A Permission System for Secure AOP

Wouter De Borger, Bart De Win, Bert Lagaisse and Wouter Joosen
Dept. of Computer Science, KULeuven, Belgium
{wouter.deborger,bart.dewin,bert.lagaisse,wouter.joosen}@cs.kuleuven.be

ABSTRACT

The integration of third-party aspects into applications creates security challenges. Due to the intrusive impact of aspects, one cannot guarantee that the dynamic composition of aspects does not lead to misbehavior. The newly composed aspect typically has many, if not unrestricted, rights to read and modify attributes of the base system. AspectJ, amongst other AOP systems, suffers from this limitation, which makes the composition of independently developed aspects riskful.

We have defined and prototyped a run-time policy enforcement model based on execution history to protect programs from untrusted aspects. The dynamic nature of the approach has the advantage that up to date run-time information allows more accurate decision making. We have built a prototype for AspectJ and illustrate its use in a realistic example. Our evaluation shows that practical use of such a solution is feasible and that run-time overhead can be limited.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]; D.2.0 [**Software Engineering**]: Protection mechanisms; D.2.4 [**Software/Program Verification**]: Assertion checkers

Keywords

Language-based security, aspect-oriented programming, permission system, execution history

1. INTRODUCTION

The integration of third-party aspects into applications creates new security challenges. Stack-based inspection, as used in OOP languages [23, 16], is no longer sufficient, as aspects invalidate the basic assumptions of stack-based inspection. Features such as *before* and *after advice* in AspectJ [20] leave no trace on the stack. *Privileged aspects* [1], have access to private members. And most important, aspects can not only cause execution of methods, they can also observe and prevent method execution.

The relevance of third party aspects can be illustrated by the use of the JBoss application server in the context of the *Platform as a Service* (PaaS) [37] methodology. PaaS means that a service provider offers a computing platform on which third parties can deploy their software. Many applications of different owners can run on the same platform instance. This allows intensive reuse of both hardware and software.

The JBoss application server is implemented using aspects, the so called infrastructure aspects. Applications deployed on the server can also contain aspects, the application aspects. To ensure integrity of the server, it is important that application aspects don't interfere with infrastructure aspects or other parts of the infrastructure. For the provider of the platform instance, it is impossible to manually verify each and every application for potential interferences. An automated policy enforcement mechanism is required to keep the untrusted aspects in check, without disabling them completely.

To address these issues, we present an approach of run-time policy enforcement, based on execution history. This model can be applied to existing AOP languages without semantic changes. Like stack-based inspection, this model makes policy decisions at run-time. Unlike stack-based inspection, the decision is not only based on the stack, but on the complete execution history.

We have evaluated our approach by building a prototype. Initial experience when applying the prototype solution to some example applications indicates that development and run-time overhead can be limited.

A number of complementary approaches have been published. A first category is based on restricting the invasive power of aspects by adapting (extending or restricting) the core aspect language [24, 17, 22]. A second category is based on redesigning the concept of an aspect and supporting this in the underlying type system [4, 8]. These approaches require a semantic change of the language they are applied to. As such they are all hard to apply to existing AOP languages.

The rest of this paper is structured as follows: In Section 2 the problem is elaborated. In Section 3 the approach is outlined. Section 4 refines the solution. Section 5 addresses the issue of restrictiveness of our approach by detailing a number of design patterns. Section 6 further evaluates the approach by presenting a prototype and illustrating the run-time and development overhead for our case study. Section 7

© ACM, (2010). This is the author's version of the work. It was posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in the Proceedings of the 9th ACM international conference on Aspect-oriented software development
http://doi.acm.org/10.1145/1739230.1739254

briefly discusses the approach. Finally, the related work is discussed in Section 8 and Section 9 concludes.

2. PROBLEM ELABORATION

In this section we give a short overview of stack inspection and its inability to protect AOP enabled programs. We also illustrate the aforementioned problems with an overview of the potential risks imposed by third party aspects. We discuss which features should be supported by the security infrastructure. Then we present the case study that is used throughout this paper.

2.1 Stack Inspection and AOP

Third party components inherently pose a security threat, for OOP as well as for AOP languages. Application servers, for example, must be able to access the file system. But the third party applications deployed in the sever must not be allowed to read each others files. When an attempt is made to open a file, the system must decide whether this operation is permitted.

In Java this problem is solved by stack based inspection[36, 16]. When an attempt is made to do a security sensitive operation, such as opening a file, the SecurityManager checks whether every method currently on the stack originates from a source that is allowed to perform this operation. When no third party methods are on the stack, all files are accessible. When methods from one third party component are on the stack, only this component's file are accessible.

Stack based inspection protects callees from their callers. When a third party component calls the server to open a file only accessible to the server, this operation will fail, as the third party method is still on the stack. However, in some cases, the server actually wants to perform security sensitive operations on behalf of a third party component. For example, if a component wants to establish a connection to a remote component, the server will determine whether this request is admissible. If it is, the server will assume responsibility for all subsequent actions required to establish the connection. This is called a privileged action.

Stack inspection protects callees from callers, however there is no protection in the other direction. A callee is no longer on the stack after it returns and the caller has no way of finding out whether the returned value is trustworthy. If this value is subsequently used as an argument to a security sensitive operation, this may result in unintended behavior. For example, if a server would ask a third party component which file it should open, the third party component will no longer be on the stack when the file is opened.

In OOP this is not a problem because each object can maintain a pool of trusted objects on which it can safely make calls. As long as these objects are stored in a private field and no reference to these objects is released, they can not be tampered with. Furthermore, method calls and return values reach only their target and do so unaltered. As such, each caller is sure that when it makes a call to an object it assumes trustworthy, the call will be received unaltered and undisclosed by the trusted object and the return value will reach the callee unaltered and unobserved.

For AOP these two assumptions are no longer valid [11]. Private fields for example are not only threatened by priviledged aspects, which can directly access them. The interception mechanisms inherent to AOP can yield references

```
public aspect BadAspect{
  //force class load
  before():execution(void *.main(..)){}
  //disable all security
  void around(): execution(
    void SecurityManager+.check*(..)){}
}
```

Listing 1: Harmfull AspectJ interaction

to any object and as such disclose a private member. Once a reference is obtained, state may be altered and information may be disclosed. Furthermore, method calls can be intercepted, altered and observed.

It is arguably true that stack inspection can be extended to enforce access control on fields and take advices into account. This would solve some of the listed problems, but it would not protect callers from their callees. Listing 1 illustrates that the mechanism of method interception can be used to disable the Java security manager and as such disable all Java security features.

2.2 Requirements

In general different scenarios exist in which an aspect can influence the execution of a system in a potentially harmful manner. Different sources [11, 24] have described a number of these scenarios. Without the intention of being exhaustive, we briefly summarize the most important ones:

- reading and writing private fields
- executing private methods
- preventing execution of methods
- inspecting and manipulating data that flows between methods
- accessing instances without explicitly obtaining references to them
- bypassing the execution of other aspect's advices by directly accessing the target joinpoint
- influencing aspect ordering

As demonstrated above, these interactions make stack inspection insufficient to protect aspect enabled software. Therefore we extend stack inspection to history-based access control, which is more restrictive, though not overly complex.

Such access control model should support a broad spectrum of security policies. We distinguish three categories, each containing a number of typical policy scenarios, which correspond to the previously mentioned issues:

R1 **Enforcing standard Java permissions**

R2 **Protecting a base class against aspects**
This category of policy scenarios is useful for protecting security-aware base classes from possibly harmful aspects.

(a) *Protect the execution of security-sensitive methods.*

(b) *Protect the results of the execution of a method.*

(c) *Restrict the access to particular private members.*

(d) *Prevent aspects from advising a joinpoint.*

R3 **Protecting an aspect against other untrusted aspects**
In this case, all scenarios that were covered in the previous category are relevant as well for protecting the functionality of the aspect.

(e) *Protect security-critical aspects from other aspects.*

2.3 Case Study

Throughout this paper, we use an example application, namely jFTPd [29]. jFTPd is an FTP server written in Java. We have refactored authorization into aspects. The server as well as the authorization aspects are considered the trusted core application. More information about this case study is available in [10, 9].

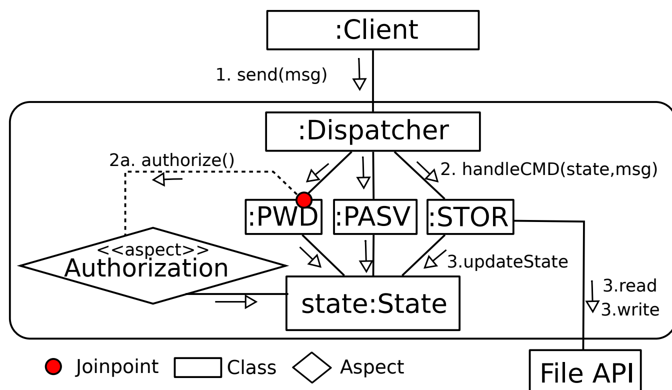

Figure 1: General overview of the jFTPd server

Figure 1 gives a conceptual overview of the server. A client sends messages to the server (1). In the server, the dispatcher receives the message and dispatches it to the proper handler (2). Each FTP command is handled by a specific handler. The handlers can inspect and update the state of the session and they can read and write files (3). Command dispatch is intercepted by the Authorization aspect (2a). This aspect inspects the state to see if the command is authorized. If it is not, the aspect aborts the command.

We add to the server a plug-in that enforces a *chroot* policy. The plug-in intercepts any attempt to change the current directory, which is part of the state. If an attempt is made to change the current directory to a directory that is not a sub-directory of the home directory of the current user, the command is aborted. This plug-in also contains nine malicious aspects, which intercept various calls in the server. The malicious aspects enable attacks such as denial of service, escalation of rights, selective tampering with logs and others.

The policy we enforce on this server is one where the trusted part of the code has access to the required systems resources, such as network ports and files. The plug-in is allowed to prevent a change of the current directory and has read access to the directory that contains all home directories. No further rights are granted. If this policy is implemented by the Java Security infrastructure [16], all of the malicious aspects go undetected.

3. APPROACH

This section outlines the two basic parts of our approach: a weaver-based approach to run-time enforcement and the history-based access control model.

3.1 Weaver-based Approach

We utilize a weaver-based approach to enforce our policy model at run-time. The weaver augments the woven software with logic to maintain the permission state of the software. As such, only the weaver is altered and no modification of the virtual machine or language semantics is required. This approach is shown in Figure 2: at development time, the source is annotated with actions of the permission system, such as the demand for a particular permission. All modules are then processed by a weaver which, apart from doing the aspect-oriented weaving, augments the woven software with logic to maintain the permission state of the software. At run-time, the executing software is monitored by the permission system, which consults a configurable aspect policy about the rights of specific aspects.

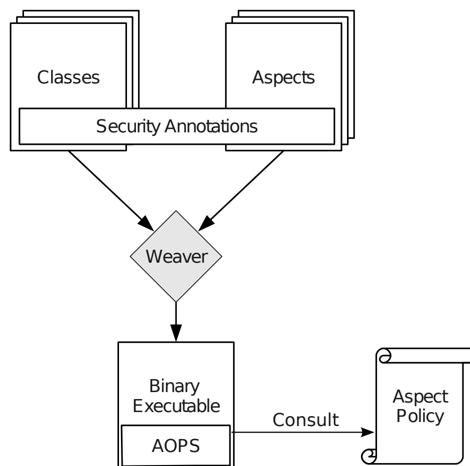

Figure 2: General overview of a weaver-based permission system

As such, this weaver-based approach builds upon two security assumptions: the weaver as well as execution environment are controlled and trusted. The weaver protects against scenarios in which untrusted aspects, which can originate from third party libraries, are incorrectly woven into the rest of the application code. In the same context, we also rely on the safety of the type system to protect against other types of problems.

It is important to notice that the weaver-based approach works independent of the moment of weaving and the weaving techniques. Our approach only requires a trusted weaver and run-time that will enforce security updates. Whether weaving is done dynamically or statically, at compile-time or at load-time, at once or in stages is of no importance.

However, Scoped weaving however is important. Scoped weaving restricts the application of aspects to a scope. Aspect scoping can be based on lexical scope, class loading domains or any other scope. Often class loading techniques are used to realize scoped weaving. This offers a rudimentary form of security as scoped weaving prevents direct aspect interference. Indirect interference however is not prevented.

Scoped weaving can be used to protect the security infrastructure from direct aspect interference. No aspect should be allowed to directly prevent security updates. This allows the security infrastructure to protects itself from indirect interference. When an aspect attempts to interfere with security updates, this will cause a nested security update, which is detected and prevented.

3.2 History-based Access Control

The second part of our approach is the *execution history-based access control model* (HBAC) [2]. In this work we leverage upon HBAC for the protection against untrusted aspects in woven programs. This model is based on two sets of rights: static rights and current rights. The static rights are assigned to aspects by the aspect policy. The current rights are calculated at run-time, based on the static rights. In particular, modifications to the weaver ensure that the following properties hold at all times, and can be checked against at run-time:

1. The current rights are the intersection of the static rights of all aspect-related code that has executed before.
2. The current rights can only be elevated by explicit requests to augment these rights.
3. The current rights upon exit of an aspect-related piece of code can never be more extensive than the rights that were applicable when entering that code.

Figure 3 illustrates HBAC in more detail. In this figure, an execution scenario is depicted in which two base classes interact with two aspects. The `Authorization` aspect intercepts any message that was received from the user prior to handling. The `Authorization` aspect inspects the content of the message and aborts handling if the message contains an unauthorized command. Otherwise normal processing is resumed. The `BadAspect` also intercepts any message prior to execution and after authorization. This allows the `BadAspect` to change the commands after they have been authorized and as such bypass security.

When using our approach, static rights are assigned to all classes and aspects. The names of the static rights are written above the lifelines. At point D, the current rights are examined to decide whether the results can be sent back to the user.

We now illustrate what the current rights look like when using the history-based model and the stack-based model. History based access control is illustrated in Table 1, column 2. For history-based access control, at point A, no aspects have been executed, so the set of current rights is the universal set. At point B the `Authorization` aspect is executing and the current rights are reduced to the intersection of the universal set and the static rights of the `Authorization` aspect. This yields a set containing the static rights of the `Authorization` aspect, which is called `Auth`. At point C the third party aspect `BadAspect` is executed and the current rights are further reduced to the intersection of the current rights and the static rights of the `BadAspect`. Up to point D, no new aspects execute and the current rights remain the same.

If we compare this to the stack-based model, illustrated in Table 1, column 1, point A has the `Dispatcher.send()`

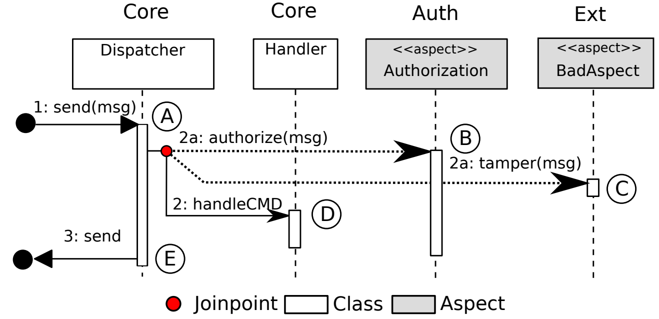

Figure 3: The execution scenario described in Section 2.

	Stack-Based	History-Based
A	Core	All
B	Core ∩ Auth	Auth
C	Core ∩ Auth ∩ Ext [1]	Auth ∩ Ext
D	Core ∩ Auth	Auth ∩ Ext
E	Core	Auth ∩ Ext

Table 1: Run-time rights at the places marked in Figure 3.

method on the stack. This means that the current rights at A are those in the set `Core`. At point C, `Dispatcher.send()`, `Authorization.authorize()` and `BadAspect.tamper()` are on the stack. The run-time rights are thus: `Core ∩ Auth ∩ Ext`. At point D, only `Dispatcher.send()`, `Handler.handleCMD()` and `Authorization.authorize()` are on the stack and the current rights are `Core ∩ Auth`.

The current rights of the stack-based model show no trace of the presence of `BadAspect` even though it has had its influence on the execution. In particular the `BadAspect` changed the message passed on to be handled after it was authorized. With stack inspection alone, this cannot be prevented. With history-based access control, the modification leaves a clear trace that can be used to prevent the modified command from sending anything to the client (point D).

This history-based model will be used to monitor and control the impact aspects have on the execution of an application. While the run-time rights of a block of code are influenced only by aspect executions, the checking of rights can occur anywhere in the application. Consequently, the system can be used to protect classes from untrusted aspects as well as aspects from other untrusted aspects. Protecting classes from untrusted classes can be achieved using the existing permission systems. By combining history-based access control with the existing support for stack-based inspection, the run-time overhead of HBAC is minimized and its restrictive nature must only be taken into account when aspects are involved.

4. DETAILED SOLUTION

This section describes the structure of our Aspect Oriented Permission System (AOPS). This system implements

[1] in AspectJ only around advice remains on the stack

history-based access control for AspectJ [20]. It is designed to support aspects and advices in single-threaded systems. Inter-type declarations are currently not supported. This is not an inherent limitation of the approach, but part of the future work.

The run-time structure of AOPS has three layers, as depicted in Figure 4. The lowest layer is the supporting platform, in our case the Java Virtual Machine (JVM). On top of this platform, the AOPS run-time is deployed. This runtime collects execution information, used to calculate the current rights. The top layer is the application layer, containing the base application and the third party aspects.

The application uses AOPS for demanding rights and updating the current rights. When an application demands a right, AOPS checks whether that right is implied by the current rights. If it is, execution continues, otherwise an exception is thrown. Updating of the current rights is realized through two mechanisms: implicit and explicit modification. The implicit modifications are performed by code that was inserted by the weaver. Implicit modifications can only reduce the current rights. The explicit modifications on the other hand allows the application to explicitly elevate the current rights.

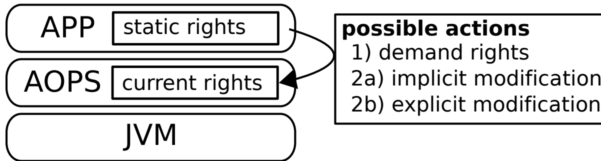

Figure 4: Runtime Structure of AOPS.

The rest of this section will discuss the different parts of the run-time structure in detail.

4.1 Rights as Permissions

Rights are implemented in the form of permissions, all of which inherit from a single root permission that is specific to AOPS, the `AOPPermission` (Listing 2). The model is compatible with standard Java permissions by means of a special permission, `JavaPermissionWrapper`, that wraps standard permissions.

The permission model is extensible, so new types of permissions can be defined for specific purposes. Similar to the Java permission model, every permission instance has two parameters (name and actions) and methods to compute equality and implication. AOPS requires two extra methods: one to calculate the intersection of two permissions and one to compute the union of two permissions. The AOPS permission model is comparable to the .NET CLR permission model in this respect [23].

Because of security reasons, permissions are not inherited between aspects, since this could enable scenarios in which permissions assigned to trusted aspects would be unwillingly copied to untrusted aspects via inheritance.

4.2 Static Rights

At compile time (or load time), *static rights* can be assigned to aspects. The granularity of assignment is an entire aspect, no rights can be assigned to individual advices or any other parts of an aspect. This is a deliberate choice for two

```
interface Permission {
  boolean implies(Permission permission);
  boolean equals(Object obj);

  String getName();
  String getActions();
}

interface AOPPermission extends Permission {
  AOPPermission intersect(AOPPermission perm) ;
  AOPPermission union(AOPPermission perm) ;
}
```

Listing 2: Java Permission vs AOPPermission

reasons: (i) it is similar to the existing language-based protection mechanisms and (ii) no practical, convincing cases have been encountered that required a more fine-grained assignment of rights. Hence, the static rights of any executable subpart of an aspect are determined based on the static rights of the aspect that lexically contains the executable subpart.

The policy file used for the assignment of static rights to aspects is based on the standard Java security policy file. An example is depicted in Listing 3.

```
grant aspect "security.Authentication" {
 permission security.AuthPermission;
 permission java.io.FilePermission "/tmp/auth.txt";
 permission java.io.FilePermission "/tmp/tmp.txt";
}
```

Listing 3: Example policy for an aspect called 'security.Authentication'

4.3 Current Rights

The core of AOPS is the `PermissionManager`, which is responsible for storing the current rights, and for all actions that must be performed on those rights: updating the current rights, checking rights and augmenting the current rights. The `PermissionManager` maintains a single set of current rights and, hence, focuses on single-threaded applications. There are no fundamental obstacles, however, to extend this to multi-threaded applications.

The computation of the current rights is based on intersection: every time the current rights must be updated, they are intersected with the static rights of the aspect caused the update, using the `intersect()` method defined on the permission.

4.4 Checking

A security check in AOPS is requested by invoking the `demand` method of the `PermissionManager` as shown in Listing 4. When this method is called, the current rights are compared with the requested permission. Only if the requested permission is available in (or implied by) the current rights, the application is allowed to proceed. Otherwise, an `AOPSecurityException` is thrown.

Due to the use of aspects it is not always feasible to include checks programmatically. For instance, in order to restrict the privileged access to a particular private class member, one should actually include a check in the accessor-method generated by the weaver for this purpose, which is clearly not

```
PermissionManager mngr =
  PermissionManager.getPermissionManager();
AOPPermission perm =
  new SensitiveOperationPermission();
mngr.demand(perm);
<sensitive operation>
```

Listing 4: Protecting a sensitive operation

accessible at development time. Similarly, an object pre-initialization joinpoint is not visible for a developer. This problem can be addressed by the use of automatic insertion of checks in the (possibly generated) code during the weaving phase. Annotations such as the ones shown in Listing 5, can be used to instruct the weaver where to include such checks. Alternatively aspects can be used to insert the checks, as in Listing 6.

```
class Secret{
  @CheckAOPPermission(
    permission="SecretPermission"
  )
  private String secret;
  ...
```

Listing 5: Annotation based permission demand

```
aspect Enforcer {
  pointcut protect():
    get(private Secret.secret) ||
    set(private Secret.secret);

  before():protect(){
    demand(new SecretPermission());
  }

  declare precedence: *,Enforcer;
}
```

Listing 6: Aspect based permission demand

When using the aspect based approach, aspect ordering is very important. By declaring an aspect of least precedence, the check is guaranteed to be executed just before the actual joinpoint. If any other advice tries to declare itself as lower precedence, the weaver will detect the conflict and abort weaving. Obviously the aspect demanding the permission should hold the permission itself, or the check is bound to fail.

In this implementation we opted for the aspect based approach, to reduce the impact on the weaver.

4.5 Implicit Modifications

One of the challenges of AOPS, is related to updating the current rights. As a result of the weaving process, the aspects become scattered over different modules, and tangled with other code. In order to update the current rights correctly, every execution of a piece of aspect code must be identified and the static rights must be determined correctly.

Each time an aspect related piece of code is entered, the current rights must be updated according to the static rights of the newly entered aspect (by taking the intersection of the current rights and the static rights of the aspect being entered). It is important to update as early as possible to prevent aspects or exceptions to bypass the update. In the case of advice, care must be taken to update *every* occurrence of the advice body: since the weaver might copy and inline the body into various places, this is not necessarily a localized modification.

Aspects can influence the execution history in various ways: an advice is being executed on some joinpoint, an aspect method is being invoked or an if-pointcut is executed. No other, more specific events can influence the current rights. A proceed statement in an around advice does not influence the execution history: since an advice or a method, upon returning, can never increase the current rights compared to when it was entered, the current rights after a proceed statement will never be increased compared to before the execution of the proceed statement. Therefore, it is not necessary to update the current rights again. Similarly, exception handlers do not require special attention.

The updating of the current rights depends on the identity of the current aspect. Since it is impossible to rely on the identity of the run-time module to determine the identity of the newly entered aspect (because of inlining), the weaver is responsible for inserting the identity in every relevant piece of code. Listing 7 illustrates the body of an advice before and after it has been processed by the AOPS weaver.

```
//advice before weaving
before{}: ... {
  <something useful>
}

//advice after weaving
before{}: ... {
  PermissionManager permmngr =
     PermissionManager.getPermissionManager() ;
  permmngr.updateCurrent(<the full aspect name>);
  <something useful>
}
```

Listing 7: Placement of automatic update statements

4.6 Explicit Modifications

In some situations, it is useful for security-aware aspects (or classes) to increase the current rights with a selection of rights from its static permission set. For instance, consider a scenario in which the `Authentication` aspect has to read a password file. This scenario only works if either all previously executed aspects have a permission to access the password file, or the aspect is capable of augmenting the current rights.

At least two types of rights amplification are necessary to make the system useful in practice: (i) to execute a piece of code with elevated rights (known as *grant*) and (ii) to increase the trust in the return result of a particular piece of code (*accept*). From [2]:

Grant(P){B} Before running statement B, the initial value of the current permissions is saved and the permission P is added to the current permissions. When B completes (possibly with an exception), the current permissions are assigned the intersection with their initial values.

Rephrased, code can temporarily increase the rights to run a block of code, but upon returning all extra rights

and any other rights that were lost during computation will be removed from the current set.

Accept(P){B} Before running statement B, the initial value of the current permissions is saved. If B completes normally, then the intersection of this initial value and P is added to the current permissions. (If B terminates with an exception, then the current permissions are not modified.)
Rephrased, if during the computation of a block a permission contained in the initial set and implied by P was lost, this permission will be added to the current set again.

It is clear that the augmentation of rights is a potentially dangerous action and, hence, its use should be controlled. This can be achieved, for instance, using special permissions that represent this type of right and assigning those permissions only to aspects trusted for the augmentation of rights. In order to not overly complicate the model, AOPS allows every aspect to augment the current rights, under the restriction that only rights that are part of the static set can be used for augmentation. As such, an aspect with few static rights cannot increase its permissions at run time.

5. USAGE PATTERNS

With history-based access control, the permission set always decreases. This might make HBAC seem very strict, however, it is quite easy to simulate stack-based access control with it [2]. This implies that if HBAC is used correctly, every policy that can be enforced by the stack-based model can also be expressed by the history based model.

To make optimal use of AOPS, it is important to know how to use it. We have identified three patterns that may provide guidance: *guard*, *grant-demand* and *kernel-accept*. We now explain these patterns and refer to the related requirements, as defined in Section 2.2.

Guard: a field or method is decorated with a demand operation that prevents access when a certain permission is not implied by the current permissions. This pattern guarantees that malicious code cannot influence the operation. Guards can be used to secure private methods or fields (R2a, R2c). It is also the key to enforcement of standard Java permissions (R1).

Grant-demand: a permission is granted to make a direct or indirect call to a guarded member or sensitive codepath. When the sensitive path returns, the permission is demanded. This ensures that the sensitive code path was not influenced by malicious code, not holding this permission. This pattern should only be used when the parameters used to construct the sensitive path are trustworthy. This pattern is important to ensure correct return values (R2b) and ensure integrity of methods and advices (R3e). This patterns can also be used to prevent aspects from interfering with certain joinpoints (R2d). This is done by placing a grant-demand structure in a high precedence aspect. Such an aspect, `Enforcer`, is shown in Listing 8.

```
aspect Enforcer {

  pointcut protect(): ...;

  void around():protect(){
    AOPPermission pp = new ProtectedPermission();
    grant(pp){
        Object o = proceed();
        demand(pp);
        return o;
    }
  }

  declare precedence: Enforcer, *;
}
```

Listing 8: Grant-demand pattern as aspect

The `Enforcer` aspect will effectively enclose all other aspects. If a malicious aspect tries to declare itself as higher precedence than the `Enforcer`, the weaving process will fail. If any aspect influences the same joinpoint, this will be detected upon return. If the policy violation must not only be detected but also prevented, a low precedence aspect can be used to demand the permission just before the joinpoint is entered.

Kernel-accept: a small kernel is locked tight with guards so that its integrity can be guaranteed at all times. This kernel accepts responsibility for a potentially untrusted code path it constructs. This path is constructed based only on information available inside the trusted kernel. When the path terminates, permissions are restored and another untrusted path can be constructed. As each loop is isolated from each other loop, the kernel can safely accept responsibility for the next loop. This pattern is useful in processes that run independent loops, such as event processing systems and servers. The Kernel-accept pattern is illustrated in Listing 9.

```
//AllPermission implies all permissions
AOPPermission full = new AllPermission();
AOPPermission minimal = new CoreLoopPermission();
PermissionManager m =
   PermissionManager.getPermissionManager();
while (!done) {
  //grant all permission in static rights
  Accept(full){
    String msg = in.readLine();
    // this is point A of the overview diagram
    if (line != null) {
      String reply = handleCMD(msg);
      // this is point D of the overview diagram
      // make sure no unauthorized aspect
      // influenced the command handling
      m.demand(minimal);
      out.write(reply);
    }
  }
}
```

Listing 9: Kernel accept pattern in the main server loop (server as in Figure 1)

This pattern is not used to protect against malicious code, but to allow untrusted code to execute in a secure manner. A Kernel-accept indicates an extension point were a less trusted piece of code can augment the base code without breaching security.

As was shown, these patterns can be installed using aspects, which allows for efficient deployment. Correctness of the pointcuts is of course critical: if the guards are not applied, the system is not protected.

6. EVALUATION

In this section we will discuss the implementation of a prototype of our enforcement mechanism and evaluate the complexity of the prototype. With this prototype we implemented the jFTPd case study to evaluate the practical value of AOPS. The evaluation will address development overhead and run-time overhead. We will also apply the aforementioned patterns to illustrate that AOPS is not overly restrictive.

6.1 Prototype

In the ideal case, AOPS is integrated in the compiler and run-time of the base language (the AspectJ compiler and the Java virtual machine). From a security perspective this is preferable since the language run-time acts as a security micro-kernel that cannot be tampered with by the executing application. The standard Java permission system has been designed this way. Furthermore, this would enable the unification of the standard permission system (targeted at classes) and the AOPS (targeted at aspects). Hence, policies can be specified and enforced uniformly. However, this option is hard to achieve in practice since (i) the Java compiler does not include an aspect weaver and (ii) this would require adaptations to the virtual machine, which then has to be replaced on all machines where the AOPS is used. Therefore, we opted for the strategy used by most weavers: extending (or replacing) the base language compiler and relying on add-on libraries for the necessary run-time support.

AOPS has been implemented using the AspectBench Compiler (ABC [6]). ABC is an extensible compiler and weaver framework that implements the AspectJ language, an ideal basis for the implementation of AOPS.

The prototype consists of two core components: a (modified) compiler/weaver and a run-time library. The compiler part is responsible for identifying all locations where aspect switches occur (advices, aspect methods and so forth) and for adapting these to include aspect-specific invocations to the `PermissionManager` in order to update the current rights. The compiler must also make sure that no aspects can influence the `PermissionManager` or the constructors of `Permissions`. The run-time library contains the `PermissionManager`, which is responsible for managing and safeguarding the current rights, and a `Policy` component that represents the static aspect policy which is read from file (via the `PolicyParser`).

The AspectJ language was extended with a `Grant` and an `Accept` keyword, which are type checked and transformed into regular Java code in the early stages of compilation. This modification is independent of the weaving strategy. The `PermissionManager.demand()` method is safeguarded from aspect interactions by aborting compilation if any joinpoint matches any call towards the PermissionManager. A similar method is used to safeguard the constructors of `Permissions`. The internal methods in the `PermissionManager` that are used to implement Grant and Accept commands are safeguarded by offering the type checker a version of the PermissionManager that does not have these methods.

6.2 Development Scenario

In this section we will illustrate how some of the policies that were discussed in Section 4.6 can be enforced using AOPS and give an indication of the development overhead. We consider three stages of the development process: security protocol design, extension protocol design and implementation. The security protocol defines which permissions are required for each sensitive operation. The extension protocol defines where and under which conditions the current rights are elevated. The implementation consists of placing all required security annotations, (`Grant`, `Accept` and `demand`) and create the policy file, as described in previous sections.

The remainder of this section will discuss these phases, based on the example application, as described in Section 2.3.

Security protocol

The design phase requires a thorough analysis of the structure of the server, as described in Section 2.3. This allows us to identify the important security areas of the application. In this case, we identify three security areas: 1) the server configuration, 2) the command handling and the session state (as shown in Figure 1) and 3) the authorization process and the authorization state. Each security area has its own custom permission:

- `ConfigPermission` is a coarse-grained permission used to secure the configuration of the server. Before the server is started, its configuration is read from file and stored in a `Config` object. Every attempt to alter the configuration is blocked by an aspect that demands `ConfigPermission` (Guard pattern). This means that the configuration can only be altered if all previously executed aspects had `ConfigPermission` in their static rights.

  The `ConfigPermission` is coarse-grained, in a sense that if it is in the current rights, all configuration is accessible and if it is not, no configuration is accessible.

- `CoreLoopPermission` is a fine-grained permission that secures the main server loop as depicted in Figure 1. To achieve this, a grant-demand pattern is used. Before a message is received from the client, the `CoreLoopPermission` is granted. As long as this permission remains in the current set, the processing of the command is trustworthy. Before the results are returned to the client, the `CoreLoopPermission` is demanded. This makes sure that if the message was not handled properly, only error messages can be sent to the client.

  The `CoreLoopPermission` is also used to protect the session state. Each attempt to read or modify the state is blocked by an aspect that demands the `CoreLoopPermission` (Guard pattern). To allow fine-grained extension, each field containing session state is protected by a `CoreLoopPermission` with a specific name.

- `AuthPermission` is used to protect the authentication aspects, similar to the use of the `CoreLoopPermission`. Before authorization, `CoreLoopPermission` is demanded, to make sure the message that is authorized has not been tampered with. Then a grant-demand pattern is used to safeguard the flow of control. All authorization specific state is also protected by guards.

The aspects in the 'chroot' plug-in, including the malicious aspects, are granted permissions to read the directory to be served and alter the part of the session state that

contains the current directory. This allows no malicious behavior, except for sending small amounts of data through the error messages sent to the client.

Extension protocol

The main concern that drives the extension protocol design is the principle of least privilege: every aspect must only be granted the rights it needs to fulfill its task. Due to the nonincreasing nature of the current rights, this implies that if two aspects have unrelated tasks (and as such disjoint static rights) the current rights will be empty after they have both executed. In order to allow fine-grained extensions while respecting the principle of least privilege, the current rights must be elevated after each extension point.

When using OOP, the current rights are usually elevated when a trusted API is entered. The trusted API validates its arguments and then elevate the current rights. When using AOP however, it may not be clear where the untrusted code will intercept the trusted code. Conversely it is not always clear where the validation and the rights extension must take place. Therefore the upfront design of an extension protocol is required, to define all legal areas of extension.

In our example, we allow three types of extension, one for each permission type:

- `ConfigPermission.` Any aspect having `ConfigPermission` has full access to any part of configuration and the configuration process. After configuration is loaded, `ConfigPermission` is demanded to make sure the configuration is trustworthy. Then all rights required for further operation are granted, to make sure the required system resources are accessible.

- `CoreLoopPermission.` The core loop is executed once for each command that is received. After each loop all static rights of the `Dispatcher` are restored (Listing 9). This can be done safely, as each loop only depends on session state, configuration and user input, which are all guarded (Kernel-Accept pattern). The accept is required because aspects influencing the core loop may not have access to all parts of the session state and all required system resource. If the accept were missing and the loop would be executed a second time, some required permissions might be missing, causing the system to fail.

  This approach allows extension of the core loop by aspects holding the `CoreLoopPermission`. However ordering of operations within one loop is important. If an aspect without rights to system resources interferes with command handling, any subsequent access to system resources will be blocked. A more fine-grained extension protocol would grant rights to system resources in all handlers if `CoreLoopPermission` is in the current rights. We chose not to do this.

- `AuthPermission.` AuthPermission has an extension protocol similar to the `ConfigPermission`.

Implementation

The security protocol is almost entirely implemented using aspects. The use of aspects allows efficient deployment of guards over large bodies of code. The extension protocol is more complex and required more manual annotations of the source code. During its implementation, many subtle feature interactions were detected and refinements of the design were required. Creation of the policy file itself is almost trivial once the protocol had been implemented.

In terms of lines of code, we measured a 6.5% overhead: the unsecured server has about 4280 lines of codes, the secured version 4559 [2].

6.3 Runtime Overhead

The run-time overhead introduced by the model is dependent on a number of factors: the statements introduced by the developer, which was discussed previously, the joinpoints that involve aspect switches, the aspect policy and the complexity of permissions. For each joinpoint, a call is made to the `PermissionManager` and an intersection is calculated. The run-time impact of the aspect policy is high for aspects that are assigned many permissions, since the intersection of permission sets is computed frequently and this can be a costly operation. Finally, also the complexity of permissions influences the run-time overhead: the cost of *intersection*, *union* and *implies* operations can considerably differ between permission types.

To get a better view on the run-time overhead, we benchmarked the trusted core of the jFTPd server. We tested a normal operation scenario and a worst case scenario. The normal operation scenario used the FTP server over the loopback interface and downloaded a 100 small files. The worst case scenario was identical, but all file access was replaced by no-ops. This makes sure that file access is not the dominant factor. Note that the FTP server is a multithreaded application, which is not well supported by the AOPS prototype. However, since our experiment uses a single connection at a time, it was possible to ensure that this did not affect the validity of the experiment.

	Normal		No-op io	
	basic	**secured**	**basic**	**secured**
σ	0.0055	0.0117	0.001	0.001
μ	12.2933	12.3126	0.08	0.23
overhead	0%	0.16%	0%	200%

Table 2: Execution times for the different FTP experiments in seconds, with standard deviation and and mean

These results demonstrate that the impact of using AOPS are minimal (under 1%) compared to executing the software without AOPS. However with CPU intensive tasks, the overhead is significant (200%).

A second performance benchmark was performed on the product-line example from the AspectJ benchmark [12]. This test did no `demand`, `grant` or `accept` operations and measured only the update operations. Each run performed roughly 0.9 million update operations. The test was ran in four versions. The first version did no security updates as it was not instrumented by our weaver. The second version perfomed the security updates with an empty policy. The third version uses a simple policy assigning the same permission to every aspect. The final version uses a complex policy assigning a few different permissions to each aspect of which one was shared between all of them.

[2] counted using ‘wc’

	Unsecured	Empty	Single	Complex
σ	0.04	0.04	0.04	0.05
μ	0.84	1.05	1.13	1.63
overhead	0%	25%	34%	94%

Table 3: Execution times for the different product-line experiments in seconds, with variance and and mean

These results clearly show that the overhead rises with the complexity of the policy and with the size of the current set, as was to be expected. Caching would most likely reduce the overhead to a number close to the Single case, as the intersections would have to be calculated only once, as opposed to several thousand times.

7. DISCUSSION

First of all, AOPS increases the reliability of aspect-oriented software. Currently, no extensive validation has been performed, but based on a number of internal experiments and on our familiarity with permission-based models, we are confident that the AOPS will be a valuable add-on to the existing set of aspect-oriented tools.

Furthermore, in addition to the basic goal of AOPS, the infrastructure also supports identifying possible feature interactions. A feature interaction might not only be caused by two aspects influencing a single joinpoint; more indirect feature interactions can also exist. For instance, two aspects influence a single joinpoint and a third aspect influences one of those via some other joinpoint. Such indirect (or *transitive*) interactions are typically harder to identify. Using AOPS, an application can only execute correctly if all feature interactions are controlled: either by assigning sufficient permissions to aspects, or by augmenting rights. As such, the model can help in tracking unknown feature interactions.

While the AOPS has been shown useful in addressing a wide spectrum of security policies, the model has a number of inherent issues that are important to consider when using the model.

- The execution history-based model is rather strict: a number of applications that would run correctly and securely in practice, will not be allowed to execute. This is an implication of the choice of a model in which all the problems as described in Section 2 can be addressed. Moreover, compared to stack inspection, the history-based model provides a safe default for potentially dangerous situations. A more optimal strategy would consist in enforcing an information flow model and use this as a basis for deciding which permissions to grant and revoke. Unfortunately, information flow models have a very large run-time overhead and they are most often provided as off-line protection mechanisms (such as Jflow [27]).

- The model is based on execution history and, hence, the order in which aspects are applied is important. The ordering of aspects (which can be controlled, to some extent, using the `declare precedence` construct in AspectJ) influences how a program executes and determines whether it meets the global property of the AOPS model. The problem is that it is not always possible for a developer to, given a particular aspect policy file, write code that is *guaranteed* to work correctly once woven (e.g., in case of unspecified orderings, the weaver will decide on the order seemingly randomly). This problem can be addressed by either improving the aspect ordering or, less likely, relaxing the security properties that have to be enforced.

8. RELATED WORK

This work is the first to present a permission system that is capable of safeguarding software developed using aspect-oriented programming. In that sense, no earlier work exists with which the AOPS system can be compared to its full extent. However, there are many results in a number of related research domains.

The performance and usability of run-time enforcement can be improved by attuning it to the type system and language features of the underlying language. In this paper we have presented a generic implementation, covering different features. In a more restricted system, where for example privileged aspects are regulated by the type system, policies can be expressed and evaluated more efficiently. Much work has been done on the subjects of typing, safety and language design in the context of aspect orientation. We briefly highlight some of the approaches. Join point encapsulation [24], for instance, is a technique that supports the shielding of inner parts of a module from aspects. Similarly, by specifying the pointcuts that can be used for aspects more explicitly in pointcut interfaces [17, 22], the internals of modules can be safeguarded. More recent work in the area of type systems is also relevant in this context. For instance, the concept of an aspect is redesigned in open modules [4] to enable modular reasoning. Similarly, harmless advice [8] restricts the way in which advice can influence the computation. Moreover, the detection of aspect-interactions [3] and weaving-interactions [21] at development time can reduce the run-time overhead and facilitate development.

Another possibility for optimization is the use of inline reference monitors, which use program rewriting techniques to enforce security policies in a software artifact [33]. Different types of security automata [14, 25, 15, 18] represent a security policy as a state machine, which is then transformed and inlined into the software at relevant places. In [13], the inline reference monitor technique has been used to implement stack inspection. A more efficient implementation of stack inspection is the security-passing style as described in [35].

Apart from stack and history based security, other models exist. The most promising is information flow based security [30, 32, 28, 5]. This model is more fine-grained and powerful in comparison with the history based model. However, run-time enforcement of information flow based security requires even more security updates compared to HBAC. This may cause a significant run-time overhead. It remains a promising alternative and part of the future work.

There are various alternatives for the expression of policies. Security policies in Polymer [7], for example, are defined based on two types of methods: query methods that determine how a policy should be handled for a security-sensitive action and state update methods to manage the security state of a policy. This allows programmatic combination of policies. The specification language for policies is more restricted compared to AspectJ-like languages. Cur-

rently it is not compatible with AOP. For security enforcement in AOP, there is also the expressive approach proposed by Kallel et al. [19]. This policy language supports very natural expression of security concerns, but relies on complete knowledge of the execution history, which may be hard to realize in practice.

To apply our approach in practice, the technicalities of the run-time environment have to be taken into account. This would raise some additional issues. A first important concern is the interaction of aspects to the class loading mechanism used in Java as described by Sewe et al. [34]. Taking into account classloading mechanics also invalidates the assumption that an aspect can be identified by its name. Furthermore, the integration with the existing Java Security Infrastructure could be redesigned to achieve a more uniform model. Most likely this would mean that permissions are not assigned to individual aspects, but to their origin (i.e. their jar file or issuer) as is the case in the Java Security Infrastructure. This simplification would also significantly impact the identification mechanism. Finally, the semantics of a privileged action in Java base-code would have to be redefined. These issues are still under investigation.

Aspect-oriented programming itself has been used to enforce security policies within applications [10, 31]. One closely related result in the context of this work is the enforcement of an execution history-based access control model using AOP [26]. The most important difference with the AOPS model is that most of the above results are still susceptible to the problems that were discussed during this paper and, hence, they only execute securely under very controlled circumstances.

9. CONCLUSION

This paper presents AOPS, a permission system for aspect-oriented software that is based on execution history. The primary goals of AOPS is to control and minimize the harm that advices of untrusted aspects can cause at run time by enforcing policies on these aspects. To this aim, the developer annotates the software (classes and aspects) with extra statements to trigger the AOPS run-time system. During the weaving phase, the software modules are modified such that global security properties can be correctly maintained at run time. A configurable aspect policy, which assigns permissions to aspects, is consulted at run time to check whether aspects have sufficient rights for executing their behavior.

Experiments have shown that the AOPS increases the reliability and security of aspect-oriented programs. More extensive validation is necessary to improve our understanding of the most effective policies. This will allow the finetuning of the language extensions and the weaver behavior to improve the ease of use of the system. Furthermore, this can aid in extending the basic permission set.

The most important drawback of the permission system is the restrictiveness of the execution history-based model. The effort required to get applications with complex aspect interactions operational using non-trivial policies may become considerable. On the other hand, the benefit of this investment is that all possibly harmful interactions must be identified and appropriately dealt with, which contributes to the overall quality of the software.

The development overhead of using and constructing this security model are smaller than expected, due to the use of advanced AOSD techniques. With limited implementation effort, a powerful enforcement model can be prototyped. The prototype is not overly complex and as such open to extension and refinement. The use of AOSD for research into run-time enforcement of security policies is to be recommended [26].

Our future work will in the first place study and optimize the performance overhead of this infrastructure in the context of complex policies and multi-threaded applications. Furthermore we will investigate the impact of inter type declarations on security, and explore solutions to these problems. Another interesting track is the study of specific aspect-oriented run-time concepts, such as cflow, and the investigation of policies that can be enforced when using such constructs.

10. REFERENCES

[1] The aspectj programming guide. `http://www.eclipse.org/aspectj/doc/released/progguide/`, Dec 2009.

[2] Abadi, M., and Fournet, C. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium* (2003), Internet Soc, pp. 107–121.

[3] Aksit, M., Rensink, A., and Staijen, T. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development* (2009), ACM, pp. 39–50.

[4] Aldrich, J. Open modules: Modular reasoning about advice. In *ECOOP 2005 - Object-Oriented Programming: 19th European Conference* (2005), Springer-Verlag, pp. 144–68.

[5] Austin, T. H., and Flanagan, C. Efficient purely-dynamic information flow analysis. In *PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security* (2009), ACM, pp. 113–124.

[6] Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhotak, J., Lhotak, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. abc: an extensible aspectj compiler. In *Proceedings of the 4th international conference on Aspect-oriented software development* (2005), ACM, pp. 293–334.

[7] Bauer, L., Ligatti, J., and Walker, D. Composing security policies with polymer. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation* (2005), ACM, pp. 305–314.

[8] Dantas, D. S., and Walker, D. Harmless advice. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2006), ACM, pp. 383–396.

[9] De Win, B. *Engineering application-level security through aspect-oriented software development.* PhD thesis, KULeuven, March 2004.

[10] De Win, B., Joosen, W., and Piessens, F. Developing secure applications through aspect-oriented programming. In *Aspect-Oriented Software Development.* Addison-Wesley, 2004, pp. 633–650.

[11] De Win, B., Piessens, F., and Joosen, W. How secure is aop and what can we do about it? In *Proceedings of the 2006 international workshop on Software engineering for secure systems* (2006), ACM, pp. 27–34.

[12] Dufour, B., Goard, C., Hendren, L., De Moor, O., Sittampalam, G., and Verbrugge, C. Measuring the dynamic behaviour of aspectj programs. *ACM SIGPLAN Notices 39*, 10 (2004), 150–169.

[13] Erlingsson, Ú., and Schneider, F. Irm enforcement of java stack inspection. In *Proceedings of the 2000 IEEE symposium on security and privacy* (2000), IEEE, pp. 246–255.

[14] Erlingsson, Ú., and Schneider, F. B. Sasi enforcement of security policies: a retrospective. In *Proceedings of the 1999 workshop on New security paradigms* (1999), ACM, pp. 87–95.

[15] Fong, P. W. L. Access control by tracking shallow execution history. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy* (2004), pp. 43–55.

[16] Gong, L., Ellison, G., and Dageforde, M. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley Professional, 2003.

[17] Griswold, W. G., Shonle, M., Sullivan, K., Song, Y. Y., Tewari, N., Cai, Y. F., and Rajan, H. Modular software design with crosscutting interfaces. *IEEE software 23* (2006), 51.

[18] Hamlen, K. W., and Jones, M. Aspect-oriented in-lined reference monitors. 11–20.

[19] Kallel, S., Charfi, A., Mezini, M., Jmaiel, M., and Klose, K. From formal access control policies to runtime enforcement aspects. 16–31.

[20] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An overview of aspectj. *Lecture Notes in Computer Science* (2001), 327–353.

[21] Kniesel, G. Detection and resolution of weaving interactions. *Transactions on Aspect-Oriented Software Development V Volume 5490/2009* (2009), 135–186.

[22] Lagaisse, B. Explicit interfaces for robust ao-composition. In *A Comprehenisive Integration of AOSD and CBSD Concepts in Middelware*. December 2009, pp. 79–136.

[23] LaMacchia, B. A. . *NET Framework Security*. Addison-Wesley/Pearson Education, 2002.

[24] Larochelle, D., Scheidt, K., Sullivan, K., Wei, Y., Winstead, J., and Wood, A. Join point encapsulation. In *Proceedings of the AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies* (2003).

[25] Ligatti, J., Bauer, L., and Walker, D. Edit automata: enforcement mechanisms for run-time security policies. *International Journal of Information Security 4*, 1 (2005), 2–16.

[26] Minevskiy, I. Aspectizing security concerns. `http://www3.telus.net/minevskiy/ivan/data/539c/IMinevskiy_AOP.pdf`, april 2005.

[27] Myers, A. C. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1999), ACM, pp. 228–241.

[28] Nair, S. K., Simpson, P. N. D., Crispo, B., and Tanenbaum, A. S. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science 197*, 1 (2008), 3–16.

[29] Nenninger, B. jftpd, ftp server with remote administration. `http://macfreebees.tripod.com/readme/jFTPd.html`, dec 2009.

[30] Pistoia, M., Banerjee, A., and Naumann, D. A. Beyond stack inspection: A unified access-control and information-flow security model. In *2007 IEEE Symposium on Security and Privacy. To appear* (2006), IEEE, pp. 152–166.

[31] Ramachandran, R., Pearce, D. J., and Welch, I. Aspectj for multilevel security. In *The 5th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)* (2006), pp. 1–5.

[32] Roy, I., Porter, D. E., Bond, M. D., McKinley, K. S., and Witchel, E. Laminar: practical fine-grained decentralized information flow control. *SIGPLAN Not. 44*, 6 (2009), 63–74.

[33] Schneider, F. B., Morrisett, G., and Harper, R. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead* (2000), Springer, pp. 86–101.

[34] Sewe, A., Bockisch, C., and Mezini, M. Aspects and class-based security: a survey of interactions between advice weaving and the java 2 security model. In *VMIL '08: Proceedings of the 2nd Workshop on Virtual Machines and Intermediate Languages for emerging modularization mechanisms* (New York, NY, USA, 2008), ACM, pp. 1–7.

[35] Wallach, D. S., Appel, A. W., and Felten, E. W. Safkasi: a security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology (TOSEM) 9*, 4 (2000), 341–378.

[36] Wallach, D. S., and Felten, E. W. Understanding java stack inspection. In *IEEE SYMPOSIUM ON SECURITY AND PRIVACY* (1998), IEEE COMPUTER SOC, pp. 52–63.

[37] Weissman, C. D., and Bobrowski, S. The design of the force.com multitenant internet application development platform. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data* (2009), ACM, pp. 889–896.