History-based Access Control and Secure Information Flow

Anindya Banerjee$^{\star 1}$ and David A. Naumann$^{\star\star 2}$

[1] Department of Computing and Information Sciences
Kansas State University, Manhattan KS 66506 USA
ab@cis.ksu.edu
[2] Department of Computer Science
Stevens Institute of Technology, Hoboken NJ 07030 USA
naumann@cs.stevens-tech.edu

Abstract. This paper addresses the problem of static checking of programs to ensure that they satisfy confidentiality policies in the presence of dynamic access control in the form of Abadi and Fournet's history-based access control mechanism. The Java virtual machine's permission-based stack inspection mechanism provides dynamic access control and is useful in protecting trusted callees from untrusted callers. In contrast, history-based access control provides a stateful view of permissions: permissions after execution are *at most* the permissions before execution. This allows protection of both callers and callees.
The main contributions of this paper are to provide a semantics for history-based access control and a static analysis for confidentiality that takes history-based access control into account. The static analysis is a type and effects analysis where the chief novelty is the use of security types dependent on permission state. We also show that in contrast to stack inspection, confidential information can be leaked by the history-based access control mechanism itself. The analysis ensures a noninterference property formalizing confidentiality.

1 Introduction

Since Denning and Denning's early work on static certification of secure information flow [6], there have been several advances in specifying static analyses; these advances have been comprehensively summarized in Sabelfeld and Myers's survey [16]. Many of these analyses are given in the style of a security type system that is shown to enforce a *noninterference* property [8]. Noninterference is expressed in terms of a lattice, with $L \leq H$ the canonical example: the property says that output channels labeled L are not influenced by input channels labeled H. The mnemonic is H for high security and L for low: with this interpretation,

* Supported in part by NSF grants CCR-0209205 and CCR-0296182.

** Supported in part by NSF grant CCR-0208984 and New Jersey Commission on Science and Technology.

noninterference says that public outputs do not reveal secret inputs. But noninterference also formalizes integrity.[1] Security type systems use labels not only for external channels but also for program variables and other internal channels, in order to impose restrictions such as absence of assignment from an H variable to a L one.

Despite the advances, security type systems have not seen much use as noninterference is difficult to achieve in practice for various reasons, e.g., covert channels and declassification. One way to introduce flexibility is to consider security type systems for information flow that take access control into account. As Sabelfeld and Myers note, access control mechanisms, by themselves, control the release of information but not the flow of information once access has been granted [16]. In previous work [4, 2], we studied the access control mechanism of Java [9] and the .NET CLR [10], called stack inspection, and established a connection between authorization of information access and the subsequent flow of the information. With respect to security type systems, the chief technical novelty was the use of a *permission-dependent* security type system and the formalization of noninterference for such a type system. Permissions are typically used to license sensitive operations. Permission-dependent types can express, for example, that to obtain a secret by reading a confidential file a certain permission is required; moreover the read operation yields no secret if the permission is absent. Assumptions about permissions are used in the typing system to allow certain subprograms to be ignored, namely in branches conditioned on permission tests known never to succeed.

In simple terms, the noninterference property guarantees that the access control mechanism is serving correctly to enforce flow policy, in the sense that once access has been granted, there is no subsequent leak of secret information.

This paper continues the investigations of our previous work [4, 2] but considers *history-based* access control proposed by Abadi and Fournet [1]. Stack inspection is designed for extensible systems, where computation proceeds with trusted and untrusted code calling each other. The stack inspection mechanism is useful in providing protection to trusted code when it is called by untrusted code; untrusted code can execute the trusted code with reduced powers – namely, with permissions common to both. This provides protection because the untrusted code is prevented from using trusted code as deputy and executing sensitive operations. However, as Abadi and Fournet note, stack inspection is not useful in providing protection to the *caller*. Thus, if trusted code calls untrusted code and proceeds with the result returned by the latter – using the *same permissions* as it used for the call – undesired results may occur: in proceeding with the result returned by untrusted code, stack inspection forgets that security may depend on

[1] Whereas confidentiality is about what information is leaked, integrity is about what information is corrupted: highly trusted data should not be influenced by untrusted inputs. If we use the same lattice as for confidentiality then integrity is dual to confidentiality. But if we read H as "hacked" and L as "licensed" then to check integrity is to check that H does not influence L. So for simplicity we confine attention to confidentiality.

how, i.e., with what permissions, the result was computed in the first place. This is because the stack frame containing the permissions is popped upon return, so the permissions are no longer available on the stack.

To illustrate the problem, we recall the central example from Abadi and Fournet's paper. The Main method of a trusted class NaiveProgram, with permission *FileIO*, among others, calls method TempFile of an untrusted class, BadPlugIn, whose permissions do not include *FileIO*. The method happens to return a sensitive document. Next, the Delete method of class File is called, with the sensitive document as argument. Class File has all permissions, and method Delete first checks whether *FileIO* is present in the currently enabled permissions; if so, the document is deleted, otherwise, the method aborts.

In a stack inspection régime, the call NaiveProgram.Main results in deletion of the sensitive document. This is because Delete's code is executed with permissions common to NaiveProgram and File, so the check for *FileIO* succeeds.

In a history-based régime, the document survives, because we track permission state both before and after the call. Where stack inspection is functional, in the sense that the security context is passed as an argument to sub-commands including method invocations, the history based mechanism is imperative, in the sense that the security context is a (special) state variable. The call to TempFile returns the document together with permissions in the intersection of NaiveProgram and BadPlugIn, so that *FileIO* is excluded. Effectively, the history of how the return result was created is recorded. Now the call to Delete takes place with this reduced set of permissions; the check fails.

The contributions of this paper are to formalize the informal development of common programming patterns of history-based access control in [1] and to provide a type-based analysis for secure information flow that takes history-based access control based on such patterns into account. Some familiarity with our previous work [3, 4, 2] will be helpful; in fact, readers familiar with our previous work will readily observe the substantial overlap with that work. The modest variation in this work is that the type-based analysis rules also involve an effect analysis; this effect analysis tracks not only assumptions about the initial permission state, as in stack inspection, but also provides a conservative approximation of the final permission state. A second difference is that, in contrast to the stack inspection mechanism, the history-based mechanism is itself a covert channel subject to nontrivial attacks. To account for such flows, our rule for method calls in high contexts requires that the invoked method has certain permissions; by contrast, the other rules require absence of permissions in certain contexts.

One of our goals is to demonstrate the flexibility of the framework described in our previous work by showing how to handle a different access control mechanism. To reinforce the similarity, the technical development in this paper is structured as in [2]. Proof cases are omitted since they are easily adapted from corresponding ones in [2].

In passing, we note that different access control mechanisms are possible. Our previous work on stack inspection is motivated in part because it is widely deployed. The limitations of stack inspection with respect to method calls were explained by Abadi and Fournet [1] who proposed history-based access control as a way out. Their mechanism is designed to be similar to stack inspection which increases its potential for use in practice and thus it merits study. It would also be interesting to seek a more general notion of access control which subsumes these mechanisms and others. There may be some interesting parallels between such mechanisms and recent work on resource usage analysis, history effects, etc. [12, 14, 13].

The rest of the paper. Section 2 introduces code-based access control via an example and discusses stack-based and history-based access control. It also explains the security type formalism used in type-based information flow analyses. Section 3 is the key section of the paper. It shows how access control can be used to provide more fine-grained confidentiality policies. It also shows how, in contrast to stack inspection, the history-based access control mechanism can itself be employed to leak secrets. Section 4 formalizes the syntax and semantics of the object-oriented language we study. Section 5 gives a type-based static analysis that enforces confidentiality. Section 6 shows the analysis at work on the central example in Abadi and Fournet's paper [1]. Section 7 and Section 8 give the technical details of the static analysis; the latter section states the main result that a program deemed safe by the analysis satisfies the noninterference property. Section 9 concludes.

2 Access control and information flow

Access control via stack inspection. In the Java access control mechanism [9], 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.

Permission checks are used to guard sensitive operations. Following previous work [7], we refrain from modeling exceptions and instead 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.

Example. We consider the following example from Section 4.1 of Abadi and Fournet's paper [1], adapted to our notation (e.g., type **unit** for `void`).

```
class BadApplet extends Object { // some permissions but not FileIO
  unit Main() {
    result:= NaiveLibrary.CleanUp(..."password file"...); }}
```

The comment indicates our assumption about the static access policy: BadApplet does not have permission *FileIO*, whereas NaiveLibrary and File below are trusted classes with all permissions.

```
class NaiveLibrary extends Object { // all permissions
  unit CleanUp((string, L) s) {
    File.Delete(s);}}

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

The call, BadApplet.Main() will abort. The call to NaiveLibrary.CleanUp will occur in a context with permissions common to NaiveLibrary and BadApplet and this context does not contain *FileIO*. As CleanUp calls Delete, the body of Delete will also be executed with permissions common to the current permissions (without *FileIO*) and Delete; hence the test in Delete fails – the password file survives. This is a situation where stack inspection is satisfactory, and the history based mechanism works the same way.

History based access control. At runtime, both stack inspection and the history based mechanism involve a set of currently enabled permissions; it is a subset of the statically authorized permissions of the class of the currently-executing code. When a method is invoked, the initial permission set for the method body is the intersection of the caller's current set and the static permissions of the called code. (Note that it is the class of the dynamically dispatched code that matters, not the class of the target object.) In stack inspection, the method call has no effect on the current permission set of the caller. In the history based mechanism, the caller's permissions become the intersection of their initial value with the final permissions of the called method.

Both mechanisms include means to test and branch on the currently enabled permissions, which we model with the **test** construct.

Permissions P get enabled by the construct **enable** P **in** S in stack inspection and a similar construct **grant** P **in** S for the history-based mechanism; in both cases, what gets added to the current permission set is the intersection of P with the statically authorized permissions, $Auth\, C$, of the current code's class, C.

Whereas the permission set on termination of **enable** P **in** S is the same as its initial value, the history based construct **grant** P **in** S deals with the final permissions of S. Specifically, the final permissions of **grant** P **in** S are the intersection of its initial permissions and the final ones of S. Together with the behavior of method calls described above, this ensures that for any command, the final permission set is a subset of its initial value (see Lemma 1). (In stack inspection the final permissions are equal to the initial ones, so permission passing can be modeled as just a parameter in the semantics.)

The history based mechanism needs a construct, **accept**, to allow a privileged caller to retain its permissions after calling less privileged code (e.g., to delete a file named by an untrusted applet, after checking that the named file is not important). The effect of **accept** P **in** S is to execute S with the current permissions Q. This results in a final permission set Q' from S, which may be a proper subset of Q. The permission set after **accept** P **in** S becomes $Q' \cup (Q \cap P)$.

Note that constructs **grant** and **accept** abstract two useful programming patterns for modifying permissions in code. Abadi and Fournet show how they can be implemented using low-level constructs [1]. For purposes of formal analysis, however, we will stick to **grant** and **accept** in the sequel.

Checking information flow using security types. Based on earlier work on static certification of information flow by Denning and Denning [6], the idea developed by Volpano *et al.* [18] 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 [15, 3]. In [3], 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 assignments and newly allocated objects). Annotated arrow types are used for modular checking in the case of methods (or procedures or functions [11]): the type $(T, \kappa_1) \text{−}\langle\kappa_2\rangle\!\!\rightarrow (U, \kappa_3)$ designates an assumed input level at most κ_1; on this assumption, the heap effect (min level of fields written) is at least κ_2 and result level at most κ_3. A method body is checked with respect to its type, which is used as an assumption for checking method calls. As with ordinary types in Java-like languages, the same type is used for all overriding declarations.

3 Using access control for confidentiality

As mentioned in the introduction, a primary aim of our work is the static checking of method bodies to ensure that they satisfy confidentiality policies. However, we also want our confidentiality policies to be flexible enough to admit a large number of programs. We allow a method to be given several types, to allow different information flow policies to be imposed for callers with different permissions. We explain the idea with the motivating example from our previous work [4, 2].

Consider a trusted class Kern, having permissions *stat* and *sys*, and with a method getStatus that can be called in more than one context. If called by untrusted code, getStatus returns public (L) information. Trusted code, however, can obtain private (H) information.

```
class Kern extends Object {
  String Hinfo; // H
  String Linfo; // L
  String getHinfo() { // type () → H
    test sys then result := self.Hinfo else abort }
  String getStatus() { // types () −⟨∅⟩→ H and () −⟨{stat}⟩→ L
    test stat
    then enable sys in result := self.getHinfo()
    else result := self.Linfo }
... "other methods that manipulate Linfo and Hinfo"}
```

Method getHinfo is useful only to callers with permission sys. Because the security type of self.Hinfo is H, it can only be assigned to a H variable. In this case, the body of getHinfo, it is the special variable, result, that gives the method result. So, for the policy expressed by the type $() \rightarrow H$, the code can be accepted under a Smith-Volpano style analysis [18].

For getStatus, the Smith-Volpano analysis will also insist that result be typed H, so that it satisfies the policy is $() \rightarrow H$. But getStatus is useful both for callers with permission $stat$ and for those without; only the former obtain H info. So we parameterize the policy by permission $stat$: if called in a context that does *not* have permission $stat$, getStatus returns L; otherwise it returns H. The use of negative permissions allows types that mention only permissions relevant to implementations of the method.

More formally, method types in [4, 2] have the form

$$\kappa_0, \bar{\kappa} \text{-}\langle P; \kappa \rangle\!\!\rightarrow \kappa_1 \tag{1}$$

This means: suppose the level of self is at most κ_0, and the level of the parameters of the method have level at most $\bar{\kappa}$, and the method is called in a context with permissions disjoint from P. Then the result has level at most κ_1, and fields written have level at least κ.

Using this notation, getStatus can be assigned both types $L, () \text{-}\langle \{stat\}; H \rangle\!\!\rightarrow L$ and $L, () \text{-}\langle \varnothing; H \rangle\!\!\rightarrow L$.

In the BadApplet example in section 2, Delete can be assigned both types $L, L \text{-}\langle \{FileIO\}; H \rangle\!\!\rightarrow ()$ and $L, L \text{-}\langle \varnothing; H \rangle\!\!\rightarrow ()$. The body of BadApplet.Main needs to be typechecked in a context where $FileIO$ is absent. Hence BadApplet.Main can be assigned the type $L, () \text{-}\langle \{FileIO\}; H \rangle\!\!\rightarrow ()$.

A method body must be checked against each of its declared types. A particular type gives an assumption that certain permissions are absent, and under this assumption certain branches are known not to be taken. Typically the ignored branches involve H assignments and thus ignoring these branches allows the result to be considered L.

History based. In (1), the levels $\kappa_0, \bar{\kappa}$ can be read as the input channel and κ_1 as the output level. For a sound static analysis, the type also tracks the write effect κ and the corresponding judgement for commands gives a command level.

In the sequel we carry out this same idea for the history based mechanism, for which we have to track the update effect on permissions. Thus method types and command judgements include an upper bound on the final permissions; this is made precise in section 5.

Unlike stack inspection, the history based access mechanism introduces a covert channel. The reason is that permission changes made in the context of a high conditional can persist outside the scope of that conditional. If some command S changes the final permission state then it can be used to leak information by executing S under a H guard and then, after the conditional, updating L state based on testing whether the permission state is changed. In [2] we point this out in regard to a variation on **enable** that does not have a scoped sub-command in which the permission is enabled.

The history based mechanism does not allow persistent increase of permissions, but it allows persistent decrease. Here is an example attack, using an explicit **grant** to establish some permission which may then be decreased by a suitable method call.

```
grant p // assuming p statically authorized
in
  if H-exp
  then e.m() // assuming p not authorized for m
  else skip;
  test p then L-var := "H-exp is false" else L-var := "H-exp is true"
```

A variation uses the **accept** statement to allow m to be invoked in every case.

```
grant p // assuming p statically authorized
in
  if H-exp
  then e.m() // assuming p not authorized for m
  else accept p in e.m();
  test p then L-var := "H-exp is false" else L-var := "H-exp is true"
```

The security typing rule for method call disallows any writes to L fields by m — this is enforced by requiring that its heap effect be H, as it is called in the scope of a high guard. Because any code is allowed to invoke **grant**, **test**, and **accept**, the (implicit) variable holding the current permissions must be treated as L and thus changes to it too must be disallowed in H conditionals. It turns out that this can be achieved in the static analysis by requiring that permissions that can be enabled at the site of the call must be included in the static permissions of all classes that provide an implementation of m. For a method call in a H command, any implementation that could be dispatched to at that method call site should be *at least as trusted as* the caller. (This condition has no counterpart in [2].)

Just as a method type serves to specify all implementations thereof, one can imagine stipulating the permissions that all implementations are required to have (a link-time requirement). But in this paper we express the restriction explicitly in the call rule.

T	$::= \mathbf{bool} \mid \mathbf{unit} \mid C$	data type; C ranges over class names
CL	$::= \mathbf{class}\ C\ \mathbf{extends}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \}$	public fields $\bar{f}$, public methods $\bar{M}$
M	$::= T\ m(\bar{T}\ \bar{x})\ \{S\}$	method; result type T, param. types $\bar{T}$
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
	$\mid\ \mathbf{grant}\ P\ \mathbf{in}\ S$	enable permissions
	$\mid\ \mathbf{accept}\ P\ \mathbf{in}\ S$	accept 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

Table 1. Grammar.

4 Language

This section formalizes the sequential class-based language for which our results are given. It is adapted from [2], but with the history-based constructs and with the semantics changed to return the permission state from commands and methods as discussed in Section 2. We assume given a finite set, *Perms*, of permissions. The semantics is given with respect to a given function $Auth : ClassNames \to \mathcal{P}(Perms)$ that specifies access policy. The semantic definitions are independent of any particular information flow policy.

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 $\bar{T}$ 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, Q, R range over sets of permissions.

We include unrestricted recursion but omit loops, super calls, and constructor 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.

The subtyping relation $\leq$ on types is defined as follows. For base types, $\mathbf{bool} \leq \mathbf{bool}$ and $\mathbf{unit} \leq \mathbf{unit}$. For classes C and D, we define $C \leq D$ iff either $C = D$ or the class declaration for C is $\mathbf{class}\ C\ \mathbf{extends}\ B\ \{\ \ldots\ \}$ for some $B \leq D$. The typing rules are syntax-directed: Subsumption is built into the rules

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

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

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

Table 2. Selected typing rules for commands.

rather than appearing as a separate rule, so that the semantics can be defined by recursion on typing derivations.

Some auxiliary notations: let $CT(C) = \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ \bar{T}_1\ \bar{f};\ \bar{M}\ \}$ and let M be in the list $\bar{M}$ of method declarations, with $M = T\ m(\bar{T}_2\ \bar{x})\{S\}$; let $\mathit{mtype}(m, C) = \bar{T}_2 \to T$ record typing information and let $\mathit{pars}(m, C) = \bar{x}$ record the parameter names. Let $\mathit{super}\,C = D$. For fields, we define $\mathit{fields}\,C = \bar{f} : \bar{T}_1 \cup \mathit{fields}\,D$ and assume $\bar{f}$ 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 $\mathit{mtype}(m, C)$ is defined to be $\mathit{mtype}(m, B)$, so that $\mathit{mtype}(m, C)$ 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.

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

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. Typing rules for some commands appear in Table 2. The rule for **accept** is the same as for **grant**. For other elided rules we refer the reader to our earlier papers [4, 2].

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, a *store* η, which assigns locations and primitive values to local variables and parameters, and the currently enabled *permissions* P_1. Every store of interest includes the distinguished variable self which points to the target object. A command denotes a function from initial

$\theta ::= T$	values of type T
$\mid \mathit{\Gamma}$	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 loc.
$\mid \mathit{Heap} \otimes \mathit{\Gamma} \otimes \mathcal{P}(\mathit{Perms})$	(global) states with no dangling locations
$\mid \mathit{Heap} \otimes T \otimes \mathcal{P}(\mathit{Perms})$	triples (h, d, P) where value d is not a dangling loc. w.r.t. h
$\mid \theta_\perp$	lifting
$\mid \mathit{perms}\, C$	permission sets authorized for C
$\mid (C,\, \bar{x},\, \bar{T} {\to} T)$	method of C with parameters $\bar{x} : \bar{T}$ and return type T
$\mid \mathit{MEnv}$	method environments

$$
\begin{aligned}
&\llbracket \textbf{bool} \rrbracket &&= \{\mathit{true}, \mathit{false}\}\\
&\llbracket \textbf{unit} \rrbracket &&= \{\mathit{it}\}\\
&\llbracket C \rrbracket &&= \{\mathit{nil}\} \cup \{\ell \mid \ell \in \mathit{Loc} \wedge \mathit{type}\, \ell \le C\}\\
&\llbracket \mathit{\Gamma} \rrbracket &&= \{\eta \mid \mathit{dom}\, \eta = \mathit{dom}\, \mathit{\Gamma} \wedge \eta\, \mathsf{self} \neq \mathit{nil} \wedge \forall x \in \mathit{dom}\, \eta \bullet \eta\, x \in \llbracket \mathit{\Gamma}\, x \rrbracket\}\\
&\llbracket \mathit{state}\ C \rrbracket &&= \{s \mid \mathit{dom}\, s = \mathit{dom}(\mathit{fields}\, C) \wedge \forall (f : T) \in \mathit{fields}\, C \bullet s f \in \llbracket T \rrbracket\}\\
&\llbracket \mathit{Heap} \rrbracket &&= \{h \mid \mathit{dom}\, h \subseteq_{\mathit{fin}} \mathit{Loc} \wedge \mathit{closed}\, h \wedge \forall \ell \in \mathit{dom}\, h \bullet h\ell \in \llbracket \mathit{state}\ (\mathit{type}\ \ell) \rrbracket\}\\
& &&\quad \text{where } \mathit{closed}\, h \text{ iff } \mathit{rng}\, s \cap \mathit{Loc} \subseteq \mathit{dom}\, h \text{ for all } s \in \mathit{rng}\, h\\
&\llbracket \mathit{Heap} \otimes \mathit{\Gamma} \otimes \mathcal{P}(\mathit{Perms}) \rrbracket\\
&\quad = \{(h, \eta, P) \mid h \in \llbracket \mathit{Heap} \rrbracket \wedge \eta \in \llbracket \mathit{\Gamma} \rrbracket \wedge P \subseteq \mathit{Perms} \wedge \mathit{rng}\, \eta \cap \mathit{Loc} \subseteq \mathit{dom}\, h\}\\
&\llbracket \mathit{Heap} \otimes T \otimes \mathcal{P}(\mathit{Perms}) \rrbracket\\
&\quad = \{(h, d, P) \mid h \in \llbracket \mathit{Heap} \rrbracket \wedge d \in \llbracket T \rrbracket \wedge P \subseteq \mathit{Perms} \wedge (d \in \mathit{Loc} \Rightarrow d \in \mathit{dom}\, h)\}\\
&\llbracket \theta_\perp \rrbracket &&= \llbracket \theta \rrbracket \cup \perp \quad (\text{where } \perp \text{ is some fresh value not in } \llbracket \theta \rrbracket)\\
&\llbracket \mathit{perms}\ C \rrbracket &&= \{P \mid P \subseteq \mathit{Auth}\, C\}\\
&\llbracket C,\, \bar{x},\, \bar{T} {\to} T \rrbracket &&= \llbracket \mathit{Heap} \otimes (\bar{x} : \bar{T}, \mathsf{self} : C) \otimes \mathcal{P}(\mathit{Perms}) \rrbracket \to \llbracket (\mathit{Heap} \otimes T \otimes \mathcal{P}(\mathit{Perms}))_\perp \rrbracket\\
&\llbracket \mathit{MEnv} \rrbracket &&= \{\mu \mid \forall C, m \bullet \mu C m \text{ is defined iff } \mathit{mtype}(m, C) \text{ is defined,}\\
& &&\qquad \text{and } \mu C m \in \llbracket C, \mathit{pars}(m, C), \mathit{mtype}(m, C) \rrbracket \text{ if } \mu C m \text{ defined }\}
\end{aligned}
$$

Table 3. Semantic domains, for given policy *Auth*. We write *dom* and *rng* for the domain and range of a function.

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 $\mathit{type} : \mathit{Loc} \to \mathit{ClassNames}$ such that for each C there are infinitely many locations ℓ with $\mathit{type}\ \ell = C$. 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. An *allocator* is a location-valued function *fresh* such that $\mathit{type}(\mathit{fresh}(C, h)) = C$ and $\mathit{fresh}(C, h) \notin \mathit{dom}\, h$, for all C, h.

Methods are associated with classes, in a *method environment*. For any data type T, the domain $\llbracket T \rrbracket$ is the set of values of type T. For any typing environment

$\mathit{\Gamma}$, $[\![\mathit{\Gamma}]\!]$ is the set of stores assigning values of appropriate type to the variables in $\mathit{dom}\,\mathit{\Gamma}$. There are several other domains for which there is no corresponding notation in the syntax. Such semantic categories, θ, together with semantic domains, $[\![\theta]\!]$, for each category are defined in Table 3; T and $\mathit{\Gamma}$ are included in θ. 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 $\mathit{Heap} \otimes \mathit{\Gamma} \otimes \mathcal{P}(\mathit{Perms})$ and $\mathit{Heap} \otimes T \otimes \mathcal{P}(\mathit{Perms})$.

For expressions and commands, the semantics is defined only for derivable typing judgements. The meaning of an expression $\mathit{\Gamma} \vdash e : T$ is a function $[\![\mathit{Heap} \otimes \mathit{\Gamma}]\!] \to [\![T_\bot]\!]$ that takes $(h, \eta) \in [\![\mathit{Heap} \otimes \mathit{\Gamma}]\!]$ and returns either a value $d \in [\![T]\!]$, such that $(h, d) \in [\![\mathit{Heap} \otimes T]\!]$, 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 omit the semantics of expressions and refer the reader to our previous work [2].

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

$$[\![\mathit{MEnv}]\!] \to [\![\mathit{Heap} \otimes \mathit{\Gamma} \otimes \mathit{perms}(\mathit{\Gamma}\,\mathsf{self})]\!] \to [\![(\mathit{Heap} \otimes \mathit{\Gamma} \otimes \mathit{perms}(\mathit{\Gamma}\,\mathsf{self}))_\bot]\!] \quad (2)$$

that takes a method environment μ (see below)and a state (h, η, R), where the enabled permissions $R \in \mathit{perms}(\mathit{\Gamma}\mathsf{self})$; it returns a possibly updated state together with the updated permissions, or $\bot$ which indicates divergence or error. In history-based access control, permissions get updated, e.g., in method calls. The semantics of command, in Table 4, is defined by recursion on the typing derivation. A nontrivial proof obligation is that if $\mathit{\Gamma} \vdash S$ is derivable then its semantics satisfies (2), which embodies the important property that if the permissions before execution are included in $\mathit{Auth}(\mathit{\Gamma}\mathsf{self})$, permissions after execution are also included in $\mathit{Auth}(\mathit{\Gamma}\mathsf{self})$.

The main distinction between stack inspection and history-based access control is that in the former, permission state (i.e., enabled permissions) after evaluation equals the permission state before evaluation; in the latter, permission state after evaluation is *at most* the permission state before evaluation. Accordingly, the semantics of commands satisfies the following property, shown by a structural induction on commands.

Lemma 1. *Suppose* $(h_0, \eta_0, Q_0) = [\![\mathit{\Gamma} \vdash S]\!]\mu(h, \eta, R)$. *Then* $Q_0 \subseteq R$.

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 $[\![C, \bar{x}, \bar{T} \to T]\!]$, i.e., $[\![\mathit{Heap} \otimes \mathit{\Gamma} \otimes \mathcal{P}(\mathit{Perms})]\!] \to [\![(\mathit{Heap} \otimes T \otimes \mathcal{P}(\mathit{Perms}))_\bot]\!]$ where T is the return type and $\mathit{\Gamma} = \mathsf{self} : C, \bar{x} : \bar{T}$ is the parameter store with $\bar{x} = \mathit{pars}(m, C)$. The result from a method, if not $\bot$, is a triple (h, d, Q) with d in $[\![T]\!]$ and Q in $\mathcal{P}(\mathit{Perms})$ such that, if d is a location then d is in the domain of h.

For a method declaration $M = T\ m(\bar{T}\ \bar{x})\{S\}$ in class C, define

$$\begin{array}{l} [\![M]\!]\mu(h, \eta, R) \\ \quad = \mathsf{let}\ R' = R \cap \mathit{Auth}\,C\ \mathsf{in}\ \mathsf{let}\ \eta_1 = [\eta \mid \mathsf{result} \mapsto \mathit{default}]\ \mathsf{in} \\ \qquad \mathsf{let}\ (h_0, \eta_0, Q_0) = [\![\bar{x} : \bar{T}, \mathsf{self} : C, \mathsf{result} : T \vdash S]\!]\mu(h, \eta_1, R')\ \mathsf{in}\ (h_0,\ \eta_0\,\mathsf{result}, Q_0) \end{array}$$

$$
\begin{array}{l}
[\![\mathit{\Gamma} \vdash e_1.f := e_2]\!]\mu(h,\eta,R) \\
\quad = \mathsf{let}\ \ell = [\![\mathit{\Gamma} \vdash e_1 : C]\!](h,\eta)\ \mathsf{in} \\
\qquad \mathsf{if}\ \ell = \mathit{nil}\ \mathsf{then}\ \bot\ \mathsf{else\ let}\ d = [\![\mathit{\Gamma} \vdash e_2 : U]\!](h,\eta)\ \mathsf{in}\ ([h \mid \ell \mapsto [h\,\ell \mid f \mapsto d]], \eta, R) \\
[\![\mathit{\Gamma} \vdash x := e.m(\bar{e})]\!]\mu(h,\eta,R) \\
\quad = \mathsf{let}\ \ell = [\![\mathit{\Gamma} \vdash e : D]\!](h,\eta)\ \mathsf{in\ if}\ \ell = \mathit{nil}\ \mathsf{then}\ \bot\ \mathsf{else\ let}\ \bar{x} = \mathit{pars}(m,D)\ \mathsf{in} \\
\qquad \mathsf{let}\ \bar{d} = [\![\mathit{\Gamma} \vdash \bar{e} : \bar{U}]\!](h,\eta)\ \mathsf{in\ let}\ \eta_1 = [\bar{x} \mapsto \bar{d}, \mathsf{self} \mapsto \ell]\ \mathsf{in} \\
\qquad \mathsf{let}\ (h_0, d_0, Q_0) = \mu(\mathit{type}\ \ell)m(h,\eta_1,R)\ \mathsf{in}\ (h_0, [\eta \mid x \mapsto d_0], R \cap Q_0) \\
[\![\mathit{\Gamma} \vdash \mathbf{grant}\ P'\ \mathbf{in}\ S]\!]\mu(h,\eta,R) \\
\quad = \mathsf{let}\ (h_0,\eta_0,Q_0) = [\![\mathit{\Gamma} \vdash S]\!]\mu(h,\eta,R \cup (P' \cap \mathit{Auth}(\mathit{\Gamma}\,\mathsf{self})))\ \mathsf{in}\ (h_0,\eta_0,R \cap Q_0) \\
[\![\mathit{\Gamma} \vdash \mathbf{accept}\ P'\ \mathbf{in}\ S]\!]\mu(h,\eta,R) \\
\quad = \mathsf{let}\ (h_0,\eta_0,Q_0) = [\![\mathit{\Gamma} \vdash S]\!]\mu(h,\eta,R)\ \mathsf{in}\ (h_0,\eta_0,Q_0 \cup (P' \cap R \cap \mathit{Auth}(\mathit{\Gamma}\,\mathsf{self}))) \\
[\![\mathit{\Gamma} \vdash \mathbf{test}\ P\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2]\!]\mu(h,\eta,R) \\
= \quad \mathsf{if}\ P \subseteq R\ \mathsf{then}\ [\![\mathit{\Gamma} \vdash S_1]\!]\mu(h,\eta,R)\ \mathsf{else}\ [\![\mathit{\Gamma} \vdash S_2]\!]\mu(h,\eta,R)
\end{array}
$$

Table 4. Semantics of selected commands, for given policy *Auth* and allocator *fresh*. The metalanguage construct, let $d = E_1$ in E_2, has 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 or extension is written, e.g., $[\eta \mid x \mapsto d]$.

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 $\mu \in \mathbb{N} \to [\![\mathit{MEnv}]\!]$ is defined as follows.

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

5 Safety

The syntactic property given by static analysis is called *safety.* The analysis is specified by a typing system.

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.

$$
CL ::= \mathbf{class}\ C\ \mathbf{extends}\ C\ \{\ \bar{\tau}\ \bar{f};\ \bar{M}\ \} \qquad\qquad S ::= \ \ldots \mid \tau\ x := e\ \mathbf{in}\ S \mid \ldots
$$

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 $\kappa, \bar{\kappa} \!-\!\langle P; \kappa_1; Q\rangle\!\!\rightarrow \kappa_2$. 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 $\geq \kappa_1$ and the result level $\leq \kappa_2$ and the enabled permissions after the call are disjoint from Q.

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

Definition 1 (subtyping). $\kappa, \bar{\kappa} \!-\!\langle P; \kappa_1; Q\rangle\!\!\rightarrow \kappa_2 \quad\leq\quad \kappa', \bar{\kappa}' \!-\!\langle P'; \kappa_1'; Q'\rangle\!\!\rightarrow \kappa_2'$ *iff* $\kappa' \leq \kappa$, $\bar{\kappa}' \leq \bar{\kappa}$, $P \subseteq P'$, $\kappa_1' \leq \kappa_1$, $Q' \subseteq Q$, *and* $\kappa_2 \leq \kappa_2'$. □

Note that P, Q are interpreted negatively, so the conditions $P \subseteq P'$ and $Q' \subseteq Q$ are effectively contravariant and covariant respectively.

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, smtypes*(m, C) *is defined iff mtype*(m, C) *is defined. Second, if smtypes*(m, C) *is defined then it is a* non-empty *set of annotations of the form* $\kappa, \bar{\kappa} \!-\!\langle P; \kappa_1; Q\rangle\!\!\rightarrow \kappa_2$. *Third, if* $C \leq D$ *and mtype*(m, D) *is defined then smtypes*$(m, C) =$ *smtypes*(m, D). □

Note that we do not require $P \subseteq \mathit{Auth}\,C$ or $Q \subseteq \mathit{Auth}\,C$. 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.

Definition 3 (safe class table and method declaration). *An annotated class table CT is* safe *provided that each class satisfies the rule*

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

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

$$\frac{\begin{array}{c} \mathit{mtype}(m, C) = \bar{T} \rightarrow T \qquad \mathit{pars}(m, C) = \bar{x} \\ \textsf{\textit{self}}\colon (C, \kappa_0), \bar{x} : (\bar{T}, \bar{\kappa}), \textsf{\textit{result}}\colon (T, \kappa_4); (P \cap \mathit{Auth}\,C) \vdash S : (\mathit{com}\ L, \kappa_3); (Q \cap \mathit{Auth}\,C) \\ \textit{for each}\ (\kappa_0, \bar{\kappa} \!-\!\langle P; \kappa_3; Q\rangle\!\!\rightarrow \kappa_4) \in \mathit{smtypes}(m, C) \end{array}}{C\ \textbf{\textit{extends}}\ D \vdash T\ m(\bar{T}\ \bar{x})\{S\}}$$

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

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

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

$$\frac{\begin{array}{c}\Delta, x\!:\!(T,\kappa) \vdash e\!:\!(D,\kappa_0) \qquad \mathit{mtype}(m,D) = \bar{T} \to T' \qquad \Delta, x\!:\!(T,\kappa) \vdash \bar{e}\!:\!(\bar{U},\bar{\kappa}) \\ \bar{U} \leq \bar{T} \qquad x \neq \mathsf{self} \qquad T' \leq T \qquad \kappa_0', \bar{\kappa}' \!-\!\langle P'; \kappa_1'; Q' \rangle\!\!\rightarrowtail \kappa' \in \mathit{smtypes}(m,D) \\ \kappa_0', \bar{\kappa}' \!-\!\langle P'; \kappa_1'; Q' \rangle\!\!\rightarrowtail \kappa' \leq \kappa_0, \bar{\kappa} \!-\!\langle P'; \kappa_1; Q' \rangle\!\!\rightarrowtail \kappa \\ P' \cap \mathit{Auth}(\Delta^{\dagger}\mathsf{self}) \subseteq P \qquad Q \subseteq Q' \cap \mathit{Auth}(\Delta^{\dagger}\mathsf{self}) \\ \kappa_0 \leq \kappa \sqcap \kappa_1 \qquad \kappa = H \wedge \kappa_1 = H \ \Rightarrow\ (\mathit{Auth}(\Delta^{\dagger}\mathsf{self}) - P) \subseteq (\cap_{E \leq D} \mathit{Auth}\, E)\end{array}}{\Delta, x\!:\!(T,\kappa); P \vdash x := e.m(\bar{e}) : (\mathit{com}\ \kappa, \kappa_1); Q}$$

$$\frac{\Delta; P \vdash S_1 : (\mathit{com}\ \kappa_1, \kappa_2); Q_1 \qquad \Delta; Q_1 \vdash S_2 : (\mathit{com}\ \kappa_1, \kappa_2); Q}{\Delta; P \vdash S_1;\ S_2 : (\mathit{com}\ \kappa_1, \kappa_2); Q}$$

$$\frac{\begin{array}{c}\Delta \vdash e\!:\!(\mathbf{bool}, \kappa) \\ \Delta; P \vdash S_1 : (\mathit{com}\ \kappa_1, \kappa_2); Q \qquad \Delta; P \vdash S_2 : (\mathit{com}\ \kappa_1, \kappa_2); Q \qquad \kappa \leq \kappa_1 \sqcap \kappa_2\end{array}}{\Delta; P \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 : (\mathit{com}\ \kappa_1, \kappa_2); Q}$$

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

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

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

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

$$\frac{P' \cap P \neq \varnothing \vee P' \not\subseteq \mathit{Auth}(\Delta^{\dagger}\mathsf{self}) \qquad \Delta; P \vdash S_2 : (\mathit{com}\ \kappa_1, \kappa_2); Q}{\Delta; P \vdash \mathbf{test}\ P'\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 : (\mathit{com}\ \kappa_1, \kappa_2); Q}$$

$$\frac{\Delta; P \vdash S : (\mathit{com}\ \kappa_1, \kappa_2); Q \qquad \kappa_3 \leq \kappa_1 \qquad \kappa_4 \leq \kappa_2 \qquad P \subseteq P' \qquad Q' \subseteq Q}{\Delta; P' \vdash S : (\mathit{com}\ \kappa_3, \kappa_4); Q'}$$

Table 5. Security typing rules for commands, for given *Auth*.

This rule depends on rules for expressions and commands. The rules for commands are given in Table 5 but the rules for expressions are exactly the same as the ones in [2] and hence elided. [2] □

In the rules for expressions and commands, we write Δ for typing environments that assign security types. A judgement $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2); Q$ says that if no permissions in set P are enabled initially, then S is safe, assigns only to variables (locals and parameters) of level $\geq \kappa_1$ and to object fields of level $\geq \kappa_2$ (see Lemma 4), and no permissions in set Q are enabled finally, i.e., after execution of S.

The rules use versions of the auxiliary functions that take security levels into account. Let $CT(C) = \textbf{class}\ C\ \textbf{extends}\ D\ \{\ \bar{\tau_1}\ \bar{f};\ \bar{M}\ \}$. Corresponding to *fields*, we define $\mathit{sfields}\, C = \bar{f} : \bar{\tau_1} \cup \mathit{sfields}\, D$.

Judgement $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2); Q$ is derivable provided $P \subseteq \mathit{Auth}(\Delta^\dagger \mathsf{self})$ and $Q \subseteq \mathit{Auth}(\Delta^\dagger \mathsf{self})$ and the judgement is derivable using the security typing rules.

The last rule in Table 5 is a subsumption rule.

The rule for method call is different from our work on stack inspection [2] in that it has an extra condition for high commands: permissions that may be enabled at the site of the call (namely, $\mathit{Auth}(\Delta^\dagger \mathsf{self}) - P$) must be included in the static permissions of all classes that provide an implementation of the method (namely, $\cap_{E \leq D} \mathit{Auth}\, E$). This condition essentially disallows high commands from making calls that may dynamically dispatch to untrusted code. Because permission state is an implicit low variable, and a call to untrusted code causes a loss of permissions, this loss can be tested by a low observer and used to reveal secrets —recall the example attacks in section 2. Those attacks are untypable in our system because they require permission p at the call site for m to be included in the static permissions of m.

Note that the rule for method declaration does not restrict assignments to local variables, i.e., it allows effect L in the hypothesis. Subsumption may be used to get L there from H. The rule for method call has a form of subsumption built in: it requires there to be some declared type for the method that matches its invocation.

The second rule for **test** is the one that removes from consideration a branch that cannot be taken under the assumption: 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.

Properties of security typing. For any judgement $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2); Q$ derivable using the security typing rules for expressions and commands, the

[2] 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 precise formulation [2] uses † but we omit the unilluminating complication here.

erased judgement $\Delta^\dagger \vdash S^\dagger$ is derivable using the ordinary typing rules for commands. Conversely, any program typable using the ordinary typing rules for commands can be annotated everywhere by L and typed by the security typing rules for expressions and commands, taking $\mathit{smtypes}(m, C) = \{L, \bar{L} -\!\!\langle \varnothing; L; \varnothing\rangle\!\!\rightarrow L\}$ for all m, C. In other words, for the trivial security policy encoded by the above security type, the analysis rejects no well formed program.

6 Example

The following example, discussed in the introduction, shows our type system at work; in contrast to the scenario in section 3, where untrusted code calls trusted code, here we consider the dual scenario of trusted code calling untrusted code.

```
class NaiveProgram extends Object {//static permissions contain all permissions
  unit Main() {
    (string, L) s := BadPlugIn.TempFile();
    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 {//static permissions do not contain FileIO
  (string, L) 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 {//static permissions contain all permissions
  unit Delete((string, L) s) {
    test FileIO then Win32.Delete(s) else abort;} }
```

We decorate BadPlugin.TempFile() with the history-based flow policy

$$\mathit{smtypes}(\mathsf{TempFile}, \mathsf{BadPlugin}) = \{L, () -\!\!\langle \varnothing; H; \{FileIO\}\rangle\!\!\rightarrow L\}$$

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

$$\{L, L -\!\!\langle \{\mathit{FileIO}\};\ H\ ; \{\mathit{FileIO}\}\rangle\!\!\rightarrow (), \qquad L, L -\!\!\langle \varnothing;\ H\ ; \varnothing\rangle\!\!\rightarrow ()\}$$

The first is used in a context where *FileIO* is absent and asserts that *FileIO* is absent after the call is finished. Indeed, for this policy, the type system checks the body of Delete in the permission context $\{\mathit{FileIO}\} \cap \mathit{Auth}(File)$, i.e., the context $\{\mathit{FileIO}\}$. To type check the test, note that $\{\mathit{FileIO}\} \cap \{\mathit{FileIO}\} \neq \varnothing$, and Delete is accepted.

If we check NaiveProgram.Main for the policy $L, () -\!\!\langle \varnothing; H; \{FileIO\}\rangle\!\!\rightarrow ()$, we have the following situation: the call to TempFile results in the excluded permission set *FileIO*, which is the excluded permission set for the call to

Delete. We have two possibilities for the type of Delete now, but only the type $L, L{-}\!\langle\{\mathit{FileIO}\}; H; \{\mathit{FileIO}\}\rangle\!\!\rightarrowtail()$ will do: from Table 5, rule for method call, we have to establish $\{\mathit{FileIO}\} \cap \mathit{Auth}(\mathsf{NaiveProgram}) \subseteq \{\mathit{FileIO}\}$ and $\{\mathit{FileIO}\} \subseteq \{\mathit{FileIO}\} \cap \mathit{Auth}(\mathsf{NaiveProgram})$. Both succeed. Hence NaiveProgram.Main is well-typed.

Note that we could not have chosen the type $L, L{-}\!\langle\varnothing; H; \varnothing\rangle\!\!\rightarrowtail()$ as the type of Delete, since we cannot establish $\{\mathit{FileIO}\} \subseteq \{\mathit{FileIO}\} \cap \varnothing$ for the postcondition.

In our previous work [2], on stack inspection, the typechecker rejected NaiveProgram.Main as ill-typed. It is instructive to recall the reason: after the call to TempFile, the call to Delete occurs in the context where *no* permissions are excluded, and then the antecedent $\{\mathit{FileIO}\} \cap \mathit{Auth}(\mathsf{NaiveProgram}) \subseteq \varnothing$ fails.

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 $\mathit{com}\,H, H$ 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 [5].

The formalization uses the indistinguishability relation $\sim$ which is also used to formulate noninterference in Section 8. States (h, η, P) and (h', η', P') 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 $\mathit{dom}\, h$ and $\mathit{dom}\, h'$ that, informally, are or have been visible to L.

Definition 4. *A* typed bijection *is a bijective finite partial function, σ, from Loc to Loc, such that $\mathit{type}(\sigma\,\ell) = \mathit{type}\,\ell$ for all ℓ in $\mathit{dom}\,\sigma$.* □

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 5 (indistinguishable by L). *For any σ, we define relations $\sim_\sigma$ for data values, object states, heaps, and stores.*

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

□

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.

In our model, permissions are atomic values and indistinguishability is just equality for permission sets. In a more detailed model, permissions would be heap objects.

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, e.g., if $h_1 \sim_\sigma h_2$ and $h_2 \sim_\tau h_3$ then $h_1 \sim_{\tau\circ\sigma} h_3$.

One use of $\sim$ is to formulate, in Lemma 4 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 of initially existing objects 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.

The security typing rules depend on permission effects. Thus, we first show some results on the soundness of effects. The intuition is that if a safe command is executed with permissions disjoint from the initially excluded permissions, then the permissions produced as a result of execution are disjoint from the final set of excluded permissions. Formalizing this intuition is the purpose of Definition 6, Lemma 2 and Lemma 3 below.

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

Definition 6 (disjoint effects in method environment). *Method environment μ has* disjoint effects*, written $disj\ \mu$, if the following holds for all C, m and all $\kappa_0, \bar{\kappa}\!-\!\langle P;\kappa;Q\rangle\!\!\rightarrow\!\kappa_1$ in $smtypes(m, C)$. If $R \# P$ and $\mu C m(h, \eta, R) \neq \bot$ then $Q \# Q_0$, where $(h_0, d, Q_0) = \mu C m(h, \eta, R)$.* □

Lemma 2 (disjoint effects in commands). *Suppose $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2); Q$ and $disj\ \mu$. For all η, h, R such that $P \# R$, and $R \subseteq Auth(\Delta^\dagger\ \mathsf{self})$, if $(h_0, \eta_0, Q_0) = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h, \eta, R)$ then $Q \# Q_0$.*

Lemma 3 (safe programs have disjoint effects). *If annotated class table CT is safe then $disj\ [\![CT^\dagger]\!]$ and also $disj\ \mu_i$ for each μ_i in the approximation chain for semantics of CT.*

The purpose of Definition 7, Lemma 4 and Lemma 5 below is to establish that H-commands do not assign to L-variables and L-fields of objects.

Definition 7 (write confined method environment). *Method environment μ is* write confined*, written $wconf\ \mu$, if the following holds for all C, m and all $\kappa, \bar{\kappa}\!-\!\langle P;H;Q\rangle\!\!\rightarrow\!\kappa_1$ in $smtypes(m, C)$. If $R \# P$ and $\mu C m(h, \eta, R) \neq \bot$ then $h \sim_\iota h_0$ where $(h_0, d, Q_0) = \mu C m(h, \eta, R)$ and ι is the identity on $dom\,h$. Moreover, if $\kappa_1 = H$, then $R' = Q_0$ where $R' = R \cap Auth\,C$.* □

Lemma 4 (write confinement of commands). *Suppose disj μ, wconf μ, and $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2); Q$. For all η, h, R such that $P \# R$, and $R \subseteq Auth(\Delta^\dagger\ \mathsf{self})$, if $(h_0, \eta_0, Q_0) = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h, \eta, R)$ then: (i) $\kappa_1 = H$ implies $\eta \sim_\iota \eta_0$; (ii) $\kappa_2 = H$ implies $h \sim_\iota h_0$ and (iii) $\kappa_1 = H$ and $\kappa_2 = H$ imply $R = Q_0$, where ι is the identity on dom h.*

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

Lemma 5 (safe programs are write confined). *If annotated class table CT is safe then wconf $[\![CT^\dagger]\!]$ and also wconf μ_i for each μ_i in the approximation chain for semantics of CT.*

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 6 (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 $d \sim_\sigma d'$.

8 Safety implies noninterference

This section states 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.

Noninterference for a class table is defined in terms of noninterference of method meanings with respect to their security types. Roughly, the idea is that a method executed under related stores, related heaps, and related permission states, yields related heaps. Provided *smtypes* of the method declares that the level of the return result is L, the return results are also related. This idea can be formalized (as in [2]) by saying that a method meaning d *satisfies* a method type $\kappa_0, \bar{\kappa} \text{–}\langle P; \kappa_1; Q\rangle\!\!\rightarrow \kappa_2$ iff the following holds for all $\sigma, h, h', \eta, \eta', P_1, P_1'$: Let $(h_0, d_0, Q_0) = d(h, \eta, P_1)$, and $(h_0', d_0', Q_0') = d(h', \eta', P_1)$. If $h \sim_\sigma h'$, $\eta \sim_\sigma \eta'$, $P_1 \sim_\sigma P_1'$, and $P_1 \# P$ then there is $\tau \supseteq \sigma$ such that $h_0 \sim_\tau h_0'$, $Q \# Q_0$, $Q_0 \sim_\tau Q_0'$, and $(\kappa_2 = L \Rightarrow d_0 \sim_\tau d_0')$

Definition 8 (noninterfering method environment). *A method environment is noninterfering, written nonint μ, iff for all C, m, the meaning $\mu C m$ satisfies every $\kappa_0, \bar{\kappa} \text{–}\langle P; \kappa_1; Q\rangle\!\!\rightarrow \kappa_2$ in smtypes(m, C).* □

Our main result is that the method environment denoted by a safe class table is noninterfering. The proof uses lemmas which express noninterference for the expression and command constructs, respectively.

The proof of the main theorem goes by proving noninterference of each method environment in the approximation chain, using the following.

Lemma 7 (safe commands are noninterfering). *Suppose disj* μ, *wconf* μ, *nonint* μ, *and* $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2); Q$. *Suppose also* $R \# P$, $R \subseteq Auth(\Delta^\dagger\ \mathsf{self})$, $\eta \sim_\sigma \eta'$, $h \sim_\sigma h'$, $R \sim_\sigma R'$, $(h_0, \eta_0, Q_0) = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h, \eta, R)$ *and* $(h'_0, \eta'_0, Q'_0) = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h', \eta', R')$. *Then there is* $\tau \supseteq \sigma$ *such that* $\eta_0 \sim_\tau \eta'_0$ *and* $h_0 \sim_\tau h'_0$ *and* $Q_0 \sim_\tau Q'_0$.

Theorem 1 (safety implies noninterference). *If annotated class table* CT *is safe then its meaning* $[\![CT^\dagger]\!]$ *is noninterfering.*

9 Discussion

We have formalized the history based mechanism of Abadi and Fournet and shown how, by tracking updates of the permission state, static rules can ensure noninterference in programs that depend on access control to prevent illegal information flow. Unlike stack inspection, the mechanism itself introduces a new channel of information flow, but one that can be controlled using the same sort of type-and-effect analysis that we previously developed for stack inspection [2]. For modular (per-method) checking of dynamically dispatched method calls, we rely on a flow policy that specifies a set of types for every method; this flow policy is invariant with respect to subclassing. Whereas these flow types describe flows that occur in the absence of certain permissions, a fully modular system would also specify certain permissions required to be present. This set would be used to check method calls in H commands. In this paper we chose to omit the latter form of interface specification, but the call rule imposes essentially the same condition.

Specifying the analysis in terms of a type system is convenient for proving our noninterference result. But for practical application of the result, type inference is needed to reduce the burden of annotation. We have developed a modular inference algorithm for inferring security types for an object-oriented language without dynamic access control [17]. In future work, we plan to report on security type inference for a language with dynamic access control supporting, e.g., the stack inspection mechanism or the history-based mechanism.

Another practical issue is to provide a syntax for flow policy. By Lemma 1, permission state after execution is at most the permission state before execution. So it is reasonable to expect that in the analysis, the final set of excluded permissions contain the initial set of excluded permissions. Thus in a flow policy, $\kappa_0, \bar{\kappa} \multimap\!\langle P; \kappa; Q\rangle\!\!\rightarrowtail \kappa_1$, Q can contain the permissions not already in P. To translate this into the framework of this paper, Q can be replaced by $P \cup Q$.

References

1. M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, pages 107–121, Feb. 2003.

2. A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming, Special Issue on Language-based Security.* To appear.
3. 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.
4. A. Banerjee and D. A. Naumann. Using access control for secure information flow in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 155–169. IEEE Computer Society Press, 2003.
5. D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, MITRE Corp., 1973.
6. D. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
7. C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. *ACM Trans. Prog. Lang. Syst.*, 25(3):360–399, 2003.
8. 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.
9. L. Gong. *Inside Java 2 Platform Security.* Addison-Wesley, 1999.
10. J. Gough. *Compiling for the .NET Common Language Runtime.* Prentice Hall, 2001.
11. 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.
12. A. Igarashi and N. Kobayashi. Resource Usage Analysis. *ACM Trans. Prog. Lang. Syst.*, 2004. To appear.
13. Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *Proceedings of the the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, Aug. 2003.
14. K. Marriott, P. J. Stuckey, and M. Sulzmann. Resource usage verification. In *Proceedings of the First Asian Programming Languages Symposium (APLAS)*, 2003.
15. A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, 1999.
16. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
17. Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In R. Giacobazzi, editor, *Static Analysis Symposium (SAS)*, volume 3148 of *Lecture Notes in Computer Science*, pages 84–99. Springer-Verlag, 2004.
18. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.