
& \mathsf{let}\ \eta_1 = [\eta \mid \mathsf{result} \mapsto \mathit{default}]\ \mathsf{in} \\
& \mathsf{let}\ (h_0, \eta_0) = [\![\bar{x} : \bar{T}, \mathsf{self} : C, \mathsf{result} : T \vdash S]\!]\mu(h,\eta)Q_1\ \mathsf{in} \\
& (h_0,\ \eta_0\ \mathsf{result})
\end{aligned}$$

On fixpoints. We have chosen to formulate the semantic definition as a least upper bound, as this facilitates straightforward proofs of the main results. It is in fact a least fixpoint, of a function that we have not made explicit; it is the function used in Definition 2 to obtain μ_{j+1} from μ_j. The interested reader can check that this

[1] Warning: Elsewhere in the paper we use, e.g., $\kappa, \kappa_1, \kappa_2$ as unrelated identifiers, some of which happen to be decorated with subscripts.

function is continuous and thus its least fixpoint is the least upper bound of our approximation chain.

If the semantics is described as a least fixed point then proofs can use fixpoint induction. Fixpoint induction requires not only continuity of the relevant function but also admissibility of the predicate, say P, to be proved: $\forall i \bullet P(\mu_i)$ implies $P(lub\ \mu)$. In the proofs we prove the implication but use it directly, without need to mention fixpoints. This presentation has been chosen mainly because the standard characterizations of admissibility do not happen to apply to our predicates and there is a simple direct proof of admissibility using Lemma 4.1.

5 Safety

The syntactic property given by static analysis is called *safety*. The analysis is specified by a typing system which is convenient for proving our results. For practical purposes, type inference is needed and concrete syntax is needed to designate policy separately from the code (see section 9). The short word "safe", used in preference to "security-typable", has a semantic connotation but this is justified by our main results.

In this section we annotate the syntax of section 4 with security labels. Where types T occur in declarations of fields and local variables, we use pairs (T, κ) where κ is a security level, L or H. Such a pair, written τ, is called a *security type*. The grammar is revised as follows.

$$\begin{array}{rcl} \kappa & ::= & L \mid H \\ \tau & ::= & (T, \kappa) \\ CL & ::= & \textbf{class}\ C\ \textbf{extends}\ C\ \{\ \bar{\tau}\ \bar{f};\ \bar{M}\} \\ S & ::= & \ldots \mid \tau\ x := e\ \textbf{in}\ S \mid \ldots \end{array}$$

Note that there is no change for cast and test.

We refrain from giving concrete syntax for the security types of method parameters, results, and effects. By analogy with the auxiliary function *mtype* which gives the declared type of a method (see section 4.1), we assume that a function *smtypes* is given. It may assign multiple security types for a method, each of the form $\kappa, \bar{\kappa} \text{–}\!\langle P\,;\kappa_1 \rangle\!\!\rightarrow \kappa_2$. The intended meaning is as follows: if the method is called with arguments compatible with $\bar{\kappa}$, target object compatible with κ, and enabled permissions disjoint from P, then the heap effect is $\geqslant \kappa_1$ and the result level $\leqslant \kappa_2$. This is made precise in Definition 10.

There is an ordering on method typings $\kappa, \bar{\kappa} \text{–}\!\langle P\,;\kappa_1 \rangle\!\!\rightarrow \kappa_2$. It is contravariant on inputs $\kappa, \bar{\kappa}$ and P and on assignables κ_1, covariant on the result value κ_2.

Definition 4 (*subtyping*)
$\kappa, \bar{\kappa} \text{–}\!\langle P\,;\kappa_1 \rangle\!\!\rightarrow \kappa_2 \ \leqslant\ \kappa', \bar{\kappa}' \text{–}\!\langle P'\,;\kappa_1' \rangle\!\!\rightarrow \kappa_2'$ iff $\kappa' \leqslant \kappa$, $\bar{\kappa}' \leqslant \bar{\kappa}$, $P \subseteq P'$, $\kappa_1' \leqslant \kappa_1$, and $\kappa_2 \leqslant \kappa_2'$. $\square$

Note that P is interpreted negatively, so the condition $P \subseteq P'$ is effectively contravariant.

We refrain from defining the induced orderings on (T, κ) and $(com\ \kappa, \kappa)$; it is simpler to build them into the subsumption rules.

$$\Delta \vdash x : \Delta\, x \qquad\qquad \Delta \vdash \mathbf{null} : (D, \kappa) \qquad\qquad \Delta \vdash \mathbf{true} : (\mathbf{bool}, \kappa)$$

$$\frac{\Delta \vdash e_1 : (T_1, \kappa) \qquad \Delta \vdash e_2 : (T_2, \kappa)}{\Delta \vdash e_1 = e_2 : (\mathbf{bool}, \kappa)} \qquad\qquad \frac{\Delta \vdash e : (D, \kappa) \qquad B \leqslant D}{\Delta \vdash (B)\ e : (B, \kappa)}$$

$$\frac{\Delta \vdash e : (C, \kappa_1) \qquad f : (T, \kappa) \in \mathit{sfields}\, C}{\Delta \vdash e.f : (T, \kappa \sqcup \kappa_1)} \qquad\qquad \frac{\Delta \vdash e : (D, \kappa) \qquad B \leqslant D}{\Delta \vdash e\ \mathbf{is}\ B : (\mathbf{bool}, \kappa)}$$

$$\frac{\Delta \vdash e : (T, \kappa) \qquad \kappa \leqslant \kappa'}{\Delta \vdash e : (T, \kappa')}$$

Fig. 7. Security typing rules for expressions.

Definition 5 (*annotated class table*)
An annotated class table is a class table with annotations according to the grammar above, together with a partial function *smtypes* satisfying the following conditions. First, $\mathit{smtypes}(m, C)$ is defined iff $\mathit{mtype}(m, C)$ is defined. Second, if $\mathit{smtypes}(m, C)$ is defined then it is a *non-empty* set of annotations of the form $\kappa, \bar{\kappa} \!\prec\!\!\langle P\, ; \kappa_1 \rangle\!\!\succ\!\!\to \kappa_2$. Third, if $C \leqslant D$ and $\mathit{mtype}(m, D)$ is defined then $\mathit{smtypes}(m, C) = \mathit{smtypes}(m, D)$. □

Note that we do not require $P \subseteq \mathit{Auth}\, C$. A method may be declared in one class and inherited or overridden in a subclass with different permissions. The third condition allows us to reason about method calls in terms of the static type of a called method, because any implementation that can be invoked by dynamic dispatch is checked with respect to the same security types.

We use the symbol † to erase annotations: $(T, \kappa)^\dagger = T$, and this extends to erasure for typing environments, commands, and method declarations in an obvious way.

A method can have more than one type so for flexibility in checking method declarations the rule must allow local variable declarations to be annotated differently for different types. The rule in the sequel uses the following notion: S' is a *local variant* of S iff $S^\dagger$ is syntactically identical to $(S')^\dagger$. Note that S and S' have the same semantics.

Definition 6 (*safe class table and method declaration*)
An annotated class table CT is *safe* provided that each class satisfies the rule

$$\frac{C\ \mathbf{extends}\ D \vdash M \text{ for each } M \in \bar{M}}{\vdash \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ \bar{\tau}\ \bar{f};\ \bar{M}\}}$$

The hypothesis of this rule requires each method declaration to be checked with respect to its security types according to the following rule:

$$\frac{\begin{array}{c} \mathit{mtype}(m, C) = \bar{T} \to T \qquad \mathit{pars}(m, C) = \bar{x} \\ \text{For each } (\kappa_0, \bar{\kappa} \!\prec\!\!\langle P\, ; \kappa_3 \rangle\!\!\succ\!\!\to \kappa_4) \text{ in } \mathit{smtypes}(m, C) \text{ there is a local variant } S' \text{ of } S \\ \text{such that } \mathsf{self} : (C, \kappa_0), \bar{x} : (\bar{T}, \bar{\kappa}), \mathsf{result} : (T, \kappa_4);\ (P \cap \mathit{Auth}\, C)\ \vdash\ S' : (\mathit{com}\ L, \kappa_3) \end{array}}{C\ \mathbf{extends}\ D \vdash T\ m(\bar{T}\ \bar{x})\{S\}}$$

This rule depends on rules for expressions and commands, which are given in Figures 7 and 8. □

$$\frac{x \neq \mathsf{self} \qquad \Delta, x:(T,\kappa) \vdash e:(U,\kappa) \qquad U \leqslant T}{\Delta, x:(T,\kappa); P \vdash x := e : (com\ \kappa, H)}$$

$$\frac{\Delta \vdash e_1 : (C,\kappa_1) \qquad f:(T,\kappa) \in \mathit{sfields}\, C \qquad \Delta \vdash e_2 : (U,\kappa) \qquad U \leqslant T \qquad \kappa_1 \leqslant \kappa}{\Delta; P \vdash e_1.f := e_2 : (com\ H, \kappa)}$$

$$\frac{x \neq \mathsf{self} \qquad B \leqslant D}{\Delta, x:(D,\kappa); P \vdash x := \mathbf{new}\ B : (com\ \kappa, H)}$$

$$\frac{\begin{array}{c}\Delta, x:(T,\kappa) \vdash e:(D,\kappa_0) \qquad mtype(m,D) = \bar{T} \rightarrow T' \qquad \Delta, x:(T,\kappa) \vdash \bar{e}:(\bar{U},\bar{\kappa}) \\ \bar{U} \leqslant \bar{T} \qquad x \neq \mathsf{self} \qquad T' \leqslant T \qquad \kappa_0', \bar{\kappa}' \prec\!\langle P'; \kappa_1' \rangle\!\rightarrowtail \kappa' \in smtypes(m,D) \\ \kappa_0', \bar{\kappa}' \prec\!\langle P'; \kappa_1' \rangle\!\rightarrowtail \kappa' \leqslant \kappa_0, \bar{\kappa} \prec\!\langle P'; \kappa_1 \rangle\!\rightarrowtail \kappa \qquad P' \cap Auth(\Delta^{\dagger}\mathsf{self}) \subseteq P \qquad \kappa_0 \leqslant \kappa \sqcap \kappa_1\end{array}}{\Delta, x:(T,\kappa); P \vdash x := e.m(\bar{e}) : (com\ \kappa, \kappa_1)}$$

$$\frac{\Delta; P \vdash S_1 : (com\ \kappa_1, \kappa_2) \qquad \Delta; P \vdash S_2 : (com\ \kappa_1, \kappa_2)}{\Delta; P \vdash S_1;\ S_2 : (com\ \kappa_1, \kappa_2)}$$

$$\frac{\Delta \vdash e : (\mathbf{bool}, \kappa) \qquad \Delta; P \vdash S_1 : (com\ \kappa_1, \kappa_2) \qquad \Delta; P \vdash S_2 : (com\ \kappa_1, \kappa_2) \qquad \kappa \leqslant \kappa_1 \sqcap \kappa_2}{\Delta; P \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 : (com\ \kappa_1, \kappa_2)}$$

$$\frac{\Delta \vdash e : (U,\kappa) \qquad U \leqslant T \qquad \Delta, x:(T,\kappa); P \vdash S : (com\ \kappa_1, \kappa_2)}{\Delta; P \vdash (T,\kappa)\ x := e\ \mathbf{in}\ S : (com\ \kappa_1, \kappa_2)}$$

$$\frac{\Delta; (P - (P' \cap Auth(\Delta^{\dagger}\mathsf{self}))) \vdash S : (com\ \kappa_1, \kappa_2)}{\Delta; P \vdash \mathbf{enable}\ P'\ \mathbf{in}\ S : (com\ \kappa_1, \kappa_2)}$$

$$\frac{P' \cap P = \emptyset \wedge P' \subseteq Auth(\Delta^{\dagger}\mathsf{self}) \qquad \Delta; P \vdash S_1 : (com\ \kappa_1, \kappa_2) \qquad \Delta; P \vdash S_2 : (com\ \kappa_1, \kappa_2)}{\Delta; P \vdash \mathbf{test}\ P'\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 : (com\ \kappa_1, \kappa_2)}$$

$$\frac{P' \cap P \neq \emptyset \vee P' \nsubseteq Auth(\Delta^{\dagger}\mathsf{self}) \qquad \Delta; P \vdash S_2 : (com\ \kappa_1, \kappa_2)}{\Delta; P \vdash \mathbf{test}\ P'\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 : (com\ \kappa_1, \kappa_2)}$$

$$\frac{\Delta; P \vdash S : (com\ \kappa_1, \kappa_2) \qquad \kappa_3 \leqslant \kappa_1 \qquad \kappa_4 \leqslant \kappa_2}{\Delta; P \vdash S : (com\ \kappa_3, \kappa_4)}$$

Fig. 8. Security typing rules for commands, for given *Auth*.

In the rules for expressions and commands, we write Δ for typing environments that assign security types. The rules use a version of the *fields* auxiliary function that takes security levels into account. Let $CT(C) = \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ \bar{\tau}_1\ \bar{f};\ \bar{M}\}$, to define $\mathit{sfields}\, C = \bar{f} : \bar{\tau}_1 \cup \mathit{sfields}\, D$.

A judgement $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2)$ says that S is safe and assigns only to variables (locals and parameters) of level $\geqslant \kappa_1$ and to object fields of level $\geqslant \kappa_2$ (see Lemma 7.2) provided that no permissions in set P are enabled initially. For commands, we allow only judgements where $P \subseteq Auth(\Delta^{\dagger}\mathsf{self})$; only such P is relevant to the behavior of a command declared in class $\Delta^{\dagger}\mathsf{self}$. This is reflected in the rule for method declaration in Definition 6.

Assignments to variables within a method have no effect on the caller's state; this is why it is useful to have two separate levels in command types. The rule for method declaration does not restrict assignments to local variables, i.e. it allows effect L in the hypothesis.

The last rule in both Figures 7 and 8 is a subsumption rule. These rules help simplify the syntax-directed rules, e.g. the rule for assignment requires the level of the target variable x to be the same as the assigned expression e.

A form of subsumption is built in to the rule for method call, which requires there to be some declared type for the method that matches its invocation. Note constraints in the rule prevent use of a type $\kappa_0, \bar{\kappa} \!\prec\!\langle P\,;\kappa_1\rangle\!\!\rightarrow\!\kappa$ unless $\kappa_0 \leqslant \kappa_1$. So a type not satisfying this condition is useless, but we have refrained from cluttering definitions by imposing this condition on *smtypes*.

Subsumptions could be used to simplify the conditional rule, but at the cost of imprecision in the case that the guard condition is L but the commands are H.

The rule for **enable** accurately tracks the effect on permissions. In it we write "$-$" for set subtraction. The second rule for **test** is the one that removes from consideration a branch that, in the given context, cannot be taken: The test of P' fails if P' contains permissions assumed to be excluded or permissions that are not authorized for the class in which this command occurs. The first rule for **test** handles the case where it cannot be statically determined, from the information tracked in the judgements, whether the test of P' succeeds.

On first reading it is advisable to skip to section 6 which shows the rules in action. We conclude the present section with minor technicalities.

Properties of security typing. For any judgement $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2)$ derivable using the rules in Figures 7 and 8, the erased judgement $\Delta^\dagger \vdash S^\dagger$ is derivable using the rules of Figure 2. Conversely, any program typable using the rules of Figure 2 can be annotated everywhere by L and typed by the rules in Figures 7 and 8, taking $smtypes(m, C) = \{L, \bar{L} \!\prec\!\langle \varnothing; L\rangle\!\!\rightarrow\! L\}$ for all m, C.

For brevity we write $Q \# P$ for $Q \cap P = \varnothing$.

For reasoning about method calls we repeatedly use the following.

Lemma 5.1

Suppose $\Delta, x : (T, \kappa); P \vdash x := e.m(\bar{e}) : (com\ \kappa, \kappa_1)$ is derived using the method call rule instantiated with $\kappa_0', \bar{\kappa}' \!\prec\!\langle P'\,; \kappa_1'\rangle\!\!\rightarrow\!\kappa' \in smtypes(m, D)$, and $(\kappa_0', \bar{\kappa}' \!\prec\!\langle P'\,; \kappa_1'\rangle\!\!\rightarrow\!\kappa') \leqslant (\kappa_0, \bar{\kappa} \!\prec\!\langle P'\,; \kappa_1\rangle\!\!\rightarrow\!\kappa)$. If $Q \# P$ and $Q \subseteq Auth(\Delta^\dagger\, \mathsf{self})$ then $Q \# P'$.

Proof

By the typing rule we have $P' \cap Auth(\Delta^\dagger\, \mathsf{self}) \subseteq P$ hence, from $Q \# P$, we have $P' \cap Auth(\Delta^\dagger\, \mathsf{self}) \cap Q = \varnothing$. Then $Q \subseteq Auth(\Delta^\dagger\, \mathsf{self})$ implies $Q \cap P' = \varnothing$. $\square$

6 Examples

In this section, we first show how the rules work with the examples in section 3, and extend the examples to consider subclassing and inheritance. Next, we show several examples of leaks that may occur in object-oriented programs due to control flow and aliasing. The security type system in Figure 8 can be deployed to reject such programs. Finally, we show an example concerning integrity.

Examples from section 3. For the information flow policy, the fields of class Kern are annotated as

(**string**,H) Hinfo; (**string**,L) Linfo;

and fields of class Comp1 are annotated as

(Kern,L) k; (**string**,L) v;

The most interesting method typings are given as

$$\begin{aligned}&\mathit{smtypes}(\mathsf{getHinfo,\ Kern}) = \{H, ()\!\prec\!\langle\emptyset; H\rangle\!\!\rightarrowtail\! H\}\\ &\mathit{smtypes}(\mathsf{getStatus,\ Kern}) = \{H, ()\!\prec\!\langle\{\mathit{stat}\}; H\rangle\!\!\rightarrowtail\! L,\ H, ()\!\prec\!\langle\emptyset; H\rangle\!\!\rightarrowtail\! H\}\\ &\mathit{smtypes}(\mathsf{status,Comp1}) = \{H, ()\!\prec\!\langle\emptyset; H\rangle\!\!\rightarrowtail\! L\}\end{aligned}$$

For example, the intended meaning of type $H, ()\!\prec\!\langle\{\mathit{stat}\}; H\rangle\!\!\rightarrowtail\! L$ is that if method getStatus is called with self of level $\leqslant H$, empty argument list, and enabled permissions disjoint from *stat*, then the heap effect is $\geqslant H$ and the result level is $\leqslant L$.

The body of Comp1.status is checked in the context of result : $(\textbf{string}, L)$ and excluded permission set $\emptyset \cap \mathit{Auth}(\mathsf{Comp1})$, i.e. $\emptyset$. The call to getStatus can be checked using the type $H, ()\!\prec\!\langle\{\mathit{stat}\}; H\rangle\!\!\rightarrowtail\! L$, because $\{\mathit{stat}\} \cap \mathit{Auth}(\mathsf{Comp1}) = \{\mathit{stat}\} \cap \{\mathit{other}\} \subseteq \emptyset$.

Method Comp1.status2 can also be checked using the same types, as the rule for **enable** takes into account that the attempt to enable *stat* fails because $\mathit{stat} \notin \mathit{Auth}(\mathsf{Comp1})$.

Consider the policy $\mathit{smtypes}(\mathsf{statusH,\ Comp2}) = \{H, ()\!\prec\!\langle\emptyset; H\rangle\!\!\rightarrowtail\! H\}$. This requires checking the body of Comp2.statusH in the context of result : $(\textbf{string}, H)$ and $\emptyset$. To check the command

enable *stat* **in** result := k.getStatus()

we check result := k.getStatus() in the context of $\emptyset$, which is $\emptyset - \{\mathit{stat}\}$. This check succeeds, using the type $H, ()\!\prec\!\langle\emptyset; H\rangle\!\!\rightarrowtail\! H$ for getStatus.

Finally, consider checking getStatus. To check it with respect to $H, ()\!\prec\!\langle\emptyset; H\rangle\!\!\rightarrowtail\! H$ requires checking both branches of

test *stat* **then enable** sys **in** result := getHinfo() **else** . . .

in the context with result : $(\textbf{string}, H)$. This succeeds. To check with respect to $H, ()\!\prec\!\langle\{\mathit{stat}\}; H\rangle\!\!\rightarrowtail\! L$, note that the assignment to result in the **then** branch of the **test** is not compatible with the context result : $(\textbf{string}, L)$. But, on the assumption that the caller has excluded $\{\mathit{stat}\}$, the second of the rules for **test** is applicable and the **then** branch is not checked.

Leaks via the access mechanism. The security context is in effect an implicit mutable variable, so there is a potential for the access mechanism to be used for information flow. For example, suppose permission *p* is statically authorized for some class and consider the following fragment assuming that eH is a high expression.

if eH **then** (**enable** *p* **in** S1) **else** S2

If p is not enabled initially then the security state for S1 and S2 differs in a way that depends on eH. One could try to exploit this in an assignment to a low variable vL:

test p **then** vL := 0 **else** vL := 1

But note that this assignment is not allowed in the branches of the conditional guarded by eH, as it assigns a low variable. That is, the standard rule for conditionals handles in particular flows via security context. Moreover, because enable is a construct, with an effect limited to a lexically scoped constituent command, p is no longer enabled after the branches join (as highlighted by parentheses above). This is why sound static analysis rules for **test** and **enable** are straightforward.

Consider this alternative: a primitive command "**enable'** p" that enables p and leaves it enabled until, say, the current method body terminates. This could be used to leak information because its effect would be retained following the join of the branches:

(**if** eH **then enable'** p; S1 **else** S2); **test** p **then** vL := 0 **else** vL := 1

To handle **enable'**, a static analysis could track the security level of the security context in the manner of a typestate. This is much like the formulation where the "program counter" security level is tracked, which is convenient for handling exceptional control flows. But we pursue the idea no further.

Subclassing. Here is an untrusted subclass, KernSub of Kern that overrides method Kern.getStatus and inherits Kern.getHinfo. Suppose $Auth$(KernSub) = $\{other\}$.

```
class KernSub extends Kern {
  string getStatus() {
    /* smtypes(getStatus, KernSub) = {H, ()-⟨∅; H⟩→H,  H, ()-⟨{stat}; H⟩→L} */
    enable sys in result := self.getHinfo() } . . . }
```

For simplicity, in section 5 we let $smtypes$(getHinfo, Kern) = $\{H, () \text{-}\langle\emptyset; H\rangle\!\!\to H\}$. However, to take into account callers who do not have permission sys, we have, more precisely, $smtypes$(getHinfo, Kern) = $\{H, () \text{-}\langle\emptyset; H\rangle\!\!\to H,\ H, () \text{-}\langle\{sys\}; H\rangle\!\!\to L\}$.

The definition of $smtypes$ (Definition 5) requires $smtypes$(getStatus, KernSub) = $smtypes$(getStatus, Kern), and thus two checks of the method body. The interesting case is for $H, () \text{-}\langle\{stat\}; H\rangle\!\!\to L$. Here the body of getStatus is checked in the context $\{stat\} \cap Auth$(KernSub), (*not* in context $\{stat\} \cap Auth$(Kern)) i.e., $\{stat\} \cap \{other\}$, which is $\emptyset$. The permission sys is not enabled as it is not in $Auth$(KernSub). Now type checking of getHinfo occurs using type $H, () \text{-}\langle\{sys\}; H\rangle\!\!\to L$ and this succeeds.

Level of self*.* In the examples so far, method types have the form $H, \kappa_1 \text{-}\langle P; \kappa\rangle\!\!\to \kappa_2$. The H in the method type is the type of self. We could have as well typed self as L, because by contravariance in subtyping, $H, \kappa_1 \text{-}\langle P; \kappa\rangle\!\!\to \kappa_2 \leqslant L, \kappa_1 \text{-}\langle P; \kappa\rangle\!\!\to \kappa_2$. But what if a method is *non-anonymous* (Vitek & Bokowski, 2001), i.e. the method returns self as result or passes self as argument? For example, suppose method leakSelf was added to class Kern, defined as follows:

Kern leakSelf() { result := self}

If k is L then the invocation k.leakSelf() returns L but if k is H then it returns H. This poses a difficulty in Banerjee & Naumann (2002c) and Strecker (2003) because methods are checked with respect to a level for self given by the level of the class and a method can be inherited into another class. In the present paper, classes have no level and we simply need

$$smtypes(\mathsf{leakSelf}, \mathsf{Kern}) = \{L, () \dashv\!\langle \varnothing ; H \rangle\!\rightarrow L,\ \ H, () \dashv\!\langle \varnothing ; H \rangle\!\rightarrow H\}$$

In other words, self is treated like any other parameter.

This example also illustrates the benefit of level polymorphism.

Leaks. We consider a class Doctor with a field pat that is a reference to a patient record and a field hivRef that is a reference to an HIVSpecialist. We consider the scenario where a doctor refers a patient to a specialist. The class HIVSpecialist has a field hivPat that is used as an H alias to field pat whose level is L.

Class PatientRecord, below, has fields that are references to public and confidential information about a patient. For example, field name of a patient has level L, whereas field hiv contains confidential information about a patient's hiv status and field drug contains confidential information about medications taken by the patient. Hence both of these fields have level H.

```
class Doctor extends Object {
  pat: (PatientRecord, L);
  hivRef: (HIVSpecialist, L);
  ...}
class HIVSpecialist extends Doctor {
  hivPat: (PatientRecord, H); // will be used as high alias to field pat
  ...}
class PatientRecord extends Object {
  name: (string, L); hiv : (bool, H); drug: (string, H);
  ... }
```

The above classes may be used as follows.

```
class Main extends Object {
  (PatientRecord, L) r := new PatientRecord;
  (Doctor, L) dr := new Doctor;
  (HIVSpecialist, L) hivSp := new HIVSpecialist;
  dr.pat := r;
  dr.hivRef := hivSp;
  dr.hivRef.hivPat := dr.pat; //referral set up
  (PatientRecord, H) hpatient := dr.pat;
  (PatientRecord, L) lpatient := dr.pat;
  ...}
```

The code in Main sets up a referral where dr refers a patient to hivSp. The field accesses dr.pat, dr.hivRef and dr.hivRef.hivPat then have the following types:

dr.pat: (PatientRecord, L), dr.hivRef: (HIVSpecialist, L), and dr.hivRef.hivPat: (PatientRecord, H). Note that hpatient and lpatient are aliases, although their types vary.

Leaks via control flow: conditionals. We first consider the example of a conditional with a high guard.

if lpatient.hiv **then** lpatient.drug := "azt" **else** lpatient.drug := "generic";

Note that lpatient.hiv : (bool, H), hence the guard of the conditional has level H. Were the type of field drug L, the hiv status could be revealed, as lpatient.drug would have level L. The rule for conditionals with high guard requires that both branches have $(com\, H, H)$, as is true here with drug typed H (in this case there are no assignments to variables).

Leaks via control flow: Dynamic dispatch. Next we consider leaks via dynamic dispatch in method calls. To the class PatientRecord, we add the methods setDrug and set. We also add a method leak that branches on self.hiv and returns an object of class YES or NO.

```
class PatientRecord extends Object {
  name: (string, L);   hiv: (bool, H);    drug: (string, H);
  (unit, L) setDrug((string, L) d) {
    /* smtypes(setDrug, PatientRecord) = {H, L-<∅; H>→L} */
    self.drug := d}
  (unit, L) set() {
    /* smtypes(set, PatientRecord) = {H, ()-<∅; H>→L} */
    self.setDrug("azt"); }
  (PatientRecord, H) leak() {
    /* smtypes(leak, PatientRecord) = {H, ()-<∅; H>→H} */
    if self.hiv then result:= new YES else result:= new NO } }
```

Classes YES and NO are subclasses of PatientRecord where the set method is overridden as depicted below:

```
class YES extends PatientRecord {
  (unit, L) set() {
    self.setDrug("azt"); } }
class NO extends PatientRecord {
  (unit, L) set() { self.setDrug("generic"); } }
```

Note that method PatientRecord.leak returns a result of type (PatientRecord, H). This makes sense because the body of the method is a conditional with a high guard, self.hiv. Were the type of field drug L, the hiv status could be revealed as follows.

```
(PatientRecord, H) p := hpatient.leak(); p.set(); . . . p.drug. . .
```

The call to hpatient.leak assigns an object of class YES or class NO to p depending on the patient's hiv status. This can be checked directly, using a type test p **is** YES. But the point of the example is that the subsequent call p.set() is a dynamic dispatch to the overriding set method in class YES or NO that sets the drug field; if drug and p had level L then the expression p.drug would violate confidentiality. Hence, in a method call (e.g. p.set()), if the target object reference (here p) has level H, then only H fields may be updated during the call. Note that *smtypes*(set, PatientRecord) has annotation H on the arrow.

Leaks via aliasing in field update. Leaks can occur due to aliasing of high and low object references and a subsequent update of a low field of the object. The following example, due to Sun Qi, shows how. We will add the field bloodGroup of type (**string**, L) to PatientRecord.

```
class PatientRecord extends Object {(string, L) bloodGroup; . . . }
```

Now we consider a test method in class Main:

```
(string, L) test((bool, H) g) {
  (PatientRecord, L) lp1 := new PatientRecord;
  (PatientRecord, L) lp2 := new PatientRecord;
  (PatientRecord,H) hp;
  (String, L) bg := lp1.bloodGroup; // initial value in lp1's bloodGroup
  if g then hp := lp1 else hp := lp2; // hp aliases lp1 or lp2 depending on g
  hp.bloodGroup := "Z"; // update of L field of H object reference
  if bg = lp1.bloodGroup then result := "no" else result := "yes"; // g is leaked
  }
```

After the field update hp.bloodGroup := "Z", either lp1.bloodGroup is "Z" or lp2.bloodGroup is "Z" depending on which of lp1, lp2 aliases hp. This aliasing, in turn, depends on the H input g. The conditional, if g then hp := lp1 else hp := lp2, is legitimate since only the H variable hp is assigned under the H guard. However, the subsequent field update of bloodGroup allows one to infer the value of g by checking the value of lp1.bloodGroup: if its value is identical to its initial value, then g must have been false. Therefore, we disallow updates to L fields of H object references. The rule for field update in the security type system requires the level of hp to be at most the level of bloodGroup. As this is not the case, the test method is rejected by the type system.

Examples like this might lead one to suggest disallowing any aliasing between L and H variables or fields. But this is hard to enforce while still allowing L-to-H information flows. A natural design pattern is exemplified by the pair lpatient, hpatient where lpatient is the L alias, as in one of the examples above. In that context, a field update lpatient.name := "Sue" would be allowed, but not hpatient.name := "Sue". Note that this means that the object is effectively L from the point of view of the analysis.

Integrity. As remarked in the introduction, confidentiality can be seen as the absence of dependency of L on H, and this can also be read as a form of integrity: Licensed data is not influenced by Hacked data. The following example, an adaptation of one from Abadi & Fournet (2003), shows how our type system can be deployed to check integrity.

Our main example on the use of stack inspection based access control for confidentiality (section 3) considers untrusted or partially trusted code calling trusted code (method getStatus). Now we consider the dual scenario of trusted code calling untrusted code.

```
class NaiveProgram extends Object {
  unit Main() {
    (string, H) s := BadPlugIn.TempFile();
    enable FileIO in File.Delete(s); } }
```

Suppose that NaiveProgram is a trusted class with all permissions. Next, we consider the partially trusted class BadPlugIn whose static permissions do not include *FileIO*.

```
class BadPlugIn extends Object {
  (string, H) TempFile() { result := "...password file..."} }
```

The trusted class File has all permissions and contains the method Delete, where the file deletion operation is protected by a test of permission *FileIO*.

```
class File extends Object {
  unit Delete((string, H) s) {
    test FileIO then Win32.Delete(s) else abort;} }
```

As Abadi and Fournet note, stack inspection will not be able to prevent the deletion of the password file in the call NaiveProgram.Main(). They propose a history-based access control mechanism for such situations. Our static analysis can be used to reject the program.

We decorate BadPlugIn.TempFile() with the (integrity) flow policy

$$smtypes(\mathsf{TempFile}, \mathsf{BadPlugIn}) = \{L, () \prec\!\langle \emptyset ; H \rangle\!\succ\!\rightarrow H\}$$

The code for File.Delete can be checked against flow policy

$$smtypes(\mathsf{Delete}, \mathsf{File}) = \{L, H \prec\!\langle \{FileIO\} ; H \rangle\!\succ\!\rightarrow (),\ L, L \prec\!\langle \emptyset ; H \rangle\!\succ\!\rightarrow ()\}$$

The first flow policy concerns hacked inputs, and is used in a context where *FileIO* is absent. For example, BadPlugIn is welcome to call Delete directly and no harm would occur. The second policy says that callers with permission to delete files should not pass tainted data to Delete – this of course is the point of the example. Indeed, for this policy, the type system checks the body of Delete in the permission context $\{FileIO\} \cap Auth(File)$, i.e., the context $\{FileIO\}$, and Delete is accepted.

Observe that if we check NaiveProgram.Main for the policy $L, () \prec\!\langle \emptyset ; H \rangle\!\succ\!\rightarrow ()$, the call to Delete is rejected by our type system as it violates the flow policy for Delete.

7 Indistinguishability and confinement

In this section we show that if an expression is safe, i.e., accepted by the security typing rules of section 5, and has level L, then it is *read confined*: its value does not depend on H-fields or H-variables. Moreover, if a command is safe and it has level *com* H, H then it is *write confined*: it does not assign to L-fields or L-variables. These two properties are the semantic counterparts of the rules "no read up" and "no write down" that underly information flow control; the terms "simple security" and "*-property" are also used (Bell & LaPadula, 1973).

The formalization uses the indistinguishability relation $\sim$ which is also used to formulate noninterference in section 8. States (h, η) and (h', η') may be indistinguishable to an L observer while having different allocation of objects visible only to H. For this reason, indistinguishability is formalized using a bijective correspondence between those locations in $dom\, h$ and $dom\, h'$ that, informally, are or have been visible to L.

Definition 7
A *typed bijection* is a bijective finite partial function, σ, from *Loc* to *Loc*, such that $loctype(\sigma\, \ell) = loctype\, \ell$ for all ℓ in $dom\, \sigma$. $\square$

In the sequel, σ and its decorated variants range over typed bijections. We treat partial functions as sets of ordered pairs, so $\sigma' \supseteq \sigma$ expresses that σ' is an extension of σ.

Definition 8 (*indistinguishable by L*)
For any σ, we define relations $\sim_\sigma$ for data values, object states, heaps, and stores. To be pedantic we could write ${\sim_\sigma}^\theta$, which would let us precisely specify the typing: ${\sim_\sigma}^\theta \subseteq [\![\theta]\!] \times [\![\theta]\!]$. But in the sequel the category should be clear from context so we just indicate it informally in the following.

$$\begin{array}{lll}
\ell \sim_\sigma \ell' \quad \text{in } [\![C]\!] & \iff & \sigma\,\ell = \ell' \vee \ell = nil = \ell' \\
d \sim_\sigma d' \quad \text{in } [\![T]\!] & \iff & d = d' \quad \text{for primitive types } T \\
s \sim_\sigma s' \quad \text{in } [\![state\, C]\!] & \iff & \forall (f : (T, \kappa)) \in sfields\, C \bullet \kappa = L \Rightarrow sf \sim_\sigma s'f \\
\eta \sim_\sigma \eta' \quad \text{in } [\![\Delta^\dagger]\!] & \iff & \forall (x : (T, \kappa)) \in \Delta \bullet \kappa = L \Rightarrow \eta\, x \sim_\sigma \eta'\, x \\
h \sim_\sigma h' \quad \text{in } [\![Heap]\!] & \iff & dom\, \sigma \subseteq dom\, h \wedge rng\, \sigma \subseteq dom\, h' \wedge \\
 & & \forall \ell, \ell' \bullet \ell \sim_\sigma \ell' \Rightarrow h\,\ell \sim_\sigma h'\,\ell' \\
d \sim_\sigma d' \quad \text{in } [\![T_\bot]\!] & \iff & d = \bot = d' \vee (d \neq \bot \neq d' \wedge d \sim_\sigma d' \text{ in } [\![T]\!])
\end{array}$$

$\square$

For classes C, the formulation above exploits the convention that equations involving partial functions are interpreted as false when the function is undefined. Thus, for $\ell \neq nil$, the relation $\ell \sim_\sigma \ell'$ holds only if ℓ is in $dom\, \sigma$. The last clause, for $T_\bot$, is needed to handle errors (null dereferences) in expressions. There is no need to define $\sim_\sigma$ for command outcomes, $(Heap \otimes \Gamma)_\bot$, because the noninterference condition is termination-insensitive.

Note that $\eta \sim_\sigma \eta'$ depends on the levels given by the context Δ, though this is not explicit in the notation. We have occasion to consider stores for differing contexts, in which case we take care to mention which context is considered. The relations $\sim_\sigma$ for object states, and hence for heaps, depend implicitly on class declarations, but a fixed class table is considered throughout.

To extend our definitions to a lattice of levels larger than H, L, the main change would be that a relation $\sim^{\gamma}_{\sigma}$ is needed for each level γ, to express indistinguishability with respect to that level.

Indistinguishability is not symmetric or reflexive in general. But $h \sim_{\iota} h$ where ι is the identity on $dom\, h$. Limited transitivity and symmetry hold.

Lemma 7.1
For any heaps h_1, h_2, h_3, if $h_1 \sim_{\sigma} h_2$ and $h_2 \sim_{\tau} h_3$ then $h_1 \sim_{\tau\circ\sigma} h_3$.

For any Δ and η_1, η_2, η_3 in $[\![\Delta^{\dagger}]\!]$, if $\eta_1 \sim_{\sigma} \eta_2$ and $\eta_2 \sim_{\tau} \eta_3$ then $\eta_1 \sim_{\tau\circ\sigma} \eta_3$.

If $dom\, \sigma = rng\, \sigma$ then all the relations $\sim_{\sigma}$ are symmetric. $\square$

The proof is straightforward. An important special case is where τ or σ is a partial identity function. Note that for stores the relations $\sim_{\sigma}$, $\sim_{\tau}$, and $\sim_{\tau\circ\sigma}$ are all with respect to the same Δ.

One use of $\sim$ is to formulate, in Lemma 7.2 below, that if a command is typable as $(com\ H, \kappa)$ it does not assign to L-variables, and if it is typable as $(com\ \kappa, H)$ it does not assign to L-fields of objects. For this purpose we use $h \sim_{\iota} h_0$, for initial h and final h_0, where ι is the identity on $dom\, h$. This expresses that no L fields are changed.

Each of our results about the meaning of a class table CT is proved by induction on the approximation chain by which $[\![CT]\!]$ is defined. The induction step is treated as a separate lemma about commands, in which the induction hypothesis is an assumption about the method environment. That is the purpose of the following.

Definition 9 (write confined method environment)
Method environment μ is *write confined*, written *wconf* μ, if the following holds for all C, m and all $\kappa, \bar{\kappa}\!-\!\langle P\,; H\rangle\!\!\rightarrow\!\kappa_1$ in $smtypes(m, C)$. If $Q \# P$ and $\mu Cm(h, \eta)Q \neq \bot$ then $h \sim_{\iota} h_0$, where $(h_0, d) = \mu Cm(h, \eta)Q$ and ι is the identity on $dom\, h$. $\square$

Lemma 7.2 (write confinement of commands)
Suppose $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2)$ and *wconf* μ. For all η, h, Q such that $Q \# P$, and $Q \subseteq Auth(\Delta^{\dagger}\ \mathsf{self})$, if $(h_0, \eta_0) = [\![\Delta^{\dagger} \vdash S^{\dagger}]\!]\mu(h, \eta)Q$ then

$$\kappa_1 = H \ \Rightarrow\ \eta \sim_{\iota} \eta_0 \quad \text{and} \quad \kappa_2 = H \ \Rightarrow\ h \sim_{\iota} h_0$$

where ι is the identity on $dom\, h$.

Note that no condition is imposed if $[\![\Delta^{\dagger} \vdash S^{\dagger}]\!]\mu(h, \eta)Q = \bot$.

Proof
We proceed by induction on a derivation of $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2)$ and thus by cases on the last rule used in the derivation. In every case except the subsumption rule, the antecedents involve proper constituents S' of S so in those cases we say "induction on S'".

Case $\Delta, x : (T, \kappa); P \vdash x := e : (com\ \kappa, H)$: Recall the semantic definition

$$[\![\Delta^{\dagger} \vdash x := e]\!]\mu(h, \eta)Q \quad = \quad \mathsf{let}\ d = [\![\Delta^{\dagger} \vdash e : T]\!](h, \eta)\ \mathsf{in}\ (h, [\eta \mid x \mapsto d])$$

Thus η_0, h_0 in the statement of the Lemma have the values $\eta_0 = [\eta \mid x \mapsto d]$ and $h_0 = h$. For the store, we need to show that $\kappa = H \ \Rightarrow\ \eta \sim_{\iota} \eta_0$. This follows using definition $\sim_{\iota}$ for $\Delta, x : (T, H)$. For the heap, we have $h \sim_{\iota} h_0$ by reflexivity of $\sim_{\iota}$.

In subsequent proof cases we let h_0, η_0 be as in the statement of the Lemma, and other identifiers be as in the relevant semantic definition and security typing rule, with only occasional reminders of this convention. This saves ink at the cost of some page flipping for the reader.

Case $\Delta; P \vdash e_1.f := e_2 : (com\ \ H, \kappa)$: Let $\ell = [\![\Delta^\dagger \vdash e_1 : C]\!](h, \eta)$ and $d = [\![\Delta^\dagger \vdash e_2 : T]\!](h, \eta)$ as in the semantic definition. Thus $h_0 = [h \mid \ell \mapsto [h\ell \mid f \mapsto d]]$ and $\eta_0 = \eta$. Field assignment has no effect on the store, so $\eta \sim_\iota \eta_0$ holds by definition $\sim_\iota$. For the heap we must show $\kappa = H \Rightarrow h \sim_\iota h_0$. The rule requires f to have level κ, and if κ is H then $h \sim_\iota h_0$ holds by definition $\sim_\iota$.

Case $\Delta; x : (D, \kappa); P \vdash x := \mathbf{new}\ B\ : (com\ \kappa, H)$: Let $\ell = fresh(B, h)$, so that $\eta_0 = [\eta \mid x \mapsto \ell]$ and $h_0 = [h \mid \ell \mapsto [fields\, B \mapsto defaults]]$ by semantics. For the store, if $\kappa = H$ then $\eta \sim_\iota \eta_0$ follows by definition of $\sim_\iota$. For the heap we have $h \sim_\iota h_0$ by definition of $\sim_\iota$, as ι is the identity on $dom\, h$ and thus does not involve ℓ.

Case $\Delta, x : (T, \kappa); P \vdash x := e.m(\bar{e}) : (com\ \kappa, \kappa_1)$: Let $\ell = [\![\Delta^\dagger \vdash e : D]\!](h, \eta)$ and $\bar{d} = [\![\Delta^\dagger \vdash \bar{e} : \bar{U}]\!](h, \eta)$ as in the semantics. As we only need to consider the non-$\perp$ case, there is d_0 with $(h_0, d_0) = \mu(loctype\, \ell)m(h, \eta_1)Q$, where $\eta_1 = [\bar{x} \mapsto \bar{d}, \mathsf{self} \mapsto \ell]$, and we have $\eta_0 = [\eta \mid x \mapsto d_0]$.

For the store, if $\kappa = H$ then $\eta \sim_\iota [\eta \mid x \mapsto d_0]$ follows by definition of $\sim_\iota$ for $\Delta, x : (T, H)$. For the heap, suppose $\kappa_1 = H$. Then we must show $h \sim_\iota h_0$. By the security typing rule there is some $(\kappa_0', \bar{\kappa}' \multimap \langle P'; \kappa_1' \rangle \to \kappa') \in smtypes(m, D)$, and so $(\kappa_0', \bar{\kappa}' \multimap \langle P'; \kappa_1' \rangle \to \kappa') \in smtypes(m, (loctype\, \ell))$ by ordinary typing ($loctype\, \ell \leqslant D$) and Definition 5 of annotated class table. Now, by the subtyping condition in the rule we have $(\kappa_0', \bar{\kappa}' \multimap \langle P'; \kappa_1' \rangle \to \kappa') \leqslant (\kappa_0, \bar{\kappa} \multimap \langle P'; \kappa_1 \rangle \to \kappa)$ which by definition implies $\kappa_1 \leqslant \kappa_1'$. Hence $\kappa_1' = H$. Using hypotheses $Q \# P$ and $Q \sqsubseteq Auth(\Delta^\dagger\ \mathsf{self})$ we can apply Lemma 5.1 to get $Q \# P'$. So we can use assumption *wconf* μ of the present Lemma to get $h \sim_\iota h_0$.

Case $\Delta; P \vdash S_1;\ S_2 : (com, \kappa_1, \kappa_2)$: Let $(h_1, \eta_1) = [\![\Delta^\dagger \vdash S_1^\dagger]\!]\mu(h, \eta)Q$ and thus $(h_0, \eta_0) = [\![\Delta^\dagger \vdash S_2^\dagger]\!]\mu(h_1, \eta_1)Q$. For the store, assume $\kappa_1 = H$. Then by induction on (the derivation for) S_1 we get $\eta \sim_\iota \eta_1$. So by induction on S_2, we get $\eta_1 \sim_\iota \eta_0$. Now by Lemma 7.1, we get $\eta \sim_\iota \eta_0$. For the heap, assume $\kappa_2 = H$. Now $h \sim_\iota h_0$ follows by induction on S_1 followed by induction on S_2 and finally Lemma 7.1.

Case $\Delta; P \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 : (com, \kappa_1, \kappa_2)$: Let $b = [\![\Delta^\dagger \vdash e : \mathbf{bool}]\!](h, \eta)$. For the store, assume $\kappa_1 = H$. Then if $b = true$ (resp. $b = false$) the result $\eta \sim_\iota \eta_0$ follows by induction on the derivation for S_1 (resp. S_2). Similarly for the heap.

Case $\Delta; P \vdash (T, \kappa)\ x := e\ \mathbf{in}\ S : (com\ \kappa_1, \kappa_2)$: Let $d = [\![\Delta^\dagger \vdash e : T]\!](h, \eta)$ and thus $(h_0, \eta_0) = [\![(\Delta^\dagger, x : T) \vdash S^\dagger]\!]\mu(h, [\eta \mid x \mapsto d])Q$. (Note that the rule allows $x : (T, L)$ but if S assigns to x then $\kappa_1 = L$.) For the store, assume $\kappa_1 = H$. Then by induction on S we get $[\eta \mid x \mapsto d] \sim_\iota \eta_0$ with respect to $\Delta, x : (T, \kappa)$, hence $\eta \sim_\iota (\eta_0 \lfloor x)$ with respect to Δ. For the heap, if $\kappa_2 = H$ then by induction on S we get $h \sim_\iota h_0$.

Case $\Delta; P \vdash \mathbf{enable}\ P'\ \mathbf{in}\ S : (com\ \kappa_1, \kappa_2)$: By hypothesis of the rule we have

$$\Delta; (P - Q') \vdash S : (com\ \kappa_1, \kappa_2)$$

where $Q' = P' \cap Auth(\Delta^\dagger \mathsf{self})$. By semantics, $(h_0, \eta_0) = [\![\Delta^\dagger \vdash S]\!]\mu(h,\eta)(Q \cup Q')$. So, to use induction on S it suffices to show that $(Q \cup Q') \# (P - Q')$. This follows by set theory from hypothesis $Q \# P$.

Case $\Delta; P \vdash$ **test** P' **then** S_1 **else** $S_2 : (com\ \kappa_1, \kappa_2)$: We argue by cases on whether $P' \subseteq Q$. If $P' \subseteq Q$ then $P' \# P$ by hypothesis $Q \# P$ and $P' \subseteq Auth(\Delta^\dagger \mathsf{self})$ by hypothesis $Q \subseteq Auth(\Delta^\dagger \mathsf{self})$. Thus the first of the rules for **test** in Figure 8 must have been used. Since $P' \subseteq Q$, the test condition is true and the semantics is given by semantics of S_1. We have $\Delta; P \vdash S_1 : (com\ \kappa_1, \kappa_2)$ by hypothesis of the rule, so to obtain both results $\eta \sim_\iota \eta_0$ and $h \sim_\iota h_0$ we can use induction on S_1.

In the other subcase, $P' \not\subseteq Q$, the test condition is false so the semantics is given by semantics of S_2. Both of the rules for **test** P' **then** S_1 **else** S_2 have hypothesis $\Delta; P \vdash S_2 : (com\ \kappa_1, \kappa_2)$ so we get both $\eta \sim_\iota \eta_0$ and $h \sim_\iota h_0$ by induction on S_2.

Case $\Delta; P \vdash S : (com\ \kappa_3, \kappa_4)$ by subsumption rule: Let $(h_0, \eta_0) = [\![\Delta^\dagger \vdash S]\!]\mu(h,\eta)Q$. For the store, suppose $\kappa_3 = H$. By the rule, $\kappa_3 \leqslant \kappa_1$, so we have $\kappa_1 = H$. Then the result $\eta \sim_\iota \eta_0$ follows by induction on the smaller derivation tree for $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2)$. For the heap, if $\kappa_4 = H$ then by the typing rule we have $\kappa_4 \leqslant \kappa_2$, so $\kappa_2 = H$ and the result $h \sim_\iota h_0$ follows by induction on the derivation of the rule's hypothesis $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2)$. $\square$

Lemma 7.3 (*safe programs are write confined*)
If annotated class table CT is safe then $wconf\,[\![CT^\dagger]\!]$ and also $wconf\ \mu_i$ for each μ_i in the approximation chain for semantics of CT (see Definition 2).

Proof
First, we show the following, by induction on i:

$$wconf\ \mu_i \quad \text{for all } i \in \mathbb{N} \tag{4}$$

For the base case, $wconf\ \mu_0$ holds because the antecedent $\mu_0 Cm(h,\eta)Q \neq \bot$ in the definition of $wconf$ is falsified by μ_0. For the induction step, suppose that $wconf\ \mu_i$. To show $wconf\ \mu_{i+1}$, consider any C, m, and $\kappa, \bar{\kappa} \text{−}\langle P; H \rangle \!\!\rightarrow \kappa_1$ in $smtypes(m, C)$. For any Q, h, η with $Q \# P$, suppose $(h_0, d) = \mu_{i+1} Cm(h,\eta)Q$. We must show $h \sim_\iota h_0$ where ι is the identity on $dom\, h$. The argument is by cases on whether m is inherited in C.

If m is inherited in C from B then, by definition of μ, we have $\mu_{i+1} Cm(h,\eta)Q = \mu_{i+1} Bm(h,\eta)Q$. So we can obtain the result for C directly from the result for B, using a secondary induction on inheritance chains.

For the case that m has declaration M in C, we have $\mu_{i+1} Cm(h,\eta)Q = [\![M]\!]\mu_i(h,\eta)Q$. Suppose $mtype(m, C) = \bar{T} \rightarrow T$ and $pars(m, C) = \bar{x}$. By Definition 3 we have

$$\begin{aligned}[\![M]\!]\mu_i(h,\eta)Q = \ & \mathsf{let}\ Q_1 = Q \cap Auth\ C\ \mathsf{in} \\ & \mathsf{let}\ \eta_1 = [\eta \mid \mathsf{result} \mapsto default]\ \mathsf{in} \\ & \mathsf{let}\ (h_0, \eta_0) = [\![\bar{x} : \bar{T}, \mathsf{self} : C, \mathsf{result} : T \vdash S]\!]\mu_i(h,\eta)Q_1\ \mathsf{in} \\ & (h_0, \eta_0\ \mathsf{result})\end{aligned}$$

By the security typing rule for method declaration with respect to $\kappa, \bar{\kappa} \multimap \langle P\,; H \rangle \!\!\rightarrow\! \kappa_1$, there is some local variant S' of S such that

$$\mathsf{self} : (C, \kappa), \bar{x} : (\bar{T}, \bar{\kappa}), \mathsf{result} : (T, \kappa_1);\ (P \cap Auth\ C) \ \vdash\ S' : (com\ L, H)$$

As $Q_1 \subseteq Auth\ C$, we can apply Lemma 7.2, for μ_i and S', noting that $S'^{\dagger} = S^{\dagger}$, to obtain $h \sim_{\iota} h_0$. This concludes the proof of (4).

Finally, we prove that (4) implies $wconf\ (lub\ \mu)$.[2] Suppose μ is an ascending chain and we have $\forall i \bullet wconf\ \mu_i$. To show $wconf(lub\ \mu)$, consider any C, m, and $\kappa, \bar{\kappa} \multimap \langle P\,; H \rangle \!\!\rightarrow\! \kappa_1$ in $smtypes(m, C)$. Consider any h, η and Q such that $Q \# P$ and $(lub\ \mu)Cm(h, \eta)Q \neq \bot$. Let $(h_0, d) = (lub\ \mu)Cm(h, \eta)Q$. We must show $h \sim_{\iota} h_0$ where ι is the identity on $dom\, h$. By Lemma 4.1 there is j with $(lub \mu)Cm(h, \eta)Q = \mu_j Cm(h, \eta)Q$ and by hypothesis $wconf\ \mu_j$ we obtain $h \sim_{\iota} h_0$. $\square$

The last result in this section can be seen as a simple form of noninterference. It says that if an expression can be typed $\Delta \vdash e : (T, L)$ then its meaning is the same in two L-indistinguishable states.

Lemma 7.4 (*safe expressions are read confined*)
Suppose $\Delta \vdash e : (T, L)$ and $h \sim_{\sigma} h'$ and $\eta \sim_{\sigma} \eta'$. If $d = [\![\Delta^{\dagger} \vdash e : T]\!](h, \eta)$ and $d' = [\![\Delta^{\dagger} \vdash e : T]\!](h', \eta')$ then $d \sim_{\sigma} d'$.

Proof
The proof is by induction on a derivation of $\Delta \vdash e : (T, L)$ with cases on the last rule used.

In this proof and subsequent ones, we extend the convention described earlier in which we refer to identifiers in the semantic definitions without explicit mention in the proof. When comparing semantics for a pair of states (h, η) and (h', η'), we use corresponding primes on identifiers in the semantic definitions.

Case $\Delta \vdash x : \Delta x$: Then $[\![\Delta^{\dagger} \vdash x : T]\!](h, \eta) = \eta x$ and $[\![\Delta^{\dagger} \vdash x : T]\!](h', \eta') = \eta' x$. We must show $\eta x \sim_{\sigma} \eta' x$. This follows by assumptions $\eta \sim_{\sigma} \eta'$ and $\Delta\, x = (T, L)$.

Case $\Delta \vdash e_1 = e_2 : (\mathbf{bool}, L)$: Let $d_1 = [\![\Delta^{\dagger} \vdash e_1 : T_1]\!](h, \eta)$ and let $d_2 = [\![\Delta^{\dagger} \vdash e_2 : T_2]\!](h, \eta)$. We must show $(d_1 = d_2) \sim_{\mathbf{bool}} (d'_1 = d'_2)$, that is, $d_1 = d_2$ iff $d'_1 = d'_2$. By the typing rule, we have $\Delta \vdash e_1 : (T_1, L)$ and $\Delta \vdash e_2 : (T_2, L)$. By induction on (the derivation of) e_1 we obtain $d_1 \sim_{\sigma} d'_1$ and induction on e_2 yields $d_2 \sim_{\sigma} d'_2$. The result follows because σ is bijective.

Case $\Delta \vdash e.f : (T, L)$: By the typing rule, $\kappa \sqcup \kappa_1 = L$, so $\kappa = L = \kappa_1$. Hence we can use induction on e; this yields locations ℓ, ℓ' where $\ell = [\![\Delta^{\dagger} \vdash e : C]\!](h, \eta)$ and $\ell' = [\![\Delta^{\dagger} \vdash e : C]\!](h', \eta')$ and $\ell \sim_{\sigma} \ell'$. (Or else both are $\bot$ or both are *nil* which leads to result $\bot$, and $\bot \sim_{\sigma} \bot$ by definition.) From $h \sim_{\sigma} h'$, by definition of $\sim_{\sigma}$ on heaps we have $h\ell \sim_{\sigma} h'\ell'$. Thus $h\ell f \sim_{\sigma} h'\ell' f$ by definition of $\sim_{\sigma}$ on object states and we are done.

[2] This implication is exactly the admissibility of *wconf*; by admissibility, one obtains *wconf* $(lub\ \mu)$ by fixpoint induction. We refrain from formulating $(lub\ \mu)$ as a fixpoint.

Case $\Delta \vdash (B)\ e : (B, L)$: Let $\ell = [\![\Delta^\dagger \vdash (B)\ e : B]\!](h, \eta)$ and likewise for ℓ'. By the typing rule, we have $\Delta \vdash e : (D, L)$ where $B \leqslant D$. Hence by induction on e we get $\ell \sim_\sigma \ell'$, that is, either $\ell' = \sigma\ \ell$ or $\ell = nil = \ell'$ or $\ell = \bot = \ell'$. The latter two cases are easy. For proper locations $\ell' = \sigma\ \ell$, we have $loctype\, \ell = loctype\, \ell'$ because σ is a typed bijection; hence $loctype\, \ell \leqslant B$ iff $loctype\, \ell' \leqslant B$. So by semantics the results are either $\bot, \bot$ or ℓ, ℓ' and in either case they are related by $\sim_\sigma$.

Case $\Delta \vdash e : (T, L)$ by the subsumption rule: By the rule, $\kappa \leqslant L$ and hence $\kappa = L$. We can use induction on the smaller derivation tree for $\Delta \vdash e : (T, L)$ to obtain the result $[\![\Delta^\dagger \vdash e : T]\!](h, \eta) \sim_\sigma [\![\Delta^\dagger \vdash e : T]\!](h', \eta')$.

Case e **is** B, **null**, **true**, **false** : these are straightforward. □

8 Safety implies noninterference

This section proves the main result: if a class table is accepted by the security typing rules then the method environment that it denotes satisfies noninterference. That is, if it is safe with respect to a given flow policy then its semantics for the given access policy does satisfy the flow policy.

8.1 Satisfaction and noninterference

Noninterference for a class table is defined in terms of noninterference of method meanings with respect to their security types.

Definition 10 (*satisfaction*)
Suppose $d \in [\![C, \bar{x}, \bar{T} \to T]\!]$ and the length of $\bar{\kappa}$ is the same as $\bar{x}$. Then d *satisfies* a typing $\kappa_0, \bar{\kappa} \prec\!\langle P\,; \kappa_1 \rangle\!\!\rightarrowtail \kappa_2$ iff for all $\sigma, h, h', \eta, \eta', Q$, if $h \sim_\sigma h'$, $\eta \sim_\sigma \eta'$, $Q \# P$, $(h_0, d_0) = d(h, \eta)Q$, and $(h'_0, d'_0) = d(h', \eta')Q$, then there is $\tau \supseteq \sigma$ such that $h_0 \sim_\tau h'_0$ and $(\kappa_2 = L \Rightarrow d_0 \sim_\tau d'_0)$. □

Note that $\eta \sim_\sigma \eta'$ is used here with respect to Γ defined as $\Gamma = [\mathsf{self} : (C, \kappa_0), \bar{x} : (\bar{T}, \bar{\kappa})]$. The significance of this point is illuminated in the proof of Proposition 8.1 below.

Definition 11 (*noninterfering method environment*)
A method environment is noninterfering, written *nonint* μ, iff for all C, m, the meaning $\mu C m$ satisfies every $\kappa_0, \bar{\kappa} \prec\!\langle P\,; \kappa_1 \rangle\!\!\rightarrowtail \kappa_2$ in $smtypes(m, C)$. □

Our main result is that the method environment denoted by a secure class table is noninterfering. The proof uses lemmas which express noninterference for the expression and command constructs, respectively. First, we explore properties of security subtyping, which in passing justifies the use of a set of types for methods.

Proposition 8.1 (*satisfaction is monotonic*)
Suppose that $\kappa_0, \bar{\kappa} \prec\!\langle P\,; \kappa_1 \rangle\!\!\rightarrowtail \kappa_2 \leqslant \kappa'_0, \bar{\kappa}' \prec\!\langle P'\,; \kappa'_1 \rangle\!\!\rightarrowtail \kappa'_2$. If d in $[\![C, \bar{x}, \bar{T} \to T]\!]$ satisfies $\kappa_0, \bar{\kappa} \prec\!\langle P\,; \kappa_1 \rangle\!\!\rightarrowtail \kappa_2$ then it also satisfies $\kappa'_0, \bar{\kappa}' \prec\!\langle P'\,; \kappa'_1 \rangle\!\!\rightarrowtail \kappa'_2$.

Proof
Suppose d is in $[\![C, \bar{x}, \bar{T} \to T]\!]$. To show that d satisfies $\kappa'_0, \bar{\kappa}' \text{-}\!\!\langle P' ; \kappa'_1 \rangle\!\!\to \kappa'_2$, suppose that $h \sim_\sigma h'$, $\eta \sim_\sigma \eta'$, and $Q \# P'$. Suppose $(h_0, d_0) = d(h, \eta)Q$ and $(h'_0, d'_0) = d(h', \eta')Q$. Note that here $\eta \sim_\sigma \eta'$ is with respect to $\Delta' = [\mathsf{self} : (C, \kappa'_0), \bar{x} : (\bar{T}, \bar{\kappa}')]$.

From $\kappa_0, \bar{\kappa} \text{-}\!\!\langle P ; \kappa_1 \rangle\!\!\to \kappa_2 \leqslant \kappa'_0, \bar{\kappa}' \text{-}\!\!\langle P' ; \kappa'_1 \rangle\!\!\to \kappa'_2$ we have $\kappa'_0 \leqslant \kappa_0$ and $\bar{\kappa}' \leqslant \bar{\kappa}$. Thus by definition of $\sim$ we get $\eta \sim_\sigma \eta'$ with respect to $\Delta = [\mathsf{self} : (C, \kappa_0), \bar{x} : (\bar{T}, \bar{\kappa})]$. By $P \subseteq P'$ we have $Q \# P$. Now we can use that d satisfies $\kappa_0, \bar{\kappa} \text{-}\!\!\langle P ; \kappa_1 \rangle\!\!\to \kappa_2$ to obtain $\tau \supseteq \sigma$ such that $h_0 \sim_\tau h'_0$ and $\kappa_2 = L \Rightarrow d_0 \sim_\tau d'_0$. It remains to show that $\kappa'_2 = L$ implies $d_0 \sim_\tau d'_0$; this follows using $\kappa_2 \leqslant \kappa'_2$. □

Because there are finitely many security levels and permissions, the ordered set of method typings is finite. In fact it is a lattice, because permission sets and security levels form lattices. We can express the greatest lower bound of $\kappa_0, \bar{\kappa} \text{-}\!\!\langle P ; \kappa_1 \rangle\!\!\to \kappa_2$ and $\kappa'_0, \bar{\kappa}' \text{-}\!\!\langle P' ; \kappa'_1 \rangle\!\!\to \kappa'_2$ pointwise, using $\sqcap$ on levels in contravariant positions:

$$(\kappa_0 \sqcup \kappa'_0), (\bar{\kappa} \sqcup \bar{\kappa}') \text{-}\!\!\langle (P \cap P') ; (\kappa_1 \sqcup \kappa'_1) \rangle\!\!\to (\kappa_2 \sqcap \kappa'_2)$$

As remarked in section 3, satisfaction does not distribute over $\sqcap$.

Example 1
The denotation of method getStatus satisfies both $H, () \text{-}\!\!\langle \{stat\} ; H \rangle\!\!\to L$ and $H, () \text{-}\!\!\langle \varnothing ; H \rangle\!\!\to H$. But $(H, () \text{-}\!\!\langle \{stat\} ; H \rangle\!\!\to L) \sqcap (H, () \text{-}\!\!\langle \varnothing ; H \rangle\!\!\to H) = H, () \text{-}\!\!\langle \varnothing ; H \rangle\!\!\to L$, and getStatus does not satisfy $H, () \text{-}\!\!\langle \varnothing ; H \rangle\!\!\to L$.

The proof of the main theorem goes by proving safety of the approximation chain. The final step, to safety of $[\![CT]\!]$, we give as a separate result.

Lemma 8.2 (*noninterference is admissible*)
Suppose $\mu \in \mathbb{N} \to [\![MEnv]\!]$ is an ascending chain and *nonint* μ_i for all i. Then *nonint* (*lub* μ).

Proof
To show *nonint* (*lub* μ), consider any C, m, and $\kappa, \bar{\kappa} \text{-}\!\!\langle P ; \kappa_1 \rangle\!\!\to \kappa_2$ in *smtypes*(m, C). We must show that $(lub\, \mu)Cm$ satisfies $\kappa, \bar{\kappa} \text{-}\!\!\langle P ; \kappa_1 \rangle\!\!\to \kappa_2$.

Consider any $\sigma, h, h', \eta, \eta', Q$ such that $h \sim_\sigma h'$, $\eta \sim_\sigma \eta'$, and $Q \# P$. Suppose $(h_0, \eta_0) = (lub\, \mu)Cm(h, \eta)Q$ and $(h'_0, \eta'_0) = (lub\, \mu)Cm(h', \eta')Q$. By Lemma 4.1 there are j, j' such that $(h_0, \eta_0) = \mu_k Cm(h, \eta)Q$ for all $k \geqslant j$ and $(h'_0, \eta'_0) = \mu_{k'} Cm(h', \eta')Q$ for all $k' \geqslant j'$. Choosing i to be the maximum of j and j' yields $(h_0, \eta_0) = \mu_i Cm(h, \eta)Q$ and $(h'_0, \eta'_0) = \mu_i Cm(h', \eta')Q$. Then to conclude the argument for satisfaction by $(lub\, \mu)$ it suffices to use satisfaction by μ_i. □

8.2 Main result

Lemma 8.3 (*safe commands are noninterfering*)
Suppose $\Delta; P \vdash S : (com\ \ \kappa_1, \kappa_2)$, *wconf* μ, and *nonint* μ. Suppose also $Q \# P$, $Q \subseteq Auth(\Delta^\dagger\ \mathsf{self})$, $\eta \sim_\sigma \eta'$, $h \sim_\sigma h'$, $(h_0, \eta_0) = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h, \eta)Q$ and $(h'_0, \eta'_0) = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h', \eta')Q$. Then there is $\tau \supseteq \sigma$ such that $\eta_0 \sim_\tau \eta'_0$ and $h_0 \sim_\tau h'_0$.

Proof
We proceed by induction on a derivation of $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2)$ with cases on the last rule used in the derivation. In each case we define τ and show $\eta_0 \sim_\tau \eta'_0$ and $h_0 \sim_\tau h'_0$. Only **new** and method calls allocate new objects; in all other cases for primitive commands, namely, assignment and field update, we take $\tau = \sigma$.

Recall the convention described earlier: we refer to identifiers in the semantic definitions and security typing rules.

Case $\Delta, x : (T, \kappa); P \vdash x := e : (com\ \kappa, H)$: Recall the semantic definition

$$[\![\Delta^\dagger \vdash x := e]\!]\mu(h,\eta)Q \quad = \quad \mathsf{let}\ d = [\![\Delta^\dagger \vdash e : T]\!](h,\eta)\ \mathsf{in}\ (h, [\eta \mid x \mapsto d])$$

Thus η_0, h_0 in the statement of the Lemma have the values $\eta_0 = [\eta \mid x \mapsto d]$ and $h_0 = h$. We must show $[\eta \mid x \mapsto d] \sim_\sigma [\eta' \mid x \mapsto d']$. By hypothesis, $\eta \sim_\sigma \eta'$, so it only remains to show that $\kappa = L$ implies $d \sim_\sigma d'$. This follows from Lemma 7.4 on e. As assignment has no effect on the heap, we have $h_0 \sim_\sigma h'_0$ by assumption $h \sim_\sigma h'$.

Case $\Delta; P \vdash e_1.f := e_2 : (com\ H, \kappa)$: Let $\ell = [\![\Delta^\dagger \vdash e_1 : C]\!](h,\eta)$ and let $d = [\![\Delta^\dagger \vdash e_2 : T]\!](h,\eta)$ as in the semantic definition. Recall that we are proving termination-insensitive noninterefereence and thus considering non-$\perp$ outcomes, so $\ell \neq nil$ and $d \neq \perp$. Thus $h_0 = [h \mid \ell \mapsto [h\ell \mid f \mapsto d]]$ and $\eta_0 = \eta$. Field assignment has no effect on the store, so $\eta_0 \sim_\sigma \eta'_0$ holds by assumption. We show $h_0 \sim_\sigma h'_0$ by cases on κ. If $\kappa = H$, then by Lemma 7.2 and *wconf* μ we have $h \sim_\iota h_0$ and $h' \sim_{\iota'} h'_0$ where ι (resp. ι') is the identity on *dom* h (resp. *dom* h'). But by hypothesis, $h \sim_\sigma h'$. Hence by Lemma 7.1, we get $h_0 \sim_\sigma h'_0$.

For the case $\kappa = L$, note that the heap is only modified at ℓ, ℓ'. Unfolding the definition of $h_0 \sim_\sigma h'_0$, we must show

$$\ell \sim_\sigma \ell'' \Rightarrow h_0\ell \sim_\sigma h'_0\ell'' \quad \text{for all } \ell'' \tag{5}$$

and, symmetrically, $\ell'' \sim_\sigma \ell' \Rightarrow h_0\ell'' \sim_\sigma h'_0\ell'$ for all ℓ''. The rule requires $\kappa_1 \leqslant \kappa$, so $\kappa_1 = L$ and by Lemma 7.4 for e_1 we have $\ell \sim_\sigma \ell'$. Because σ is bijective on locations, $\ell \sim_\sigma \ell''$ implies $\ell'' = \ell'$ so this is the only instance of (5) to consider. The rule requires the level of e_2 to be κ, i.e., L; so by Lemma 7.4 for e_2 we have $d \sim_\sigma d'$ and hence $h_0\ell \sim_\sigma h'_0\ell'$.[3]

Case $\Delta, x : (D, \kappa); P \vdash x := \textbf{new}\ B\ : (com\ \kappa, H)$: Let $\ell = fresh(B, h)$ so that $\eta_0 = [\eta \mid x \mapsto \ell]$ and $h_0 = [h \mid \ell \mapsto [fields\, B \mapsto defaults]]$ by semantics. Note that $h_0 \sim_\sigma h'_0$ by freshness of ℓ, ℓ' and the requirement (in the definition of $h \sim_\sigma h'$) that the domain (resp. range) of σ is contained in $dom\, h$ (resp. $dom\, h'$).

In the case $\kappa = H$ we can take $\tau = \sigma$ as we have $\eta_0 \sim_\tau \eta'_0$ from $\eta \sim_\sigma \eta'$ and $x : (D, H)$.

For the case $\kappa = L$ we define $\tau = \sigma \cup \{(\ell, \ell')\}$ which yields $\eta_0 \sim_\tau \eta'_0$ because $\ell \sim_\tau \ell'$. Moreover, $h_0 \sim_\tau h'_0$ as the default values $(nil, false)$ are related to themselves. Note

[3] The penultimate example in section 6 depicts how leaks via aliasing in field update can occur thus showing necessity of the condition $\kappa_1 \leqslant \kappa$. The example leads to a situation where $\ell \sim_\sigma \ell''$ for $\ell'' \neq \ell'$, where ℓ is the value of hp, but ℓ'' is the value of its low alias, lp1 or lp2.

that τ is a bijection between subsets of $dom\,h_0$ and $dom\,h'_0$ by freshness, and it respects types because $loctype\,\ell = B = loctype\,\ell'$.

Case $\Delta, x:(T,\kappa); P \vdash x := e.m(\bar{e}) : (com\ \kappa,\kappa_1)$: Let $\ell = [\![\Delta^\dagger \vdash e : D]\!](h,\eta)$ and $\bar{d} = [\![\Delta^\dagger \vdash \bar{e} : \bar{U}]\!](h,\eta)$ as in the semantics. As we only need to consider the non-$\bot$ case, there is d_0 with $(h_0,d_0) = \mu(loctype\,\ell)m(h,\eta_1)Q$, where $\eta_1 = [\bar{x} \mapsto \bar{d}, \mathsf{self} \mapsto \ell]$, and we have $\eta_0 = [\eta \mid x \mapsto d_0]$.

We go by cases on the level, κ_0, of e. If $\kappa_0 = H$ then it is possible that $\neg(\ell \sim_\sigma \ell')$ and thus the two calls can have different behavior. By the security typing rule there is some $(\kappa'_0, \bar{\kappa}' \multimap \langle P'; \kappa'_1 \rangle \rightarrowtail \kappa') \in smtypes(m,D)$ and hence $(\kappa'_0, \bar{\kappa}' \multimap \langle P'; \kappa'_1 \rangle \rightarrowtail \kappa') \in smtypes(m,(loctype\,\ell))$ by ordinary typing ($loctype\,\ell \leqslant D$) and Definition 5 of annotated class table. Now, by the subtyping condition in the rule we have

$$(\kappa'_0, \bar{\kappa}' \multimap \langle P'; \kappa'_1 \rangle \rightarrowtail \kappa') \leqslant (\kappa_0, \bar{\kappa} \multimap \langle P'; \kappa_1 \rangle \rightarrowtail \kappa)$$

which by definition implies $\kappa_1 \leqslant \kappa'_1$. By the typing rule we have, $\kappa_0 \leqslant \kappa \sqcap \kappa_1$. Thus $\kappa = \kappa_1 = \kappa'_1 = H$. Using hypotheses $Q \# P$ and $Q \subseteq Auth(\Delta^\dagger\ \mathsf{self})$ we can apply Lemma 5.1 to get $Q \# P'$. So we can use assumption $wconf\ \mu$ of the present lemma to get $h \sim_\iota h_0$ (resp. $h' \sim_{\iota'} h'_0$) where ι is the identity on $dom\,h$ (resp. ι' is the identity on $dom\,h'$). Hence using $h \sim_\sigma h'$ and Lemma 7.1 we can define $\tau = \sigma$ and obtain $h_0 \sim_\tau h'_0$. And, note that $\eta_0 \sim_\sigma \eta'_0$ follows from assumption $\eta \sim_\sigma \eta'$ because the level of x is H.

It remains to consider the case $\kappa_0 = L$. In this case, we have $\ell \sim_\sigma \ell'$ by Lemma 7.4 applied to e. Because $Q \# P$, and $Q \subseteq Auth(\Delta^\dagger \mathsf{self})$ we have $Q \# P'$ by Lemma 5.1. We claim that $\eta_1 \sim_\sigma \eta'_1$. Then we can use assumption $nonint\ \mu$ of the present lemma. This means, by Definition 11, that $\mu(loctype\,\ell)m$ satisfies $(\kappa'_0, \bar{\kappa}' \multimap \langle P'; \kappa'_1 \rangle \rightarrowtail \kappa') \in smtypes(m,(loctype\,\ell))$. Now we can apply the definition of satisfaction (Definition 10) to conclude that there exists $\tau \supseteq \sigma$ with $\kappa' = L \Rightarrow d_0 \sim_\tau d'_0$, whence $\eta_0 \sim_\tau \eta'_0$ and $h_0 \sim_\tau h'_0$. It remains to prove the claim. We give the argument for the case that $\bar{x}$ is a single identifier, as the generalization is obvious but awkward to put into words. As $\ell \sim_\sigma \ell'$, it suffices to deal with $\bar{d}, \bar{d}'$. Assume $\bar{\kappa} = L$ to show $\bar{d} \sim_\sigma \bar{d}'$. Then $\bar{d} \sim_\sigma \bar{d}'$ follows by Lemma 7.4 on $\bar{e}$.

Case $\Delta; P \vdash S_1;\ S_2 : (com\ \kappa_1,\kappa_2)$: Let $(h_1,\eta_1) = [\![\Delta^\dagger \vdash S_1^\dagger]\!]\mu(h,\eta)Q$ and thus $(h_0,\eta_0) = [\![\Delta^\dagger \vdash S_2^\dagger]\!]\mu(h_1,\eta_1)Q$. By induction on S_1 we get $h_1 \sim_{\tau_1} h'_1$ and $\eta_1 \sim_{\tau_1} \eta'_1$ for some $\tau_1 \supseteq \sigma$. Now by induction on S_2 we get $h_0 \sim_\tau h'_0$ and $\eta_0 \sim_\tau \eta'_0$ for some $\tau \supseteq \tau_1$. Hence the result holds for $\tau \supseteq \sigma$.

Case $\Delta; P \vdash$ **if** e **then** S_1 **else** $S_2 : (com\ \kappa_1,\kappa_2)$: Let $b = [\![\Delta^\dagger \vdash e : \mathbf{bool}]\!](h,\eta)$. We proceed by cases on level κ of the guard e. Suppose $\kappa = L$. Then by Lemma 7.4 for e, we have $b \sim_{\mathbf{bool}} b'$, i.e. $b = b'$. If $b = true$, the result follows by induction on S_1 and if $b = false$, the result follows by induction on S_2.

Consider the other case, $\kappa = H$. By the typing rule, $\kappa \leqslant \kappa_1 \sqcap \kappa_2$, hence $\kappa_1 = H = \kappa_2$. By Lemma 7.2 we have $\eta \sim_\iota \eta_0$, $\eta' \sim_{\iota'} \eta'_0$, $h \sim_\iota h_0$ and $h' \sim_{\iota'} h'_0$, where ι (resp. ι') is the identity on $dom\,h$ (resp. $dom\,h'$). Using assumptions $\eta \sim_\sigma \eta'$ and $h \sim_\sigma h'$ we can choose $\tau = \sigma$ and get $\eta_0 \sim_\tau \eta'_0$ and $h_0 \sim_\tau h'_0$ using Lemma 7.1.

Case $\Delta;P \vdash (T,\kappa)\ x := e$ **in** $S : (com\ \kappa_1,\kappa_2)$: Let $d = [\![\Delta^\dagger \vdash e : T]\!](h,\eta)$ and thus $(h_0,\eta_0) = [\![(\Delta^\dagger, x : T) \vdash S^\dagger]\!]\mu(h,[\eta \mid x \mapsto d])Q$. First, we have $[\eta \mid x \mapsto d] \sim_\sigma [\eta' \mid x \mapsto d']$, because if $\kappa = L$ then by Lemma 7.4 on e, $d \sim_\sigma d'$. So we can use induction on S to get $\eta_0 \sim_\tau \eta_0'$ and $h_0 \sim_\tau h_0'$, for some $\tau \sqsupseteq \sigma$. Hence $(\eta_0 \lfloor x) \sim_\tau (\eta_0' \lfloor x)$.

Case $\Delta;P \vdash$ **enable** P' **in** $S : (com\ \kappa_1,\kappa_2)$: By hypothesis of the rule we have

$$\Delta;(P - Q') \vdash S : (com\ \kappa_1,\kappa_2)$$

where $Q' = P' \cap Auth(\Delta^\dagger \mathsf{self})$. By semantics, $(h_0,\eta_0) = [\![\Delta^\dagger \vdash S]\!]\mu(h,\eta)(Q \cup Q')$. So, to use induction on S it suffices to show that $(Q \cup Q') \# (P - Q')$. This follows by set theory from hypothesis $Q \# P$.

Case $\Delta;P \vdash$ **test** P' **then** S_1 **else** $S_2 : (com\ \kappa_1,\kappa_2)$: We argue by cases on whether $P' \subseteq Q$. If $P' \subseteq Q$ then $P' \# P$ by hypothesis $Q \# P$ and $P' \subseteq Auth(\Delta^\dagger \mathsf{self})$ by hypothesis $Q \subseteq Auth(\Delta^\dagger \mathsf{self})$. Thus the first of the rules in Figure 8 must have been used for **test** P' **then** S_1 **else** S_2. Since $P' \subseteq Q$, the test condition is true and the semantics is given by semantics of S_1. We have $\Delta;P \vdash S_1 : (com\ \kappa_1,\kappa_2)$ by hypothesis of the rule, so we can use induction on S_1 to obtain $\eta_0 \sim_\tau \eta_0'$ and $h_0 \sim_\tau h_0'$ for some $\tau \sqsupseteq \sigma$.

In the other subcase, $P' \not\subseteq Q$, the test condition is false so the semantics is given by semantics of S_2. Both of the rules for **test** P' **then** S_1 **else** S_2 have hypothesis $\Delta;P \vdash S_2 : (com\ \kappa_1,\kappa_2)$ so we can use induction on S_2 to get $\eta' \sim_\tau \eta_0'$ and $h' \sim_\tau h_0'$ for some $\tau \sqsupseteq \sigma$.

Case $\Delta;P \vdash S : (com\ \kappa_3,\kappa_4)$ by the subsumption rule. Here the result follows directly by induction on the smaller derivation tree for $\Delta;P \vdash S : (com\ \kappa_1,\kappa_2)$, as the induction hypothesis does not involve the command levels. □

Theorem 8.4 (*safety implies noninterference*)
If annotated class table CT is safe then its meaning $[\![CT^\dagger]\!]$ is noninterfering.

Proof
Because $[\![CT^\dagger]\!]$ is defined as the least upper bound of an approximation chain μ, we first show *nonint* μ_i for all i, by induction on i. Then *nonint* $[\![CT^\dagger]\!]$ follows by Lemma 8.2.

We have *nonint* μ_0 because $\mu_0 Cm(h,\eta)Q$ is $\perp$.

Suppose *nonint* μ_i, to show *nonint* μ_{i+1}. We must show, for each C,m and each $\kappa_0,\bar{\kappa} \multimap \langle P ; \kappa_3 \rangle \rightarrowtail \kappa_4 \in smtypes(m,C)$, that $\mu_{i+1}Cm$ satisfies $\kappa_0,\bar{\kappa} \multimap \langle P ; \kappa_3 \rangle \rightarrowtail \kappa_4$. There are two cases, depending on whether m is declared or inherited.

Suppose m has declaration

$$M \;=\; T\ m(\bar{T}\ \bar{x})\{S\}$$

in C. Let $\Delta = \bar{x} : (\bar{T},\bar{\kappa}), \mathsf{self} : (C,\kappa_0), \mathsf{result} : (T,\kappa_4)$ and let S' be a local variant of S satisfying the conditions of the method declaration rule with respect to $\kappa_0,\bar{\kappa} \multimap \langle P ; \kappa_3 \rangle \rightarrowtail \kappa_4$. ($S'$ exists because CT is safe.) By Lemma 7.3 we have *wconf* μ_i. Suppose that $h \sim_\sigma h'$, $\eta \sim_\sigma \eta'$, $Q \# P$ and $\mu_{i+1}Cm(h,\eta)Q \neq \perp \neq \mu_{i+1}Cm(h',\eta')Q$. Let $Q_1 = Q \cap Auth\ C$ and $(h_0,\eta_0) = [\![(\bar{x} : \bar{T}, \mathsf{self} : C, \mathsf{result} : T) \vdash S^\dagger]\!]\mu_i(h,\eta)Q_1$. Now $Q_1 \# P$, so by Lemma 8.3, noting that $S^\dagger = S'^\dagger$, there is $\tau \sqsupseteq \sigma$ with $h_0 \sim_\tau h_0'$ and

$\eta_0 \sim_\tau \eta'_0$. It remains to show that if the result level κ_4 for m is L we have $d \sim_\tau d'$, where d (respectively d') is $\eta_0\,\mathsf{result}$ (resp. $\eta'_0\,\mathsf{result}$). This follows from $\eta_0 \sim_\tau \eta'_0$ by definition of $\sim$ and $\Delta\,\mathsf{result} = (T, \kappa_4) = (T, L)$. This concludes the proof of satisfaction by $\mu_{i+1}Cm$ for m declared in C.

Suppose m is inherited in C from superclass B. Towards proving that $\mu_{j+1}Cm$ satisfies $\kappa_0, \bar{\kappa} \!-\!\!\langle P\,; \kappa_3\rangle\!\!\rightarrow \kappa_4$, let

$$\Delta_C = \bar{x} : (\bar{T}, \bar{\kappa}), \mathsf{self} : (C, \kappa_0) \quad \text{and} \quad \Delta_B = \bar{x} : (\bar{T}, \bar{\kappa}), \mathsf{self} : (B, \kappa_0).$$

Suppose also that $h \sim_\sigma h'$ and $\eta \sim_\sigma \eta'$. Note that the $\sim_\sigma$ relation for Δ_B is equivalent to $\sim_\sigma$ for Δ_C. Suppose $Q \# P$. Then, because $\mathit{smtypes}\, C = \mathit{smtypes}\, B$ (by Definition 5 annotated class table), satisfaction by $\mu_{j+1}Cm$ follows from the same for $\mu_{j+1}Bm$.

Note that the last step goes through because satisfaction (Definition 11) quantifies over all Q disjoint from P, without regard to the permissions of C and B.

Note also that we are using a secondary induction on inheritance chains, so we may use satisfaction by $\mu_{j+1}Bm$ to prove it for $\mu_{j+1}Cm$. □

9 Discussion

We have given a static analysis for secure information flow that accounts for correct use of runtime access control to ensure that confidential information is returned only if the caller has been given access. This improves on previous information flow analyses where a system call is given a fixed security level (in some cases polymorphic) and only static access control is considered. Our analysis is justified by a noninterference result. This is given for a language significantly more complex than previous object-oriented languages for which information flow has been proved.

Integration with access control shows that even strong noninterference conditions which disallow declassification may be useful and admit practical static checking, once access control is taken into account. We only take a step in this direction, demonstrating the idea in the context of a non-trivial language but using a language-centric access control mechanism devised primarily for protecting trusted programs from untrusted mobile code. The key idea is to make flow policy dependent on access permissions; we expect that our treatment can be extended easily to other permission-based access mechanisms. Like Pottier & Conchon (2000) and unlike Heintze *et al.* (1998), we refrain from conflating access permissions with confidentiality levels.

Our analysis is modular in the sense that, for a given type, a method declaration is checked using only security types of methods that it calls and fields it accesses. As a type depends on a set of permissions, it is possible to specify a number of types for a method that is exponential in the number of existing permissions, if there are enough security levels. In a naive implementation of the rule, a method declaration is checked once for each type. We conjecture that in practical applications the specification for given method will use only a few types (fewer if level polymorphism is used). Moreover, if the types are minimal, as suggested in section 3, then for a method *call* at most one type will be applicable, so checking of a method body remains linear in the size of the code. The rule for checking a method declaration requires finding levels to annotate local variables. This can be done by an intra-procedural inference algorithm such as that used in Jif (Myers, 1999) which is fast in practice.

For practical deployment of programs using stack inspection for access control, interface specifications need to express expected or recommended policies. Our method typings suggest a way to do so, not just for access but for information flow. If an information flow policy is specified for public and protected fields and methods, a checker based on our analysis could be used by developers to find bugs and trojan horses. Vendor-supplied defaults for access policy could help ensure that deployments are consistent with what is checked by developers. But many engineering challenges remain. Techniques for certifying compilation could be used for checks of untrusted code at the deployment site (Necula, 1997; Morrisett *et al.*, 1999). This is a subject of current research (Barthe *et al.*, 2004).

9.1 Related work

Our previous work. This work grew out of a study of data abstraction. We found that a straightforward compositional semantics is adequate for a sequential Java-like language, even with recursive types and dynamically bound method calls (which are typically viewed as being akin to higher order procedures). The language-oriented formulation of noninterference using simulations is related to relational parametricity, as is more explicit in Heintze & Riecke (1998), Abadi *et al.* (1999) and Sabelfeld & Sands (2001). Due to pointer aliasing, conventional object-oriented languages are not relationally parametric *per se*. But suitable confinement of pointers suffices to yield a strong representation-independence result for user-defined abstractions (Banerjee & Naumann, 2002a; Banerjee & Naumann, 2002b). The term "confinement" originated in the literature on operating system security, but its use is natural in object-oriented programming where pointer confinement has been proposed for encapsulation at the level of modules, classes, or instances (Hogg, 1991; Clarke *et al.*, 2001; Leino & Nelson, 2002; Boyland *et al.*, 2001; Vitek & Bokowski, 2001).

In earlier stages of the information flow work (Banerjee & Naumann (2002c; 2003)), we imposed a strong pointer-confinement condition called L-confinement: each object is labeled with a level, given by its class, and fields/variables of level L never point to H objects. To achieve this strong invariant, it is assumed that the allocator is parametric in the sense that H allocations do not influence the behavior of L allocations. The L-confinement discipline is not motivated by practical examples, and restrictions are needed on inheritance between classes at different levels to prevent leaks of `self`. In the present paper we use the standard technique of maintaining a bijection on locations, here just those locations visible to L, thereby proving noninterference without requiring the labeling of classes or parametricity of the allocator.

The conference papers (Banerjee & Naumann (2002c; 2003)) use purely syntax-directed rules, close to a checking algorithm. Here we simplify the rules for program constructs (and the proofs) by using a separate subsumption rule. We also study the structure of security types for methods and clarify that the most succinct method specifications are those giving a set of minimal types.

Information flow. Beyond the early work mentioned in section 1, several researchers have given similar analyses for possibilistic and probabilistic noninterference for multi-threaded programs (Smith & Volpano, 1998; Volpano & Smith, 1999; Sabelfeld & Sands, 2000; Smith, 2001). Mantel and Sabelfeld (Mantel & Sabelfeld, 2001) study connections between program-centric formulations and the formulation of noninterference in terms of abstract event systems (Goguen & Meseguer, 1982). Barthe and Serpette prove noninterference for a purely functional instance-based object calculus (Barthe & Serpette, 1999). For sequential programs, Abadi *et al.* (Abadi *et al.*, 1999), Sabelfeld & Sands (Sabelfeld & Sands, 2001) and Heintze & Riecke (Heintze & Riecke, 1998) consider higher order procedures. They also make explicit the connections between the relational formulation of noninterference and other dependency analyses such as slicing and binding time analysis (Abadi *et al.*, 1999; Barthe & Serpette, 1999), building noninterference properties into the semantics in the manner of Reynolds' relationally parametric models (Reynolds, 1984). However, it is difficult to extend such models in a tractable way to encompass language features such as recursive types and shared mutable objects which are extensively used in languages such as Java (Arnold & Gosling, 1998). What makes it possible for us to give straightforward proofs of strong results is that we can use a simple, operationally transparent denotational semantics, because Java-like languages are first-order in the sense that method dictionaries are not first-class values.

Myers (Myers, 1999) gave a security typing system for a fragment of Java even richer than ours, but left open the problem of justifying the rules with a noninterference result. This is hardly surprising, as the rules are quite complicated. Some of the complications are inherent in the complexity of the language; others are introduced with the aim of accomodating dynamic access control and sophisticated security policies including declassification (Ferrari *et al.*, 1997; Myers & Liskov, 1998; Myers, 1999). One strength of this project is that experience is being gained in ongoing development of the Jif prototype.[4]

Another substantial prototype for information flow has recently appeared. Flow-Caml[5] is based on the work of (Pottier & Conchon, 2000; Pottier & Simonet, 2003). The language is a substantial fragment of Objective Caml, though omitting object-oriented features. This is perhaps the most advanced work on inference for flow types and level polymorphism and it is supported by noninterference results. They have an attractive approach to proving noninterference by translation to ordinary unannotated types and by then appealing to a type soundness theorem.

Strecker (2003) recently gave a machine-checked proof of noninterference for a language similar to that of Banerjee & Naumann (2002c), using similar rules as well.

[4] On the web at `http://www.cs.cornell.edu/jif/`.
[5] On the web at `http://cristal.inria.fr/~simonet/soft/flowcaml/`.

Like the latter, it uses equality for low-indistinguishability, assumes parametricity of the allocator, and imposes a form of L-confinement.

Sabelfeld & Myers (2003) give a comprehensive review of the literature on information flow.

Access control. Previous research on static analysis of information flow in programs has connections with data access. But the connections are made using constraints that allow entirely static checking of access and the access mechanisms considered are akin to ordinary scoping mechanisms like private fields and opaque types. Similar connections have been made in more abstract models (Rushby, 1992; Hennessy & Riely, 2002). We are not aware of previous work connecting runtime access control with noninterference.

Stoughton (1981) compares access control and information flow in a simple imperative language with semaphores. No formal results are proven, nor is there a static analysis for information flow. Rushby (1992) proves (and mechanically checks) results on noninterference for an access control mechanism that amounts to assigning levels to variables. The SLam calculus is a framework where access control and information flow coexist, but the noninterference result is restricted to information flow (Heintze & Riecke, 1998). Access control in SLam consists of labels which are checked for compatibility by typing rules; there is no runtime significance. Similar remarks apply to Pottier and Conchon's work (Pottier & Conchon, 2000) and that of Hennessy & Riely (2002). The Jif system includes dynamic labels and other features that can encode some form of runtime access control but an analysis of what is achieved has not appeared.

Concerning the "stack inspection" mechanism, Skalka *et al.* (2000; 2001) give a static analysis for access checks that never fail, which could serve as basis for program optimizations. Such optimizations are thoroughly studied by Fournet & Gordon (2002), who also explore an access-based security property. Wallach *et al.* (2000) use a logic of authentication to account for access policies achieved by the mechanism. Abadi & Fournet (2003) criticize the lack of clear security goals achieved by the mechanism and point out that it only protects from untrusted code in the current calling chain, whereas common object-oriented design patterns involve calls from trusted to untrusted code. They propose a history-based mechanism but do not give formal results or a connection with information flow.

9.2 Future work

Naumann (2004) has encoded the language and semantics herein by a deep embedding in the PVS theorem prover (Owre *et al.*, 1992), with the omission of access control which is being added in ongoing work. The static analysis has been defined and the proof of noninterference machine checked. This development closely follows that in the present paper, but allowing an arbitrary lattice of security levels.

Conventional languages like Java have quite a few features beyond the language treated here. To extend our language to the remaining features of JavaCard (Chen, 2000), the semantics can be extended using standard techniques. To treat expressions with side effects, both the environment and the heap would be threaded through the semantics of expressions. We have avoided this in the current paper because it is unilluminating. Exceptional control flow would add further semantic complications of a similar kind. Static fields and static methods should also be straightforward. Constructors, in full generality, pose a challenge even for ordinary type soundness because partially initialized objects can be leaked. The other missing features have to do with scope and visibility: protected fields, private and protected classes, interfaces, and packages.

Features of Java beyond those of JavaCard pose a bigger challenge: threads, class loading, reflection, and serialization. Specifying noninterference for such constructs would probably go hand-in-hand with specification of pointer confinement and data abstraction properties. This raises a question. Program constructs such as local variables, private fields, opaque types, as well as disciplines for alias control, are intended to provide information hiding. Yet work on information flow has used typing systems that are orthogonal to these constructs. In early versions of this paper we used private fields but public fields can as well be treated by the same flow typing rules. For practical use of flow checking, it could be helpful to exploit existing features. How can this be done? Ideally, it should be based on the established theories of parametricity.

An important implementation question is which security annotations can be left implicit, to be inferred by a type reconstruction algorithm. In this paper we have not addressed type reconstruction, but we expect that techniques from Pottier *et al.* can be adapted (Pottier & Conchon, 2000; Pottier & Simonet, 2002). For the language of Banerjee & Naumann (2002c) without access control, our student Sun Qi has developed and implemented an inference algorithm and work is currently underway to incorporate level polymorphism, access control, and modular inference for libraries (Sun *et al.*, 2004).

Acknowledgements

We are grateful for helpful feedback from anonymous referees and from audiences at Microsoft Research, the Cornell IAI, SRI International, and the Dagstuhl Seminar on Language Based Security. We had helpful discussions with Torben Amtoft, Andrei Sabelfeld, Geoffrey Smith, Andrew Gordon, François Pottier, Scott Smith, Steve Zdancewic, and Andrew Myers – the last two also helped us understand the Jif system. Thanks to Manuel Fähndrich for insisting that negative information could be used in type annotations and to Sun Qi for refining the rules for test and for experimenting with Jif. Patrick Lincoln's group at SRI International provided a stimulating environment in which the second author worked on machine checking the proofs.

References

Abadi, M. and Fournet, C. (2003) Access control based on execution history. *Proc. 10th Annual Network and Distributed System Security Symposium*, pp. 107–121.

Abadi, M., Banerjee, A., Heintze, N. and Riecke, J. G. (1999) A core calculus of dependency. *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 147–160.

Arnold, K. and Gosling, J. (1998) *The Java Programming Language, 2nd ed.* Addison-Wesley.

Banerjee, A. and Naumann, D. A. (2002a) *Ownership confinement ensures representation independence for object-oriented programs.* Revised and extended from *ACM Symposium on Principles of Programming Languages (POPL).* ACM Press. Submitted.

Banerjee, A. and Naumann, D. A. (2002b) Representation independence, confinement and access control. *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 166–177. ACM Press.

Banerjee, A. and Naumann, D. A. (2002c) Secure information flow and pointer confinement in a Java-like language. *IEEE Computer Security Foundations Workshop (CSFW)*, pp. 253–270. IEEE Press.

Banerjee, A. and Naumann, D. A. (2003) Using access control for secure information flow in a Java-like language. *IEEE Computer Security Foundations Workshop (CSFW)*, pp. 155–169. IEEE Press.

Barthe, G. and Serpette, B. (1999) Partial evaluation and non-interference for object calculi. In: Middeldorp, A. and Sato, T. (eds.), *Proceedings of FLOPS'99: Lecture Notes in Computer Science 1722*, pp. 53–67. Springer-Verlag.

Barthe, G., Basu, A. and Rezk, T. (2004) Security types preserving compilation. *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI): Lecture Notes in Computer Science 2937*. Springer-Verlag.

Bell, D. E. and LaPadula, L. J. (1973) *Secure computer systems: Mathematical foundations.* Technical report MTR-2547, MITRE Corp.

Boyland, J., Noble, J. and Retert, W. (2001) Capabilities for sharing: A generalisation of uniqueness and read-only. *European Conference on Object Oriented Programming (ECOOP)*, pp. 2–27.

Chen, Z. (2000) *Java Card Technology for Smart Cards*. Addison-Wesley.

Clarke, D. G., Noble, J. and Potter, J. M. (2001) Simple ownership types for object containment. *European Conference on Object Oriented Programming (ECOOP): Lecture Notes in Computer Science*. Springer-Verlag.

Denning, D. (1976) A lattice model of secure information flow. *Comm. ACM*, **19**(5), 236–242.

Denning, D. and Denning, P. (1977) Certification of programs for secure information flow. *Comm. ACM*, **20**(7), 504–513.

Erlingsson, U. and Schneider, F. B. (1999) SASI enforcement of security policies: A retrospective. *Proc. New Security Paradigm Workshop*, pp. 87–95.

Ferrari, E., Samarati, P., Bertino, E. and Jajodia, S. (1997) Providing flexibility in information flow control for object-oriented systems. *Proc. IEEE Symp. on Security and Privacy*, pp. 130–140.

Fournet, C. and Gordon, A. D. (2002) Stack inspection: Theory and variants. *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 307–318. ACM Press.

Goguen, J. and Meseguer, J. (1982) Security policies and security models. *Proc. IEEE Symp. on Security and Privacy*, pp. 11–20.

Gong, L. (1999) *Inside Java 2 Platform Security*. Addison-Wesley.

Gough, J. (2001) *Compiling for the .NET Common Language Runtime*. Prentice Hall.

Heintze, N. and Riecke, J. G. (1998) The SLam calculus: programming with secrecy and integrity. *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 365–377.

Hennessy, M. and Riely, J. (2002) Information flow vs. resource access in the asynchronous pi-calculus. *ACM Trans. Program. Langu. & Syst.* **24**(5), 566–591.

Hogg, J. (1991) Islands: Aliasing protection in object-oriented languages. *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA).* ACM Press. (*SIGPLAN Notices*, **26**, 11.)

LaMacchia, B. A., Lange, S., Lyons, M., Martin, R. and Price, K. T. (2003) *.NET framework security.* Addison-Wesley.

Leino, K. R. M. and Nelson, G. (2002) Data abstraction and information hiding. *ACM Trans. Program. Lang. & Syst.* **24**(5).

Mantel, H. and Sabelfeld, A. (2001) A generic approach to the security of multi-threaded programs. *Proc. 14th IEEE Computer Security Foundations Workshop*, pp. 126–142. Cape Breton, Nova Scotia, Canada. IEEE Press.

McLean, J. (1994) A general theory of composition for trace sets closed under selective interleaving functions. *Proc. IEEE Symp. on Security and Privacy*, pp. 79–93.

Mitchell, J. C. (1996) *Foundations for Programming Languages.* MIT Press.

Morrisett, G., Crary, K., Glew, N. and Walker, D. (1999) From system F to typed assembly language. *ACM Trans. Program. Lang. & Syst.* **21**(3), 528–569.

Myers, A. and Liskov, B. (1998) Complete, safe information flow with decentralized labels. *Proc. IEEE Symp. on Security and Privacy*, pp. 186–197.

Myers, A., Sabelfeld, A. and Zdancewic, S. (2004) Enforcing robust declassification. *IEEE Computer Security Foundations Workshop (CSFW).* pp. 172–186.

Myers, A. C. (1999) JFlow: Practical mostly-static information flow control. *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 228–241.

Naumann, D. A. (2004). *Machine-checked correctness of a secure information flow analyzer (preliminary report).* (Unpublished typescript, available at http://www.cs.stevens-tech.edu/~naumann/accinfPVS.ps.)

Necula, G. C. (1997) Proof-carrying code. *Proceedings, POPL.* ACM Press.

Ørbæk, P. and Palsberg, J. (1997) Trust in the λ-calculus. *J. Funct. Program.* **7**(6), 557–591.

Owre, S., Rushby, J. M. and Shankar, N. (1992) PVS: A prototype verification system. In: Kapur, Deepak (ed.), *11th International Conference on Automated Deduction (CADE): Lecture Notes in Artificial Intelligence 607*, pp. 748–752. Springer-Verlag.

Pottier, F. and Conchon, S. (2000) Information flow inference for free. *Proc. Fifth ACM International Conference on Functional Programming*, pp. 46–57.

Pottier, F. and Simonet, V. (2002) Information flow inference for ML. *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 319–330.

Pottier, F. and Simonet, V. (2003) Information flow inference for ML. *ACM Trans. Program. Lang. & Syst.* **25**(1), 117–158.

Pottier, F., Skalka, C. and Smith, S. (2001) A systematic approach to static access control. In: Sands, D. (ed.), *European Symposium on Programming (ESOP): Lecture Notes in Computer Science 2028*, pp. 30–45. Springer-Verlag.

Reynolds, J. C. (1984) Types, abstraction, and parametric polymorphism. In: Mason, R. E. A. (ed.), *Information Processing '83*, pp. 513–523. North-Holland.

Rushby, J. (1992) *Noninterference, transitivity, and channel-control security policies.* Technical report SRI.

Sabelfeld, A. and Myers, A. C. (2003) Language-based information-flow security. *IEEE J. Selected Areas in Comm.* **21**(1), 5–19.

Sabelfeld, A. and Sands, D. (2000) Probabilistic noninterference for multi-threaded prgorams. *IEEE Computer Security Foundations Workshop (CSFW)*, pp. 200–215.

Sabelfeld, A. and Sands, D. (2001) A Per model of secure information flow in sequential programs. *Higher-order & Symbolic Comput.* **14**(1), 59–91.

Skalka, C. and Smith, S. (2000) Static enforcement of security with types. *Proc. Fifth ACM International Conference on Functional Programming*, pp. 34–45.

Smith, G. (2001) A new type system for secure information flow. *Proc. 14th IEEE Computer Security Foundations Workshop*, pp. 115–125.

Smith, G. and Volpano, D. (1998) Secure information flow in a multi-threaded imperative language. *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 355–364.

Stoughton, A. (1981) Access flows: A protection model which integrates access control and information flow. *Proc. IEEE Symp. on Security and Privacy*, pp. 9–18.

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

Sun, Q., Banerjee, A. and Naumann, D. A. (2004) *Modular and constraint-based information flow inference for an object-oriented language*, Static Analysis Symposium (SAS), Springer Verlag. Lecture Notes in Computer Science, **3148**, 84–99.

Vitek, J. and Bokowski, B. (2001) Confined types in Java. *Software–Practice & Experience*, **31**(6), 507–532.

Volpano, D. (1999) Safety versus secrecy. *Static Analysis Symposium (SAS): Lecture Notes in Computer Science 1694*, pp. 303–311. Springer-Verlag.

Volpano, D. and Smith, G. (1997) A type-based approach to program security. *Proc. TAPSOFT'97: Lecture Notes in Computer Science 1214*, pp. 607–621. Springer-Verlag.

Volpano, D. and Smith, G. (1999) Confinement properties for multi-threaded programs. *Electr. Notes in Theor. Comput. Sci.* **20**.

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

Wallach, D., Appel, A. and Felten, E. (2000) SAFKASI: a security mechanism for language-based systems. *ACM Trans. Software Eng. & Methodology*, **9**(4).