Secure and Modular Access Control with Aspects

Rodolfo Toledo Éric Tanter

PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Chile
www.pleiad.cl

Abstract. Can access control be fully modularized as an aspect? Most proposals for aspect-oriented access control are limited to factoring out access control checks, still relying on a non-modular and ad hoc infrastructure for permission checking. Recently, an approach for modular access control was proposed, called ModAC. ModAC successfully modularizes both the use of and the support for access control by means of restriction aspects and scoping strategies. However, ModAC is only informally described and therefore does not provide any formal guarantee with respect to its effectiveness. In addition, like in many other proposals for aspect-oriented access control, the presence of untrusted aspects is not at all considered, thereby jeopardizing the practical applicability of such approaches. This paper demonstrates that it is possible to fully modularize aspect control, even in the presence of untrusted aspects. It does so by describing a self-protecting aspect that secures ModAC. We validate this result by describing a core calculus for AspectScript, an aspect-oriented extension of JavaScript, and using this calculus to prove effectiveness and non-interference properties of ModAC.

1 Introduction

Access control [24] is a cornerstone of every security architecture: it is the component in charge of ensuring that sensitive resources are accessed only by authorized entities.

In modern runtime environments such as the JVM [15] and the CLR [4], access control architectures rely on a fine-grained specification based on permissions. 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 these architectures allows one to assign different sets of permissions to different entities. Furthermore, stack inspection [14] is used to dynamically examine if a sensitive operation can be performed or not. This is known as basic permission checking.

The Java access control architecture also includes two other mechanisms: privileged execution and permission contexts. Privileged execution allows a trusted entity to take responsibility for a certain action. This makes it possible for untrusted entities to access sensitive resources—such as the screen—in a controlled manner. Permission contexts allow the programmer to capture the set of permissions at a certain point and restore it later on, for instance to incrementally perform a long task—such as classloading—in different threads safely.

While these three mechanisms together provide a very powerful access control system, they also introduce modularity issues. Indeed, using basic permission checking is a crosscutting concern: in order to trigger stack inspection, explicit calls to the access control architecture are 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 addition to the crosscutting nature of the *use* of access control, the *implementation* of access control is itself non-modular. For instance, the Java access control architecture is implemented in part in the Java libraries (the stack inspection algorithm), and in part in the JVM (reification of permission contexts). This native support in the VM is specific to (and can only be used for) access control enforcement. This tends to suggest that access control needs to be supported as a primitive in the language, and that therefore, access control is not something that can be plugged into an existing language without having to modify its semantics.

Considering these modularity issues, and the fact that security has long been considered a typical aspect, this paper addresses the following research question:

Can access control be fully *modularized as an aspect?*

Here, we are concerned not only with modularizing the use of access control—a somewhat easy and well-explored problem [35,34,25,9,18,23,21]—, but with expressing the *whole* access control infrastructure as an aspect, including the support for advanced features ignored in the literature, like privileged execution and capturable permission contexts. By "fully" modularizing access control, we refer to the question: *is it possible to leave the programming language semantics* completely *oblivious to the presence of access control?* If so, can we ensure that malicious code, including other aspects, do not interfere with the access control aspect, and how? What are the requirements on the underlying general-purpose aspect language?

A positive answer to these questions should also contribute the formulation of a general-purpose aspect model that can be used to add access control to languages that do not include any support for it, like JavaScript. Indeed, in previous work [31], we have explored how it is possible to aspectize stack-based access control with support for privileged execution and capturable permission contexts. The approach, called ModAC (for Modular Access Control) consists of expressing access control using *restriction aspects* scoped with an appropriate *scoping strategy* [26]. Restriction aspects modularize the use of access control whereas scoping strategies make it possible to modularly provide basic permission checking, privileged execution, and capturable permission contexts.

The ModAC approach was instantiated in AspectScript, an aspect-oriented extension of JavaScript that supports scoping strategies [30]. The resulting implementation (hereafter called ModAC/AS) was used to provide an extensible access control library for JavaScript, called ZAC [33]. However, previous work on ModAC answers only part of the above research question. First, the formulation of ModAC is only informal; its actual effectiveness in controlling accesses to sensitive resources has not been proven. Second, it leaves open the possibility for untrusted aspects to interfere with access control aspects, thereby ruining its effectiveness.

Contribution. This paper shows that it is indeed possible to fully modularize access control as an aspect, even in the presence of untrusted aspects. The approach is based on introducing a *self-protecting* restriction aspect that impedes untrusted aspects to interfere with critical access control components (Sect. 3). In order to validate our approach formally, we develop λ_{AS}, a core calculus for AspectScript based on λ_{JS} [16] (Sect. 4). λ_{AS} is general-purpose and therefore oblivious to access control; it is a major side contribution of (and not restricted to) this work. We prove the desired effectiveness and non-interference properties of an instantiation of ModAC in λ_{AS}, ModAC/λ_{AS} (Sect. 5) and discuss the extension of the result to ModAC/AS (Sect.6).

Section 2 briefly introduces access control, and aspect-oriented approaches to it, in particular ModAC. Section 7 describes related work and Section 8 concludes. The full proofs of the results exposed in Section 5 are available online [32]. This work is fully implemented in the ZAC library for AspectScript; also, the executable formal model of λ_{AS} is implemented in PLT Redex [12].

2 Background & Motivation

We briefly introduce stack-based access control, illustrating its main features (Section 2.1). We then describe aspect-oriented approaches to access control, including ModAC (Section 2.2). Finally, Section 2.3 classifies various threats to modular access

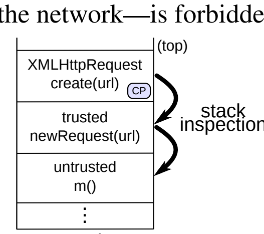
2.1 Access control by example

In this section we describe the three access control features based on stack inspection: basic permission checking, privileged execution, and permission contexts. We illustrate each one with real-world examples from the JavaScript realm.

Basic permission checking. When a sensitive resource is about to be accessed, a call to the access control infrastructure triggers a stack inspection algorithm [14], which *checks* whether all the entities in the current stack of execution (starting from the top of the stack) possess the necessary permission to access the resource. If not, an exception is thrown. Stack inspection is triggered in Java by calling SecurityManager.checkPermission, passing it the required permission; in C#, this is done by invoking Demand() on a permission object. In both systems, the entities to which permissions are assigned to are classes. In the following examples, permissions are assigned to individual objects, since JavaScript is prototype based.

This basic behavior prevents the confused deputy problem [17] from happening: an untrusted entity cannot lead a trusted one to access a sensitive resource on its behalf by simply invoking a method, because the stack inspection algorithm will eventually notice the presence of the untrusted entity on the stack. This is exemplified in the following piece of code, in which accessing a sensitive resource—the network—is forbidden:

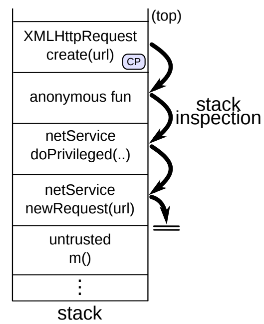
```
var trusted = {
  newRequest: function(url){
    return new XMLHttpRequest(url);
  } };
var untrusted = {
  m: function(){
    var req = trusted.newRequest("...");
  } };
untrusted.m();
```



When the function m is executed, the untrusted object invokes newRequest on trusted to create a new XMLHttpRequest object. Assuming that the stack inspection algorithm is triggered as in Java with a call to checkPermission (signaled by the CP gray square on the above figure), the instantiation is prevented by throwing an exception. This is so because the stack inspection algorithm eventually checks the permissions of untrusted and discovers that it does not hold the necessary permission to access the network.

Privileged execution. In some scenarios, it is necessary for an entity to access a sensitive resource on behalf of another—possibly untrusted—entity. For this, the JVM supports *privileged execution*. For instance, suppose that we want to provide a netService object that allows any client to access the network, provided that the target site pertains to a list of known sites. In this case, the creation of an XMLHttpRequest object should be allowed even when there are untrusted objects participating in the current call stack.

```
var netService = {
  newRequest: function(url){
    if(/* url pertains to the known sites list */){
      return this.doPrivileged(function(){
        return new XMLHttpRequest(url);
      });
    }
    return null;
  } };
var untrusted = {
  m: function(){
    var req = netService.newRequest("...");
  } };
untrusted.m();
```



A self call to doPrivileged initiates a privileged action[1]. Consequently, stack inspection only considers the permissions of objects on the stack corresponding to the dynamic extent of the privileged action, *including* the initiator of the action; *i.e.* the stack inspection algorithm stops at the frame of the initiator of the call to doPrivileged.

Permission contexts. When accessing a sensitive resource, it can be necessary for an entity to use the permissions present at another point in the execution of the application. The JVM provides the means to capture a *permission context* and restore it later on.

In JavaScript, this can be used to capture the permission context at the time a network connection is initiated, and reinstall it when the response from the server is received (asynchronously). This way, the response processing is performed with the same permissions as the call, similarly to a synchronous communication. Note that since JavaScript is a single-threaded language, this is the only way to correctly manage the switch permissions.

2.2 Access control with aspects

Due to its inherently crosscutting nature, access control has been a repeated target for applying aspects. We briefly explain these approaches in the following, and then dive into a recent proposal for fully modularizing access control.

[1] As opposed to Java, where a privileged action is started by calling the static doPrivileged method of the AccessController class.

Permission aspects. The most obvious source of crosscutting due to access control is the necessity of explicitly triggering stack inspection upon access to sensitive resources. Many approaches based on aspects have been proposed in order to factor out these calls into advices [35,34,25,9,18,23,21]. In all these approaches, aspects follow the same pattern: their pointcuts match accesses to sensitive resources, and their advice triggers access control. For example, the following aspect, declared in AspectScript [30], guards the accesses to the network:

```
var netPermission = {
  pointcut: function(jp){ return jp.kind == new && jp.fun === XMLHttpRequest; },
  advice: function(jp){
    checkPermission(new Permission("network")); //triggers stack inspection
    return jp.proceed();
  } };
```

This aspect[2] successfully modularizes the triggering of basic permission checking for network accesses. Aspects following this pattern are classified as *permissions aspects* due to their use of the permissions infrastructure and the stack inspection algorithm [31].

Restriction aspects. While permission aspects modularize calls to check if the necessary permissions are available, they do not *fully* modularize access control; indeed, they rely on additional libraries and support from the runtime environment in order to perform stack inspection. Recently, we described an approach for fully modular access control, ModAC [31], based on restriction aspects and scoping strategies.

In contrast to permission aspects, *restriction aspects* do not rely on any permission infrastructure or stack inspection algorithm. Instead, the *scoping mechanisms* of the aspect language are used to ensure proper resource protection. A restriction aspect works by adhering to a different, dual pattern: the pointcut selects accesses to a sensitive resource (just like a permission aspect), but the advice immediately aborts the access by not proceeding with the primitive operation; scoping strategies are used to ensure that the aspect only *sees* forbidden accesses. Consider the following restriction aspect:

```
var netRestriction = {
  pointcut : function(jp){ return jp.kind == new && jp.fun === XMLHttpRequest; },
  advice: function(jp){ throw "Cannot access the net."; }
};
```

This aspect forbids the access to the network. Its pointcut identifies instantiations of XMLHttpRequest objects, and the advice throws an exception with an informative message. Another possibility is not to throw an exception but to silently abort the sensitive resource access. For instance:

```
var alertRestriction = {
  pointcut : function(jp){ return jp.kind == exec && jp.fun === alert; },
  advice : function(jp){ /* do nothing */ }
};
```

This restriction aspect simply annihilates the execution of the alert method, in order to prevent the degradation of the user experience.

[2] Aspects are plain objects in AspectScript. They have one pointcut and one advice, defined by the pointcut and advice attributes respectively. Both pointcuts and advices receive a join point as parameter. All advices are around advices.

The scoping strategy for access control. Restriction aspects are limited to see only illegal resource accesses by means of scope control. However, scope control based on control flow only, as provided by AspectJ, is insufficient to directly support features like privileged execution and permission contexts [31]. For this reason, ModAC relies on a more expressive scoping control mechanism, *scoping strategies* [26,29,27].

A scoping strategy permits fine-grained control over the scoping semantics of a deployed aspect. 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[3]. Intuitively, the former allows controlling dynamic scoping of aspects, stopping propagation when a certain condition is met. The latter allows an aspect to follow an object: the aspect sees all join points occurring *lexically* within all methods of the object (and may potentially propagate further in method calls done by the object depending on the call stack propagation function). Propagation functions are predicates over join points: the call stack propagation function matches *call* join points for which the aspect should propagate, while the delayed evaluation propagation function matches *object creation* join points.

Scoping strategies in AspectScript are provided as an (optional) first argument to the aspect deployment constructs: deploy(s,asp,fun), which deploys the aspect asp on the body of fun; and deployOn(s,asp,obj), which deploys asp on the object obj. In both cases, s is a scoping strategy, and asp can be a single aspect or an array of aspects.

The scoping strategy for access control that supports basic permission checking, privileged execution, and permission contexts is:

```
var acs = [ //access control strategy
  function(jp){ return !(jp.fun === doPrivileged && jp.target === jp.context);},
  function(jp){ return jp.target instanceof ACContext; }
];
```

The call stack propagation function expresses both basic permission checking and privileged execution. Essentially, it specifies that a restriction aspect always propagates on the call stack, except on privileged calls. A privileged call is a self call to doPrivileged (a self call occurs when jp.target, the target of the call, and jp.context, the currently executing object, are equal). This way, a restriction aspect propagating through the stack stops its propagation upon a privileged call, and hence does not see resource accesses that occur in the control flow of that call. Only considering self calls for privileged execution permits to maintain the aspects of the object initiating the action.

The delayed evaluation propagation function expresses the capture of permission contexts. It ensures that restriction aspects propagate to instances whose prototype is ACContext; therefore, creating such an object is a means to take a snapshot of the restriction aspects present at that point in time. Later on, it is enough to include these objects in the stack to restore the permission context. This is done by an overloaded version of doPrivileged that accepts an ACContext as extra parameter—more details can be found in [31].

Figure 1 depicts the propagation of an aspect asp deployed with the access control strategy. If asp is currently deployed (*i.e.* it is in the current aspect environment), it prop-

[3] Scoping strategies also include a third component, called *activation function*. Activation is not used in this work, so we omit it.

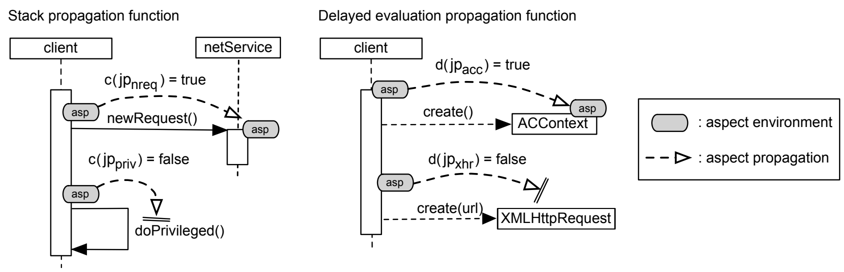


Fig. 1. Propagation of aspects with the access control strategy.

```
var Deployer = {
  acs: ..., //access control strategy
  pc: function(jp){ return jp.kind == new; }, // creation of objects
  adv: function(jp){
    var obj = jp.proceed();
    var restrictions = getRestrictionsFor(obj);
    deployOn(acs, restrictions, obj); //per−object deployment
    return obj;
  } };
deployOn([false,true],Deployer, function(){/∗ main program ∗/});
```

Fig. 2. Deployer aspect for deploying restriction aspects.

agates on calls to newRequest (jp_{nreq}) but not on self-calls to doPrivileged (jp_{priv}). Therefore asp sees join points occurring during the execution of newRequest. Similarly, asp gets captured in new ACContext objects (jp_{acc}), and not in new XMLHttpRequest objects (jp_{xhr}). Hence, asp sees the subsequent activity of these ACContext objects.

ModAC fully modularizes aspect control, by relying only on the aspect language. As a matter of fact, scoping strategies replace the need for an ad-hoc, VM-supported mechanism *specific* to access control, as is the case of access control in the JVM and the CLR. For sure, the aspect language must support scoping strategies; however, scoping strategies are a *general-purpose* construct, with a wide range of applications beyond access control (*e.g.* [26,29,27]).

Bootstrapping access control. Since access control is fully modularized, it is just one more aspect. In order for it to be effective in a given system, it has to be activated. In a language with dynamic aspect deployment, the only way is to do so explicitly in the program (*e.g.* around the main method, around the loading of a script, etc.). In a language with static deployment, access control must still be equivalently activated (*e.g.* on the command line, in a configuration file, etc.).

In the case of ModAC/AS, the activation of access control is performed by wrapping the main program in a deployment of the Deployer aspect (Figure 2). Deployer ensures that the relevant parts of the activity of all objects are under control of restriction aspects. It does so by deploying these restriction aspects on newly-created objects with the access control scoping strategy acs defined previously. Crucially, the deployment of restriction aspects must be done exactly in between the creation of an object and the beginning of its initialization. This way, when the object initiates computation, the necessary restriction aspects are already deployed on it.

The Deployer aspect deploys restriction aspects on objects when they are created. First, its pointcut matches all object creations (line 3). Then, the advice (lines 4-8) deploys the corresponding restriction aspects on the newly-created object (line 5), using deployOn (line 7) and specifying the access control scoping strategy (line 2). Finally, the object is returned (line 8). The set of restriction aspects that corresponds to a particular object is determined by the getRestrictionsFor method (line 6). This method abstracts the process of determining the needed restrictions. A possible implementation is to mimic the access control architecture of the JVM by returning the restriction aspects that correspond to the permissions declared in a policy file. Another implementation is to return restrictions based on dynamic conditions, such as the kind of user currently interacting with the application, as in role-based access control [13]. Line 10 deploys Deployer such that it propagates in all created objects (delayed evaluation is set to true); this ensures that it sees all object creations.

2.3 Threats to modular access control

ModAC seems to be a proof by existence that access control *can* be fully modularized using aspects, provided the aspect language supports sufficiently expressive scoping mechanisms. However, our previous work does not provide any formal guarantee in this respect. Most importantly, it does not consider the threats posed by the presence of other, possibly untrusted, aspects.

Inhibition. De Borger *et al.* showed how easy it is to interfere with access control by means of aspects [8]. For instance, this AspectJ aspect completely inhibits access control in the JVM:

```
public aspect MaliciousAspect{
  void around(): execution(void SecurityManager+.check*(..)){ /* do nothing */ }
}
```

As opposed to the JVM and the CLR, ModAC does not exhibit the previous vulnerability, simply because there are no explicit calls to a stack inspection algorithm. However, there are other alternatives for untrusted aspects to inhibit access control, to which ModAC is vulnerable: *i.e.* to prevent access control components—restriction aspects, the access control strategy, and the Deployer aspect—from achieving their purpose.

We introduce the distinction between implicit and explicit inhibition. *Implicit inhibition* is based on using the aspect weaving mechanism to inhibit access control, such as in the above AspectJ example. *Explicit inhibition* consists of using other means provided by the base language (*e.g.* side effects) to prevent the different components of the access control system to fulfill their role.

Explicit inhibition. There are many kinds of explicit inhibition, depending on the considered programming language. In a purely functional language, it is impossible to alter a function or mutate existing bindings and data structures. But in a stateful world, risks exist if the state of the access control components can be aliased and mutated. Such risks are exacerbated in languages like JavaScript, where it is possible to dynamically remove object members.

Fortunately, explicit inhibition requires the malicious entity to perform explicit actions, which can be observed and prevented by dedicated restriction aspects. For instance, the following restriction forbids any action on netRestriction (*e.g.* modification of its properties, invocation of its methods):

```
var metaNetRestriction = {
  pointcut: function(jp){ return jp.target === netRestriction; }, //any action on netRestriction
  advice: function(jp){ throw "Cannot manipulate the netRestriction aspect"; }
};
```

For any kind of explicit inhibition, a dedicated restriction must be defined. This shows how ModAC elegantly protects itself from explicit inhibition.

Implicit inhibition. Because explicit inhibition can be prevented by menans of restriction aspects, this paper focuses on implicit inhibition. Indeed, implicit inhibition is peculiar because it is directly enabled by the use of an aspect-oriented language; also, implicit inhibition can be achieved in any aspect language, regardless of whether or not the language is purely functional.

In the case of ModAC, there are three kinds of implicit inhibition: pointcut inhibition, advice inhibition, and scoping strategy inhibition. Pointcut inhibition consists in preventing the pointcut of an access control component from matching at relevant join points. For instance, the following malicious aspect inhibits the pointcut pc of a restriction aspect:

```
var maliciousAspect = {
   pointcut: function(jp){ return jp.kind == pcexec && jp.fun === pc; },
   advice : function(jp){ return false; }
};
```

The other kinds of inhibition follow a similar pattern: making a pointcut return false as above, making an advice do nothing by matching its execution but never proceeding, or impeding propagation of restriction aspects by making their propagation functions return always false, etc.

3 R̊: One Aspect to Rule them All

In this section we present R̊ (pronounced "ring"), a self-protecting restriction aspect that prevents untrusted aspects from inhibiting access control in ModAC. We first introduce some terminology to discriminate different kinds of aspects (Section 3.1). We then describe and justify our design goals for secure modular access control (Section 3.2), and a general approach to control untrusted aspects (Section 3.3). We finally present R̊ and explain how it prevents inhibition of both access control and itself (Section 3.4).

3.1 Aspect classification

First, we call *access control aspects* all aspects that are part of ModAC: *i.e.* restriction aspects and the Deployer aspect. We then make the distinction between *trusted aspects*, which should be given unrestricted freedom; and *untrusted aspects*, which are potentially trying to inhibit acces control. Classifying aspects as trusted or untrusted depends on the access control policy of a given application. For example, a possible policy consists in considering all aspects defined in local code as trusted, whereas aspects defined in remote code are deemed untrusted.

In addition, we introduce a set of *protected aspects*. By definition, this set contains all aspects whose inhibition must be prevented. In order to secure ModAC, this set must include all access control aspects (but is not restricted to those aspects).

3.2 Securing ModAC: design goals

Our design goals for secure and modular access control are as follows:

G1 *The base language must be completely oblivious to access control.*
G2 *Untrusted aspects must not inhibit protected aspects, but are free to see other join points.*
G3 *Trusted aspects should see any join point.*

The first goal (G1) is the *raison d'être* of ModAC as discussed previously. Fully modularizing access control with aspects, beyond being an important validation for AOP itself, allows access control to be added to other aspect languages, without requiring ad hoc support for it. The two other design goals are concerned with securing ModAC without restricting too much the programming model.

Design goal (G2) states that the non-inhibition property must be achieved without simply ruling out untrusted aspects. Untrusted aspects must be able to do whatever their access policies specify; the only strong requirement is that they do not inhibit protected aspects. Design goal (G3) states that trusted aspects should be able to see any join point. This goal discards a restrictive approach that prohibits any kind of weaving (trusted or not) in certain core classes—thereby strongly coupling access control and weaving.

For instance, the Aspect-Oriented Permission System (AOPS) [8] ensures non-inhibition by disallowing any kind of weaving at join points lexically located in access control aspects and other sensitive components such as permission classes and the `PermissionManager` class. Doing so impedes even trusted aspects to affect these classes. In addition, it means that the weaver (and hence the aspect language semantics) is specifically tailored to take access control into account, something that we discard as of design goal (G1). Therefore, AOPS violates two of our design goals, (G1) and (G3).

3.3 Preventive inhibition

In order to reconcile goals (G2) and (G3)—*i.e.* preventing the inhibition of protected aspects by untrusted aspects, while allowing trusted aspects to see any join point—we introduce a simple technique: *preventive inhibition*. Preventive inhibition consists in inhibiting untrusted aspects before they inhibit protected aspects.

To achieve preventive inhibition, it is sufficient to ensure that untrusted aspects do not apply at join points occurring in the control flow of protected aspects. For restriction aspects, this means that untrusted aspects cannot interfere with the identification of resource accesses nor with the process of aborting these accesses. For the `Deployer` aspect, this means that untrusted aspects cannot interfere with the identification of object creations nor with the calculation and deployment of restriction aspects.

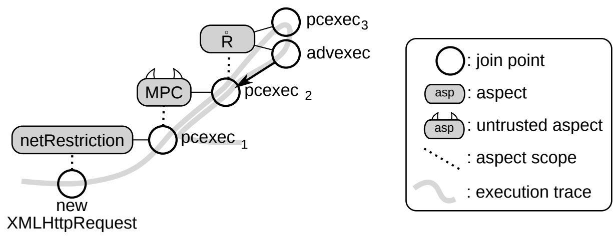


Fig. 3. Pointcut inhibition prevented by R̊.

How can preventive inhibition be achieved while maintaining (G1), *i.e.* without requiring modifications to the aspect language semantics? A tentative answer is to slightly update untrusted aspects by conjuncting their pointcuts with the following:

```
!cflow(function(jp){ return protectedAspects.contains(jp.target); });
```

This effectively makes the pointcuts of untrusted aspects evaluate to false for join points that are in the control flow of join points whose target is in the set of protected aspects.

It is hard to reconcile this global transformation of all untrusted aspects with (G1). Indeed, the required global transformation can be performed by means of general-purpose constructs, such as the global pointcut proposed by the abc team [3]. However, identifying untrusted aspects may depend on knowledge only available at runtime, especially in dynamic languages like JavaScript.

3.4 The R̊ restriction

Fortunately, there is a simple solution to reconcile all three design goals, and it does not require any fundamental extension to ModAC; rather, it is just a pattern of ModAC. The approach relies on using a specific restriction aspect, called R̊. R̊ is in charge of preventive inhibition for all protected aspects, including itself, thereby fulfilling goals (G2) and (G3). Because R̊ is a restriction like any other, goal (G1) is fulfilled as well.

Inhibition with R̊. R̊ is deployed on untrusted objects at creation time, just like other restriction aspects. Its definition is:

```
var R̊ = {
  pointcut: function(jp){
    return jp.kind == pcexec && cflow(function(jp){ return protectedAspects.contains(jp.target); });
  },
  advice: function(jp){ return false; }
};
```

R̊ inhibits every pointcut execution it sees, provided that the execution is in the control flow of a join point whose target is in the protected aspects set. In consequence, all aspects in the protected aspects set cannot be inhibited by untrusted aspects, simply because untrusted aspects do not even get a chance to see the join points they would potentially affect. Note that R̊ is the first-class equivalent of the pointcut conjunction discussed in the previous section. Making it a restriction aspect like any other is the key to enforce this inhibition check without affecting the language semantics.

Illustration. Figure 3 illustrates how R̊ avoids pointcut inhibition by an untrusted aspect MPC on the netRestriction aspect presented before. When a new XMLHttpRequest instance is created, a join point is generated. The netRestriction aspect observes this creation, and therefore, its pointcut is evaluated. This generates a pointcut execution join point (pcexec_1), which is observed by MPC. Consequently, the MPC pointcut is evaluated, which generates another pointcut execution join point (pcexec_2). Since MPC is untrusted, R̊ was deployed on it. Hence, the pointcut of R̊ sees pcexec_2, and matches it (it is a pointcut execution join point and a protected aspect, netRestriction, is in the control flow). In consequence, R̊ inhibits the pointcut of MPC. Advice and scoping strategies inhibitions are prevented in a similar way.

Self-protection. Crucially, R̊ can protect *itself* from inhibition by untrusted aspects, following the exact same principle. To do so, R̊ is added to the protected set. Self-protection of R̊ can be observed in the same Figure 3, by replacing the reference to netRestriction on the figure with R̊. An untrusted aspect can try to inhibit R̊ as many times as it wants in the same flow of execution. If the interaction is infinite, the program does not terminate[4]. If the interaction is finite, R̊ eventually rules the untrusted aspect. Self-protection of R̊ elegantly secures ModAC by not introducing any additional mechanism; R̊ is just a restriction aspect protecting access control aspects, including itself, and other protected aspects, from inhibition by untrusted aspects.

Bootstrapping. R̊ uses the protectedAspects set to identify the aspects it must protect from implicit inhibition. Naturally, untrusted entities must not be allowed to interfere with this data structure. As discussed previously, inhibiting access control by interfering with the protectedAspects set can either be done implicitly via weaving, or through explicit manipulation. Implicit inhibition is already prevented by R̊ itself. As prescribed previously, explicit inhibition is avoided by using a dedicated restriction aspect:

```
var paRestriction = {
  pointcut: function(jp){ return jp.target === protectedAspects; }, //any action on protectedAspects
  advice: function(jp){ throw "Cannot manipulate the protected aspects set"; }
};
```

This restriction follows the same pattern as the metaNetRestriction presented before; it forbids *any* action over protectedAspects. This restriction must be deployed on all untrusted entities at creation time. Note that this restriction is just another restriction, and therefore (G1) is still fulfilled.

4 A Core Calculus for AspectScript: λ_{AS}

The previous section has shown how ModAC can be made secure thanks to the R̊ restriction aspect. However, our descriptions of ModAC and the R̊ restriction aspect so far are informal. First, ModAC itself has never been proven to be effective, even in the

[4] Any untrusted piece of code is (a priori) given the power of the base language (which is Turing-complete) and can therefore always provoke non-termination. Different mechanisms (including restriction aspects!) can be used to avoid this misbehavior (*e.g.* timeout, limit on the number of produced join points), but this is out of the scope of this work [33].

$$
\begin{array}{rl}
Value\ v & ::= c \mid \textbf{fun}(x\cdots)\{e\} \mid o \mid l \\
Bool\ b & ::= \textbf{true} \mid \textbf{false} \\
Const\ c & ::= n \mid str \mid b \mid \textbf{undefined} \mid \textbf{null} \\
Object\ o & ::= \{str : v \cdots\} \\
Expr\ e & ::= x \mid v \mid \textbf{let}\ (x = e)\ e \mid e(e\cdots) \mid e[e] \mid e[e] = e \mid e = e \mid \textbf{ref}\ e \mid \textbf{deref}\ e \\
Store\ \mu & ::= \epsilon \mid \mu + (l \mapsto o)
\end{array}
$$

$n \in \mathscr{N}$, the set of numbers; $str \in \mathscr{S}$, the set of strings; $x \in \mathscr{X}$, the set of variable names;$l \in \mathscr{L}$, the set of locations.

Fig. 4. Syntax of the λ_{JS} language (excerpt; slightly modified).

absence of untrusted aspects [31]. The fact that a working JavaScript library like ZAC based on ModAC has been implemented [33], and has been extended to include R̊, does not prove that the approach is correct. In order to do so, we focus on AspectScript and establish a formal basis for it: the λ_{AS} calculus, described in this section. Section 5 then states formally that ModAC/λ_{AS}, the implementation of ModAC in the λ_{AS} calculus, is both correct and secure. Proofs and executable semantics are provided online [32].

λ_{AS} is a core calculus for AspectScript, developed as an extension of the λ_{JS} calculus [16]. We first give a brief overview of λ_{JS}, and then describe its extension to support aspect weaving with dynamic aspect deployment and scoping strategies.

4.1 Core JavaScript: λ_{JS}

Guha *et al.* designed λ_{JS} as a core subset of JavaScript to which JavaScript programs are desugared. The interest of λ_{JS} is its compactness. We briefly describe the syntax of λ_{JS}, the desugaring process, and a few reduction rules.

Syntax. Figure 4 shows part of the syntax of λ_{JS}. The language has primitive values such as numbers, strings, booleans, and two special values **null** and **undefined**, in addition to functions (**fun**) and objects o. Objects are a series of attribute-value pairs enclosed in curly braces. Expressions include identifiers, values, a **let** construct, function application, property access, and property write. In order to support first-class mutable references, values are augmented with store locations. Objects in the store are explicitly referenced and dereferenced using **ref** and **deref**, respectively. λ_{JS} also includes typical control operators and primitive n-ary operators; we omit these for brevity.

Desugaring. Several JavaScript constructs are left aside of λ_{JS} and are instead expressed via desugaring [16]. For example, the desugaring of function creation is:

```
desugar[[ function(x···){e} ]] =     ref { "code": fun(this, fthis, x···) { desugar[[ e ]] },
                                           "prototype": ref {"__proto__": (deref Object)["prototype"]}}
```

A function is desugared into an object (using the {...} notation) with two attributes: code and prototype. The code attribute is the actual function (note that function is a JavaScript term, and **fun** is a λ_{JS} term). Also, this is an ordinary identifier: it is the first formal parameter of a desugared function. In JavaScript, a method is a function, which is a value, and can be shared between objects; this refers to the currently-executing object. For the sake of properly dealing with aspect environments in λ_{AS}, we slightly extend

λ_{JS} and pass a second parameter to every desugared function; the parameter, named fthis, is bound to the function object thus created by the desugaring process. Note that desugaring reveals some of JavaScript peculiarities: the prototype attribute of a function object is an object whose prototype is the prototype attribute of Object.

The semantics of λ_{JS} is defined as a small-step reduction relation $\hookrightarrow$. A program configuration $\langle \mu, e \rangle$ consists of a store and an expression. The reduction relation is standard. Evaluation contexts [12] are used to specify a call-by-value, left-to-right evaluation semantics. E.g., the reduction rule for object creation is:

$$\langle \mu, E[\textbf{ref}\ \{str : v \cdots\}] \rangle \hookrightarrow \langle \mu', E[l] \rangle \qquad \textsc{New}$$
$$\text{where } l \notin dom(\mu) \text{ and } \mu' = \mu + (l \mapsto \{str : v \cdots\})$$

ref simply allocates a new location in the store and returns it. The function application rule is the standard β_v reduction:

$$\langle \mu, E[\textbf{fun}(x \cdots)\{e\}(v \cdots)] \rangle \hookrightarrow \langle \mu, E[e[v \cdots / x \cdots]] \rangle \quad \textsc{Call}$$

4.2 AspectScript Semantics

We now describe the syntax and operational semantics of λ_{AS}, a core calculus for AspectScript based on λ_{JS}. Its operational semantics is defined via the reduction relation $\hookrightarrow: \mathscr{M} \times \mathscr{A} \times \mathscr{J} \times \mathscr{E} \to \mathscr{M} \times \mathscr{A} \times \mathscr{J} \times \mathscr{E}$.

We extend the λ_{JS} configuration with two additional elements: a λ_{AS} program configuration $\langle \mu, \alpha, J, e \rangle$ consists of a store $\mu \in \mathscr{M}$, an aspect environment $\alpha \in \mathscr{A}$, a join point stack $J \in \mathscr{J}$, and an expression $e \in \mathscr{E}$. The *stack aspect environment* α is used to maintain the aspects propagated through the stack by means of the call stack propagation function[5].

In the following we describe the semantics of join points, aspects and their deployment, as well as the weaving semantics. The formalism is inspired by the formalism previously used by Tanter [28] (for an aspect-oriented variant of Scheme)[6], itself based on a combination of Clifton and Leavens's work [6] (modeling of the join point stack) and Dutchyn *et al.* [10] (weaving semantics). By convention, when we introduce new user-visible syntax (*e.g.* the aspect deployment expression), we use **bold** font. Internal terms added only for the sake of the semantics are written in `typewriter` font.

Join Points The join point stack J is a list of *join point abstractions* j, which are tuples $\lceil k, l_o, l_f, p \rceil$ (Figure 5). We introduce five kinds of join points: `new` for object creation, `call` for function application/method invocation, and `exec`, `pc-exec`, `adv-exec` for function, pointcut, and advice execution, respectively. Figure 6 describes the different values for the components of join point abstractions, depending on their kind. For instance, p is always the primitive operation (used to perform the original computation);

[5] We also maintain the currently-executing object/function in the program configuration, omitted here for simplicity. The online Redex model includes the full configuration.

[6] λ_{AS} is novel in that it is tailored for λ_{JS} (including objects and mutable state), supports a more complete join point model, and deals with execution levels in a simpler manner than [28]; it is also the first formalization of (a restricted form of) scoping strategies for aspects.

$$J ::= \epsilon \mid j + J$$
$$j ::= \lceil k, l_o, l_f, p \rceil$$
$$k ::= \texttt{new} \mid \texttt{call} \mid \texttt{exec} \mid \texttt{pc-exec} \mid \texttt{adv-exec}$$
$p \in \mathscr{T}$, the set of thunks
$J \in \mathscr{J}$, the set of join point stacks

$$Expr\ e ::= \ldots \mid \texttt{jp}\,(j, \alpha) \mid \texttt{in-jp}\,(e) \mid \texttt{c/asp}\,k\,e\,e \cdots$$
$$EvalCtx\ E ::= \ldots \mid \texttt{in-jp}\,(E) \mid \texttt{c/asp}\,k\,v \cdots E\,e \cdots$$
$$v ::= \ldots \mid \boxed{J}$$

Fig. 5. Join points

k	new	call / exec / pc-exec / adv-exec
l_o	object prototype	target object
l_f	**null**	target function
p	primitive operation	

Fig. 6. Join point abstraction attributes per kind.

l_o denotes the prototype of the object being created in a `new` join point, and the target object for `call` and the three execution join points.

In order to keep track of the join point stack in the semantics we introduce two internal expression forms. `jp` (j, α) introduces a join point j whose underlying computation via proceed will be executed with aspect environment α. `in-jp` (e) keeps track of the fact that execution of e is proceeding under a dynamic join point. The definition of the evaluation context is updated accordingly (Figure 5).The expression `c/asp` (which stands for "call/aspect") is used later to treat similarly pointcut and advice execution join points. It is a function application annotated with the kind of join point k that needs to be created; this expression is generated by the weaver, discussed later on.

A join point abstraction captures the minimum context information necessary for ModAC to work (target object and function), as well as to trigger its corresponding computation when necessary (the p function). We write $\boxed{J}$ to denote the reification of the join point stack J as a λ_{AS} value. A number of introspection primitives are provided; for instance, **kind** ($\boxed{J}$) is the λ_{AS} equivalent of jp.kind in AspectScript. Similarly, **tobj** (resp. **tfun**) can be used to retrieve the (location of the) target object (resp. function).

Aspects and Deployment For the sake of conciseness and simplicity, we make the three following simplifications to λ_{JS} in this paper: *i)* scoping strategies have constant boolean components (instead of join point predicates); *ii)* only per-object deployment (**deployOn**) is described; *iii)* we do not account for context exposure (*i.e.* pointcuts simply return **true** if they match, instead of an environment). These simplifications do not affect the validity of our results: constant propagation functions are enough to state and prove the desired properties of ModAC, **deployOn** is strictly more expressive than **deploy** [29], and context exposure is an orthogonal feature for this work.

As described on Figure 7, an aspect environment α is a list of tuples (b_c, b_d, l) where l denotes the reference to the aspect, and the first two boolean values corresponds to the `c` and `d` components of the scoping strategy specified at deployment time. An aspect can be any object whose pointcut attribute is a function that takes a join point stack as input and produces either **true** or **false**. To compensate for the absence of context exposure from pointcuts, an advice function also receives as first argument the current join point stack. An advice proceeds using the **proceed** ($\boxed{J}$) primitive.

$$
\begin{array}{rl}
Expr\ e & ::= \ldots \mid \textbf{deployOn}[e,e](e,e) \\
EvalCtx\ E & ::= \ldots \mid \textbf{deployOn}[E,e](e,e) \mid \textbf{deployOn}[b,E](e,e) \mid \\
& \quad \textbf{deployOn}[b,b](E,e) \mid \textbf{deployOn}[b,b](v,E)
\end{array}
$$

$$
\begin{array}{rl}
AspectEnv\ \alpha & ::= \alpha + (b_c, b_d, l) \mid \epsilon \\
Store\ \mu & ::= \epsilon \mid \mu + (l \mapsto o^{\alpha})
\end{array}
\qquad \mathtt{asps}(l) = \alpha, \text{ where } \mu(l) = o^{\alpha}
$$

$$
\langle \mu, \alpha, J, E[\textbf{deployOn}[b_c, b_d](l_{asp}, l_{obj})] \rangle \hookrightarrow \langle \mu', \alpha, J, E[l_{obj}] \rangle \qquad \text{DEPLOYON}
$$
$$
\text{where} \quad \mu(l_{obj}) = o^{\alpha'} \text{ and } \mu' = \mu(l_{obj} \mapsto o^{\alpha' + (b_c, b_d, l_{asp})})
$$

Fig. 7. Aspects and deployment.

$$
\langle \mu, \alpha, J, E[\textbf{ref}\ \{str : v \cdots\}] \rangle \hookrightarrow \langle \mu, \alpha, J, E[\mathtt{jp}(\lceil \mathtt{new}, proto, \textbf{null}, p \rceil, \alpha)] \rangle \qquad \text{NEW}
$$
where
$$
\begin{array}{l}
proto = v_i \text{ if } str_i = \texttt{"\_\_proto\_\_"} \\
\alpha' = (\mathtt{asps}(\mathtt{cobj}()) \oplus \mathtt{asps}(\mathtt{cfun}()) \oplus \alpha)|_d \\
p = \textbf{fun}()\{\ \mathtt{new/prim}\ \{str : v \cdots\}^{\alpha'}\ \}
\end{array}
$$

$$
\langle \mu, \alpha, J, E[\textbf{fun}(x \cdots)\{e\}(l_0\ l_1\ v \cdots)] \rangle \hookrightarrow \langle \mu, \alpha, J, E[\mathtt{jp}(\lceil \mathtt{call}, l_0, l_1, p_c \rceil, \alpha')] \rangle \qquad \text{CALL}
$$
where
$$
\begin{array}{l}
\alpha' = (\mathtt{asps}(\mathtt{cobj}()) \oplus \mathtt{asps}(\mathtt{cfun}()) \oplus \alpha)|_c \\
p_e = \textbf{fun}()\{\ \mathtt{app/prim}\ \textbf{fun}(x \cdots)\{e\}\ l_0\ l_1\ v \cdots\ \} \\
p_c = \textbf{fun}()\{\ \mathtt{app/prim}\ \textbf{fun}()\{\mathtt{jp}(\lceil \mathtt{exec}, l_0, l_1, p_e \rceil, \alpha')\}\}
\end{array}
$$

$$
\langle \mu, \alpha, J, E[\mathtt{c/asp}\ k\ \textbf{fun}(x \cdots)\{e\}\ l_0\ l_1\ v \cdots] \rangle \hookrightarrow \langle \mu, \alpha, J, E[\mathtt{jp}(\lceil k, l_0, l_1, p \rceil, \alpha)] \rangle \qquad \text{C/ASP}
$$
$$
\text{where} \quad p = \textbf{fun}()\{\ \mathtt{app/prim}\ \textbf{fun}(x \cdots)\{e\}\ l_0\ l_1\ v \cdots\ \}
$$

Fig. 8. Join point creation.

An aspect is deployed with **deployOn**. Because **deployOn** embeds an aspect within an object, the stack aspect environment of the program configuration is not enough; each object needs to have its own aspect environment as well. To do so, we annotate an object o with its aspect environment α as o^{α}. By construction, as will be described below, an object is annotated with its aspect environment as soon as it is allocated in the store (with **ref**). We therefore extend the definition of the store, and introduce an internal function `asps` in order to access the aspects of an object in the store.

The DEPLOYON rule shows the semantics of per-object deployment: the aspect (at location) l_{asp} is added at the end of the aspect environment of the object (at location) l_{obj}, along with the specified scoping strategy components.

Join Point Creation & Disposal We change the NEW rule of λ_{JS} to account for the creation of `new` join points (Figure 8). The join point abstraction components are filled according to Figure 6. The primitive operation p is a thunk that returns a fresh reference to the newly-created object. Actual object creation is done using `new/prim`, an internal expression that performs creation without generating any join point. Note that the object value passed to `new/prim` is annotated with its initial aspect environment, α'. This environment is calculated as the order-preserving union ($\oplus$) of three aspect environments: the ones deployed on the currently-executing object and function (obtained with `cobj` () and `cfun` (), respectively); and the stack aspect environment. Only aspects that propagate in newly-created objects are included in α'. The notation $\alpha|_d$ refers to the aspects in α whose `d` component is true.

To account for the creation of `call` and `exec` join points, we change the λ_{JS} evaluation rule for function application/method invocation as well. The new CALL rule

$$\langle \mu, \alpha, j + J, E[\texttt{in-jp}\ (v)] \rangle \hookrightarrow \langle \mu, \alpha, J, E[v] \rangle \qquad \text{OUTJP}$$
$$\langle \mu, \alpha, j + J, E[\texttt{in-jp}\ (\mathbf{err}\ v)] \rangle \hookrightarrow \langle \mu, \alpha, J, E[\mathbf{err}\ v] \rangle \qquad \text{OUTJP-ERR}$$

Fig. 9. Join point disposal.

$$\langle \mu, \alpha, J, E[\texttt{jp}(\lceil k, l_o, l_f, p \rceil, \alpha_p)] \rangle \hookrightarrow \langle \mu, \alpha, J', E[\texttt{in-jp}(\texttt{swap}(\texttt{app/prim}\ W[\![\alpha']\!]_{\alpha_p, J'}, \epsilon))] \rangle \qquad \text{WEAVE}$$

where
$J' = \lceil k, l_o, l_f, p \rceil + J$
$\alpha_s = \epsilon$ if $k \in \{\texttt{pc-exec}, \texttt{adv-exec}\}$, α otherwise
$\alpha' = \texttt{asps}(\texttt{cobj}()) \oplus \texttt{asps}(\texttt{cfun}()) \oplus \alpha_s$

$$W[\![\epsilon]\!]_{\alpha, \lceil k, l_o, l_f, p \rceil + J} = \mathbf{fun}()\{\texttt{swap}(\texttt{app/prim}\ p, \alpha)\}$$

$$W[\![\alpha_w + (b_c, b_d, l_{asp})]\!]_{\alpha, \lceil k, l_o, l_f, p \rceil + J} =$$

```
app/prim
 fun(next){
  let(pc = (deref l_asp)["pc"])
  if(c/asp pc-exec (deref pc)["code"] l_asp pc [j_p + J]){
   let(adv = (deref l_asp)["adv"])
   fun(){ c/asp adv-exec (deref adv)["code"] l_asp adv [j_a + J] }
  }else{ next }
 }
  W[[α_w]]_{α, ⌈k, l_o, l_f, p⌉ + J, p}
```

where $j_a = \lceil k, l_o, l_f, \mathbf{fun}()\{\texttt{app/prim}\ next\} \rceil$, and $j_p = \lceil k, l_o, l_f, \mathbf{fun}()\{\mathbf{err}\ \text{"pc cannot proceed"}\} \rceil$

Fig. 10. Aspect weaving.

generates a `call` join point whose components are filled according to Figure 6. The primitive operation p_c is a thunk that generates an `exec` join point when applied. The primitive operation of this `exec` join point, p_e, performs the actual function execution by means of `app/prim`, another internal expression that does not generate join points. Note that the `jp` expressions associated to both join points specify that the stack aspect environment must change to α' when p_c or p_e are applied in order to reflect the propagation of aspects through the stack. This aspect environment is determined by taking the order-preserving union of three aspects environments: the ones deployed on the currently-executing object and function; and the stack aspect environment; and projecting the resulting environment along the `c` component (written $\alpha|_c$), which discriminates which aspects should propagate on the call stack.

Rule C/ASP accounts for the creation of `pc-exec` and `adv-exec` join points. This rule matches a function application/method invocation, but receives a first argument (k) that specifies which join point must be generated. Since invocations of pointcuts and advices are implicit, C/ASP does not generate `call` join points as rule CALL does. Join point attributes are filled according to Figure 6; the primitive operation p performs the pointcut/advice execution by means of `app/prim`, just like in the case of `exec` join points.

Once the computation underlying a join point is reduced to a value, the OUTJP rule gets rid of the join point and the `in-jp` expression (Figure 9). OUTJP-ERR does the same in the case of an error.

Weaving We now turn to the semantics of aspect weaving, specified by the WEAVE rule (Figure 10). A `jp` expression reduces to an `in-jp` expression (to signal the fact that the

$$\begin{array}{rcl} Expr\ e & ::= & \ldots \mid \texttt{app/prim}\, e\, e \cdots \mid \texttt{new/prim}\, e \\ EvalCtx\ E & ::= & \ldots \mid \texttt{app/prim}\, v \cdots\ E\, e \cdots \mid \texttt{new/prim}\, E \end{array}$$

$$\langle \mu, \alpha, J, E[\texttt{app/prim}\,\mathbf{fun}(x\cdots)\{e\}\ v\cdots]\rangle \hookrightarrow \langle \mu, \alpha, J, E[e[v\cdots/x\cdots]]\rangle \quad \textsc{AppPrim}$$

$$\langle \mu, \alpha', J, E[\texttt{new/prim}\, o^{\alpha}]\rangle \hookrightarrow \langle \mu', \alpha', J, E[l]\rangle \quad \textsc{NewPrim}$$

$$\text{where} \quad l \notin dom(\mu) \text{ and } \mu' = \mu + (l \mapsto o^{\alpha})$$

Fig. 11. Primitive function application and object allocation.

upcoming computation is associated to a join point), and the join point is pushed onto the stack (we discuss the use of `swap` and α_s later below). The list of aspects in scope α' is calculated as the order-preserving union of the aspect environments of the object and function in context, and the aspects propagated through the stack.

The weaving process is based on evaluating the function returned by the W meta-function. W recurs on α' and returns a composed procedure whose structure reflects the way advice is going to be dispatched. The base case, $W[\![\epsilon]\!]$, corresponds to the execution of the primitive operation. Otherwise, for each aspect (b_c, b_d, l_{asp}) in the environment, W first applies its pointcut to the current join point stack (which generates a `pc-exec` join point using the `c/asp` construct). If the pointcut matches, then W returns a function that applies the advice of l_{adv} (and generates an `adv-exec` join point). All this process is parameterized by the function to proceed with, $next$. In order to allow an advice to call **proceed** to trigger either the base computation or the next advice in the chain, rule WEAVE creates an auxiliary join point j_a whose p component is a thunk that applies $next$. To be complete, an auxiliary join point j_p is also created and passed to the pointcut; its p component triggers an error if **proceed** is called. Finally, If an aspect does not apply, then W simply returns $next$.

Primitive forms. The semantics of λ_{AS} use two internal primitive forms, `app/prim` and `new/prim`, described in Figure 11. `app/prim` is an application that does not trigger a join point: rule APPPRIM simply performs the classical β_v reduction. `app/prim` is used to perform the actual application of a function, as well as to hide "administrative" application, *i.e.* the initial application of the composed aspect chain, and its recursive applications. Similarly, `new/prim` allocates an object in the store and reduces to the corresponding location without producing a join point[7].

Execution levels. The weaving semantics explained previously is insufficient, because any aspect language must take precautions with infinite regression. Indeed, if we omitted the use of `swap` and α_s in Figure 10, a λ_{AS} program would never terminate. Tanter addressed this issue with execution levels [28], which ensure that pointcut and advice computation by default always happen at a higher level than base computation, avoiding infinite loops such as those due to pointcuts matching against themselves. Recall that

[7] These primitive forms are necessary for the semantics to allow actual computation to happen. The fact that they are *internal* means that it is not necessary to protect them from untrusted aspects: they cannot be used by any user code, and cannot be advised since they do not produce join points. Recall that in a higher-order aspect language, the use of execution levels is key to supporting these primitive forms as internal only [28].

$$Expr\ e ::= \ldots \mid \texttt{swap}(e, \alpha) \mid \texttt{in-swap}(e, \alpha)$$
$$EvalCtx\ E ::= \ldots \mid \texttt{in-swap}(E, \alpha)$$

$$\langle \mu, \alpha, J, E[\texttt{swap}(e, \alpha')] \rangle \hookrightarrow \langle \mu, \alpha', J, E[\texttt{in-swap}(e, \alpha)] \rangle \qquad \text{IN-SWAP}$$
$$\langle \mu, \alpha', J, E[\texttt{in-swap}(v, \alpha)] \rangle \hookrightarrow \langle \mu, \alpha, J, E[v] \rangle \qquad \text{OUT-SWAP}$$
$$\langle \mu, \alpha', J, E[\texttt{in-swap}((\textbf{err}\ v), \alpha)] \rangle \hookrightarrow \langle \mu, \alpha, J, E[\textbf{err}\ v] \rangle \qquad \text{OUT-SWAP-ERR}$$

Fig. 12. Swapping aspect environments.

in λ_{AS}, pointcuts and advices are standard functions. With execution levels, pointcuts and advices are always evaluated at the level above the expression that generates a join point. When the last advice in the chain proceeds, execution shifts back to the original level in order to run the base computation[8].

We introduce a simple modeling of execution levels, that does not require having to explicitly track the current execution level in the program configuration. Instead, we use the call stack with internal expressions so as to *swap* aspect environments and restore them when appropriate (Figure 12). Swapping per se is a very simple process: given an expression e and an aspect environment α', `swap` installs the aspect environment, and evaluates the expression (IN-SWAP). `in-swap` is used to restore the swapped aspect environment α when the expression is fully reduced (OUT-SWAP). Additionally, α_s is used to remove aspects in the stack aspect environment from scope when weaving `pc-exec` and `adv-exec` join points. This prevents the aspects deployed on the currently-executing object/function from seeing their own activity. Note that this approach does support multiple levels of execution.

Weaving (Figure 10) uses `swap` exactly where the original levels semantics [28] uses **up** and **down** shifting. The whole weaving process is wrapped by a `swap`, so that the current aspect environment is swapped with an empty environment ϵ that represents the upper level environment . This environment is used to evaluate pointcuts and advices. Of course, the fact that the stack aspect environment starts empty does not prevent aspects that have been deployed in objects and functions to take effect. If the last advice proceeds (the base case of W), aspect environments are swapped again, in order to restore the original environment to evaluate the base computation. The environment in which weaving is carried out is restored once the base computation has completed. Finally, when the whole weaving is complete, the original aspect environment is restored.

5 Properties of Modular Access Control

In this section we state two theorems corresponding to the following properties of ModAC:

Basic effectiveness. ModAC is effective in absence of untrusted aspects. This means that restriction aspects are actually deployed on untrusted objects, see illegal resource accesses, and effectively prevent them.

[8] Full-fledged execution levels include the possibility to explicitly shift execution up and down if needed, as well as to define level-capturing functions [28]. We do not include these advanced facilities in this work.

Non-inhibition. ModAC with $\mathring{\text{R}}$ is effective even in presence of untrusted aspects. This means that $\mathring{\text{R}}$ effectively prevents untrusted aspects from inhibiting protected aspects.

More precisely, we show the results for ModAC/λ_{AS}. The proofs are provided online [32]. First, we describe the three properties that define basic effectiveness.

Definition 1 (Basic effectiveness). *A ModAC implementation is said to comply with basic effectiveness if three properties are fulfilled:*

- Restrictions deployment. *Restrictions are deployed on all the corresponding objects before they can be used.*
- Restrictions scope. *A restriction aspect sees all the computation produced by the objects it is deployed on.*
- Restrictions effectiveness. *A restriction aspect always prevents the resource accesses it identifies.*

Theorem 1 (ModAC/λ_{AS} basic effectiveness). *ModAC/λ_{AS} complies with basic effectiveness.*

This theorem is a direct consequence of Lemmas 1, 2, and 3, exposed below, which address each property of basic effectiveness separately.

Lemma 1 states that any aspect (referenced by l_{depl}), in particular Deployer, deployed with scoping strategy (**false**,**true**) propagates to every new object in the store, and does so in the first position in the aspect environment of these objects. This ensures that l_{depl} sees all object creations in the application and gets a reference to these objects before any other entity. The only prerequisite is that l_{depl} is already deployed in the first position on all objects in a given point. This can be straightforwardly achieved in the bootstrapping process by exhaustively deploying Deployer on every object.

Lemma 1 (Restrictions deployment). *Let C = $\langle \mu, \cdot, \cdot, \cdot \rangle$ be a program configuration where $\forall\ (l \mapsto o^{\alpha}) \in \mu$, $\alpha = (\textbf{\textit{false}}, \textbf{\textit{true}}, l_{depl}) + \alpha'$, for some α', and $l_{depl} \in dom(\mu)$. If $C \hookrightarrow \langle \mu', \cdot, \cdot, \cdot \rangle$, then $\forall\ (l \mapsto o^{\alpha}) \in \mu'$, $\alpha = (\textbf{\textit{false}}, \textbf{\textit{true}}, l_{depl}) + \alpha''$, for some α''.*

Lemma 2 states that all aspects in the stack aspect environment deployed with c = **true**, propagate through the stack if the same level of execution is considered; *i.e.* the stack inspection algorithm is correctly implemented by means of scoping strategies.

Lemma 2 (Restrictions Scope). *Let C = $\langle \cdot, \alpha, \cdot, \cdot \rangle$ be a program configuration where $\alpha_s = \alpha|_c$. If $C \hookrightarrow^* \langle \cdot, \alpha', \cdot, \cdot \rangle$, and the sequence of reductions starts and ends at the same execution level, then $\alpha_s \subseteq \alpha'$.*

Lemma 3 states that if a restriction aspect R matches a join point j and does not proceed, then the primitive operation associated to j is not evaluated. Consequently a restriction aspect fulfills its role no matter in which position it is woven at the illegal resource access join point.

Lemma 3 (Restrictions effectiveness). *Let* $C = \langle \mu, \cdot, J, E[e] \rangle$ *be a program configuration where* $J = \lceil \cdot, \cdot, \cdot, p \rceil + J'$*, for some* J'*,* $e = \texttt{app/prim}\, W[\![\alpha]\!]_{\cdot, J}$*,* $(\cdot, \cdot, l_R) \in \alpha$*, and* l_R *is a valid aspect reference in* μ *to a restriction aspect that matches* J *and does not proceed for* J*. If* $C \hookrightarrow^* \langle \cdot, \cdot, J, E[v] \rangle$*, for some* v*, then* p *is not applied in these reductions.*

Finally, we present the non-inhibition theorem. This theorem states that if the evaluation of a pointcut whose aspect has $\mathring{R}$ deployed on it reduces to a value, this value is either **false** or (**err** $\cdot$). This holds whenever the join point stack contains a join point whose target is in the set of protected aspects. Notice that the theorem implicitly permits the existence of other untrusted aspects trying to inhibit $\mathring{R}$ itself.

Theorem 2 (Non-inhibition). *If* l_{asp} *is a valid aspect reference in* μ*,* $\mathring{R} \in \texttt{asps}(l_{asp})$*, and* $\lceil \cdot, s, \cdot, \cdot \rceil \in J$*; where* $s \in$ *the set of protected aspects, then:*
If $\langle \mu, \alpha, J, E[\texttt{jp}(\lceil \texttt{pc-exec}, l_{asp}, \cdot, \cdot \rceil, \cdot)] \rangle \hookrightarrow^* \langle \cdot, \alpha, J, E[v] \rangle$*, then* $v =$ ***false*** *or* $v =$ (***err*** $\cdot$).

6 Discussion

We discuss how to extend our results from λ_{AS} to full-fledged AspectScript, and the requirements for a general-purpose aspect language to securely support ModAC.

From λ_{AS} to AspectScript. Due to desugaring, results obtained in λ_{JS} do not immediately apply to JavaScript [16]. This is because desugaring introduces new behavior that was not present in the original code. When going from λ_{AS} to AspectScript, the theorems remain valid because they are based on the aspect-oriented features of the language, which have no relation to the desugaring process. However, access control aspects can be led to behave incorrectly if they use "exploitable" features that introduce holes upon desugaring. For example, consider a slight modification of the pointcut of the netRestriction aspect in order to allow communication with safe.cl:

```
function(jp){ return /* same as before */ && !(jp.args[0] == "safe.cl"); }
```

The equality operator == forces both operands to be of the same type [19]. For this reason, jp.args[0] is transformed to a string by an invocation of toString. The problem is that this extra method call opens the opportunity for bypassing access control:

```
var req = netService.newRequest({ t: 0, toString: function(){return {0: "safe.cl", 1: "evil.com"}[this.t++]}});
```

The toString method of the argument to newRequest returns "safe.cl" the first time it is invoked (in the pointcut of netRestriction) and "evil.com" the second time (in the body of newRequest).

In order to avoid such holes, the first possibility is to simply avoid using exploitable features in the definition of restriction aspects. For instance, it is safe to use reference equality === because it does not perform any kind of type conversion [19] (notice that all restriction aspects defined in this work follow this guideline). A less drastic solution is to permit the use of exploitable features, but to carefully examine access control aspects in order to check if their *particular usage* of the feature is safe. For example, the equality operator == is safe if both operands are of the same type! As detailed by Guha *et al.*, this checking can be automated by a specialized type system [16].

Finally, AspectScript uses a scoping strategy acs, which supports privileged execution and capturable permission contexts; acs is expressed with propagation functions, c and d. We made a simplification in λ_{AS} by supporting only constant boolean propagation values. As we said in Section 4.2, this simplification does not affect our results. In fact, supporting propagation functions only requires that $\mathring{\mathsf{R}}$ prevents inhibition of these functions; this is achieved by extending the pointcut of $\mathring{\mathsf{R}}$:

```
function(jp){return ... || cflow(function(jp){return acs.contains(jp.fun);});}
```

This way, $\mathring{\mathsf{R}}$ also inhibits untrusted aspects in the control flow of acs components. Theorem 2 and its proof must be reformulated accordingly, but this is direct.

Aspect languages for secure ModAC. This paper focuses on JavaScript to be as close as possible to our practical implementation, ZAC [33]. Still, both ModAC and the approach for securing it using $\mathring{\mathsf{R}}$ are independent of AspectScript. They can be realized in any aspect language, provided it meets certain key conditions. First of all, the language must support scoping strategies, or an equivalently expressive scoping mechanism. Per-object aspects are only necessary if one wants to provide per-object access control. Execution levels are necessary to avoid infinite loops whenever pointcut and/or advice execution join points are exposed to weaving; in order to control implicit inhibition, $\mathring{\mathsf{R}}$ relies on matching pointcut execution join points.

A crucial point in ModAC that is directly informed by the formal framework and explicitly used in Lemma 1 is related to aspect precedence: Deployer must always be the aspect with least precedence in the aspect environment to be woven at a `new` join point. This allows Deployer to deploy restriction aspects on objects before they get a chance to execute any piece of code. The semantics of λ_{AS} ensures this premise because per-object aspects are "engrained" within the object following the semantics of dynamically-deployed aspects in AspectScheme [10]. In AspectScheme this is a design decision. Here, it is not; it is a requirement. If an aspect language uses a different approach to ordering aspects, or permits to undeploy aspects, then it must provide a mechanism to guarantee the above invariant related to the presence and position of Deployer. For example, in AspectJ [20], aspects cannot be undeployed, but manual ordering is provided. Therefore, some mechanism must be added, as in AOPS [8]. On a related note, it is necessary that Deployer can deploy the restrictions on a newly-created object before any code is run on behalf of this object. In λ_{AS}, this is obtained thanks to the desugaring, which creates an empty object and then calls an initializer. In AspectJ, this can be achieved thanks to the pre-initialization join points. If an aspect language does not exhibit this specific event of an object life time, then it is not possible to guarantee that restrictions see all the computation of untrusted objects.

7 Related Work

The relation between aspects and security has a long history. We now discuss a number of related approaches. To the best of our knowledge, AOPS is the only approach that supports untrusted aspects while preventing inhibitions of access control aspects.

Modularization of access control. There are several proposals that modularize (part of) access control into aspects, particularly in Java [35,34,25,9,18,23,21]. However, these solutions implicitly assume that no other entities can affect the behavior access control aspects. This implies that access control is vulnerable to inhibition. Work on *inlined reference monitors* [11] is also related. These monitors are used to maintain access control state in the application, executing security actions whenever certain events occur. Monitors are inlined in the application code at appropriate places. Here again, it is assumed that no further code transformations can change the semantics of the security policies.

Limiting the effects of advice. A number of static reasoning approaches deal with ensuring that advice cannot have unwanted effect on the base program. A Harmless Advice [7] cannot change the value returned by a piece of code, but can produce non-termination and perform I/O tasks. This is enforced by a type and effect system. In EffectiveAdvice [22], the effects of advices are explicitly modeled using monads; the Haskell type system can then enforce that advice does not have unintented effects. Both approaches could be extended to prevent inhibition of access control aspects, but it is hard to reconcile them with the fully dynamic nature of JavaScript. Augmenting ModAC with some static reasoning is future work.

Treatment of permission contexts. Caromel and Vayssière addressed the issue of correctly handling permission contexts in the presence of metaobjects [5]. The issue is to ensure that the permission context at the base level does not affect that of the metalevel, and vice-versa. The proposed solution relies on capturing the permission context when jumping to the metalevel, and restoring it when going back to the base level. Because permission contexts are part of aspect environments in ModAC, we generalize this approach to deal with aspect environments (using `swap` and `in-swap`); also, our execution-level based approach properly deals with `proceed`.

Preventing access control inhibition. The aspect-oriented permission system (AOPS) [8] is the most related approach. AOPS relies on history-based access control (HBAC) [1], in which the decision of allowing access to a sensitive resource is taken based on *all* the entities that have participated in the execution trace. This characteristic makes HBAC a good alternative for discovering interferences produced by untrusted aspects. As discussed in Section 3.2, AOPS sacrifices two of our design goals: the aspect language semantics is customized to prevent weaving of crucial elements of the access control architecture (G1), thereby impeding even trusted aspects to apply at these points (G3).

8 Conclusion

Access control has been a recurrent target for aspect-oriented programming, mainly because of the obvious crosscutting nature of basic permission checking. However, security is a delicate concern, and therefore a correct aspectization cannot take the liberty of ignoring potentially malicious aspects. Can access control be fully modularized as an aspect? The answer is yes.

In this paper we show how ModAC, our previous work on aspectizing access control, can be made secure in presence of untrusted aspects, while maintaining its modular definition. To do so, we first prove that ModAC complies with *basic effectiveness*, *i.e.* that it actually works; a property only informally stated before. Then, we define R̊, a self-protecting restriction aspect in charge of ensuring *non-inhibition* of access control, and prove that its inclusion makes ModAC secure. We define λ_{AS}, a core calculus for AspectScript, and use it as the base for stating and proving these properties.

In order to secure ModAC without imposing overly restrictive constraints, we define three design goals: the aspect language must be oblivious to access control, untrusted aspects must not inhibit access control, and trusted aspects should be able to see any join point. Our solution fulfills all these goals. First, language obliviousness is achieved because λ_{AS} is general-purpose and therefore unaware of access control. Second, untrusted aspects are controlled by R̊, that inhibits them before they can inhibit access control. And third, trusted aspects are free to apply at any join point because R̊ is only targeted at untrusted aspects. In conclusion, we make ModAC secure thanks to R̊; and we keep it modular because R̊ is defined as a normal restriction aspect.

References

1. Abadi, M., Fournet, C.: Access control based on execution history. In Proceedings of the 10th annual Network and Distributed System Security Symposium pp. 107–121 (2003)
2. Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010). ACM Press, Rennes and Saint Malo, France (Mar 2010)
3. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: an extensible AspectJ compiler. In: Transactions on Aspect-Oriented Software Development. Lecture Notes in Computer Science, vol. 3880, pp. 293–334. Springer-Verlag (2006)
4. Box, D., Sells, C.: Essential .NET: The common language runtime, vol. 1. Addison-Wesley (Nov 2002)
5. Caromel, D., Vayssière, J.: A security framework for reflective Java applications. Software: Practice and Experience 33(9), 821–846 (2003)
6. Clifton, C., Leavens, G.T.: MiniMAO$_1$: An imperative core language for studying aspect-oriented reasoning. Science of Computer Programming 63, 312–374 (2006)
7. Dantas, D.S., Walker, D.: Harmless advice. In: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006). pp. 383–396. ACM Press, Charleston, South Carolina, USA (Jan 2006)
8. De Borger, W., De Win, B., Lagaisse, B., Joosen, W.: A permission system for secure AOP. In: AOSD 2010 [2], pp. 205–216
9. De Win, B., Joosen, W., Piessens, F.: Developing secure applications through Aspect-Oriented programming. In: Aspect-Oriented Software Development, pp. 633—650. Addison-Wesley Professional (Oct 2004)
10. Dutchyn, C., Tucker, D.B., Krishnamurthi, S.: Semantics and scoping of aspects in higher-order languages. Science of Computer Programming 63(3), 207–239 (Dec 2006)
11. Erlingsson, U., Schneider, F.: IRM enforcement of Java stack inspection. In: Proceedings of the IEEE Symposium on Security and Privacy. pp. 246–255 (2000)
12. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT Press (2009)

13. Ferraiolo, D., Kuhn, R.: Role-Based access control. 15th NIST-NCSC National Computer Security Conference pp. 554–563 (1992)
14. Fournet, C., Gordon, A.D.: Stack inspection: theory and variants. ACM Transactions on Programming Languages and Systems (TOPLAS) 25(3), 360 – 399 (2003)
15. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edition. Addison-Wesley (2005)
16. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of JavaScript. In: D'Hondt, T. (ed.) Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP 2010). pp. 126–150. No. 6183 in Lecture Notes in Computer Science, Springer-Verlag, Maribor, Slovenia (Jun 2010)
17. Hardy, N.: The confused deputy. SIGOPS Operating Systems Review 22(4), 36–38 (1988)
18. Huang, M., Wang, C., Zhang, L.: Toward a reusable and generic security aspect library. In: AOSD Technologies for Application-Level Security (2004)
19. International, E.: ECMAScript Language Specification. ECMA-262. 5th edn. (Apr 2009)
20. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An overview of AspectJ. In: Knudsen, J.L. (ed.) Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001). pp. 327–353. No. 2072 in Lecture Notes in Computer Science, Springer-Verlag, Budapest, Hungary (Jun 2001)
21. Mourad, A., Laverdire, M., Debbabi, M.: An aspect-oriented approach for the systematic security hardening of code. Computers & Security 27(3-4), 101–114 (Jun 2008)
22. Oliveira, B.C.d.S., Schrijvers, T., Cook, W.R.: EffectiveAdvice: discplined advice with explicit effects. In: AOSD 2010 [2], pp. 109–120
23. Ramachandran, R.: AspectJ for Multilevel Security. Master Thesis, Victoria University of Wellington (2006)
24. Samarati, P., di Vimercati, S.D.C.: Access control: Policies, models, and mechanisms. In: Foundations of Security Analysis and Design, Lecture Notes in Computer Science, vol. 2171, pp. 137–196. Springer Berlin / Heidelberg, London, UK (2001)
25. Słowikowski, P., Zieliński, K.: Comparison study of aspect-oriented and container managed security. In: Proceedings of the Workshop on Analysis of Aspect Oriented Software. Germany (2003)
26. Tanter, É.: Expressive scoping of dynamically-deployed aspects. In: Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008). pp. 168–179. ACM Press, Brussels, Belgium (Apr 2008)
27. Tanter, É.: Beyond static and dynamic scope. In: Proceedings of the 5th ACM Dynamic Languages Symposium (DLS 2009). pp. 3–14. ACM Press, Orlando, FL, USA (Oct 2009)
28. Tanter, É.: Execution levels for aspect-oriented programming. In: AOSD 2010 [2], pp. 37–48
29. Tanter, É., Fabry, J., Douence, R., Noyé, J., Südholt, M.: Expressive scoping of distributed aspects. In: Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development (AOSD 2009). pp. 27–38. ACM Press, Charlottesville, Virginia, USA (Mar 2009)
30. Toledo, R., Leger, P., Tanter, É.: AspectScript: Expressive aspects for the Web. In: AOSD 2010 [2], pp. 13–24
31. Toledo, R., Núñez, A., Tanter, É., Noyé, J.: Aspectizing Java Access Control. IEEE Transactions on Software Engineering (2010), in Press
32. Toledo, R., Tanter, É.: Secure and modular access control with aspects—supplementary material, http://www.dcc.uchile.cl/~rtoledo/modac-ecoop
33. Toledo, R., Tanter, É.: ZAC: Access Control in Javascript. IEEE Software (2010), in Press
34. Vanhaute, B., De Decker, B., De Win, B.: Building frameworks in AspectJ. Workshop on Advanced Separation of Concerns (ECOOP) pp. 1–6 (2001)
35. Viega, J., Bloch, J., Chandra, P.: Applying Aspect-Oriented programming to security. Cutter IT Journal 14(2), 31–39 (Feb 2001)