Vrije Universiteit Brussel - Belgium

Faculty of Sciences

In collaboration with Ecole des Mines de Nantes - France

1999

Towards a security aspect for Java

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Sciences in Computer Science
(Thesis research conducted in the EMOOSE exchange
project funded by the European Community)
Master in Computer Science

By Andrés Farías

Promotor: Prof. Theo D'Hont (Vrije Universiteit Brussel)

Co-Promotor: Mario Südholt (Ecole des Mines de Nantes)

Sunday, February 04, 2001

13:02

Table of contents

1 Introduction

The Java™ language is an object oriented language that has become widespread in commercial use. It allows Java-compatible Web browsers to download code dynamically and then to execute that code locally. However, users must worry about executing any code that comes from untrusted sources or that passes through an insecure network. Programs that come from remote sites are called *applets* and they are used to add services and features to web pages. Applets allow normal programmers to create very interesting decorators and graphics. Nevertheless applets can attack the system in several ways. This is why security has become a very important issue: if Internet seems be insecure, then people hesitate to use it as a commercial trustable environment.

Security consists in providing mechanisms to protect a system. Most of these mechanisms consist in the separation of system's functionalities and the control to access them. In this thesis, security will be considered as the way to ensure that external programs (like applets) do not access (certain) security-sensitive resources without passing appropriate runtime checks.

Java provides a *security manager* that allows programmers to define a security policy for the system in such a way that the functional code (also called *base code*) and the security policy are largely separated. Therefore, the functional code is not affected and can be written quite independent from the security specification.

Until now, many bugs have been found in the Java implementation and applet attacks that bypass the security protection of Java, endangering the vital information and user privacy. This facts oblige the programmer to go further than the security provided by Java and force them to merge the base code with security protection that make code lose some desirable properties such as reusability, clarity and understandability. Then a new issue arises: Is it possible to write program secure and understandable, in such a way that it will be easy to reuse?

Separation of concerns is a paradigm that study how to separate concerns from each others, and from the source code in order to make it more understandable. Programming techniques to separate concerns have recently lead to *Aspect Oriented Programming* (AOP) that consist in writing the code and the different concerns (called *aspects* as well) separately and then merge them using a tool called "*weaver*" to generate the final code.

The aim of this thesis is to study security and security models to make a security aspect for Java based on the AOP technique. This security aspect is the integration of type-based and policy-based security strategies. Using this aspect, programmers should be able to write a secure specification in a very expressive language, and implement this language using program transformations.

As a result of using this security aspect, some of the bugs found in the Java implementation have no impact. Moreover, the protection against applets that attack the system using only the power given by the Java language is enforced.

This thesis is divided in two main parts. In a first part are described the separation of concern paradigm, the aspect oriented programming technique and the Java security model. The second part of the thesis present the contributions that are a definition of the aspect language, the program transformations and an overview of the implementation.

2 Aspect oriented programming

2.1 Separation of concerns

Today's software applications have to deal with concerns like *concurrency, distribution, real-time constrains, debugging and security*. Unfortunately, when programmers deal with one or more of these concerns in the same program, they see themselves involved in a complex code. Then, this code becomes hard to understand, write, modify and maintain, and less reusable because the functional code is merged with concerns and then the identity of the code lose generality.

This problem arises because the code associated with the concern is scattered throughout the source code of the different program components. For example, *figure 1* shows a peace of the Java 1.2.2 implementation source code [Sun99b] where it is scattered by the aspect (marked in bold).

```
public Win32FileSystem() {
  slash = ((String) AccessController.doPrivileged(
              new GetPropertyAction("file.separator"))).charAt(0);
  semicolon = ((String) AccessController.doPrivileged(
              new GetPropertyAction("path.separator"))).charAt(0);
  altSlash = (this.slash == '\\') ? '/' : '\\';
    }
}

 handle = create(cmdstr, envstr, stdin_fd, stdout_fd, stderr_fd);
 java.security.AccessController.doPrivileged(
             new java.security.PrivilegedAction() {
     public Object run() {
    stdin_stream =
        new BufferedOutputStream(new FileOutputStream(stdin_fd));
    return null;
     }
 });
```

Figure 1: Security runtime checks code scattered along the code

Normal abstraction mechanisms provided by languages such as procedures, classes and objects, do not package the concern into a single unit of encapsulation. The paradigm called "Separation of Concerns" proposes to separate the concerns of the program in order to make it easy to reuse, write, understand and modify.

There are several techniques to accomplish the separation of concerns, and each technique can be applied to more than one concern. Some techniques address specific concerns because they permit to make the separation in a more natural way. In [Hür95] different techniques are identified with the concerns which are more appropriated to separate with[1]. Some examples of them are *metalevel programming* [Str96], *pattern-oriented programming* [Lor98], *composition Filters* [Aks98], etc. This thesis focuses on Aspect Oriented Programming.

2.2 The aspect oriented programming technique

Until now we have been talking about different concerns that are scattered throughout the functional code (called ***base code*** in the AOP terminology) of the programs. An ***Aspect*** will be defined as *a feature or concern that crosscuts the different components of our program (as it was showed in the previous figure) and is responsible for code tangling* [Kic97].

[1] This classification was made considering the state of the art in AOP until this time. Each technique can surely offer new solutions to separate different concerns of those named here.

Aspect Oriented Programming (AOP) is a technique that enables programs involving such aspects to be expressed clearly, including appropriate isolation and composition.

The principal goal of AOP is to precisely separate the aspect code from the base functionality code by means of aspect languages, which offer the ability to express the different aspects separately. Once the user has written both the base function code and the aspect specification, a tool called "*weaver*" merge them. *Figure 2* shows the AOP scheme:

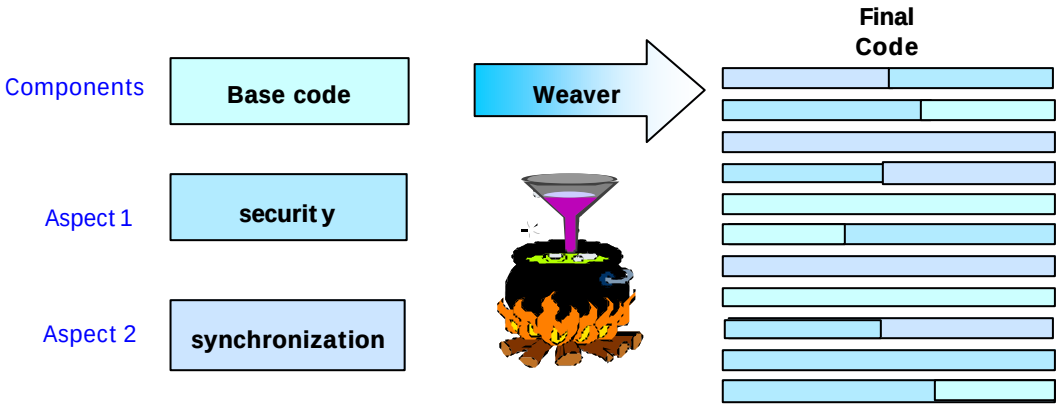


Figure 2: AOP scheme

The weaver is the most important component in the system. This tool takes the different aspects plus the base code and generates a new program that has all the components merged. The weaver taking elements from the aspect definition and transforming the base code in function of these elements. These elements that relate the aspect and the base code are called *join points* [Kic97]. The final code produced by the weaver is clearly more complex than the original code.

2.3 Characteristics of aspect oriented programming

As it was said before, AOP is a new technique that is still under development. The application of AOP to object-oriented languages raises some issues relevant for this thesis.

✍ ***Conciseness***

When the aspect language is expressive enough, many lines of code could be resumed in a fewer quantity. Moreover, specification of the aspect can be written only one time, representing many appearances in the base code [Fra98].

✍ ***Understandability of the aspect***

Using the aspect language, the programmer deals with a specific domain language and should understand it better. Furthermore, the programmer may easier acquire programming skills in this specific domain due that the aspect language express the concern very concisely and clear. Finally, the programmer deals now only with the source code or one aspect language at the same time.

✍ ***Efficiency***

In general, the weaver inserts and changes all the code necessary to satisfy the requirements specified in the aspect language. Then, the efficiency relative with the code that does not use the aspect language should depend of the implementation of the weaver.

In the application of AOP to security these characteristics are important. A discussion about the applicability of those characteristics to the security aspect defined can be found at the end of chapters 5 and 6.

3 The Java security model

3.1 Security

Since computational systems have been built to be used by multiple users sharing resources, applications and data, it has become necessary to implement mechanisms to ensure that each user has only access to the resources, applications and data that has been assigned for him.

Security consists in providing mechanisms to protect a system. Most of these mechanisms consist in the separation of system's functionalities and the control to access them.

The particularity of each system makes necessary to define different security mechanisms to protect the system, and then, many protocols of security have been created until today. In an effort to establish common profiles for secure systems, the U.S. Department of Defense has published the "Orange book" in 1985 [Nat85].

We will define a security mechanism as a piece of software that provides any combination of the following functionalities:

✍ ***authorization***

Authorization is the process of giving someone permission to do or have something. In multi-user computer systems, a system administrator defines for the system which users are allowed access to the system and what privileges of use (such as access to which file directories, hours of access, amount of allocated storage space, and so on).

✍ ***authentication***

The process of identifying an individual usually based on a username and password. In security systems, authentication is distinct from authorization, which is the process of giving individuals access to system objects based on their identity. Authentication merely ensures that the individual is who he or she claims to be, but says nothing about the access rights of the individual.

✍ ***privacy***

Privacy is a mechanism by which each user can protect its personal information and will not be shared with anyone else without his permission.

✍ ***integrity***

Integrity refers to the validity of the data within such a system, and covers the topics associated with guaranteeing that data are correctly updated and maintained. Integrity can also be understood as the reasonable assurance that data is not changed while en route from a sender to its intended recipient

✍ ***non-repudiation***

The reasonable assurance that a principal cannot deny being the originator of a message after sending it. Non-repudiation is achieved by encrypting the message digest using a principal's private key. The public key of the principal must be certified by a trusted certification authority.

✍ ***delegation***

The ability to empower a principal to act on behalf of another principal.

✍ ***Cryptography***

The art of protecting information by transforming it (encrypting it) into an unreadable format called cyphertext. Only those who possess a secret key can decipher (or decrypt) the message into plaintext. Encrypted messages can sometimes be broken by cryptanalysis, also called codebreaking, although modern cryptography techniques are virtually unbreakable.

As the Internet and other forms of electronic communication become more prevalent and protecting an execution environment is becoming more and more important. Today is more frequent to find programs that are transferred from an unknown origin and executed, in the local system. In the context of those programs security has the same meaning of the definition given above: programs have access to a well defined set of system's functionalities preventing that its execution impact the execution environment.

In this thesis we focus on security models for program languages that permits the creation of secure programs.

3.2 Security in Java

Security in the Java™ language has been an issue since the language was created. Its security model has been extended and improved in each new version of the language.

Java has become widespread because its presence in the Internet via applets, which are applications (usually small) implemented in Java that can by embedded in an HTML page and downloaded in order to be executed. Applets are used to add features to web pages that can not be obtained with the simple use of HTML. Applets are a good alternative to programs distributed instead to execute them in the server side, because then can be downloaded and executed by any Java-enable browser as well as they can be executed directly from the local machine. Nevertheless, applets can potentially attack the system that tries to load it.

The security in Java consists basically in providing a good environment to ensure that foreign code can run safely in your system. Java security includes two main concepts [Gon98]:

- ✍ The Java platform (primarily through JDK) as a secure, ready-built platform on which to run Java-enable applications in a secure fashion.
- ✍ Provide security tools and services implemented in the Java programming language that enable a wider range of security-sensitive applications, for example, in the enterprise world.

The security model of Java involves not only prevention against applets, but also to ensure that any code satisfies some conditions, such as type soundness, in order to avoid any malicious code to run in the system. However, most of the danger comes from applets loaded from remote sites.

This chapter presents an overview of the Java™ language and a description of the security model and capabilities implemented by Java. In the last section of the chapter different issues compromising the security in Java such as applet attacks and bug in the Java implementation are discussed.

3.3 The Java™ language

The Java language is a general-purpose object-oriented language that was introduced by Sun Microsystems in 1995. One of the major design goals for Java was portability. The result is that not only the Java source code, but also the binary code is executable on all processors. This is accomplished by compiling the source code into platform independent bytecode, which is then run by the Java virtual machine. In this section the most important characteristics of the Java language will be presented.

3.3.1 Main characteristics

Some features of the Java language that make it simpler and supposedly more secure are that it is strongly typed, there are no preprocessor statements (like C's #define and #include),

there are no explicit pointers (and then there is no pointer arithmetic operation), no global variables, and no global procedures.

A Java program is a collection of classes and instances of classes. Each class is compiled into an intermediate format, called bytecode, which is then interpreted to execute the program. A major characteristic of Java is that pointers are not supported; object references are provided instead. When a class instance (an object) is needed, it is created explicitly and a reference to it is returned; when a method is invoked on an object, the interpreter selects the method to be executed according to the class hierarchy and method overloading. Java uses only single inheritance. Object destruction is automatically handled by a garbage collector, so that memory management is completely in the control of the interpreter.

Java supports concurrent programming via threads. The Java Virtual Machine instructions are all one byte long, and that is why they are called bytecodes. Bytecode can also be generated from other high level languages, such as Ada or C, or it could be generated manually.

3.3.2 Types and scope in Java

Java defines eight primitive types. Variables that are declared as primitive types are not objects. They are only placeholders to store primitive values. In Java, primitive types are passed by copy and objects are passed by reference to functions.

Blocks consist of sequences of local variable declarations and statements. Blocks are statement sequences which are delimited by braces. The scope of the variables in Java is limited by the *block* where the variable was declared.

Java allows the definition of nested *blocks* (as in C or C++). When a variable name that is defined in a nested scope has the same name that in the super scope (and eventually different type), the variable of the super scope is hidden and only the locally-defined name is visible in the nested scope.

The same phenomenon happens when instances variables have the same name as variables defined in their super class or in the class that they implement. In this case, the variables defined in the super class (or interface) are hidden.

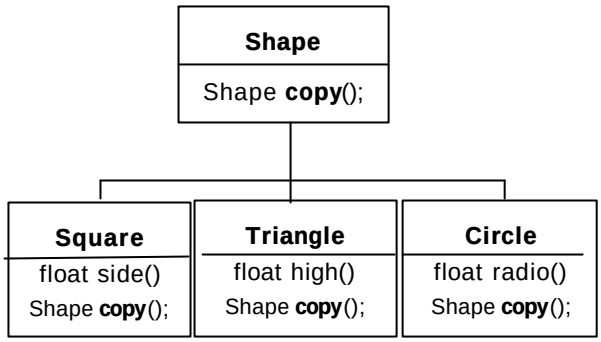
3.3.3 Cast

The "cast" is a mechanism used in Java to convert types of objects and primitive types. Supertyping in Java is made without explicit specification, i.e. an object of a given type can be seen as an object of any of its supertypes. For example, consider the class hierarchy present in *figure 3*. An object of class Triangle can be seen as an object of class Shape because inheritance.

In this class hierarchy, all the subclasses implemented the *copy()* method defined in class Shape in such a way that they all return an copy of itself, for example, an object square returns a copy of itself. Nevertheless, the object returned by the method copy is of type Shape.

If we want to change the type of the object returned to its real type (in order to have a complete view of it) it is necessary to make an explicit cast. This is done using the expression:

```
(subtype) objectExpression
```



For example, in the following piece of code:

```
Shape sh = new Square(0, 0, 2, 2);
Square sq = (Square)sh.copy();
```

we can see that an instance of class ‘Square’ called "*sh*" is created and a variable of class ‘Shape’ reference to it. Afterwards a second object of class ‘Square’ is created by means of the method *copy()* using an explicit cast to convert the Shape object in to a Square object.

Figure 3: Shape hierarchy

This piece of code is accepted by the compiler without further analysis. This is because the compiler does not check the exact type of the object. Nevertheless, the method *copy*() returns a generic object of type Shape whose real type could be Triangle or Circle. In the following code:

```
Shape sh = new Triangle(1, 0, 2);
Square sq = (Square)sh.copy();
```

an object of class Triangle is created and then cast to an incompatible object of class Square. As we have said before, the compiler does not check if the class is correct or not, so such a code is accepted by the compiler. However, at runtime, the JVM will throws an exception saying that the types of the cast's member are incompatible.

3.3.4 Applets

Applets are applications (usually small) implemented in Java. They can by embedded in an HTML page and downloaded in order to be executed. Applets can be downloaded and executed by any Java-enable browser. Alternatively, they can be executed directly from the local machine using a tool called *Java applet viewer*.

Several applets can be downloaded from the same page and run concurrently within the same web page context. The class applet defines a set of methods to control its behavior. This methods are *start()*, *init()* and *stop()*. The method *start()* define initialization tasks. The method *init()* is used to initialize the applet after it is downloaded. The method *start()* and *stop()* are used to start, pause, resume or stop the applet's execution. When the windows of the browser which is running the applet, is minimized or closed, the *stop()* method is invoked and the applet should stop to run (because the default implementation of the *stop()* method). Those methods are not final and it is possible to override them and define special behaviors for events like during finalization make the applets restart and making it 'immortal' (see concrete examples in subsection 3.6.1).

3.4 Security features in Java

In general, the Java environment provides basic security mechanisms such as type soundness, which focus on ensuring Java program safety. For Java applications, this is the only kind of security that exists; however, for Java applets a number of additional issues are addressed, as discussed in the next section.

Four security layers [McG99] provides special features to secure the environment against trusted or untrusted applets that run in the local machine. Those features are geared towards securing the system itself, independent from if the code is trusted or not, and allow or disallow several accesses from the applets, depending on the level of trust that they have. These four layers are:

Layer 1: Language and Compiler - moves the memory allocation and layout decision to runtime and removes pointers from the Java language.

Layer 2: Bytecode Verifier - Uses a simple theorem prover to verify basic safety properties of the code.

Layer 3: ClassLoader - ensures that imported classes loaded from the network execute within their own separated name space.

Layer 4: API-Specific Security - provides tools to implement different levels of security such as:

1. Disallowing all network accesses.
2. Allowing network accesses only to the host from which the code was imported.
3. Allowing network accesses only outside the firewall if the code came from outside.
4. Allowing all network accesses.

3.4.1 The class loader

Normally, the JVM loads classes from the local file system. However, not all the classes are generated from files in the file system, they can be generated also from other sources such as

the network, or can be created by an application. A ***class loader*** is the entity of the environment responsible for loading classes.

Java supplies an abstract ClassLoader class for this purpose. Because abstract classes cannot be used directly, each browser needs to declare a subclass of this class to be used by the browser for downloading classes. Each subclass must include a customized implementation of the *loadClass()* method to retrieve and download a class from the network. A class is downloaded as an array of bytes that must be converted to an instance of class 'Class'. That is, the array of bytes must be translated to the structure of a class. The ClassLoader method that actually does the conversion is *defineClass()*. Every class object contains a reference to the class loader that defined it, so related classes can be downloaded by the same class loader. These features make Java suitable for writing programs in a distributed, heterogeneous environment such as the web.

The security manager is an abstract class defined to control access to resources of the system. The job of the Security Manager is to keep track of who is allowed to do which dangerous operations with respect to a policy file which contain all the policies set by the programmer. A standard Security Manager will disallow most operations when they are requested by untrusted code, and will allow trusted code to do whatever it wants. To implement a specific security policy it is necessary to subclass the SecurityManager class and install it in the system.

3.4.2 Dynamic code verification

Even though the compiler performs through type checking, there is still the possibility of generating malicious code via the use of a "hostile compiler". Applications such as the HotJava™, Netscape™ and Internet Explorer™ browsers do not download source code, which they then compile; these applications download already-compiled class file. The HotJava browser has no way of determining whether the bytecode were produced by a trustworthy Java compiler or by an adversary attempting to exploit the interpreter.

As mentioned above, Java code was designed to run on any client; therefore, compiled Java programs are network and platform independent. The absence of physical pointers and automatic memory management help to achieve this independence. Moreover, the bytecode has been designed to fully support the typing mechanism of Java so that dynamic code verification can be performed. This is a safety and a security feature designed to prevent one from executing corrupted or malicious code.

Every Java virtual machine has a class file verifier, which ensures that loaded class files has a proper internal structure. The class-file verifier operates in two distinct phases: internal checks and verification of symbolic references.

In phase one, the class-file verifier makes sure the imported class file is properly formed, internally consistent, adheres to the constraints of the Java programming language, Once the class-file verifier has successfully completed the checks for proper format and internally consistency, it turns its attention to the bytecodes. During this part of the phase, which is commonly called the "*bytecode verifier*" the Java virtual machine perform a data-flow analysis on the streams of bytecodes that represent the methods of the class.

The *bytecode verifier* includes a mini theorem prover, which verifies that the language ground rules are respected. It checks the code to ensure that it does not forge pointers, does not violate access restrictions, accesses objects as what they are, which they call methods with appropriate arguments of the appropriate type and that there are no stack overflows. Once the verification is done, a number of important properties are known:

- There are no operand stack overflows or underflows
- The types of the parameters of all bytecode instructions are known to always be correct.
- Object field accesses are known to be legal – private, public, package or protected.

Knowing these properties also makes the Java interpreter much faster, because it does not have to check the items named before.

Phase two verifies symbolic references. A symbolic reference is a character string that gives the name and possibly other information about the referenced item – enough information to uniquely identify a class, field, or method. The JVM follows the references from the class file being verified to the referenced class files, to make sure the references are correct. Because phase two

has to look at other classes external to the class file being checked, phase two may require that new classes be loaded.

3.4.3 The Sandbox model and beyond

The ***Sandbox Model*** is the name for the security model of Java 1.0. This model consists in a very restricted environment where untrusted code obtained from the network could run. These applets could be executed in their environment and use some restricted resources such as the screen to display beans or play sounds. Meanwhile local code is considered 'trusted' and can have full access to vital system resources (such as the file system).

But this model was much too restrictive and needed to extend the model. In the next version of Java (1.1) a new concept called "***signed applet***" was introduced. Applets are signed using a key and they are recognized as trusted (having access to all the resources of the system) if the key is correct. Unfortunately, this scheme of "black or white" was still weak in the sense that applets that are not trusted are very restricted and have no permission in domains that does not represent any danger to the system.

In the latest version of Java (1.2) a more fine-grained scheme was introduced. This scheme allows some permission to be granted over the system's resources to some applets depending of where the applet comes from or its signature. This scheme is based in the introduction of a new arquitecture model called Protection Domain

3.4.4 Protection Domain based security architecture

The Protection Domain model can be seen as an extension of the sandbox model. In this model, applets loaded in the system are grouped in several domains called ***application domain***. Java programs that run from the local system are all stored in the same space called ***system domain***[2]. Each class/object belongs to one domain, and each domain is given permissions according to policy, see *figure 4*.

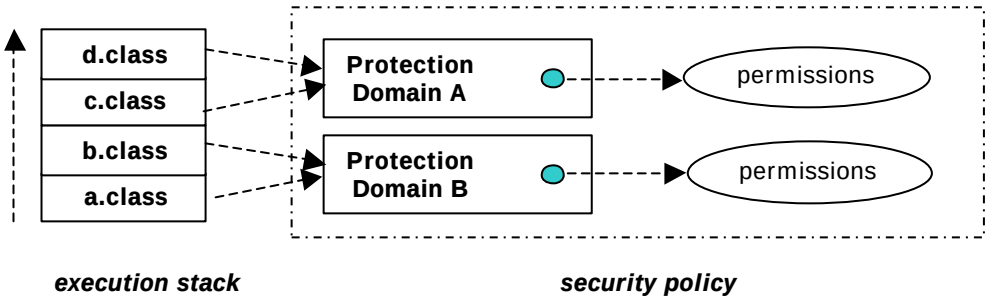


Figure 4: Complete caption

When a given object wants to access some resource in the system domain, it must make a call to the system asking for the correspondent permission. The system must answer depending of the policy associated to the domain from where the call was made.

The system executes security-check code to examine the runtime stack. Each thread of execution has its own runtime stack. The purpose of the stack is to keep a track of which method calls which other method in order to be able to return to the appropriate program location when an invoked method has finished its work.

In this way security decisions can be made with reference to this check. This is called stack inspection [McG99]. Trusted code has access to more resources invoking the *doPrivileged()* method, which will be checked in order to determine if the application that invokes this method belong to the application domain which has access to those system domains.

[2] When local applets are loaded with the applet viewer, it is possible to specify that those applets will be subject of the security polices defined in the system

3.4.5 The policy file

The capability to specify a security policy for applets and applications represent one of the powerful security features introduced in JDK 1.2. A policy file consist of a series of statements, referred to as grant entry that identify the permissions granted to code (applet or application) based on the location from which it is loaded and any signers of the code.

The grant entries of the security policy identify a code source (URL and list of signers), followed by the permissions granted to that code source. The permissions specify the action that a code source may take with respect to a protected resource (entries between "[" and "]" are optional and those symbols does not belongs to the language). The syntax of a grant entry follows:

```
grant [SignedBy "signer_names"] [, CodeBase "URL"] {
  permission_entries
}
```

Each grant specifies one or more permission entries to define the permissions that are granted to the code source described by the SignedBy and CodeBase clauses. If only CodeBase clause is specified then the grant will be given to any code that comes from this URL.

A permission entry (*permission_entry*) consists of the keyword *Permission*, followed by the fully qualified name of a Java permission class, followed by an optional target name, action list, and *SignedBy* clause. The syntax of a permission entry is as follows:

```
Permission permission_class_name [ "target_name" ]
  [ , "action_list" ] [ , SignedBy "Signer_names" ];
```

The permission class name identifies the permission to be granted. It is the fully qualified name of the Java class that implements the permission. Most of those permissions have "targets" and "actions". For example, the targets of the ***java.io.FilePermission*** permission are files or directories of the files system, and the actions associated are "read", "write" and "execution". In the following, we will see the most important permission classes.

3.4.6 Permissions classes

The permission classes represent access to system resources. As an example of a permission class, the following code can be used to produce a permission to read the file named "abc" in the /tmp directory:

```
Perm = new Java.io.FilePermission("/tmp/abc", "read");
```

java.security.Permissions represents a collection of collections of permission objects. There are several classes already subclassed which implement the most important permission. All of them are described in detail in [Sun98a].

✍ *java.security.Permission*

This abstract class is the ancestor of all permissions. It defines the essential functionalities required for all permissions.

Each permission instance is typically generated by passing one or more string parameters to the constructor. In a common case with two parameters, the first parameter is usually "the name of the target" (such as the name of a file for which the permission is aimed), and the second parameter is the action (such as "read" action on a file). Generally, a set of actions can be specified together as a comma-separated composite string.

✍ *java.security.BasicPermission:*

The base class for permissions that want to follow the same naming convention as BasicPermission (see below). The action string (inherited from Permission) is unused. Thus, a BasicPermission is commonly used as the base class for "named" permissions (ones that contain a name but no actions list, you either have the named permission or you do not.) Subclasses may implement actions on top of BasicPermission, if desired.

Some of the BasicPermission subclasses are

- java.lang.RuntimePermission

- java.security.SecurityPermission
- java.util.PropertyPermission
- java.net.NetPermission.

✍ ***Specific Permission classes***

There are several classes that inherit from the class java.security.Permission. Each implements a particular permission.

- ***java.io.FilePermission:*** This class is an important dass in that it is used to grant permission for file and directory operations. This class sets the read, write, deletion and execution permission for files in the system.
- ***java.util.propertyPermission:*** This class is used to control access to system properties. The actions are read and write which allows the applet to call *`getProperty()`* and *`setProperty()`* method in java.lang.System respectively.
- ***java.lang.RuntimePermission:*** This class is used to control access to services of the Java runtime environment For example, RuntimePermission("exitVM") denotes the permission to exit the Java Virtual Machine.
- ***java.net.NetPermission:*** This class is used to control access to network resources.
- ***java.lang.reflect.ReflectPermission:*** This class is used to circumvent the access checks performed on reflected objects. It allows all members of an object to be accessed, no matter what access is specified via the public, protected, and private keywords.
- ***java.securiaty.securityPermission:*** This class is used to grant a variety of security-related permissions to guard access to the Policy, Security, Provider, Signer, and Identity objects
- ***java.security.AllPermission:*** This class is used to grant all the permissions. Note that AllPermission also implies new permissions that are defined in the future. Clearly much caution is necessary when considering granting this permission.

3.4.7 An example of the Policy File

The following example shows a Policy file with many permissions that exemplify the way of set permissions and grants.

*An example of **Java.Policy** File*

```
grant{
       // allows anyone to listen on un-privileged ports
       permission java.net.SocketPermission “localhost:1024-“, “listen”;

       // “standard” properties that can be read by any one:
       permission java.util.PropertyPermission “java.version”, “read”;
       permission java.util.PropertyPermission “os.name”, “read”;

};


grant SignedBy “Mario, Andres, Remi”, CodeBase “http://www.emn.fr” {
       // Only the code signed by Mario, Andres or Remi, coming from
       // the site www.emn.fr to write the file “abc” in /tmp
       permission java.io.FilePermission “/tmp/abc”, “write”;
};
```

3.5 The Java security model and AOP

The Java security model has been made in such a way that it shares some characteristics with the AOP technique. In fact, all the security specifications are written in a file that is stored in the system. Those specifications are not related at all with any specific program. Thus, security becomes independent in the logical level from other aspects and from the base code. This way, the user gains *separation of concerns* (see section 2.1) in its model.

This similarity is very important for this thesis, because the aspect that is defined in section 5 is constructed using the Java security model. Since the Java security model has these AOP characteristics it becomes easier to integrate into our security aspect language.

However, when Java security model is applied using a kind of AOP technique, it is very different in the implementation level. As presented in subsection 3.4.4 many checks of permissions are inserted in many points of the classes that are related with the permissions.

3.6 Compromising Java security through applet attacks and security bugs

Today, applets have become widespread because their features make them ideal to improve web-based services. Applets are used in several ways, from graphics adorns in web pages to commercial activities (for example electronic commerce).

However, applets present also some risks to the user who loads the pages that contain them. Applets can potentially attempt several kinds of attacks, even under the constrained environment in which they run. Applets of this nature are usually called “hostile applets” [Isr99], [LaD96].

Moreover, if applets achieve bypassing the security layers of the Java model the applet can gain dangerous abilities such as read, write files in the file system, load and change any class, etc. Until today, it is known that the Java security layers can be bypassed by applets that exploit bugs in the Java implementation [Dea96]. Many bugs have been discovered in different Java implementation in the Java-enable browsers ever since Java appeared.

In this section some hostile applets are described in more detail, the types of attacks that they can attempt and the bugs that have been found in the different versions of Java and browsers such as Netscape, HotJava, and Internet Explorer.

3.6.1 Hostile applets

There are many kinds of attacks that an applet can attempt. They have been classified in four groups of attacks [McG99]:

✍ ***Attacks that modify the system***

This kind of attacks involves intrusion into the system itself. Applets that perform these attacks have been developed only in research laboratory and it is not known that they do exist outside the laboratory [McG99].

Most of the time, the applet uses some holes in the Java system (see subsection 3.6.2) to get special permissions that allows it to damage the computer where it is running. This kind of attack is considered severe because that the applet can eventually delete, write or read whatever in the victim's site. This kind of attacks is fortunately difficult to build because they require a very deep knowledge of the Java security model and its implementation

An example of this attack is an applets that gain access to the list of trusted signers and then insert its own name in the list. Afterwards, the applet can have all the privileges granted to those signers.

✍ ***Attacks that invade a user’s privacy***

Sensitive information is stored in some machines connected to the Internet such as password files, system configuration files, files containing sensitive personal or company information. For example, if an applet get access the /etc/passwd[3] file of an Unix system, the owner of the attacker applet can then intent a *password attack* in order to become "root" which can have severe consequences.

[3] The /etc/passwd is an encrypted file that contains the password (encrypted) of all the users of a Unix/Linux system.

Forging mail is also considered a kind of invasion of privacy. If some one can gain enough information to forge mail that appears to be from the victim, then the victim is exposed to a large number of serious risks. For example, the victim can be target of spam attacks.

Another issue in this context are the security holes presented by the use of software that run in the system under some privileges and risk to enable applets to get access to different resources of the system. An example of this is "*Java Wallet*" [Sun98b]: a Java Electronic Commerce Framework (JECF). This software allows applets, for example, to read vital information from your computer, has access to serial ports to dial up through modems, or modify the behavior of an element of the Java Wallet interface [Jou98]. These attacks are targeted towards the user privacy because they can access information like personal information, credit card numbers and so on.

✍ ***Attacks that deny legitimate use of the machine by hogging resources***

This attack consists in make the system resources unavailable. They are performed essentially against servers, but also can work against individual clients. This kind of attack is quite common and we can find them in many flavors, as for example:

- ✍ Applets that creates threads until the kernel of the of the machine panics [Ash99]. With this attack, at least the browser should be quitted, or if the system is not a multitasking system it will be necessary to reboot the machine.
- ✍ Applets that can stop any applets that are running and kill any other applets that are downloaded [Isr99].
- ✍ Applets that use up all available file pointers creating thousands of windows (effectively denying access to the output screen or window event queue).

The defense against those applet's attack provided by Java is very weak and can by surely improved as is discussed in section 3.3.4. Implement such as attack is not hard [Obe97] [LaD96]. Stopping denial attack is very hard and [McG99] says that "*is expected that Java will not have strong defenses against denial of service any time soon*".

✍ ***Attacks that antagonize a user***

The last kind of attacks are those ones that for example, play some bizarre noise [Isr99] through a speaker or display obscene pictures in the screen of the user. There are also some attacks that makes your browser visit a given web site over and over again, whether you want to or not, popping up a new copy of the browser each time [Isr99].

3.6.2 Bugs in the Java implementation

The Java system has been implemented by several companies such as Sun, Microsoft, Netscape, Symantec, IBM, etc. There are several security bugs in the implementations that allow untrusted code to take control of the system resources and perform malign computation.

From the first version until the version 1.2 of Java, several security bugs have been discovered. We will briefly describe the most important bugs that affect directly the bases of the security principles of Java.

✍ ***Bug in the Bytecode verifier, Princeton University, March 1996***

Java code that is not accepted by the compiler must not be accepted by the bytecode verifier either, because they should have the same semantics. Nevertheless, this bug consists precisely in that code that is not accepted by the compiler it is accepted by the bytecode verifier. This way incorrect code can be loaded.

There are many attacks that are possible [Dea96] because of this bug. In Java, all constructors must call the constructor of their superclass. Classes like SecurityManager, FileInputStream and ClassLoader have checks in its constructors in order to know if it is an applet who is invoking their constructors. Java forbids the definition of classes that

extend the SecurityManager, the FileInputStream and the ClassLoader, but corresponding "bytecode programs" may be accepted.

For example, If an applet can create an instance of its own FileInputStream class, then it will be allowed to create streams that will not be subject of the normal security checks of Java.

✍ ***Bug in Internet Explorer, August 1996***

This bug allows attacker to booby-trap a Web page that can be used to execute any DOS command in the victim's machine that run Internet Explorer 3.0, including commands that delete files, for example.

✍ ***Bug in Explorer and Netscape, Princeton University, August 1996***

Two different security flaws were found in Netscape 3.0 beta 5 and Internet Explorer 3.0 beta 2. These two flaws allow applet to gain grant to at least read/write access to the victim's files.

✍ ***Signature control bug, Princeton University, April 1997***

As it have been said in the subsection 3.4.3 the first security model was the sandbox model in which foreign code is considered 'untrusted' and then is confined to the sandbox. In the next version of Java, applets had digital signatures. If an applet's signer is labeled as trusted by the local system, then the applet is not subject to the normal security restrictions. A very serious flaw was found in April 1997 at Princeton University [Sec97]. It was present in version 1.1.1 of the Java Development Kit (JDK) and version 1.0 of the HotJava browser, both from Sun. These systems allow digitally signed applets. The flaw allows an applet to change the system's idea of who signed it. The applet can get a list of the all signers known to the local system, determine which if any of those signers is trusted, and then the applet can re-label itself so it appears to have been signed by a trusted signer. In this way, the applet can completely evade Java's security mechanism.

✍ ***Bug in the ClassLoader, Princeton University, July 1998***

This Java security flaw allows a malicious applet to disable all security controls in Netscape Navigator 4.0x. After disabling the security controls, the applet can do whatever it likes on the victim's machine, including arbitrarily read, modify, or delete files. This flaw is not directly exploitable unless the attacker uses a secondary flaw.

✍ ***Bug in the JVM, University of Marburg, Germany, April 1999:***

Another serious flaw affects the current versions of the JVM, including Sun's JDK 1.1 and Java 1.2, and Netscape's Navigator 4.x (the last version is 4.5). The flaw allows an attacker to create a booby-trapped Web page, so that when a victim views the page, the attacker seizes control of the victim's machine and can do whatever he wants, including reading and deleting files, and snooping on any data and activities on the victim's machine. The flaw is in an essential security component of the JVM. Under some circumstances the JVM fails to check all of the code that is loaded into the JVM. Exploiting the flaw allows the attacker to run code that breaks Java's type safety mechanisms. This code can set up a type confusion attack, which leads to a full-blown security breach.

4 Security in typed applets based on [Leroy/Rouaix, POPL 98]

4.1 Introduction

An applet is a program written in the Java™ programming language that can be called in an HTML page[4]. Java enabled Web browsers can display a page that contains an applet. Then the applet's code is transferred to the local system and executed by the browser.

When a client downloads an applet from a site, a security process is started by the component responsible for loading the applet that is normally the ClassLoader. The process can consist in static type checks both at the client and at the server side, cryptographic signature checks and dynamic type checks at the client. Nevertheless, a necessary condition that the program will run without violations of type soundness will be that applets are strongly typed.

In the entire spectrum of security violations performed by Java applets we can distinguish between attacks that make use of security holes and those that make use of the public functions for the applet's execution environment. In case of applets that exploit security holes it becomes necessary that security properties must be enforced by the applet's execution environment.

In [Ler98] are formulated and proved several security properties that can be expressed as well-typed conditions on applets. Those conditions can be achieved by using procedural encapsulation, type abstraction, and systematic type based placement of runtime checks. Those runtime checks are inserted in the code defining program transformations.

The aim of this chapter is to introduce a certain level of knowledge about security properties and its conditions that are presented in [Ler98]. Most of the concepts introduced in this chapter are used to build the security aspect that is defined in chapter 5.

4.2 Storing and instrumenting the environment

The four properties defined in [Ler98] are based on several concepts that help to store and control environment information. Below these concepts are first described followed by a reformulation of the four security properties.

4.2.1 Determining sensitive locations of the environment

✍ ***Sensitive locations***

Sensitive locations (also *sensitive store locations*) are variables, locations or references that have a special importance in the environment and must be protected. For example, variables that can never be written, or variables that can be written but must satisfy a given invariant.

Different devices like input/output streams or network connections or files are represented as references or locations in the environment. Usually, those devices are the targets of applets attacks and special control over them by declaring them explicitly as sensitive locations can provide a higher level of security.

✍ ***The store control***

The ***store control*** is an entity that maps *sensitive locations* (called also *store locations*) to sets of values. In this way it is possible to control all the writes to any sensitive location declared. If the value written in the location is not in the set of permitted values an error is raised. In the case of sensitive location that have not allowed to be written at all, the store control is simply defined empty (?) for this location.

---

[4] Currently, an applet is not necessarily written in the Java™ language. The Smalltalk language version developed by Dolphin [Obj99] provides an Applet development kit to write programs in Smalltalk that can be included in an HTML file and can be downloaded and executed with the help of a Web Applet Virtual Machine plug-in. The plug-in works for most of the web browsers.

It is strictly necessary that the Store Control be given at the beginning of the program's execution, before any class is loaded or any applet is run. This is because, if it is not be given from the start, some malign applet could write any value in sensitive locations and before the Store Control check the value of the sensitive locations it could restore the environment to its permitted values in order to avoid to be discovered. By giving the store control in advance, we ensure that the any error will be detected and the corresponding error message will be generated.

✍ ***The reachable locations set***

A variable ***a*** is reachable from a variable ***b***, if it is possible to obtain a reference to ***a*** starting from ***b***. For example, let us imagine a reference ***x*** to an array that contains references to objects of a given class. Then, according to the previous definition all those objects are reachable from ***x***, because there is a way to obtain a reference to them starting from ***x***.

The set of "*reachable locations* ***RL*** *from* ***v*** *to* ***s***" is defined as all the variables that are reachable from a given value '**v**' in a store '**s**'. In particular there is the set that represent all the locations reachable from any variable in the environment. This set is used to formulate the first security property (see section 4.3).

✍ ***Named types***

As presented in subsection 3.3.3 a mechanism to transform the objects type is the cast. From the point of view of security, the cast mechanism can be used to introduce invalid objects to the environment. For example, let a security policy defined for a type TextFile that is subtype of the File Type. Then to introduce an invalid object my be enough to create a new subtype of TextFile (that we will call MyBadTextFile), create an instance of this object and make a reference from a variable of type File:

```
File fd = new MyBadTextFile(ilegalValue1, ilegalvalue2);
```

And then, make a cast from the type of the illegal object to the type, which is subject of security policies:

```
TextFile text = (TextFile) fd;
```

Therefore, controlling constructors of specific types is not enough to avoid the creation of invalid objects via subtyping. Then, what the programmer can do is to define more precisely which classes can be casted to which other classes.

The set of types that will be restricted in casts is called the set of ***named types*** and is defined by a mapping ***TD (Type Definitions)*** from type names to type expressions, stating that the type ***t*** is interconvertible with its implementation type TD(t). The programmer could define these named types in advance. Then, casts associated to certain types can be introduced automatically. We will see later the advantage to do this and which properties this implies.

Making the coercion explicit facilitates the definition of the program transformations as we will see, ensuring in particular that each term has a unique type.

✍ ***The set of permitted values***

For some types of the system it is possible to define the values that are permitted for them. This is achieved by a mapping from types to values called *permitted values of* ***t*** denoted ***PV(t)***.

If the function PV is undefined for some type ***t***, this means that any value is valid for this type and it has no restrictions. This function helps to define policies that affects certain types such as paths, permissions, usernames, counters, etc.

To reference those types that are mapped by this function, we will write ***Dom(PV)*** that is the domain of the function PV.

4.2.2 Instrumentation and runtime checks

Instrumentation and checks are introduced in order to control that sensitive locations can never have values that do not belongs to its permitted values set. Finally they are used in the last two properties to ensure the integrity of the environment.

✍ ***Runtime validation of values***

The operator OK_t (one for each *named type* t) is a function used to ***perform runtime validation*** of its argument. The use of this function assumes that a set of *possible values* ***PV*** is already defined, and it will check that the argument belongs to this set.

If the runtime checking is passed, the value of the expression is returned, if the test fails, it aborts the execution of the applet and reports an error.

✍ ***Instrumented writes***

A transformation scheme called ***Instrumented Writes*** (denoted ***IW***) is defined as an insertion of an OK_t check before any write to a reference of type ***t*** with $t \in Dom(PV)$. This way it is possible to control all writes to a reference ***t*** and ensure that all the values stored in those locations will belong to *PV(t)*. This scheme is defined as the following program transformation:

$$IW(a^{t\ ref} = b^t) \quad \rightarrow \quad IW(a^{t\ ref}) = Ok_t(IW(b^t))$$

In other words this definition means that any assignment between a term that is a reference to a type t (expressed by ***t ref***) is transformed on both sides in order to ensure that the value stored in the location belongs to PV(t). Moreover, we can remark that the definition is recursive and this is necessary because in Java an assignment or a class declaration like inner classes [Sun97] can be found in a expression.

✍ ***Instrumented coercions***

This instrumentation add checks to all creations of values of type $t \in Dom(PV)$ in the execution environment, i.e. to coercions of the form **t(a)**, following the instrumentation scheme called ***Instrumentation of Coercions*** (denoted **IC**) below

$$IC(t(a)) \quad \rightarrow \quad Ok_t(t(IC(a))) \text{ if } t \in Dom(PV)$$

Both the scheme **IW** of instrumented writes and the IC scheme of instrumented coercions can be clearly expressed as program transformations. These transformations will be defined in detail in section 8.

4.3 Four security properties for typed applets

In [Ler98] four security properties are presented, which are based in the concepts presented in the section before. The aim of these properties is to ensure the safeness of the execution environment. In the following, we will present and explain them in general terms. More details and proofs can be found in the [Ler98].

As all the security properties presented in [Ler98], the aim of the first security property is to preserve a safe environment. This property requires strong conditions in the sense that the environment is strongly constrained. The security property can be reformulated as:

> *"Let **p** be a set of sensitive Locations and **R** the set of all the variables that are reachable from any variable, reference or location in the system. If **p** ∩ **R** = ∅, then for all applets **a**, we have that no applet will trigger an error by writing to a location in **p**."*

This property is based mainly in the fact that if applets have no access to any sensitive locations ***p*** through the system, then it is impossible to write in them.

To ensure the safeness of the environment is introduced the *'writes instrumentation'* scheme *(**IW**)*. All the code in the local environment is instrumented using this scheme in order to ensure that writes in the environment are safe.

Nevertheless, applet code is not instrumented and if it has a reference to a sensitive location, it will be able to perform illegal writes on it. But if in the environment applet there is no reference to any type $t \in Dom(PV)$, then will be impossible for it to have access to a sensitive location.

Then, the second security property states:

> *"Assume a semantically correct environment **e** and a semantically correct storage location **s**. Further assume that the applet environment contains no occurrence of references to any type `t in Dom(PV)`, and that all function closures in the environment and in the storage have been instrumented with the **IW** scheme (that is, **e** and **s** are obtained by evaluating source terms instrumented with **IW**). Then, for every well-typed applet in the type environment in which all the sensitive location are, it will not trigger an error writing in sensitive locations. "*

A semantically correct environment means that the variables have values that correspond to their types, and then the environment is considered correct. In other words, all the local code is instrumented used the IW scheme, assuming that the initial environment is correct. The instrumentation of writes will ensure that in the local execution environment, all the writes are valid, and because the applet does not manage sensitive locations it is impossible for the applet to write them. Then writes to sensitive locations will trigger no error and they will store only permitted values of its type ***t***, i.e. values in *PV(t)*.

Compared with the first property, the conditions of this second property do not constrain so much the environment. However, it requires different conditions: no references of type ***t*** must occur in the environment of the applet. Unless types of sensitive locations are very rare or unused, this condition can constrain much the system than property one.

The main problem of the second property is that impose a strong condition over the applets (no references to types in *Dom(PV)*). However, if all the values of type `t in Dom(PV)` that flaws in the applet's execution environment always belong to PV(t), then the applet will never write an illegal value in a sensitive location, even if it has references to types t.

Introducing the scheme to instrument coercions ***IC*** (presented in section 4.2.2) it is possible to ensure that all the values of type ***t*** ? *Dom(PV)* created in the execution environment belongs to *PV(t)*. Nevertheless, this is not enough because the applet can introduce unchecked values of the type ***t***. Then, a new less strong condition can be imposed to the applet: it must not contain any creation of types `t in Dom(PV)`. This is achieved reducing the set of named types TD to only those types ***t*** that does not belong to Dom(PV).

Mainly, the ***security property three*** says that:

> *"Assume that all function closures in the environment and the initial store have been instrumented with the **IC** scheme (that is, the environment and the initial store are obtained by evaluating source terms instrumented with **IC**). Assume also that the environment and the initial store are semantically correct. Then, for every applet well-typed in the environment and in the restricted set type definitions, we have that the applet will not trigger an error in the environment."*

The subset of type definitions that we mention here has been explicitly created in order to avoid that applets can create their own values of types that belongs to Dom(PV) in their code. This objective is reached by making those types abstract, and then, the applet can manipulate values but it can not create any value of types in Dom(t). Therefore, all the values that flow in the applet environment have been created and checked in the initial environment.

One practice that the property three formally justifies is capability-based systems: by making the type of capabilities abstract to the applets, runtime checks are necessary only at points where new capabilities are constructed and returned to the applet. Capabilities presented by the applet can then be trusted without further checks. Unlike property 2, property 3 does not require that types t ref do not occur in the typing environment E.

In some cases is not possible or convenient make the types `t ? Dom(PV)` abstract, but it is possible to adapt the approach of the security property three, by reverting to procedural abstraction and putting checks not only at coercions, but also on all values of types `t ? Dom(PV)` that come from the applet. Procedural abstraction of a certain types consists basically in allow an applet only to manage values of this type, forbidding the creation of variables of this type.

This is achieved by a standard wrapping scheme applied to all functions of the execution environment. This scheme consist in create new types for each type `t in Dom(PV)` where its values are first checked and then transformed to the original type.

The ***security property four*** can be written as follow:

> *"Assume that the execution environment and the initial store are obtained by evaluating a set of transformed bindings that have been wrapped in order to ensure that all the values that come from the applet have been checked and belongs to the set PV(t). Assume also that the environment and the initial store are semantically correct. Then, for every applet well-typed in the environment and in the initial set of restricted type definitions TD of named types, we have that the applet will not trigger an error in the environment."*

The mentioned set of transformed bindings refers to those types that have been binding with its wrapped version. Then, then applet environment work with the wrapped version that is checked in creation and then passed to the local execution environment.

5 A security aspect for the integration of type-based properties into the Java security model

In this chapter, a security aspect for Java is presented which integrate the security properties for typed applets into the current Java security model. The presentation is as follows: first, properties of the key concepts (such as sensitive locations) with respect to the integration are discussed. After having defined the syntax of the security aspect, its semantics is defined informally and examples of its use are given. Finally, a method for a formal correctness proof is sketched and some important properties of the aspect language are discussed.

5.1 Integrating the approach of typed applets

The approach presented in chapter 4 shows properties that every strongly-typed applet has. The fundamental idea is to instrument writes in sensitive locations in order to ensure that they always have permitted values. [Ler98] presents an approach to security based on an imperative language. In this thesis, a preparatory step for the definition of the aspect is to adapt and implement their work in the context of object oriented language.

5.1.1 Sensitive locations

In [Ler98] sensitive locations have been defined as variables that must not be written or must always have some restricted set of values.

In OOP variables are references to objects that are stored in memory. The state of an object can be modified both by directly accessing its instance variables or altering its behavior through its method.

Therefore, when declaring a sensitive location, we be able to which values are accepted for instance variables and which methods by which may be used to affect objects.

Sensitive locations can be seen as a fine-grained visibility modifier. The standard Java modifier *'private'* restricts the visibility of the field of a method defined in a class to only this class. A sensitive location declaration can be used to enlarge or reduce the visibility of certain objects or classes to very fine-grained domains.

For example, consider a class "Employer" that has a public method *getSalary()* that returns the employer's salary. The visibility of this method can not be restricted to subclasses, nevertheless "Workers" should not be able to use this method, only "Managers". A solution to this problem is to define "Employer" as sensitive with respect to *getSalary*() and specify that it can be invoked only by managers.

5.1.2 Objects and classes

A second important problem concerning the integration is that in class-based languages there are classes and objects. Classes can have *class variables* (called static variables in Java) that store information common for all the objects instance of this class. Classes can have also *class methods* (called static methods in Java) that do not need an instance of the class to be invoked.

Therefore, classes and objects have the necessary characteristics to consider them as sensitive locations. Declaring classes as sensitive locations means that all the objects that are instances of this class (and of its subclasses) will be considered sensitive locations and will be subject to the conditions associated to this class.

Therefore, writing a sensitive location will be understood as "*send a message to an object or access (read or write) a given instance variable of an object or class*". Consequently, the definition of a *sensitive location* is first to specify the class or object that will be considered as sensitive and then the member by which the class or object becomes sensitive.

5.2 Syntax of the security aspect language

The following syntax uses standard EBNF conventions)

securityAspect *::=* *typeDefinition*
| *sensitiveLocDec*
| *javaPermission*

javaPermission *::=* ***"PERMISSION"*** ***["TO"*** *unionList****]*** ***["FROM"*** *unionList****]*** ***"OF"*** *javaSecurityEntry*
| *peopleSetDefinition*
| *urlSetDefinition*

sensitiveLocDec *::=* ***"SENSITIVE"*** *location* ***"SATISFY"*** *booleanJavaExpression*

typeDefinition *::=* ***"SUBTYPE"*** *identifier* ***"OF"*** *identifier*

javaSecurityEntry *::=* *javaPermissionName target action*

peopleSetDefinition *::=* ***"GROUP"*** *identifier = unionList*

urlSetDefinition *::=* ***"URL"*** *identifier = unionList*

location *::=* *classMember*
| *objectMember*

classMember *::=* *varName****"."*** *memberName*

objectMember *::=* *classMember* ***"::"*** *varName****"->"*** *className*

memberName *::=* *identifier*
| *identifier****"("*** ***[****identifier* ***(****identifier****)*]")"***

booleanJavaExpression *::=* *stringLiteral*

varName *::=* *identifier****("."****identifier****)****

unionList *::=* *intersectionList* ***["UNION"*** *intersectionList****]***

intersectionList *::=* *poepleList* ***["INTERSECTION"*** *peopleList****]***
| *urlList* ***["INTERSECTION"*** *urlList****]***

peopleList *::=* *identifier* ***[","*** *peopleList****]***
| *stringLiteral* ***[","*** *peopleList****]***

urlList *::=* *identifier* ***[","*** *urlList****]***
| *stringLiteral* ***[","*** *urlList****]***

Identifiers are used to refer to names in general. They are used to reference variables defined in the aspect or to refer class name or member name. The stringLiteral is used to represent Java Boolean expressions.

5.3 General description of the security aspect

Now that the security aspect syntax is defined, its semantics is informally presented.

5.3.1 Sensitive locations

Sensitive locations, as defined in subsection 4.2.1. are used to define those classes and objects that will be eventually subject to security checks performing runtime validation and write/coercion instrumentations.

A sensitive location declaration consists in the specification of the location (that correspond to a class or an object) and the specification of the invariant (that will define the set of sensitive values PV for this class). The syntax of this declaration is:

sensitiveLocDec *::=* ***"SENSITIVE"*** *location* ***"SATISFY"*** *booleanJavaExpression*

The symbol ***location*** represents the complete description of the sensitive location. A location can be an *object* or a *class*[5]. A *class* declaration is declared as follow:

class *::= varName* ***"."*** *memberName*

Basically it defines the full class name (***varName***) consisting of an optional package definition and a class name. Then it specifies the ***memberName*** that correspond to the member of the class (instance variable or method) through which it becomes sensitive.

A declaration of an object sensitive is of the form:

object *::= class* ***"::"*** *varName* ***"->"*** *class*

It first part consists in a ***class*** declaration (explained above) that specifies the exact scope where the object is declared and instantiated[6], followed by two colons (::). Follows the name of the object followed by an arrow (->), and then another class declaration that will specify the class of the object and the member by which it becomes sensitive.

For example, we want to declare as sensitive an object of class *Manager* called *myManager because* it manages important system security policies. Let this object be declared in the method *main(String args[])* of the class *Browser*, then the declaration should be:

```
SENSITIVE Browser.main(String[]) :: Manager->myManager SATISFY …
          \_____________________/    \_____________/
          Location of the object      Object type and name
          declaration                 definition
```

The specification of a location is followed by an invariant declaration (for the sake of simplicity is based on the transformations of source code). An invariant is composed by the keyword "SATISFY" followed by a Java boolean expression (Note that we can simply use a string representation because the implementation is based on the transformation of source code.).

SATISFY *booleanJavaExpression*

This declaration means that the Java boolean expression will be checked each time that the member of the sensitive object/class is accessed (or invoked in case of methods). In the boolean expression, all the fields or methods that belong to the sensitive locations are preceded by its class or object name. This is shown in more detail in the following examples.

Three examples should clarify these declarations. The first example shows the declaration of a sensitive class and its possible values:

```
SENSITIVE Visitor.username SATISFY
          "(Visitor.username.equals(“root”) == false)"
```

The class "Visitor" is sensitive via accessing its instance variable "*username*". The set of possible values for this instance variable is any string but "root".

[5]We treat an *interface* as a *class*, because it is place where behavior (and then information) is defined. Then, policies can be defined over them to project them to all its implementers.

[6] We assume that Java programs are normalized in that object are created in initialization statements occurring in the same scope as the corresponding declaration.

As a second example consider a class BankAcount that is declared as sensitive by the invocation to its methods *"setPassword(String)"* and *"setName()"*. The requirements of the programmer are that only the administrator of the system (represented by a class of this name) is the only entity that can set the password of any BankAcount; the other requirement is that the method *"setName(String)"* can never be invoked even when it is visible[7]. Those conditions are expressed as:

```
SENSITIVE Username.setPassword(String)
        SATISFY "this.geClass().equals("Administrator")"

SENSITIVE Username.setName(String)
        SATISFY "false"
```

As it is shown, to make a method or a variable completely inaccessible the condition must be "false".

The third example shows how to declare an object as a sensitive location. To motivate the example we will suppose that in a military program all the information about the generals of this country are managed in a list of type SoldierList called "generals" declared as an instance variable of the TopSecret class. Nevertheless, other lists of militaries are stored as lists of this type. Then, to impose some restrictions only to the list of generals, the next declaration can be used:

```
SENSITIVE TopSecret :: SoldierList -> generals.getNext()
        SATISFY "false"
```

5.3.2 Named types

Named Types (as introduced in subsection 4.2.1) are used to avoid creation of objects by casting from non-valid types. The syntax of *named types* entries is:

typeDefinition *::=* ***"SUBTYPE"*** *identifier* ***"OF"*** *identifier*

This declaration specifies those types that can be cast to specific subtypes and subtypes that can cast to its super-type. For example:

SUBTYPE *Boss* ***OF*** *Employer*

This declaration has two direct implications. First, it will not be possible anymore to make direct cast from Employer to Boss without a direct injection of the type name Boss(). For example, to obtain an object of class Boss from an object Employer one must write:

```
Employer myBadBoss = new BadBoss();
Boss myBoss = Boss(myBadBoss);
```

The Second implication is that now, code that makes direct subtyping from any subclass of Employer will be refused unless it has been defined as an already known subtype of Employer. But even in this case the subtype will be explicitly cast. Taking the above example, the first sentence will be rejected. Nevertheless, giving a subtype by its type will be allowed and done by program transformations. The following code:

```
Boss myBoss = new Boss();
Employer anEmployer = myBoss;
```

will be accepted, and then transformed to:

```
Boss myBoss = new Boss();
Employer anEmployer = Boss(myBoss);
```

The use of named types, relies strongly on that the program transformations will detect every relevant transformation. For more specifics details about the program transformations done in the code, see subsection 8.2.4.

[7] This kind of requirements could occur when methods already exist (because the have been inherited or implemented because a good programming stile says that every instance variable should have an accessor), but they should not be used because security reasons.

5.3.3 Java permissions

The last alternative "*javaPermission*" of the security aspect permits the inclusion of the Java security model. Those definitions are represented by expressions that follow the syntax:

javaPermission *::=* ***"PERMISSION" ["TO"*** *unionList****] ["FROM"*** *unionList****] "OF"*** *javaSecurityEntry*

In a permission declaration, the permission is represented by a Permission class, and can specify two more elements: signers of the code and the locations where this code comes from (for further details see subsection 3.4.6).

The name of the signers and the place from where the code of the applets comes is optional and omission means that the permission is granted to everybody or to code that comes from any place.

Signers and *location* specification can be done using lists of signers and lists of locations that have been declared before using *peopleSetDefinition* and *urlSetDefinition*. These set entries are defined using the usual union and intersection operators.

Consider the following code, which presents an example in the use of specification of Java permissions and set of people and URLs:

```
GROUP ooEmn = “Annya, “Mario”, “Noury”, “Andres”
GROUP emoose = “Lucia”, “Majo”, “Andres”
GROUP friends = "Andres, Sinagi" UNION emoose

URL dcc = “www.dcc.uchile.cl”
URL ecole = “www.emn.fr, www.eleves.emn.fr”

PERMISSION TO (ooEmn INTERSECTION emoose) FROM (dcc UNION ecole) OF
           java.io.FilePermission “/temp/abc” “read”

PERMISSION TO ("Andres, Majo" INTERSECTION friends) FROM dcc OF
          java.security.SecurityPermission “Security.setProperty.*”
```

In this example, we see that the three first lines define three *groups of signers*: “ooEmn”, “emoose” and "friends". They are sets of different possible signers for the applets. Afterwards two sets of URLs are defined. The fifth declaration is a Java permission declared for all programs coming from the union of the URLs *url1* and *url2*, which are signed by the intersection of the people set defined in *ooEmn* and *emoose*.

5.4 Leroy/Rouaix's security reconsidered

In this section, we briefly discuss the impact of the security aspect defined above on the security properties defined in section 4.3.

Property one is non-constructive and not considered further in the remainder of this thesis.

Property two says that instrumenting the execution environment with the IW scheme and asking that there are no occurrence of references to any *t in Dom(PV)* in the applet, the system will be safe typed. Nevertheless, the condition of no occurrence of any type *t in Dom(PV)* is too strong and is not viable for our aspect because can become very restrictive. Our implementation therefore only relies on properties three and four.

Mainly, the property three says that an applet may access variables of type *t in Dom(PV)* only if it may not create them. Consequently, because the execution environment is the only one that can create *checked* values of those type, all the values that flow in the applet's execution environment will always belong to PV(t). However, to achieve this, it is necessary to make all the types *t in Dom(PV)* abstract in the applet. Applied to our approach, this means that any applet that wants to create objects of those types will be rejected. This restriction sounds again very strong and depends of the kind of types in PV. For example if the class *Object* has restrictions, then no applet could be accepted because in Java every class inherits from Object.

Finally, property four solves this problem of property three by wrapping the types in Dom(PV). For each type *t in Dom(PV)* an equivalent type ***t'*** is defined. Afterwards, when any applet wants to import a variable of type ***t'*** it is checked and then passed to the environment as the

real type ***t***. In our aspect it is not necessary to wrap the types. It is enough to instrument all the local calls where constructors of types *t in Dom(PV)* are included.

This is possible because in [Ler98] types have no constructor and then, the creation of types is more difficult to control. In our approach we can ensure that no applet will not create an object of a type *t in Dom(PV)* without invoking the constructor already instrumented in the execution environment.

Nevertheless, is not enough to know that constructors and functions of the execution environment are instrumented, because applets can introduce new subclasses of types *t in Dom(PV)* and then make cast to those sensitive types.

To solve this problem all the cast to types *t in Dom(PV)* in the applet code must been transformed to explicit injections to the type name to ensure that the creation of the type is being instrumented. This transformation implies a reduction in the performance of the applet execution, but fewer transformations are applied compared with those proposed in [Ler98].

5.5 Towards a formal semantics for the security aspect

The syntax definition of the security aspect (cf. the rule for the non terminal securityAspcet on page 28) clearly indicates that it is composed of the ordinary Java security model and the type-based security model.

This definition raises three main semantics issues:

1. The definition of a formal operational semantics. Concretely, such a semantics could be developed by integrating the type-based part into a semantics for plain Java or by integrating Java's stack-based approach to security into the operational semantics proposed in [Ler98].
2. The two approaches should be complementary in the sense that the combination of the two approaches to security provides a stronger security model than any of its parts.
3. Obviously, there is a certain overlap between the standard Java security model and the type-based approach. Nevertheless, they should be 'pragmatically complementary' in the sense that each individual model should be better suited for the specification of some part of the shared security properties of the stronger model.

5.5.1 Definition of an integrated operational semantics

In this subsection, we sketch how the standard Java model for security could be integrated on the basis of the operational semantics defined by Leroy and Rouaix. Basically, the set of syntactic terms has to be extended with terms for the stack-based implementation (essentially, doPrivileged() and checkPermission()) of the Java model and the evaluation rules have to be changed in order to take into account these new terms.

As an example of the terms and the rules that may be added, we could write:

```
Terms  ::=     …     (as before)
       |    doPrivileged()
       |    checkPermission_t(a)
```

And add the following evaluation rules:

$$\frac{e \to \mathit{Domain}_A \quad \mathit{Domain}_A \text{ has Permission P}}{e \to \text{doPrivileged}() \to v/s}$$

$$\frac{\sigma, e, s \to a \to \forall x.a(x) \quad e \to \mathit{Domain}_A \quad e \to \text{doPrivileged}() \to v/s}{\sigma, e, s \to \mathit{checkPermission}(a)}$$

Once this new system defined, it should not be difficult to prove that the security properties stated in [Ler98] are still valid and then prove that the integration of the two security models does not affect the validity of each one.

5.5.2 Complementarity of type-based and policy-based security

The two integrated approaches to security are quite complementary, precisely because these two approaches are different and independent from each other.

Consider a situation where a specific file is subject of some policies. Using the Java security model, the programmer is allowed to apply security specifications related with access actions like read, write or delete, for example declaring:

```
PERMISSION TO "Thomas" FROM ("www.kurupt.org") OF
     java.io.LogFilePermission "/kurupop/log.txt" "write"
```

Using the approach for typed applets, policies related directly with the class File could be specified. This way a more fine grained policy can be specified, if programmer wants to allows writes on the file when valid values are written in it, for example, continuing with the previous policy:

```
SENSITIVE PopUp.main(String[]) :: LogFile -> kurupLog.writeIn
       SATISFY " !kurupLog.writeIn.equals("ERROR") "
```

This declaration says that the file log.txt is represented in the execution environment by the instance variable created in the method main of the class PopUp called kurupLog, and that each time that the instance variable writeIn is accessed should satisfy to be different of the String "ERROR". This declaration clearly complements the previous one adding more restriction in the level of the code.

The Java security model has the advantage that the policy to apply for a given applet depends of where the applet comes from or of its signature. Therefore, the model offer a fine-grained policy application. In other hand, the approach for typed applets establish conditions to satisfy for all the applets without make distinction between them.

When Java permission classes are too coarse and a more well fine-grained definition for the policy is required, then a specification via declaring a class as sensitive and specifying its possible values and its behavior can be done using the approach for typed applets.

5.5.3 Discussion of Overlap

There are many examples where it is possible to secure some specific target using the two approaches. For example, if a file is the target of security restrictions, it will be possible to secure the file using the two models. Nevertheless, each model can offer different abstraction levels of security that are not equivalent.

The typed-based security model provides a different perspective to the problem. This model can be integrated seemingly into the object-oriented paradigm (see subsection 5.1). Files are represented in programs as classes and then it is possible to establish restrictions over instances of this class and its subclasses. In this way it is possible to control not only if applets will have access to the file, but also to control any property of the file object. Nevertheless, using this approach is not possible to specify which applets will be subject of the restrictions.

Therefore, even when the two approaches can attack the same problem, they work different levels of abstraction, transforming overlap to complementation.

5.6 Issues and considerations of the aspect

There are some issues that arise when a new paradigm like AOP is applied to some concrete concern such as security. In [Pre99] are presented some issues that appear in the application of AOP in the implementation of an aspect for robustness. In this section are presented those issues and other that have appeared during this thesis.

5.6.1 AOP paradigm

As it was explained in chapter 0, AOP is a technique where the base code and the aspect are written separately and then they are woven using a tool called weaver. In the specific application to the security aspect, some code is initially in the system, but more code can be received dynamically.

5.6.2 Inheritance and scoping

Security policies of our aspect can affect objects and classes. In the case of classes, those policies should be applied also to all the subclasses of the class affected by the policy. This is necessary because important classes such as File and System can be subclassed by an applet and then used to perform dangerous actions without being subject of checks or instrumentations.

When subclassing in Java, it is possible to overload methods and change the type of variables defined in superclasses hiding their visibility. This raises three new issues:

- When policies are defined for an instance variable of a certain type in a class, are those policies applied to any instance variable with the same name but different type defined in a subclass?
- What happens with those variables that are defined in a scope inside classes where such variable names already exist? Are the policies applied to these variables also?
- On declaration of sensitive locations, concrete methods names are specified. What happens with methods overridden in subclasses? Are they not subject to restrictions?

In the presence of inheritance it is imperative to answer these questions, otherwise the security of the system can not be warranted. This implies that an analysis of the system should be possible at every moment (compile time and runtime), and perhaps it should be necessary to keep information such as the class hierarchies.

5.6.3 Aliases

References to locations (and its recursive definition), can represent a problem, because when declaring an object as sensitive, many references to this object (and to this reference) can be done. Fortunately, in Java all references to variables are direct references to the object. Thus, it is important to protect writes to the initial variable that point to the object, as well as check for all the methods invocations and methods accessing objects of the class of the initial object.

To achieve this, the transformation written for instrument writes to object has been defined as inserting an “if” statement in all the occurrences of objects that invoke methods or access variables with the same name of the members of a sensitive location class. Then inside the “if” statement it is checked if the suspected object points to the original object by simply using the equals method defined for every object.

In Java many variables can reference to the same object. Direct writes to those references are not relevant if they are not declared as sensitive. This is because make an assignment to those references does not modify the object that they point, but it change the references. Thus, the only reference that must be subject of writes controls is the one that has been declared as sensitive.

5.6.4 Dynamic definition of sensitive locations

In [Ler98] is stated that new sensitive locations can not be protected. In the adaptation to our approach, this condition persists. Protecting classes introduced dynamically by an applet is not possible because methods and constructors of this class are not instrumented. Nevertheless, it is possible to introduce more transformations in the system and instrument all the new classes declared as sensitive, but the performance is again reduced.

5.6.5 Detecting violations

Once policies have been defined and the system is running, applets can perform execute code that violate the policies defined in the aspect.

When the violation corresponds to a Java permissions policies an exception called *SecurityException* is thrown on the applet side, then the applet can catch the exception or simply finish its program execution; the local system does not stop its execution.

However, checks and instrumentations are inserted in code of the local system as well as in applet code. Therefore, when applets perform some violation, errors may be detected in the code of the local system. It is not completely clear what should be done. There are many solutions:

stop the applet's execution, stop the local system, make the StoreControl throw an exception or make the applet throws an exception.

Unfortunately, the three last possibilities are not practical. Stop the local system is very drastic. If the StoreControl throws an exception then the security control is stopped and this is not admissible. Of course, there are cases where stop the local system can be the one alternative and the cost to do it may be less than the cost of the damage caused by the applet. If the applet throws an exception only the applet is affected and its behavior is modified because of the exceptions inserted, for which the applet has is not been designed.

From the three possibilities, the third one is less severe and could be considered together with stopping the applet execution.

6 Security results

The aspect presented in chapter 5 provides not only a simple way to specify a security policy, but also enables more possibilities to defend the system against attacks.

Moreover, this model is useful to avoid attacks based on certain bugs found in the Java implementation API (see subsection 3.6.2). In this chapter some concrete examples of attacks are presented that can be avoided when the aspect is used.

6.1 Avoiding consequences of bugs

In this section two bugs are presented that can be avoided using the security aspect.

6.1.1 Signature control bug, Princeton University, April 1997

As explained in [Sec97], this bug exploits a bug in a public method (for any applet) that erroneously return a reference to information about trusted signers instead of a reference to a copy of this information. The list of trusted signers in Java is stored in an array that is modifiable as all the arrays in Java. After that applet get this information, it can change this array and write its name or modify its own signature to a trusted one of the list.

Certainly, after having declared this list of signers as a “sensitive location” of the system, the system will never be hacked because confusion of signers, or at least because some sensible data of the disk was accessible in some way to applets.

In fact, a way to secure the system in Java is hidden this information, but -by using a sensitive location.- even when an applets knows the list of signers it is still a hard problem to falsify signatures. Moreover, when the information is public it will not be possible for applets to modify those locations.

In general, declaring that kind of information as sensitive, attacks of this class can be avoided. The next bug presented in this section exploits a bug that allows writing variables that are final or private. Nevertheless, even when the attack can introduce code that gives that enables such a facility, the use of this security aspect will prevent those writes stopping the execution if an applet tricks to access sensitive locations.

6.1.2 Bug in the JVM, University of Marburg, Germany, april 1999:

As explained in subsection 3.6.2 this serious bug allows applet to perform a type confusion attack [Sec97]. Applets may write any variable (even if it is declared static or private) incurring serious risk in the system.

However, when using the security aspect, variables that are declared as sensitive can not be written, even when there are not restrictions imposed by the language. Then, attacks of this type are useless against security specifications on sensitive locations. The only chance of the applet is to gain reflective permissions and then write variables using reflective functions provided by the Java API.

6.2 Avoiding applet attacks

We will consider two kinds of applet attacks that can be avoided by applying the security aspect.

6.2.1 Attacks that modify the system

There have not been registered applets that performs this kind of attacks in the network, but in laboratories. These applets exploit bugs like those presented before, and without them it is less much probable that this happens.

Moreover, the application of our security aspect ensures that the sensitive variables in the system will not be modified, even when the applet may access them. Further this consideration,

even when applets get access to write over those variables the application of the aspect will ensure that the system will halt before to be hacked.

6.2.2 Attacks that invade a user's privacy

As mentioned in subsection 3.6.1 sensitive information is stored in both the machine where the local main program run and in the program itself. Applets should use this information to perform other attacks or simply use the information obtained to other purposes.

The integration of the approach for typed applets is quite useful for avoiding this kind of attacks because most of the important information flowing for the system is stored in variables. To protect sensitive information like system variables it is only necessary to define them as sensitive locations.

Attacks affected by a security prevention are: extraction of local system information (like password files), user information in the browser or in the program, attacks that forge mail, and attacks that capture the browser actions to obtain behavior information.

7 Implementation

This section describes the different elements that take part in the implementation of the entire security system. Basically, there are two main parts:

- Parse and interpret the policy defined by the user using the security aspect to obtain the necessary information to create the Java policy file, the StoreControl class and the program transformations.
- Weave the base code with the policy specifications.

Along this chapter, different code generations are presented where the following font meaning is used:

`normal code`
`new inserted code`
meta variables

Meta variables are used to represent values that are determined by the aspect parser and then inserted in the generated code.

7.1 Overview

As we have said before, the first step in the process consists in parsing the policy defined by the user. During the parsing process, the Java permission declarations are transformed directly to a standard file that is used in Java to define the security policies. At the same time, a class called StoreControl is created according to the policy specified by the user. This class will be the responsible of the different controls of writes and coercions made in the system. Finally, a parser is used to build the weaver, which will apply program transformations. Those program transformations are defined in the chapter 8.

The second step consists in the application of the program transformations by the weaver in the applets code. Program transformations will be implemented using a tool called *TXL* [Cor95] (for details of the transformations see chapter 8). *Figure 6 and Figure 6* shows the general scheme of the entire process

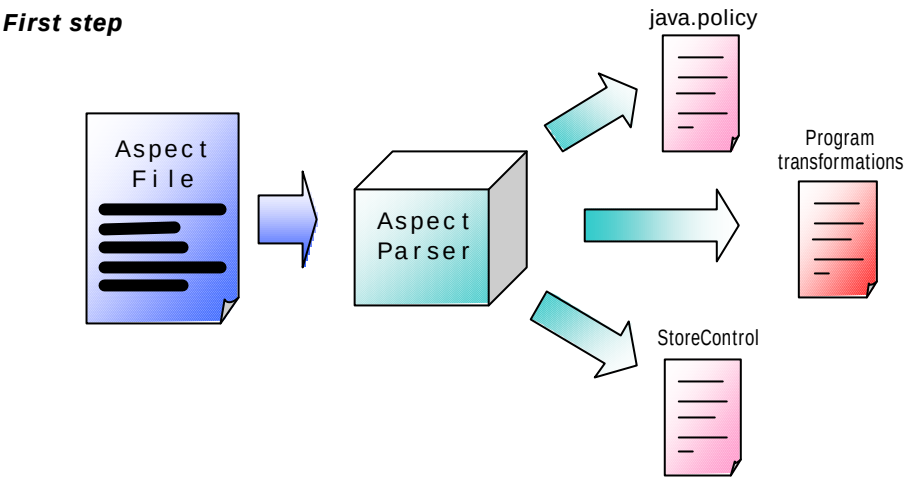


Figure 5: Parsing the *aspect file*

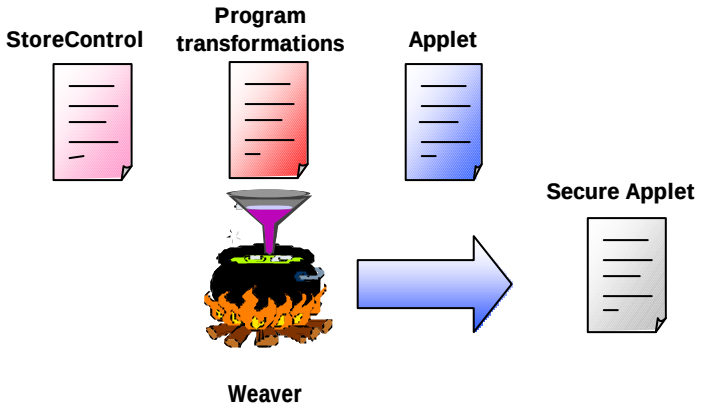


Figure 6: The weaving process

Next, the four components are presented in detail: *the aspect parser*, the *StoreControl class*, the *program transformations* and the *weaver*.

7.2 The aspect parser

The aspect parser is a program written in JavaCC [Sun99a] that parses a file with the security aspect specification and generates or modifies three different output files: the *Java policy file*, the *StoreControl class* and *program transformations* for the weaver.

An overview of the code of the parser is given in the Appendix A.

7.2.1 The Java policy file

As presented in subsection 3.4.5. the Java policy file is where the policies related with Java permission are specified. When the aspect parser reads a Java permission entry it generate the equivalent permission entry in the Java policy file.

7.2.2 The StoreControl class

The *StoreControl* class is created in order to store information related with sensitive locations and its checks and named types. The StoreControl class is composed of two instance variables, one constructor and several methods. Before parsing the aspect file, the StoreControl class has the following appearance:

```
public final class StoreControl{

    private ListOfSensitiveLocations sensitiveLocations;
    private ListOfNamedTypes namedTypes;

    public StoreControl(){
    }
}
```

The StoreControl class components are:

List of Sensitive location

The StoreControl class has a list of sensitive locations that stores all the sensitive locations of the system. This list is used to perform runtime checks.

List of Named types

The StoreControl class has a list of *named types* that is used to control the instrumentation of coercion, cast and subtyping.

✍ ***The StoreControl constructor***

This constructor is completed while the aspect file is parsed. Inside this constructor statements are generated to add sensitive locations and named types.

✍ ***Methods for Runtime checks***

Two kinds of method called "*ok*" and "*ok_assig*", are inserted in the StoreControl class to perform runtime checks for sensitive locations defined in the system. The "ok" methods are introduced in order to perform runtime checks for method invocations of sensitive locations. The "*ok_assig*" methods are inserted to perform runtime checks when instance variables of sensitive locations are accessed (for write or read).

There is one method "*ok*" and "*ok_assig*" for each sensitive location, they are distinguished by the type of the first parameter that is the type of the sensitive location. However, because a sensitive location can have several associated invariants, a second argument defined as an integer is added to specify the invariant to be checked.

The "*ok*" method tests the invariants specified in the sensitive declaration and returns true if the test succeeds and false otherwise. The method signature is:

```
boolean ok (Class a, int index)
```

The body of the "*ok*" method correspond to a "*switch*" block that checks a condition associated to the index "*index*".

The "*ok_assig*" method tests the invariants specified in the sensitive declaration and returns the object given as parameter if the test has succeeded and throws an exception if it does not. The method signature is as follow:

```
Class ok_assig (Class a, int index) throws SecurityWriteReadException
```

As the method "*ok*", the body of the "*ok_assig*" method corresponds to a "*switch*" block that checks a condition associated to the index "*index*".

7.2.3 Program transformations

Program transformations are programs written in TXL programming language [Cor95] that look for a pattern in the base code and replace the match by a predefined expression. These program transformations are generated after parsing the aspect file. Afterwards, an instance of the StoreControl class is created and the program transformations are generated.

The implementation of the program transformation in TXL is presented in Appendix B.

7.3 Parsing the aspect file

When parsing the aspect file, several actions are performed for each kind of statement of the security aspect that is read.

7.3.1 Sensitive locations

When sensitive locations are parsed, they are stored in the list of sensitive locations. Then, after parsing the sensitive declaration, two different code generations are performed:

1. Declaration of sensitive location in the StoreControl Class

The declaration of a sensitive location in the security aspect definition is implemented by inserting an object *SensitiveLocation* to the sensitive location list in the *StoreControl*. For example, when a sensitive location declaration as the following one:

```
SENSITIVE Class.field SATISFY JavaBooleanExpression
```

is parsed, the constructor of the StoreControl is modified and the following code (marked in bold) is added:

```
private StoreControl(){
```

```
        sensitiveLocations.add("Class", "field");
        …
    }
```

The execution of the method "add" will insert a sensitive location in the StoreControl class as an objet. Then, a second code generation is performed. The Java boolean expression declared in the aspect file is inserted in the OK method as stated in the section before.

2. Creation of runtime checks

The second code generation step consists in the insertion of the condition specified in the aspect to perform the runtime check. As it was said in section 7.2.2 the condition must be inserted as part of a switch block defined in the methods "ok" and "ok_assig". If the method "ok" (or "ok_assig") does not exist when the declaration is read, then the following expression is created:

```
boolean ok (Class a, int index){
      Switch (index) {
      }
}
```

The method *"ok_assig"* is created in a similar way. The parser determines the value of the meta variable *'Class'* that corresponds to the class name of the sensitive location specified in the declaration.

According to the specification of the sensitive location declaration the following code is added in the switch block (marked in bold) of the method "*ok*":

```
boolean ok (Class a, int index){
      Switch (index) {
         case number:
            return JavaBooleanExpression;
      }
}
```

For the method "*ok_assign*" the code inserted is slightly different. The condition is checked and if the check succeeds, the object is returned:

```
Class ok (Class a, int index){
      Switch (index) {
         case number:
      if JavaBooleanExpression;
               return a;
      }
}
```

The parser keeps a counter for each sensitive location and determines the value '*number*'. For each declaration of a sensitive location the counter associated to this sensitive location is incremented. The *'JavaBooleanExpression'* is that specified in the sensitive location definition.

7.3.2 Definitions of named types

When a *named type* definition is parsed, two basic code generation steps are done. The first step consist in add the *named type* (if it does not exist) and its subtype to the list of *named types*. The second step consists in the introduction of a method that will check the creation of objects of this type.

✍ Adding the named types to the class StoreControl

As described in section 7.2.2, the StoreControl class has an instance variable that is a list of types named objects. Each node of the list contains the name of the *named type* and a list of all its subtypes that are accepted as a source of objects for this *named type*.

The first code generation step consist in the addition of a *named type* element to the list of named types. In order to do this, the code to add an element to the named types

list will be inserted in the constructor of the StoreControl class. Supposing that the *named type* is *namedType* and the source type is *subType*, the code inserted will be like:

```
StoreControl(){
   …
      namedTypes.add("namedType", "subType");
   …
   }
```

✍ ***Coercion methods***

The second transformation is related to the creation of the method that will make the coercion from the named type to the subtype.

As presented in 4.2.2 the aim of coercions is to control the object creation of sensitive classes via cast from subtypes of those classes. It is performed by replacing explicit cast with injections from the subtype.

Given that the relation between subtypes and types is n to n[8], a method will be created with the name of the subtype and an argument the same type of the type declared for each subtype defined. These methods will be inserted in StoreControl class and will have the following structure:

```
public subType subType(Type source){

   subType auxObj = (subType)source;

   /** Here are performed the checks */
   allChecks("subType", auxObj);

   return auxObj;

}
```

Inside the method a cast to from the subtype to a new variable is inserted and then all the conditions declared for this subtype are checked in new variable. The method *"allCheckAlways()"* is defined inside the StoreControl class and check all the conditions associated to the class passed as first parameter.

7.3.3 The Java policy file

When a Java permission statement is parsed according to the syntax definition (see section 5.2) a plain Java policy file is generated. Java permission declarations (see subsection 3.4.6) are inserted and no further changes are necessary.

7.4 The weaver

After the aspect file has been parsed, an object of class Weaver is created. This object has an instance variable of type StoreControl that is used to obtain information about the security policies of the system and delegate the runtime checks.

The Weaver implements a method called "weave" that receives a class in both bytecode format and source form (String):

```
ByteCodeArray weave(ByteCodeArray aClass)
String weave(String aClass)
```

The method *weave(ByteCodeArray)* is an extension of the method weave that receives and returns a String. This method receives the bytecode of a class, decompile the code transforming it in a String, then invokes the method *weave(String)* giving the new string as parameter and receives a new program with the integration of the aspect and the base code. Finally the string is compiled and is returned the bytecode array of the new compiled program.

[8] A given class can have many subtypes and a given type can have a class inheritance grater than 1.

The method *weave(String)* is responsible to perform the program transformations specified over its argument.

The first action of a weaver instance is to generate the file with program transformations that contain the specialized program transformations. This file is used later by the weaver to perform the program transformations on the target code. Because those program transformations are written in TXL [Cor95], a detailed explanation is found in section 8.2 and the implementation of this transformations are written in Appendix B.

7.5 Comments

In the MMM browser[9] implementation two ways are proposed to ensure type-safeness. The first one in related with our approach because it is based in the same mechanism. Applets can be transmitted in source form and then compiled with the local compiler that ensures that the bytecode generated is not corrupted. Another way is to receive the applet bytecode and decompile it for verifications. The experience with this mechanism shows that decompilation is fast and source code is not so large than Caml bytecode, what indicates that the mechanism could be viable.

In our approach, the weaver offers two methods to weave the applets: one works with the source code and the other one with the bytecode. Source code transformations instead of bytecode transformations have been choose as a way to implement the weaver only because simplicity. Nevertheless, like in the MMM browser, it can be used as a way to enforce the security model.

An immediate consequence of this mechanism is that applet attacks based on bugs in the Java implementation where illegal bytecode were accepted, will be rejected by the process of compilation.

An example of this is the bug found in the bytecode verifier (March 1996) explained in subsection 3.6.2. The bug is in the Bytecode verifier that can accept bytecode whose equivalent in Java code is not accepted. This bug has many consequences and there are many applet attacks build in laboratory that exploit this bug for other purposes.

However, when bytecode is received it must be decompiled. Afterwards, when it is compiled (independent of the program transformations) corrupt code is rejected by the trusted compiler of the system. Given that the code is de-compiled and re-compiled, it will never happen that the bytecode verifier will receive code that has not been checked against a secure compiler.

[9] The MMM browser [Rou96] is a Web browser with applets. It uses several security approaches that are similar to those presented in [Ler98]. The browser and the applets are written in Object Caml [Ler96] and compiled to bytecode by the Caml bytecode compiler. After being compiled, applets are loaded in memory and linked with the browser by the Caml dynamic linker.

8 Program transformations

The weaver combines the base code and the aspect language defined by the programmer in one final program. There are many ways to implement the behavior of the security aspect in the target program, for example, the use of reflection. However, there is another technique that has more facilities fitting the requirements of the security aspect and provides a more flexible solution: *program transformations*.

Program transformation is a technique that consists in the definition of a set of functions making the necessary changes in the target program. These functions receive the syntax tree of the target program and return a new tree including the modifications. Using a generic framework of program transformations and analysis aspects most aspects can be implemented in a more easily way [Fra98]. In this thesis, all the program transformations are generated in TXL.

The goal of this section is to express the security aspect language in term of program transformations used to implement the weaver. As presented in chapter 7, after parsing the aspect file, an instance of the StoreControl class generates the program transformations based on the information stored in it. There are only two sets of transformations: those associated to write instrumentation and coercion instrumentation.

8.1 Syntax and semantics of program transformations

The semantic of the transformations is as follows:

υ υ υ υ? *arguments* ? ? pattern ? new tree

This is a function whose domain is a syntax tree and the result of the function is a modified tree. Each function has a name which is capitalized and (between brackets) a set of arguments. The arguments can be variables or words defining the context of the expression. Then definition sign "? " separates it from the transformation definition.

The transformation definition is composed of a pattern that is matched on the program syntax tree, a sign "? ", and another syntax tree by which the pattern will be replaced. The pattern can be defined using the parameters that are given to the function (always written in lower case).

It is also possible to use variables local to the program transformation definition that denote the subtree that has a pattern matched. For example, if the objective is to replace all the assignments to a given variable (varName) by a function call, then a program transformation can be written as follow:

```
υ? varName ? ?

    varName = Expression ?  function(Expression);
```

In this example, each time that a match is produced (when an assignment to a variable of name *varName* has been found), the variable "Expression" is bound to the syntax tree that fits in the pattern definition. Then the right-hand side tree that uses the variable "Expression" will replace the entire syntax tree that has been matched by the pattern. The pattern matched is bound (always and by default) in the variable '***match***'.

It is also possible to write some applicability conditions that are verified before accomplishing the transformation. These are written using a ";" before the condition, as shown by the example:

```
υ? var ?
       ; if var = 0
     ?
       pattern ?  new tree
```

In this example, the program transformation called "E" has an argument that is checked against the value 0. If it is the case, the transformation is executed, otherwise not.

8.2 Transformations for instrumentations

8.2.1 Program transformations write instrumentation

Those program transformations are defined in order to satisfy the conditions of the second and third properties described in [Ler98]. Those transformations are oriented to perform runtime check in objects declared as sensitive locations and instances of classes declared as sensitive locations.

The second property for typed applets requires an environment that has been instrumented using the scheme **IW** described in section 4.2.2.

The weaver constructs a specialization of this transformation for the specifications in the aspect file. The transformation consist basically in applying the transformation that checks the writes to a specific sensitive location by its field or method, for every sensitive (class or object) location defined in the aspect file.

For example, the following code shows the program transformation generated to instrument writes for a given aspect file parsed:

```
υυ υυυ_υ υ υυυυ ? ?
    ?
        υ υ υυυυ ? CLASS "c1" FIELD "f1" ?
        υ υ υυυυ ? CLASS "c1" FIELD "f2" ?
        υ υ υυυυ ? CLASS "c1" METHOD "m1" ?

        υ υ υυυυ ? CLASS "c2" FIELD "g1" ?
        υ υ υυυυ ? CLASS "c1" METHOD "n1" ?
        ...

        υ υ υυυυ ? OBJECT "o1" FIELD "a1" ?
        υ υ υυυυ ? OBJECT "o1" METHOD "m1" ?
```

This program transformation consists in the application of another program transformation called "υ υ υυυυ" (defined below in this section) for every member of every sensitive location.

8.2.2 Program transformations for instrument coercions

These program transformations are defined in order to satisfy the conditions of the third and four property described in [Ler98]. They introduce coercions in all creations of variables via casts. Constructors are not analyzed because they are transformed explicitly and any creation via constructor will already have been checked.

As in the program transformation presented before, the generation of this transformation consist in a specialization for the current aspect file. This transformation consists in applying a program transformation that makes the coercion over specific subtypes.

An example of a particular case is showed at follow:

```
υυ υυυ_υ υ υυυυυυ υ ? ?
    ?
        υ υ υυυυυυ υ ? "subType1" ?
        υ υ υυυυυυ υ ? "subType2" ?
        υ υ υυυυυυ υ ? "subType3" ?
        ...
```

The program transformation called "υ υ υυυυυυ υ" (defined below) inserts the coercions in casts and the checks for the constructor of the determined subtype.

8.2.3 Checking writes

This program transformation is used to insert code where a given sensitive variable is modified according to the definition given in section 5.1.

```
υ υ ωυ υ υ ? CLASS className FIELD fieldName ?
    ?
        Aclass "." fieldName = Expression

        ?  if (st.TypeOf(Aclass, className)){
                className aux_st = ωυ υ υ υ _υ υ ωυ υ υ  ?  ? Expression
                Aclass.fieldName = st.ok_assig((className)aux_st, index))
                }
           else
                match

υ υ ωυ υ υ ? CLASS className METHOD methodName index?
    ?
        Aclass "." methodName

        ?  st.TypeOf(Aclass, className)){
                if(st.ok((className)Aclass, index))
                   match
                else
                   st.Error()
                }
```

There are other similar transformations for those cases where the method to be matched is found in a condition statement.

When a class invokes or accesses its owns methods and instance variables, it does not need to specify the class of such members, but can also specify using “this” that is calling variables of the same class. Usually this is done to clarify code or to enlarge the visibility of the member because has been hidden by a local variable. Therefore, when the sensitive class is the target of the transformation to instrument them, slightly different transformations are applied.

8.2.4 Coercing types

This program transformation replaces any cast by a specific injection to the casted type.

```
υ υ  υ υ υ ωυ  υ  ? typeName ?
    ?
        "(" typeName ")" Expression

        ? ?  st.subType(υ υ  υ υ υ ωυ  υ   ?  typeName ? Expression)
```

In the **IC** scheme of [Ler98] this is defined slightly different from here. A runtime check is performed over the result. Here, the runtime check is performed inside the subtype’s method. Then, the semantics of both definitions are equal.

9 Conclusions

Two different security models have been presented and studied: the Java security model and security properties for typed applets. Furthermore a security aspect for Java has been defined showing that the application of Aspect-oriented programming to security permits to define easily strong security models.

The security aspect presented in this thesis is the result of merging two different approaches. These approaches are the security model of Java and the properties for typed applets as well as some extensions. The security aspect defined inherits the security features of the two models.

The most important consequences of this aspect are:

- Security holes caused by some bugs in the Java implementation can be avoided. This is done by exploiting extra protection gained from the integration that helps to secure it against applet attacks. There are at least two important bugs that have been found in the Java implementation whose consequences can be avoided using the techniques proposed in this thesis.
- The application of AOP to the security concern was successful and security is really separated from the base code, i.e. achieving all the benefits of AOP: better understandability, reusability and maintainability of the programs.
- The security aspect helps programmers to deal with security concerns on two different levels: specifying policies related with domain's entities and specifying restrictions at the code level by for example, declaring sensitive classes and named types of the system.
- The syntax of the security aspect is clear and intuitive, facilitating the specification of policies by the programmer. The expressiveness of the security aspect language permits programmers to have access to the features of complex approaches without deal directly with technical details.

We have implemented a prototype of this security aspect and have been used a generic framework for program transformations called TXL, to implement the aspect weaver.

9.1 Future work

A concern like security is very intricate and very difficult to express in a single model. We have seen the advantage to integrate two models in one security aspect resulting in a very powerful tool. Nevertheless, there are still many other models that could be incorporated in the security aspect, and there are also many problems that are not addressed by the current approaches.

One of the fields where security plays an important role is applet's attack. There are only few kinds of applet attacks that can be avoided using the current security techniques. The combination of two approaches has proven to be more powerful than each component. Perhaps joining more approaches will result in a security aspect where the system is more secure and applets can not gain more privileges than those privileges that are clearly defined.

The current implementation of the security aspect is based on programs transformations that are performed in compiled and load time and many checks at runtime time are done in order to determine if a given policy must be applied or not to a given peace of code. Those calculations affect the performance of the program execution. The use of analyses at compile or load time can help to reduce those checks in the base code.

Policies of sensitive locations are formulated independently from the applet origin or signature. Nevertheless simple modifications in the implementations can extend the semantics of the security aspect, allowing sensitive locations and to types depend on origin or signature of applets.

References

[Aks98] Mehmet Aaksit, Bedir Tekinerdogan. *Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters*. Published in Ecoop98 Workshop on AOP.

[Ash99] Joseph Ashwood. *Windows thread overrun from a Java Applet*. Web page at http://www-scf.usc.edu/~ashwood

[Cor95] James R. Cordy, Ian H. Carmichael, Russell Halliday. *The TXL Programming Language*. Software Technology Laboratory, Department of Computing and Information Science, Queen's University, Kingston, Canada. April 1995. http://www.cs.queensu.ca/~legasys/TXL_Info/index.html.

[Dea96] Drew Dean, Edward W. Felten and Dan S. Wallach. *Java Security: From HotJava to Netscape and Beyond*. Published in the 1996 IEEE Symposium on Security and Privacy, Oakland, May 6-8, 1996.

[Fra98] Pascal Fradet and Mario Südholt. *AOP: towards a generic framework using program transformation and analysis*. International Workshop on Aspect-Oriented Programming at *ECOOP*, July 1998

[Fra99] Pascal Fradet and Mario Südholt. *An aspect language for robust programming*. Is not published yet.

[Gon98] Li Gong, *Java Security Architecture (JDK 1.2)*. In the Java Site: http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html, October 2, 1998.

[Isr99] Israel Java User Group. *JUG Hostile applet collection*. http://www.java.org.il/hostile/hostile_applet.html.

[Hür95] Walter L. Hürsch an Cristina Videira Lopes. *Separation of Concerns*. Northeastern University technical report NU-CCS-95-03, Boston, February 1995.

[Jou98] Mark D. LaDue. *How secure is the Java Wallet?* Published in developer.com journal: tech focus http://www.developer.com/journal/techfocus/062998_wallet.html.

[Kic97] Gregor kiczales, John Lamping, Anurag mendhekar, Chris Maeda, Cristina Videira Lopez, Jean-Mark Loingtier, Jhon Irwin. *Aspect-Oriented Programming*. Published in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, June 1997.

[LaD96] Mark D. LaDue. *Hostile Applets on the Horizon*. Published in the Web at http://www.informatik.fh-wiesbaden.de/~turau/java/HostileArticle.html, 1996.

[Ler96] Xavier Leroy, J. Vouillon, D. Doligez, et al. *The Object Caml system*. Software and documentation available on the Web, http://caml.inria.fr/ocaml/, 1996

[Ler98] Xavier Leroy and François Ronuaix. *Security properties of typed applets*. Published in *POPL 98* San Diego CA USA.

[Lor98] David H. Lorenz. *Visitor Beans: An Aspect-Oriented Pattern*. Published in Ecoop98 Workshop on AOP.

[McG99] Gary McGraw, Edward W. Felten. *Securing Java: Getting Down to Business With Mobile Code*. Published by: John Wiley & Sons, Inc, February 1999.

[Nat85] National Computer Security Center, Fort Meade, Meryland. *Department of Defense Trusted Computer System Evaluation Criteria (The Orange Book)*, December 1985.

[Obe97] Sumit Oberai, Fariba Shaker, Michael van Dam. *Designing a Hostile Applet*. Reporter for part I of the course ECE 1741 - Trustworthy Computer Systems in the University of Toronto, http://www.eecg.toronto.edu/~mvandam/project.html March 20, 1997.

[Obj99] Object Arts Ltd. *Object Arts Products: Dolphin Smalltalk*. Web page located in http://www.object-arts.com/Products.htm, 1999.

[Pre99] Maria José Presso, Miro Casanova and Marcelo Machado. *Object Oriented Programming + Aspect Oriented Programming*. Reporter for Specialization Training of the EMOOSE master, February 22, 1999.

[Rou96] François. Rouaix. *A Web navigator with applets in Calm*. In proceedings of the 5th International World Wide Web Conference, Computer Networks and Telecomunications Networking, volume 28, pages 1365-1371. Elsevier, May 1996.

[Sec97] Secure Internet Programming Group. *HotJava 1.0 Signature Bug*. Web page: http://www.cs.princeton.edu/sip/news/april29.html. Department of Computer Science, Princeton University.

[Str96] Robert J. Stroud and Zhixue Wu. Chapter three of the book Advances in Object Oriented Metalevel Architectures and Reflection. *Using Metaobject Protocols to Satisfy Non-Functional Requirements.* Edited by Chris Zimmermann, 1996

[Sun97] Sun Microsystems Inc. *Inner Classes Specification*. Guide published at the Java site: http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc.html, February 4, 1997.

[Sun98a] Sun Microsystems Inc. *Permissions in JDK 1.2. Guide* in the Java site: http://java.sun.com/products/jdk/1.2/docs/guide/security/permissions.html, October 30, 1998.

[Sun98b] Sun Microsystems Inc. *Java wallet Frequently Asked Questions*. Published at the Java site: http://java.sun.com/products/commerce/faq.html.

[Sun99a] Sun Microsystems Inc. *Java Compiler Compiler (JCC), the Java Parser generator*. http://www.suntest.com/JavaCC/

[Sun99b] Sun Microsystems Inc. *Java 1.2.2 implementation source code.* http://java.sun.com/products/jdk/1.2/

Appendix A. The parser aspect

```
/**
 *
 * Parser for the Security Aspect
 * Made by Andrés Farías
 *
 * 8/7/1999
 *
 */

options {
  JAVA_UNICODE_ESCAPE = true;
}

PARSER_BEGIN(SecAspParser)

package parser;

import parser.util.*;
import java.io.*;


public class SecAspParser {

  /* Two list: one for the people name list and another one for the Urls */
  static ListNames groupsList = new ListNames();
  static ListNames urlList = new ListNames();
  static PrintStream outl = new PrintStream(System.out);

  public static void main(String args[]) {
    SecAspParser parser;
    if (args.length == 0) {
      System.out.println("Security Aspect Parser Version 0.1:  Reading from standard
input . . .");
      parser = new SecAspParser(System.in);
    } else if (args.length == 1) {
      System.out.println("Security Aspect Parser Version 0.1:  Reading from file " +
args[0] + " . . .");
      try {
        parser = new SecAspParser(new java.io.FileInputStream(args[0]));
      } catch (java.io.FileNotFoundException e) {
       System.out.println("Security Aspect Parser Version 0.1:  File " + args[0] + "
not found.");
        return;
      }
    } else {
      System.out.println("Security Aspect Parser Version 0.1:  Usage is one of:");
      System.out.println("         java SecAspParser < inputfile");
      System.out.println("OR");
      System.out.println("         java SecAspParser inputfile");
      return;
    }
    try {
      parser.Policy();
      System.out.println("Security Aspect Parser Version 0.1:  Java program parsed
successfully.");
    } catch (ParseException e) {
      System.out.println("Security Aspect Parser Version 0.1:  Encountered errors
during parse.");
    }
  }

}

PARSER_END(SecAspParser)


SKIP : /* WHITE SPACE */
{
  " "
| "\t"
```

```
| "\n"
| "\r"
| "\f"
}

SPECIAL_TOKEN : /* COMMENTS */
{
  < SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
| < FORMAL_COMMENT: "/**" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/">
| < MULTI_LINE_COMMENT: "/*" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/">
}



TOKEN : /* RESERVED WORDS AND LITERALS */
{
  < ACTION: "ACTION">
| < CHARACTER_LITERAL:
      "'"
      (   (~["'","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
        )
      )
      "'"
  >
| < COERCIONS: "COERCIONS" >
| < EQUAL: "=">
| < FROM: "FROM" >
| < GROUP: "GROUP" >
| < INSTRUMENT: "INSTRUMENT" >
| < INTERSECTION: "INTERSECTION" >
| < OF: "OF" >
| < PERMISSION: "PERMISSION" >
| < NOTEXECUTE: "NOTEXECUTE" >
| < SATISFY: "SATISFY" >
| < SENSITIVELOCATION: "SENSITIVELOCATION" >
| < STRING_LITERAL:
      "\""
      (   (~["\"","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
        )
      )*
      "\""
  >
| <TARGET: "TARGET">
| < TO: "TO" >
| < TYPE: "TYPE" >
| < UNION: "UNION" >
| < URL: "URL" >
| < WRITES: "WRITES" >

}

TOKEN : /* IDENTIFIERS */
{
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
|
  < #LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
```

```
        "\u3400"-"\u3d2d",
        "\u4e00"-"\u9fff",
        "\uf900"-"\ufaff"
       ]
  >
|
  < #DIGIT:
       [
        "\u0030"-"\u0039",
        "\u0660"-"\u0669",
        "\u06f0"-"\u06f9",
        "\u0966"-"\u096f",
        "\u09e6"-"\u09ef",
        "\u0a66"-"\u0a6f",
        "\u0ae6"-"\u0aef",
        "\u0b66"-"\u0b6f",
        "\u0be7"-"\u0bef",
        "\u0c66"-"\u0c6f",
        "\u0ce6"-"\u0cef",
        "\u0d66"-"\u0d6f",
        "\u0e50"-"\u0e59",
        "\u0ed0"-"\u0ed9",
        "\u1040"-"\u1049"
       ]
  >
}

TOKEN : /* SEPARATORS */
{
  < SEMICOLON: ";" >
| < COMMA: "," >
| < DOT: "." >
}

/* TOKEN : OPERATORS */


/*********************************************************
 * THE JAVA SECURITY ASPECT LANGUAGE GRAMMAR STARTS HERE *
 *********************************************************

/*
 * Program structuring syntax follows.
 */

void Policy() :
{}
{
  ( SecurityDeclaration() )*
  <EOF>
}

void SecurityDeclaration () :
{}
{
  Permission()
| SensitiveLocation()
| TypeDefinition()
| Instrumentation()
| PeopleListDefinition()
| UrlListDefinition()

}


/**
 *
 *
 *   PERMISSION NONTERMINAL FUNCTIONS
 *
 */


/** The PERMISSION declaration */
void Permission() :
{ String people = null, urls= null, javaPermission; }
{
```

```
  <PERMISSION>
   [ <TO> people = UnionList() ] [ <FROM> urls = UrlUnionList() ]
   <OF> javaPermission = JavaPermission()

{
  outl.print("grant");
  if (people != null) outl.print(" SignedBy \"" + people + "\"");
  if (urls != null) outl.println(", CodeBase \"" + urls + "\" {");
  outl.println(" {");
  outl.println(" " + javaPermission + ";");
  outl.println("}");
}
}


/** The declaration of the Java Permissions */
String JavaPermission() :
{
  Token target = null, action = null;
  String permissionName, result = "PERMISSION ";
}

{
  permissionName = Name() [ <TARGET> target = <STRING_LITERAL> ] [ <ACTION> action =
<STRING_LITERAL> ]
{
  result += " " + permissionName;
  if (target != null) result += " " + target.image;
  if (target != null) result += ", " + action.image;
  return result;
}
}

/** To define List of people's */
void PeopleListDefinition():
{
String peopleList, nom;
Token t;
}
{

  <GROUP> t = <IDENTIFIER>
  <EQUAL>
  peopleList = UnionList()
  {
  nom = t.image;
  System.out.println("DEFINITION OF " + nom + " = " + peopleList);
  groupsList.add(nom, peopleList);
  }
}


/** UnionList is an Union of Intersectionlists */
String UnionList():
{
  String result, another = null;
  ListNames auxResult;
}
{
  result = InterList()
  { auxResult = new ListNames("Union1", result); }

  ( <UNION> another = InterList()
    { if (another != null)
      auxResult = auxResult.union(new ListNames("Union2", another));
    }
  )*

  { return auxResult.toString(); }
}



/** The interseciont between peopleLists */
String InterList():
{
  String result, another = null;
```

```
  ListNames auxResult;
}
{

  result = PeopleList()
  { auxResult = new ListNames("people", result);}

    (
   <INTERSECTION> another = PeopleList()
    { if (another != null)
      auxResult = auxResult.intersection(new ListNames("aux2", another));
    }
     )*

  { return auxResult.toString(); }
}


/** Declaration of People */
String PeopleList() :
{
 Token t;
 String result, more = null;
}
{
  t = <IDENTIFIER> [ more = PeopleList() ]
  { result = groupsList.toString(t.image);
    if (more != null) result += ", " + more;
    return result;
  }
| t = <STRING_LITERAL> [ more = PeopleList() ]
  {
    result = t.image;

    /** We transform a bit result */
    result = result.substring(1, result.length()-1);
    if (more != null) result += ", " + more;
    return result;
  }

}




/** To define List of URL's */
void UrlListDefinition():
{
 String anUrl, nom;
 Token t;
}
{
  <URL> t = <IDENTIFIER>
  <EQUAL>
  anUrl = UrlUnionList()
    {
    nom = t.image;
    System.out.println("DEFINITION OF URL " + nom + " = " + anUrl);
    urlList.add(nom, anUrl);
    }
}


/** UnionList is an Union of Intersectionlists */
String UrlUnionList():
{
  String result, another = null;
  ListNames auxResult;
}
{
  result = UrlInterList()
  { auxResult = new ListNames("Union1", result); }

  ( <UNION> another = UrlInterList()
    { if (another != null)
      auxResult = auxResult.union(new ListNames("Union2", another));
```

```
    }
  )*

  { return auxResult.toString(); }
}

/** The intersecion between peopleLists */
String UrlInterList():
{
  String result, another = null;
  ListNames auxResult;
}
{

  result = UrlList()
  { auxResult = new ListNames("url", result);}

    (
   <INTERSECTION> another = UrlList()
    { if (another != null)
      auxResult = auxResult.intersection(new ListNames("aux2", another));
    }
     )*

  { return auxResult.toString(); }
}

/** Declaration of Urls */
String UrlList():
{
 Token t;
 String result, more = null;
}
{
  t =<IDENTIFIER> [ more = UrlList() ]
  { result = urlList.toString(t.image);
    if (more != null) result += ", " + more;
    return result;
  }

| t = <STRING_LITERAL> [ more = UrlList() ]
  {
    result = t.image;

    /** We transform a bit result */
    result = result.substring(1, result.length()-1);
    if (more != null) result += ", " + more;
    return result;
  }
}


/**
 *
 *
 * SENSITIVE LOCATIONS NONTERMINAL FUNCTIONS
 *
 */

void SensitiveLocation() :
{}
{
  <SENSITIVELOCATION> <IDENTIFIER> [ <SATISFY> InvarianStatement() ]
}

void Instrumentation() :
{}
{
  <INSTRUMENT> [ <WRITES> | <COERCIONS> ]
}


void TypeDefinition() :
{}
{
  <TYPE> <IDENTIFIER> <TO> <IDENTIFIER>
}
```

```
void InvarianStatement() :
{}
{
  JavaBooleanExpresion()
| MethodRestriction()

}

void JavaBooleanExpresion() :
{}
{
  <STRING_LITERAL>
}

void MethodRestriction() :
{}
{
  <NOTEXECUTE>
}


String Name() :
/*
 * A lookahead of 2 is required below since "Name" can be followed
 * by a ".*" when used in the context of an "ImportDeclaration".
 */
{
   String className;
   Token t;
}
{
 t = <IDENTIFIER>
  {
  className = t.image;

  }
  (LOOKAHEAD(2) "." t = <IDENTIFIER>
   { className += "." + t.image; }
  )*
  { return className; }
}
```

Appendix B. The TXL program transformations

This appendix contain the implementation in TXL [Cor95] of the program transformations presented in chapter 8.

The first transformation introduce the IW scheme.

```
include "java.grm"

rule InstrumentWrites

    replace [Expression]
  yea [Expression]
    by
  yea
end rule

rule wriClassMethBef className [id] fieldName [id]

    replace [Statement]
       NewStateme [PostfixExpression] ';

    by
       NewStateme [wriClassMethExpr className fieldName] ';

end rule


rule wriClassMethExpr className [id] fieldName [id]

    construct ST [id]
       st

    construct TypeOf [id]
       typeOf

    construct OkMeth [id]
       ok

    construct Index [id]
       index

    construct Aux_Var [id]
       aux_st

    replace [Statement]
       AclassName [id] '. fieldName ASel [Selector] ';

    construct IfExp [Expression]
       ST '. TypeOf '( AclassName ')

    construct Cond [PostfixExpression]
       ST '. OkMeth '( '( className ') Aux_Var ', Index ')

    construct ThenState2 [StatementNoShortIf]
       AclassName '. fieldName ASel ';

    construct Error [id]
       error

    construct ElseState [Statement]
       ST '. Error '( ') ';

    construct ThenState [IfThenElseStatement]
       'if '( Cond ') ThenState2 'else ElseState
```

```
    construct IfElseStatement [IfThenStatement]
       'if '( IfExp ') ThenState

    by
        IfElseStatement

end rule


rule writesClassField className [id] fieldName [id]

    construct ST [id]
       st

    construct TypeOf [id]
       typeOf

    construct Aux_Var [id]
       aux_st

    construct OkAssign [id]
       ok_assign

    construct Index [id]
       index

    construct IfExp [Expression]
       ST '. TypeOf '( className ')

    replace [Statement]
       Aclass [id] '. AfieldName [id] '= Expr [AssignmentExpression] ';


    construct Dec1 [LocalVariableDeclarationStatement]
       className Aux_Var '= Expr [InstrumentWrites] ';

    construct Dec2 [Assignment]
       AClass '. fieldName '= ST '. OkAssign '( '( className ') Aux_Var
', Index ')

    construct State [StatementNoShortIf]
       '{ Dec1 Dec2 '; '}

    construct StatNoShortIf [Statement]
       Aclass '. fieldName '= Expr ';

    construct IfElseStatement [IfThenElseStatement]
       'if '( IfExp ') State 'else StatNoShortIf

    by
       IfElseStatement

end rule


function main
    replace [program]
       P [program]

    construct Cn [id]
       myClass

    construct Me [id]
       method1

    by
  P [wriClassMethExpr Cn Me]
end function
```