Mechanisms for Secure Modular Programming in Java

Lujo Bauer Andrew W. Appel Edward W. Felten
Department of Computer Science
Princeton University
Princeton, NJ 08544 USA

Tech Report TR-603-99

July 28, 1999

Abstract

We present a new module system for Java that improves upon many of the deficiencies of the Java package system and gives the programmer more control over dynamic linking. Our module system provides explicit interfaces, multiple views of modules based on hierarchical nesting, and more flexible name-space management than the Java package system. Relationships between modules are explicitly specified in module description files. We provide more control over dynamic linking by allowing import statements in module description files to require that imported modules be annotated with certain properties, which we implement by digital signatures. Our module system is compatible enough with standard Java that we have implemented it as a source-to-source and bytecode-to-bytecode transformation wrapped around a standard Java compiler.

1 Introduction

Operating systems have long dealt with the problem of safe cooperation between programs that don't trust each other. Though sufficient in its own way, the solution of communicating via system calls is far from perfect. It leaves much to be desired in terms of the efficiency of communication and the richness and flexibility of the interfaces over which it takes place.

The traditional method of providing software-based protection within a program is by using abstract data types and information hiding. The primary purpose of ADTs has been to protect programs from non-malicious mistakes made by other parts of the same software system. Since ADTs have rarely been used to provide robust security, programming languages often provide only mediocre support for them. While perhaps good enough for use in essentially benevolent environments, the Java package system's implementation of ADTs [10] leaves much to be desired. Java packages have limited ability to control access to their member classes, they don't have explicit interfaces, and they don't support multiple views of modules. These characteristics make the Java package system deficient for modular programming tasks in which security is important; for example, for writing mobile applications.

Some languages, such as Standard ML [17] with its associated Compilation Manager [3], extend the idea of module-level ADTs by providing the facility for structuring modules hierarchically. Lower levels in a module hierarchy can communicate across more expressive interfaces; higher levels can enforce more restrictive ones.

An additional problem confronts dynamically linked programs: a piece of code is designed to behave properly only when its unresolved symbols are matched against the particular set of external objects with which the author intended his module to be linked [4]. But since linking is often not under the control of the programmer who wrote the module—as in the Java virtual machine, for example—steps must be taken to ensure that after linking a program will behave in a manner consistent with the programmer's intentions. The typical way of ensuring safe linking is through type-checking. This method guarantees that the types of symbols in the interfaces between modules match, but it does nothing else to ensure that the objects with which a program links will behave in the manner that the programmer expects.

We present a hierarchical module system similar to that of Standard ML that improves upon Java packages by providing explicit interfaces, multiple views of modules based on hierarchical nesting, and more flexible name-space management. Our solution to the problem of dynamic linking is to give the programmer more control over the linking process. Regardless of his inherent lack of knowledge about and control over the circumstances under which linking will occur, we wish to be able to guarantee in advance certain properties about the linking environment. Our module system facilitates such control by allowing the programmer to specify a key with which a foreign module must be digitally signed in order for linking to be allowed. The details of the linking process remain abstract to the programmer, and the linking specifications are simple and declarative.

Our module system is compatible enough with standard Java that we have implemented it as a source-to-source and bytecode-to-bytecode transformation wrapped around a standard Java compiler.

2 An Example

Our module system, like Java packages, groups classes into larger units. A module in our system consists of a set of source files and a module description file. The module description file consists of three parts:

- an export interface;
- a membership list;
- a set of import statements.

The example in Figure 1 shows what the code-generation module of a compiler might look like.

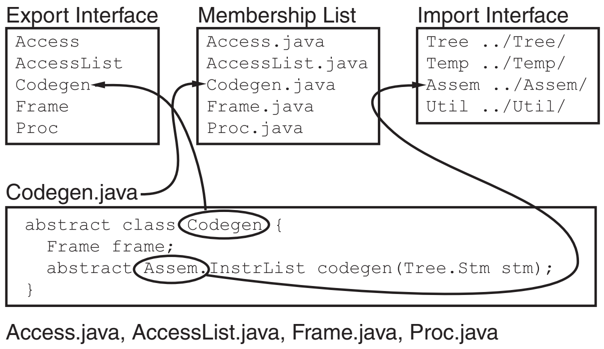

Figure 1: The code-generation module from a compiler.

The export interface of the module is a filter that allows only select classes to be visible externally. In this example, the class `Codegen` is listed in the export interface, which means that it can be accessed by other modules, rather than just from the source code of the current one. Any classes that are not listed in the export interface remain internal to the module, as if they were declared package-scope.

The source files that comprise the module are listed in the membership list. Every class that needs to be part of the module must be defined in one of these source files. In this case, the membership list includes the source file `Codegen.java` which defines the `Codegen` class. Explicitly keeping track of the members of a module is useful both from a software engineering and a security standpoint.

The only way to reference classes that are not in the module is through the import interface. The import interface introduces new Java identifiers that are bound to external modules, packages[1], or classes. In our example, the `Codegen` class needs to reference class `InstrList` from the module located in directory `../Assem/`. The import interface therefore introduces the new identifier `Assem` and binds it to the required module. All the classes that are listed in the export interface of this module can now be referenced by prefixing their names

[1]For compatibility with software written in Java that does not use our module system.

with `Assem`; e.g., `Assem.InstrList`. The names of the identifiers and directories listed in the import interface in this example match only out of convenience; no feature of the module system compels them to do so.

3 Description

The source files in our module system are standard Java source files, with a few exceptions:

- `package` and `import` declarations are omitted;
- the `public` and *package-scope* access modifiers in class declarations, though allowed, are superseded in functionality by the module description file;
- symbols defined in the module description file may be used in the source code as identifiers;
- references to classes external to the module are allowed only via the identifiers defined in the module description file.

The name space in which the source files are compiled and executed is composed of all the classes that are defined within the current module and all classes imported through the module description file.

The syntax of module description files is given by this grammar:

`Module` | `Library`
  [*classname* | *classAlias*] +
`is`
  filename *
[`imports`
  ImportStatement *]

ImportStatement:
  moduleAlias packagename |
  moduleAlias location [*keyAlias*] |
  classAlias moduleAlias.classname

where *classname* is the simple name of a Java class (e.g., `Codegen`), *filename* is the name of a Java source file, *location* is a relative or absolute pathname that must end with a path separator (e.g., `../Assem/`), and *moduleAlias* and *classAlias* are Java identifiers.

A typical module description file begins with the the keyword `Module`. The keyword `Library` indicates that the JAR file (containing the compiled classes, the module description file, and some extra information) produced by compiling the current module should include the compiled versions of all of the modules on which the current one depends.[2] Otherwise, it would contain the compiled versions of only the classes defined by the current module.

The keyword `Module` or `Library` is followed by a list of exported symbols. Each exported symbol is a class name, either from the set of locally defined classes or one that has been imported and aliased in the imports section of the module description file.

The keyword `is` concludes the list of exported symbols and starts the enumeration of the classes that comprise the module.

The module description file has an optional *imports* section which can be used to establish bindings to any external classes that are to be visible to the source code of the module. Each import statement introduces a new Java identifier (*moduleAlias* or *classAlias*). A *moduleAlias* can be bound to a package or module, a *classAlias* to a particular class from a module or package that has already been assigned an alias. Import statements refer to modules by their locations. A location could be a directory or a URL, though our system currently supports only directories. An import statement that binds a *moduleAlias* to a directory may optionally specify a *keyAlias* which represents the key with which the imported module's JAR file must be digitally signed.

The aliases introduced by the module description file can be used in the source code of the module to reference classes from imported modules. The aliased symbols may not occur bound in the source code of the module.

[2] Our module system permits modules to import standard Java packages. The linkage specifications and security features of our module system, however, do not apply to them. Any Java packages on which the current module might depend will not be included in the module's JAR file.

4 Fixing Java Packages

Our module system contains a number of features that either are not present or are insufficiently developed in the Java package system. The most important are explicit export interfaces and membership lists, hierarchical scalability and multiple interfaces, and convenient name-space management. In addition to their value as software engineering tools, we believe these are all instrumental in forming a base for developing secure software systems in Java.

Export Interfaces and Membership Lists A well-established principle of software engineering is that the interface of a module should be separate from its implementation. This enables a client of a module to be written and type-checked against the interface before the the module's implementation is written. It also allows the module's implementation to be type-checked against the same interface to ensure that the implementation adheres to its own specification. This means that the implementation of the module and any of its clients can be developed in parallel and modified independently of each other, separately type-checked and compiled, and later linked safely without further checking. Separating the interface from the implementation also aids in the construction of ADTs by making it clear which parts of the ADT form its public interface and which should remain private.

Some programming languages provide adequate support for this model of programming. C [12] allows the separation of interfaces from implementations and even the hiding of representations [11], though without enforcing it as programming discipline. Modula-3 [18] and Standard ML [1] do a good job of both separating interfaces from implementations and supporting ADTs. Java, in its native form, is lacking in both respects.

Java supports modular programming at both the class level and the package level. At the class level the `interface` facility of the language provides support for the model of modular programming in which interfaces are separate from their implementations. It has some notable deficiencies: it is impossible to flag methods declared in interfaces as either `final` or `static`, which limits the degree to which its clients can take advantage of separate type-checking and compilation. Java classes, also, are too fine-grained a structure to be particularly suitable as units of modularity for traditional modular programming.

For this purpose, Java uses the package mechanism, which provides support for modularity above the class level. Java packages do not have explicitly specified interfaces. The interface of a package is implicitly specified by the access modifiers that are part of the class declarations of its member classes, i.e., the interface is defined by the set of classes from that package that declared themselves public. Since it is defined by the component classes of a module, the interface is inseparable from the implementation. Such a scenario is clearly incompatible with the goals achieved by separating the interface from the implementation. The only way to specify the interface to a package is to write at least the skeletons of the implementations of the visible member classes. And because the implementation of the package defines the interface, there is no way to type-check an implementation *against* its own interface, so there is no way to ensure that the implementation matches its specification. As a vehicle for ADTs and separate compilation, therefore, the Java package system is sorely lacking.

Aside from the traditional software engineering goals, module systems have recently been asked to fulfill additional roles as well. With the widespread use of mobile code (e.g., applets, plugins) it has become necessary to protect systems from damage that malicious mobile code might inflict, as well as to provide environments in which mutually untrusted groups of mobile code can run simultaneously but without danger of unwanted interaction. Since mobile applications (those written in Java, in particular) typically consist of several classes, it is natural that they be organized in modules. Even when this is not done explicitly, a collection of classes that comprises a mobile application is likely to share the same set of security properties and will, from the standpoint of the system within which it is running, in many respects be treated as a de facto module. For the purpose of conceptual clarity, which helps both with formal reasoning about such applications and with the practical details of writing and executing them, it seems expedient that the status of the de facto modules be made explicit. If mobile code systems are to rely on modules to organize code, it is important for module systems to assist in providing the security functionality needed for mobile code, or at the very least not to interfere with other mechanisms used to provide security.

The Java package system is unsuited for this role. Because of the lack of explicit module interfaces and descriptions, it is inconvenient to use packages as units for enforcing security policies. The combination of implicit interfaces and the lack of explicit membership lists would make it easy for a malicious attacker to take advantage of a system for running mobile code that based its security facilities on Java packages [6].

Let us consider an example. Suppose a particular mobile application, composed of several classes grouped into one package, is trusted by the system on which it is running. The application controls access to its components by declaring certain sensitive classes private to the package and letting clients access them only through public classes which filter out any undesired or potentially malicious uses of the private classes. Because of the deficiencies of the Java package system, this manner of implementing a trusted application isn't very secure. An attacker could write a class that declared itself to be part of the same package as the trusted application—this is possible because Java packages don't have membership lists—which could then directly access the private classes of the trusted application, circumventing the filtering provided by the public classes, and use them to malicious ends.

The module system we have developed prevents any such security breach by using module description files which explicitly specify both the membership of a module and its public interface by listing all the classes that belong to each.

There are other ways of solving the security problem posed by this example; for instance, by stack inspection [20]. A disadvantage of most of these schemes is that they require dynamic run-time checking and that they are needlessly restrictive. Our scheme, on the other hand, would prevent a hostile applet such as the one described from even linking with the trusted application.

```
Module
      Graph
      Node
      NodeList
is
      Graph.java
      FlowGraph.java
      Node.java
      NodeList.java
      FlowNode.java
      GraphUtils.java
```

Figure 2: The module description file of a submodule of a register allocator.

The module description file in Figure 2 demonstrates the use of explicit export interfaces and membership lists. Only classes defined in the listed source files are considered to be part of the module. The module defines several classes, but only `Graph`, `Node`, and `NodeList` are visible to clients outside the module.

Though a significant improvement from the standpoint of security and program organization, the interfaces of our module system don't address the issue of separate compilation. The interfaces are merely lists of classes and do not describe their types, so an implementation cannot be type-checked against them. They present an improvement over the Java package system's implicit interfaces by providing the programmer with a convenient way of specifying the sets of classes that form a module and its public interface. We do not see a suitably non-intrusive way of adding support for separate compilation to Java, but as our primary goal was to explore the security aspects of modular programming, we decided against extending and complicating our module system in an attempt to solve this problem.

Our approach to organizing modules is similar to, though simpler than, the mechanism for defining units in MzScheme [8], which does support separate compilation. But whereas the primary motivation in that work is extensibility and code reuse, we are more concerned with the security aspects of modular programming.

Hierarchical Scalability and Multiple Interfaces The basic ways in which our modules support ADTs are not dissimilar from those offered by Java packages. Java's module interfaces are implicit; ours are explicit, but our interface descriptions consist only of classes, and don't describe public fields and methods of classes which are also part of a full interface. Though our module system is not powerful enough to fully describe the types of modules, it makes it simpler to control and enforce the visibility of member classes. The interfaces of both systems have similar access control capabilities: a class can be either publicly visible or visible only to other classes inside the same module. The feature that sets our module system off from Java packages, however, is the ability to structure modules so as to provide different views to different clients.

Often when using ADTs we come across situations in which we would like a module to export a richer interface to a few select modules and a more restrictive one to everyone else. In a language like Standard ML a module can supply different export interfaces to different clients. Modula-3 also has that ability, though module interfaces in Modula-3 may not overlap in the sets of members they expose [18]. Java's methods of controlling accessibility (through making classes and their fields private, protected, package-scope, or public) aren't expressive enough, so Java resorts to using a security manager to determine at run time whether a client is allowed to access a particular restricted class. The security manager suffers from a number of problems, from run-time overhead to the inability to interact with the programmer except interactively. Its complexity and ambiguities have made it vulnerable to security breaches and made it difficult to reason about and form security policies [5].

An elegant approach to the problem of multiple interfaces has been presented by research in hierarchical modularity. Hierarchical modularity is the idea of grouping several modules and attaching to such a group its own interface. The group is itself a module whose publicly visible members can be imported by other modules. The members of the group can communicate among themselves through their own interfaces, which can be much less restrictive than the group's top-level interface. This approach can be applied repeatedly to create a hierarchy of modules. For a comprehensive treatment of hierarchical modularity see Blume and Appel [3]. We use a similar approach for Java.

The natural way to use hierarchical modularity to provide different levels of access to different modules is to group together the modules that wish to share a high level of access with each other, and let them have appropriately unrestrictive interfaces. The entire group can have a more restrictive interface that exports only those parts of its member's interfaces that ought to be available to the public. Our module system supports hierarchical modularity by allowing modules to explicitly list the sub-modules on which they depend. Modules can export not only classes that have been defined in their own source files, but also classes that have been defined in imported modules. When its module description file begins with the keyword `Library`, compiling a module produces a JAR file that includes the bytecode of all the imported modules, which are then kept hidden by the export interface.

The modules that comprise a compiler, for example, are likely to need a high degree of access to each other. At the same time, we may wish to treat the entire compiler as a module that exports only a few of its classes. Figure 3 is a module description file of the main module of a compiler; it illustrates this approach. The main module imports all the sub-modules that implement different parts of the compiler and defines only a few classes that tie the sub-modules together into a working system. One of the modules it imports is `Codegen`, the code-generation module. `Codegen` defines and exports the classes `Access`, `AccessList`, `Codegen`, `Frame`, and `Proc`. Though these are visible to the source code of the top-level module, they are not publicly accessible. Only the class `Main`, the top-level interface to the compiler, is left visible as the export interface of the group. The hierarchical structure is transparent to a user; he has no way of knowing that the compiler module is composed of sub-modules.

Apart from the need for modules to support multiple interfaces, there is another reason for introducing hierarchical modularity. Windows 95 has over 10,000,000 lines of code [2]. If it were structured in just a two-level

```
Library
       Main
is
       Main.java
       NullOutputStream.java
imports
       Codegen ../Codegen/
       RegAlloc ../RegAlloc/
       Absyn ../Absyn/
       Tree ../Tree/
       ...
       Types ../Types/
       Util ../Util/
```

Figure 3: The module description file of the top-level module of a compiler.

framework of classes and modules, either there would be too many modules or each module would be much too big. This strongly suggests that a hierarchy of modules is necessary.

Name-Space Management An additional software engineering benefit is our module system's flexible and convenient name-space management scheme. Although the naming convention used with Java packages suggests that they support a hierarchical naming scheme, packages with names like `java.awt` and `java.awt.color` have no more in common than packages with completely different names.

One of the reasons for grouping code into packages is to avoid name clashes between classes. But Java packages are themselves named, so that merely lifts the problem to the package level. Instead of a name clash between two classes called `Parser`, we might have a clash between two classes called `Util.Parser`. The accepted way of solving this problem is to give packages long, unique names. This isn't a particularly appealing solution, however, since it interferes with the package system's ability to provide convenient name-space management; classes must now either be referred to individually using their cumbersome package name (e.g., `java.awt.image.renderable.RenderableImage`) or be imported en masse using the `*` notation, which again introduces the possibility of name clashes because the names of the imported classes are stripped of their unique package prefixes.

Our modules, on the other hand, are not named, so they don't suffer from this problem. Modules are assigned names only via import statements of individual module description files; this type of name-space thinning makes it easy to keep their names short and simple. In source code the names of external classes are always prefixed with the name of their module, so name clashes between classes with same names are easily avoided.

The name-space management scheme we use has been borrowed without much modification from the approach Blume and Appel have developed for Standard ML [3].

5 Secure Linking

The behavior of a program fragment depends not only on its own code but also on the libraries with which it is linked. Under the static linking model, compiling and linking a piece of code generates an executable that is fully self-contained. The libraries with which the program is linked, as well as the finished product, are available for the programmer's perusal. He therefore has good reason to expect that the self-contained executable will behave in the desired manner, even if it is executed on a machine that has a different software environment and a different set of libraries.

Today most executables aren't fully self-contained, but need to dynamically link with system libraries when they are executed. This provides us with the flexibility to update or change parts of all programs on a system simply by swapping in a new module. Should we swap in a new I/O library to replace an old one, all executables that use that library will automatically have access to the updated code. If the executables were statically linked, on the other hand, we would have to relink each of them—inefficient and inconvenient, at best.

Dynamic linking has become very popular, especially with languages such as Java, which adopt it as a key feature [14]. But despite the proliferation of dynamic linking, only a few attempts have been made to extend the model of correctness that holds for statically linked code [5, 7]. Programmers believe that programs will behave in their intended manner even though much of the programs' behavior depends on the system libraries of foreign and unknown systems.

This belief is based mostly on the existence of standards that seek to ensure the uniformity of library code (e.g., all Java virtual machines and their associated system classes are expected to meet Sun's standard). There are very few guarantees, however, about adherence to a standard that are expressed in a way that *programs* can understand. The guarantees are largely verbal or written in English, and can't be reasoned about or manipulated at the level of program code. Additionally, standardization does not apply when linking with third-party libraries. The only widely used method of ensuring safe linking, and the method used by Java, is type-checking the interfaces between program fragments. Recent research has formally shown that strongly typed mobile code has desirable security properties [13], and provided ways of ensuring that type-safety is preserved by the linking process [9]. Still, though type-checking is useful in ensuring that programs and libraries at least agree on the types they are using, it falls far short of guaranteeing that code will behave in the expected manner.

Stronger guarantees are needed, especially when a system must trust the behavior of a particular executable, such as an applet. Java often uses code signing for such purposes [19, 15]. While it does not guarantee acceptable behavior, code signing does provide a way to identify the author of a piece of code, and thus to attribute blame if the program misbehaves.

Both the theory and the practice of code signing are insufficiently developed. Though it may be somewhat useful to annotate code with certificates of authorship, we really want to annotate it with properties that describe its behavior. While providing some protection to the virtual machine against code that runs on it, code signing provides no guarantees to code about the virtual machine, nor to different code fragments about each other. Ironically, current code signing practices allow a programmer to be held responsible for the behavior of his code, while not providing him with the means of ensuring that the system on which his code is running is itself behaving in the expected manner. The implementation of code signing in Java, too, renders it susceptible to various security attacks. It is often possible for unauthorized clients to gain access to the services of a trusted applet or plugin and use them to malicious ends.

Our module system makes headway on this issue by allowing the programmer to require certain properties of the modules on which his code depends. If the required properties are not present, the program won't link or execute. If they are present, the programmer can more realistically expect that his program, once linked, will behave in the desired manner. Furthermore, the programmer can annotate his own module with certain guarantees which are held to be valid once linking has succeeded. We thus establish a system in which a module can assert that if the modules it imports can guarantee certain behavioral properties, then it, too, will behave in a certain manner.

We implement annotation of properties through digital signatures. The JAR file of a module may be signed with one or more keys, each of which represents a property. The import statements of a module description file can specify the key alias of the key with which an imported module must be signed. One of the modules the compiler in Figure 4 imports is a parser. The compiler wishes to advertise itself as unicode-friendly, but in order to make such a claim it must rely on the unicode-friendliness of its parsing module. Since the various modules of the compiler might be dynamically linked at run time by the Java virtual machine, the top-level module of the compiler needs to be sure that it will be linked with a parsing module that has the appropriate functionality. An import statement in the module description file of the compiler specifies that the parsing module must be annotated with the `unicode` property. Linking will be allowed only if the parser's JAR file is signed with the key that corresponds to this property. The main module of the compiler can itself be signed with the same key, which makes it possible for the compiler's clients to require the compiler to have the `unicode` property.

Since a program will not execute unless it is convinced that its subcomponents are usable, our approach com-

```
Module
       Main
is
       Main.java
       NullOutputStream.java
imports
       ...
       Parse ../Parse/ unicode
       ...
```

Figure 4: The module description file of the top-level module of a compiler, annotated with additional linkage directives.

plements traditional code signing well. Authorship can be regarded as just another property, and the author of a program may now actually be willing to be held responsible for the correct behavior of his code.

It should be noted that our use of explicit import interfaces somewhat restricts the flexibility of dynamic loading. In Java it is possible, at run time, to load classes whose names are unknown at compile time. Explicit import interfaces require the programmer to specify, prior to compilation, the locations of the modules on which his code depends. Though class names do not have to be specified in the import interface, the locations of the modules, at least, need to be known at compile time, which precludes some interesting uses of dynamic loading. This restriction isn't too limiting, however, since in most cases it should be possible to structure code so that even if the name of a particular class isn't known at compile time, the location of its module is.

6 Implementation

We have developed a prototype implementation that illustrates the features of our module system. Our main goal in designing the prototype implementation was to enable our system to be used easily with various existing Java compilers and virtual machines.

Our modules can be translated into Java packages. Some of the features of our module system, however—in particular its ability to place various constraints on linking—cannot be expressed just using Java bytecode. Because of this, our prototype implementation needs to provide additional features both to the compiler and to the virtual machine.

Compilation The compilation phase of our implementation is a wrapper around a standard Java compiler that consists of a preprocessing and a postprocessing step.

The job of the preprocessing phase is to translate the source code used in our module system into equivalent standard Java source code. The first step of this process is to represent our modules as Java packages. Each module is assigned an artificially generated package name, mapping the hierarchical set of modules into a flat name space of packages. We rely on the assignment of artificial package names to avoid name clashes. In addition to assigning each module a package name and adding appropriate package declarations to source files, this step must also translate class references made through identifiers introduced in the module description file (henceforth called symbolic names) into class references that can be interpreted by a Java compiler (henceforth called actual names). Because identifiers in Java are classified into several name spaces, and to detect and avoid conflicts with locally bound identifiers, we have to parse the source code to determine which tokens need to be changed. As qualified names from the original source code are resolved by replacing identifiers introduced in module description files with the package names of the modules they represent, our compilation manager assures that the restrictions imposed by export interfaces and digital signature requirements are obeyed.

At this point our modules have been translated into ordinary Java source code and can be compiled with any standard Java compiler, without the loss of any functionality added by our module system.

The compilation phase also has a post-compilation step. Our modules can export symbols that have been defined in imported modules, so it is possible that several module description files need to be traversed to discover to which class a qualified identifier is pointing. This resolved name is the one used when the code is being compiled. Consequently, the bytecode of one module can depend on the source code of several; from a viewpoint

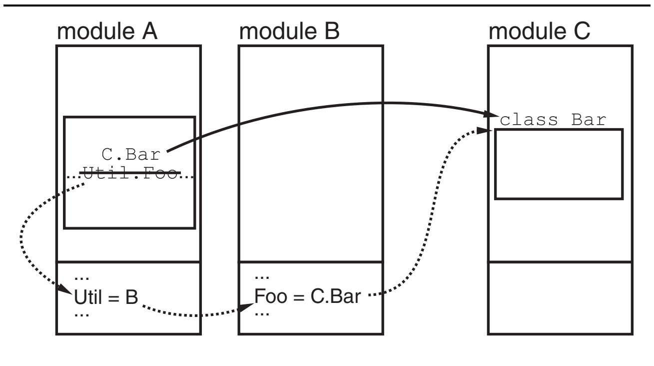

Figure 5: Resolving class references.

that favors separate compilation, this is undesirable.

To allow separate compilation of modules, we replace the resolved references in the compiled bytecode with their original names. Thus all external references are again made only through identifiers defined in the module description files, releasing each compiled module from unwanted dependencies on the source code of others.

There are cases, unfortunately, in which it is difficult to restore a resolved identifier to its original name. A particular module description file, for example, might bind two different identifiers to the same class. Preprocessing would replace the two different identifiers with the same new one. After compilation, we might not be able to discover which of the two is which. In this situation our compilation manager arbitrarily picks one and adds an annotation to the module's JAR file. This annotation can later be used to check whether the bindings that were used at compile time are still valid, and otherwise warn that recompilation is necessary.

Figure 5 shows an example of name rewriting. To resolve the reference to `Util.Foo`, module A first consults its module description file to discover that the identifier `Util` is bound to module B. From module B's description file we learn that class `Foo` is reexported rather than defined in B, and that the real name of the class is `C.Bar`. The reference to `Util.Foo` is replaced by a reference to `C.Bar`. But since module C is part of the hidden implementation of module B, it is possible that it may change after module A has been compiled. After compiling module A, therefore, the rewritten reference is returned to its original name, `Util.Foo`.

Execution in the Virtual Machine Dynamic linking in the Java virtual machine is managed by class loaders. Class loaders can be extended to allow the virtual machine to load bytecode from sources other than the local file system, and they can also be modified to support arbitrary mappings from class names to objects. The latter feature makes them useful for our purposes.

Each module description file sets up a mapping from identifiers to the classes they represent. The same identifier can therefore represent different classes in different modules. A request to load a certain class, too, may be allowed or denied depending on whether the class is signed by the digital signature required by the calling module. To deal with this issue, we have to provide the Java virtual machine with the ability to answer `loadClass` requests differently depending on the module from which they originate, which it otherwise has no way of doing.

Since `loadClass` requests are handled by the class loader that loaded the class that is making the call, our solution is to extend the `ClassLoader` class with the functionality we desire. We instantiate a new copy of this class loader for every module that is loaded by the virtual machine. Our class loader uses the module description file to set up the appropriate class environment and control linking in the manner specified by export filters and digital signature requirements. After the virtual machine is initialized, a wrapper class loads our customized class loader, which then loads the modules to be executed.

Each of our class loaders has direct access only to its own module description file. When a class requests that a class from a different module be fetched, the requestor's class loader passes the request to the appropriate module's class loader. That class loader, in turn, verifies whether the request can be fulfilled vis-à-vis that module's export interface and digital signatures. If the requested class is merely being reexported, the request will be passed on to the next class loader in the chain.

Name Hacking The process we described for running code written using our module system isn't quite complete. The ability to customize class loaders can easily be misused. If a class loader, for example, was asked twice to fetch the same class and returned two different objects, the type system would be broken and the security of the system would be compromised [5, 7]. Newer Java virtual machines have instituted a host of ad hoc security measures to guard against such breaches.

The full name of every compiled class is encoded in its bytecode. Among other restrictions, the virtual machine now verifies that the encoded name of a class returned in response to a `loadClass` request matches the name with which `loadClass` was invoked. Class names in our module system contain identifiers defined in module description files; these names may bear little relation to the actual package names assigned to the classes they reference. With the new security checks, it is no longer possible for our class loader to naively redirect `loadClass` requests to classes whose names don't match the requested ones.

Our solution is to rewrite the bytecode of compiled modules, replacing symbolic names (those defined through module description files) with actual ones. This is done while a class is being loaded into the virtual machine, before linking or bytecode verification. The procedure for resolving symbolic names is virtually identical to the one we use during preprocessing when source code is rewritten.

Since modules may reexport classes, resolving symbolic names requires tracing through module description files to locate the module in which a given class is defined. This is necessary in order to find which package name has been assigned to the module to which that class belongs. An unfortunate consequence, therefore, of the bytecode rewriting is a slight restriction on the laziness of dynamic linking. A Java virtual machine might delay the loading and linking of a referenced class until the point of execution at which the class is actually needed. Our rewriting technique, on the other hand, resolves all references at load time, so at that point it must access the module description files of all referenced modules. Since it doesn't need to actually load classes from the referenced modules, the chain of modules that need to be accessed for a particular reference to be resolved ends as soon as the module that defines the referenced class has been found.

The Reflection API Unsurprisingly, our system interacts badly with Java's reflection API [16]. The purpose of the reflection API, including the `getName` and `forName` methods of `java.lang.Class`, is to discern run-time information about classes that may not be available at compile time. Regardless of our module system, it is dubious whether such a facility should be available for public use. Though it is often convenient for the programmer, use of the reflection API undermines the goals of programming with ADTs, revealing information that may be purposefully hidden by subclassing and the use of Java `interfaces`.

The security features of class loaders require that the implementation of our naming scheme differ considerably from the view presented to the programmer. This effectively renders the `forName` and `getName` methods useless. The former is used to create new instances of classes with a given name. But the name a programmer would use in source code has been changed during compilation and kept hidden. Even though a class loader could resolve the requested name to its new version, the security restrictions placed on class loaders would prevent it from returning the correct object. `getName`, on the other hand, would reveal the internal names of objects. Classes that form the interface of a module may be either local or imported from elsewhere; revealing one or the other would be a breach of security.

Redirecting method calls from `forName`, `getName`, and other methods of the reflection API to specialized functions that would prevent certain information from being revealed might restore most of the functionality of the API. It would require extensive bookkeeping and indirection, however, and would not be completely transparent to the user. For the time being we have decided to set aside concerns about reflection.

7 Conclusions and Future Work

Our module system is based on explicit module descriptions. Membership lists and explicit export interfaces protect module integrity. Unnamed modules and declarative import statements provide simple and convenient name-space management. Variable levels of access to modules are supported by arranging modules in hierarchies. Increased control over the linking process, implemented by allowing import statements to specify digital keys for imported modules, helps guarantee correct program behavior in the presence of dynamic linking.

Any attempt to develop a secure programming environment is likely to be based on a module system. In the case of Java, such a module system should provide modularity at the level of Java packages, but should also provide explicit interfaces, which Java packages do not. Explicit module descriptions seem to be a very useful feature, both for providing an increased level of security and for simplifying the task of designing and understanding modular software systems. We have discovered that it is possible to implement such a module system as an extension to Java and retain the use of existing compilers and virtual machines.

The reflection API is a serious obstacle to sophisticated module systems that support nesting and reexporting and have opaque interfaces. Although there may be ways to limit the impact of the reflection API on the security of such systems, the purpose of the API is contradictory to the goals of using ADTs, and it would be preferable if its necessary features were provided in a different way.

Dynamic linking is an area that deserves more study. It is important to provide guarantees—ones that programs can reason about—about the behavior of dynamically linked libraries. Only thus can we trust dynamically linked programs that rely on them to behave in their intended manner. Our module system provides a good framework for annotating code with such guarantees. We demonstrate a method for allowing interrelated modules to require certain rudimentary properties of each other. We plan to continue work on making these linking requirements more expressive and giving modules even more control over the linking process.

References

[1] Andrew W. Appel and David B. MacQueen. Separate Compilation for Standard ML. In *ACM Conference on Programming Language Design and Implementation*, pages 13–23, June 1994.

[2] Ronald Baecker and Blaine Price. The Early History of Software Visualization. In John Stasko, John Domingue, Marc Brown, and Blaine Price, editors, *Software Visualization*, chapter 2, pages 29–34. MIT Press, 1998.

[3] Matthias Blume and Andrew Appel. Hierarchical Modularity. To appear in *ACM Transactions on Programming Languages and Systems*, 1999.

[4] Luca Cardelli. Program Fragments, Linking, and Modularization. In *24th ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, pages 266–277, January 1997.

[5] Drew Dean. The Security of Static Typing with Dynamic Linking. In *Fourth ACM Conference on Computer and Communications Security*, Zurich, Switzerland, April 1997.

[6] Drew Dean, Edward W. Felten, Dan S. Wallach, and Dirk Balfanz. Java Security: Web Browsers and Beyond. In Dorothy E. Denning and Peter J. Denning, editors, *Internet Beseiged: Countering Cyberspace Scofflaws*. ACM Press, October 1997.

[7] Richard Drews Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, January 1999.

[8] Robert Bruce Findler and Matthew Flatt. Modular Object-Oriented Programming with Units and Mixins. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 94–104, Baltimore, Maryland, September 1998.

[9] Neal Glew and Greg Morrisett. Type-Safe Linking and Modular Assembly Language. In *Conference Record of POPL '99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 250–261, San Antonio, Texas, January 1999.

[10] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java series. Addison-Wesley, Reading, Massachusetts, June 1996.

[11] David R. Hanson. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachussetts, 1996.

[12] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Software Series. Prentice Hall, Englewood Cliffs, New Jersey, 2nd edition, 1988.

[13] Xavier Leroy and François Rouaix. Security properties of typed applets. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 391–403, San Diego, California, 19–21 January 1998.

[14] Tim Lindholm and Frank Yellin. *The Java virtual machine specification*. Addison-Wesley, 1997.

[15] Gary McGraw and Edward Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley and Sons, New York, 1999.

[16] Sun Microsystems. Java Core Reflection. `http://java.sun.com/products/jdk/1.2/docs/guide/reflection/spec/java-reflection.doc.html`, 1998.

[17] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[18] Greg Nelson, editor. *Systems programming with Modula-3*. Prentice Hall Series in Innovative Technology. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[19] Monica Pawlan and Satya Dodda. Signed Applets, Browsers, and File Access. *Java Developer Connection*, April 1998. `http://developer.java.sun.com/developer/technicalArticles/Security/Signed/index.html`.

[20] Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. In *IEEE Symposium on Security and Privacy*, May 1998.