Access Control based on Execution History

Martín Abadi
University of California at Santa Cruz

Cédric Fournet
Microsoft Research

Abstract

Security is a major, frequent concern in extensible software systems such as Java Virtual Machines and the Common Language Runtime. These systems aim to enable simple, classic applets and also, for example, distributed applications, Web services, and programmable networks, with appropriate security expectations. Accordingly, they feature elaborate constructs and mechanisms for associating rights with code, including a technique for determining the run-time rights of a piece of code as a function of the state of the execution stack. These mechanisms prevent many security holes, but they are inherently partial and they have proved difficult to use reliably.

We motivate and describe a new model for assigning rights to code: in short, the run-time rights of a piece of code are determined by examining the attributes of any pieces of code that have run (including their origins) and any explicit requests to augment rights. This history-based model addresses security concerns while avoiding pitfalls. We analyze the model in detail; in particular, we discuss its relation to the stack-based model and to the policies and mechanisms of underlying operating systems, and we consider implementation techniques. In support of the model, we also introduce and implement high-level constructs for security, which should be incorporated in libraries or (even better) in programming languages.

1 Introduction

In the access control model of security, an access control matrix associates rights for operations on objects with subjects. The objects may for example be files and devices; the subjects may for example be users; the operations may be reading and writing. In systems that rely on access control for security (and most do), a frequent, delicate issue is the association of rights with code. For example, a piece of code may be given the rights of the subject who executes the code, those of the author of the code, or some combination of the two. These rights determine whether the code can perform sensitive operations (e.g., reading and writing files).

Runtime environments such as Java Virtual Machines (JVMs) [13, 9] and the Common Language Runtime (CLR) [5, 2, 12] provide rich support for associating rights with code, under configurable security policies. These environments aim to enable simple mobile code (classic applets) and also, for example, distributed applications, Web services, and programmable networks, with appropriate security expectations. They feature elaborate constructs and mechanisms for managing rights, including a technique for determining the run-time rights of a piece of code as a function of the state of the execution stack, and an associated requirement that programmers code certain security checks. These run-time mechanisms prevent many security holes, but they are inherently partial, and remain blind to any interaction not recorded on the current execution stack. These mechanisms also have performance and usability costs: for most programmers, their effects are difficult to predict (and even to interpret).

In this paper, we motivate and describe a new model and practical techniques for assigning rights to code at runtime. In short, the run-time rights of a piece of code are determined by examining the attributes of any pieces of code that have run (including their origins) and any explicit requests to modify rights. Our model addresses security concerns while simplifying the tasks of programmers and thereby avoiding security pitfalls. Although widely applicable, it is particularly motivated by the characteristics and needs of JVMs and of the CLR: it is largely compatible with the existing stack-based model, but it protects sensitive operations more systematically, and it also enables a smoother integration with the security mechanisms of an underlying operating system (such as security tokens in NT and its descendants). Our model can be implemented efficiently using a small amount of auxiliary state. In addition, we introduce constructs for high-level languages (such as Java or C$^\sharp$) that facilitate security-aware programming within the model.

The rest of this paper is organized as follows. In Section 2, we review some aspects of associating rights with code, and in particular stack-based access control. In Section 3, we present our history-based access control approach. In Sections 4 and 5, we give some examples and we sketch a high-level language extension in the context of C$^\sharp$. In Sections 6 and 7, we further relate our approach

To appear at the 10th Annual Network and Distributed System Security Symposium (NDSS'03), San Diego, California, 6–7 February 2003..

to stack-based techniques and system security. We close with a discussion of related work and some brief conclusions. An appendix provides some additional code.

Throughout, when we rely on the precise context of a system, we focus mainly on the CLR. (We do not assume a detailed knowledge of the existing CLR security model and of the corresponding mechanisms.) In this concrete context, we emphasize design, but also investigate implementation techniques. In particular, we have pieces of code that embody parts of our model, and we have studied matters of performance and of compatibility with existing libraries. However, we have yet to attempt a full development and integration into the CLR (or Rotor, a shared-source implementation [20]). This integration is likely to be a substantial task. In particular, this integration could include enabling optimizations currently illegal because of the stack-based model; removing data-structure customizations for that model; and compiler support for the new history-based model. We suspect that it is not too meaningful to conduct detailed performance measurements without this integration work.

2 Associating Rights with Code and Stack Inspection

In an extensible software system where subjects and pieces of code are trusted to varying degrees, it is both important and challenging to manage the permissions of running programs in order to avoid security holes.

Type safety provides a base line of protection and enables fine-grained access control. Although type safety is crucial for security in JVMs, the CLR, and related systems (such as SPIN [1]), it is not by itself sufficient. In this section, assuming type safety, we discuss some security problems that type safety does not solve, as well as a popular, stack-based technique for addressing these problems. We point out some shortcomings of this technique (of which some, but not all, are well known), thus motivating history-based rights computation.

One particular difficulty that has attracted considerable attention is the so-called "confused deputy" problem [10], which goes as follows. Suppose that a piece of untrusted code calls a piece of trusted code, such as a library function, perhaps passing some unexpected values as arguments to the call, or in an unexpected execution state. Later, the trusted code may invoke some sensitive, security-critical operations, for example operations on an underlying file system. It is crucial that these operations be invoked with the "correct" level of privilege, taking into account that the call is the result of actions of untrusted code. Moreover, this security guarantee should be achieved under the constraint that we would not expect every library function to be rewritten; only a fraction of the code may ever be security-aware.

One approach to addressing this problem is the technique called stack inspection, which is presently embodied in JVMs and in the CLR. Following this technique, an upper bound on its permissions is associated statically (that is, before execution) with each piece of code, typically by considering the origin of the piece of code. For example, whenever a piece of code is loaded from an untrusted Internet site, it may be decided that this piece will have at most the right to access temporary files, but will have no other rights during execution. At run-time, the permissions of a piece of code are the intersection of all the static permissions of the pieces of code on the stack. Thus, the run-time permissions associated with a request made by a trusted piece of code when it is called by an untrusted piece of code include only permissions granted statically to both pieces of code. An exception to this policy is made for situations in which a trusted piece of code explicitly amplifies the run-time permissions. Such amplifications are dangerous, so they should be done only after adequate checking.[1]

Although the stack inspection technique has been widely deployed, it has a number of shortcomings. One of the main ones is that it attempts to protect callees from their callers, but it ignores the fact that, symmetrically, callers may be endangered by their callees. (Similar issues arise in connection with exception handling, multiple threads, shared mutable data structures, callbacks, and higher-order programming.) If A calls B, B returns (perhaps with an unexpected result or leaving the system in an unexpected state), and then A calls C, the call to C depends on the earlier call to B, and security may depend on tracking this dependency, which stack inspection ignores. (See Section 4 for programming examples.) In theory, one could argue that A should be responsible for checking that B is "good" or that it does not do anything "bad". However, this checking is difficult and impractical, for a variety of reasons. In particular, A may be a library function, which was coded without these security concerns in mind, and which we may not like to recode—indeed, one of the appeals of stack inspection is that it avoids some security problems without the need to recode such functions. Moreover, the call to B may be a virtual call (that is, a dynamically dispatched call), whose target (B) is hard to determine until run-time.

This shortcoming of stack inspection is a source of errors with serious security ramifications. From a more fundamental perspective, stack inspection is a partial protection mechanism, which addresses only one aspect of the

[1] The details of stack inspection and of the operations that deal with permissions vary across systems. In particular, the operation that performs amplifications is coarser-grained in the JDK 1.2 than in earlier JVMs and than in the CLR. In the CLR, on which we focus, this operation can give individual permissions, and another operation can remove individual permissions.

"confused deputy" problem. Other techniques are needed in order to achieve a more complete solution, with satisfactory practical and theoretical properties.

Stack inspection is also a source of performance concerns, and these concerns can in turn contribute to errors. In a naive implementation of stack inspection, each security decision requires "walking" the execution stack and testing permissions. These operations can be expensive.[2] Therefore, programmers that think more about efficiency than about security often replace stack inspection with riskier but faster operations, such as LinkDemand in the CLR [12, page 73]. At least in principle, these performance concerns could partly be addressed through "security-passing style" implementation techniques [21].

Stack inspection presents other difficulties because of its somewhat exotic, ad hoc character. It is a unique mechanism, largely motivated by an implementation idea, separate and distinct from other security mechanisms such as may be provided by an underlying operating system, or by a distributed environment. As a result, it is hard to translate the security state of a runtime that uses stack inspection into a corresponding state that would be meaningful at the operating system level. Such a translation is often desirable when a thread in the runtime makes an external call (a local system call, or even a call across a network). In another direction, it is hard to relate stack inspection to execution models for certain high-level languages that target these runtimes. For example, programmers in functional languages such as Haskell are not encouraged to think in terms of stacks, so the runtime stacks are not an appropriate abstraction for their understanding of security. Finally, stack inspection is directly related to a particular stack-based execution strategy. Although this strategy might be reasonable in the context of an interpreter, it is not always satisfactory in the context of a compiler. Stack inspection complicates and hinders compiler optimizations that would affect the stack, such as tail-call elimination and method inlining.

In light of these difficulties and shortcomings, we should look for alternatives to stack inspection. An interesting idea is to rely on information-flow control, of the kind studied in the security literature, particularly in the context of multilevel security [4]. Unfortunately, despite recent progress (e.g., [14]), information-flow control is often too restrictive and impractical for general-purpose runtimes. Nevertheless, it provides an interesting point of comparison and theoretical background; the work of Fournet and Gordon explores the application of techniques directly based on information-flow control [8].

We propose another alternative to stack inspection: we rely on the execution history (rather than the stack, which is an imperfect record of the history) for security, as explained below.

3 History-Based Rights Computation

Next, we detail our design and mechanisms for assigning rights to code at run-time.

In short, the run-time rights of a piece of code are determined systematically by examining the attributes of the pieces of code that have run before and any explicit requests to augment rights. The pieces of code that have run include those on the stack but also those that have been called and returned. In our basic example—A calls B, B returns, then A calls C—the run-time rights in effect within C will in general depend on the fact that A, B, and C have executed. The attributes in question include in particular the origins of the pieces of code (whether they come from the local disk, digitally signed by a trusted party, from an untrusted Internet site, . . .); they may also include properties that can be determined by automated code analysis. Thus, the general idea of our approach is to remember the history of the computation (or some abstraction of this history) in computing run-time rights.

An important way to compute run-time rights is as the intersection of rights associated with each of the pieces of code that have run. Specifically, our approach is as follows:

1. *Static rights*: We associate some rights with each piece of code, statically (at compile time or load time). We refer to these rights as static rights.

2. *Current rights*: At run-time, we associate current rights with each execution unit at each point in time.

3. *Checking*: These current rights are the ones considered by default when security decisions need to be taken or when security information needs to be communicated to other system components.

4. *Storage*: These current rights are stored in such a way that programs can read them and update them (subject to the conditions given next). In particular, an ordinary variable can represent these current rights.

5. *Automatic updates*: Whenever a piece of code executes, the current rights are updated: the new current rights are the intersection of the old current rights with the static rights of this code.

6. *Explicit modifications*: At its discretion, a piece of code may invoke a special operation to modify the current rights. This operation will at most restore the static rights of this code, and it may be further constrained.

[2] Debuggers and garbage collectors also perform stack walks, for constructing traces and for finding pointers into the heap, respectively. However, their algorithms are quite different from those for stack-based security, at least in the CLR, and they are subject to different performance constraints.

7. *Syntax*: The controlled modification of rights results in some useful programming patterns. These patterns can be supported with special syntax for "granting" rights and "accepting" results after running untrusted code.

We expand on each of these points in what follows. First, however, we illustrate some of them with a trivial example, the following tiny program fragment from a trusted library:

$$m\,();\;\; File\,.Delete\,(\,s\,);$$

At run-time, the call *m*() may affect the value of the parameter *s* (for example, if *s* is an instance variable and *m* is an overridden method that sets *s*). Independently of *m*'s behavior, if the static rights of *m* do not include the right to delete files, then the set of current rights after *m*() will not include that right either, so $File\,.Delete\,(\,s\,)$ will be prevented.

3.1 Static Rights and Current Rights

Concerning point 1 (Static rights), the static rights of a piece of code typically depend on the origin and the properties of the code, as explained above. They represent the maximal rights for that code. They do not change once the code is loaded. Each piece of code can read its associated static rights (but not update them).

This point is fairly standard, and is also a prerequisite for stack inspection mechanisms. In fact, the details can be worked out so as to keep compatibility with the existing mechanisms. In particular, we can represent rights by collections of objects that implement a standard *Permission* interface (which we may informally call permissions), and we can rely on existing methods for expressing the security policies that associate pieces of code with permissions.

Concerning point 2 (Current rights), there are implementation and usability issues in choosing the size of execution units. The execution unit will typically be a thread. In that case, whenever a thread is forked, it should start with the current rights of its parent, by default. When two threads join, their current rights should be intersected. As usual, shared mutable memory should be treated with caution. Alternatively, with appropriate synchronization, the execution unit may be a collection of threads, such as an application domain in the CLR, possibly a complete process containing many related threads.

According to point 3 (Checking), the current rights are the ones used by default in security decisions, in calls on services of an underlying operating system, and in calls to execution environments on remote machines (see Section 7). In any case, there is no need to walk an execution stack in order to make security decisions.

As for point 4 (Storage), storing the current rights in a variable has several advantages:

- Security-aware programmers get flexibility and control, without the need for any additional run-time support. For instance, it is possible to code constructors that store private copies of the current rights and methods that use them later to perform security checks or modify the current rights.

- A variety of standard optimizations can be applied. The problem of optimizing programs with mutable variables is a well-understood one—and we need not be concerned about interactions between optimizations and stack-based security. For instance, tail-call elimination, which changes the stack but not the inter-method control flow, is safe in our model. Furthermore, many optimizations that change the inter-method control flow (hence the stack) can be performed, with some care. For instance, we can inline a method if we also inline the corresponding code that performs the automatic rights update before the method. (In contrast, method inlining is limited in the CLR because of potential interactions with stack inspection.) In all these respects, the current rights are just like other ordinary variables.

- From the point of view of a programmer in a high-level language, a global variable is easy to understand. Even in functional languages such as Haskell, there is the possibility of modeling mutable variables such as the one in question. (It is less standard and mundane to model a stack, although the current rights obtained by stack inspection might still be explained to the programmer as a dynamic variable with implicit bindings, or even as a local variable passed as an extra parameter for every call, as suggested by security-passing style implementations of stack inspection [21].)

The variable may be per-thread or per-process, depending on the chosen level of execution unit. We note that the CLR already includes similar mechanisms, with different information and for different purposes, so this implementation strategy appears viable and generally in tune with existing infrastructure.

The set of rights may be explicitly represented by a list. However, alternative representations are possible, such as the following:

- A symbolic expression whose value is a set of rights. The symbolic expression can be constructed using standard set union, intersection, and difference operations. It may be simplified lazily (as the rights are used) or eagerly (as the expression is constructed).

- A symbolic expression whose value is a set of code origins (such as users or network zones). Again, the symbolic expression can be constructed using standard operations, and the expression may be simplified lazily or eagerly. Moreover, the associated rights can be computed as a function of these code origins, lazily, whenever they are needed. In this case, we use the disjunction of code origins as the dual to the intersection of rights. In other words, we may, for example, keep track of the fact that all the code comes from P or Q, rather than the intersection of the rights associated with P and Q.

- A mixture of the two, where some of the components refer to code origins and others to abstract representations of permissions. For instance, if the security policy associates rights with a few "code groups", one may represent intersections of static rights as sets of groups, and still represent explicit modifications of rights by sets of permissions.

- At the other extreme, a simple bit pattern (a mask) that represents which rights are present and which are not. This representation is particularly efficient, but is applicable only in the case where there is a fairly limited and fixed set of rights.

3.2 Updating Current Rights

Point 5 (Automatic updates) says that, whenever a piece of code executes, the current rights are intersected with the static rights of this code. This update occurs automatically, independently of the code itself, so that security-unaware code is protected by default from untrusted code (see examples in Section 4). Thus, when A calls B, B returns, then A calls C, the run-time rights in effect within C will in general depend on the fact that A and B have executed. If C needs rights lost in A or B, then C may choose to restore them explicitly, as explained below, but those rights are not present by default.

Automatic updates can be efficiently implemented taking advantage of the following observations:

- An update can be skipped when the current rights are already included in the static rights of the code. This inclusion can be determined by static analysis, and taken into account in calling conventions.

- If the ways of going from one piece of code to another are method calls and returns (assuming that our "pieces of code" are at least as large as methods), then the updates to the current rights need to happen only when there are method calls and returns. This can directly be extended to exception throwing and exception handling.

- In this setting, an effective calling convention is that, whenever a call completes (either normally or with an exception), the current rights after the call are included in the current rights before the call. With this convention, the automatic updates can be enforced by calculating an intersection of rights at most once for every call (before transferring control to the callee).

We expect most updates to be (conservatively) eliminated, for two reasons: many updates (including in particular many updates in direct calls) will not actually change the current rights, and many updates are irrelevant (e.g., because the resulting current rights are never used). We may implement the remaining updates as follows:

- For every method (or for every remaining call), we may use a source-language transformation that inserts a code prefix that explicitly performs the update. This transformation can be implemented on top of a platform with no specific support for security.

- We may proceed similarly at a lower level in a just-in-time (JIT) compiler, using a native-code prefix. In addition, we may provide several entry points for the same methods, before and after the automatic update, so that the compiler can skip the update when compiling direct calls from code with the same static rights (or lower static rights).

- In the case in which there is no amplification, or few amplifications, we may actually perform all updates in the JIT compiler, as each piece of code is compiled before execution. (Later amplifications may force some recompilation.)

- We may maintain a cache for common intersections. We expect the same intersections to be computed again and again.

Concerning point 6 (Explicit modifications), the modification of rights is a sensitive operation, which should be done only with care and after adequate checking. Whereas certain reductions of rights happen automatically as described in point 5 (Automatic updates), other modifications of rights—amplifications or not—require an explicit step, which can be taken only by security-aware code. The explicit step gives us a specific point on which to focus auditing efforts, and also to place blame when things go wrong. Code that is not security-aware need not be concerned with such explicit management of rights.

The special operation that modifies rights may fail. Of course, code may acquire at most its static rights via modifications; any request to acquire more will fail. In general, configurable security policies can define the allowed modifications, much like they define static rights. A policy may say, in particular, that certain permissions can never be acquired via modifications. A policy may also say that untrusted code should not perform any modification at all, in order to simplify the writing of code that interacts with it.

From an implementation perspective, the explicit modification of rights is straightforward; it may benefit from static analysis much like the automatic updates discussed above. From a language perspective (point 7, Syntax), it can benefit from high-level syntactic support, as discussed in Section 5.

4 Examples

In this section we illustrate history-based rights computation through several examples, written in $C^\sharp$. The examples do not show explicit rights modifications. The appendix contains examples with that feature.

4.1 Basic Examples

In these examples, untrusted code attempts to use some trusted-but-naive code for deleting a file. The examples rely on *FileIOPermission* objects for representing access rights for files.

In the first example, some untrusted code (such as an applet) calls some trusted code (such as a library) that in turn performs a sensitive operation (such as deleting a file). For this example, the situation is much like with stack inspection. We mention our assumptions on static permissions in comments. (These assumptions would be enforced by the runtime security policy.)

```
 // Mostly untrusted : static permissions don't
 // contain any FileIOPermission.
 class BadApplet {
   public static void Main() {
     NaiveLibrary.CleanUp("..\\password");
   }
 }
 // Trusted: static permissions contain all permissions.
public class NaiveLibrary {
   public static void CleanUp(string s) {
     File.Delete(s);
   }
}
```

The sensitive operation can be protected in the *File* library class by requiring a permission—in our example, some *FileIOPermission*:

```
 // Trusted: static permissions contain all permissions.
public class File { ...
   public static void Delete(string s) {
     FileIOPermission p = new FileIOPermission(s ...));
     p.Demand();
     Win32.Delete(s);
   }
}
```

Here, p.*Demand*() checks that the permission to delete s is available. Our history-based mechanism keeps track of the execution of *BadApplet* and then prevents the deletion of arbitrary files: since the invocation of the delete operation occurs after the execution of untrusted code, the check fails and raises a security exception. Thus, the naive library is protected by default from untrusted callers.

The sequence of operations on the current rights goes as follow:

- As control is transferred to *BadApplet*.*Main*, the current permissions are intersected with the static permissions of *BadApplet*, thereby removing any *FileIOPermission* from the current permissions.

- As *CleanUp*, *Delete*, and *Demand* are invoked, the current permissions are intersected with their respective static permissions. Since these functions have at least the static permissions of *BadApplet*, these intersections do not actually change the current permissions (and may actually be skipped).

- Finally, p.*Demand*() checks whether the current permissions specifically contain *FileIOPermission* p and, since this is not the case, raises a security exception. Thus, p.*Demand*() prevents the deletion of the file "..\\password".

In the second example, conversely, some trusted code (such as a local application) calls untrusted code (such as a plug-in), then proceeds with the result of the call. Unlike stack inspection, our mechanism still prevents the deletion of the file.

```
 // Trusted: static permissions contain all permissions.
 class NaiveProgram {
   public static void Main() {
     string s = BadPlugIn.TempFile();
     File.Delete(s);
   }
 }
 // Mostly untrusted : static permissions don't
 // contain any FileIOPermission.
public class BadPlugIn {
   public static string TempFile() {
     return "..\\password";
   }
}
```

Operationally, the situation here is much as in the first example:

- Initially, the current permissions contain all the static permissions of *NaiveProgram*.

- When *BadPlugIn*.*TempFile* is invoked, the current permissions are intersected with the static permissions of *BadPlugIn*.

- When *BadPlugIn* returns, and later in the computation, further intersections may be performed, but the current permissions always remain included in those of *BadPlugIn*, hence they never contain any *FileIOPermission*.
- Finally, *p.Demand* raises a security exception, as above.

4.2 Further Examples

The two following examples are complete (synthetic) C$^\sharp$ programs. They illustrate two limitations of stack-based security that are addressed by history-based security. They resemble problematic programs that occur in practice, although those are typically much longer.

The examples rely on features of the CLR that may not be familiar for all readers. In particular, they rely on declarative attributes (rather than assumptions on static permissions) in order to specify the security policy for selected methods and classes—for instance, in order to lower the rights of selected applet methods. We provide these details so that the examples, when executed, actually behave as we describe in the text, but the details are otherwise unimportant.

In the first example, untrusted code creates an object of a library class (*Task*), returns it, then trusted code triggers a call to a dangerous operation (*File.Delete*(*s*)). Such patterns—and, in general, higher-order programming—are especially common with event- or delegate-based libraries, for instance those that provide graphical user interfaces.

```
public sealed class Task {
  private string s;
  public Task(string s ) { this.s = s; }
  public void Start () { File.Delete(s ); }
}
public class Untrusted {
  // The following declarative attribute removes
  // all FileIOPermissions for this method.
  [ FileIOPermissionAttribute
   ( SecurityAction .Deny, Unrestricted =true )]
  public static Task applet () {
    return new Task(”..\\password”);
  }
}
class Program {
  static void Main() {
    Untrusted.applet (). Start ();
  }
}
```

The situation is similar to the one in the previous basic example, but less direct. The program erases the file with stack inspection but triggers a security exception with our mechanism. With stack inspection, the responsibility for preventing the file deletion seems unclear. There is no way to perform an adequate test in *Program.Main*: the object that *Untrusted.applet* returns is opaque. *Program.Main* may not be aware that this object encapsulates a file name and that *Task.Start* can delete a file. Perhaps, conservatively, the *Task*(**string** *s*) constructor could immediately check the permissions that may later be requested by *File.Delete*. However, the details on these permissions (e.g., how to normalize file names, which *FileIOPermission*s are demanded to delete a file, and their relation to *s*) belong to class *File*, not to class *Task*.

The second example is more involved. It combines inheritance and exception handling. Inheritance makes it easier for an attacker to cause a library to invoke untrusted code by a virtual call to a method of a well-known trusted class.

Abstractly, throwing and handling an exception is much like calling and handling a method. However, by the time the exception handler proceeds, the stack that contained any evidence of the origin of the exception has been discarded. Therefore, with stack-based access control, one should implement any exception handler under the conservative assumption that the exception itself and its parameters are not trustworthy. This conservative assumption can complicate handling the exception.

With history-based access control, on the other hand, exceptions are like ordinary method calls. When a piece of code throws an exception, the exception handler will start running with at most the static rights of the code. If that code is untrusted, then those rights will be limited, so security checks in the exception handler may fail.

```
public class Naive {
  protected string tempFile = ”C:\\temp\\myFile”;
  virtual protected void proceed () { ... }

  public void m() {
    try {
      proceed();
    }
    catch (SystemException e) {
      File.Delete(tempFile);
      Console.WriteLine(”Deleted␣{0}.”, tempFile);
    }
  }
}

public class PlugIn : Naive {
  [ FileIOPermissionAttribute
   ( SecurityAction .Deny, Unrestricted =true )]
  override protected void proceed() {
    try {
      tempFile = ”..\\password”;
      File.Delete(tempFile);
    }
```

```
      catch ( SecurityException e) {
        Console.WriteLine("The␣ first ␣attempt␣ failed ." );
        throw new SystemException("Out␣of␣memory.");
      }
    }
  }
  class Top {
    static void Main() { new PlugIn().m(); }
  }
```

Here, the untrusted class *PlugIn* is a subclass of the trusted but naive class *Naive*, and overrides one of its methods, *proceed*. The call *PlugIn* (). *m*() triggers a call to *proceed*, which will in turn first cause a security exception with an attempt to delete a file, then will throw a system exception. The exception handler in *Naive* finally attempts to delete a file—but not a temporary file as was presumably intended in *Naive*. With history-based access control, the fact that the new code for *proceed* is untrusted is automatically considered in deciding whether to delete the file.

4.3 A History-Based Policy

In the security literature, some policies use a history of past sensitive operations as an input to later access-control decisions. For example, with Chinese Wall policies, access to data is not constrained by attributes of the data in question but by what data the subject has already chosen to access [3]. See Section 8 for further references and discussion.

While our model does not embody those policies, it can sometimes help in supporting them. As an example, we show how it can help in building a simple Chinese Wall.

In the example, a program initially has access to code from two companies, A and B, but it can actually use code from at most one of the companies. The code may include proprietary data and procedures from the two companies, and might send information back to A and B, respectively.

First, we create a specific class of permission with a constant string parameter whose presence indicates that code from a given origin is still allowed to run.

```
// Specific permission for compartments.
// Most method implementations are omitted.
public class CompartmentAccess : Permissions {
  // permission to access a specific compartment
  string id;
  public CompartmentAccess(string id ) { this.id = id; }
  // permission to access all compartments
  public CompartmentAccess () { ... }
}
```

The code received from company A need not be modified, or even inspected beyond a normal type-safety verification. However, a security policy should be attached to that code, for instance using attributes:

```
// Static rights of type CompartmentAccess contain
// at most CompartmentAccess("A"); demanded rights
// for all code include CompartmentAccess("A");
// in CLR parlance:
[assembly:CompartmentAccessAttribute
 ( SecurityAction .RequestOptional, id="A")]
[assembly:CompartmentAccessAttribute
 ( SecurityAction .Demand, id="A")]

// Code from company A, unchanged.
class libraryA { ... }
public class A : Contractor { ... }
```

The code received from company B is handled similarly.

Now, automatically, a program that initially has access to code from the two companies can actually use code from at most one of them. In other words, our Chinese Wall policy is enforced by the underlying history-based machinery, without any extra state and extra bookkeeping at compartment boundaries. For example, one may write a program that selects an offer:

```
public class CompliantCustomer {
  int examine() {
    ...
    if ( should_consider_A ) {
      Contractor a = new A(query);
       // No B code will ever run past this point.
      return A. offer ();
    }
  }
  public void main() {
    int offer = examine();
    Contractor b = new B();
    // Raises a security exception if any A code
    // has run.
    ...
  }
}
```

The security policy may further specify that the permissions to access the two compartments should not be restorable via explicit modifications. Thus, the program would raise a security exception even if it tries to restore the right to use B's code after calling A's code.

Going beyond the separation between A and B, one may enforce policies that constrain access to code in the compartments, that is, to contractor code. For instance, access to that code may be allowed only up to a certain program stage, and certain sensitive operations might even require that contractor code has never run:

```
public void PrivateStuff () {
  // First exclude further contractor code:
  new CompartmentAccess().Deny (); ...
}
public void SensitiveStuff () {
  // First check that no contractor code has ever run:
  new CompartmentAccess().Demand(); ...
}
```

5 High-Level Programming Constructs (in C$^\sharp$)

Even if they are not strictly part of the security model, high-level language constructs can help programmers understand and live in harmony with rights management. Consider, as an analogy, the related situation in exception handling. In principle, it would be possible to provide access to exception handling using a special library for registering callbacks to be triggered when an exception occurs. Nonetheless, using scoped constructs such as **try** $\{\}$ **catch** () $\{\}$ **finally** $\{\}$ helps. We believe that, similarly, language constructs are helpful in dealing with security.

We identify two common programming patterns for the controlled modification of rights in security-aware code. These patterns, named "Grant" and "Accept", consist of the following operations:

Grant: When running after less trusted code (e.g., when called by that code):

- check that the execution state is ok,
- amplify rights (to a specific subset of the code's static rights),
- perform sensitive operations, and
- reduce rights (at least to their initial state).

This pattern is analogous to certain explicit amplifications mechanisms used with stack inspection (*DoPrivileged* in Java, *Assert* in the CLR) with similar effects but different implementations.

Accept: When running less trusted code (e.g., when calling less trusted methods):

- save parts of the execution state,
- perform untrusted operations,
- check that the execution state is ok, and
- amplify rights (at most to their initial state).

This pattern does not explicitly reduce rights before the untrusted operations: since these operations have limited static rights, this reduction occurs automatically. (In Section 6, we show that, with some care, Accepts can be used to implement the same behavior as stack inspection.)

Both patterns comply with the efficient calling convention outlined in Section 3.2: their final current rights are always included in their initial current rights. In both cases, the operation that checks whether the execution state is ok depends on the security policy, and typically involves validating some of the values passed as parameters and checking the presence of some current rights.

These patterns would greatly benefit from direct syntactic support in programming languages (as is already the case with stack-based rights computations). Next, we describe corresponding high-level programming-language constructs, in the context of C$^\sharp$, and sketch their implementation in terms of lower-level operations on current rights. A more detailed implementation in C$^\sharp$ and examples are given in the appendix.

We extend the grammar of statements with two constructs, $\mathit{Grant}(P)\ \{B\}$ and $\mathit{Accept}(P)\ \{B\}$, where P is a subset of the static permissions to be amplified and $\{B\}$ is a block of code containing the operations to be performed (and which are the scope of the constructs). Optionally, P may be omitted, its default value being all static permissions. These statements are executed as follows:

***Grant*(*P*){*B*}:** Before running B, the initial value of the current permissions is saved and the selected permissions P are added to the current permissions. When B completes (possibly with an exception), the current permissions are assigned the intersection of their initial and final values. Note that *Grant* does not leave extra rights after completion of the block B, and preserves the loss of any right while running B.

***Accept*(*P*) {*B*}:** Before running 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 uncaught exception, then the current permissions are not modified.)

The code that executes *Accept* takes responsibility for the effect of the operations performed by B on the rest of the program, and should therefore perform sufficient checks within B before its normal completion. Note that *Accept* does not provide any extra right before the completion of B.

6 Further Comparison with Stack Inspection

As Sections 3 and 4 explain, history-based access control has safety and simplicity benefits, and it is also attractive from a performance perspective (in particular, because it enables compiler optimizations). We now revisit the relation between history-based and stack-based access control, further relating their effects.

Technically, the key difference between history-based and stack-based access control occurs when a method terminates (normally or with an exception). In our model, with the calling convention given in Section 3.2, the current rights when a method terminates are lower than or equal to their value before the method call. In contrast, with stack inspection, the current rights are restored to their value before the method call, possibly augmenting them. Therefore, given the same static rights and assuming our calling convention, history-based current rights are always included in stack-based current rights. This property can be quite helpful in transitioning from stack-based to history-based access control: legacy code may generate additional security exceptions, but should remain at least as secure.

History-based and stack-based access control can also be compared by studying encodings of each in terms of the other.

- To recover a stack-inspection semantics on top of history-based mechanisms, we can add an "Accept" around every call that may lower the current rights (typically, around every call to potentially less trusted code). For example, we may write:

  ```
  Accept( current ) { applet.run (); }; SQL.run();
  ```

  If the call to *applet.run*() entails the loss of some permissions because *applet.run*() is untrusted code with few static rights, then the "Accept" may restore those permissions. Those permissions may be needed for executing *SQL.run*().

  We believe that there are relatively few such calls in existing libraries and, more importantly, that the presence of an "Accept" or not for such calls should not make any difference in most legacy applications: at this stage, there are not many partially trusted libraries, and even the libraries designed to be callable from partially trusted code do not often rely (or should not rely) on partially trusted code for their implementation.

- Conversely, to implement history-based rights on top of stack inspection mechanisms, for any given call, we can (in theory) apply a "continuation-passing-style" transform that passes an extra function parameter to be called with the result, upon completion of the callee. Hence, the callee still appears on the stack while its result is used by the caller's continuation. However, this encoding is not practical, except maybe for a few sensitive interfaces.

As an example of the latter encoding, consider the following method, which takes a delegate parameter:[3]

```
 // Define the class StringCode of " code pointers " to
 // methods that take no argument and return a string .
public delegate string StringCode();

public class Plain {
  public static void Foo(StringCode badCode)
  {
    string s = badCode();
    // When checking permissions,
    // badCode is not on the stack anymore.
    File.Delete(s);
  }
}
```

With stack-inspection, this dangerous method enables any untrusted code *badCode* passed as a parameter to pick the file to be deleted. To implement the more secure behavior of history-based permissions, on top of stack-based permissions, for this method only, one may use instead:

```
 // Define two auxiliary delegate classes :
public delegate void StringCont(string s);
public delegate void StringApplet (StringCont c);

public class Encoded {
   // Same function , with a different API
  public static void Foo(StringApplet badCode)
  {
    // Use a built−in constructor of class StringCont
    // to create a delegate to method Callback:
    StringCont sc = new StringCont(Callback);

    // Pass this delegate to badCode:
    badCode(sc);
  }

  private static void Callback(string s)
  {
    // When checking permissions,
    // badCode is still on the stack .
    File.Delete(s);
  }
}
```

7 History-Based Rights and System Security

When a piece of code is verified and managed by a language runtime, its sensitive operations (hence its rights) still affect a broader, layered security infrastructure which may include, for instance, a local-host operating system and some distributed components. Each layer provides its own security models and mechanisms, with sometimes unfortunate overlaps and discrepancies.

For instance, rights management in the CLR is generally finer and more expressive than in NT, since the CLR can rely on type safety rather than memory isolation, but there is also a deep mismatch between permissions in the CLR and access control in NT. Even when they manipulate the same abstract rights, such as file access rights, the two layers use distinct models and interfaces, with no obvious mapping between the two. Pragmatically, the CLR

[3] In C♯, an instance of a delegate class encapsulates an object and a method on that object with a particular signature. So a delegate is more than a C-style function pointer, but slightly less than a closure. When it encounters a delegate declaration, the compiler provides an implementation for the class of delegates with the given method signature.

expects to run as a highly-trusted application and, after performing its own security checks, calls the system with all its privileges (see also [21, page 7]). Hence, erroneous amplifications of rights within the CLR can typically be exploited despite NT security.

Our model has a more direct counterpart in terms of system security. In the context of NT [19, page 506], the "restricted token" mechanism enables us to construct a disjunction of a user id with some set of special security identifiers (SIDs) to represent limitations of rights; this disjunction corresponds to the intersection of current rights described in point 5 of Section 3 (Automatic updates). Specifically, we can interpret restricted tokens as the result of a coarse-grained history-based computation:

- The system represents rights symbolically, using sets of SIDs. From these SIDs, an access-control policy computes the access rights for specific objects, on demand. For every process, the system represents the current rights as a security token, consisting of one or more such sets of SIDs. The process is granted a right when each set of SIDs independently grants the right. At any point, one can update the current rights by adding another set of SIDs to the security token, which further restricts the process.

- There is no counterpart for the explicit amplification of rights. Restricted tokens correspond to permissions that cannot be acquired via explicit modifications (see Section 3.2). Thus, restricted tokens provide strong guarantees (a process will never obtain a right once the right is denied) at the cost of expressiveness (sensitive operations must instead be requested using inter-process communication).

Beyond the reuse of well-known security concepts, a mapping from current rights to security tokens has concrete advantages. Crucially, system calls can be performed with the appropriate privileges. Also, SIDs can be shared across machines within the same domain. In contrast, at present, code-based security policies and permissions in the CLR are relative to the local host machine (for example, dealing with permissions for all files on the local drive *C*:\).

When handling calls on services of an underlying operating system (or calls to execution environments on remote machines), it is particularly attractive to avoid complicated translations of rights, as those translations can be expensive and inaccurate. Such translations are easy to avoid if the current rights are exactly those of one particular user of the underlying operating system. In that special case, the user id can be employed as the representation for those rights. We can generalize from this special case by keeping track of rights partly in terms of code origins, as discussed in Section 3.

8 Related Work

The security literature contains much related work. Some of it is mentioned above, for example the use of information-flow control (e.g., [4, 14]).

History-based access control may be viewed as a pragmatic approximation to information-flow control that keeps track of code execution but not data dependencies. Going further back, there is related work in the classic literature on operating systems, expressed in terms of protection rings [18, 16]. These rings might be seen as a very simple, fixed hierarchy of sets of static rights, with an automatic update mechanism for current rights, and with hardware support. In the remainder of this section we focus on recent related work, particularly on work about stack inspection.

Stack-based mechanisms for access control are widely documented for JVMs [9] and the CLR [5, 12]. In the research literature, many works treat the analysis and optimization of permissions (see for instance [11, 15]). Others deal with interesting, non-trivial implementations of stack inspection, for instance with inlined reference monitors [7] or in security-passing style [21]. These implementations suggest that an eager computation of current rights can be made as efficient as a lazy computation by stack inspection. (The operations performed by these implementations to simulate a stack-based semantics are similar but generally more complex than the operations for computing history-based rights described in Section 3.2.)

At a more semantic level, Wallach et al. explicate stack inspection in terms of a logic of access control (ABLP logic) [21]. They model security contexts and decisions in terms of logical statements. The powerful idea of relating stack inspection to logic should also apply, mutatis mutandi, to our history-based technique. We discuss this point only in order to highlight differences with stack inspection, so we avoid formal details. Basically, Wallach et al. associate a set of logical statements E_F with each stack frame F. They map operations to logical statements: $Ok(T)$ means that it is ok to perform operation T. The access control problem consists in deciding whether a frame can perform an operation T, and is reduced to deciding whether E_F logically implies $Ok(T)$. When a frame F calls a frame G, for each of F's statements s, one adds F $says$ s to G's statements. Significantly, there is no corresponding modification when G returns. In contrast, with history-based rights, the current rights are affected whenever there is any transfer of control—whether the transfer corresponds to a method call or return, and also for example if it results from exception handling.

Fournet and Gordon also consider an abstract model of stack inspection mechanisms [8], based on that of Pottier et al. [15]. In a simple functional setting (a lambda calculus), they discuss limitations of stack inspection. Using formal operational semantics, they also explore several alternatives to stack inspection with stronger properties by refining the reduction rule that discards a security frame after an evaluation. The present paper can roughly be seen as an elaboration of one of these alternatives, focused on control transfers (rather than more general flows of information), and targeted at a full-fledged runtime system (quite different from the lambda calculus).

Execution history also plays a role in Schneider's security automata [17] and in the Deeds system of Edjlali et al. [6]. However, those works focus on collecting a selective history of sensitive access requests and use this information to constrain further access requests: for instance, network access may be explicitly forbidden after reading certain files. In contrast, our approach considers the history of control transfers, rather than a history of sensitive requests.

9 Conclusions

From a functional perspective, history-based rights computation is largely compatible with existing security machinery and libraries, although it requires runtime modifications and suggests optimizations and language extensions. From a security perspective, we believe that the benefits of access control based on execution history are substantial. It provides a simpler alternative to stack inspection, and supports a safer, wiser posture with respect to security checks.

Acknowledgments We are grateful to Praerit Garg, Andy Gordon, Tony Hoare, Brian LaMacchia, Butler Lampson, Paul Leach, and Erik Meijer for discussions on the subject of this paper, to Mike Burrows for help with the title, and to Dan Wallach and anonymous reviewers for help with the presentation. Most of Martín Abadi's work was done at Microsoft Research, Silicon Valley, with Microsoft's support. Martín Abadi's work was also partly supported by the National Science Foundation under Grants CCR-0204162 and CCR-0208800.

References

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. J. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 267–284, 1995.

[2] D. Box. *Essential .NET Volume I: The Common Language Runtime*. Addison Wesley, 2002. To appear.

[3] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[4] D. Denning. *Cryptography and Data Security*. Addison Wesley, 1982.

[5] ECMA. Standard ECMA-335: Common Language Infrastructure, Dec. 2001. Available from `http://msdn.microsoft.com/net/ecma/`.

[6] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *ACM Conference on Computer and Communications Security*, pages 38–48, 1998.

[7] Ú. Erlingsson and F. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246–255. IEEE Computer Society Press, 2000.

[8] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. In *29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 307–318, Jan. 2002.

[9] L. Gong. *Inside Java*$^{\mathrm{TM}}$ *2 Platform Security*. Addison Wesley, 1999.

[10] N. Hardy. The confused deputy. *ACM Operating Systems Review*, 22(4):36–38, Oct. 1988. Available from `http://www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html`.

[11] T. Jensen, D. L. Metayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 89–103. IEEE Computer Society Press, 1999.

[12] S. Lange, B. LaMacchia, M. Lyons, R. Martin, B. Pratt, and G. Singleton. *.NET Framework Security*. Addison Wesley, 2002.

[13] T. Lindholm and F. Yellin. *The Java*$^{\mathrm{TM}}$ *Virtual Machine Specification*. Addison Wesley, 1997.

[14] A. C. Myers. JFlow: Practical, mostly-static information flow control. In *26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 228–241, 1999.

[15] F. Pottier, C. Skalka, and S. Smith. A systematic approach to access control. In *Programming Languages and Systems (ESOP 2001)*, volume 2028 of *LNCS*, pages 30–45. Springer, 2001.

[16] J. H. Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7), July 1974.

[17] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1): 30–50, Feb. 2000.

[18] M. D. Schroeder and J. H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3):157– 170, 1972.

[19] D. A. Solomon and M. E. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, third edition, 2000.

[20] D. Stutz. The Microsoft shared source CLI implementation. Mar. 2002. Available from `http://msdn.microsoft.com/library/en-us/Dndotnet/html/mssharsourcecli.asp`.

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

```
public class Permissions
{
   // Static permissions attributed to the immediate caller :
  public static Permissions Static ;

   // Dynamic permissions at this stage
  private static Permissions now;
   // Automatically updated whenever some code runs,
   // with an implicit : now = now. Intersect ( Static ).

   // Dynamic permissions can be read and updated:
  public static Permissions Current {
    get { return now; }
    set {
      if ( value .IsSubsetOf ( Static )) now = value;
      else throw new SecurityException (" Amplification␣not␣permitted ." );
    }
  }

   // Imperative actions on permissions (same interface as in the CLR):
  public void Demand() {
    if ( this .IsSubsetOf (Current )) return;
    else throw new SecurityException ("Operation␣not␣permitted ." );
  }
  public void Assert ()       { Current = now.Union(this ); }
  public void Deny()          { now = now. SetDifference ( this ); }
  public void PermitOnly () { now = now. Intersect ( this ); }

   // Data methods (same methods as in the CLR):
  public Permissions Union(Permissions p) {}
  public Permissions SetDifference (Permissions p) {}
  public Permissions Intersect (Permissions p) {}
  public bool IsSubsetOf (Permissions p) {}
  }
}
```

Figure 1. The class *Permissions*.

Appendix:
A Partial Implementation in C$^\sharp$

In this appendix, we provide a modified interface to permissions and its partial implementation, in the context of C$^\sharp$ and the CLR. Two essential aspects of an implementation are omitted here: the automatic update mechanism for the current rights (represented as the public property *Permissions .Current*), and an access mechanism to the static rights associated with a given piece of code (represented as local variables *Permissions . Static*).

First, in Figure 1, we define a class *Permissions*, used below, that provides the base interface to history-based permissions—specific permission classes would be represented as subclasses of *Permissions*. Whenever possible, we use the same method names as in the existing (stack-based) system class *CodeAccessSecurityPermission* in the CLR.

Next, we illustrate in some detail our two amplification patterns, “Grant” and “Accept” (Figures 2 and 3). For each pattern, we first rely on the high-level syntax defined in Section 5, then we implement the same behavior in terms of lower-level operations on the current rights (defined in the class *Permissions* in Figure 1).

```
public class GrantExample
{
  // Static permissions attributed to this class by the security policy:
  static Permissions Static;

  // Handpicked sets of permissions (application-specific):
  static Permissions prior; // rights of a minimally-trusted valid caller.
  static Permissions extra; // extra rights of a privileged callee.

  // GRANT is a controlled form of privilege elevation
  // that temporarily gives extra permissions to a specific block.
  public void LibraryGate() {
    // usually checks preconditions
    // such as the presence of some permissions:
    prior.Demand();

    // elevates permissions for this block of code:
    Grant (extra) {
      /* run sensitive code requesting elevated privileges */
    }
    /* continue with ordinary code */
  }

  // idem, using lower-level operations on Permissions.Current.
  public void ImplementLibraryGate () {
    prior.Demand();
    Permissions before = Permissions.Current;
    try {
      extra.Assert (); // privilege elevation
      /* run sensitive code requesting elevated privileges */
    }
    finally {
      before.PermitOnly(); // cancels privilege elevation
    }
    /* continue with ordinary code */
  }
}
```

Figure 2. “Grant”.

```
delegate int IntCode(); // some basic interface to untrusted code

public class AcceptExample
{
  // Static permissions attributed to this class by the security policy
  static Permissions Static;

  // Handpicked sets of permissions (application−specific):
  static Permissions saved; // rights restored after untrusted calls

  // ACCEPT is a controlled form of privilege elevation
  // that restores some or all permissions possibly lost in
  // a specific block (for the benefit of any following code).
  private static int LibraryProxy(IntCode badCode) {
    int i;
    Accept (saved) {
      // Runs code that may interact with less trusted code;
      i = badCode();
      // usually checks post−conditions;
      // by design, unhandled exceptions won't restore permissions.
      if (i<0) throw new InvalidOperationException(”bad␣integer”);
    }
    return i;
    // From the caller's viewpoint, the resulting permissions are
    // the same as if this method had produced i itself.
  }

  // idem, using lower−level operations on Permissions.Current:
  private static int ImplementLibraryProxy(IntCode badCode) {
    int i;
    Permissions before = Permissions.Current;
    i = badCode();
    if (i<0) throw new InvalidOperationException(”bad␣integer”);
    before.Intersect(saved).Assert();
    return i;
  }
}
```

Figure 3. “Accept”.