Beyond Stack Inspection:
A Unified Access-Control and Information-Flow Security Model

Marco Pistoia
IBM T. J. Watson Research Ctr.
Hawthorne, New York, USA
pistoia@us.ibm.com

Anindya Banerjee*
Kansas State University
Manhattan, Kansas, USA
ab@cis.ksu.edu

David A. Naumann†
Stevens Institute of Technology
Hoboken, New Jersey, USA
naumann@cs.stevens.edu

Abstract

Modern component-based systems, such as Java and Microsoft .NET Common Language Runtime (CLR), have adopted Stack-Based Access Control (SBAC). Its purpose is to use stack inspection to verify that all the code responsible for a security-sensitive action is sufficiently authorized to perform that action. Previous literature has shown that the security model enforced by SBAC is flawed in that stack inspection may allow unauthorized *code no longer on the stack to influence the execution of security-sensitive code. A different approach, History-Based Access Control (HBAC), is safe but may prevent* authorized *code from executing a security-sensitive operation if less trusted code was previously executed. In this paper, we formally introduce Information-Based Access Control (IBAC), a novel security model that verifies that all and only the code responsible for a security-sensitive operation is sufficiently authorized. Given an access-control policy α, we present a mechanism to extract from it an implicit integrity policy ι, and we prove that IBAC enforces ι. Furthermore, we discuss large-scale application code scenarios to which IBAC can be successfully applied.*

1 Introduction

Modern component-based systems allow code from different sources and with different levels of trust to be executed at the same time. From an integrity point of view, it is important to verify that, any time a security-sensitive action guarded by a permission q is attempted, all the code responsible for that action has been granted q (or a permission stronger than q). Among the solutions that have been adopted to enforce this principle are Stack-Based Access Control (SBAC) and History-Based Access Control (HBAC). This section shows the limitations of SBAC and HBAC, and introduces a new security model, called *Information-Based Access Control* (IBAC), which significantly improves upon SBAC and HBAC. The programs presented in this section are not contrived examples, but represent problems encountered in real production-level Eclipse code [14] in the process of adopting Java's SBAC, as will be discussed in Section 4.2.

1.1 Stack-Based Access Control Systems

When access to a restricted resource is attempted, SBAC systems, such as Java and Microsoft .NET Common Language Runtime (CLR), have a primitive function walk the execution stack and verify that all the callers currently on that stack have been granted the permission q guarding access to that resource [20]. This primitive function is called `checkPermission` in Java and `Demand` in CLR; in this paper, we call it **`test`**. The purpose of the SBAC stack traversal is to prevent Confused Deputy attacks, in which unauthorized code indirectly causes the execution of security-sensitive functions by calling (or being called by) authorized code [24].

Unfortunately, SBAC systems may inadvertently allow unauthorized code to influence the execution of security-sensitive code—an integrity violation. This is illustrated in Figure 1, where a Java program and its corresponding access-control policy are shown. The security-sensitive operation performed by the program in Figure 1 is the construction of a `FileOutputStream` object. The `FileOutputStream.<init>(File, boolean)` constructor triggers a stack walk by calling `checkPermission` on the active `SecurityManager` with a `FilePermission` parameter guarding the `passwords.txt` file against write access, as shown in Figure 2.[1] The purpose of the stack walk is to verify that all

*Supported by National Science Foundation (NSF) grants CCR-0296182, ITR-0326577, and CNS-0627748.

†Supported by NSF grants CCF-0429894 and CNS-0627338.

[1] Only the fragment of the `FileOutputStream` code relevant to the

```
public class A {
  public static void main(String[] args) throws Exception {
    B b = new B();
    String fileName = b.m1();
    FileOutputStream f = new FileOutputStream(fileName);
  }
} public class B {
  public String m1() {
    return "passwords.txt";
  }
}
```

Component	Permission Set
A	$R_1 =$ {`FilePermission "<<ALL FILES>>", "write"`}
B	$R_2 = \varnothing$
System	$R_3 =$ {`AllPermission`}

Figure 1. Classes `A` and `B`, with the Corresponding Access-control Policy

```
public class FileOutputStream {
  private boolean append;
  public FileOutputStream(String name) throws FileNotFoundException {
    File file;
    if (name != null)
      file = new File(name);
    this(file, false);
  }
  public FileOutputStream(File file, boolean append) throws FileNotFoundException {
    String name = (file != null ? file.getPath() : null);
    SecurityManager security = System.getSecurityManager();
    if (security != null)
      security.checkPermission(new FilePermission(name, "write"));
    this.append = append;
    if (append)
      openAppend(name);
    else
      open(name);
  }
  private native void open(String name) throws FileNotFoundException;
  private native void openAppend(String name) throws FileNotFoundException;
}
```

Figure 2. Fragment of System Class `FileOutputStream`

the callers currently on the stack have been granted at least the required `FilePermission "passwords.txt", "write"`.

Since `A` is not responsible for setting the file name, the access-control policy has to conservatively grant it a broad `FilePermission "<<ALL FILES>>", "write"` for the program to execute without run-time authorization failures. Furthermore, the access-control policy grants `AllPermission` to all the system classes, such as `FileOutputStream` and `SecurityManager`.

When the **test** is performed, the callers on the stack, in reverse order, are `security.checkPermission`, `FileOutputStream.<init>(File, boolean)`, `FileOutputStream.<init>(String)`, and `A.main`. The **test** succeeds because these callers have all been granted permissions at least as strong as `FilePermission "passwords.txt", "write"`. The problem is that `B` influences the file-access operation, since the file name is defined by `B`, but `B`'s right to perform that operation is never checked because `b.m1` is not on the stack when the stack walk is performed, and the file access succeeds despite `B`'s having been granted no permissions.

In cases in which trusted code needs to perform a security-sensitive operation—such as reading from a configuration file or writing to a log file—not explicitly requested by its (possibly untrusted) callers, SBAC systems allow marking a block of code as *privilege asserting*. When **test** encounters privilege-asserting code on the stack, it stops stack inspection and does not proceed fur-

examples of Section 1.1 and 1.2 is shown in Figure 2. Moreover, the `FileOutputStream` code has been slightly simplified for expository purposes.

```
public class C {
  public String logFileName = "C:\\log.txt";
  public void m2() throws Exception {
    FileOutputStream f = (FileOutputStream)
          AccessController.doPrivileged(new PrivilegedExceptionAction() {
      public Object run() throws Exception {
        return new FileOutputStream(logFileName);
      }
    });
    PrintStream ps = new PrintStream(f);
    ps.print("Log file started.");
  }
}
```

Component	Permission Set
C	$R'_1 = \{$FilePermission "<<ALL FILES>>", "write"$\}$
Client	$R'_2 = \varnothing$
System	$R'_3 = \{$AllPermission$\}$

Figure 3. Class C with the Corresponding Access-control Policy

ther, thereby temporarily *granting* the asserted permission to the callers of the privilege-asserting code, recursively. The function used to mark code as privilege-asserting is called doPrivileged in Java and Assert in CLR; in this paper we call it **grant**. Privilege-asserting code is particularly vulnerable to this type of integrity violations because attacks can be mounted even by code on the current stack of execution [33].

Consider for example the trusted library method m2 in Figure 3. Here, any untrusted client can instantiate an object c of type C, set the value of c.logFileName to an operating system file or a password file, and cause the contents of that file to be overwritten by calling c.m2. The untrusted client code will be on the stack when f is created, but the presence of doPrivileged will prevent the stack inspection from reaching that client code, which will succeed in altering the contents of any file of the file system.

1.2 History-Based Access Control Systems

HBAC [18, 1] was proposed to alleviate the limitations of SBAC. In HBAC systems, when a security-sensitive resource is accessed, all the code previously executed (and not just the code currently on the stack) must be sufficiently authorized to access that resource, regardless of the fact that some of that code may not be responsible for the resource access.

As an example, the HBAC **test** triggered by the FileOutputStream.<init>(File, boolean) constructor detects the presence of b.m1 in the execution history of the program in Figure 1, and since b.m1 is not sufficiently authorized, the program fails with a SecurityException, which is appropriate because b.m1 influences the values used in the security-sensitive operations initiated by A.main, and B does not have the necessary permission. However, HBAC rejects also the program in Figure 4 because the executions of unauthorized methods G.<init> and g.m3 are in the history, even though that code does not influence the file-access operation initiated by F.main.

The execution of the FileOutputStream constructor in F.main does not depend on the execution of g.m3. Therefore, the order of these two calls in F.main can be safely reversed. In this case, the program will succeed under HBAC. This inconsistent behavior may be a source of confusion for end users, and can render an application unstable; an authorization failure may remain undiscovered until run time if program components are not tested in the particular order that causes that failure.

The **grant** primitive can be used also in HBAC to prevent previously-executed untrusted code from reducing the permissions dynamically granted to a trusted library performing a security-sensitive action. However, previously-executed code, though shielded from the permission requirement by the **grant** call, can still cause an integrity violation by influencing the execution of that action through a *tainted variable*, such as logFileName in Figure 3.

1.3 Contributions

In this paper, we introduce IBAC, a new access-control model that, for any security-sensitive operation, verifies that all the code responsible for that operation is sufficiently authorized. IBAC does not limit an authorization check to the current execution stack, since code responsible for a security-sensitive operation may no longer be on that stack. Therefore, IBAC is more restrictive and more precise than SBAC. On the other hand, when a security-sensitive operation is attempted, IBAC does not restrict the permissions of the program based on the permissions granted to *all* the

```
public class F {
  public static void main(String[] args) throws Exception {
    G g = new G();
    g.m3();
    FileOutputStream f = new FileOutputStream("passwords.txt");
  }
}
public class G {
  public void m3() {
    System.out.println("The program has started");
  }
}
```

Component	Permission Set
F	$R_1'' = \{$`FilePermission "<<ALL FILES>>", "write"`$\}$
G	$R_2'' = \varnothing$
System	$R_3'' = \{$`AllPermission`$\}$

Figure 4. Classes `F` and `G`, with the Corresponding Access-control Policy

code that has ever executed (as in HBAC), but only based on the permissions granted to the code that has effectively influenced the execution of that security-sensitive operation. Therefore, IBAC is less restrictive and more precise than HBAC.

IBAC is based on the concept that every access-control policy α implicitly defines an information-flow policy ι. Specifically, α assigns sets of permissions to a program's components, and identifies which permissions are necessary to execute the program. IBAC uses the permission sets granted by α as the labels for ι. Furthermore, IBAC transforms the existing calls to the access-control **test** primitive, which are already embedded into the program, into information-flow check points. Subsequently, for any security-sensitive operation requiring permission q, ι imposes the integrity property that no code component without permission q be allowed to influence the execution of that security-sensitive operation. To achieve this result, IBAC attaches dynamic labels on values and on the program counter, and augments ordinary stack inspection with tracking of information flows to security-sensitive operations. More specifically, a call to the **test** primitive checks all the callers on the current stack of execution up to the first **grant** call, just like in ordinary stack inspection. However, in addition to that, an IBAC **test** verifies the labels on all the values read in the security-sensitive operation guarded by the **test** call. (One way to carry out this reinterpretation is to retain the standard semantics of **test** and to insert additional checks for the values read, using a new operation for that purpose. This is the approach taken in our formalization.) This way, IBAC prevents untrusted code no longer on the stack from influencing the execution of a security-sensitive operation.

The advantage of IBAC is that it avoids the need for an explicit information-flow policy specification or code annotations: the information-flow policy itself is automatically obtained from the program's access-control policy (which, if absent, can be automatically computed [27, 10]), and its enforcement is automatically derived from the existing calls to the **test** and **grant** primitives. Unlike the Data Mark Machine approach [16], flows that violate integrity are not prevented; rather, they are recorded and later detected by the new IBAC **test** semantics.

This paper formally describes how to define ι from α and proves that the enforcement of ι performed by IBAC guarantees noninterference [12, 13, 19]. Specifically, if a **test** succeeds, which indicates the output of the flow should be trusted, then indeed that output is not influenced by any untrusted code, in the standard sense of noninterference [19].

1.4 Organization

The remainder of this paper is organized as follows: Section 2 defines the syntax and semantics of the language. In Section 3, we prove that a program passing the new IBAC permission **test** is noninterferent in a novel sense. Moreover, we connect that notion to an ordinary notion of noninterference by considering a program that ends by testing the permissions on its trusted outputs. Section 4 revisits the running examples of Sections 1.1 and 1.2, and shows that IBAC properly enforces the information-flow policies extracted from the access-control policies on those examples. Furthermore, Section 4 demonstrates the applicability of IBAC to large-scale application code. Section 5 discusses possible implementations of IBAC. Section 6 presents related work, and Section 7 concludes this paper.

2 Language

To formalize the model, we choose a language similar to that of Fournet and Gordon [18], but with the addition of mutable variables and dynamically allocated mutable

Types	T	$::=$	$\mathbf{int} \mid \mathbf{bool} \mid \{\overline{f} : \overline{T}\}\mathbf{ref}$	primitives; pointer to record with fields $\overline{f}$
Declarations	K	$::=$	$\mathbf{var}\ x : T$	global variable
			$\mathbf{let}\ p(x : T) = R[C]$	procedure with static permissions
Programs	M	$::=$	$\{K;\}^* \ \mathbf{in}\ R[C]$	main program with body $R[C]$ in context of decls.
Commands	C	$::=$	$x := E \mid x.f := E \mid$	assignment to variable and field
			$x := \mathbf{ref}\{\overline{f} = \overline{E}\} \mid$	allocate and initialize variable of record type
			$p(E) \mid C; C \mid \mathbf{if}\ E\ \mathbf{then}\ C\ \mathbf{else}\ C \mid$	procedure call; command sequence; conditional
			$\mathbf{grant}\ R\ \mathbf{in}\ C \mid$	assert dynamic permissions
			$\mathbf{test}\ R\ \mathbf{then}\ C\ \mathbf{else}\ C \mid$	check and branch on stack permissions
			$\mathbf{test}\ R\ \mathbf{for}\ E$	check value permissions
Expressions	E	$::=$	$\mathbf{true} \mid \mathbf{false} \mid n \mid$	boolean and integer literals
			$E \oplus E \mid x \mid x.f \mid \mathbf{null}$	binary operators; variable; field access; null

Figure 5. Language Syntax

records to more closely model conventional object-oriented (OO) languages. For simplicity, subclassing and dynamic dispatch are omitted, and we use explicitly declared procedures, rather than untyped λ terms, in order to treat mutable variables and objects while avoiding semantic complications.

Another feature we adopt from Fournet and Gordon's language is that the static assignment of permissions to code is modeled using *framed expressions* of the form $R[C]$, where C is ordinary code and the *frame*, R, is a set of permissions. Values stored in variables and object fields have the form $R[v]$, where v is a primitive value (an integer, a boolean, a pointer, or *null*); here, R reflects the fact that v can be trusted at level at most R. Unlike Fournet and Gordon, we do not need the framing operation that applies a frame to λ abstractions within an expression, because we do not use λ abstractions (and hence do not need nested frames).

In Java, permissions carry nontrivial structure and an implication ordering. For example, the `FilePermission` to write all files implies the `FilePermission` to write file `a.txt`. On this, we follow Fournet and Gordon and use a simple concrete representation: instead of reasoning about sets of Java permissions, we assume there is a fixed universe of atomic permissions and work with atomic-permission sets. Thus, `AllPermission` is a set, *All*, likely to be infinite, which contains for example the `FilePermission` to read `a.txt`. One could as well work with an arbitrary lattice; our formulation makes the connection with stack inspection slightly more transparent, and allows us to use the intuitive union and intersection operators.

2.1 Syntax

Figure 5 gives the syntax of our language. Typing rules are straightforward and omitted. In a program K **in** $R[C]$, the frame R represents the static permissions assigned to the main program, which for practical purposes will be *All*, as in the Sun Microsystems reference implementation of the Java Virtual Machine (JVM) embedded in the Java 2 platform and in the Microsoft implementation of the JVM embedded in Internet Explorer V4.0 [43].

Security-sensitive events are security-sensitive operations, such as `System.exit`, that take no parameters and have no receiver. Framing values may not be sufficient to prevent untrusted code from triggering the execution of a security-sensitive event. Therefore, IBAC includes the ordinary stack-inspection mechanism. Our formulation uses the eager semantics [18, 44, 43]: A command C_1 is executed in the context of a set D of *dynamic permissions*, which represent the permissions for which a stack inspection would be successful; D is inspected by the command **test** R **then** C_1 **else** C_2, which models `checkPermission` in Java and `Demand` in CLR, together with exception handling.

To model ordinary stack inspection, the body of every procedure is framed with a set of permissions. Specifically, in a procedure declaration, $p(x : T) = R[C]$, the frame R represents the set of the *static permissions* granted to procedure p. In practice, R is the set of permissions that the current access-control policy assigns to the class in which p is implemented. Multiple arguments for a procedure can be encoded using records.

The novel command, **test** R **for** E, tests whether the frame on the value of E contains R. If so, there is no effect on the state; if not, the command aborts. Note that there is not an **else**-branch to model exception handling. Indeed, if the outcome of the **test** could be determined, frames would serve as a new storage channel (just as the dynamic permissions do in HBAC [5]). It could be interesting to investigate frames on frames to allow and track such flows, but in this paper we simply prevent them.

2.2 Semantics

The semantics involves framed values. A *value* is either a reference, an integer or boolean literal, or $null$. (Note that a record is not a value and is not framed; this provides a proper treatment of shared mutable objects [6].) A *store* s maps each global variable to a framed value $R[v]$. In addition, it maps the special program-counter variable pc to a set of permissions. Thus, $s(pc)$ is like a label on the program counter. A *state* is a pair (s, h), where s is a store and h a *heap*; the latter is a finite partial function from references to records (that is, mapping pointers to labeled tuples of framed values).

Our semantics is designed to resemble that of Fournet and Gordon; it uses an eager semantics for stack inspection, thereby avoiding the need for an explicit run-time stack. However, it differs in several ways. We add the heap and store to model mutable variables, framed values to track information flow, and a variable pc to track information flow via the program counter.

We write $(K \textbf{ in } R[C],\ s,\ h) \Downarrow (s', h')$ to express that a program K **in** $R[C]$, executed from initial state (s, h), terminates in state (s', h'). (Our security conditions pertain to terminating computations only.)

This relation is defined in terms of a relation on commands: we write

$$(C, s, h) \Downarrow^S_D (s', h')$$

to express that command C terminates in state (s', h') when executed from initial state (s, h) under static permissions S and dynamic permissions D—where D is invariably a subset of S. In fact, $\Downarrow^S_D$ depends on the procedure declarations in K, but we elide K for brevity since it is fixed.

The semantics of a complete program is defined as follows:[2]

$$\frac{(C, s_0, h) \Downarrow^R_R (s', h') \qquad s_0 = [s \mid pc \mapsto R]}{(K \textbf{ in } R[C],\ s,\ h) \Downarrow (s', h')}$$

where, as observed, R is typically *All*.

The relation $(C, s_0, h) \Downarrow^R_D (s', h')$ is defined by induction[3] on $\Downarrow$ and cases on C, but first we give the semantics of expressions. The relation $(E, s, h) \Downarrow^S_D R[v]$ means that, in state (s, h), expression E evaluates, under static permissions S and dynamic permissions D, to framed value $R[v]$. The intention, which will be confirmed by Theorem 3.4, is that R records the influences on v—which are only by data flow, since we omit conditionals at the level of expressions. Relation $\Downarrow^S_D$ is defined inductively by the following rules:[4]

$$(\textbf{true}, s, h) \Downarrow^S_D S[true] \qquad (\textbf{false}, s, h) \Downarrow^S_D S[false]$$

$$(n, s, h) \Downarrow^S_D S[n]$$

$$\frac{(E_1, s, h) \Downarrow^S_D R_1[v_1] \qquad (E_2, s, h) \Downarrow^S_D R_2[v_2]}{(E_1 \oplus E_2, s, h) \Downarrow^S_D R_1 \cap R_2[v_1 \oplus v_2]}$$

$$\frac{s(x) = R[v]}{(x, s, h) \Downarrow^S_D S \cap R[v]}$$

$$\frac{(x, s, h) \Downarrow^S_D R[o] \qquad hof = R'[v]}{(x.f, s, h) \Downarrow^S_D R \cap R'[v]}$$

Next, we specify two unusual operations for the semantics of commands. We assume given two functions, $write\_oracle$ and $taint$, such that:

- For any C, s, h, $write\_oracle(C, s, h)$ is a pair (V, F), where V is a set of variables and F a set of locations. Each location is of the form (o, f), where o is a reference and f a field name.
- For any R, V, F, s, h, $taint(R, V, F, s, h)$ is a state (s', h') such that $s'(pc) = s(pc)$. Moreover, the domains of h and h' are the same.

Furthermore, these operations satisfy the following conditions:

- $write\_oracle(C, s, h)$ represents the writes of C in (s, h). If $(C, s, h) \Downarrow^S_D (s', h')$ and $write\_oracle(C, s, h) = (V, F)$, then V is the set of variables updated from s to s' and F is the set of locations updated from h to h'.
- $taint$ changes the state only by shrinking frames. That is, if $taint(R, V, F, s, h) = (s', h')$, $s(x) = P[v]$, and $s'(x) = P'[v']$, then $P \supseteq P'$ and $v = v'$, and similarly for record fields.
- $taint$ imposes R on (V, F). That is, for any x in V, if $s(x) = P[v]$ and $s'(x) = P'[v']$ then $R \supseteq P'$, and similarly for record fields.

For programs using only variables, $write\_oracle$ can be overapproximated by simple static analysis that tracks assignment targets. To take the heap into account, techniques from program analysis and verification (where this is known as a "modifies clause") can be used; see, for example, [2] and references therein. A dynamic oracle can be given

[2] The notation $[s \mid x \mapsto \ldots]$ indicates an update of variable x in store s, where x is either a global variable or the program-counter variable, pc. Similarly, the notation $[h \mid o.f \mapsto \ldots]$ indicates an update of field f for the record of reference o in the heap.

[3] To be precise, our rules inductively define a family of relations $\Downarrow^S_D$ for all S and all $D \subseteq S$. This is needed because the rules for procedure call and $\textbf{grant}$ change the static and dynamic permissions.

[4] In the rules, parse $R_1 \cap R_2[v]$ as $(R_1 \cap R_2)[v]$. For field access, note that ho is the record at reference o and thus hof is the value of its field f.

$$\frac{p(x\colon T) = R[C] \qquad (E,s,h) \Downarrow_D^S P[v] \qquad (C,[s \mid x \mapsto s(\mathit{pc}) \cap S \cap P[v]],h) \Downarrow_{D\cap R}^R (s',h')}{(p(E),s,h) \Downarrow_D^S (s',h')}$$

$$\frac{R \subseteq D \qquad (C_1,s,h) \Downarrow_D^S (s',h')}{(\mathbf{test}\ R\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2,s,h) \Downarrow_D^S (s',h')} \qquad \frac{R \not\subseteq D \qquad (C_2,s,h) \Downarrow_D^S (s',h')}{(\mathbf{test}\ R\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2,s,h) \Downarrow_D^S (s',h')}$$

$$\frac{(E,s,h) \Downarrow_D^S P[v] \qquad R \subseteq P}{(\mathbf{test}\ R\ \mathbf{for}\ E,s,h) \Downarrow_D^S (s,h)} \qquad \frac{(C,s,h) \Downarrow_{D\cup(R\cap S)}^S (s',h')}{(\mathbf{grant}\ R\ \mathbf{in}\ C,s,h) \Downarrow_D^S (s',h')}$$

$$\frac{(C_1,s,h) \Downarrow_D^S (s_1,h_1) \qquad (C_2,s_1,h_1) \Downarrow_D^S (s',h')}{(C_1; C_2,s,h) \Downarrow_D^S (s',h')}$$

$$\frac{\begin{array}{c}(E,s,h) \Downarrow_D^S R[\mathit{false}] \qquad s_0 = [s \mid \mathit{pc} \mapsto s(\mathit{pc}) \cap R] \\ (C_2,s_0,h) \Downarrow_D^S (s_2,h_2) \qquad (V,F) = \mathit{write\_oracle}(C_1,s,h) \qquad (s',h') = \mathit{taint}(s_0(\mathit{pc}),V,F,s_2,h_2)\end{array}}{(\mathbf{if}\ E\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2,s,h) \Downarrow_D^S ([s' \mid \mathit{pc} \mapsto s(\mathit{pc})],h')}$$

$$\frac{(x,s,h) \Downarrow_D^S R[o] \qquad (E,s,h) \Downarrow_D^S R'[v]}{(x.f := E,s,h) \Downarrow_D^S (s,[h \mid o.f \mapsto s(\mathit{pc}) \cap S \cap R \cap R'[v]])}$$

$$\frac{(\overline{E},s,h) \Downarrow_D^S \overline{R}[\overline{v}] \qquad o \notin \mathit{dom}(h)}{(x := \mathbf{ref}\{\overline{f} = \overline{E}\},s,h) \Downarrow_D^S ([s \mid x \mapsto s(\mathit{pc}) \cap S[o]],[h \mid o.\overline{f} \mapsto s(\mathit{pc}) \cap S \cap \overline{R}[\overline{v}]])}$$

Figure 6. Command Semantics

by simulating the command for some bounded number of steps. The most precise *taint* function changes (s,h) to (s',h') by intersecting R with the frames of the values in the variables and locations in V and F, respectively.

Finally, we can give the semantics of commands. For assignments, the rule is as follows:

$$\frac{(E,s,h) \Downarrow_D^S R[v]}{(x := E,s,h) \Downarrow_D^S ([s \mid x \mapsto s(\mathit{pc}) \cap S \cap R[v]],h)}$$

Expression E evaluates to some framed value $R[v]$; the store is updated using the frame $s(\mathit{pc}) \cap S \cap R$ to take into account both the control dependence recorded in pc and the static permissions of the code performing the assignment.

It is well known that control dependence allows information to flow via the absence of an assignment. This is tracked in the semantics of conditional, which is defined as follows:

$$\frac{\begin{array}{c}(E,s,h) \Downarrow_D^S R[\mathit{true}] \\ s_0 = [s \mid \mathit{pc} \mapsto s(\mathit{pc}) \cap R] \qquad (C_1,s_0,h) \Downarrow_D^S (s_1,h_1) \\ (V,F) = \mathit{write\_oracle}(C_2,s,h) \\ (s',h') = \mathit{taint}(s_0(\mathit{pc}),V,F,s_1,h_1)\end{array}}{(\mathbf{if}\ E\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2,s,h) \Downarrow_D^S ([s' \mid \mathit{pc} \mapsto s(\mathit{pc})],h')}$$

Here, C_1 is executed with initial program counter pc that reflects the control dependence [16]. *write_oracle* is applied to the branch not taken, and *taint* is used to shrink the frames of values of potentially-updated variables and heap locations using the frame of the branch condition. An example showing the usage of the *write_oracle* and *taint* functions is discussed in Section 2.3.

The remaining rules are similar and can be found in Figure 6. As remarked earlier, we do not allow observable branching on frames of expressions, since this would introduce a new storage channel. Thus, nothing like *write_oracle*/*taint* is needed in the semantics of $\mathbf{test}\ R\ \mathbf{for}\ E$.

2.3 Example

For brevity, let us augment the set of commands with **skip**, with semantics $(\mathbf{skip},s,h) \Downarrow_D^S (s,h)$. Now consider the evaluation of the code

$$\begin{array}{c}\mathbf{var}\ l,x,y\colon\mathbf{int};\ \mathbf{var}\ c\colon\mathbf{bool};\ \mathbf{in} \\ l := x;\ \mathbf{if}\ c\ \mathbf{then}\ l := y\ \mathbf{else\ skip}\end{array}$$

in a store s, where $s(\mathit{pc}) = P$, $s(x) = R_0[0]$, $s(y) = R_1[1]$, and $s(c) = R[\mathit{true}]$. Since this code is simple imperative, we elide the heap h in the semantics. We have

$(l\!:=x, s) \Downarrow^S_D s_1$, where $s_1 = [s \mid l \mapsto P \cap S \cap R_0[0]]$. The evaluation of $(\mathbf{if}\ c\ \mathbf{then}\ l\!:=y\ \mathbf{else\ skip}, s_1)$ yields the store s' in the following manner:

$$\frac{\begin{array}{c} (c, s_1) \Downarrow^S_D R[\mathit{true}] \qquad s_0 = [s_1 \mid pc \mapsto P \cap R] \\ s' = [s_0 \mid l \mapsto P \cap S \cap R \cap R_1[1]] \\ (l\!:=y, s_0) \Downarrow^S_D s' \qquad \varnothing = \mathit{write\_oracle}(\mathbf{skip}, s_1) \\ s' = \mathit{taint}(P \cap R, \varnothing, s') \end{array}}{(\mathbf{if}\ c\ \mathbf{then}\ l\!:=y\ \mathbf{else\ skip}, s_1) \Downarrow^S_D [s' \mid pc \mapsto P]}$$

On the other hand, consider the evaluation of the same code in a store t where $t(pc) = P'$, $t(x) = R'_0[0]$, $t(y) = R'_1[1]$, and $t(c) = R'[\mathit{false}]$. We have $(l\!:=x, t) \Downarrow^S_D t_1$, where $t_1 = [t \mid l \mapsto P' \cap S \cap R'_0[0]]$. The evaluation of $(\mathbf{if}\ c\ \mathbf{then}\ l\!:=y\ \mathbf{else\ skip}, t_1)$ yields the store t' in the following manner:

$$\frac{\begin{array}{c} (c, t_1) \Downarrow^S_D R'[\mathit{false}] \qquad t_0 = [t_1 \mid pc \mapsto P' \cap R'] \\ (\mathbf{skip}, t_0) \Downarrow^S_D t_0 \qquad \{l\} = \mathit{write\_oracle}(l\!:=y, t_1) \\ t' = \mathit{taint}(P' \cap R', \{l\}, t_0) \end{array}}{(\mathbf{if}\ c\ \mathbf{then}\ l\!:=y\ \mathbf{else\ skip}, s_1) \Downarrow^S_D [t' \mid pc \mapsto P']}$$

Note that, as a result of applying the *taint* function, $t'(l) = P' \cap R' \cap S \cap R'_0[0]$.

3 Noninterference

The idea of our information-flow policy is that sections of code *guarded* (immediately dominated) by **test** calls are sensitive and thus should also be guarded by permission checks for the values read in these sections of code. In static labeling, an observer at some level L is assumed to read only the parts of state with integrity label at most L, that is, trustworthy for the observer. Here, the observer is expected to check permissions on values it reads. This motivates the main definitions of indistinguishability that are used to define noninterference, which is the semantic interpretation of the information-flow policy.

This section proves two main results: Theorem 3.4 establishes that all programs are noninterferent in a nonstandard sense: given any two initial states that agree on variables and records that are *trustable* (with frames containing some set Q of permissions required by an observer), then the final states agree on the values that are trustable for Q. Theorem 3.5 connects this property with a standard notion of noninterference [19], by considering programs that include explicit permission checks.

For simplicity, in this version of the paper, the results in this section are formalized without the heap. Note that, except for the choice of fresh references in the semantics of **ref**, the language is deterministic. In the full paper, we assume an arbitrary deterministic allocator and track allocation behavior using renamings as in [6]. This lets us use a notion of noninterference suitable for deterministic systems.

Definition 1 (Q-indistinguishability, $\sim_Q$) *States s and t are Q-*indistinguishable *for an observer with permission set Q, written $s \sim_Q t$, iff for all R, R', v, v' and all variables x (other than pc) the following is true:*

$$\begin{array}{l} \text{if } s(x) = R[v] \text{ and } t(x) = R'[v'], \\ \text{then } R \supseteq Q \wedge R' \supseteq Q \Rightarrow v = v'. \end{array}$$

Lemma 3.1 gives a sense in which indistinguishability is preserved by expressions (analogous to the simple security property, or "read confinement" [6]). Its proof is a straightforward induction on $\Downarrow^S_D$.

Lemma 3.1 *If $s \sim_Q t$, $(E, s) \Downarrow^S_D R[v]$, and $(E, t) \Downarrow^S_D R'[v']$, then $R \supseteq Q \wedge R' \supseteq Q \Rightarrow v = v'$.*

Two additional technical results are needed. Because *taint* only reduces frames, we get the following lemma using the definition of $\sim_Q$:

Lemma 3.2 *If $s \sim_Q t$ and $s_0 = \mathit{taint}(R, V, s)$, then $s_0 \sim_Q t$ for any R, V. Symmetrically, if $s \sim_Q t$ and $t_0 = \mathit{taint}(R, V, t)$, then $s \sim_Q t_0$.*

Since the semantics tracks control dependence in variable pc and uses pc in the semantics of variable (and field) writes, we get the following lemma:

Lemma 3.3 *Suppose $(C, s) \Downarrow^R_S t$ and $t(x)$ differs from $s(x)$, that is, x is written by C. If $t(x) = P[v]$, then $P \subseteq s(pc)$.*

In the following sense, every program is noninterferent!

Theorem 3.4 *If $s \sim_Q s'$, $(C, s) \Downarrow^S_D t$, and $(C, s') \Downarrow^S_D t'$, then $t \sim_Q t'$.*

Proof: The proof is by induction on $\Downarrow^S_D$. We give just the case where C is $\mathbf{if}\ E\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2$. As in the semantics, let $(E, s) \Downarrow^S_D R[b]$ and $(E, s') \Downarrow^S_D R'[b']$. Let $s_0 = [s \mid pc \mapsto s(pc) \cap R]$ and $s'_0 = [s' \mid pc \mapsto s'(pc) \cap R']$. We have two subcases depending on whether or not the two runs take the same branch:

1. *The same branch is taken.* Without loss of generality, let the branch taken be C_1. Let $(C_1, s_0) \Downarrow^S_D t_1$ and $(C_1, s'_0) \Downarrow^S_D t'_1$. By Definition 1 and hypothesis $s \sim_Q s'$ we have $s_0 \sim_Q s'_0$, so by induction on $\Downarrow^S_D$ for C_1 we have $t_1 \sim_Q t'_1$. Let $V = \mathit{write\_oracle}(C_2, s_0)$ and let $V' = \mathit{write\_oracle}(C_2, s'_0)$. Thus, by semantics, $t = \mathit{taint}(s(pc) \cap R, V, t_1)$ and $t' = \mathit{taint}(s'(pc) \cap R', V', t'_1)$. From $t_1 \sim_Q t'_1$, using Lemma 3.2, we get $t \sim_Q t'_1$, and using Lemma 3.2 again, we get $t \sim_Q t'$.

2. *Different branches are taken.* Without loss of generality, let $b = \mathit{true}$ and let $b' = \mathit{false}$. Let $(C_1, s_0) \Downarrow^S_D t_1$ and $(C_2, s'_0) \Downarrow^S_D t'_1$. Let $V = \mathit{write\_oracle}(C_2, s_0)$ and let $V' = \mathit{write\_oracle}(C_1, s'_0)$. Thus, by semantics, the final states are $t = \mathit{taint}(s(pc) \cap R, V, t_1)$ and $t' = \mathit{taint}(s'(pc) \cap R', V', t'_1)$. To show $t \sim_Q t'$, consider any x and let $t(x) = P[v]$ and $t'(x) = P'[v']$. If $t(x) = s(x)$ and $t'(x) = s'(x)$ then the Q-indistinguishibility condition holds from hypothesis $s \sim_Q s'$. It remains to consider two cases: either $t(x)$ differs from $s(x)$ or $t'(x)$ differs from $s'(x)$.

   These cases are symmetric so, without loss of generality, suppose *$t(x)$ differs from $s(x)$*. Thus, x was updated by C_1, so $t(x) = P[v]$ for some P, v with $P \subseteq s(pc) \cap R$ (owing to semantics and Lemma 3.3 applied to $(C_1, s_0) \Downarrow^S_D t_1$). Note that we have either $Q \not\subseteq R$ or $Q \not\subseteq R'$ since otherwise, by Lemma 3.1, both runs would have taken the same branch. If $Q \not\subseteq R$, then $Q \not\subseteq P$ (because $P \subseteq s(pc) \cap R$), so the antecedent for Q-indistinguishability of x is falsified and we are done. Otherwise, $Q \not\subseteq R'$. Now, since x is updated by C_1 from s_0, by assumption on $\mathit{write\_oracle}$ we have $x \in V'$ and thus, by assumption about taint, we have $P' \subseteq s'(pc) \cap R'$, so $Q \not\subseteq P'$ and again the antecedent for Q-indistinguishability of x is falsified. This concludes the proof of $t \sim_Q t'$. □

Finally, we connect Theorem 3.4 with the standard notion of noninterference [19] for policies that are specified by labeling input and output channels with fixed levels, and such that inputs labeled as untrusted do not influence outputs labeled as trusted.

Without loss of generality, consider a main program of the form

$$K;\ \mathbf{in}\ R[\mathbf{test}\ R\ \mathbf{for}\ w; C; \mathbf{test}\ R\ \mathbf{for}\ w]$$

where w is one of the global variables. We interpret this to specify an implicit information-flow policy ι wherein w is trusted and the other variables are not. For this policy, the interesting observer level is R.

We say that two states s, t are *ι-indistinguishable* provided that $s(w) = t(w)$, and for all other variables x, the frames of $s(x)$ and $t(x)$ do not contain R. We say s and t are *weakly ι-indistinguishable* if they agree on the value of w though not necessarily its frame: $s(w) = P[v]$ and $t(w) = P'(v')$ with $v = v'$). (There is no condition on other frames.)

Theorem 3.5 *Consider two ι-indistinguishable initial states s, t. If s_1 is the final state from running the program on s and t_1 is the final state from running the program on t, then s_1 is weakly ι-indistinguishable from t_1.*

That is, if there is no access error when the final test is performed, then w has not been influenced by untrusted inputs.

Proof: If the initial $\mathbf{test}\ R\ \mathbf{for}\ w$ fails in one of the computations there is nothing to prove. If it succeeds in both computations then we have $s \sim_R t$. Let s_1 and t_1 be the states after executing C. By Theorem 3.4 we have $s_1 \sim_R t_1$. Let $s_1(w) = P[v]$ and $t_1(w) = P'[v']$. If the final tests $\mathbf{test}\ R\ \mathbf{for}\ w$ both succeed then by semantics we have $P \supseteq R$ and $P' \supseteq R$. Thus, by definition of $\sim_R$ we have $v = v'$ as was to be proved. □

4 Case Study

Section 1 presented examples of Java programs for which SBAC was shown to be too permissive and HBAC too restrictive. This section revisits those examples, showing that IBAC performs as expected by enforcing the information-flow policy extracted from the access-control policy. Additionally, this section covers examples from production-level Java code, demonstrating the need for IBAC when enabling security on Java code.

For simplicity, the language that we defined in Section 2 to introduce IBAC only dealt with global variables. In this section, to model Java more closely, we allow the language to have local variables too; just like a global variable, each local variable is mapped to a framed value. Additionally, we assume that a command of the form `B b = new B()`, which allocates and constructs a new object of type `B`, and assigns it to variable `b`, causes that object to be framed by $s(pc) \cap S \cap R$, where S is the set of the static permissions of the code performing the allocation and R is the set of permissions that the access-control policy grants to class `B`. A similar approach is taken when the type of the assigned value is primitive, with the understanding that, in that case, $R = \mathit{All}$.

4.1 Revisiting the Running Examples

For the example of Figure 1, according to the semantics of Section 2.2, we have the following:

- Initially, $s(pc) = \mathit{All}$.
- The body of `A.main` is executed with static permissions R_1 and dynamic permissions R_1.
- The object of type `B` pointed to by `b` in `A.main` is framed with $\mathit{All} \cap R_1 \cap R_2 = \varnothing$.
- The `String` object `"password.txt"` allocated in `m1` is framed with $\mathit{All} \cap R_2 \cap R_3 = \varnothing$.
- The `String` object assigned to `fileName` in `A.main` is framed with $\mathit{All} \cap R_1 \cap \varnothing = \varnothing$.

- The `FileOutputStream` object pointed to by `f` in `A.main` is framed with $\mathit{All} \cap R_1 \cap R_3 = R_1$.

- The `String` object pointed to by the `name` parameter in the `FileOutputStream.<init>(String)` constructor is framed with $\mathit{All} \cap R_1 \cap \varnothing = \varnothing$.

- The `File` object allocated in the `FileOutputStream.<init>(String)` constructor and pointed to by `file` is framed with $\varnothing \cap R_3 \cap R_3 = \varnothing$ since now the program counter is control-dependent on the `String` object pointed to by `name`, so $s(pc) = \varnothing$.

- The `String` object pointed to by `name` in the `FileOutputStream.<init>(File, boolean)` constructor is framed with $\varnothing$ since it is the return value of `file.getPath`, and this method returns exactly the object pointed to by `fileName` in `A.main`. Moreover, the value of `name` is control-dependent on the `File` object pointed to by `file`, which implies that $s(pc) = \varnothing$.

- The value of the `boolean` parameter `append` in the `FileOutputStream.<init>(File, boolean)` constructor is framed with $\mathit{All} \cap R_3 \cap \mathit{All} = \mathit{All}$.

- The `FileOutputStream.<init>(String)` constructor body is executed with static permissions R_3 and dynamic permissions $R_1 \cap R3 = R_1$.

- The `FileOutputStream.<init>(String, boolean)` constructor body is executed with static permissions R_3 and dynamic permissions $R_1 \cap R3 = R_1$.

The execution of the IBAC **test** command in constructor `FileOutputStream.<init>(String, boolean)` performs the following operations:

1. Verify that the set D of the dynamic permissions associated with the current stack contains the singleton $R = \{$`FilePermission "passwords.txt", "write"`$\}$. This test succeeds because $D = R_1 \supset R$.[5] This would have been the only check performed by the SBAC **test**.

2. Verify that the frame of the `boolean` value pointed to by `append` contains R. This test is caused by the fact that the value of `append` is read in the security-sensitive operation guarded by the `checkPermission` call. This test also succeeds since that frame is All.

3. Similarly, verify that the frame of the `String` object pointed to by `name` contains R. This test fails because that frame is $\varnothing$.

Therefore, IBAC causes the security-sensitive operation to fail since it has detected an integrity violation at the point in which the IBAC **test** has been executed. As observed, SBAC would not have recognized the integrity violation, thereby allowing `b.m1` to cause the contents of `passwords.txt` to be overwritten. Coincidentally, HBAC would have caused this program to fail, but only because the execution of `b.m1` was in the history.

Another interesting example is the program of Figure 3, which uses the **grant** primitive `doPrivileged`. Assume that an untrusted client invokes `m2` on an object `c` of type `C` after setting `c.logFileName` to `"passwords.txt"`. The frame for this value will be $\mathit{All} \cap R'_2 \cap R'_3 = \varnothing$. Following the same reasoning as in the example of Figure 1, we would conclude that the following must hold:

- The `String` object pointed to by `name` in constructor `FileOutputStream.<init>(File, boolean)` is framed with $\varnothing$ since it is the return value of `file.getPath`, and this method returns exactly the object assigned to `logFileName` by the untrusted client. Moreover, the value of `name` is control dependent on the `File` object pointed to by `file`, which is framed with $\varnothing$, so $s(pc) = \varnothing$.

- The value of the `boolean` parameter `append` in the `FileOutputStream.<init>(File, boolean)` constructor is framed with $\mathit{All} \cap R'_3 = \mathit{All}$ since that value originated from constructor `FileOutputStream.<init>(String)`.

The IBAC **test** performs the following operations:

1. Verify that the set D of the dynamic permissions associated with the current stack of execution contains singleton $R = \{$`FilePermission "passwords.txt", "write"`$\}$. This test succeeds because $D = R'_3 \cap R'_1 = R'_1 \supset R$. The presence of `doPrivileged` on the stack prevents R'_2 from being part of the computation of D. This would have been the only test performed by SBAC.

2. Verify that the frame of the `boolean` value pointed to by `append` contains R. This test also succeeds since that frame is All, like in the previous example.

3. Verify that the frame of the `String` object pointed to by `name` contains R. This test fails because that frame is $\varnothing$.

Therefore, IBAC dynamically detects the presence of tainted variables in privilege-asserting code, solving a serious problem previously identified in the literature [33], and forces interferent programs permitted by SBAC to fail.

[5] We are abusing the notation here by considering the permission in R_1 as a set of atomic permissions, one of which is the permission to write file `password.txt`, as explained in Section 2.

The program of Figure 4 is useful to illustrate the difference between IBAC and HBAC. Here, when the IBAC **test** is performed, the values used in the security-sensitive operation dominated by the **test** are framed as follows:

- The String object pointed to by name in the FileOutputStream.<init>(File, boolean) constructor is framed with R''_1 since it is the return value of file.getPath, which returns exactly the String object "passwords.txt" allocated in F.main and passed to FileOutputStream.<init>(String). Moreover, the value of name is control dependent on the File object pointed to by file, which also is framed with R''_1.
- The value of the boolean parameter append in the FileOutputStream.<init>(File, boolean) constructor is framed with All since that value originated from the FileOutputStream.<init>(String) constructor.

The IBAC **test** performs the following operations:

1. Verify that the set D of the dynamic permissions associated with the current stack of execution contains singleton $R = \{$FilePermission "passwords.txt", "write"$\}$. This test succeeds because $D = R''_3 \cap R''_1 = R''_1 \supset R$.
2. Verify that the frame of the boolean value pointed to by append contains R. This test also succeeds since that frame is All.
3. Verify that the frame of the String object pointed to by name contains R. This test succeeds because that frame is $R''_1 \supset R$.

The HBAC **test** would have unjustly caused this program to fail because g.m3 is in the history, and class G has been granted no permissions.

4.2 Production-level Code

The programs of Sections 1.1 and 1.2 that we have just revisited are not contrived examples, but have been inspired by production-level code. Currently, the core of the Eclipse platform, called the Rich Client Platform (RCP), is undergoing extensive code rewriting to permit it to run with security enabled [15]. This work is being performed by using a static analyzer to model the stack inspection mechanism [39] and detect the permission set required by each RCP component [27] and the security-sensitive operations that should be wrapped into privilege-asserting blocks of code to shield other Eclipse components from unnecessary permission requirements [33]. The problem with SBAC, as we have noted, is that an untrusted component may influence the security-sensitive operations performed in a more trusted component, without being on the stack of execution when the security check is performed. A concrete example is the following: Method evaluate in SystemTestExpression, in the org.eclipse.core.expressions RCP component, accesses a security-sensitive system property through the following code:

```
String str = (String)
    AccessController.doPrivileged(
    new PrivilegedAction() {
  public Object run() {
    return System.getProperty(fProperty);
  }
});
```

Unfortunately, the String object pointed to by fProperty is influenced, through a series of direct and indirect flows, by the String object pointed to by the ELEMENT_ACTIVE_WHEN constant in class HandlerPersistence. This class is located in the org.eclipse.ui.workbench RCP component, which in the current version of the RCP has not been granted the PropertyPermission required to execute the code above. In an SBAC system such as the current version of Java on top of which Eclipse runs, a malicious attacker could modify the value of ELEMENT_ACTIVE_WHEN and condition the property being accessed. Since no method of HandlerPersistence is on the stack when System.getProperty calls checkPermission, this integrity exposure would not disappear by simply removing the privilege-asserting block around the call to System.getProperty. Conversely, by tracking information flows to each security-sensitive operation and testing not only the callers on the current stack of execution up to the doPrivileged caller, but also the labels on all the values read in the security-sensitive operation guarded by the checkPermission call, IBAC detects the integrity exposure. It is now up to the developers and system administrators to decide whether the code needs to be corrected to eliminate this exposure or whether it is safe to grant HandlerPersistence the appropriate PropertyPermission.

5 Proposed Implementation

We propose two different implementations for IBAC enforcement: a completely static enforcement, and one that combines static analysis with dynamic enforcement.

A completely static IBAC enforcement can be achieved by considering the set of all the statements guarded by the **test**. For each statement s guarded by a **test**R command, it must be the case that no statement $s' \in$ *backslice*(s) originates from code with static permission set P such that $P \not\supseteq R$. Here, *backslice* is the function that maps each statement s to its *static backwards slice*, consisting of all the (transitive) predecessors of s along control- and data-dependence edges in the Program Dependence Graph (PDG). The set of statements guarded by a **test** can be assumed to be the set of statements immediately following the **test** call.[6] For example, in the `FileOutputStream` code of Figure 2, the statements guarded by the `checkPermission` call are the assignment to `append` and the calls to `openAppend` and `open`.

The stack inspection mechanism necessary to prevent untrusted code from causing the execution of a security-sensitive event can also be modeled statically. A call graph and a points-to graph [21] representing the execution of the program under analysis are built. For each node in the call graph corresponding to a `checkPermission` method call, the set P of abstract `Permission` objects that, according to the model, could have flowed to the `Permission` parameter of `checkPermission` is computed. The level of abstraction could be by allocation site, as in Andersen's analysis [3]. Next, each of these `Permission` sets is propagated backwards in the call graph, performing set unions at merge points, until a fix point is reached [26]. The only nodes in this fix-point iteration that *kill* the reverse propagation of the `Permission` sets are the `doPrivileged` nodes, to model the requirement that a call to `doPrivileged` on a stack of execution causes the stack inspection to stop regardless of the `Permission` being checked.[7] Once a fix point has been reached, each node in the call graph is mapped to a set of abstract `Permission` objects, overapproximating the permissions required at run time by the method represented by that node. The IBAC enforcement will require each class in the program under analysis to be granted at least the permissions required by each of its declared methods according to the model.

This approach has the advantage that it is *sound*; it identifies all the integrity violations. From a security perspective, this is important because all such violations will be detected. Its disadvantage is that it is potentially *conservative*, with the result that some of the reported violations may in reality be *false alarms*. The precision of the analysis can significantly affect its scalability. Alternatively, the stack-inspection mechanism can still be enforced dynamically, while static analysis is used only to track direct and indirect flows causing integrity violations.

Dynamic IBAC enforcement can be carried out by associating a label with every value, based on the static permissions granted by the current access-control policy. At each existing **test**R command (corresponding to calls to `checkPermission` in Java and `Demand` in CLR), an ordinary stack inspection is performed. However, it is necessary for the frame P of each expression E guarded by the **test** to satisfy $P \subseteq R$. This requires inserting additional **test** R **for** E calls for each such expression E. This approach must be coupled with a static analysis implementing the $write\_oracle$, which is responsible for overapproximating the set of values that would be modified at every branch not taken, as explained in Section 2.2.

6 Related Work

Early work by Denning and Denning [13] focuses on static analysis for information flow. Subsequently, Goguen and Meseguer [19] introduce a more general notion of information flow based on noninterference. Volpano, et al. [41] show a type-based algorithm that certifies implicit and explicit flows and also guarantees noninterference.

Noninterference is traditionally the technical criterion used for proving correctness of security analysis algorithms or type systems. However, it is also hard to check noninterference directly. Snelting et al. [37] connect PDGs with noninterference. Hammer et al. [23] present a PDG-based algorithm for verifying noninterference: for any output statement s, it must be the case that any statement in *backslice*(s) must have a security labels lower than the security label of s. It should be noted, however, that type-system-based algorithms have the advantage of supporting compositional analysis, which means that parts of programs can be analyzed in isolation. This is generally hard to do in PDG-based analysis.

Taint analysis is an integrity problem, in which the focus is whether untrusted data obtained from the user might influence other data that the system trusts. The notion of tainted variables became known with the Perl language. In Perl, using the `-T` option allows detecting tainted variables [42]. Shankar et al. present a tainted-variable analysis for CQual using constraint graphs [36]. To find format-string bugs, CQual uses a type-qualifier system [17] with two qualifiers: *tainted* and *untainted*. A constraint graph is constructed for a CQual program. If there is a path from a tainted node to an untainted node in the graph, an error

[6] In fact, this should be a security coding guideline, which, if enforced, helps preventing violations of the Principle of Complete Mediation [35] as well as Time-Of-Check-To-Time-Of-Use (TOCTTOU) attacks [11].

[7] We are assuming that the program is a Java program. For other languages, the algorithm is almost the same, with only slight changes. For example, in CLR, `checkPermission` would have to be replaced by `Demand`, `doPrivileged` by `Assert`, and `Permission` by `IPermission`. Additionally, it should be noted that a `doPrivileged` node kills the universe of all the `Permission` objects associated with the program under analysis, whereas `Assert` only kills the `IPermission` objects that have been passed to its `IPermission` parameter. The reason behind this difference is that `doPrivileged` stops the stack inspection indiscriminately, while `Assert` only stops it for those `IPermission` objects that have been passed as parameters to it [32].

is flagged. Newsome and Song [31] propose a dynamic taint analysis that catches errors by monitoring tainted variables at run time. Data originating or arithmetically derived from untrusted sources, such as the network, are marked as tainted. Tainted variables are tracked at run time, and when they are used in a dangerous way an attack is detected. Volpano et al. [41] relate taint analysis to enforcing information-flow policies through typing. Ashcraft and Engler [4] also use taint analysis to detect software attacks due to tainted variables. Their approach provides user-defined sanity checks to untaint potentially tainted variables. To perform taint analysis, one needs to identify sources and sinks of possibly-tainted data. This amounts to identifying methods that originate a tainted value and methods that use a possibly-tainted value. Livshits and Lam's analysis algorithm [28] requires prior computation of a specific flow-insensitive points-to heap analysis and assumes the presence of programmer-supplied descriptors for sources and sinks. Additionally, their algorithm requires the presence of descriptors for library methods handling objects through which taintedness may flow. This approach does not handle indirect flows. By contrast, the algorithm described in this paper takes into account indirect flows and does not require any programmer-defined descriptor. Pistoia et al.[33] use program slicing to detect tainted variables in privilege-asserting code. The work presented in this paper subsumes that work, as observed in Section 4.1.

Stoughton [38] 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. Myers' Jif tool [29] uses type-based static analysis [41, 13] to track information flow in Java, but like in many other works the information-flow policy is expressed by a static labeling assumed given. Banerjee and Naumann [6] augment such a type system with an effect analysis for SBAC, and allow that a procedure's labeling may depend on the permissions authorized for it at runtime; noninterference is proved (and has even been machine-checked [30]). However, their information lattice is separate from permissions. They also adapt their system to HBAC, where the dynamic permissions can be exploited as a storage channel [5]. Barthe and Rezk [8] prove noninterference for a security type system for Java bytecode. Barthe et al. [7] prove that typable source code compiles to typable bytecode. Zhao and Boyland [45] combine type-based and dynamic checks to improve security of stack inspection by tracking data flows. However, they do not consider implicit flow. In contrast to static checking of noninterference, Le Guernic, et al. [22] consider dynamic, automaton-based, monitoring of information flow for a single execution of a sequential program. The mechanism is based on a combination of dynamic and static analyses. The dynamic analysis accepts or rejects a single execution of a program without necessarily doing the same for all other executions. The automaton is used to guarantee confidentiality of secret data and takes into account direct as well as implicit flows. The static analysis is used to overapproximate *implicit indirect flows*—that is, the branches not executed during the execution of conditionals—and to generate corresponding branch-not-taken inputs to the automaton. This is similar in spirit to the $write\_oracle$ in our semantics. On the other hand, that work does not consider the derivation of an information-flow policy from an access-control policy.

A number of works focus on stack inspection as an access-control policy enforcement, with the purpose of defining alternative implementations and studying optimization techniques. Wallach, et al. [44, 43] present an approach called Security Architecture Formerly Known as Stack Inspection (SAFKASI), which uses the calculus of Security-Passing Style (SPS) to enforce a form of access control equivalent to stack inspection. Pottier et al.[34] extend and formalize the SPS calculus via type theory using a λ-calculus, called λ_{sec}. Jensen, et al. [25] focus on proving that code is secure with respect to a global security policy. Their model uses operational semantics to prove the properties, using a two-level temporal logic, and shows how to detect redundant authorization tests. Bartoletti, et al. [9] are interested in optimizing performance of run-time authorization tests by eliminating redundant tests and relocating others as needed. Rather than analyzing security policies as embodied by existing code, Erlingsson and Schneider [40] describe a system that inlines reference monitors into the code to enforce specific security policies. This approach can reduce or eliminate redundant authorization tests. Koved, et al. [27] use static analysis for identification of permission requirements. Besson, et al. [10] present a static analysis technique for verifying the security of libraries in SBAC systems.

7 Discussion

In this paper, we have introduced IBAC, a new access-control model that, for any security-sensitive operation, verifies that all the code responsible for that operation is sufficiently authorized. IBAC automatically infers the information-flow policy labels for a program from an existing access-control policy associated with the program, and transforms existing access-control tests into information-flow tests. The idea behind IBAC comes from a process that happens quite often in production-level code; precisely, code that was written to run without SBAC enforcement, is later required to run with SBAC enabled. System administrators must figure out which permissions that code requires, and developers must insert **grant** calls appropriately to prevent unnecessary permission requirements from percolating up the stack and affect client code. However, an inherent flaw in SBAC is that code influencing a security-sensitive operation may no longer be on the stack of execution for that operation. Therefore, an SBAC **test** call may fail to verify the permission assignments of all the code associated with a security-sensitive action, and a **grant** call may allow untrusted code to influence, through tainted variables, the execution of trusted library code. Furthermore, in this paper, we have compared IBAC to HBAC and demonstrated that IBAC permits the execution of safe programs that HBAC would instead reject.

SBAC is also vulnerable to confidentiality attacks in systems that enforce capability-based security. Consider for example the Java program in Figure 4. As observed in Section 4.1, IBAC accepts that program. Suppose, however, that the `FileOutputStream` object created in `F.main` is inadvertently allowed to escape the security context in which it was created. For example, it could be passed as a parameter to a method `m4` of an object `h` of type `H`, or stored into the heap by `F.main` and subsequently accessed by `h`, where `H` is a class with no `FilePermission "passwords.txt", "write"`. Now, `h.m4` could call `write` on that `FileOutputStream` object, and its permissions are not checked since no object of type `H` was on the stack when that `FileOutputStream` object was created—a confidentiality violation. This paper has discussed the integrity aspects of IBAC. In the future, we would like to extend this work to confidentiality as well. Another interesting area of research is how to integrate IBAC with a mechanism for declassification.

So far, we have implemented a subsystem of IBAC, which only enforces the rejection of integrity violations caused by tainted variables in privilege-asserting code. We plan to have a full implementation of IBAC enforcement and to validate its usefulness on production-level code that has adopted SBAC as its form of access control.

References

[1] M. Abadi and C. Fournet. Access Control Based on Execution History. In *Proceedings of the 11th Network and Distributed System Security Symposium (NDSS 2003)*, San Diego, CA, USA, Feb. 2003.

[2] T. Amtoft, S. Bandhakavi, and A. Banerjee. A Logic for Information Flow in Object-Oriented Programs. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, pages 91–102, Charleston, SC, USA, Jan. 2006. Extended version as KSU CIS-TR-2005-1.

[3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, Copenhagen, Denmark, May 1994.

[4] K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 143–159, Oakland, CA, USA, May 2002. IEEE Computer Society.

[5] A. Banerjee and D. A. Naumann. History-based Access Control and Secure Information Flow. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop (CASSIS 2004), Revised Selected Papers*, volume 3362 of *Lecture Notes in Computer Science*, 2005.

[6] A. Banerjee and D. A. Naumann. Stack-based Access Control for Secure Information Flow. *Journal of Functional Programming*, 15(2):131–177, Mar. 2005. Special Issue on Language Based Security.

[7] G. Barthe, D. A. Naumann, and T. Rezk. Deriving an Information Flow Checker and Certifying Compiler for Java. In *27th IEEE Symposium on Security and Privacy*, pages 230–242, Oakland, CA, USA, May 2006.

[8] G. Barthe and T. Rezk. Non-interference for a JVM-like Language. In M. Fähndrich, editor, *Proceedings of 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementatio (TLDI 2005)*, pages 103–112, Long Beach, CA, USA, Jan. 2005. ACM Press.

[9] M. Bartoletti, P. Degano, and G. L. Ferrari. Static Analysis for Stack Inspection. In *Proceedings of International Workshop on Concurrency and Coordination, Electronic Notes in Theoretical Computer Science*, volume 54, Amsterdam, The Netherlands, 2001. Elsevier.

[10] F. Besson, T. Blanc, C. Fournet, and A. D. Gordon. From Stack Inspection to Access Control: A Security Analysis for Libraries. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW-17 2004)*, pages 61–75, Pacific Grove, CA, USA, June 2004. IEEE Computer Society.

[11] M. Bishop and M. Dilger. Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[12] D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[13] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[14] Eclipse Project, `http://www.eclipse.org`.

[15] Equinox Java Security Project, `http://www.eclipse.org/equinox/incubator/security/java2security.html`.

[16] J. S. Fenton. Memoryless Subsystems. *The Computer Journal*, 17(2):143–147, 1974.

[17] J. S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2002)*, pages 1–12, Berlin, Germany, June 2002.

[18] C. Fournet and A. D. Gordon. Stack Inspection: Theory and Variants. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 307–318, Portland, OR, USA, Jan. 2002. ACM Press.

[19] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, USA, May 1982. IEEE Computer Society Press.

[20] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, USA, Dec. 1997.

[21] D. Grove and C. Chambers. A Framework for Call Graph Construction Algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, November 2001.

[22] G. L. Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. Automata-based Confidentiality Monitoring. In *Proceedings of 11th Annual Asian Computing Science Conference (ASIAN 2006)*, Tokio, Japan, Dec. 2006.

[23] C. Hammer, J. Krinke, and G. Snelting. Information Flow Control for Java Based on Path Conditions in Dependence Graphs. In *Proceedings of IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, Mar. 2006.

[24] N. Hardy. The Confused Deputy. *ACM SIGOPS Operating Systems Review*, 22(4):36–38, Oct. 1988.

[25] T. P. Jensen, D. L. Métayer, and T. Thorn. Verification of Control Flow Based Security Properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 89–103, Oakland, CA, USA, May 1999.

[26] G. A. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206, Boston, MA, USA, 1973. ACM Press.

[27] L. Koved, M. Pistoia, and A. Kershenbaum. Access Rights Analysis for Java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 359–372, Seattle, WA, USA, November 2002. ACM Press.

[28] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, USA, July 2005.

[29] A. C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1999)*, pages 228–241, San Antonio, TX, USA, Jan. 1999.

[30] D. A. Naumann. Verifying a Secure Information Flow Analyzer. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 211–226, Oxford, UK, Aug. 2005. Springer.

[31] J. Newsome and D. X. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS 2005)*, San Diego, CA, USA, Feb. 2005. IEEE Computer Society.

[32] N. Paul and D. Evans. .NET Security: Lessons Learned and Missed from Java. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC 2004)*, pages 272–281, Washington, DC, USA, December 2004. IEEE Computer Society.

[33] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 362–386, Glasgow, Scotland, UK, July 2005. Springer-Verlag.

[34] F. Pottier, C. Skalka, and S. F. Smith. A Systematic Approach to Static Access Control. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 30–45. Springer-Verlag, 2001.

[35] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, Sept. 1975.

[36] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, USA, Aug. 2001.

[37] G. Snelting, T. Robschink, and J. Krinke. Efficent Path Conditions in Dependence Graphs for Software Safety Analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(4):410–457, October 2006.

[38] A. Stoughton. Access Flows: A Protection Model which Integrates Access Control and Information Flow. In *Proceedings of the 1981 IEEE Symposium on Security and Privacy*, pages 9–18, Oakland, CA, USA, May 1981.

[39] IBM Security Workbench Development Environment for Java (SWORD4J), `http://www.alphaworks.ibm.com/tech/sword4j`.

[40] Úlfar Erlingsson and F. B. Schneider. IRM Enforcement of Java Stack Inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, CA, USA, May 2000. IEEE Computer Society.

[41] D. Volpano, C. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2-3):167–187, Jan. 1996.

[42] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, third edition, July 2000.

[43] D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: A Security Mechanism for Language-based Systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):341–378, 2000.

[44] D. S. Wallach and E. W. Felten. Understanding Java Stack Inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 52–63, Oakland, CA, USA, May 1998.

[45] T. Zhao and J. T. Boyland. Type Annotations to Improve Stack-Based Access Control. In *18th IEEE Computer Security Foundations Workshop (CSFW-18 2005)*, pages 197–210, Aix-en-Provence, France, June 2005. IEEE Computer Society.