Using Access Control for Secure Information Flow in a Java-like Language

Anindya Banerjee*
Computing and Information Sciences
Kansas State University
Manhattan KS 66506 USA
ab@cis.ksu.edu

David A. Naumann†
Computer Science
Stevens Institute of Technology
Hoboken NJ 07030 USA
naumann@cs.stevens-tech.edu

Abstract

Access control mechanisms are widely used with the intent of enforcing confidentiality and other policies, but few formal connections have been made between information flow and access control. Java and C♯ are object-oriented languages that provide fine-grained access control. An access control list specifies local policy by authorizing permissions for principals (code sources) associated with class declarations; a mechanism called stack inspection checks permissions at run time. An example is given to show how this mechanism can be used to achieve confidentiality goals in situations where a single system call serves callers of differing confidentiality levels and dynamic access control prevents release of high information to low callers. A novel static analysis is given which applies to such examples. The analysis is shown to ensure a noninterference property formalizing confidentiality.

1. Introduction

Many security policies involve confidentiality requirements. Confidentiality properties of programs, formalized in terms of noninterference [9], can be checked using static information flow analysis [25, 24]. But this has seen little use in practice. As discussed in the recent survey by Sabelfeld and Myers [21], extant static analyses are somewhat restrictive and a satisfactory treatment of declassification remains elusive. It is access control that is widely used.

One approach to checking confidentiality of code using access control is to somehow designate those atomic steps by which information can flow, and prove that the steps are only taken when the appropriate access control events have occurred. This approach has obvious merit but leaves open the question of making a rigorous connection with strong information flow policies like "the user's private key is not leaked to the seller". What is the connection between authorization of a data-access event and the subsequent dissemination of information contained therein?

In fact, previous work on static analysis of information flow in programs (e.g., [7, 25, 24, 12, 17]) is based on control of data access, as we discuss further in Section 8. 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, name-equivalence of types, and pointer confinement [4]. Similar connections have been made in more abstract models [20, 13].

In this paper we are concerned with programs that use access control dynamically in the sense that permissions determine runtime behavior. Permissions are not simply identified with data items and principals are not identified with confidentiality levels. We show how dynamic access control can allow flexible program interfaces where a data channel can be used for more than one purpose, while ensuring confidentiality.

We consider the access control mechanism of Java [10], 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.

We consider programs that use access control to enforce an information-flow policy expressed by labelling of input and output channels with levels in a lattice [5, 6]. We give a static analysis in the style of security typing [25, 24] but also tracking the permissions manipulated by the access control constructs. We prove a form of noninterference for typable programs. As in previous work, our analysis validates code with respect to a given policy. Here, policy designates both trust (authorization of different permissions for different programs) and confidentiality levels.

Our rules account for calls into the trusted computing base that might return high security information, but which use access checks to ensure that only low security informa-

*Supported by NSF grants CCR-0209205 and NSF Career award CCR-0296182.

†Supported by NSF grant CCR-0208984.

tion is returned unless the caller has been given access.

We use a simple form of noninterference, suited to observation only of the initial and final states of terminating computations of sequential programs. Mantel and Sabelfeld [14] study connections between program-centric formulations and the formulation of noninterference in terms of abstract event systems [9].

In Section 2 we give a streamlined description of the relevant features of the Java access control mechanism, which is supported by the Java Virtual Machine (a very similar mechanism is supported by the Common Language Runtime [11] which is the typical target for compilation of C♯). We also review the use of labelled types to specify confidentiality in imperative [25] and object-oriented [4, 16] 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. We give two types to a method providing a system service. One type says that it returns only low information, if called by a program for which designated permissions are not authorized. The other type applies to programs that are granted the permissions and says that for them the result is high.

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 all the features treated in [4] (pointers and mutable state, private fields and class-based visibility, dynamic binding and inheritance, recursive classes, casts and type tests, and recursive methods). The reason for this choice is to show that the idea scales to a realistic language and to lay a broader foundation for future work (e.g., permissions and protection domains as first-class objects as in Java).

The main contributions of the paper are the ideas in Section 3 and their technical development in conjunction with inheritance and dynamic binding. For pointers and confinement the treatment in [4] carries over without difficulty.

Section 5 formalizes a static analysis using permission-dependent method types in syntax-directed rules which admit the examples of Section 3. Section 6 proves basic results about the analysis, which are used in Section 7 to prove the main result: the static analysis ensures noninterference.

2. Access control and information flow

Access control by stack inspection. In the Java access control mechanism [10], each class C has a set $Auth\,C$ of permissions 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, 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 *checkPermission* 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.

Following previous work [8], we refrain from modeling exceptions and instead consider a construct, **test** p **then** S_1 **else** S_2, which performs the check, executing S_1 if the check succeeds and S_2 if it fails. A method body that simply performs a check can be written **test** p **then skip else abort**. To model the case where an exception is thrown and caught, the **else** part can return some value that indicates to the caller that a check has failed.

In fact, what is checked is slightly more subtle than authorization for all code on the stack. In keeping with the principle of least privilege, permissions must be explicitly enabled by an operation *doPrivileged*. This is implemented using a callback object with the effect of lexical scoping. We model it by a construct, **enable** p **in** S, the effect of which is to mark p as enabled in the current frame. The relevant information in a frame is a pair $\langle C, Q\rangle$ where C is the class providing the code for the frame and $Q \in \mathcal{P}(Permissions)$ is a set of permissions. The effect of **enable** p **in** S is to add p to Q for the duration of S. The precise meaning of **test** p is as follows: check whether p is marked in some frame for which p is authorized, and in addition p is authorized for all subsequent frames including the current one. This allows a method to, in effect, temporarily grant a permission to methods in the call chain to it. It cannot, however, grant permissions to code that it invokes. System code can invoke plug-ins without risk of giving them unintended permissions and it can also perform sensitive operations on behalf of untrusted callers. The class C in the pair $\langle C, Q\rangle$ is that which declares the method. Due to inheritance, it may be a proper superclass (and have different permissions) than the class of the target object.

As described above, stack inspection is lazy in that authorization checks are only performed when needed. For theoretical analysis, it is convenient to use the equivalent eager semantics [10, 19, 8] which works as follows. The effect of **enable** p **in** S is to add p to the current frame only

if p is authorized for the current class. When a method is invoked from a frame $\langle C, Q\rangle$, the new frame is initialized to be $\langle D, Q'\rangle$ where D is the class of the invoked code and $Q' = Q \cap Auth\, D$. Finally, **test** p just checks whether p is in the current frame. This description does not involve the operational notion of traversing the stack. In the sequel we use a compositional semantics where Q is simply an implicit parameter.

Checking information flow using security types. The idea developed by Volpano *et al.* [25] 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. As usual, we consider the representative two-element lattice $L \leq H$ of levels. 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, there is the possibility of observing differing behavior of the allocator if objects allocated conditionally are accessible. Such issues are treated in [16, 4]. In [4], commands are given types $(com\ \kappa_1, \kappa_2)$ where κ_1 is a lower bound on the level of assigned variables and κ_2 is a lower bound on the heap effect (field assignements and newly allocated objects). Annotated arrow types are used for modular checking in the case of methods (or procedures or functions [1]): the type $(T, \kappa_1) \xrightarrow{\kappa_2} (U, \kappa_3)$ designates input level κ_1, heap effect κ_2, and result level κ_3. A method body is checked with respect to its type, which is used as an assumption for checking method calls.

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

Consider a system composed of components, some of which are from untrusted sources. Class *Kern* is a trusted system class and *Vend1* is from a less trusted source. To a first approximation, a confidentiality goal would be that information confidential to *Kern* is not leaked to *Vend1* . This policy could be expressed by labelling certain inputs to *Kern* as level H and the others as L. In practice, the levels in the security lattice might correspond to code sources and thus be correlated with permissions. But, like [17] and unlike [12], 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 untrusted ones manipulating L. Access policy $Auth$ is as follows:

class	permissions
Vend1	$other$
Vend2	$stat, other$
Kern	$stat, sys$

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

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

Class *Vend1* 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 Vend1 extends Object { // permissions: other
  Kern k; // L
  string v; // L
  string status(){ // type L → 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.

For an information flow analysis to allow *Vend1.status*, it is necessary to take into account the **test** in *getStatus* and

also the access policy for *Vend1*. Otherwise, a sound analysis of *getStatus* would say that it can return H which violates the flow policy for *Vend1.status*.

Vend1 could try to gain access to *Hinfo* as follows:

string *status2()*{ // *type* $L \to L$
enable *stat* **in** *result* := *self.v ++ k.getStatus()* }

But because $stat$ is not authorized for *Vend1*, the **enable** has no effect and the policy is not violated.

No code in classes *Vend1* or *Vend2* can successfully invoke *getHinfo* because in $Auth$ neither is granted permission sys, without which method *getHinfo* aborts. An attempted **enable** sys does not help.

Our access policy does, however, grant permission $stat$ to *Vend2*. The flow policy also indicates a degree of trust in that method *statusH* below returns H.

class *Vend2* { // *permissions:* $stat, other$
Kern k;
String statusH(){ // *type* $L \to H$
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 *Vend2*. 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 and Smith, among others, consider procedure typings that are polymorphic in levels [24] (to handle cases where level-α inputs yields level-α outputs). Such systems cannot handle our examples, because the result level for *getStatus* depends not on explicit input parameters but on enabled permissions.

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. The meaning of a type $L \xrightarrow{P} \kappa$ 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 $L \xrightarrow{\varnothing} H$ and $L \xrightarrow{\{stat\}} L$. The call from *Vend1.status* can be typechecked with respect to $L \xrightarrow{\{stat\}} L$, because $stat$ is not in $Auth(Vend1)$. The typechecking rule does not allow the call from *Vend2.statusH* to be checked using $L \xrightarrow{\{stat\}} L$, because $stat$ is authorized for *Vend2*. It can be checked using $L \xrightarrow{\varnothing} H$.

Consider this additional method for *Vend2*:

string *statusH2()*{ *result* := *k.getStatus()* }

It can be given type $L \xrightarrow{\varnothing} H$ just like *statusH*. Although $stat$ is not enabled by *statusH2*, it could be enabled by a caller thereof. So it is not sound to check the body of *statusH2* using *getStatus*: $L \xrightarrow{\{stat\}} L$ unless we disallow such callers, which yields the type $L \xrightarrow{\{stat\}} L$ for *statusH2*.

We have discussed method types of particular interest, but others are also sound, e.g., *getStatus*: $L \xrightarrow{\{stat,sys\}} L$ and *getHinfo*: $L \xrightarrow{\{sys\}} L$. There is an evident notion of subtyping which we do not consider in this paper. For practical application, one would specify the security of a method using a set of types that are minimal with respect to subtyping.

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.[1]

4. Language

Our results are for a sequential class-based language similar to the one in the predecessor paper [4] with the addition of access control. This section presents the language without security annotations; it is this language for which the semantics is defined. As compared with [4], some improvements have been made to the language and semantics following [2] to which we refer for more extensive explanations.

We assume given a finite set of $Permissions$ as well as function $Auth : ClassNames \to \mathcal{P}(Permissions)$. Finiteness is not essential, but it saves us from imposing explicit finiteness restrictions at various points in the syntax and semantics.

[1] In an earlier version of this paper, we annotated the typing arrow with an upper bound on the caller's permissions, so for example *getStatus* would have a type $L \xrightarrow{P} L$ 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.

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

Table 1. Grammar.

$$\Gamma \vdash x : \Gamma x \quad \Gamma \vdash \textbf{null} : B \quad \Gamma \vdash \textbf{unit} : \textbf{unit} \quad \Gamma \vdash \textbf{true} : \textbf{bool} \quad \Gamma \vdash \textbf{false} : \textbf{bool}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 == e_2 : \textbf{bool}} \qquad \frac{\Gamma \vdash e : (\Gamma\ \mathit{self}) \quad (f : T) \in \mathit{dfields}(\Gamma\ \mathit{self})}{\Gamma \vdash e.f : T}$$

$$\frac{\Gamma \vdash e : D \quad B \leq D}{\Gamma \vdash (B)\ e : B} \qquad \frac{\Gamma \vdash e : D \quad B \leq D}{\Gamma \vdash e\ \textbf{is}\ B : \textbf{bool}}$$

$$\frac{\Gamma \vdash e : T \quad T \leq \Gamma x \quad x \neq \mathit{self}}{\Gamma \vdash x := e} \qquad \frac{\Gamma \vdash e_1 : (\Gamma\ \mathit{self}) \quad (f : T) \in \mathit{dfields}(\Gamma\ \mathit{self}) \quad \Gamma \vdash e_2 : U \quad U \leq T}{\Gamma \vdash e_1.f := e_2}$$

$$\frac{\Gamma \vdash e : D \quad \mathit{mtype}(m, D) = \overline{T} \to T \quad \Gamma \vdash \overline{e} : \overline{U} \quad \overline{U} \leq \overline{T} \quad x \neq \mathit{self} \quad T \leq \Gamma x}{\Gamma \vdash x := e.m(\overline{e})} \qquad \frac{B \leq \Gamma x \quad x \neq \mathit{self} \quad B \neq \textbf{Object}}{\Gamma \vdash x := \textbf{new}\ B}$$

$$\frac{\Gamma \vdash e : \textbf{bool} \quad \Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2} \qquad \frac{\Gamma \vdash e : U \quad U \leq T \quad x \neq \mathit{self} \quad (\Gamma, x : T) \vdash S}{\Gamma \vdash T\ x := e\ \textbf{in}\ S} \qquad \frac{\Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash S_1\ ;\ S_2}$$

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

Table 2. Typing rules for expressions and commands.

4.1. Syntax

The grammar is given by Table 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 $\overline{T}$ with bars on top indicate finite lists, e.g., $\overline{T\ f}$ stands for a list $\overline{f}$ of field names with corresponding types $\overline{T}$. 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.

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.

Methods and classes are considered public. The rules for field access and update enforce private visibility: fields declared in a class are accessible only to methods of that class.

Subsumption is built in to the rules using the subtyping relation $\leq$ specified as follows. For base types, **bool** $\leq$ **bool** and **unit** $\leq$ **unit**. For classes C and D, we have $C \leq D$ iff either $C = D$ or the class declaration for C is **class** C **extends** B { ... } for some $B \leq D$.

To define some auxiliary notations, let

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

and let M be in the list $\overline{M}$ of method declarations, with $M = T\ m(\overline{T}_2\ \overline{x})\{S\}$. We record the typing information by defining $\mathit{mtype}(m, C) = \overline{T}_2 \to T$ and let $\mathit{pars}(m, C) = \overline{x}$ to record the parameter names. Let $\mathit{super}\,C = D$. For the declared fields, we define $\mathit{dfields}\,C = \overline{T}_1\ \overline{f}$ and $\mathit{type}(\overline{f}, C) = \overline{T}_1$. To include inherited fields, we define $\mathit{fields}\,C = \mathit{dfields}\,C \cup \mathit{fields}\,D$ and assume $\overline{f}$ is disjoint from the names in $\mathit{fields}\,D$. 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.

$$\frac{\begin{array}{c}\overline{x}:\overline{T}, \mathit{self}: C, \mathit{result}: T \vdash S \\ \mathit{mtype}(m, \mathit{super}\, C) \text{ is undefined or equals } \overline{T}\to T \\ \mathit{pars}(m, \mathit{super}\, C) \text{ is undefined or equals } \overline{x}\end{array}}{C \vdash T\ m(\overline{T}\ \overline{x})\{S\}}$$

Table 2 gives the other typing rules. A typing environment Γ is a finite function from variable names to types. A judgement of the form $\Gamma \vdash e : T$ says that e has type T in the context of a method of class Γ *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 $\bot$.

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 $\mathit{loctype} : \mathit{Loc} \to \mathit{ClassNames}$ such that for each C there are infinitely many locations ℓ with $\mathit{loctype}\ \ell = C$. We write $\mathit{locs}\ C$ for $\{\ell \mid \mathit{loctype}\ \ell = C\}$.

For functions of various kinds we write dom or rng for the domain or range.

Methods are associated with classes, in a *method environment*, rather than with instances. For this reason the semantic domains, given in Table 3, are relatively simple; there are no recursive domain equations to be solved. In addition to domains like $[\![T]\!]$ and $[\![\Gamma]\!]$ that correspond directly to syntactic notations, we use the following: $[\![\mathit{Heap}]\!]$ 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, $[\![C, \overline{x}, \overline{T} \to T]\!]$ is the set of meanings for methods of class C with result T and parameters $\overline{x}:\overline{T}$. 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 as expression values. Capturing this in the semantics is the purpose of the special cartesian products $\mathit{Heap} \otimes \Gamma$ and $\mathit{Heap} \otimes T$.

The semantics is defined for an arbitrary allocator, but the noninterference theorem depends on parametricity.

Definition 1 (Allocator, parametric)
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. An allocator is *parametric* if $\mathit{dom}\, h_1 \cap \mathit{locs}\, C = \mathit{dom}\, h_2 \cap \mathit{locs}\, C$ implies $\mathit{fresh}(C, h_1) = \mathit{fresh}(C, h_2)$. □

For example, if $\mathit{Loc} = \mathbb{N}$ the function $\mathit{fresh}(C, h) = \mathit{min}\{\ell \mid \mathit{loctype}\ \ell = C \wedge \ell \notin \mathit{dom}\, h\}$ is parametric. In practice, allocators are typically not parametric: locations are addresses in a single space used for all objects. It is possible to drop the assumption of parametricity at the cost of maintaining a bijection on the visible subset of allocated objects (in the definition of L-indistinguishability in the sequel). For our language, this is worked out in detail in [2]; here we assume parametricity to reduce notational clutter.

The semantics is defined by induction on typing judgements. The meaning of an expression $\Gamma \vdash e : T$ is a function $[\![\mathit{Heap} \otimes \Gamma]\!] \to [\![T]\!]_\bot$ that takes a state $(h, \eta) \in [\![\mathit{Heap} \otimes \Gamma]\!]$ and returns a value $d \in [\![T]\!]$ (such that $(h, d) \in [\![\mathit{Heap} \otimes T]\!]$) or the improper value $\bot$ which represents errors. Table 4 gives the definition.

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

$$[\![\mathit{MEnv}]\!] \to [\![\mathit{Heap} \otimes \Gamma]\!] \to [\![\mathit{perms}(\Gamma\ \mathit{self})]\!] \to ([\![\mathit{Heap} \otimes \Gamma]\!])_\bot$$

that takes a method environment μ (see below), a state (h, η), and the enabled permissions $Q \in [\![\mathit{perms}(\Gamma\ \mathit{self})]\!]$; it returns a state or $\bot$ which indicates divergence or error or access control violation. See Table 5.

To streamline the treatment of $\bot$ in the semantic definitions, we use a metalanguage construct, **let** $d = E_1$ **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.

Function update is written, e.g., $[\eta \mid x \mapsto d]$. In the semantics of local variables, we write $\lfloor$ for domain restriction: if x is in the domain of function η then $\eta \lfloor x$ is the function like η but without x in its domain.

The semantic domains are partially ordered. The sets $[\![\mathit{Heap}]\!]$, $[\![\mathbf{bool}]\!]$, $[\![C]\!]$, $[\![\mathit{state}\, C]\!]$, $[\![\mathit{perms}\, C]\!]$, and $\mathcal{P}(\mathit{Permissions})$ are ordered by equality. We write $\to$ for continuous function space, ordered pointwise, and $X_\bot$ for domain X with added bottom element $\bot$. Each set $[\![C, \overline{x}, \overline{T} \to T]\!]$ has a least element (the constantly-$\bot$ function) and least upper bounds of ascending chains, and this suffices for the fixpoint semantics of recursive methods.

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

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

θ	::=	T	values of type T
		$\mid \Gamma$	store (maps variables to values)
		$\mid \mathit{state}\, C$	object state (maps fields to values)
		$\mid \mathit{Heap}$	heap (maps locations to object states) with no dangling locations
		$\mid \mathit{Heap} \otimes \Gamma$	(global) states with no dangling locations
		$\mid \mathit{Heap} \otimes T$	pairs (h, d) where value d is not a dangling location w.r.t. h
		$\mid \theta_\perp$	lifting
		$\mid \mathit{perms}\, C$	permission sets authorized for C
		$\mid (C, \overline{x}, \overline{T}{\to}T)$	method of C with parameters $\overline{x}:\overline{T}$ and return type T
		$\mid \mathit{MEnv}$	method environments

$[\![\textbf{bool}]\!]$	=	$\{\mathit{true}, \mathit{false}\}$
$[\![\textbf{unit}]\!]$	=	$\{\mathit{it}\}$
$[\![C]\!]$	=	$\{\mathit{nil}\} \cup \mathit{locs}(C{\downarrow})$ where $\mathit{locs}(C{\downarrow}) = \{\ell \mid \mathit{loctype}\ \ell \leq C\}$
$[\![\Gamma]\!]$	=	$\{\eta \mid \mathit{dom}\,\eta = \mathit{dom}\,\Gamma \wedge \eta\,\mathit{self} \neq \mathit{nil} \wedge \forall x \in \mathit{dom}\,\eta.\ \eta\, x \in [\![\Gamma\, x]\!]\}$
$[\![\mathit{state}\, C]\!]$	=	$\{s \mid \mathit{dom}\, s = \mathit{dom}(\mathit{fields}\, C) \wedge \forall (f:T) \in \mathit{fields}\, C.\ s\, f \in [\![T]\!]\}$
$[\![\mathit{Heap}]\!]$	=	$\{h \mid \mathit{dom}\, h \subseteq_{\mathit{fin}} \mathit{Loc} \wedge \mathit{closed}\, h \wedge \forall \ell \in \mathit{dom}\, h.\ h\ell \in [\![\mathit{state}\ (\mathit{loctype}\ \ell)]\!]\}$ where $\mathit{closed}\, h$ iff $\mathit{rng}\, s \cap \mathit{Loc} \subseteq \mathit{dom}\, h$ for all $s \in \mathit{rng}\, h$
$[\![\mathit{Heap} \otimes \Gamma]\!]$	=	$\{(h, \eta) \mid h \in [\![\mathit{Heap}]\!] \wedge \eta \in [\![\Gamma]\!] \wedge \mathit{rng}\,\eta \cap \mathit{Loc} \subseteq \mathit{dom}\, h\}$
$[\![\mathit{Heap} \otimes T]\!]$	=	$\{(h, d) \mid h \in [\![\mathit{Heap}]\!] \wedge d \in [\![T]\!] \wedge (d \in \mathit{Loc} \Rightarrow d \in \mathit{dom}\, h)\}$
$[\![\theta_\perp]\!]$	=	$[\![\theta]\!] \cup \perp$ (where $\perp$ is some fresh value not in $[\![\theta]\!]$)
$[\![\mathit{perms}\, C]\!]$	=	$\{P \mid P \subseteq \mathit{Auth}\, C\}$
$[\![C, \overline{x}, \overline{T}{\to}T]\!]$	=	$[\![\mathit{Heap} \otimes (\overline{x}:\overline{T}, \mathit{self}: C)]\!] \to \mathcal{P}(\mathit{Permissions}) \to [\![(\mathit{Heap} \otimes T)_\perp]\!]$
$[\![\mathit{MEnv}]\!]$	=	$\{\mu \mid \forall C, m.\ \mu C m$ is defined iff $\mathit{mtype}(m, C)$ is defined, and $\mu C m \in [\![C, \mathit{pars}(m, C), \mathit{mtype}(m, C)]\!]$ if $\mu C m$ defined $\}$

Table 3. Semantic categories θ and semantic domains $[\![\theta]\!]$, for given policy Auth.

where T is the return type and $\Gamma = \mathit{self}: C, \overline{x}:\overline{T}$ is the parameter store, where $\overline{x} = \mathit{pars}(m, C)$. The result from a method, if not $\perp$, 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.

The semantics of a class table CT is the method environment, written $[\![CT]\!]$, given as the least upper bound of the ascending chain $\mu \in \mathbb{N} \to [\![\mathit{MEnv}]\!]$ defined as follows.

$\mu_0\, C\, m = \lambda(h, \eta).\ \lambda Q.\ \perp$
$\mu_{j+1}\, C\, m = [\![M]\!]\mu_j$ if m is declared as M in C
$\mu_{j+1}\, C\, m = \mu_{j+1}\, B\, m$ if m is inherited from B in C

To be very precise for an inherited method, if $\mathit{mtype}(m, C) = \overline{T} \to T$ then $\mu_{j+1}\, C\, m$ should apply to stores for $\overline{x}:\overline{T}, \mathit{self}: C$ whereas $\mu_{j+1}\, B\, m$ applies to stores for $\overline{x}:\overline{T}, \mathit{self}: B$. But the latter contains the former, as $C \leq B$ implies $[\![C]\!] \subseteq [\![B]\!]$ (see [2]). 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 $[\![C, \overline{x}, \overline{T}{\to}T]\!]$ is defined using $\mathcal{P}(\mathit{Permissions})$ rather than $[\![\mathit{perms}\, C]\!]$.

For a method declaration $M = T\ m(\overline{T}\ \overline{x})\{S\}$ in class

$$
\begin{array}{lcl}
[\![\Gamma \vdash x : T]\!](h,\eta) & = & \eta x \\
[\![\Gamma \vdash \textbf{null} : B]\!](h,\eta) & = & \mathit{nil} \\
[\![\Gamma \vdash \textbf{unit} : \textbf{unit}]\!](h,\eta) & = & \mathit{it} \\
[\![\Gamma \vdash \textbf{true} : \textbf{bool}]\!](h,\eta) & = & \mathit{true} \\
[\![\Gamma \vdash \textbf{false} : \textbf{bool}]\!](h,\eta) & = & \mathit{false} \\
[\![\Gamma \vdash e_1 == e_2 : \textbf{bool}]\!](h,\eta) & = & \textbf{let } d_1 = [\![\Gamma \vdash e_1 : T_1]\!](h,\eta) \textbf{ in} \\
 & & \textbf{let } d_2 = [\![\Gamma \vdash e_2 : T_2]\!](h,\eta) \textbf{ in if } d_1 = d_2 \textbf{ then } \mathit{true} \textbf{ else } \mathit{false} \\
[\![\Gamma \vdash e.f : T]\!](h,\eta) & = & \textbf{let } \ell = [\![\Gamma \vdash e : (\Gamma\ \mathit{self})]\!](h,\eta) \textbf{ in if } \ell = \mathit{nil} \textbf{ then } \bot \textbf{ else } h\,\ell\,f \\
[\![\Gamma \vdash (B)\ e : B]\!](h,\eta) & = & \textbf{let } \ell = [\![\Gamma \vdash e : D]\!](h,\eta) \textbf{ in if } \ell = \mathit{nil} \vee \mathit{loctype}\ \ell \leq B \textbf{ then } \ell \textbf{ else } \bot \\
[\![\Gamma \vdash e \textbf{ is } B : \textbf{bool}]\!](h,\eta) & = & \textbf{let } \ell = [\![\Gamma \vdash e : D]\!](h,\eta) \textbf{ in if } \ell \neq \mathit{nil} \wedge \mathit{loctype}\ \ell \leq B \textbf{ then } \mathit{true} \textbf{ else } \mathit{false}
\end{array}
$$

Table 4. Semantics of expressions

C, here is the semantics:

$$
\begin{array}{l}
[\![M]\!]\mu(h,\eta)Q = \\
\quad \textbf{let } Q_1 = Q \cap \mathit{Auth}\, C \textbf{ in} \\
\quad \textbf{let } \eta_1 = [\eta \mid \mathit{result} \mapsto \mathit{default}] \textbf{ in} \\
\quad \textbf{let } (h_0, \eta_0) = [\![\overline{x} : \overline{T}, \mathit{self} : C, \mathit{result} : T \vdash S]\!]\mu(h,\eta)Q_1 \textbf{ in} \\
\quad (h_0, \eta_0\ \mathit{result})
\end{array}
$$

5. Security typing

In this section we annotate the syntax of Section 4 with security labels. Where a type T could occur, we use pairs (T, κ) where κ is a security level, L or H. The grammar is revised as follows.

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

Note that there is no change for cast and test. As discussed in Section 8, labels for local variables can be inferred.

In [4], the annotation of a method appears as $\tau\ m(\overline{\tau}\ \overline{x})\ \kappa\ \{S\}$, with κ designating the heap effect and τ the level of the result. By analogy with the auxiliary function *mtype* which gives the declared type of a method, function *smtype* is used in [4] to give declared annotation $\overline{\tau} \xrightarrow{\kappa} \tau$ as discussed in Section 2.

In this paper we allow multiple typings of a method, each of the form $\overline{\kappa} \xrightarrow{P;\kappa_1} \kappa_2$. The intended meaning is as follows: if the method is called with arguments compatible with $\overline{\kappa}$ and enabled permissions disjoint from P then the heap effect is $\geq \kappa_1$ and the result level $\leq \kappa_2$. By contrast with [4] as discussed above, we do not annotate method parameters, heap effect, or result levels in the concrete syntax. Instead we assume a function *smtypes* is given.

Definition 2 (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 $\overline{\kappa} \xrightarrow{P;\kappa_1} \kappa_2$. Third, if $C \leq D$ 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 *smtypes*.

As in our previous work [4], each class is assigned a level which determines the type of *self*. For L classes, flows are tracked in terms of fields (which may be H or L), variables and control flow (and the rules are similar to those in Jif [16]). Classes labelled H are also subject to a confinement condition that separates all values of the type from L fields and variables. This can be seen as a form of access control, as will be explored in future work. Here the main benefit is that we avoid more complicated tracking of information flow via allocation.

Tables 6 and 7 give typing rules for annotated programs. We write Δ for typing environments that assign security types. A judgement $\Delta; P \vdash S : (\mathit{com}\ \kappa_1, \kappa_2)$ says that S is safe and assigns only to variables (locals and parameters) of level $\geq \kappa_1$ and to object fields of level $\geq \kappa_2$ (see Lemma 6.4) provided that no permissions in set P are enabled initially.

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. We write "$-$" for set subtraction.

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

Table 5. Semantics of commands, for given policy $Auth$ and allocator $fresh$.

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

$$
\frac{\Delta \vdash e_1 : (T_1,\kappa_1) \quad \Delta \vdash e_2 : (T_2,\kappa_2)}{\Delta \vdash e_1 == e_2 : (\textbf{bool}, \kappa_1 \sqcup \kappa_2)}
\qquad
\frac{C = \Delta^\dagger \mathit{self} \quad (T,\kappa_1) f \in \mathit{sdfields}\, C \quad \Delta \vdash e : (C,\kappa_2)}{\Delta \vdash e.f : (T, \kappa_1 \sqcup \kappa_2)}
\qquad
\frac{\Delta \vdash e : (D,\kappa) \quad B \le D}{\Delta \vdash (B)\ e : (B,\kappa)}
\qquad
\frac{\Delta \vdash e : (D,\kappa) \quad B \le D}{\Delta \vdash e \textbf{ is } B : (\textbf{bool},\kappa)}
$$

Table 6. Security typing rules for expressions.

For commands, it suffices to consider judgements where $P \subseteq \mathit{Auth}(\Delta^\dagger \mathit{self})$; only such P is relevant to the behavior of a command declared in class $\Delta^\dagger \mathit{self}$, as can be seen in the rule for method declaration below.

The rule for class declarations is as follows.

$$
\frac{\begin{array}{c}\mathit{level}\, D \le \kappa \\ C\ \kappa \textbf{ extends } D \vdash M \ \text{ for each } M \in \overline{M} \\ \text{If } \mathit{level}\, D \ne \kappa \text{ then every } m \text{ with } \mathit{mtype}(m,D) \\ \text{defined is overridden in } C \text{ by some } M \in \overline{M}\end{array}}{\vdash \textbf{class } C\ \kappa \textbf{ extends } D\ \{\ \overline{\tau}\ \overline{f};\ \overline{M}\ \}}
$$

The condition on overriding prevents bad flows in the case where code is checked where *self* is L but inherited in an object of H class [4]. Here is the rule for method declaration.

$$
\frac{\begin{array}{l}\text{For each } (\overline{\kappa} \xrightarrow{P;\kappa_3} \kappa_4) \in \mathit{smtypes}(m,C) \text{ we have} \\ \Delta; (P \cap \mathit{Auth}\, C) \vdash S : (\mathit{com}\ \kappa_2, \kappa_3) \\ \text{where } \ \overline{T} \to T = \mathit{mtype}(m,C) \\ \qquad\quad \overline{x} = \mathit{pars}(m,C) \\ \qquad\quad \Delta = \overline{x} : (\overline{T},\overline{\kappa}), \mathit{self} : (C,\kappa_1), \mathit{result} : (T,\kappa_4)\end{array}}{C\ \kappa_1 \textbf{ extends } D \vdash T\ m(\overline{T}\ \overline{x})\{S\}}
$$

The hypothesis checks the method body against all typings in *smtypes*, restricting the permissions to those that can possibly be enabled for this class. There is no constraint on κ_2 because it tracks assignments to local variables.

The rules in Table 6 and 7 use versions of the auxiliary functions that take security levels into account. Let

$$
CT(C) = \textbf{class } C\ \kappa_1 \textbf{ extends } D\ \{\ \overline{\tau_1}\ \overline{f};\ \overline{M}\ \}
$$

$$\frac{x \neq \mathit{self} \quad T_2 \leq T_1 \quad \kappa_2 \sqcup \kappa_3 \leq \kappa_1 \quad \Delta, x:(T_1,\kappa_1); P \vdash e:(T_2,\kappa_2)}{\Delta, x:(T_1,\kappa_1); P \vdash x := e:(com\ \kappa_3, \kappa_4)} \qquad \frac{C = \Delta^\dagger \mathit{self} \quad (T,\kappa_2) f \in \mathit{sdfields}\, C \quad \Delta \vdash e_1:(C,\kappa_1) \quad \Delta \vdash e_2:(U,\kappa_3) \quad U \leq T \quad \kappa_1 \sqcup \kappa_3 \sqcup \kappa_5 \leq \kappa_2}{\Delta; P \vdash e_1.f := e_2:(com\ \kappa_4, \kappa_5)}$$

$$\frac{x \neq \mathit{self} \quad B \leq D \quad \mathit{level}\, B \sqcup \kappa_2 \leq \kappa_1 \quad \kappa_3 \leq \mathit{level}\, B}{\Delta, x:(D,\kappa_1); P \vdash x := \mathbf{new}\ B:(com\ \kappa_2, \kappa_3)}$$

$$\frac{\overline{\kappa} \xrightarrow{P';\kappa_1} \kappa_2 \in \mathit{smtypes}(m,D) \quad P' \cap \mathit{Auth}(\Delta^\dagger \mathit{self}) \subseteq P \quad \mathit{mtype}(m,D) = \overline{T} \to T' \quad \Delta, x:(T,\kappa) \vdash e:(D,\kappa_3) \quad \Delta, x:(T,\kappa) \vdash \overline{e}:(\overline{U},\overline{\kappa}_4) \quad T' \leq T \quad \overline{U} \leq \overline{T} \quad \overline{\kappa}_4 \leq \overline{\kappa} \quad \kappa_2 \sqcup \kappa_3 \sqcup \kappa_5 \leq \kappa \quad \kappa_3 \sqcup \kappa_6 \leq \kappa_1}{\Delta, x:(T,\kappa); P \vdash x := e.m(\overline{e}):(com\ \kappa_5, \kappa_6)} \qquad \frac{\Delta; P \vdash S_1:(com\ \kappa_1, \kappa_2) \quad \Delta; P \vdash S_2:(com\ \kappa_3, \kappa_4) \quad \kappa_5 \leq \kappa_1 \sqcap \kappa_3 \quad \kappa_6 \leq \kappa_2 \sqcap \kappa_4}{\Delta; P \vdash S_1\ ;\ S_2:(com\ \kappa_5, \kappa_6)}$$

$$\frac{\Delta \vdash e:(\mathbf{bool},\kappa_5) \quad \Delta; P \vdash S_1:(com\ \kappa_1,\kappa_3) \quad \Delta; P \vdash S_2:(com\ \kappa_2,\kappa_4) \quad \kappa_5 \leq \kappa_6 \sqcap \kappa_7 \quad \kappa_6 \leq \kappa_1 \sqcap \kappa_2 \quad \kappa_7 \leq \kappa_3 \sqcap \kappa_4}{\Delta; P \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2:(com\ \kappa_6, \kappa_7)} \qquad \frac{\Delta \vdash e:(U,\kappa_4) \quad \Delta, x:(T,\kappa_1); P \vdash S:(com\ \kappa_2,\kappa_3) \quad U \leq T \quad \kappa_4 \leq \kappa_1 \quad \kappa_2 \leq \kappa_5 \quad \kappa_3 \leq \kappa_6}{\Delta; P \vdash (T,\kappa_1)\ x := e\ \mathbf{in}\ S:(com\ \kappa_5, \kappa_6)}$$

$$\frac{\Delta; (P - (P' \cap \mathit{Auth}(\Delta^\dagger \mathit{self}))) \vdash S:(com\ \kappa_1,\kappa_2) \quad \kappa_3 \leq \kappa_1 \quad \kappa_4 \leq \kappa_2}{\Delta; P \vdash \mathbf{enable}\ P'\ \mathbf{in}\ S:(com\ \kappa_3, \kappa_4)}$$

$$\frac{P' \cap P = \varnothing \quad \kappa_5 \leq \kappa_1 \sqcap \kappa_3 \quad \kappa_6 \leq \kappa_2 \sqcap \kappa_4 \quad \Delta; P \vdash S_1:(com\ \kappa_1,\kappa_2) \quad \Delta; P \vdash S_2:(com\ \kappa_3,\kappa_4)}{\Delta; P \vdash \mathbf{test}\ P'\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2:(com\ \kappa_5, \kappa_6)} \qquad \frac{P' \cap P \neq \varnothing \quad \Delta; P \vdash S_2:(com\ \kappa_1,\kappa_2) \quad \kappa_3 \leq \kappa_1 \quad \kappa_4 \leq \kappa_2}{\Delta; P \vdash \mathbf{test}\ P'\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2:(com\ \kappa_3, \kappa_4)}$$

Table 7. Security typing rules for commands, for given $Auth$.

Corresponding to *dfields*, *fields* and *type*, we define *sdfields*, *sfields* and *stype* which differ only in that they give security types, e.g., $\mathit{sdfields}\, C = \overline{\tau_1}\ \overline{f}$. We also need a function *level* that gives the level associated with the class itself: for the declaration above, $\mathit{level}\, C = \kappa_1$. Define $\mathit{level}\ \mathbf{Object} = L$. For locations, define $\mathit{level}\ \ell = \mathit{level}(\mathit{loctype}\ \ell)$.

Examples. Let us consider how the examples of Section 3 are formalized. For the information flow policy, class *Kern* is annotated as

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

and class *Vend1* is annotated as

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

The most interesting method typings are given as

$\mathit{smtypes}(\mathit{getHinfo, Kern}) = \{L \xrightarrow{\varnothing;H} H\}$
$\mathit{smtypes}(\mathit{getStatus, Kern}) = \{L \xrightarrow{\{stat\};H} L,\ L \xrightarrow{\varnothing;H} H\}$
$\mathit{smtypes}(\mathit{status,Vend1}) = \{L \xrightarrow{\varnothing;H} L\}$

The body of *Vend1.status* is checked in the context of $\mathit{result}:(\mathbf{string}, L)$ and excluded permission set $\varnothing \cap \mathit{Auth}(\mathit{Vend1})$, i.e., $\varnothing$. The call to *getStatus* can be checked using the type $L \xrightarrow{\{stat\};H} L$, because $\{stat\} \cap \mathit{Auth}(\mathit{Vend1}) = \{stat\} \cap \{other\} \subseteq \varnothing$.

Method *Vend1.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 $stat \notin \mathit{Auth}(\mathit{Vend1})$.

Consider the policy $\mathit{smtypes}(\mathit{statusH, Vend2}) = \{L \xrightarrow{\varnothing;H} H\}$. This requires checking the body of *Vend2.statusH* in the context of $\mathit{result}:(\mathbf{string}, H)$ and $\varnothing$. To check the command

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

we check *result* := *k.getStatus()* in the context of $\varnothing$, which is $\varnothing - \{stat\}$. This check succeeds, using the type $L \xrightarrow{\varnothing;H} H$ for *getStatus*.

Finally, consider checking *getStatus*. To check it with respect to $L \xrightarrow{\varnothing;H} H$ requires checking both branches of **test** *stat* **then enable** *sys* **in** *result* := *getHinfo()* **else** ... in the context with $\mathit{result}:(\mathbf{string}, H)$. This succeeds. To check with respect to $L \xrightarrow{\{stat\};H} L$, 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 $\{stat\}$, the second of the rules for **test** is applicable and the **then** branch is not checked.

Finally, we consider subclassing. Here is an untrusted subclass of *Kern* that adds a method; suppose $Auth(\mathit{KernSub}) = \{other\}$.

class *KernSub* **extends** *Kern* { // *permissions: other*
string *myStatus()*{ // *type* $L \to L$
result := *self.getHinfo()* } ... }

In fact the call to *getHinfo* aborts, for lack of permission sys. This program is untypable for policy $L \to L$ only if we include $L \xrightarrow{\{sys\}} L$ in *smtypes*(*getHinfo*, *Kern*).

Suppose *KernSub* also included an overriding declaration for *getStatus*. Definition 2 requires *smtypes*(*getStatus, KernSub*) $= \{L \xrightarrow{\varnothing;H} L,\ L \xrightarrow{\{stat\};H} H\}$ and thus two checks of the method body. The interesting case is for $L \xrightarrow{\{stat\};H} H$. Here the body is checked in the context $\{stat\} \cap \{other\}$ which is $\varnothing$. In particular, this precludes a call to *getHinfo*. For lack of space we do not give a thorough discussion of overriding. Moreover, we omit super-calls, although they pose no difficulty.

Properties of security typing. For any judgement $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2)$ derivable using the rules in Tables 6 and 7, the erased judgement $\Delta^\dagger \vdash S^\dagger$ is derivable using the rules of Table 2. Conversely, any program typable using the rules of Table 2 can be annotated everywhere by L and typed by the rules in Tables 6 and 7, taking *smtypes*$(m, C) = \{L \xrightarrow{\varnothing} L\}$ for all m, C.

If a method is typable with respect to $\overline{\kappa} \xrightarrow{P;\kappa_1} \kappa_2$ then it is also typable with respect to $\overline{\kappa}' \xrightarrow{P';\kappa_1'} \kappa_2'$ for any $\overline{\kappa}' \le \overline{\kappa}$, $P' \supseteq P$, $\kappa_1' \le \kappa_1$, and $\kappa_2' \ge \kappa_2$. We do not prove this fact here because it is not needed for our main result. But it suggests a notion of subtyping and subsumption that would be important for practical application.

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 $\Delta, x : (T, \kappa); P \vdash x := e.m(\overline{e}) : (com\ \kappa_5, \kappa_6)$ is derivable using $\overline{\kappa} \xrightarrow{P';\kappa_1} \kappa_2 \in$ *smtypes*(m, D). If $Q \# P$ and $Q \subseteq Auth(\Delta^\dagger\ \mathit{self})$ then $Q \# P'$.

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

Remarks about proofs. The proofs involve detailed analysis of the semantics and the security typing rules. For each specific case, the semantic definition may involve several values (e.g., the value of e is needed in the semantics of $x := e$), and the rule may involve several types and security labels. In writing a given proof case, we found it convenient to write down both the rule and the semantics for reference. It is impractical to include such redundancy in the paper, however. Instead, when it comes to proving something about a particular construct we make free use of identifiers in the typing rule (in Table 6 or 7), for types and labels, and identifiers in the semantic definition for semantic values (in Table 4 or 5). Note that the semantic definition may use different identifiers for types, as the semantics is based on the typing rules in Table 2 rather than the security rules in Tables 6 and 7. We streamline the proofs by ignoring $\perp$ outcomes and omitting many cases.

6. Confinement

This section shows that if a program is accepted by the security typing rules of Section 5 then it maintains the invariant that L fields and variables never hold H locations. Moreover, commands with H effect do not assign to L-fields or L-variables. These results are similar to those in [4], especially for the imperative control constructs.

The formalization uses the indistinguishability relation $\sim$ also used in the main results of Section 7. In formalizing the absence of L-variables that refer to H-objects, we take advantage of the fact that $nil \notin Loc$ and $\perp \notin Loc$. We use the short name "ok" for L-confinement.

Definition 3 (L-confinement (ok))

- Define $LLoc = \{\ell \in Loc \mid \mathit{level}\,\ell = L\}$.
- For heaps, define $ok\,h$ iff for all $\ell \in dom\,h$ and every $f \in \mathit{fields}(\mathit{loctype}\,\ell)$, if $stype(f, \mathit{loctype}\,\ell) = (T, L)$ for some T and $h\ell f \in Loc$ then $h\ell f \in LLoc$.
- For stores, define $ok\,\Delta\,\eta$ iff for every x with $\Delta\,x = (T, L)$ for some T, if $\eta\,x \in Loc$ then $\eta\,x \in LLoc$.
- For method environments, $ok\,\mu$ iff the following holds for every m, C and $\overline{\kappa} \xrightarrow{P;\kappa_2} \kappa$ in *smtypes*(m, C). For all η, h, Q, if $ok\,h$, $ok\,\Delta\,\eta$, $Q \# P$, and $\mu Cm(h, \eta)Q \neq \perp$ then $ok\,h_0$ and $\kappa = L \wedge d \in Loc \Rightarrow d \in LLoc$,
where $(h_0, d) = \mu Cm(h, \eta)Q$
and $\overline{T} \to T = \mathit{mtype}(m, C)$,
and $\Delta = \mathit{pars}(m, C) : (\overline{T}, \overline{\kappa})$, *self*$: (C, \mathit{level}\,C)$.

Lemma 6.1 (L-confinement of expressions)
Let $\Delta \vdash e : (T, L)$ and let $d = [\![\Delta^\dagger \vdash e : T]\!](h, \eta)$. If $ok\,\Delta\,\eta$, and $ok\,h$ then $d \in Loc \Rightarrow d \in LLoc$.

Lemma 6.2 (L-confinement of commands)
Let $\Delta; P \vdash S : (com\,\kappa_1, \kappa_2)$. If $ok\,\mu$, $ok\,h$, $ok\,\Delta\,\eta$, $Q \subseteq Auth(\Delta^\dagger\, self)$, $Q \# P$, and $[\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h,\eta)Q \neq \bot$ then $ok\,\Delta\,\eta_0$ and $ok\,h_0$, where $(h_0, \eta_0) = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h,\eta)Q$.

Proof: By induction on the derivation of $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2)$. We consider cases on S.

- $x := e.m(\overline{e})$: We have $\eta_0 = [\eta \mid x \mapsto d_0]$, and $ok\,\Delta\,\eta$ by assumption, so it suffices to show $d_0 \in Loc \Rightarrow d_0 \in LLoc$. Moreover we must show $ok\,h_0$. Let $\Delta_1 = \overline{x} : (\overline{T}, \overline{\kappa})$, $self : (loctype\,\ell, level\,\ell)$, and $\eta_1 = [\overline{x} \mapsto \overline{d}, self \mapsto \ell]$. Lemma 5.1 is applicable and yields $Q \# P'$. We claim $ok\,\Delta_1\,\eta_1$; then we get the result by $ok\ \mu$. It remains to show the claim. If $\overline{\kappa} = L$, then by the typing rule, $\overline{\kappa}_4 = L$. So by induction on $\overline{e}$, and since the semantics is non-$\bot$, we get $\overline{d} \in Loc \Rightarrow \overline{d} \in LLoc$. Hence $ok\,(\overline{x} : (\overline{T}, \overline{\kappa}))\,[\overline{x} \mapsto \overline{d}]$. We get $ok\,(self : (loctype\,\ell, level\,\ell))\,[self \mapsto \ell]$ directly from the definitions of ok and $LLoc$, as $level\,\ell = L$ iff $\ell \in LLoc$.
- **enable** P' **in** S: To use induction on S it suffices that $(Q \cup (P' \cap Auth(\Delta^\dagger self)) \# (P - (P' \cap Auth(\Delta^\dagger self))$. This follows by set theory from hypothesis $Q \# P$.
- **test** P' **then** S_1 **else** S_2: Suppose $P' \subseteq Q$. Thus $P' \# P$, by hypothesis $Q \# P$. Thus the first of the rules in Table 7 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 which is checked by the rule. So we can use induction on S_1 which yields the result.

  In the other case, $P' \not\subseteq Q$, 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 depend on checking S_2 so we get the result by induction on S_2. □

Lemma 6.3 (L-confinement of method environments)
If annotated class table CT satisfies the security typing rules then $ok[\![CT^\dagger]\!]$ and also $ok\,\mu_i$ for each μ_i in the approximation chain defining $[\![CT^\dagger]\!]$. □

The proof is by induction on i, using Lemmas 6.1 and 6.2, and then fixpoint induction for $[\![CT^\dagger]\!]$. It follows the pattern of the proof of Theorem 7.3 and is given in the full paper.

Next we formalize the indistinguishability relation $\sim$. Object states are indistinguishable by L if their L-fields are equal, and stores are indistinguishable if their L-variables are equal. In the case of heaps and object states, the relevant levels are determined by the field declarations in the class table. By contrast, the levels for stores are determined by parameter and local variable declarations, hence the dependence is explicit in the notation $\sim_\Delta$. It is straightforward to show that each of these is an equivalence relation.

Definition 4 (Indistinguishable by L)

- For $s, s' \in [\![state\,C]\!]$, define $s \sim s'$ iff $\forall f \in fields\,C$. **let** $(T, \kappa) = stype(f, C)$ **in** $(\kappa = L \Rightarrow sf = s'f)$.
- For $h, h' \in [\![Heap]\!]$, define $h \sim h'$ iff $dom\ h \cap LLoc = dom\ h' \cap LLoc$ and $\forall \ell \in dom\ h \cap LLoc$. $h\ell \sim h'\ell$.
- For $\eta, \eta' \in [\![\Delta^\dagger]\!]$, define $\eta \sim_\Delta \eta'$ iff $\forall x \in dom\ \Delta$. **let** $(T, \kappa) = \Delta\ x$ **in** $(\kappa = L \Rightarrow \eta x = \eta' x)$. □

If a command is typable as $(com\ H, \kappa)$ it does not assign to L-variables, and if it is typable as $(com\ \kappa_2, H)$ it does not assign to L-fields of objects.

Definition 5 (H-confined method environment)
Method environment μ is H-confined, written $Hconf\ \mu$, if the following holds for all C, m and all $\overline{\kappa} \xrightarrow{P;H} \kappa$ in $smtypes(m, C)$. If $Q \# P$ and $\mu Cm(h,\eta)Q \neq \bot$ then $h_0 \sim h$, where $(h_0, d) = \mu Cm(h,\eta)Q$. □

Lemma 6.4 (H-confinement of commands)
Let $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2)$. Then for all μ, η, h, Q such that $Hconf\ \mu$, $Q \# P$, and $[\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h,\eta)Q \neq \bot$ we have

- if $\kappa_1 = H$ and $(h_0, \eta_0) = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h,\eta)Q$ then $\eta \sim_\Delta \eta_0$.
- if $\kappa_2 = H$ and $(h_0, \eta_0) = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h,\eta)Q$ then $h \sim h_0$.

Proof: By induction on $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2)$. Recall the conventions described at the end of Section 5: level identifiers in the proof are those in the relevant rules, *not* κ_1, κ_2 as used in the statement of the Lemma.

- **if** e **then** S_1 **else** S_2: First, assume $\kappa_6 = H$. Then by the typing rule, $\kappa_1 = H$ and $\kappa_2 = H$. Let $b = [\![\Delta^\dagger \vdash e : \mathbf{bool}]\!](h,\eta)$. Then if $b = true$, the result follows by induction on S_1 and if $b = false$, the result follows by induction on S_2.

  Next, assume $\kappa_7 = H$. Then by the typing rule, $\kappa_3 = H$ and $\kappa_4 = H$. Again, the result follows by induction on S_1 if $b = true$ and by induction on S_2 if $b = false$.
- $x := e.m(\overline{e})$: For the store, suppose $\kappa_5 = H$. Then by typing, $\kappa = H$. That $\eta \sim_{\Delta, x:(T,H)} [\eta \mid x \mapsto d_0]$ now follows by definition $\sim_{\Delta, x:(T,H)}$. For the heap, suppose $\kappa_6 = H$. Then we must show $h \sim h_0$, where $(h_0, d_0) = \mu(loctype\,\ell)m(h, [\overline{x} \mapsto \overline{d}, self \mapsto \ell])Q$, and $\ell = [\![\Delta^\dagger \vdash e : D]\!](h,\eta)$ and $\overline{d} = [\![\Delta^\dagger \vdash \overline{e} : \overline{U}]\!](h,\eta)$. Because $\kappa_6 = H$, we have by the typing rule, $\kappa_1 = H$. Moreover, $\overline{\kappa} \xrightarrow{P';H} \kappa_2 \in smtypes(m, (loctype\,\ell))$. Using

hypothesis $Q \# P$ and the typing rule, we can apply Lemma 5.1 to get $Q \# P'$. So we can use assumption $Hconf\ \mu$ to get $h \sim h_0$.

- **enable** P' **in** S and **test** P' **then** S_1 **else** S_2: By induction, justified by the same reasoning as in the proof of Lemma 6.2. □

Lemma 6.5 (H-confinement of method environments)
If annotated class table CT satisfies the security typing rules then $Hconf [\![CT^\dagger]\!]$ and also $Hconf\ \mu_i$ for each μ_i in the approximation chain defining $[\![CT^\dagger]\!]$. □

The proof is by induction on i, using Lemma 6.4, and then fixpoint induction.

7. 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 is safe.

A method meaning is safe, i.e., noninterfering, provided that, for terminating computations, L-indistinguishable initial heaps and stores lead to L-indistinguishable results.

Definition 6 (Safe method environment)
We define $safe\,\mu$ iff for all C, m, all $\overline{\kappa} \xrightarrow{P;\kappa_2} \kappa$ in $smtypes(m, C)$, and all h, h', η, η', Q we have:

$$\begin{array}{l}\text{if } h \sim h', \eta \sim_\Delta \eta', Q \# P, ok\,h, ok\,h', ok\,\Delta\,\eta, ok\,\Delta\,\eta' \\ \text{and } \mu Cm(h,\eta)Q \neq \bot \neq \mu Cm(h',\eta')Q \\ \text{then } h_0 \sim h_0' \text{ and } (\kappa = L \Rightarrow d = d') \\ \text{where } \begin{array}{lcl}\Delta & = & \overline{x} : (\overline{T}, \overline{\kappa}), self : (C, level\,C) \\ (h_0, d) & = & \mu Cm(h,\eta)Q \\ (h_0', d') & = & \mu Cm(h',\eta')Q\end{array}\end{array}$$

If an expression can be typed $\Delta \vdash e : (T, L)$ then its meaning is the same in two L-indistinguishable states, provided that it diverges in neither state.

Lemma 7.1 (Safe expressions)
Suppose $\Delta \vdash e : (T, L)$. Suppose that $h \sim h'$, $\eta \sim_\Delta \eta'$, $ok\,h$, $ok\,h'$, $ok\,\Delta\,\eta$, $ok\,\Delta\,\eta'$, and $[\![\Delta^\dagger \vdash e : T]\!](h,\eta) \neq \bot \neq [\![\Delta^\dagger \vdash e : T]\!](h',\eta')$. Then $[\![\Delta^\dagger \vdash e : T]\!](h,\eta) = [\![\Delta^\dagger \vdash e : T]\!](h',\eta')$.

Proof: In this proof and subsequent ones, we extend the convention described at the end of Section 5. When comparing semantics for a pair of states (h,η) and (h',η'), we use corresponding primes on identifiers in the semantic definitions. For example, the semantic definition of $[\![\Delta^\dagger \vdash x := e]\!]\mu(h,\eta)P$ involves value d denoted by e in state (h,η), so we write d' for the corresponding value for $[\![\Delta^\dagger \vdash x := e]\!]\mu(h',\eta')P$.

The proof is by induction on e. We consider the case $e.f$: By typing, $\kappa_1 = L = \kappa_2$. Because $\kappa_2 = L$ we can use induction on e; this yields that there is ℓ with $[\![\Delta^\dagger \vdash e : C]\!](h,\eta) = \ell = [\![\Delta^\dagger \vdash e : C]\!](h',\eta')$, as we only consider the case that both semantics are non-$\bot$. For the same reason, ℓ is in the domain of both h and h'. By $\kappa_2 = L$ and Lemma 6.1 we have $\ell \in LLoc$ so, by assumption $h \sim h'$, we get $h\ell \sim h'\ell$; this implies $h\ell f = h'\ell f$ because field f has label $\kappa_1 = L$. □

Lemma 7.2 (Safe commands)
Suppose $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2)$. Suppose also $ok\,\mu$, $ok\,h$, $ok\,h'$, $ok\,\Delta\,\eta$, $ok\,\Delta\,\eta'$, $Q \# P$, $safe\,\mu$, $Hconf\,\mu$, $\eta \sim_\Delta \eta'$, $h \sim h'$, and $[\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h,\eta)Q \neq \bot \neq [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h',\eta')Q$. Then $\eta_0 \sim_\Delta \eta_0'$ and $h_0 \sim h_0'$, where $(h_0,\eta_0) = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h,\eta)Q$ and $(h_0',\eta_0') = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h',\eta')Q$.

Proof: We show $\eta_0 \sim_\Delta \eta_0'$ and $h_0 \sim h_0'$ by induction on S. The most interesting case is method call.

- $x := e.m(\overline{e})$: Let $\eta_0 = [\eta \mid x \mapsto d_0]$ and $\eta_0' = [\eta' \mid x \mapsto d_0']$. We show $\eta_0 \sim_{\Delta, x:(T,\kappa)} \eta_0'$ and $h_0 \sim h_0'$ by cases on κ_3. If $\kappa_3 = H$ then it is possible that $\ell \neq \ell'$ and thus the two calls can have different behavior. But by the typing constraint $\kappa_3 \leq \kappa$ we have $\kappa = H$ and thus $\eta_0 \sim_{\Delta, x:(T,\kappa)} \eta_0'$ follows by definition $\sim_{\Delta, x:(T,\kappa)}$. Also, by the typing constraint $\kappa_3 \leq \kappa_1$ we have $\kappa_1 = H$. By Lemma 5.1 we get $Q \# P'$. Thus we can use $Hconf\ \mu$ to obtain $h_0 \sim h \sim h' \sim h_0'$.

  It remains to consider the case $\kappa_3 = L$. In this case, we have $\ell = \ell'$ by Lemma 7.1. Now let $\Delta_1 = \overline{x} : (\overline{T}, \overline{\kappa}), self : (loctype\,\ell, level\,\ell)$, and $\eta_1 = [\overline{x} \mapsto \overline{d}, self \mapsto \ell]$, and $\eta_1' = [\overline{x} \mapsto \overline{d}', self \mapsto \ell']$. We claim that $ok\,\Delta_1\,\eta_1$ and $ok\,\Delta_1\,\eta_1'$ and $\eta_1 \sim_{\Delta_1} \eta_1'$. Then, because $Q \# P'$, we get the results $\eta_0 \sim_{\Delta, x:(T,\kappa)} \eta_0'$ and $h_0 \sim h_0'$ by $safe\,\mu$.

  It remains to prove the claims. We give the argument for the case that $\overline{x}$ is a single identifier, as the generalization is obvious but awkward to put into words.

  For $\eta_1 \sim_{\Delta_1} \eta_1'$, note that since $\ell = \ell'$ it suffices to deal with $\overline{d}, \overline{d'}$ regardless of whether $level\,\ell = L$. If $\overline{\kappa} = L$ then we need $\overline{d} = \overline{d'}$. Now $\overline{\kappa} = L$ implies $\overline{\kappa}_4 = L$ by typing, and then we get $\overline{d} = \overline{d'}$ by Lemma 7.1 on $\overline{e}$; moreover Lemma 6.1 yields $ok\,(\overline{x} : (\overline{T}, \overline{\kappa}))\,[\overline{x} \mapsto \overline{d}]$. Thus $ok\,\Delta_1\,\eta_1$ because $ok\,(self : (loctype\,\ell, level\,\ell))\,[self \mapsto \ell]$ holds for any ℓ. We have $ok\,\Delta_1\,\eta_1'$ *mutatis mutandis*.

- **if** e **then** S_1 **else** S_2: We proceed by cases on level κ_5 of the guard e. Suppose $\kappa_5 = L$. Then by Lemma 7.1 for 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, $\kappa_5 = H$. By typing, $\kappa_6 = H = \kappa_7$ and $\kappa_1 = \kappa_2 = \kappa_3 = \kappa_4 = H$. Let $(h_0, \eta_0) = [\![\Delta^\dagger \vdash \textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2]\!]\mu(h, \eta)Q$ and $(h_0', \eta_0') = [\![\Delta^\dagger \vdash \textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2]\!]\mu(h', \eta')Q$. By H-confinement Lemma 6.4 we have $\eta \sim_\Delta \eta_0$, $\eta' \sim_\Delta \eta_0'$, $h \sim h_0$, and $h' \sim h_0'$. Using assumptions $\eta \sim_\Delta \eta'$ and $h \sim h'$ we get $\eta_0 \sim_\Delta \eta_0'$ and $h_0 \sim h_0'$ by transitivity.

- $x := \textbf{new } B$: For the store, we must show $[\eta \mid x \mapsto \ell] \sim_{\Delta, x:(D_1,\kappa_1)} [\eta' \mid x \mapsto \ell']$. By assumption $\eta \sim_{\Delta, x:(D_1,\kappa_1)} \eta'$ it is enough to deal with x; that is, if $\kappa_1 = L$ we need $\ell = \ell'$. By the typing rule, $\kappa_1 = L$ implies $\mathit{level}\, B = L$. Thus, by $h \sim h'$, we have $dom\, h \cap locs\, B = dom\, h' \cap locs\, B$; then $\ell = \ell'$ by parametricity of the allocator (Definition 1).

  Finally, we get $h_0 \sim h_0'$ as follows. If either ℓ or ℓ' is in $LLoc$ then $\mathit{level}\, B = L$ so by parametricity of the allocator we get $\ell = \ell'$, satisfying the domain condition for $h_0 \sim h_0'$. For the range, i.e., $h\,\ell \sim h'\,\ell$, the result holds because the new object states are identical.

- **enable** P' **in** S and **test** P' **then** S_1 **else** S_2: By induction, justified by the same reasoning as in the proof of Lemma 6.2. □

Theorem 7.3 (Noninterfering programs)
If annotated class table CT satisfies the security typing rules then its meaning $[\![CT^\dagger]\!]$ is safe.

Proof: Because $[\![CT^\dagger]\!]$ is defined as the least upper bound of an approximation chain, we first show $\mathit{safe}\, \mu_i$ for all i, by induction on i. Then the result follows by fixpoint induction.

We have $\mathit{safe}\, \mu_0$ because $\mu_0 Cm(h, \eta)Q$ is $\perp$.

Suppose $\mathit{safe}\, \mu_i$, to show $\mathit{safe}\, \mu_{i+1}$. We must show the safety property of $\mu_{i+1}Cm$ for each C, m and each $\overline{\kappa} \xrightarrow{P;\kappa_3} \kappa_4 \in \mathit{smtypes}(m, C)$. There are two cases, depending on whether m is declared or inherited.

Suppose m has declaration $M = T\ m(\overline{T}\ \overline{x})\{S\}$ in C. Let $\Delta = \overline{x} : (\overline{T}, \overline{\kappa}), \mathit{self} : (C, \mathit{level}\, C), \mathit{result} : (T, \kappa_4)$.

By Lemmas 6.3 and 6.5 we have $ok\, \mu_i$ and $Hconf\ \mu_i$. Assume that $ok\, h$, $ok\, h'$, $ok\, \Delta\, \eta$ and $ok\, \Delta\, \eta'$. Suppose also that $h \sim h'$, $\eta \sim_\Delta \eta'$, $Q \# P$ 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 let $(h_0, \eta_0) = [\![(\overline{x} : \overline{T}, \mathit{self} : C, \mathit{result} : T) \vdash S^\dagger]\!]\mu_i(h, \eta)Q_1$ (if the outcome is $\perp$ there is nothing more to prove). Now $Q_1 \# P$ so by Lemma 6.2, L-confinement of commands, we have $ok\, \Delta\, \eta_0$, $ok\, \Delta\, \eta_0'$, $ok\, h_0$, and $ok\, h_0'$. And, by Lemma 7.2, safety for commands, we have $h_0 \sim h_0'$ and $\eta_0 \sim_\Delta \eta_0'$. It remains to show that if the result level κ_4 for m is L we have $d = d'$, where d (respectively d') is $\eta_0\ \mathit{result}$ (resp. $\eta_0'\ \mathit{result}$). This follows from $\eta_0 \sim_\Delta \eta_0'$ by definition of $\sim_\Delta$ and $\Delta\ \mathit{result} = (T, \kappa_4) = (T, L)$. This concludes the proof of safety of $\mu_{i+1}Cm$ for m declared in C.

Suppose m is inherited in C from superclass B. Towards proving safety of $\mu_{j+1}Cm$, let $\Delta_C = \overline{x} : (\overline{T}, \overline{\kappa}), \mathit{self} : (C, \mathit{level}\, C)$ and $\Delta_B = \overline{x} : (\overline{T}, \overline{\kappa}), \mathit{self} : (B, \mathit{level}\, B)$. Suppose $Q \# P$, $ok\, h$, $ok\, h'$, $ok\, \Delta_C\, \eta$ and $ok\, \Delta_C\, \eta'$. Suppose also that $h \sim h'$ and $\eta \sim_{\Delta_C} \eta'$. We claim that $ok\, \Delta_B\, \eta$, $ok\, \Delta_B\, \eta'$, and $\eta \sim_{\Delta_B} \eta'$. Then, because $\mathit{smtypes}\, C = \mathit{smtypes}\, B$ (by Definition 2 annotated class table), the safety property for $\mu_{j+1}Cm$ follows from the same for $\mu_{j+1}Bm$ with respect to $\overline{\kappa} \xrightarrow{P;\kappa_3} \kappa_4 \in \mathit{smtypes}(m, C)$.

Note that the last step goes through because the safety property (Definition 6) 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 the safety property for $\mu_{j+1}Bm$ to prove it for $\mu_{j+1}Cm$.

For the claim, we only need to consider *self*, as otherwise Δ_C and Δ_B are the same. For *self*, $ok\, \Delta_B\, \eta$ requires $\mathit{level}\, B = L \Rightarrow \eta\, \mathit{self} \in LLoc$. From $C \leq B$ we get $\mathit{level}\, B \leq \mathit{level}\, C$ by the typing rule for classes. Moreover, since m is inherited from C the rule requires $\mathit{level}\, B = \mathit{level}\, C$ so we are done. □

8. Discussion

We have given a static analysis for secure information flow that accounts for calls to the trusted computing base that can be made by both trusted and untrusted callers. Our analysis allows correct use of access control to ensure that confidential information is returned only if the caller has been given access. This improves on previous static analyses, including the predecessor paper [4], where a system call is given a fixed security level.

Our analysis is justified by a noninterference result. This 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.

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.

Stoughton [23] 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 [20] 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 [12]. 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 [17] and that of Hennessy and Riely [13].

Skalka, Smith and Pottier [22, 19] give a static analysis for access checks that never fail, which could serve as basis for program optimizations, and such optimizations are explored by Fournet and Gordon [8].

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 [17, 18]. For the language of [4] without access control, our student Sun Qi has developed and implemented an inference algorithm and work is currently underway to incorporate level polymorphism and modular specification of libraries.

Modularity issues are addressed in the Jif system [16], which also incorporates the selective declassification mechanism of Myers and Liskov [15]. This uses dynamically changing permissions; our approach may offer a way to formalize the security goals achieved by their mechanisms.

Acknowledgement: We are grateful for helpful feedback from anonymous referees and from audiences at Microsoft Research and the Cornell IAI. Thanks especially to Manuel Fähndrich for insisting that negative information could be used in type annotations and to David Wagner for suggesting a succinct description of our proposal.

References

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

[2] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. Revised and extended from [3]; submitted., 2002.

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

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

[5] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, MITRE Corp., 1973.

[6] D. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.

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

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

[9] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[10] L. Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.

[11] J. Gough. *Compiling for the .NET Common Language Runtime*. Prentice Hall, 2001.

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

[13] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Trans. Prog. Lang. Syst.*, 24(5):566–591, 2002.

[14] H. Mantel and A. Sabelfeld. A generic approach to the security of multi-threaded programs. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 126–142, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society Press.

[15] A. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *IEEE Symposium on Security and Privacy*, pages 186–197, 1998.

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

[17] F. Pottier and S. Conchon. Information flow inference for free. In *Proceedings of the fifth ACM International Conference on Functional Programming*, pages 46–57, 2000.

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

[19] F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. In D. Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *Lecture Notes in Computer Science*, pages 30–45. Springer Verlag, Apr. 2001.

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

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

[22] C. Skalka and S. Smith. Static enforcement of security with types. In *Proceedings of the fifth ACM International Conference on Functional Programming*, pages 34–45, 2000.

[23] A. Stoughton. Access flows: A protection model which integrates access control and information flow. In *IEEE Symposium on Security and Privacy*, pages 9–18, 1981.

[24] D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of TAPSOFT'97*, number 1214 in Lecture Notes in Computer Science, pages 607–621. Springer-Verlag, 1997.

[25] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.