Jalapa: Securing Java with Local Policies

Massimo Bartoletti

Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari, Italy

Gabriele Costa

Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Italy

Roberto Zunino

Dipartimento di Ingegneria e Scienza dell’Informazione, Università di Trento, Italy

Abstract

We present Jalapa, a tool for securing Java bytecode programs with history-based usage policies. Policies are defined by usage automata, that recognize the forbidden execution histories. Usage automata are expressive enough to allow programmers specify of many real-world usage policies; yet, they are simple enough to permit formal reasoning. Programmers can sandbox untrusted pieces of code with usage policies. The Jalapa tool rewrites the Java bytecode by adding the hooks for the mechanism that enforces the given policies at run-time.

Keywords: Usage control, history-based security, bytecode rewriting

1 Introduction

Security has been a major concern in the design and implementation of Java, starting from its early incarnations. Building upon the “safety pillars” of bytecode verification and secure class loading, new defence mechanisms have been developed over the years.

With the release of the JDK 1.0, a mechanism was provided to run untrusted mobile code into a sandbox with limited computational functionalities. The default sandbox prevented untrusted code from, e.g. accessing the local filesystem, from redefining the security manager (otherwise one could circumvent the sandbox), from connecting to (or accepting a connection from) any URL other than the one the code was downloaded, etc. Although these functionalities were completely customizable, this required to subclass the security manager, making it difficult to separate the functional aspects of programming from the security aspects.

While retaining the basic sandbox model of the JDK 1.0, the JDK 1.1 featured a “black or white” security model, based on digital signatures. Java-enabled browsers could be configured to trust digitally-signed mobile code, provided that the signature was put by a trusted entity. Trusted code were granted full privileges, while untrusted code were run without any privilege.

Starting with the JDK 1.2, a more fine-grained mechanism was devised, based on stack inspection [9]. This provides for associating methods with “protection domains” that reflect their provenance, and for defining a global security policy that grants each protection domain a set of permissions. Code includes local checks

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
that guard access to critical resources. At run-time, an access authorization is granted when all the methods on the call stack have the required permission (a special case is that of privileged calls, that trust the methods below them in the call stack). Being strongly biased towards implementation, this mechanism suffers from some major shortcomings. For instance, since a method removed from the call stack no longer affects security, stack inspection does not offer any protection when trusted code uses objects supplied by untrusted code [8].

Although many security policies are not enforceable by stack inspection, at present Java offers no other facilities to specify and enforce user-defined policies. Therefore, it is common practice to renounce to separating duties between functionality and security, and to implement the needed enforcement mechanism with local checks explicitly inserted into the code by programmers. Since forgetting even a single check might compromise the security of the whole application, programmers have to inspect their code very carefully. This may be cumbersome even for small programs, and it may also lead to unnecessary checking.

History-based security has been repeatedly proposed as a replacement for stack inspection [1,7,11]. Clearly, the ability of checking the whole execution history, instead of the call stack only, places history-based mechanisms a step forward stack inspection, from the expressivity viewpoint. However, since many possible history-based models can be devised, it is crucial to choose one which wisely conciliates the expressive power with the theoretical properties enjoyed. It is also important that the security mechanism can be implemented in a way that makes it transparent to programmers, and with a negligible run-time overhead.

Jalapa advocates local usage policies [3] as a history-based model for securing Java applications. Some remarkable features of Jalapa are that:

• local usage policies are expressive enough to model security requirements of real-world applications. For instance, we used them to specify the typical set of policies of a realistic bulletin board system [10].
• at the same time, usage policies are simple enough to be statically amenable, e.g. they can be model-checked against abstractions of program usages [4].
• local usage policies generalise global policies and local checks. The ability of sandboxing a piece of code by localizing the scope of a policy is particularly relevant, as the current programming methodologies provide for reusing code, and for exploiting services and components, offered by untrusted third parties.
• apart from the localization of sandboxes, enforcing policies is completely transparent to programmers.
• since the enforcement mechanism is based on bytecode rewriting, it does not require a custom Java Virtual Machine.
• even when the program source code is unavailable, Jalapa allows for specifying and enforcing policies on its behavior, by directly modifying the bytecode.

This paper gives an overview of Jalapa. We start by presenting our methodology for securing Java applications through local usage policies, with the help of some examples. Then, we give some insights about the design and the implementation of our tool, and we summarise the artifacts supporting our tool. We conclude by highlighting some of the present and future challenges of Jalapa.
We illustrate our methodology for securing Java programs, as well as some key features of Jalapa, with the help of an example. Suppose you have a simple Web browser whose functionality can be extended with plugins, and with methods for handling connections and cookies. Since plugins can be downloaded from the network, possibly from untrusted sites, we want to control their behaviour, and block their execution at the moment they attempt some malicious action. In particular, we focus here on two confinement policies, that prevent plugins from transmitting data read from the local file system, either directly or by exploiting cookies to implement a covert communication channel (although stronger, these policies imply non-interference). Before formally specifying these policies, we consider a skeletal implementation of the classes Browser and Plugin.

```java
public class Browser {
    private Map<URL, String> cookies;
    public Browser() { cookies = new HashMap<URL, String>(); }
    public void connect(URL url) throws Exception {
        URLConnection uc = url.openConnection();
        out = new BufferedWriter(new OutputStreamWriter(uc.getOutputStream()));
    }
    public void writeCookie(URL u, String msg) {
        cookies.put(u, msg);
    }
    public String readCookie(URL u) {
        return cookies.get(u);
    }
}

public abstract class Plugin implements Runnable {
    Plugin(Browser browser, String name, URL codebase) {
    }
    public void doIt() {
        try {
            // invokes this.run() within the sandbox
            PolicyPool.sandbox("plugin-out", this);
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
}
```

We assume that browser plugins extend the Plugin abstract class, by implementing the method run(). The browser starts a plugin by invoking the method doIt(), which is quite peculiar. Actually, it defines a sandbox, which will enforce the policy plugin-out throughout the run of the plugin. This means that all the security-relevant methods called while executing the method run() will be monitored, and blocked if not conformant to the policy. This policy is specified by the usage automaton plugin-out below on the left, to be discussed in a while.

A usage automaton closely resembles a finite state automaton. The field tagged name just defines the name of the policy. The tag aliases defines a mapping from the signatures of security-relevant methods to events that trigger the transitions of the usage automaton. E.g. in the usage automaton plugin-out above, the event read is fired whenever a new object of the class FileInputStream is created, or a...
cookie is accessed through the method \texttt{readCookie}. Similarly, the event \texttt{write} is fired when the method \texttt{getOutputStream} is invoked on a \texttt{Socket}. The remaining fields describe the logics of the automaton. The tag \texttt{states} is for the set of states, \texttt{start} is for the initial state, and \texttt{final} is for the final state, denoting a policy violation. The tag \texttt{trans} preludes to the transition relation of the automaton. In our example, a transition from \texttt{q0} to \texttt{q1} occurs upon reading any file or cookie. A transition from \texttt{q1} to \texttt{fail} occurs upon opening an output stream on a socket. Since \texttt{fail} is offending, this indeed implements the first confinement policy.

The second policy is specified by the usage automaton \texttt{plugin-cookie}, above on the right, which introduces further peculiar features of Jalapa: parameters and guards. We start from the state \texttt{q0}. The event \texttt{init(p,u)}, signalling the creation of a new plugin \texttt{p} with codebase URL \texttt{u}, causes a transition to \texttt{q1}. Upon a \texttt{start(p)}, i.e. when \texttt{p} is launched by the browser, we reach \texttt{q2}. There, all the accesses to a cookie having a URL different from \texttt{u} lead to the offending state. When the control is transferred to another plugin, we reset the state to \texttt{q1}. At run-time, the policy \texttt{plugin-cookie} is enforced for all the possible instantiations of the formal parameters \texttt{p}, \texttt{u} and \texttt{u'). Since this policy spans over multiple activations of plugins, we enforce it globally throughout the execution of the browser.

Once the needed policies and sandboxes have been defined, the next step is to instrument the compiled program with the hooks from the security-relevant methods to the execution monitor. Our tool implements this step as a bytecode transformation, discussed in more detail below. The resulting bytecode will respect all the usage policies at hand, within their scopes (see [10] for usage details). In [2] we formally prove that the run-time mechanism implemented by Jalapa is sound and complete w.r.t. the specification of policy compliance.

\textbf{The Jalapa bytecode instrumentator.} Our approach to code instrumentation is based on class wrapping, at the bytecode level. Since this solution suffers from some known issues, when moving to a production implementation we plan to follow a bytecode rewriting approach \textit{à la} Kava [12]. First, we detect the set \(\mathcal{M}\) of all the methods involved in policies. We inspect the bytecode, starting from the methods used in the aliases, and then computing a transitive closure through the inheritance graph. We create a \textit{wrapper} for each of these methods. A wrapper \(\mathcal{W}_C\) for the class \texttt{C} declares exactly the same methods of \texttt{C}, implements all the interfaces of \texttt{C}, and extends the superclass of \texttt{C}. Indeed, \(\mathcal{W}_C\) can replace \texttt{C} in any context, in that it admits the same operations of \texttt{C}. A method \texttt{m} of \(\mathcal{W}_C\) can be either monitored or not. If the corresponding method \texttt{m} of \texttt{C} does not belong to \(\mathcal{M}\), then \(\mathcal{W}_C.m\) simply calls \texttt{C.m}. Otherwise, \(\mathcal{W}_C.m\) calls the \texttt{PolicyPool.check} method that controls whether \texttt{C.m} can actually be executed without violating the active policies. A further step substitutes (the references to) the newly created classes for (the references to) the original classes. Finally, the instrumented code is linked to the Jalapa run-time support, i.e. a library that contains the resources necessary to the monitoring process. Note that our instrumentation produces a stand-alone application, requiring no custom JVM and no further external components.

\textbf{The Jalapa runtime environment.} The core of the enforcement mechanism is the method \texttt{PolicyPool.check()} that, for each active policy, tracks the states of all
the needed usage automata. The state of the monitor is a mapping from policies to sets of pairs \( (O_1, \ldots, O_k), Q \), where \( (O_1, \ldots, O_k) \) is a tuple of weak references to the objects that substitute the formal parameters of the usage automaton, and \( Q \) is the current state of the usage automaton. Dummy instantiations are also maintained, to be concretized when new objects are discovered in the execution trace. When an object is garbage-collected, its occurrences in the mechanism state are reverted to dummies. If no usage automaton reaches an offending state, the intercepted method call is forwarded to the actual target; otherwise, a security exception is thrown.

Supporting artifacts. Jalapa is an open-source project. The sources are available through a Subversion repository at SourceForge [10]. Some further supporting material is accessible through the project Web page:

- the Jalapa Tutorial, that provides programmers with a step-by-step guide for securing Java programs with local usage policies.
- a repository of example programs and policies, including a prototype implementation of a secure bulletin board system.
- the manual page of policies, that defines their syntax and semantics.
- the manual page of the Jalapa rewriter, that defines its command-line syntax.

3 Discussion: present and future challenges

The Jalapa project started as an applicative branch of more foundational work on history-based access control [3,4,5]. Porting this theoretical machinery to a concrete setting like Java posed several issues. While our original goals have been achieved to a fair degree by the current release of Jalapa, there is room for future improvements. We devise three main research directions: (1) increasing the expressive power of usage policies, (2) reducing the run-time overhead of the enforcement mechanism, and (3) developing programming tools and methodologies to facilitate writing secure programs with Jalapa.

For the first point, although our usage policies are quite general, they do not cover all the possible real-world scenarios. For instance, we would like to require that a given low-level method (e.g. a write-file) can only be invoked within the scope of some high-level method that securely manages the low-level one. This is the case e.g. of a change-password method that calls the write-file method to update passwords. The challenge is to improve the expressive power of our usage policies, while keeping them clean and formally sound. A promising solution seems that of introducing aliases of the form \( \text{ev} := \text{C1.m1(...) \{ C2.m2(...) \} } \), meaning that the event \( \text{ev} \) is fired whenever the method \( \text{m1} \) of class \( \text{C1} \) is invoked within the scope of the method \( \text{m2} \) of class \( \text{C2} \). Another improvement would be to allow our policies to mention the values returned by methods. This can be done by generating “return” events, exposing these values.

For the second point, we are currently developing a static analyser for Java bytecode, to detect those policies that are always respected in all the possible executions of the application. The run-time enforcement can then be optimized, by discarding the wrappers, and the associated execution monitoring, for the methods involved in
policies that are always respected. This static analysis can be split in two phases:

- in the first phase, we extract from the bytecode a control flow graph, and we transform it into a history expression [4]. This is a sort of context-free grammar, the language of which over-approximates all the possible traces of events that the analysed program can generate at run-time.

- in the second phase, we reduce the infinite-state system given by the history expression to an equivalent finite one, and check it against the usage policies mentioned by the sandboxes used in the program. This is done through a model-checker. Only the policies that do not pass model checking need to be enforced at run-time. This phase has been implemented by our LocUsT tool, which runs in polynomial time in the size of the extracted history expression.

Further details about this phase can be found in [4,6].

For the third point, we are developing an Eclipse plugin that combines the previous items into a programming environment, with facilities for writing policies, sandboxing code, and for running the static analyses to discover which policies can be disregarded by the security monitor. The LocUsT model checker, a prototype of this analysis phase, and a prototype of the Eclipse plugin are distributed along with the Jalapa sources through the SourceForge Subversion repository.

Acknowledgments. Research partially supported by EU-FETPI Project IST-2005-16004 SENSORIA (Software Engineering for Service-Oriented Overlay Computers), by PRIN Project SOFT (Tecniche Formali Orientate alla Sicurezza), and by EU project IST-033817 GridTrust (Trust and Security for Next Generation Grids).

References


A Outline of the demo presentation

To present the overall methodology, we start with a toy example, and then we move towards more realistic ones.

A.1 A simple usage policy

Suppose we are given an untrusted application. We want to run that, and allow access to the filesystem. However, we have some security concerns: the application might silently copy data from a confidential file of ours to a public file. More concretely, we want to forbid malicious code as this:

```java
FileInputStream is = new FileInputStream(new File("confidential"));
FileOutputStream os = new FileOutputStream(new File("public"));
int n = is.read(b);
os.write(n);
```

Note that the Java stack inspection mechanism offers no effective protection against this kind of attacks, because after the method `read` has returned, there is no trace of it left on the call stack. We can however prevent that attack using Jalapa, since its security model allows for the specification and enforcement of history-based policies. The user of Jalapa, who wants to run the untrusted application, can define a policy forbidding the attack. A simple way to perform that would be to require that a file write can not be performed a file read. This policy can be formalised as the following usage automaton:

```java
name: chinese-wall
aliases:
read := (java.io.FileInputStream).read()
read := ... // other read methods here
write := (java.io.FileOutputStream).write(int n)
write := ... // other write methods here
states: q0 q1 fail
start: q0
final: fail
trans:
q0 -- read --> q1
q1 -- write --> fail
```

A usage automaton closely resembles a finite state automaton, composed of six fields. The field tagged name just defines the name of the policy. The tag aliases introduces some convenient shorthands for the methods mentioned in the policy. In the usage automaton `chinese-wall` above, the event `read` is fired whenever the method `read()` of the class `FileInputStream` is invoked. Similarly, the event `write` is fired when the method of class `FileOutputStream` with signature `write(int n)` is invoked. In our policy, the parameter `n` of type `int`, is immaterial, so our aliases will neglect them.

The remaining fields describe the logics of the automaton: the tag `states` is for the set of states, `start` is for the initial state, and `final` is for the final state, that denotes a policy violation. The tag `trans` precedes the transition relation of the automaton.

We can then run the application within Jalapa, asking that the `chinese-wall` policy is enforced in the whole execution. This is done as follows. Suppose that the `main` method is contained in `jalapa/app/App.class`. First we invoke the bytecode
rewriter (jisel.Jisel), and generate a secured application in app-sec. Then, we run it specifying the policy to enforce.

\%
\% cd jalapa
\% java -Dpolicies=chinese-wall -cp :../bcel/bcel-5.2.jar
\% jisel.Jisel app -o app-sec -m App
\%
\% cd app-sec
\% java -Dcheck.global=chinese-wall -cp ../../jisel.jar App

As we expected, the file write that violates the policy is stopped. Instead, a SecurityPolicyException is raised.

Reading...

Exception in thread "main" jisel.policy.SecurityPolicyException:
Event (_:java.io.FileOutputStream).write(int) Not allowed!
    at jisel.java.io.FileOutputStream_Jisel.write(FileOutputStream_Jisel.java)
    at WriteFile.main(WriteFile.java)

A.2 Usage policies with parameters

The chinese-wall policy correctly prevents the above-mentioned attack, but it also forbids many legitimate applications to run. Indeed, reading from any file will cause any write to trigger a policy violation. Most likely, the user of Jalapa only wants to watch out for reads from the “confidential” files. Unfortunately, checking this is made hard by the Java APIs. When is.read() is invoked, the name of the file is not provided as an argument, so we can not check for that at this time. We need to inspect the history, and look for the creation of the object is, that is to the constructor invocation is.<init>(File fr). Even here, the file name is not present: we have to inspect the history again, and look for the creation of the file object f, that is f.<init>(String fileName). Here, we finally find the relevant piece of information that allows us to decide whether this is a read from the “confidential” file. Jalapa can model this through a parametric policy:

name: chinese-wall2
aliases:
initF(f,name) := (f:java.io.File).<init>(String name)
initIS(is,f) := (is:java.io.FileInputStream).<init>(File f)
read(is) := (is:java.io.FileInputStream).read() // other read methods here
write := (java.io.FileOutputStream).write(int n) // other write methods here
states: q0 q1 q2 q3 fail
start: q0
final: fail
trans:
q0 -- initF(f,"confidential") --> q1
q1 -- initIS(is,f) --> q2
q2 -- read(is) --> q3
q3 -- write --> fail

Note that in the transitions we pass arguments to our aliases. Also, we use f and is to represent arbitrary objects. Indeed, f and is parametrize the whole chinese-wall2 policy. When the policy is enforced, Jalapa checks the run-time execution against all the possible instantiations of the policy parameters.

Using the refined policy, more legitimate applications can be run without triggering violations. That is, many false positives are corrected. Here is Jalapa using a legitimate read/write application, using the chinese-wall2 policy. Assume the -sec directories contain the rewritten bytecode.
And here is Jalapa running the malicious application, while enforcing the same policy:

```
% cd bad-app-sec
% java -Dcheck.global=chinese-wall2 -cp ../../../jisel.jar App
Reading...
Exception in thread "main" jisel.policy.SecurityPolicyException:
Event (_.java.io.FileOutputStream).write(int) Not allowed!
at jisel.java.io.FileOutputStream_Jisel.write(FileOutputStream_Jisel.java)
at.writeFile.main(WriteFile.java)
```

### A.3 Local Policies

Jalapa policies do not need to be enforced in a global fashion. Jalapa allows to sandbox single blocks of untrusted code. For instance, a browser running an untrusted plugin may want to use the following:

```java
import jisel.policy.PolicyPool;
// ...
PolicyPool.sandbox("chinese-wall2", new Runnable() {
    public void run() {
        // ... invoke plugins here ...
    }
}
```

### A.4 Guards

In policy `chinese-wall2`, we focus on a single “confidential” file, and watch out for reads from it. In a sense, this is a “default: allow”, or blacklist policy. In some scenarios, a whitelist policy would be more realistic. That is, we want to regard any file as a confidential one, but for some “public” ones. This can be modelled in Jalapa by exploiting **guards**.

**Policy** `chinese-wall3` has three parameters, namely `f`, `is`, and `name`. The first transition mentioned in the policy can only be taken when `name` is not `public1`. The same holds for the second transition and `public2`. We run our application again using Jalapa, and check that indeed there are no violations when only these files are read, and then a write is performed.

Jalapa also allows guards of the form `x != y` where both `x` and `y` are parameters. In Sect. 2, we defined a **plugin-cookies** policy stating that a plugin invoked by
the browser for a web page, the plugin can only access the cookies of the URL of that page. We can demonstrate that example within Jalapa.

A.5 Wildcards

Assume we have the following class for a classified file:

```java
public class ClassifiedFile {
    // ...
    public void authorize(Officer o) { // ...
        // ...
    }
    public void disclose() { // ...
        // ...
    }
}
```

The `authorize` method is invoked whenever an officer consents to declassify the file. Assume that two distinct officers must agree on declassification before the actual disclosure can be done. A naive way to express this policy would be:

```plaintext
name: naive-double-agreement
aliases:
    auth(o,f) := (f:ClassifiedFile).authorize(Officer o)
    disclose(f) := (f:ClassifiedFile).disclose()
states: q0 q1 ok fail
start: q0
final: fail
trans:
q0 -- auth(o1,f) --> q1
q1 -- auth(o2,f) --> ok when o1 != o2
q0 -- disclose(f) --> fail
q1 -- disclose(f) --> fail
```

Intuitively, when two officers authorize the disclosure the non-offending sink state `ok` is reached. If a disclosure happens before of that, the last two transitions are followed, causing a violation.

Unfortunately, the policy above does not actually work as intended. This is because the policy is checked under all the possible instantiations of its parameters `f`, `o1`, and `o2`. This universal quantification is indeed not the wanted one. Suppose Alice, Bob, and Charlie are officers. Alice and Bob authorize the declassification. However, the policy above is violated when its parameters are instantiated e.g. as follows: `o1` to Charlie, `o2` to Bob. The automaton in this case never reaches `q1` or `ok`, and will reach `fail` as soon as the disclosure happens.

To overcome this limitation, we allow a wildcard - to be used in Jalapa policies. Its meaning is "any object different from any other object mentioned in the policy". Roughly, it allows for a form of existential quantification. The policy can be now correctly stated as follows.

```plaintext
name: double-agreement
aliases:
    auth(o,f) := (f:ClassifiedFile).authorize(Officer o)
    disclose(f) := (f:ClassifiedFile).disclose()
states: q0 q1 ok fail
start: q0
final: fail
trans:
q0 -- auth(o,f) --> q1
q0 -- auth(-,f) --> ok
q1 -- auth(-,f) --> ok
q0 -- disclose(f) --> fail
```

If a disclosure happens at state q0, the policy is violated. To move from q0, an auth event is required. When the first auth(x,f) happens, then the instantiation of the policy having o equal to x reaches state q1. All the other instantiations instead reach state ok, so they effectively stop monitoring the application. Practically speaking, after the first auth(x,f), only the instantiation having o=x survives, at state q1. From here, if the disclose happen before the second authorization, we trigger a failure. If instead another officer authorizes it, we allow disclosure by moving to the ok state.

Suppose we now wish to constrain the authorizations made by officers. For instance, at any given time, only a subset of the classified files can actually be authorized. Authorizations of declassification can be suspended and resumed for a file. For instance, a Chief Officer can suspend them if the classified file is to be promoted to “top-secret”.

```java
public class ClassifiedFile {
    // ...
    public void suspend(Officer officer) {
        if (securityOfficers.contains(officer)) canAuth = false;
    }
    public void resume(Officer officer) {
        if (securityOfficers.contains(officer)) canAuth = true;
    }
    // ...
}
```

The suspended-auth policy below checks that files receive no declassification authorizations when they should not.

```plaintext
name: suspend-auth
aliases:
  auth(f,o) := (f:ClassifiedFile).authorize(Officer o)
  suspend(f,o) := (f:ClassifiedFile).suspend(Officer o)
  resume(f,o) := (f:ClassifiedFile).resume(Officer o)
states: q0 q1 fail
start: q0
final: fail
trans:
  q0 -- suspend(f,*) --> q1
  q1 -- resume(f,*) --> q0
  q1 -- auth(f,*) --> fail
```

Above we use the wildcard * in our transitions. Its meaning is simply “any object”. That is, we disregard the identity of the officer who is suspending/resuming authorizations, since it is irrelevant to this policy.

### A.6 A Larger Example

We consider the implementation of a list datatype which allows for iterating over the elements of the list. In this scenario, a relevant safety policy is that preventing list modifications when the list is being iterated (indeed, Java collections implement this behaviour through local checks).

We first show an unsafe implementation of the class ListIter (see ListIter.java), and then we shall show a usage automaton which can be used to enforce the above-mentioned policy.
public class ListIter {
    private Object[] v; // underlying array
    // ...
    public void add(Object o) { /* ... */ }
    public void remove(Object o) { /* ... */ }
    public void startIterator() { iter = 0; }
    public Object next() throws Exception { /* ... */ }
    public boolean hasNext() { return (iter<last); }

    public static void main(String[] args) throws Exception {
        ListIter l0 = new ListIter(), l1 = new ListIter();
        String[] v0 = {"a", "b", "c"}, v1 = {"d", "e", "f"};
        for (int i=0; i<v0.length; i++) { l0.add(v0[i]); }
        for (int i=0; i<v1.length; i++) { l1.add(v1[i]); }
        l0.startIterator();
        while (l0.hasNext()) {
            Object o0 = l0.next();
            l1.startIterator();
            boolean found = false;
            while (l1.hasNext()) {
                Object o1 = l1.next();
                if (o0==o1) found=true;
            }
            if (!found) { l1.add(o0); l0.remove(o0); }
        }
        System.out.println("l0 = " + l0);
        System.out.println("l1 = " + l1);
    }
}

The method main defines two lists l0 and l1, and joins them into l1, while removing the duplicates. In our faulty implementation, the resulting list l1 is [d;e;f;a;c], instead of the expected [d;e;f;a;b;c], because modifying the lists while still traversing it confuses the iterator, which is using an index inside v.

To prevent from this erroneous behaviour, we now define the usage automaton safe-iterator, that will be used to abort a computation when the method next() is invoked on a given list l after l has been modified.

```
name: safe-iterator
aliases:
start(l) := (l:ListIter).startIterator()
next(l) := (l:ListIter).next()
modify(l) := (l:ListIter).add(Object o)
modify(l) := (l:ListIter).remove(Object o)
states: q0 q1 fail
start: q0
final: fail
trans:
q0 -- modify(l) --> q1
q1 -- next(l) --> fail
q1 -- start(l) --> q0
```

The usage automaton has three states: q0, q1, and fail. In the state q0 both modifications of the list and iterations are possible. However, a modification (i.e. an add or a remove) done in the state q0 will lead to q1, where it is no longer possible to iterate on l, until a start(l) resets the situation.

Jalapa correctly spots the violation and raises an exception before the lists can get corrupted.