Aspectizing Java Access Control

Rodolfo Toledo[1], Angel Núñez[2], Éric Tanter[1], Jacques Noyé[2]
[1]PLEIAD Lab, Computer Science Department, University of Chile
[2]ASCOLA Project, École des Mines de Nantes-INRIA, LINA

Abstract—It is inevitable that some concerns crosscut a sizeable application, resulting in code scattering and tangling. This issue is particularly severe for security-related concerns: it is difficult to be confident about the security of an application when the implementation of its security-related concerns is scattered all over the code and tangled with other concerns, making global reasoning about security precarious. In this study, we consider the case of access control in Java, which turns out to be a crosscutting concern with a non-modular implementation based on runtime stack inspection. We describe the process of modularizing access control in Java by means of Aspect-Oriented Programming (AOP). We first show a solution based on AspectJ, the most popular aspect-oriented extension to Java, that must rely on a separate automata infrastructure. We then put forward a novel solution via dynamic deployment of aspects and scoping strategies. Both solutions, apart from providing a modular specification of access control, make it possible to easily express other useful policies such as the Chinese wall policy. However, relying on expressive scope control results in a compact implementation, which, at the same time, permits the straightforward expression of even more interesting policies. These new modular implementations allowed by AOP alleviate maintenance and evolution issues produced by the crosscutting nature of access control.

Index Terms—Programming languages, security, aspect-oriented programming, access control.

1 Introduction

The security architecture of Java consists of several components [1]: data typing, memory management, bytecode verification, secure classloading, cryptography, secure communications, public key infrastructure, and access control. Among them, Java access control (hereafter JAC) is a cornerstone of the platform: it is in charge of ensuring that sensitive resources (*e.g.* the filesystem) are accessed only by the classes authorized to do so.

More than a decade ago, the JAC architecture evolved from a very limited sandbox model into a more flexible model based on stack inspection [2]. In the former model, classes are considered as either trusted or untrusted. This condition determines which classes are able to access sensitive resources and which classes are not. In the latter model, finer-grained control based on permissions is introduced. Permissions represent the ability to access and use a particular resource (*e.g.* a file) in a certain manner (*e.g.* read-only or read-write). Fine-grained access control in the JAC architecture allows one to assign different sets of permissions to different classes. Furthermore, *stack inspection* is used to *dynamically* examine if a sensitive operation can be performed or not.

The three main mechanisms of the JAC architecture based on stack inspection are:

- **Basic permission checking.** When a sensitive resource is about to be accessed, a call to the JAC API triggers a stack inspection algorithm, which *checks* whether all the classes in the current stack of execution possess the necessary permission to access the resource. If not, an exception is thrown. This basic behavior prevents the confused deputy problem [3] from happening, *i.e.* an untrusted class cannot lead a trusted one to access (or modify) a sensitive resource on its behalf.
- **Privileged execution.** In some scenarios, it is necessary for a class to access a sensitive resource on behalf of another –possibly untrusted– class. For this, the JAC architecture supports *privileged execution*.
- **Permission contexts.** When accessing a sensitive resource, it can be necessary for a class to use the permissions present at another point in the execution of the application. The JAC architecture provides the means to capture a *permission context* and restore it later on.

While these three mechanisms together provide a very powerful access control system, the JAC architecture suffers from modularity issues. Indeed, in order to trigger permission checking through stack inspection, an *explicit* call to the JAC architecture is necessary. As a consequence, code related to permission checking ends up scattered at each and every place where sensitive resources are accessed, tangled with other concerns. In other words, access control is a *crosscutting concern*, which pollutes and limits the good modularity of a system. Figure 1 shows all classes of the standard distribution of Java where access control is necessary[1]. The bars represent classes and the different lengths depict the size of the class in lines of code. The black parts inside the bars point out the actual places where basic permission checking is performed.

In addition to the non-modularity of permission checking, the implementation of stack inspection is itself problematic: while part of it resides in the Java libraries

Work partially funded by INRIA-CONICYT Project RAPIDS.

1. Figure obtained by using XRef. http://www.eclipse.org/ajdt/xref.

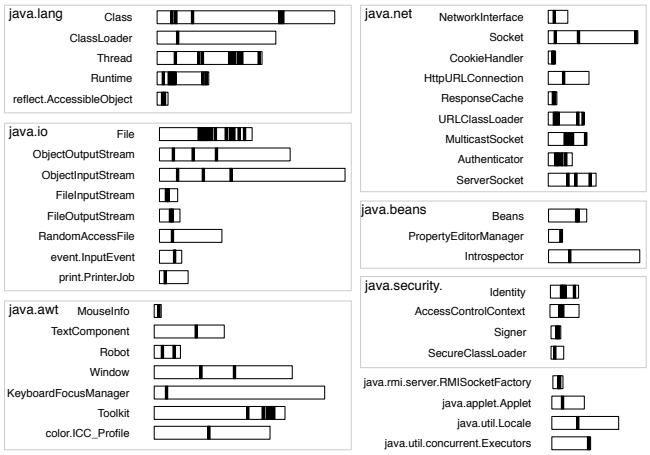

Fig. 1. Access control checks in the standard distribution of Java 1.5.

(the stack inspection algorithm), it also relies on support from the Java Virtual Machine implementation for snapshotting the current stack of execution (permission context capturing). Moreover, this native support is specific to (and can only be used for) access control enforcement.

The problems caused by crosscutting concerns in the implementation of software are well known, and are the *raison d'être* of the aspect-oriented software development community. In the particular case of access control we can mention at least three specific issues: *(1)* it is not easy to change the current access control implementation (*e.g.* to change the kind of security policies being enforced), because it is not modularly defined; *(2)* programs that do not take security into account cannot be made security-aware without directly modifying them [4]; and *(3)* forgetting to trigger access control checks at sensitive points in an application can lead to hard-to-spot security holes. If permission checks are not modularized, it is hard to reason about them globally.

To address the issue of scattered permission checks, several aspect-oriented approaches have been proposed [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]. Thanks to Aspect-Oriented Programming (AOP), access control checks can be implicitly triggered as part of the advice of aspects. The pointcuts of these aspects denote in a localized manner the different points where permission checking should happen.

While aspectizing permission checks is a valuable improvement over the existing state of affairs, none of the above-mentioned proposals address the full set of mechanisms of the JAC architecture, including both privileged execution and capturable permission contexts. Also, they rely on the existing implementation of stack inspection, which, as mentioned before, is split between a Java library and the specialized support in the JVM. In contrast, we aim to *fully* aspectize Java access control, including the implementation of stack inspection, and the three fundamental mechanisms based upon it.

In other words, this paper explores the following research questions:

- *Assuming a language with support for aspects, can we build a modularized access control architecture that is (at least) as expressive as the* JAC *architecture?*
- *If so, what general-purpose aspect-oriented language features facilitate, or enable, the definition of such an architecture?*
- *What benefits derive from the use of aspect orientation?*

We believe these questions to be of practical importance for both the language-based security and aspect-oriented programming communities, and we hope this study sheds some light on the kind of support for AOP that a modern execution environment should provide.

This study reports on our effort to define the JAC architecture entirely within AspectJ, the *de facto* standard aspect-oriented extension of Java [15]. Although we succeeded in this attempt, the result was not completely satisfactory. While basic permission checking can be defined in terms of restriction aspects using the scoping mechanisms of AspectJ, privileged execution and permission contexts require a specialized access control automata. For this reason, we tried again using the state-of-the-art proposal in terms of scope control: *scoping strategies* [16], [17]. This time, the solution fitted completely within the aspect language, providing several interesting advantages:

a) it is extremely *succinct and elegant*: the central piece, the AccessControlAspect, is only 10 lines of code.
b) it *fully supports* the three main mechanisms of the JAC architecture: basic permission checking, privileged execution, and permission contexts.
c) it is *more expressive*[2] than the JAC architecture: it supports per-object permissions (rather than just per-class), as well as more dynamic permission assignment policies.
d) it does not rely on specialized runtime mechanisms for stack inspection, but rather relies on a *general-purpose* aspect scoping construct that can be used in other domains.

The rest of this paper is organized as follows. Section 2 illustrates the use of the JAC architecture through a simple running example. Section 3 explores the use of AspectJ, highlighting why a specialized automaton for access control is necessary. Section 4 introduces scoping strategies and shows how they can be used to simply express the full JAC architecture, in a finer-grained and dynamic manner. Section 5 reflects on the advantages and future challenges derived from an aspectized access control architecture. Section 6 compares the solutions based on AspectJ and scoping strategies in detail. Section 7 discusses related work and Section 8 concludes.

2. Following [18], we say that a language *A* is more *expressive* than another language *B*, when translating a program with occurrences of a feature of *A* to *B* requires a global reorganization of the program.

2 Access Control in Java

The JAC architecture enforces a discretionary access control policy [19] in which granting access to sensitive resources depends on the identity of the requester. In this regard, the JAC architecture [1] is based on several elements, namely: protection domains, classes, permissions, and threads. A *protection domain* is defined as a set of *classes* that are loaded (by a classloader) from a certain codebase (represented by a URL). Each protection domain is granted a set of *permissions*. Through the protection domain they belong to, classes share the same set of permissions. In the JAC architecture, the smallest entity a certain permission can be granted to is a protection domain. Permissions are granted (by default) to protection domains in a separate Java policy file[3]. This is done before the application starts by the administrator of the system. In the remainder of this paper, we refer to "the permissions of the protection domain of a class" simply as "the permissions of the class" for conciseness.

The JAC architecture defines a *system* protection domain. By default, only classes in this domain are considered trusted and hence are allowed to access sensitive resources. Examples of resources accessible by the system protection domain are the filesystem, screen, mouse, and keyboard. Currently, all code shipped as part of Java belongs to the system protection domain (§4.1 [20]).

The last elements in the JAC architecture are *threads*. It is fundamental to consider them in the equation because a thread of execution can cross different protection domains. For instance, when user code wants to print some text to the standard output, the thread of execution goes from a user protection domain[4] to the system protection domain. The inverse case is also possible when a class in the system protection domain invokes a method pertaining to a class in a user protection domain (this is the case of callbacks in the Swing GUI library for instance).

For a more detailed and formal description of the JAC architecture, we refer the reader to [1], [2], [20], [21].

This section illustrates the JAC architecture through a simple incrementally-refined example.

2.1 Illustrative Example

Consider a service whose function is to periodically clean a temporary directory. This kind of service could be useful, for instance, to applications that require great amounts of disk space or to disk-cleaning applications.

Figure 2 depicts the architecture of the service: TmpCleaner is the entry point for client applications through its clean method. Each file in the temporary directory is deleted by an auxiliary CleanTask instance, which encapsulates the deletion of a particular file. The service is designed this way because if the file cannot be immediately deleted, the CleanTask instance schedules itself for later execution with a certain delay using a preexistent java.util.Timer. When the delay is over, the Timer invokes the CleanTask.run method again (the asynchronous nature of this call is denoted by the double diagonal lines at the bottom of the figure). If the file still cannot be deleted, the CleanTask schedules itself again until the deletion can be finally performed. This service exercises the three mechanisms of JAC presented in Section 1:

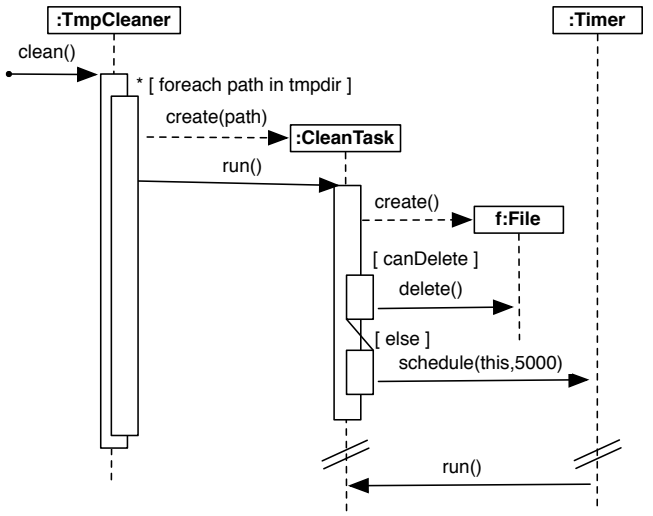

Fig. 2. Illustrative Example: a service for cleaning a temporary directory.

- **Basic permission checking.** As a first approximation, we can ensure that TmpCleaner can only be used by clients that have permissions to write to the temporary directory.
- **Privileged execution.** If we rather want to allow TmpCleaner to be invoked by any class (this is safe considering the nature of the temporary directory, and makes the service usable in a wider range of contexts), the deletion of files must be performed as a privileged execution.
- **Permission contexts.** As explained before, some deletions are postponed by scheduling CleanTask instances on a Timer. They are subsequently executed, but in a completely different context by the timer thread. This means that the permission context present at the original call to TmpCleaner.clean must be *captured* and later on *reinstalled* when the CleanTask instances are scheduled by the timer.

2.2 Basic Permission Checking

In the JAC architecture, when access to a system-sensitive resource is about to be performed, the class accessing the resource must explicitly perform a call to SecurityManager.checkPermission. This method takes as a parameter a java.security.Permission instance representing the permission to be checked. For example, here is the essence of the code in the java.io.File.delete method:

```
SecurityManager.checkPermission(
  new FilePermission(this.path, FILE_DELETE_ACTION)
);
//actual deletion goes here...
```

3. A description of this file and its syntax can be found in §3.3 of [20].

4. We use the term "user protection domain" to refer to a protection domain other that the system protection domain, potentially untrusted.

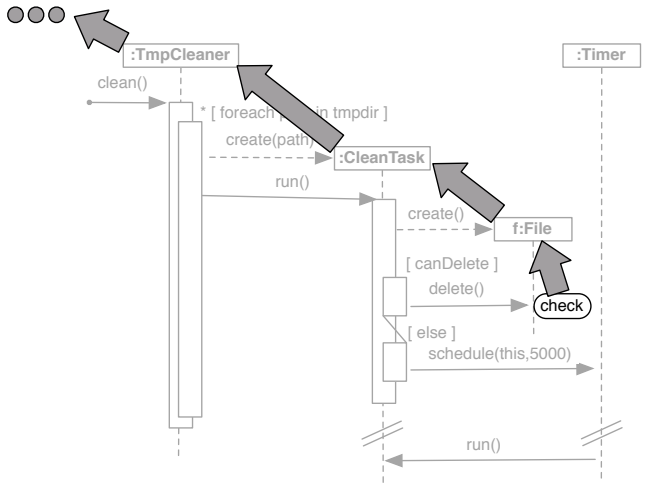

Fig. 3. Basic permission checking algorithm in the JAC architecture.

When checkPermission is invoked, the permissions of *all* classes in the stack of execution of the current thread are inspected. If one of these classes is not granted the permission to delete the file, the whole operation is canceled by throwing a java.security.AccessControlException. Following is the pseudo-code for this stack inspection mechanism[5] (m is the index of the top frame in the stack):

```
i = m;
while (i > 0) {
  if (class i does not have the permission)
    throw new AccessControlException();
  i = i − 1;
}
```

This basic access control mechanism is very useful in the most common scenario: a class in the system protection domain is being used by a class in a user protection domain. This is the situation in the running example: TmpCleaner service is not part of the system protection domain. Hence, the file deletion should be allowed only if CleanTask (the class that calls File.delete) and the other classes in the stack, *all* have been granted the corresponding permission (recall that the system protection domain is granted all permissions).

The path followed by the stack inspection algorithm can be observed in Figure 3. The arrows show the order in which the classes are inspected, starting from File, the class that triggers the basic permission checking algorithm (the circled "check" in the figure). The arrow pointing outwards of TmpCleaner denotes that its clients are also inspected.

2.3 Privileged Execution

In some cases, there is the necessity for a class to access a sensitive resource on behalf of another class. In the example, it is safe to let any class use the TmpCleaner service because only the service determines which files to delete; it cannot be told to delete other, non-temporary files (assuming that the service itself is granted access only to the temporary directory).

To allow this, the JAC architecture provides a way to perform a privileged action[6]. The semantics of privileged execution is that during the execution of the action, the permissions to consider must exclude the ones of classes in the stack above the class that started the action, but must include the ones of that class. The updated stack inspection algorithm that takes into account the execution of privileged code is:

```
i = m;
while (i > 0) {
  if (class i does not have the permission)
    throw new AccessControlException();
  if (class i started a privileged action)
    return;
  i = i − 1;
}
```

The algorithm inspects the stack class by class, just like the previous algorithm, but when it sees a class that started a privileged action, it returns normally (having checked the permission for that class before). Note that this feature does not intrinsically imply a security breach because the permissions of the entity that starts a privileged execution are maintained.

In our example, TmpCleaner should execute its complete body inside a privileged action. This way, any client will be able to use the service, regardless of its own permissions. Following is how TmpCleaner specifies which code is executed as a privileged action:

```
class TmpCleaner {
  ...
  public void clean() {
    AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
        for(String path: tmpdir){
          new CleanTask(path).run();
        }
      }});
  }}
```

The doPrivileged method starts a privileged action. The action to perform in this privileged context is specified as the body of the run method of a java.security.PrivilegedAction instance.

Figure 4 shows the sequence diagram for the previous code. The class that started a privileged action is signaled with a star (TmpCleaner), acting as a barrier through which no arrow can go. This barrier prevents the stack inspection algorithm from inspecting further classes in the stack. Executing the body of the clean method in a privileged action allows any class to use the service, no matter if it cannot write the temporary directory.

5. This algorithm and the remaining ones in this section are described in detail in the JAC architecture specification, §4 of [20]. The only difference is that in the documentation, the stack inspection algorithm continues in the stack that was active when the current thread was created (§4.3). In our case, we abstract from this implementation detail and assume a one-piece stack.

6. The word "privileged" is misleading in this context because no privilege is necessary to start a privileged action: it can be started by any class, trusted or not (§4.2.2 of [20]).

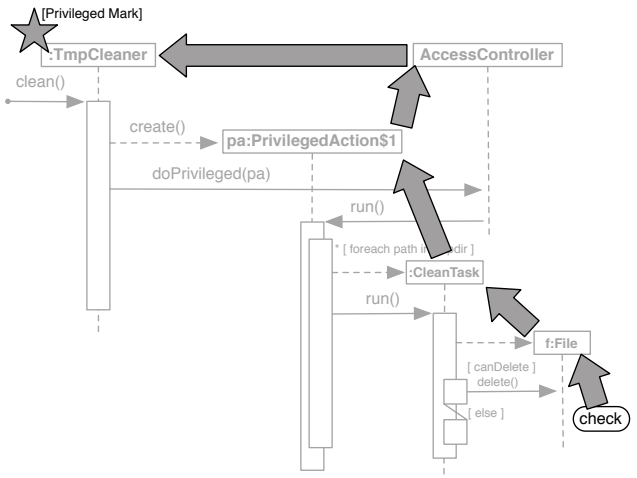

Fig. 4. Permission checking with privileged execution.

2.4 Permission Contexts

Despite the privileged action inside TmpCleaner.clean, there is a second path for basic permission checking not shown in Figure 3. This path goes from CleanTask to Timer. It is a different path because the call is asynchronous and hence, it is not in the control flow of TmpCleaner.clean. For this reason, the file deletion is not under the privileged action, so stack inspection goes over all the stack. Therefore, our intent of using the permissions of TmpCleaner when deleting files is not complete: the permission context of the original call to TmpCleaner.clean must be *captured* and used when deleting the files later on.

The use of a permission context ensures that the classes inspected in an indirect file deletion are the same than in a direct deletion. This is necessary to guarantee that changes in the permissions granted to these classes are correctly reflected in the execution.

In the JAC architecture, capturing the current permission context is done with AccessController.getContext(). We use it at the beginning of the privileged action in the clean method. The context object is essentially a reification of the current execution stack. Later, the CleanTask.run method runs the deletion of the file as a privileged action, passing the previously-captured context as an extra parameter (we modified the constructor of CleanTask to specify the context). This ensures that the deletion is done in the permission context corresponding to the TmpCleaner class: if TmpCleaner can delete files in the temporary directory, the deletion proceeds; if it cannot, an access control exception is thrown.

The code below illustrates the capture of the permission context in TmpCleaner.clean, and its subsequent use in CleanTask.run:

```
//TmpCleaner (context capture)
public void clean() {
  AccessController.doPrivileged(new PrivilegedAction() {
    public Object run() {
      AccessControlContext ctx = AccessController.getContext();
      for(String path: tmpdir){
        new CleanTask(path, ctx).run(); //pass ctx
      }
    }});
}

//CleanTask (context usage)
public void run() {
  if(canDelete()) {
    AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
        file.delete();
      }}, this.ctx);  //use ctx to delete the file
  }
  else{ /* self-scheduling on timer */ }
}
```

Following is the complete stack inspection algorithm for permission checking in Java, updated to support the use of permission contexts:

```
i = m;
while (i > 0) {
  if (class i does not have the permission)
    throw new AccessControlException()
  if (class i started a privileged action)
    if (a context was specified)
      continue inspection on context
    return;
  i = i - 1;
}
```

The semantics of passing a permission context to a doPrivileged call is that once the class that initiated the privileged action is reached, the stack inspection continues in the stack the permission context represents. If no context was supplied, the algorithm returns normally, just as in the previous case for handling privileges.

Figure 5 depicts the updated scenario. First, the permission context at the beginning of the clean method is represented as the stack of execution at that point (the involved classes are exactly the ones of Figure 4). Then, the doPrivileged call in CleanTask.run passes a reference to this permission context. When deleting a file, the captured permission context is used. This completely eliminates the second path from CleanTask to Timer.

3 Java Access Control with AspectJ

In this section we describe how to implement the three main mechanisms of the JAC architecture using AspectJ [15], the *de facto* aspect-oriented extension to Java.

3.1 Aspect-Oriented Programming

Aspect-oriented programming makes it possible to modularize crosscutting concerns, like access control. Specifically, in the pointcut-advice (PA) model for aspect-oriented programming [22], [23], crosscutting behavior is defined by means of pointcuts and advice. Execution points at which advice may be executed are called (dynamic) join points. A pointcut identifies a set of join points, and a piece of advice is the action to be taken at a join point matched by a pointcut. An aspect is a module that encompasses a number of pointcuts and pieces of advice.

Following is the shape of an aspect in AspectJ, which follows the PA model:

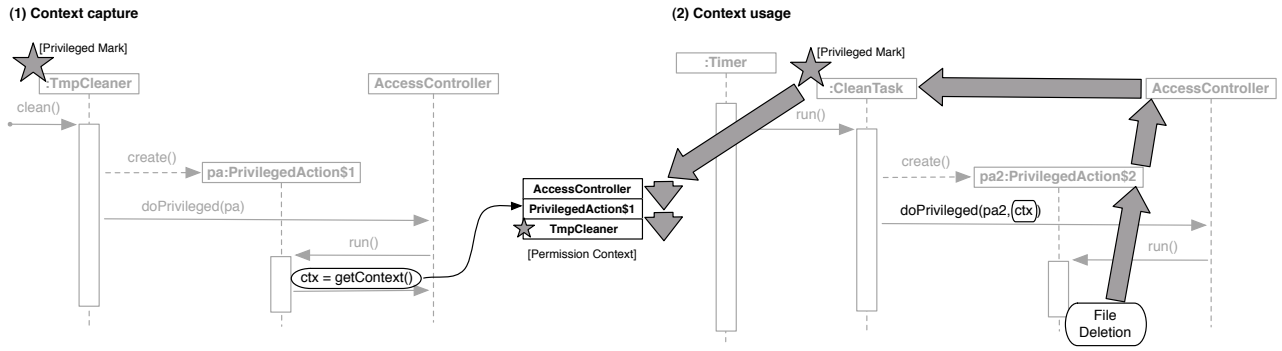

Fig. 5. Permission checking with permission context capture (1) and further usage (2).

```
aspect AspectExample {
  pointcut pc(): ... //predicate selecting join points

  before(): pc() {
    //action to take before selected join point execution
  }}
```

The aspect AspectExample declares one pointcut and one piece of advice. The pointcut pc selects the join points of interest, typically using some kind of pattern matching (*e.g.* call(void Point.set*()) for all calls to setters in Point). Pointcuts can also expose contextual information such as the target of a call (target(..)) or the currently executing object (this(..)). The piece of advice (before(): ...) declares that *before* each join point of interest, the specified action should be performed. A piece of advice can also be declared as after or around to execute them after or instead of matching join points, respectively. In the latter case, the piece of advice can use the construct proceed() to execute the original behavior associated to the join point, if so desired.

3.2 Factoring out Access-Control Checks

Through the PA model, it is straightforward to modularize access control checks. It is only necessary to define an aspect whose pointcut identifies accesses to sensitive resources, and whose advice triggers stack inspection:

```
aspect FileDeletionPermission {
  pointcut resourceAccess(File file):
      execution(boolean File.delete()) && target(file);

  before(File file): resourceAccess(file) {
    SecurityManager.checkPermission(
      new FilePermission(file.path, FILE_DELETE_ACTION)
    );
  }}
```

Deploying this aspect is equivalent to performing the explicit invocation to SecurityManager.checkPermission in File.delete (Section 2.2). However, the fundamental advantage of the aspect-oriented approach is that explicit calls to SecurityManager.checkPermission are not necessary anymore. As long as one aspect per kind of permission is defined, the calls end up modularized in these aspects.

This approach is possible because the AspectJ pointcut language is expressive enough to identify all points where access control is necessary. Although further study is required for the general case, we have conducted a study for the standard Java distribution, which shows that it is possible to specify pointcuts for all occurrences[7]. This by-product of our study constitutes, to the best of our knowledge, an original contribution.

3.3 Stack Inspection Semantics via Scope Control

Most prior proposals for the modularization of access control in Java uses the aforementioned approach for factoring out access-control checks [5], [6], [7], [8], [9], [10], [11], [13], [14]. However, this way of aspectizing access control still relies on the non-modularized and specialized stack inspection implementation of Java. It is interesting to explore if aspects can allow for a fully modular JAC architecture. To do so, we need to express the stack inspection semantics via another mechanism.

Before we go into details on how to implement stack inspection in a modular way, let us describe the kind of aspects we will need to use for the task.

Aspects like FileDeletionPermission simply delegate access control decisions to the default stack inspection algorithm. This algorithm ensures that all classes in the stack have the corresponding permission when a resource is about to be accessed. Consequently, we call these aspects *permission aspects*. However, since the default implementation of stack inspection cannot be part of a modularized solution (because it is not modularly defined), we need another kind of aspects based on a different mechanism for access control enforcement: *restriction aspects*. A restriction aspect, instead of invoking SecurityManager.checkPermission in its advice, throws an exception as soon as it *sees* the resource access its pointcut identifies:

```
aspect FileDeletionRestriction {
  pointcut resourceAccess(File file):
      execution(boolean File.delete()) && target(file);

  before(File file): resourceAccess(file) {
    if(file.path.equals("..")) {
      throw new AccessControlException();
    }
  }}
```

7. The complete list can be found at http://pleiad.cl/research/3sec

Because a restriction aspect throws an exception when its pointcut matches a join point, its *scope* must be limited so that it only sees illegal resource accesses. In other words, its scope must be controlled. In this section we explore how the scoping features of AspectJ can be used for this purpose.

3.4 Aspectizing Basic Permission Checking

Determining whether a resource access is legal or not can be considered a matter of *flow of execution*: if the access is in the control flow of a class with a restriction to access the resource, then the access is illegal. Otherwise, the access is legal.

Aspect languages usually provide means to reason about control flow: aspects can be limited to see only the join points that occur within certain flows of execution. AspectJ, in particular, provides two options in this regard. First, AspectJ features percflow deployment, which creates and deploys a new aspect instance for each flow of execution starting in certain join points (specified using a pointcut). Second, AspectJ also features a cflow pointcut, which matches join points only under a particular flow of execution. A cflow pointcut can be conjuncted with other pointcuts as an extra condition over the join points to select. Using these AspectJ control flow features to limit the scope of restriction aspects, implementing basic permission checking is straightforward. Following is the FileDeletionRestriction aspect, updated so that it only sees join points in the control flow of the execution of any method in UntrustedClass. The code includes both percflow and cflow implementation options (*(a)* and *(b)* resp.), but only one is necessary[8].

```
aspect FileDeletionRestriction
        percflow(execution(* UntrustedClass.*())) (a) {

  pointcut resourceAccess(File file):
      execution(boolean File.delete()) && target(file);

  before(File file): resourceAccess(file) &&
                     cflow(execution(* UntrustedClass.*())) (b) {
    if(file.path.equals("..")) {
      throw new AccessControlException();
    }
  }}
```

3.5 Privileged Execution

The previous implementation of basic permission checking using either percflow or cflow is simple and direct. However, it cannot be used for realizing privileged execution. Once in a privileged execution, restriction aspects must not see a resource access even if it occurs in the control flow of a method of an untrusted class. In the case of percflow, once a restriction aspect has been deployed, it cannot be undeployed for the extent of a privileged execution, which is exactly what is needed. The case for cflow is similar. This problem is even more evident if we consider multiple privileged executions, initiated by different classes, with a mixture of trusted and untrusted classes involved in the stack.

It turns out that the scoping mechanisms offered by AspectJ does not suffice to express the JAC architecture stack inspection semantics. For this reason, it becomes necessary to manually manage state upon which to decide whether a resource access is legal or not. In the following, we detail a solution that maintain access control state through a *pushdown automaton*.

3.6 Access Control Automaton

In the following, we maintain access control state through a *pushdown automaton*. This approach corresponds, in essence, to various proposals for stack inspection alternatives [24], [25], [26], [27]. The novelty is that in this work we use an aspect-oriented approach for updating the automaton. With the use of this *access control automaton*, restriction aspects are deployed with *global scope* and must check the automaton to decide when to throw an exception.

Each access control automaton[9] has two logical states: LEGAL and ILLEGAL, representing legal and illegal resource accesses, respectively. For a given automaton, it is in state LEGAL when all classes in the stack of execution have the permission to access the resource it guards. It is also in state LEGAL when a privileged execution (started by a trusted class) is in progress. If none of these conditions hold, the automaton is in state ILLEGAL. Figure 6 depicts the access control automaton for privileged execution[10]. This automaton is updated each time a (trusted or untrusted) class enters and exits the stack; and also when entering and exiting a privileged action. Transitions are prefixed with enter/exit and suffixed Trusted/Untrusted/DoPriv for this purpose. The stack of the access control automata is used to remember from which state an enter* transition originated, so that the automaton can correctly switch back to the corresponding state on an exit* event. The symbols in the stack can be: L, I, and *I; for "coming from LEGAL", "coming from ILLEGAL", and "coming from ILLEGAL, but the last class was trusted" respectively. The *I symbol is useful to distinguish a privileged execution started by a trusted class and one started by an untrusted class. The _ symbol is used to ignore what is on the top of the stack (when specified in the second position of transitions), and to prevent a push operation from occurring (when specified in the third position of transitions).

The implementation of the FileDeletionRestriction aspect supporting privileged execution, using and updating its access control automaton is presented in Figure 7.

8. The difference between these options is that percflow also controls the instantiation of aspects: each time a method of UntrustedClass is executed, a new aspect instance is created (if not already in the control flow of another UntrustedClass method execution).

9. We consider one access control automaton per restriction aspect for convenience and practical reasons: a global automaton would have an exponential number of states: 2^n, where n is the total number of restrictions.

10. We tested this automaton with JFlap. http://www.jflap.org.

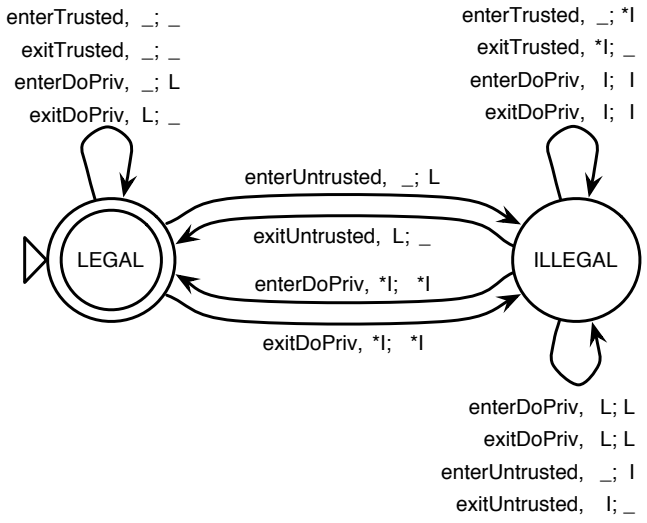

Fig. 6. Pushdown automaton for privileged execution. Transitions have the form: <*event*>,<*value on top of the stack*>;<*value pushed onto the stack*>.

```
aspect FileDeletionRestriction {
  //resource access pointcut and advice
  pointcut resourceAccess(File file): ... ;
  before(File file): resourceAccess(file) {
    if(!automaton.inLegalState()) { //scope check
      if(file.path.equals("...")) {
        throw new AccessControlException();
      } }
  }
  //automaton handling
  private static InheritableThreadLocal<AC_Automaton>
    automaton = new InheritableThreadLocal<AC_Automaton>() {
      protected synchronized AC_Automaton initialValue() {
        return new AC_Automaton();
      }
      protected AC_Automaton childValue(AC_Automaton parent) {
          return new AC_Automaton(parent);
      } };

  pointcut untrustedMethods(): ... ;

  Object around(): untrustedMethods() {
    automaton.get().enterUntrusted();
    try { return proceed(); }
    finally { automaton.get().exitUntrusted(); }
  }
  Object around(): !untrustedMethods() { //trusted
    automaton.get()enterTrusted();
    try { return proceed(); }
    finally { automaton.get().exitTrusted(); }
  }
  Object around(): call(* AccessController.doPrivileged(..)) {
    automaton.get().enterDoPriv();
    try { return proceed(); }
    finally { automaton.get().exitDoPriv(); }
  }}
```

Fig. 7. FileDeletionRestriction and its access control automaton

The top part of the figure shows the pointcut and advice of the restriction aspect. The only variation there is the check to determine if the automaton is in an ILLEGAL state, in which case an exception is thrown. The bottom part of the figure shows the code for updating the automaton. The untrustedMethods pointcut selects the set of methods belonging to classes that have the corresponding restriction. Each time one of those methods is executed, the automaton is updated accordingly. The same process is performed for trusted classes (the methods not selected by untrustedMethods), and when doPrivileged is executed. The try/finally blocks are necessary to update the automaton even when proceed throws an exception. Finally, the automaton is accessed through an InheritableThreadLocal variable, which allows each automaton to correctly reflect the access control state of the corresponding thread.

By using these updated restriction aspects, the JAC architecture semantics for privileged execution can be implemented. However, the solution combines AspectJ constructs and a specialized access control automata. Actually, each access control automaton rebuilds its own view of the execution stack.

```
class PermContext {
  public static Map<Class, AC_Automaton> getContext() {
    Map<Class, AC_Automaton> context =
                          new HashMap<Class, AC_Automaton>();

    context.put(FileDeletionRestriction.class,
                FileDeletionRestriction.getContext());
    // put automaton instances for other restrictions
    ...
    return context;
  }}
```

Fig. 8. Permission context implemented as a map.

3.7 Permission Contexts

In order to support permission contexts, the access control automata can be extended with the necessary infrastructure for snapshotting and reinstalling a permission context.

On the JAC side, a permission context is represented by the set of classes in the current thread of execution. This is so because the stack inspection algorithm needs the execution stack to tell whether a resource access is legal or not. On the access control pushdown automaton side, the current states and stacks of the automata always instantaneously distinguishes between a legal and an illegal resource access. Therefore, the permission context at an execution point can be considered as the set of access-control-automaton instances at that point.

In implementation terms, a permission context maps restriction aspects and automaton instances. As shown in Figure 8, the getContext method of the PermContext class can be used to get this map. It uses the getAutomaton method provided by each restriction aspect to get a copy of the current automaton instance.

When a piece of code is going to be executed under a given context, each restriction aspect needs to change its automaton instance by the respective instance in the given context. The current automaton instance needs to be saved in order to be restored after the piece of code is executed. Since contextual execution can be nested, the automaton instances need to be stored in a stack. The FileDeletionRestriction aspect of Figure 7 is extended to support permission contexts as Figure 9 shows. First,

```
aspect FileDeletionRestriction {
  /* code of Figure 7 */

  Object around(PrivilegedAction action, PermContext context):
           call(* AccessController.doPrivileged(..,..)) &&
           args(action,context){

    stack.get().push(automaton.get());
    automaton.set(context.get(getClass()));
    try {
      return proceed(action,context);
    }
    finally {
      automaton.set(stack.get().pop());
    }}

  public static AC_Automaton getContext() {
    return new AC_Automaton(automaton.get());
  }

  private static InheritableThreadLocal<Stack<AC_Automaton>>
    stack = new InheritableThreadLocal<Stack<AC_Automaton>>(){
      protected synchronized Stack<AC_Automaton> initialValue(){
        return new Stack<AC_Automaton>();
      }

      protected Stack<AC_Automaton>
                childValue(Stack<AC_Automaton> parent) {
        Stack<AC_Automaton> stack = new Stack<AC_Automaton>();
        stack.addAll(parent);
        return stack;
      }
    };}
```

Fig. 9. FileDeletionRestriction and its access control automaton with support for permission contexts.

the aspect is equipped with the getAutomaton method returning a copy of the current automaton instance. Second, a new piece of advice is defined that intercepts a contextual execution and changes the current automaton instance by the respective automaton instance given in the passed context. Third, the aspect is provided with a stack variable implemented as an InheritableThreadLocal object, which is used to keep the automaton instances that need to be restored after contextual executions.

3.8 Summary

The implementation presented in this section constitutes, to the best of our knowledge, the first modular realization of the JAC architecture entirely based on aspect orientation. However, the solution is not completely satisfactory. While basic permission checking can be defined in terms of restriction aspects using the scoping mechanisms of the AspectJ language, privileged execution and permission contexts require these mechanisms to be replaced with a specialized access control automaton.

The reason for falling back to an implementation based on access control automata is that AspectJ only supports a limited notion of scoping of aspects. First, it is impossible to undeploy an aspect once it has been deployed using percflow, which is necessary for privileged execution. Second, using cflow, it is not possible to distinguish a legal resource access from an illegal one once in the control flow of multiple privileged executions.

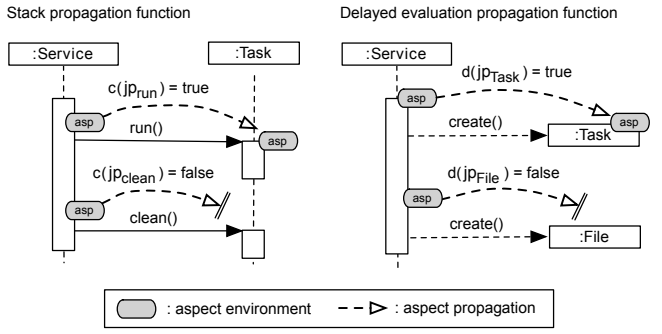

Fig. 10. Propagation of aspects with scoping strategies

4 Java Access Control with Scoping Strategies

In this section we show how scoping strategies [16], [28] can be used to define the JAC architecture in an aspect-oriented manner by controlling the scope of restriction aspects. This contrasts with the AspectJ solution, which is based on *global* deployment of restriction aspects and explicit scope control through access control automata.

After a brief introduction to scoping strategies, we explain how a simple scoping strategy allows an elegant specification of the three main mechanisms of the JAC architecture (basic permission checking, privileged execution, and permission contexts). We then describe in details why such a simple definition effectively meets the requirements.

4.1 Scoping Strategies in a Nutshell

Scoping strategies permit fine-grained control over the scoping semantics of a deployed aspect [16], [28]. The "scope" of an aspect is the set of join points that the aspect potentially matches, *i.e.* against which its pointcuts are evaluated. A scoping strategy itself is specified by two *propagation functions*: a *call stack* propagation function c specifies how an aspect propagates along with method calls, and a *delayed evaluation* function d specifies whether or not an aspect is "captured" in objects when they are created[11]. Intuitively, the former allows controlling dynamic scoping of aspects, stopping propagation when a certain condition is met. Dynamic scoping of aspects is, for instance, found in CaesarJ [29], where the construct deploy(asp){block} dynamically deploys the aspect asp along the execution of the block block. The latter allows aspects to follow certain objects, similar to per-instance aspects in AspectJ and CaesarJ: the aspect sees join points occurring *lexically* within all methods of the object. By analogy with the construct deploy, we assume that the construct deployOn(asp){obj} deploys the aspect asp on the object obj.

Propagation functions are themselves pointcuts, *i.e.* they are predicates over join points: c matches *call* join points for which the aspect should propagate, while d matches *object creation* join points. Figure 10 depicts

11. Scoping strategies also include a third component, called *activation function*. Activation is not used here, so we omit it.

propagation of an aspect asp deployed with the scoping strategy s = [call(* Task.run()), target(Task)][12]. If asp is currently deployed (*i.e.* it is in the current aspect environment), it propagates on calls to run (jp_{run}) but not on calls to clean (jp_{clean}). Therefore asp sees join points occurring during the execution of run. Similarly, it gets captured in new Task objects (jp_{Task}), and not in new File objects (jp_{File}). This means that it sees the subsequent activity of these Task objects. We assume here that scoping strategies are provided as additional arguments to the deployment constructs: deploy[s](asp){block} and deployOn[s](asp){obj}, where s is a scoping strategy.

Many examples of scoping strategies have been formulated elsewhere, for both local aspects [16] and distributed aspects [17], as well as for variable bindings [28].

4.2 The Scoping Strategy for Java Access Control

The specification of the JAC architecture by means of scoping strategies is succinct and elegant. Figure 11 presents the AccessControlAspect aspect, in charge of deploying restriction aspects. The underlying idea is to deploy restriction aspects on objects with a proper scoping strategy, so that the relevant parts of the activity of the objects is under control of the restriction aspects.

Deploying restriction aspects. The AccessControlAspect aspect deploys restriction aspects *on objects* when they are created. First, the init pointcut matches all object creations (line 3), binding the newly-created object to the instance parameter (line 4). The associated advice (lines 13-16) deploys the corresponding restriction aspects on that object, using deployOn (line 16). The set of restriction aspects that corresponds to a particular object is determined by the getRestrictionsFor method (line 14). This method encapsulates the binding of objects to their protection domain. One possible implementation is to mimic the JAC architecture by returning the restriction aspects that correspond to the permissions declared in the Java policy file. Another implementation would be to return restrictions based on dynamic conditions, such as the kind of user currently interacting with the application, as in role-based access control [30].

Access control scoping strategy. Restriction aspects are deployed on objects (line 16) with the *access control scoping strategy* defined in lines 7-11. The call stack propagation function of the strategy expresses both basic permission checking and privileged execution, as will be explained in Section 4.3 and 4.4, respectively. Essentially, it specifies that a restriction aspect always propagates on the call stack, except on privileged calls. The delayed evaluation propagation function expresses the capture of permission contexts. It ensures that restriction aspects propagate to AccessControlContext instances; therefore, creating such an object is a means to take a snapshot of the restriction aspects present at that point in time. This is explained in Section 4.5.

12. We use the concrete syntax [c,d] for deployment strategies, where c and d are defined similarly to AspectJ pointcuts.

```
1  aspect AccessControlAspect {
2    //creation of objects
3    pointcut init(Object instance): execution(*.new(..))
4                                    && this(instance);
5
6    //access control scoping strategy
7    ScopingStrategy acss = [
8      !(call(* Object.doPrivileged(..)) &&
9         if(jp.getTarget() == jp.getThis())),
10     target(AccessControlContext)
11   ];
12
13   before(Object instance): init(instance) {
14     Aspect[] restrictions = getRestrictionsFor(instance);
15     //per-instance deployment
16     deployOn[acss](restrictions){ instance };
17   }}
```

Fig. 11. Access control aspect for deploying restriction aspects.

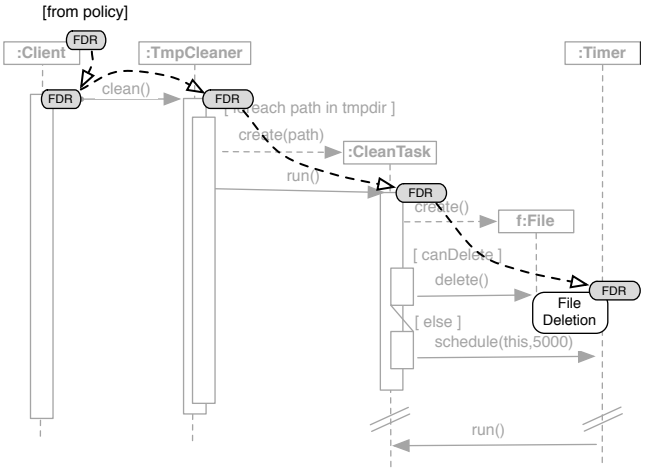

Fig. 12. Restriction aspects propagation for basic permission checking

4.3 Basic Permission Checking

Basic permission checking dictates that accesses to system-sensitive resources must be denied if a class on the stack does not have the corresponding permission. The call stack propagation function of the access control scoping strategy (lines 8-9 of Figure 11), in the absence of privileged execution (*i.e.* no calls to the doPrivileged method), always evaluates to true (line 9 is explained in Section 4.4). In consequence, the restriction aspects of the objects participating in the stack unconditionally propagate through all method invocations: they see all actions that are about to be performed, and can deny them as appropriate.

Figure 12 shows how the FileDeletionRestriction (FDR) aspect is propagated on method invocations in the TmpCleaner service. For the sake of clarity, we consider FDR to be the unique restriction aspect in the system. The only untrusted object, with FDR deployed on it, is the client that invokes TmpCleaner.clean; all remaining

instances[13] are considered trusted. The figure shows how FDR unconditionally propagates from the initial call to TmpCleaner.clean until the point of the actual file deletion, following the semantics of the stack propagation function.

4.4 Privileged Execution

Recall from Section 2.3 that only the restrictions associated with the entity that initiates a privileged action and the subsequent ones must be enforced, while those associated with entities higher in the call stack are ignored.

In terms of scoping strategy for restriction aspects, this means that restriction aspects must not see resource accesses in the dynamic extent of the privileged action: they should not propagate when a privileged action starts executing. However, the restriction aspects associated with the entity initiating the privileged action must be present during the execution of the action.

We introduce a special doPrivileged method for privileged code execution, similar to the doPrivileged static method of the AccessController class in the JAC architecture. The semantics is equivalent except that this method is now defined on Object, as an *instance* method:

```
class Object {
  ...
  public final Object doPrivileged(PrivilegedAction action){
    return action.run();
  }}
```

In order to support the requirements of privileged execution stated above, doPrivileged is only effective on *self calls*. When called on this—and only in that case—the current restriction aspects do not propagate. This is specified by the call stack propagation function of the access control scoping strategy shown on Figure 11, line 8-9: propagation on the call stack stops upon calls to Object.doPrivileged only if jp.getThis() == jp.getTarget() (*i.e.* it is a self call). Note that because doPrivileged is an instance method of the object that is initiating the privileged action, restrictions associated to that object (as specified by the policy file) *do* propagate.

To illustrate this point better, let us extend our scenario by assuming that the policy file specifies that TmpCleaner is restricted to writing to the tmp directory. Figure 13 shows how the FileDeletionRestriction (FDR) aspect is propagated on method invocations in the TmpCleaner service, and then stopped by the call to doPrivileged. However, the restriction deployed on TmpCleaner (called TOR for "Temp Only Restriction") propagates to the execution of doPrivileged (as for any instance method), and subsequently propagates during the evaluation of the action. Eventually, the file deletion is allowed, because only TOR is present.

13. We assume that all method calls are performed over an object. Static methods still fit the model if we convert them into instance methods or consider classes as objects.

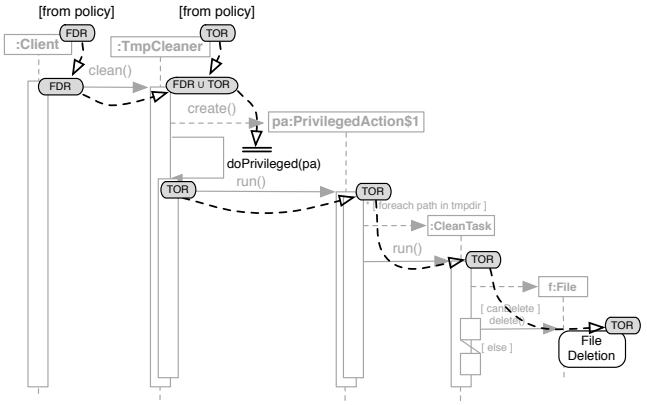

Fig. 13. Restriction aspects propagation for permission checking with privileged execution

4.5 Permission Contexts

Recall from Section 2.4 that using permission contexts involves two processes: context capture and context usage. Once a permission context is captured (by calling AccessController.getContext()), it can be reinstalled by performing a privileged execution with the context as a parameter. The semantics of using a context for a privileged execution stipulates that, in addition to the restrictions of the entity starting the privileged action (as in the previous section), the restrictions enforced also include the restrictions of the permission context.

In terms of the scoping strategy for restriction aspects, this means that it should be possible to capture the restrictions present at some point in time, associate them with a permission context, and later on use them in a privileged execution. Let us first detail how permission contexts are captured and then, how to reinstall them.

Capturing restrictions in permission contexts is handled by the scoping strategy: the delayed evaluation propagation function permits restriction aspects to propagate to all newly-created AccessControlContext instances (Figure 11, line 10). In consequence, creating such an object captures in that object all the current restriction aspects, *i.e.*. the current permission context:

```
AccessControlContext ctx = new AccessControlContext();
```

In order to reinstall the restriction aspects propagated to an AccessControlContext instance, we introduce an overloaded version of the doPrivileged method, taking a context as a second parameter, just like in the original JAC architecture:

```
public final Object doPrivileged(PrivilegedAction action,
                                  AccessControlContext context){
  return context.run(action);
}
```

The method calls the run method on the context object to include it in the stack, thereby ensuring that all captured restriction aspects are reinstalled, and consequently propagate on the stack. The AccessControlContext class is simply defined as follows:

```
public final class AccessControlContext{
  public final Object run(PrivilegedAction action){
```

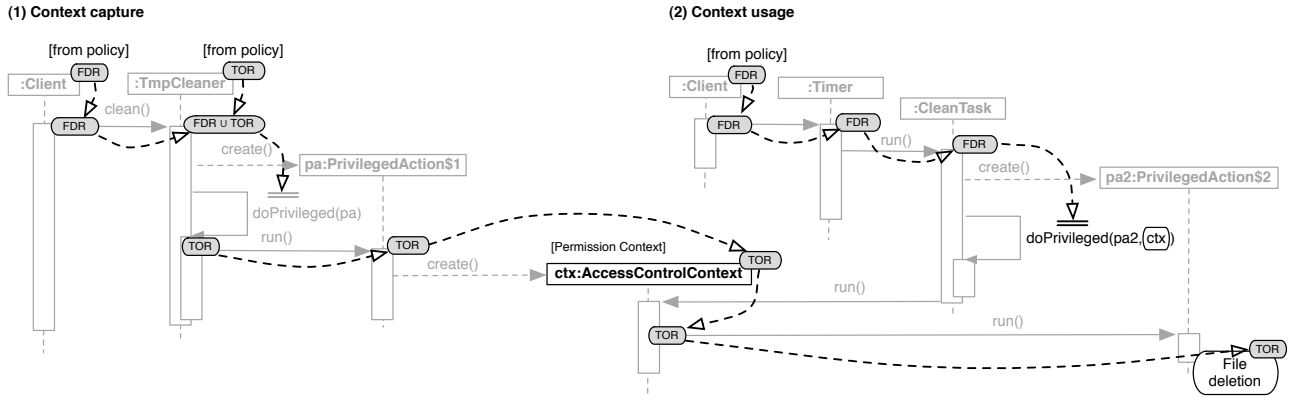

Fig. 14. Restriction aspects propagation for permission context capture (1) and further usage (2).

```
    return action.run();
  }}
```

Note that for the same reasons exposed in Section 4.4, the restriction aspects of the object that invokes doPrivileged are present in the doPrivileged body. Therefore the adequate set of restrictions (the union of the restrictions in the object initiating the privileged action and the restrictions in the context object) is reestablished.

To illustrate this process, let us again extend our scenario by considering that the entity that starts the timer also has a FileDeletionRestriction. Figure 14 shows how the FDR and TOR aspects are propagated in method invocations in the TmpCleaner service. Both aspects are stopped at doPrivileged calls, but TOR is captured on an AccessControlContext instance that is later on used by CleanTask to start a privileged action. This results in the inclusion of TOR in the stack again. At the point of the file deletion, the only restriction aspect is TOR, so the deletion is eventually allowed.

5 Evaluation of Aspectized Access Control

In this section we discuss some of the advantages and disadvantages of the aspect-oriented implementations of the JAC architecture presented in Sections 3 and 4. We consider three viewpoints in this analysis, namely modularity, security, and expressiveness.

5.1 Modularity

An aspect-oriented solution implies two major advantages in terms of modularity. First, looking at pointcut declarations immediately reveals where access control is being enforced, without having to navigate the code to find the relevant points. In consequence, it is *difficult* to miss security-sensitive operations because they are specified in a single place. Second, access control is defined exclusively in *dedicated modules*: the restriction aspects. Consequently, there is no scattering and tangling of the access control concern.

One issue that may appear when adding access control to a (probably unaware) application is how it handles the exceptions thrown on illegal resource accesses. The problem is that without considering access control, applications are not prepared for the unexpected scenario of failing to access a resource. Exception handling can be seen as another concern, and means have been proposed to deal with it in an aspect-oriented manner [31], [32].

Modularizing privileged execution (calls to doPrivileged) presents a similar challenge: certain pieces of code (bodies of privileged actions) must be handled specially with respect to the access control state of the application. Having privileged execution modularized is important to fully achieve the objective of a completely modular specification of access control. We plan to explore this in future work.

5.2 Security

Turning checkPermission calls into restriction aspects prevents malicious aspects from interfering with access control. For example, consider the following aspect:

```
aspect MaliciousAspect {
  pointcut accessControl():
      call(void SecurityManager.checkPermission(..));
  void around(): accessControl() {}
}
```

This aspect selects all calls to checkPermission and does nothing instead, thus completely annihilating access control. Restriction aspects solve this problem because they provide an implicit way of performing access control checks: having no explicit calls to checkPermission means no join points for malicious aspects to match on.

However, some aspect languages permit aspects to observe not only the computation of the base application, but also the computation of other aspects. For instance, AspectJ provides an adviceexecution pointcut to select advice execution. Using this pointcut, a malicious aspect can interfere with access control by skipping some or all advice executions. This means that advice of restriction aspects can be prevented from throwing exceptions on illegal resource accesses, thus eliminating access control.

A couple of tentative solutions to this issue, mainly based on further limiting the scope of aspects, can be devised. A first alternative is to use the notion of levels of execution, as in stratified aspects [33] and execution

levels [34]. In a nutshell, the idea is to structure computation into levels, starting with base computation at level 0. Aspect computation is by default considered as occurring at level 1, and is therefore invisible to other aspects. If needed, aspects can be deployed at higher levels of execution, thereby possibly observing other aspects. In this model, restriction aspects can be deployed on a sufficiently-high level so that no other aspect can see (or interfere with) their activity. A second alternative is to classify aspects into user (untrusted) and kernel (trusted) categories, where user aspects cannot interfere with kernel aspects. For instance, join points derived from computation of kernel aspects can be hidden from user aspects. We intend to address this issue in future work.

5.3 Expressiveness

An aspectized access control architecture makes it possible to easily express useful policies, such as the Chinese wall policy [35], which is based on the history of resource accesses.

The aim of the Chinese wall policy is to avoid conflicts of interest in decision making. In the Chinese wall model, sensitive resources are called *objects* (*e.g.* a financial report); an object pertains to a *company dataset* (*e.g.* an oil company); and a company dataset pertains to a *conflict of interest class* (*e.g.* oil companies). The idea is that once an entity accesses a certain object, it must not longer be able to access other objects in a different company dataset within the same conflict of interest class (*e.g.* financial reports of two different oil companies). This way, the entity cannot combine information from one conflict of interest class to make strategic decisions.

The Chinese wall policy is known to be difficult to implement using the JAC architecture because "[..] the JVM would need not only to monitor object interaction but also keep a history of it" (§7.2 of [1]). However it can be directly defined using a restriction aspect:

```
aspect ChineseWallRestriction {
  List accessedCompanies = new ArrayList();

  pointcut objectAccess(Object object):
      /* accesses to objects */ && target(object);

  before(Object object): objectAccess(object) {
    Company company = companyOf(object);
    if(accessedCompanies.contains(company)){
      return; //OK if the company was already accessed
    }
    for(Company c: accessedCompanies){
      //accessing an object within the same COI class?
      if(coiOf(company) == coiOf(c)){
        throw new AccessControlException();
      }
    }
    accessedCompanies.add(company);
  }}
```

The key in this implementation is that aspects intrinsically monitor the execution of the system. This contrasts with the architecture based on explicit permission checking of Java.

6 Access Control Automata vs Scoping Strategies for Access Control

In this section we present a comparison between access control using AspectJ and access control using scoping strategies. Although a solution based on AspectJ is possible in order to implement the JAC architecture, the use of scoping strategies brings many benefits, from both a conceptual and a practical viewpoint.

6.1 General Purpose vs Domain Specific

In terms of the infrastructure necessary to support a modularized JAC architecture, both solutions are similar. For the AspectJ solution, a per-thread automaton is necessary to keep track of the current access control state of the stack given the restrictions of entities participating in it. For the scoping strategies solution, a per-thread aspect environment is required to maintain the set of restriction aspects currently in scope. However, a fundamental conceptual difference between both approaches is whether they can serve purposes other access control. On the one hand, the automata infrastructure is specific and therefore, only useful for this problem. On the other hand, the aspect environments infrastructure is general-purpose and can be used for much more than access control. Examples where dynamically-deployed aspects with proper scope are useful are manifold [16], [17]. Actually, scoping strategies are generally applicable, including to other kinds of adaptations beyond aspects [28].

6.2 Expressiveness

In this section we compare both solutions in terms of expressiveness. By expressiveness, we refer to the fact that a feature that is directly expressible with dynamic deployment and scoping strategies such as per-object restrictions requires additional data structures and several modifications to be implemented in AspectJ. The reason for this, as detailed before in Section 3, is the limited scoping control in the language.

6.2.1 Increased Dynamism

Dynamic aspect deployment allows greater control over where and when an aspect affects a program [16]. While the AspectJ solution is limited to static deployment and refining the scope of aspects necessarily implies modifying the access control automata, the scoping strategies solution can take advantage of dynamic deployment. In particular, dynamically deploying AccessControlAspect introduces the possibility to activate and deactivate access control at will, depending on runtime conditions in the application.

In the original JAC architecture, all classes are subject to access control. Even classes in the system protection domain are checked by the stack inspection algorithm! The reason is that access control is global, it cannot be deactivated[14]. Analogously, in the AspectJ implementation,

14. A call to doPrivileged does not deactivate access control. It only puts a mark denoting that stack inspection must stop there.

restriction aspects are active even when the automaton state is legal. Again, this is due to the global scope of restriction aspects in that implementation.

In the scoping strategies implementation of the JAC architecture, activating (resp. deactivating) access control is just a matter of deploying (resp. undeploying) AccessControlAspect[15]. In consequence, through dynamic deployment developers can specify in which execution flows of the program restriction aspects are deployed on objects, and hence, which objects are subject to access control. Examples of scenarios where the conditional activation/deactivation of access control can be useful are:

- **Unconditional Activation**. When transferring control flow to methods on unknown, dynamically-loaded classes: execution of these methods is always untrusted, therefore, AccessControlAspect should be unconditionally deployed.
- **Conditional Activation**. When loading an untrusted site in a browser tab: only dynamic content (applets, flash animations, JavaScript code, etc.) in that specific tab is untrusted, therefore, AccessControlAspect should be deployed only in case an untrusted site is about to be loaded. This per-page security enforcement is what the Firefox add-on NoScript[16] does.
- **Unconditional Deactivation**. When a user marks an application as trusted: all code in that application is now considered trusted, therefore, AccessControlAspect should be unconditionally undeployed in future executions.
- **Conditional Deactivation**. When executing commands in a command-line interpreter: certain commands do not access sensible resources, like echo and date (only the screen, which the user already has access to at this point), therefore, AccessControlAspect can be undeployed for them.

These are only some direct advantages derived from dynamic deployment for access control. Further study may reveal more, such as a policy that dynamically enforces different security levels (with different performance trade-offs) depending on the state of the application.

6.2.2 Increased Granularity

In the original JAC architecture and in the AspectJ modularized implementation, access control is specified at the class level. However, access control at the level of classes has limits in terms of expressiveness [36], [37]. Let us borrow an example (not the solution) from [37] to illustrate how per-object access control permits the expression of finer-grained policies. In a hospital management system, users can be doctors or patients. Doctors can both read and update records of their patients, whereas patients can only read their own records. If the system is modeled with Doctor and Patient classes, assigning them the corresponding read and write permissions, a security breach is immediately introduced. Firstly, *all* doctors are able to read and write *all* patient records, not only the records of their *own* patients. At the same time, patients are able to read *other* patient records, probably including private information. Conversely, with object-based access control, one direct solution is just to deploy the following restriction aspects on Doctor and Patient instances:

```
aspect DoctorsRestriction {
  pointcut patient Access(Patient patient):
      execution(* Patient.*(..)) && this(patient);

  before(Patient patient): patientAccess(patient) {
    //reference to the doctor this aspect is deployed on
    Doctor doctor = ...;
    //this doctor does not treat the patient
    if(!doctor.treats(patient)){
      throw new AccessControlException();
    }
  }}

aspect PatientsRestriction {
  pointcut readRecord(Patient patient):
      execution(Record Patient.getRecord()) && this(patient);

  before(Patient patient): readRecord(patient) {
    //reference to the patient this aspect is deployed on
    Patient thisPatient = ...;
    //this patient is accessing the record of another patient
    if(thisPatient != patient){
      throw new AccessControlException();
    }
  }}
```

With these restriction aspects deployed on Doctor and Patient instances, doctors are not able to read or update records of other patients; and patients are not able to read records of other patients.

To illustrate another feature derived from the increased granularity, let us extend the example by adding a second kind of doctor, a doctor-in-chief, that is allowed to view and modify the records of patients of other doctors (*i.e.* no restriction is deployed on it). Now suppose that a doctor wants to pass a patient record to another doctor, but s/he does not want the record to be modified. Since the doctor receiving the record could be a doctor-in-chief, it is not safe to just pass the record.

To solve this issue, the doctor can, defensively, deploy the following restriction on the record to prevent it from being modified:

```
aspect RecordRestriction{
  pointcut writeRecord(Record record):
      execution(* Record.set*(..)) && target(record);

  before(Record record): writeRecord(record){
    //reference to the record this aspect is deployed on
    Record thisRecord = ...;
    if(thisRecord == target){
      throw new AccessControlException();
    }
  }}
```

Notice that this restriction is deployed on the Record object itself, not in the entities potentially accessing it. Even if the doctor that receives the record shares it with

15. All objects which already have restriction aspects deployed on them are still subject to access control.

16. http://noscript.net. It has more than 69 million downloads.

other entities, they will not be able to alter it, regardless of their own restrictions.

6.2.3 Increased Control

Scoping strategies can be used to define other kind of policies by easily changing the scope of restriction aspects. For instance, one can define a pervasive policy [38]. This policy ensures that all actions *derived* from the execution of a restricted object must be restricted as well. By derived we refer to the actions directly generated by the object and the actions generated by other objects it creates. This policy is implemented by deploying restriction aspects with the *pervasive* scoping strategy [16] defined as [true,true]. Actions directly generated by the object are subject to the policy because the call stack propagation function is always true, so restriction aspects always propagate on the stack. Actions indirectly generated by the object are also subject to the policy because the delayed evaluation function is always true, so restriction aspects always propagate to new objects. Consider the following example:

```
//(1) code in an untrusted object
File file = new File("/etc/passwd");
trustedObject.file = file;

//(2) code in trustedObject
this.file.delete();
```

Under stack inspection semantics, the file deletion would be allowed because there is no trace of the untrusted object in the stack. Conversely, in a pervasive policy semantics, the call to delete throws an exception. The reason is that when the untrusted object creates the File object, the restriction aspects of the former propagate to the latter, due to the delayed evaluation function.This can be seen as a kind of history-based access control [39].

Implementing this policy using the solution based on AspectJ and automata involves (besides adding per-object restrictions support) a complete rewrite of the access control automata. Conversely, as shown above, using scoping strategies is just a matter of changing the propagation functions.

6.3 Optimization Opportunities

The solution based on scoping strategies presents more direct optimization opportunities than the solution based on access control automata.

First, because access control automata are not defined within AspectJ, they cannot benefit from the optimizations made to the compiler/runtime of the language. In this regard, the cflow pointcut is a very good example of the kind of optimizations that can be achieved when scope constructs are part of the aspect language. Although cflow can be implemented as an automaton updated by a dedicated aspect (just like access control automata), it is translated, in most cases, into a simple counter which is increased at the beginning of join points matched by the argument of cflow, and decreased at the end. Even more, a cflow pointcut can be removed completely when conjuncted with an always-false pointcut. This last optimization would be much more complicated without integrated support from the compiler.

Second, although one can devise optimizations to the access control automata supporting per-class restrictions, adding support for per-object restrictions—to take advantage of the increased expressiveness shown in Section 6.2—, makes optimizations much harder due to the dynamic nature of policy enforcement. The code to update the automaton for per-object restrictions is:

```
aspect FileRestriction{
  //...pointcut and advice remain the same

  pointcut all(Object ctx): execution(* *.*(..)) && this(ctx);

  Object around(Object ctx): all(ctx){
    boolean t = isTrusted(ctx);
    if(t){ automaton.enterTrusted(); }
    else { automaton.enterUntrusted(); }
    try { return proceed(); }
    finally {
      if(t){ automaton.exitTrusted(); }
      else { automaton.exitUntrusted(); }
    }
  }}
```

When the transitions to take in the automaton depend on dynamic conditions, in this case whether the object currently in context is trusted or not, no simple general static analysis applies: every method in the system must be selected by the aspects to update the automata. Conversely, the access control scoping strategy is generic and *known in advance*. Actually, the conditions expressed by its components are all statically determinable, except for deciding whether a call to doPrivileged is a self call or not. Note that even this condition can be statically determined, by adopting a syntactic resolution of self calls, as in the encapsulation policies of Schärli *et al.* [40]. Therefore it is possible to evaluate the scoping strategy for access control at compile time, avoiding unnecessary runtime overhead. The study of partial evaluation of scoping strategies in this case, and in general, is on-going work.

7 Related Work

The relation between aspects and security has a long history. A particular line of work that is complementary to access control as considered in this paper is the notion of data flow pointcuts and properties [41], [42], [43]. With respect to access control, there are related proposals in the area of modularization of access control checks, as well as in alternative implementations of stack inspection.

Modularization of Access Control Checks. Several proposals have been presented to modularize access control, particularly in Java [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]. However, these approaches only factor out access control checks into aspects, as shown in Section 3.2. None of them tackles the issue of

modularizing the stack inspection semantics mechanism itself, nor do they support privileged execution and permission contexts.

Alternative Implementations of Stack Inspection. There are several proposals providing alternative implementations of stack inspection, but to the best of our knowledge, none of them is explicitly aspect oriented.

In [25], Wallach *et al.* present SAFKASI, in which code is transformed to follow a *security-passing style* [24]. This is similar to continuation-passing style because an additional parameter representing the current access control state is added to all methods. The value of this parameter is maintained by a pushdown automaton. This is the work that inspired the access control automaton presented in Section 3.6 (as mentioned before, the only difference is that we use aspect orientation for updating the automaton). In contrast, our approach "passes" permission specifications around in the form of restriction aspects with an appropriate propagation strategy.

In [26], *inlined reference monitors* are used to maintain access control state in the application. The approach is very close to AOP (even though the connection is not explicitly established) in that actions (*e.g.*PERFORM SECURITY UPDATE <action >) are performed whenever certain events occur (WHEN <method signature >). Note that these two proposals support privileged execution, but do not deal with capturing and restoring permission contexts.

8 Conclusions and Perspectives

We have presented two completely modular access control architectures based on aspects, one using AspectJ, the *de facto* aspect-oriented extension to Java; and another using dynamic deployment and scoping strategies. To the best of our knowledge, this is the first successful attempt to fully modularize the access control architecture of Java by means of aspect orientation. Previous proposals only considered the modularization of access control checks, whereas we also deal with the more advanced features associated to the stack inspection algorithm, *i.e.* privileged execution and permission contexts.

We showed that, contrary to the current implementation of access control in Java and in our implementation using AspectJ, the scoping strategies approach is not based on specialized support for stack introspection, but rather uses general-purpose aspect-oriented constructs. The key element for this is the advanced control of aspect scoping provided by scoping strategies.

To validate our result, we provide two implementations of the modularized JAC architecture. As a first proof of concept, the architecture is implemented in an interpreted higher-order procedural language similar to Scheme, supporting aspects and scoping strategies. The second implementation is in a real-world aspect-oriented extension of JavaScript, called AspectScript [38]. As a consequence, real-world JavaScript applications can benefit from a modularized access control architecture. This is particularly useful considering that advanced access control, as in Java, is not part of the language specification [44], despite the importance of security in this ubiquitous language.

As mentioned before, we plan to explore several research issues. First, how to take full advantage of the dynamism of aspect deployment and scoping strategies. Second, how to modularly specify the calls to doPrivileged in order to completely aspectize privileged execution. And finally, how to partial evaluate scoping strategies to increase runtime performance.

Apart from enabling a modular implementation of an expressive access control architecture, scoping strategies open various new research opportunities. First, considering per-object permissions is more expressive than per-class permissions, as recognized in [36] and [37]. Therefore, our aspect-oriented implementation of access control can bring that level of expressiveness to languages supporting scoping strategies. Second, scoping strategies have been shown to be useful in a distributed context as well [17]; this means that our proposal can be extended to support access control in a distributed setting, modularly.

Availability. The elements used in this study are available at http://pleiad.cl/research/3sec. They include (1) the Scheme interpreter for a multi-threaded Scheme-like language with scoping strategies; (2) the TmpCleaner service implemented in this language, as well as (3) in AspectScript; (4) the AspectJ implementation of the JAC architecture; and (5) the list of pointcuts for access control in the standard Java distribution.

References

[1] L. Gong and G. Ellison, *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003.

[2] L. Gong, M. Mueller, H. Prafullchandra, R. Schemers, and S. Microsystems, "Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2," *Proceedings of the USENIX Symposium for Secure Systems*, 1997.

[3] N. Hardy, "The confused deputy," *SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[4] I. Welch and R. Stroud, "Supporting real world security models in Java," in *Distributed Computing Systems, 1999. Proceedings. 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, 1999, pp. 155–159.

[5] J. Viega, J. Bloch, and P. Chandra, "Applying Aspect-Oriented programming to security," *Cutter IT Journal*, vol. 14, no. 2, pp. 31–39, Feb. 2001.

[6] B. De Win, B. Vanhaute, and B. De Decker, "Security through aspect-oriented programming," in *Advances in Network and Distributed Systems Security*, B. De Decker, F. Piessens, J. Smits, and E. Van Herreweghen, Eds. Kluwer Academic Publishers, 2001, pp. 125–138.

[7] B. D. Win, B. Vanhaute, and B. D. Decker, "How aspect-oriented programming can help to build secure software," *Informatica 26*, vol. 2, pp. 141–149, 2002.

[8] P. Sowikowski and K. Zieliski, "Comparison study of aspect-oriented and container managed security," in *Proceedings of the Workshop on Analysis of Aspect Oriented Software*, Germany, 2003.

[9] B. D. Win, W. Joosen, and F. Piessens, "Developing secure applications through Aspect-Oriented programming," in *Aspect-Oriented Software Development*. Addison-Wesley Professional, Oct. 2004, pp. 633—650.

[10] B. Vanhaute, B. D. Decker, B. D. Win, and D. Decker, "Building frameworks in AspectJ," *Workshop on Advanced Separation of Concerns (ECOOP)*, pp. 1–6, 2001.

[11] B. D. Win, W. Joosen, and F. Piessens, "AOSD security: A practical assessment," *Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT)*, vol. 2003, pp. 1–6, 2003.

[12] M. Huang, C. Wang, and L. Zhang, "Toward a reusable and generic security aspect library," in *AOSD Technologies for Application-Level Security*, 2004.

[13] R. Ramachandran, "AspectJ for multilevel security," Master Thesis, Victoria University of Wellington, 2006.

[14] A. Mourad, M. Laverdire, and M. Debbabi, "An aspect-oriented approach for the systematic security hardening of code," *Computers & Security*, vol. 27, no. 3-4, pp. 101–114, Jun. 2008.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An overview of AspectJ," in *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, ser. Lecture Notes in Computer Science, J. L. Knudsen, Ed., no. 2072. Budapest, Hungary: Springer-Verlag, Jun. 2001, pp. 327–353.

[16] É. Tanter, "Expressive scoping of dynamically-deployed aspects," in *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*. Brussels, Belgium: ACM Press, Apr. 2008, pp. 168–179.

[17] É. Tanter, J. Fabry, R. Douence, J. Noyé, and M. Südholt, "Expressive scoping of distributed aspects," in *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development (AOSD 2009)*. Charlottesville, Virginia, USA: ACM Press, Mar. 2009, pp. 27–38.

[18] M. Felleisen, "On the expressive power of programming languages," *Science of Computer Programming*, vol. 17, pp. 35–75, 1991.

[19] P. Samarati and S. D. C. di Vimercati, "Access control: Policies, models, and mechanisms," in *Foundations of Security Analysis and Design*, ser. Lecture Notes in Computer Science. London, UK: Springer Berlin / Heidelberg, 2001, vol. 2171, no. 2171, pp. 137–196.

[20] Sun Microsystems Inc., "Java security architecture," http://java.sun.com/javase/6/docs/technotes/guides/security/spec/security-specTOC.fm.html.

[21] G. Karjoth, "An operational semantics of Java 2 access control," in *Computer Security Foundations Workshop, IEEE*. Washington, DC, USA: IEEE Computer Society, 2000, p. 224.

[22] H. Masuhara, G. Kiczales, and C. Dutchyn, "A compilation and optimization model for aspect-oriented programs," in *Proceedings of Compiler Construction (CC2003)*, ser. Lecture Notes in Computer Science, G. Hedin, Ed., vol. 2622. Springer-Verlag, 2003, pp. 46–60.

[23] M. Wand, G. Kiczales, and C. Dutchyn, "A semantics for advice and dynamic join points in aspect-oriented programming," *ACM Transactions on Programming Languages and Systems*, vol. 26, no. 5, pp. 890–910, Sep. 2004.

[24] D. Wallach and E. Felten, "Understanding Java stack inspection," in *Proceedings of the IEEE Symposium on Security and Privacy*, 1998, pp. 52–63.

[25] D. Wallach, A. Appel, and E. Felten, "SAFKASI: a security mechanism for language-based systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 9, no. 4, pp. 341–378, 2000.

[26] U. Erlingsson and F. Schneider, "IRM enforcement of Java stack inspection," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2000, pp. 246–255.

[27] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, 2000.

[28] É. Tanter, "Beyond static and dynamic scope," in *Proceedings of the 5th ACM Dynamic Languages Symposium (DLS 2009)*. Orlando, FL, USA: ACM Press, Oct. 2009, pp. 3–14.

[29] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann, "An overview of CaesarJ," in *Transactions on Aspect-Oriented Software Development*, ser. Lecture Notes in Computer Science, vol. 3880. Springer-Verlag, Feb. 2006, pp. 135–173.

[30] D. Ferraiolo and R. Kuhn, "Role-Based access control," *15th NIST-NCSC National Computer Security Conference*, pp. 554–563, 1992.

[31] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranho, A. Garcia, and C. M. F. Rubira, "Exceptions and aspects: the devil is in the details," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. Portland, Oregon, USA: ACM, 2006, pp. 152–162.

[32] N. Cacho, F. C. Filho, A. Garcia, and E. Figueiredo, "EJFlow: taming exceptional control flows in aspect-oriented programming," in *Proceedings of the 7th international conference on Aspect-oriented software development*. Brussels, Belgium: ACM, 2008, pp. 72–83.

[33] E. Bodden, F. Forster, and F. Steimann, "Avoiding infinite recursion with stratified aspects," in *Proceedings of Net.ObjectDays 2006*, ser. Lecture Notes in Informatics. GI-Edition, 2006, pp. 49–54.

[34] É. Tanter, "Execution levels for aspect-oriented programming," in *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*. Rennes and Saint Malo, France: ACM Press, Mar. 2010, pp. 37–48, best Paper Award.

[35] D. Brewer and M. Nash, "The chinese wall security policy," in *IEEE Symposium on Security and Privacy*. Los Alamitos, CA, USA: IEEE Computer Society, 1989, p. 206.

[36] M. Pistoia, A. Banerjee, and D. A. Naumann, "Beyond stack inspection: A unified Access-Control and Information-Flow security model," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP '07)*. IEEE Computer Society, 2007, pp. 149–163.

[37] J. Fischer, D. Marino, R. Majumdar, and T. Millstein, "Fine-Grained access control with Object-Sensitive roles," in *23rd European Conference on Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science, vol. 5653. Genova, Italy: Springer Berlin / Heidelberg, Jul. 2009, pp. 173–194.

[38] R. Toledo, P. Leger, and É. Tanter, "AspectScript: Expressive aspects for the Web," in *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*. Rennes and Saint Malo, France: ACM Press, Mar. 2010, pp. 13–24.

[39] M. Abadi and C. Fournet, "Access control based on execution history," *In Proceedings of the 10th annual Network and Distributed System Security Symposium*, pp. 107–121, 2003.

[40] N. Schärli, A. Black, and S. Ducasse, "Object-oriented encapsulation for dynamically-typed languages," in *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2004)*. Vancouver, British Columbia, Canada: ACM Press, Oct. 2004, pp. 130–149, aCM SIGPLAN Notices, 39(11).

[41] H. Masuhara and K. Kawauchi, "Dataflow pointcut in aspect-oriented programming," in *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS'03)*, ser. Lecture Notes in Computer Science, vol. 2895, Nov. 2003, pp. 105–121.

[42] D. Alhadidi, A. Boukhtouta, N. Belblidia, M. Debbabi, and P. Bhattacharya, "The dataflow pointcut: a formal and practical framework," in *Proceedings of the 8th ACM international conference on Aspect-oriented software development*. Charlottesville, Virginia, USA: ACM, 2009, pp. 15–26.

[43] A. Mourad, A. Soeanu, M. Laverdire, and M. Debbabi, "New aspect-oriented constructs for security hardening concerns," *Computers & Security*, vol. 28, no. 6, pp. 341–358, Sep. 2009.

[44] E. International, *ECMAScript Language Specification. ECMA-262*, 5th ed., Apr. 2009.