JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2009
Vol. 8, No. 4, June 2009

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
Pierpaolo Degano Dipartimento di Informatica, Università di Pisa, Italy
Fabio Martinelli Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Italy
Roberto Zunino Dipartimento di Ingegneria e Scienza dell'Informazione, Università degli Studi di Trento, Italy

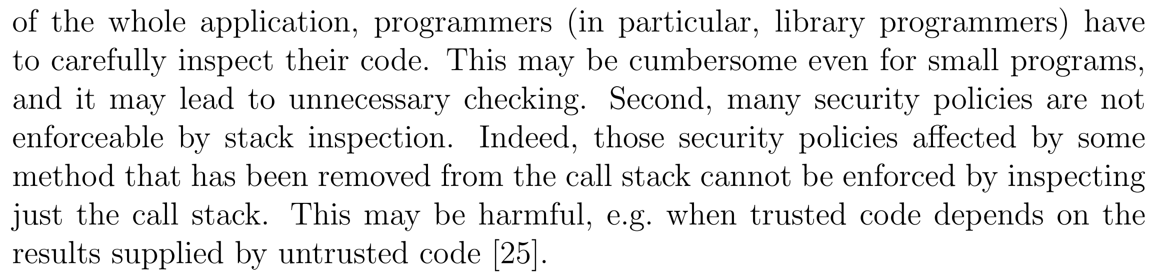
We propose an extension to the security model of Java, that allows for specifying, analysing and enforcing history-based usage policies. Policies are defined by usage automata, that recognize the forbidden execution histories. Programmers can sandbox an untrusted piece of code with a policy, which is enforced at run-time through its local scope. A static analysis allows for optimizing the execution monitor: only the policies not guaranteed to be always obeyed will be enforced at run-time.

1 INTRODUCTION

A fundamental concern of security is to ensure that resources are used correctly. Devising expressive, flexible and efficient mechanisms to control resource usage is therefore a major issue in the design and implementation of security-aware programming languages. The problem is made even more crucial by the current programming trends, which allow for reusing code, and exploiting services and components, offered by (possibly untrusted) third parties. It is common practice to pick from the Web some scripts, or plugins, or packages, and assemble them into a bigger program, with little or no control about the security of the whole.

Stack inspection, the mechanism adopted by Java [26] and the .NET CLR [37], offers a pragmatic setting for access control. Roughly, each frame in the call stack represents a method; methods are associated with "protection domains", that reflect their provenance; a global security policy grants each protection domain a set of permissions. Code, tipically within libraries, includes local checks that guard access to critical resources. At run-time, an access authorization is granted when *all* the frames 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. First, there is no automatic mechanism to decide where checks have to be inserted. Since forgetting even a single check might compromise the safety

Cite this article as follows: Massimo Bartoletti, Gabriele Costa, Pierpaolo Degano, Fabio Martinelli, and Roberto Zunino: "Securing Java with Local Policies", in Journal of Object Technology, vol. 8, no. 4, Special Issue: Workhop FTfJP and IWACO at ECOOP 08, June 2009, pp. 5-32 http://www.jot.fm/issues/issue_2009_06/article1/



of the whole application, programmers (in particular, library programmers) have to carefully inspect their code. This may be cumbersome even for small programs, and it may lead to unnecessary checking. Second, many security policies are not enforceable by stack inspection. Indeed, those security policies affected by some method that has been removed from the call stack cannot be enforced by inspecting just the call stack. This may be harmful, e.g. when trusted code depends on the results supplied by untrusted code [25].

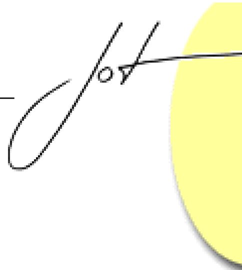
History-based access control generalizes stack inspection, as the run-time monitor can check (a suitable abstraction of) the *whole* execution. History-based policies and mechanisms have been studied at both levels of foundations [3, 24, 42] and of language design and implementation [1, 21]. A common drawback of these approaches is that a policy must be respected through the whole execution. This may involve guarding each resource access, and ad-hoc optimizations are then in order to recover efficiency, e.g. compiling the global policy to local checks [18, 33]. Also, a large monolithic policy may be hard to understand, and not very flexible either.

Local policies [7] generalise both history-based global policies and local checks spread over program code. They exploit a scoping mechanism to allow the programmer to "sandbox" arbitrary program fragments. Local policies smoothly allow for safe composition of programs with their own security requirements, and they can drive call-by-contract composition of services [9]. In mobile code scenarios, local policies can be exploited, e.g. to model the interplay among clients, untrusted applets and policy providers: before running an untrusted applet, the client asks the trusted provider for a suitable policy, which will be locally enforced by the client throughout the applet execution.

In this paper, we outline the design and the implementation of an extension to the security model of Java, which features history-based local usage policies. In the spirit of JML [32], policies are orthogonal to Java code. They are defined through our *usage automata*, a variant of finite state automata (FSA), where the input alphabet comprises the security-relevant events, parameterized over objects. So, policies can express any regular property on execution histories. Our policies are local, in the sense that programmers can define their scope through a "sandbox" construct.

The first contributions of this paper are a language for specifying usage policies, and a run-time mechanism for enforcing them on Java programs. Some remarkable features of our technique are that:

- our usage policies are expressive enough to model interesting security requirements, also taken from real-world applications. For instance, in Sec. 3 we use them to specify the typical set of policies of a realistic bulletin board system. At the same time, usage policies are statically amenable, i.e. model-checking usage policies against given abstractions of program usages (named *history expressions*) is decidable in polynomial time.
- apart from the localization of sandboxes, enforcing policies is completely transparent to programmers. Moreover, since the enforcement mechanism is based



on bytecode rewriting, it does not require a custom Java Virtual Machine.

- even in the case when the program source code is unavailable, we still allow for specifying and enforcing policies on its behavior.

The second contribution of this paper is an optimization of the run-time enforcement mechanism. This is based on a static analysis that detects the policies violated by a program in some of its executions [10]. The analysis is performed in two phases. The first phase over-approximates the patterns of resource usages in a program. The second phase consists in model-checking the approximation of a program against the policies on demand. Summing up, we optimise the run-time security mechanism, by discarding the policies guaranteed to never fail, and by checking just the events that may lead to a violation of the other policies.

The third contribution is an open-source implementation of the above-mentioned techniques as an Eclipse plugin, within the Jalapa Project [31]. We provide programmers with a development environment that offers facilities to write the usage policies, use them to sandbox Java code, and compile it with the needed hooks to the security monitor. Also, through the plugin it is possible to run the static analysis, to discover which policies can be disregarded by the security monitor.

An example. Consider a trusted component `NaiveBackup` that offers static methods for backing up and recovering files. Assume that the file resource can be accessed through the interface below. The constructor takes as parameters the name of the file and the directory where it is located. A new file is created when no file with the given `name` exists in the given `dir`. The other methods are as expected.

```
public File(String name, String dir);
public String read();
public void write(String text);
public String getName();
public String getDir();
```

In the class `NaiveBackup`, the method `backup(src)` copies the file `src` into a file with the same name, located in the directory `/bkp`. The method `recover(dst)` copies the backed up data to the file `dst`. As a naïve attempt to optimise the access to backup files, the last backed up file is kept open.

```
class NaiveBackup {
  static File last;
  public static backup(File src) {
    if(src.getName() != last.getName())
      last = new File(src.getName(), "/bkp");
    last.write(src.read());
  }
  public static recover(File dst) {
    if(dst.getName() != last.getName())
      last = new File(dst.getName(), "/bkp");
    dst.write(last.read());
  }
}
```

Consider now a malicious `Plugin` class, trying to spoof `NaiveBackup` so to obtain a copy of a secret passwords file. The method `m()` of `Plugin` first creates a file called `passwd` in the directory `"/tmp"`, and then uses `NaiveBackup` to recover the content of the backed up password file (i.e. `/bkp/passwd`).

```
class Plugin {
  public void m() {
    File g = new File("passwd","/tmp");
    NaiveBackup.recover(g);
  }
}
```

To prevent from this kind of attacks, the `Plugin` is run inside a sandbox, that enforces the following policy. The sandboxed code can only read/write files it has created; moreover, it can only create files in the directory `"/tmp"`. This policy is specified by the usage automaton `file-confine` in Fig. 1 (left). This resembles a finite state automaton, but its edges carry variables (`f` and `d`) and guards (`when d!="/tmp"`). The policy is parametric over all such variables. The policy is violated whenever some instantiation of these variables to actual objects drives the automaton to an offending state. The tag `aliases` introduces some convenient shorthands for the security-relevant methods mentioned in the policy. For instance, `new(f,d)` abstracts from invoking the constructor of the class `File` on the target object `f`.

The remaining part describes the automaton: the tag `states` is for the set of states, `start` is for the initial state, and `final` is for the final, offending state. The tag `trans` preludes to the transition relation of the automaton. The edge from `q0` to `q1` represents creating a file `f` in the directory `"/tmp"` (the name of the file is immaterial). The edge from `q0` to `fail` labelled `read(f)` prohibits reading the file `f` if no `new(f,d)` has occurred beforehand. Similarly for the edge labelled `write(f)`. The edge from `q0` to `fail` labelled `new(f,d) when d!="/tmp"` prohibits creating a file in any directory `d` different from `"/tmp"`. In Fig. 1 (right), the policy is depicted in the usual notation of finite state automata. For instance,

```
name: file-confine
aliases:
new(f,d) := (f:File).<init>(String n,String d)
read(f) := (f:File).read()
write(f) := (f:File).write(String t)
states: q0 q1 fail
start: q0
final: fail
trans:
q0 -- new(f,"/tmp") --> q1
q0 -- new(f,d) --> fail when d!="/tmp"
q0 -- read(f) --> fail
q0 -- write(f) --> fail
```

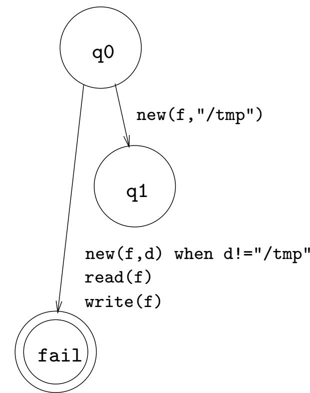


Figure 1: Usage automaton `file-confine` for the file confinement policy.

the trace `new(f,"/tmp") write(f)` respects the policy (the offending state cannot be reached), while both `new(f,"/home")` and `read(passwd)` violate it. The trace `new(f1,"/tmp") new(f2,"/etc")` violates the policy, too: while instantiating the policy with `f=f1` drives the automaton to the safe state `q1`, the instantiation `f=f2` causes the offending state `fail` to be reached.

```
class Main {
  public static void main() {
    File f = new File("passwd","/etc");
    NaiveBackup.backup(f);
    PolicyPool.sandbox("file-confine", new Runnable() {
      public void run() {
        new Plugin().m();
      }});
  }
}
```

The class `Main` first backs up the passwords file through the `NaiveBackup`. Then, it runs the untrusted `Plugin` inside a sandbox enforcing the policy `file-confine`. The `Plugin` will be authorized to create the file `"/tmp/passwd"`, yet the sandbox will block it while attempting to open the file `"/bkp/passwd"` through the method `NaiveBackup.recover()`. Indeed, `Plugin` is attempting to read a file it has not created, which is prohibited by `file-confine`. Any attempt to directly open/overwrite the password file will fail, because the policy only allows for opening files in the directory `"/tmp"`. Note that our sandboxing mechanism enables us to enforce security policies on the untrusted `Plugin`, without intervening in its code.

2 SECURING JAVA

We now introduce our extension to the security model of Java. We are interested in specifying and enforcing safety policies of *method traces*, i.e. the sequences of run-time method calls. We define policies through *usage automata*, an automata-based policy language featuring facilities to deal with method parameters. We found this model suitable for expressing a wide variety of usage policies, also taken from real-world scenarios. In the setting of usage control, finite-state policies are typically enough; using more powerful formalisms, e.g. pushdown automata, would add a marginal practical gain, at the cost of more complex enforcement mechanism and analysis. The scope of policies needs not be global to the whole program; we use *sandboxes* to localize the scope of a policy to a given program fragment. We conclude by specifying an enforcement mechanism for our policies. In the next section, we will show how we actually implemented this mechanism to secure Java programs.

Aliases and events

As mentioned above, our policies will constrain the sequence of run-time method calls. Clearly, any policy language aiming at that will need some facilities to abstract from the (possibly infinite) actual parameters occurring in program executions. To do that, one could use method signatures as a basic building block to specify policies. However, this may lead to unnecessarily verbose specifications; so, we provided our policy language with a further indirection level, which helps in keeping simple the writing of policies.

We call *aliases* our abstractions of the security-relevant methods. Let $\mathtt{m}$ be a method of class $\mathtt{C}$, with signature $(\mathtt{y} : \mathtt{C}).\mathtt{m}(\mathtt{C}_1\ \mathtt{y}_1, \ldots, \mathtt{C}_n\ \mathtt{y}_n)$, where $\mathtt{y}$ is the target object. An alias $\mathtt{ev}(\mathtt{x}_1, \ldots, \mathtt{x}_k)$ for $\mathtt{m}$ is defined as:

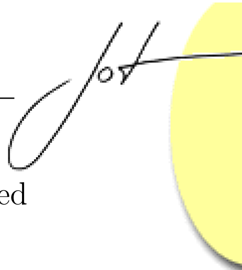
$$\mathtt{ev}(\mathtt{x}_1, \ldots, \mathtt{x}_k) := (\mathtt{y} : \mathtt{C}).\mathtt{m}(\mathtt{C}_1\ \mathtt{y}_1, \ldots, \mathtt{C}_n\ \mathtt{y}_n)$$

where $\{\mathtt{x}_1, \ldots, \mathtt{x}_k\} \subseteq \{\mathtt{y}, \mathtt{y}_1, \ldots, \mathtt{y}_n\}$ (note that the set inclusion may be strict, when some parameters in the method signature are irrelevant for the policy of interest).

For instance, recall the aliases introduced in Fig. 1:

```
new(f,d) := (f:File).<init>(String name, String d)
read(f)  := (f:File).read()
write(f) := (f:File).write(String t)
```

The first item means that `new(f,d)` is an alias for the constructor of the class `File` with parameters `name` and `d`, both of type `String`. In the other items, `read(f)` (resp. `write(f)`) is an alias for the method `read()` (resp. `write(String t)`) of the same class. The parameter `f` is the target object, in this case of class `File`. Note that the parameter `t`, irrelevant for our policy, is not mentioned in the aliases. Note also that aliasing is not an injective function from method signatures to aliases. For



instance, if multiple methods were provided to write a file, they could be abstracted to a single alias `write(f)`, so simplifing the definition of the usage automaton.

As a method call is the concrete counterpart of a method signature (the formal parameters in the latter are concretised into actual parameters in the former), an *event* is the concrete counterpart of an alias. Summing up, we have a mapping from method calls to events, and – by lifting this to sequences – a mapping from method traces to event traces. For instance, the method trace:

```
g.<init>("passwd","/etc") f.read() g.write("secret")
```

is abstracted into the event trace:

```
new(g,"/etc") read(f) write(g)
```

Usage automata

We define safety policies on the method traces through *usage automata*. They are an extension of finite state automata, where the labels on the edges may have variables and guards. Variables represent universally quantified resources, while guards express conditions among resources. A usage automaton is specified in plain text as follows:

`name:`	*name of the usage automaton*
`aliases:`	*set of aliases*
`states:`	*set of states*
`start:`	*initial state*
`final:`	*final (offending) states*
`trans:`	*set of labelled edges*

The first item is just a string, to be used when referencing the usage automaton (e.g. in the definition of sandboxes, see below). The second item lists the aliases (separated by newlines) relevant for the policy. The remaining four items define the operative part of the usage automaton, i.e. its finite set of states (separated by spaces), the initial and the final states, and the set of edges. The formal parameters of a usage automaton are the variables occurring in its edges.

To define an edge from state `q` to state `q’` we use the following syntax:

`q --` *label* `--> q’ when` *guard*

The *label* is a (possibly, partial) concretisation of an alias defined in the `aliases` item. Concretising an alias $\mathtt{ev}(\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{k})$ results in a label of the form $\mathtt{ev}(\mathtt{Z}_1, \ldots, \mathtt{Z}_\mathtt{k})$, where each $\mathtt{Z}_\mathtt{i}$ can be either:

- S, for a static object S. Static objects comprise strings (e.g. "/tmp"), final static fields (e.g. User.guest), and values of enum types.
- x, for some variable x. This means that the parameter at position i is universally quantified over any object.
- $*$, a wildcard that stands for any object.

A *guard* represents a condition among resources, defined by the following syntax:

guard ::= true | Y!=Z | *guard* and *guard*

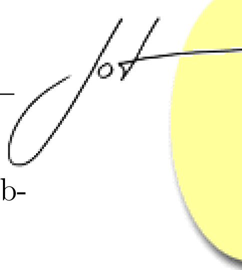
where Y and Z are either static objects or variables. The guard when true can be omitted. For the sake of minimality, we introduced only two logical operators, in Section 6 we discuss some possible extensions to the language of guards.

Note that policies can only control methods known at static time. In the case of dynamically loaded code, where methods are only discovered at run-time, it is still possible to specify and enforce interesting classes of policies. For instance, system resources – which are accessed through the JVM libraries only – can always be protected by policies.

To give semantics to usage automata, one can proceed in two equivalent ways. The first consists in reconducting them to finite state automata, by instantiating the formal parameters (i.e. the variables occurring in their edges) to actual parameters. The second way consists in exploiting them to define a run-time monitor that checks all the method invocations occurring in a program execution, and aborts it when an attempted policy violation is detected. In [5] this two approaches are shown equivalent in an abstract framework.

Here, we focus on the run-time monitor approach, detailed in the next subsection in the concrete framework of Java, and we only give below the intuition on when a trace complies with the policy defined by a usage automaton. As said above, we reconduct a usage automaton to a (finite) set of finite state automata (FSA), by instantiating the formal parameters into actual objects. A trace complies with a policy if it is *not* recognized by any of the resulting FSA. Actually, the final states in usage automata represent policy violations: a trace leading to a final state suffices to produce a violation. An alternative approach would be the "default-deny" one, that allows for specifying the permitted usage patterns, instead of the denied ones. We could deal with this approach be regarding the final states as accepting. Both the default-deny and the default-allow approaches have known advantages and drawbacks. Below we will give a more formal account, by specifying our enforcement mechanism.

The instantiation procedure roughly amounts to: (i) binding each formal parameter to each object (of compatible type) encountered in the method trace, (ii) removing the edges not respecting the conditions in the guards, and (iii) adding self-loops for all the events not explicitly mentioned in the usage automaton.



Back to our running example, consider the following event traces (easily abstracted from the actual method traces):

$$\begin{array}{lll}\eta_0 & = & \texttt{new}(\texttt{f}_0, \texttt{"/tmp"})\ \texttt{read}(\texttt{f}_1) \\ \eta_1 & = & \texttt{new}(\texttt{f}_0, \texttt{"/tmp"})\ \texttt{read}(\texttt{f}_0) \\ \eta_2 & = & \texttt{new}(\texttt{f}_0, \texttt{"/tmp"})\ \texttt{read}(\texttt{f}_0)\ \texttt{new}(\texttt{f}_1, \texttt{"/etc"})\end{array}$$

The trace η_0 violates the policy `file-confine`, because it drives to the offending state `fail` the FSA obtained by binding the parameters `f` and `d` to $\texttt{f}_1$ and `"/tmp"`. The trace η_1 respects the policy, because the `read` event is performed on a newly created file $\texttt{f}_0$ in the directory `"/tmp"` (recall that the instantiated FSA will have a self-loop labelled $\texttt{read}(\texttt{f}_0)$ on q_1). Instead, η_2 violates the policy, because it drives to the state `fail` the FSA obtained by binding the parameters `f` and `d` to $\texttt{f}_1$ and `"/etc"`. Indeed, since the guard is satisfied, the instantiated FSA has an edge labelled $\texttt{new}(\texttt{f}_1, \texttt{"/etc"})$ from q_0 to `fail`.

Enforcement mechanism

We now introduce an enforcement mechanism for usage automata, that is suitable for run-time monitoring. The configurations of our mechanism have the form:

$$\mathcal{Q} = \{\sigma_0 \mapsto Q_0, \ldots, \sigma_k \mapsto Q_k\}$$

where, for each $i \in 0..k$, σ_i is a mapping from the parameters of U to actual objects, while Q_i contains states of U. Intuitively, each σ_i singles out a possible instantiation of U into a finite state automaton $A_U(\sigma_i)$, while Q_i represents the states reachable by $A_U(\sigma_i)$ upon the method trace seen so far.

The configuration $\mathcal{Q}$ of the mechanism is updated whenever a method is invoked at run-time. The formal specification is in Fig. 2, where with $\mathsf{R}(\mathcal{Q})$ we denote the set of objects occurring in $\mathcal{Q}$. The number of instantiations recorded by the mechanism grows as new objects are discovered at run-time. Note the use of the distinguished objects $\#_1, \ldots, \#_p$, that represent the objects before they actually appear in the trace.

As an example, consider again the policy `file-confine` of Fig. 1, and the event trace $\eta_0 = \texttt{new}(\texttt{f}_0, \texttt{"/tmp"})\ \texttt{read}(\texttt{f}_1)$. Upon the first event, the state of the monitor is augmented with the following mapping (among the others, omitted for brevity):

$$\{\texttt{f} \mapsto \texttt{f}_0, \texttt{d} \mapsto \texttt{"/tmp"}\} \mapsto \{q_1\}$$

When the event $\texttt{read}(\texttt{f}_1)$ is fired, the state is updated with the following mapping:

$$\{\texttt{f} \mapsto \texttt{f}_1, \texttt{d} \mapsto \#_1\} \mapsto \{\texttt{fail}\}$$

Since the offending state `fail` has been reached, a policy violation is triggered.

INPUT: a usage automaton U and a method trace η.
OUTPUT: *true* if η complies with the policy defined by U, *false* otherwise.

1. $\mathcal{Q} := \{\, \sigma \mapsto \{q_0\} \mid \forall x : \sigma(x) \in \{\#_1, \ldots, \#_p\} \,\}$, where q_0 is the initial state of U, $\#_1, \ldots, \#_p$ are distinguished objects, p is the number of parameters of U

2. while $\eta = \mathtt{o}.\mathtt{m}(\mathtt{o}_1, ..., \mathtt{o}_n)\ \eta'$ is not empty, do:
   (a) let $\mathtt{ev}(\mathtt{o}'_1, \ldots, \mathtt{o}'_k)$ be the event abstracting from $\mathtt{o}.\mathtt{m}(\mathtt{o}_1, \ldots, \mathtt{o}_n)$
   (b) for all $\mathtt{i}$ such that $\mathtt{o}'_\mathtt{i} \notin \mathsf{R}(\mathcal{Q})$, extend $\mathcal{Q}$ as follows. For all σ occurring in $\mathcal{Q}$ and for all mappings σ' from the parameters of U to $\mathsf{R}(\mathcal{Q}) \cup \{\mathtt{o}'_1, \ldots, \mathtt{o}'_k\} \cup \{\#_1, \ldots, \#_p\}$ such that, for all x, either $\sigma'(x) = \sigma(x)$ or $\sigma(x) = \#_j$:
   $$\mathcal{Q} := \mathcal{Q}\,[\sigma' \mapsto \mathcal{Q}(\sigma)]$$
   (c) let $\mathit{step}(q)$ be the set of states q' such that there exists an edge from q to q' with label $\mathtt{ev}(\mathtt{x}_1, ..., \mathtt{x}_k)$ and guard g, where $\mathtt{ev}(\mathtt{x}_1, ..., \mathtt{x}_k)\sigma_i = \mathtt{ev}(\mathtt{o}'_1, ..., \mathtt{o}'_k)$ and $g\sigma_i$ is true. Let $\mathit{step}'(q) = \mathit{if}\ \mathit{step}(q) = \emptyset\ \mathit{then}\ \{q\}\ \mathit{else}\ \mathit{step}(q)$. Then, for all $(\sigma_i \mapsto Q_i) \in \mathcal{Q}$, update $\mathcal{Q}$ as:
   $$\mathcal{Q} := \mathcal{Q}\,[\sigma_i \mapsto \textstyle\bigcup_{q \in Q_i} \mathit{step}'(q)]$$

3. if Q_i contains no final state for all $(\sigma_i \mapsto Q_i) \in \mathcal{Q}$, then return *true* else *false*.

Figure 2: Enforcement mechanism for policies defined through usage automata.

Some optimization of the algorithm, omitted for brevity in Fig. 2, are possible. For instance, there is no need to allocate in step (b) a new instance $A_U(\sigma)$, unless $A_U(\sigma)$ can take a transition in step (c). Also, when an object `o` is garbage-collected, we can discard all the instantiations $A_U(\sigma)$ with $\mathtt{o} \in \mathit{ran}(\sigma)$; it suffices to record the states of $\mathcal{Q}(\sigma)$ in a special $\sigma^\dagger$ for disposed objects.

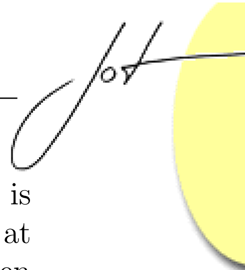
Sandboxes

The programmer defines the scope of a policy through the method `sandbox` of the class `jisel.policy.PolicyPool`. This signature of `sandbox` is:

```
public static void sandbox(String pol,Runnable c) throws SecurityException
```

The string `pol` is the name of the usage automaton that represents the policy to be enforced through the execution of the code `c`. For instance:

```
    PolicyPool.sandbox("file-confine", new Runnable() {
            public void run() {
                // sandboxed code
                ...
            }
        });
```



The effect of this is that the method trace originated after the call to `run()` is constrained to respect the policy defined by the usage automaton `file-confine`, at each step. The past method trace is not considered by the policy. Note that when `run()` returns, the policy is not enforced anymore — indeed, policies are local.

3 CASE STUDY: A SECURE BULLETIN BOARD

We illustrate the expressive power of usage policies through a simple bulletin board system inspired by phpBB, a popular open-source Internet forum written in PHP. We consider a Java implementation of the server-side of the bulletin board, and we make it secure through a set of globally enforced usage policies.

The bulletin board consists of a set of *users* and a set of *forums*. Users can create new *topics* within forums. A topic represents a discussion, to be populated by *posts* inserted by users. Users may belong to three categories: guests, registered users, and, within these, moderators. We call regular users those who are not moderators. Each forum is tagged with a visibility level: `PUB` if everybody can read and write, `REG` if everybody can read and registered users can write, `REGH` if only registered users can read and write, `MOD` if everybody can read and moderators can write, `MODH` if only moderators can read and write. Additionally, we assume there exists a single administrator, which is the only user who can create new forums and delete them, set their visibility level, and lock/unlock them. Moderators can edit and delete the posts inserted by other users, can move topics through forums, can delete and lock/unlock topics. Both the administrator and the moderators can promote users to moderators, and viceversa; the administrator cannot be demoted. The public interface of the bulletin board `SecBB` is given below.

```
User addUser(String username, String pwd);
Session login(String username, String pwd);
void logout(User u, Session s);
void deleteUser(User u0, Session s, User u1);
void promote(User u0, Session s, User u1);
void demote(User u0, Session s, User u1);
void addPost(User u, Session s, Post p, Topic t, Forum f);
void editPost(User u, Session s, Post p, Topic t, String msg);
void deletePost(User u, Session s, Post p, Topic t);
void addTopic(User u, Session s, Topic t, Forum f);
void deleteTopic(User u,  Session s, Topic t, Forum f);
void moveTopic(User u, Session s, Topic t, Forum src, Forum dst);
void lockTopic(User u, Session s, Topic t);
void unlockTopic(User u, Session s, Topic t);
List<Post> viewTopic(User u, Session s, Topic t);
void addForum(User u, Session s, Forum f);
void deleteForum(User u, Session s, Forum f);
void setVisibility(User u, Session s, Forum f, Forum.Visibility v);
void lockForum(User u, Session s, Forum f);
```

```
void unlockForum(User u, Session s, Forum f);
List<Topic> viewForum(User u, Session s, Forum f);
```

In this scenario, we assume only the bytecode of the bulletin board is available. We consider below a substantial subset of the policies implemented (as local checks hard-wired in the code) in phpBB, and we specify them through usage automata. We shall globally enforce these policies by applying the Jisel Runtime Environment to the whole bulletin board code.

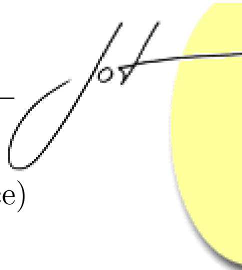
- only registered users can post to a `REG` or `REGH` forum; only moderators can post to a `MOD` or `MODH` forum.
- only registered users can browse a `REGH` forum; only moderators can browse a `MODH` forum.
- the session identifier used when interacting with the bulletin board must be the one obtained at login.
- a message can be edited/deleted by its author or by a moderator, only.
- a regular user cannot edit a message after it has been edited by a moderator.
- a regular user cannot delete a message after it has been replied.
- nobody can post on a locked topic.
- only moderators can promote/demote users.
- only moderators can lock/unlock topics.
- only the administrator can lock/unlock forums.

Also, some sanity checks can be easily specified as usage automata. For instance: session identifiers must be fresh, users have to logout before logging in again, a message cannot be posted twice, a topic cannot be created twice, a message belongs to exactly one topic, a topic belongs to exactly one forum, a deleted message cannot be edited, a deleted topic cannot be moved, non-existing topics and forums cannot be locked/unlocked, *etc.*

We now show how to formalize some of the above-mentioned policies as usage automata. It is convenient to introduce some aliases first.

```
post(u,s,p,t,f) := SecBB.addPost(User u, Session s, Post p, Topic t, Forum f)
edit(u,s,p) := SecBB.edit(User u, Session s, Post p, String msg)
delete(u,s,p) := SecBB.deletePost(User u, Session s, Post p)
lockT(t) := SecBB.lockTopic(User u, Session s, Topic t)
unlockT(t) := SecBB.unlockTopic(User u, Session s, Topic t)
promote(u0,u1) := SecBB.promote(User u0, Session s, User u1)
demote(u0,u1) := SecBB.demote(User u0, Session s, User u1)
```

Consider first the policy requiring that “nobody can post on a locked topic”. This is modelled by the usage automaton `no_post_locked_topic` below. In the start state `q0`, locking the topic `t` leads to `q1`. From `q1`, posting to the topic `t` (i.e.



firing the event `post(*,*,*,t,*)`, where `*` is a wildcard matching any resource) leads to the offending `fail` state, while unlocking `t` leads back to `q0`.

Consider now the policy "only moderators can promote and demote other users". This is modelled by the usage automaton `mod_promote_demote` below. The state `q0` models `u` being a regular user, while `q1`, is for when `u` is a moderator. The first two transitions actually represent `u` being promoted and demoted. In the state `q0`, `u` cannot promote/demote anybody, unless `u` is the administrator. For example, the trace $\eta = \mathtt{promote}(\mathtt{admin}, \mathtt{u}_1)\ \mathtt{promote}(\mathtt{u}_1, \mathtt{u}_2)\ \mathtt{demote}(\mathtt{u}_2, \mathtt{u}_1)$ respects the policy, while $\eta\ \mathtt{promote}(\mathtt{u}_1, \mathtt{u}_3)$ violates it.

```
name: no_post_locked_topic
states: q0 q1 fail
start: q0
final: fail
trans:
q0 -- lockT(t) --> q1
q1 -- unlockT(t) --> q0
q1 -- post(*,*,*,t,*) --> fail
```

```
name: mod_promote_demote
states: q0 q1 fail
start: q0
final: fail
trans:
q0 -- promote(*,u) --> q1
q1 --  demote(*,u) --> q0
q0 -- promote(u,*) --> fail when u!=User.admin
q0 --  demote(u,*) --> fail when u!=User.admin
```

We now specify the policy stating that "the session identifier used when interacting with the bulletin board must be the one obtained at login". For simplicity, we only consider `post` events; adding the other kinds of intraction is straightforward. The needed aliases are defined as follows:

```
login(u,s) := SecBB.login(String username, String pwd)
              > SessionTable.put(User u, Session s)
logout(u) := SecBB.logout(User u)
```

Note that the alias for the `login` event is defined quite peculiarly. This is because the formal parameters of the method `SecBB.login` are two strings (user name and password), while our policy needs to speak about a `User` and a `Session`. To do that, we inspect the code of the method `SecBB.login`, and find that it implements the action of logging in a user by inserting a pair `(User u, Session s)` into a `SessionTable`. This is what the alias above stands for: an event `login(u,s)` is fired whenever an `SessionTable.put(User u, Session s)` occurs between the call and the return of `SecBB.login(String username, String pwd)`. This new kind of aliases can be dealt with through a mapping to usage automata that track both the method calls and the returns. Except from requiring to track the return events, these aliases do not affect the run-time enforcement and model checking.

The policy `post_sid` follows. The state `q0` models `u` being a guest user, while in `q1`, `u` has logged in. In the state `q0`, guests are prohibited from posting a message. In the state `q1`, posting with a session id `s2` is prohibited if `s` is not the same id `s1` obtained at login.

```
name: post_sid
states: q0 q1 fail
start: q0
final: fail
trans:
q0 -- post(u,*,*,*,*) --> fail when u!=User.guest
q0 -- login(u,s1) --> q1
q1 -- logout(u) --> q0
q1 -- post(u,s2,*,*,*) --> fail when s2!=s1
```

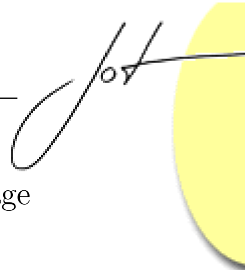
The policy `mod_author_post` states that "a message can be edited/deleted by its author (unless it is a guest) or by a moderator, only". The policy `post_after_reply` below states that a regular user cannot delete a message after it has been replied to. The states `q0`,`q1`, and `q2` correspond to the conditions "not posted", "posted, no reply", and "posted, replied", respectively. Clearly, a `delete` event from `q2` triggers failure. The states `q0'`, `q1'`, and `q2'` correspond to the same conditions, except that `u` has been promoted to the moderator role. Thus, a user `u` that attempts to delete the post `p` from the state `q2'`, does not violate the policy.

```
name: mod_author_post
states: q0 q1 q2 q3 fail
start: q0
final: fail
trans:
q0 -- post(u,*,p,*,*) --> q1
q1 -- edit(u1,*,p) --> fail when u!=u1
q1 -- edit(User.guest,*,p) --> fail
q1 -- delete(u1,*,p) --> fail when u!=u1
q1 -- delete(User.guest,*,p) --> fail
q0 -- promote(*,u1) --> q2
q2 -- demote(*,u1) --> q0
q2 -- post(u,*,p,*,*) --> q3
q1 -- promote(*,u1) --> q3
q3 -- demote(*,u1) --> q1
```

```
name: post_after_reply
states: q0 q1 q2 q0' q1' q2' fail
start: q0
final: fail
trans:
q0  -- post(*,*,p,t,*) --> q1
q1  -- post(*,*,*,t,*) --> q2
q2  -- delete(u,*,p) --> fail
       when u!=User.admin
q0  -- promote(*,u) --> q0'
q0' -- demote(*,u) --> q0
q0' -- post(*,*,p,t,*) --> q1'
q1  -- promote(*,u) --> q1'
q1' -- demote(*,u) --> q1
q1' -- post(*,*,*,t,*) --> q2'
q2  -- promote(*,u) --> q2'
q2' -- demote(*,u) --> q2
```

4 THE JALAPA PROJECT

To support our approach, we started up an open-source project, called Jalapa [31]. Its main products are:



- a runtime environment, called Jisel, that dynamically enforces local usage policies to Java programs.
- a static analyser of Java bytecode, that constructs an abstraction (called history expression) of the behaviour of a program.
- a model checker that reduces the infinite-state system given by the history expression to a finite one, and checks it against a set of policies. Only the policies that do not pass model checking need to be enforced at run-time.
- an Eclipse plugin that combines the previous items into a developer environment, with facilities for writing policies, sandboxing code, and running the static analyses.

In the remaining of this section, we describe in a little more detail the individual components of Jalapa.

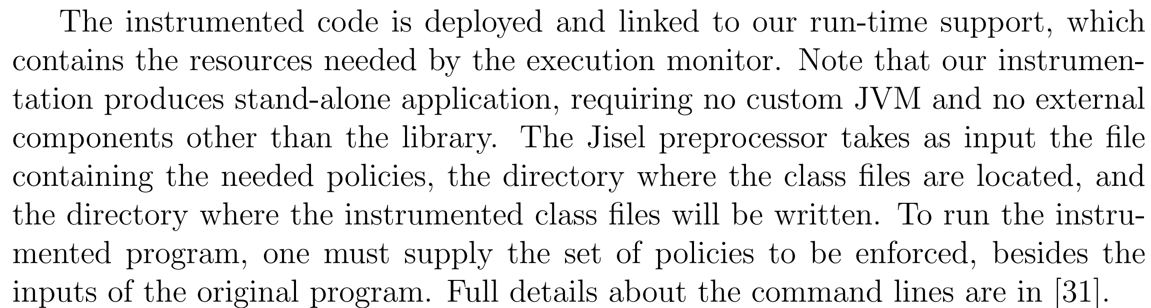
The Jisel Runtime Environment

The first step towards implementing the Jisel runtime environment consists in intercepting the security-relevant operations of the program in hand, so to promptly block them before they violate a policy.

Our approach is based on *bytecode rewriting*, similarly to [47]. We instrument the Java bytecode, by inserting the pieces of code needed to implement the security mechanism. 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 sort of transitive closure, through a visit of the inheritance graph. Similarly to JavaCloak [40], we create a *wrapper* for each of these security-relevant methods. A wrapper $\mathtt{W}_\mathtt{C}$ for the class `C` declares exactly the same methods of `C`, implements all the interfaces of `C`, and extends the superclass of `C`. Indeed, $\mathtt{W}_\mathtt{C}$ can replace `C` in any context, in that it admits the same operations of `C`. The wrapper class $\mathtt{W}_\mathtt{C}$ has a single field, which will be assigned upon instantiation of `C`.

A method `m` of $\mathtt{W}_\mathtt{C}$ can be either monitored or not. If the corresponding method `m` of `C` does not belong to $\mathcal{M}$, then $\mathtt{W}_\mathtt{C}$`.m` simply calls `C.m`. Otherwise, $\mathtt{W}_\mathtt{C}$`.m` calls the `PolicyPool.check` method that controls whether `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 classes in the original program.

The method `PolicyPool.check()` implements the enforcement mechanism in Fig. 2. Weak references [19] are used to avoid interference with the garbage collector. Standard references would prevent the garbage collector from disposing objects referenced by the `PolicyPool` only, so potentially leading to memory exhaustion. An object only referenced by weak references is considered unreachable, so it may be disposed by the garbage collector. The result of `check()` is true if and only if no policy automaton reaches an offending state. If so, the method call is authorized and forwarded to the actual class; otherwise, a `SecurityException` is thrown.



The instrumented code is deployed and linked to our run-time support, which contains the resources needed by the execution monitor. Note that our instrumentation produces stand-alone application, requiring no custom JVM and no external components other than the library. The Jisel preprocessor takes as input the file containing the needed policies, the directory where the class files are located, and the directory where the instrumented class files will be written. To run the instrumented program, one must supply the set of policies to be enforced, besides the inputs of the original program. Full details about the command lines are in [31].

Verification of usage policies

While the run-time enforcement mechanisms of Jisel ensure that policies are never violated at run-time, they perform checks at each method invocation, imposing some overhead on the code running inside a sandbox. In order to mitigate this overhead, we verify programs to detect those policies that are always respected in all the possible executions of the sandboxed code. For those policies that may fail, our technique finds (an over-approximation of) the set of method calls that may lead to violations. By exploiting this information, the run-time enforcement can be optimized, since it is now safe to skip some run-time checks. Note that, in the most general case, the Java source code could be unavailable, yet one might still want to optimize its execution. To this aim, we perform our verification on the Java bytecode.

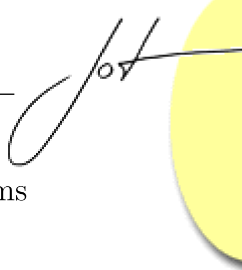
Our verification technique consists of two phases, briefly described below.

- first, we extract from the bytecode a *control flow graph* (CFG), and we transform it into a *history expression* [10], a sort of context-free grammar enriched with special constructs for dealing with policies and events.
- then, we model-check the history expression against the usage policies enforced by the sandboxes used in the program.

We now show an example where our verification technique can be exploited to remove unneeded checks. Consider the following code snippet:

```
PolicyPool.sandbox("ssl", new Runnable() {
  public void run() {
     Connection c = new Connection(url);
     c.startSSL();
     for (String s : confidentialData)
        c.send(s);
  }});
```

The `"ssl"` policy (the full definition is omitted) ensures that no data is sent over the network unless SSL is enabled beforehand, i.e. `startSSL` is invoked before every `send`. Assume the related events are `startSSL(c)` and `send(c)` – we neglect the



parameter `s` here. Our static analysis extracts the control flow graph, and transforms it into the following history expression:

$$H_{\texttt{ssl}} = \nu\texttt{c}.\,\texttt{new(c)} \cdot \texttt{startSSL(c)} \cdot \mu h.\,(\texttt{send(c)} \cdot h + \varepsilon)$$

The operator $\cdot$ models sequential composition, $+$ stands for non-deterministic choice, μh is for recursion, and the binder $\nu\texttt{c}$ defines the scope of a dynamically created resource `c`. Intuitively, $H_{\texttt{ssl}}$ represents all the traces where a new connection `c` is created, an SSL connection over `c` is started, and then a loop of `send(c)` is entered.

We then check $H_{\texttt{ssl}}$ against the `"ssl"` policy using our model checker, and discover that no possible trace violates the policy. Therefore, we can safely remove the sandbox, and directly execute the code, so improving its performance.

The static analyser

The CFG of a program is a static-time data structure that represents all the possible run-time control flows. In particular, we are interested in constructing a CFG the paths of which describe the possible sequences of method calls. This construction is the basis of many interprocedural analyses, and a large amount of algorithms have been developed, with different tradeoffs between complexity and precision [27, 38]. The approximation provided by CFGs is *safe*, in the sense that each actual execution flow is represented by a path in the CFG. Yet, some paths may exist which do not correspond to any actual execution. A typical source of approximation is dynamic dispatching. When a program invokes a method on an object `O`, the run-time environment chooses among the various implementations of that method. The decision is not based on the declared type of `O`, but on the actual class `O` belongs to, which is unpredictable at static time. To be safe, CFGs over-approximate the set of methods that can be invoked at each program point. Similarly, CFGs approximate the data flow, e.g. relating object creation (`new`) to the location of their uses (method invocations). For each method invocation occurring in the program, say `m(x)`, the CFG defines a set of all the possible sources of the object denoted by x, i.e. which `new` could have created `x`. Again, this is a safe approximation in the sense that this is a superset of the actual run-time behaviour.

For each policy for which the original program defines a sandbox, the CFG extracted at the previous step is transformed into an *event graph*. This operation involves substituting events for method signatures, according to the aliases defined in the policy, and suitably collapsing all the other nodes of the graph.

Finally, the event graph is transformed into a history expression. This is done through a variant of the classical state-elimination algorithm for FSA [16]. For instance, the history expression $H_{\texttt{ssl}}$ of the `ssl` example is obtained from the event graph depicted in Fig. 3.

One of the main issues in converting CFGs to history expressions is to correctly track objects. For instance, in Fig. 3 it is important to detect that `c` always denotes

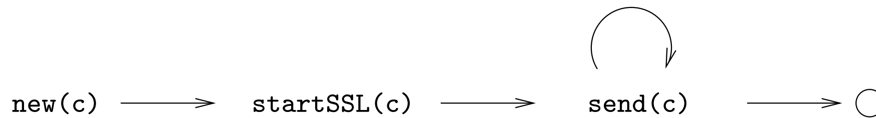


Figure 3: Event graph, from which the history expression $H_{\mathtt{ssl}}$ is obtained.

the same object, so to express this information in the history expression without losing precision. When this is not possible, e.g. because `c = new Connection()` occurs in many places in the code, we need to use a more approximated history expression. This is done by exploiting non-deterministic choice (+). In complex scenarios that involve complex loops, and where `c` is used to represent different objects, we resort to events of the form `m(?)`, where `?` stands for an unknown object.

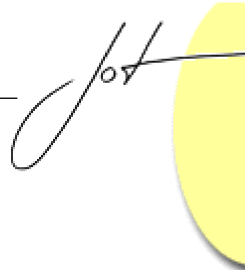
The LocUsT model checker

The final phase of the analysis consists in model-checking history expressions against usage policies. To do that, history expressions are transformed first into Basic Process Algebras (BPAs, [14]), so to enable us to exploit standard model-checking techniques [22]. Our algorithm checks that no trace of the BPA is also a trace recognized as offending by the policy. To do that, we check the emptiness of the pushdown automaton resulting from the conjunction of the BPA and the policy (which denotes the unwanted traces). The transformation into BPA preserves the validity of the approximation, i.e. the traces of the BPA respect the same policies as those of the history expression. The crucial issue is dealing with the dynamic creation of resources, which is not featured by BPAs. To cope with that, we devised a sort of skolemization of history expressions, which uses a finite number of witness resources. This transformation step provides a posteriori justification for the use of history expressions as an intermediate language for program abstraction.

We implemented the model-checker as a Haskell program that runs in polynomial time in the size of the history expression extracted from the event graph. Full details about our technique and our LocUsT model-checker can be found in [10, 11].

5 RELATED WORK.

A wide variety of languages for expressing policies has been proposed over the years, also as extensions to the security model of Java. A typical design compromise is that between expressivity and performance of the enforcement mechanism. The latter can be obtained either through an efficient implementation of the run-time monitor, or by exploiting suitable verification techniques to optimize the monitor (e.g. by removing some or all the dynamic checks).



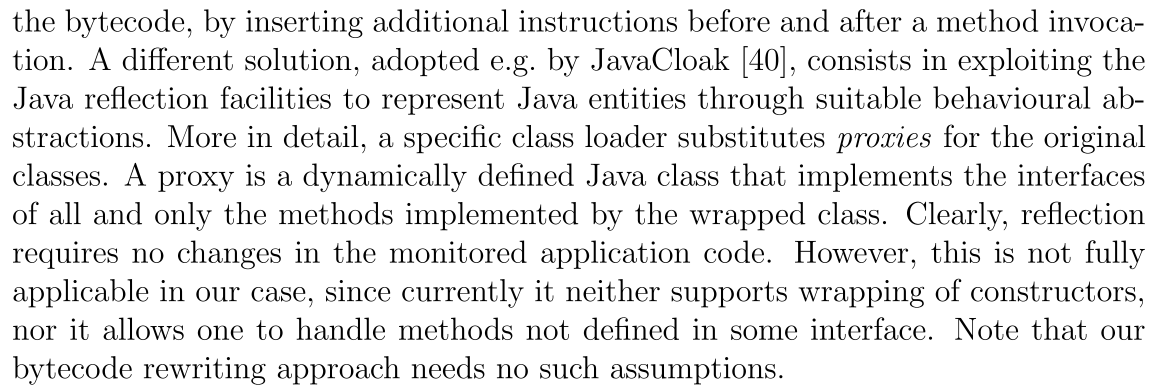
Policy languages

A characterization of the class of policies that can be enforced through run-time monitoring systems is given in [41]. There, the policy language is that of *security automata*, a class of Büchi automata that recognize safety properties. To handle the case of parameters ranging over an infinite domains (e.g. the formal parameters of a method), security automata resort to a countable set of states and input symbols. While this makes security automata hardly usable as an effective formalism for expressing policies, they can be used as a common semantic framework for policy languages. For instance, the semantics of our usage automata can be given in terms of a mapping into security automata. In [28] a characterization is given of the policies that can be enforced through program rewriting techniques. In [12] a kind of automata that can delete and insert events in the execution trace is considered.

Many approaches tend towards maximizing the expressivity, while disregarding static optimizations. In [30] a customization of the JVM/KVM is proposed for extending the Java run-time enforcement to a wider class of security policies, mainly designed for devices with reduced computational capabilities. The proposed framework does not feature any static analysis. Polymer [13] is a language for specifying, composing and dynamically enforcing (global) security policies. In the lines of *edit automata* [12], a Polymer policy can intervene in the program trace to insert or suppress some events. The access control model of Java is enhanced in [39], by specifying fine-grained constraints on the execution of mobile code. A method invocation is denied when a certain condition on the dynamic state of the system is false. Security policies are modelled as process algebras in [2, 34, 35]. There, a custom JVM is used, with an execution monitor that traps system calls and fires them concurrently to the policy. When a trapped system call is not permitted by the policy, the execution monitor tries to force a corrective event – if possible – otherwise it aborts the system call. Since the policies of [13, 39, 2, 34, 35] are Turing-equivalent, they are very expressive, yet they have some drawbacks. First, the process of deciding if an action must be denied might not terminate. Second, non-trivial static optimizations are unfeasible, unlike in our approach.

The problem of deciding whether the contract advertised by an application is compatible with that required by a mobile device is explored in [20]. To do that, a matching algorithm is proposed, based on a regular language inclusion test. In [36] the model is further extended to use *automata modulo theory*, i.e. Büchi automata where edges carry guards expressed as logical formulas. In this approach both the policy and the application behavior are expressed using the same kind of automata. Instead, using history expressions, which have the same expressivity of context-free languages, allows for modelling richer behavior.

The problem of wrapping method calls to make a program obey a given policy has been widely studied, and several frameworks have been proposed in the last few years. Some approaches, e.g. the Kava system [47], use bytecode rewriting to obtain behavioural run-time reflection. This amounts to modifying the structure of

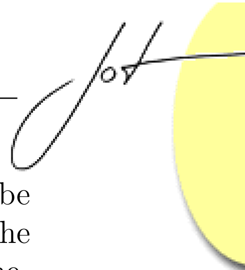


the bytecode, by inserting additional instructions before and after a method invocation. A different solution, adopted e.g. by JavaCloak [40], consists in exploiting the Java reflection facilities to represent Java entities through suitable behavioural abstractions. More in detail, a specific class loader substitutes *proxies* for the original classes. A proxy is a dynamically defined Java class that implements the interfaces of all and only the methods implemented by the wrapped class. Clearly, reflection requires no changes in the monitored application code. However, this is not fully applicable in our case, since currently it neither supports wrapping of constructors, nor it allows one to handle methods not defined in some interface. Note that our bytecode rewriting approach needs no such assumptions.

Verification of security policies

Many authors have studied verification techniques for history-based security at a foundational level. Static and dynamic techniques have been explored in [18, 45, 33, 44], to transform programs and make them obey a given policy. While these approaches consider global policies and no dynamic creation of objects, our model also allows for local policies, and for events parameterized over dynamically created objects. A typed λ-calculus with primitives for creating and accessing resources, and for defining their permitted usages, is presented in [29]. A type system guarantees that well-typed programs are resource-safe. The policies of [29] can only speak about the usage of *single* resources, while ours can span over many resources, as seen in the policies of Sect. 3. A model for history-based access control is proposed in [46], which uses control-flow graphs enriched with permissions and a primitive to check them, similarly to [6]. The run-time permissions are the intersection of the static permissions of all the nodes visited in the past. The model-checking technique can decide if all the permitted traces of the graph respect a given regular property on its nodes. Unlike our usage automata, that can enforce regular policies on traces, the technique of [46] is less general, because there is no way to enforce a policy unless encoded as an assignment of permissions to nodes. Our model checking technique allows for verifying properties of infinite structures (BPAs extracted from history expressions). Efficient algorithms for model checking pushdown systems, which share similarities with BPAs, and other infinite structures are proposed in [23, 17].

A lot of effort has been devoted to develop verification techniques for Java programs. The Soot project [43] provides a comprehensive Java optimization framework, also exploiting static analysis techniques to compute approximated data flow and control flow graphs, and points-to information. In [15] Java source programs are monitored through *tracematches*, a kind of policies that constrain the execution traces of method calls. Static analysis is used to prove that code adheres to the tracematch at hand, so the monitor can be removed. Unlike ours, these policies are global. Also, bytecode analysis is not considered. JACK [4] is a tool for the validation of Java applications, both at the levels of bytecode and of source code. Programmers specify application properties through JML annotations, which are as

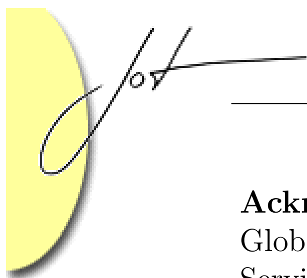


expressive as first-order logic. These annotations give rise to proof obligations, to be statically verified by a theorem prover. The verification process might require the intervention of the developer to resolve the proof obligations, while in our framework the verification is fully automated. JACK can specify history-based policies by using ghost variables spread over JML annotations to mimick the evolution of a finite-state automaton defining the policy. Our formalism allows for expressing history-based policies in a more direct and compact way. An extension of the syntax of JML to express usage policies would be desirable.

6 CONCLUSIONS

We have presented an extension to the security model of Java. This is based on history-based usage policies with local scope. These policies are naturally expressed through usage automata, an extension of finite state automata where edges may carry parameters and guards. The use of these automata is new in the context of Java. It required extending the formal model of [8] with polyadic events, i.e. events involving more than a single object, that model method invocations. We have proposed a programming construct for specifying sandboxes, then designed and implemented an execution monitor for enforcing them. We have devised a static analysis that optimizes the run-time enforcement of policies. We exploit a control flow analysis of Java bytecode and the LocUsT model-checker to predict the policies that will always be obeyed. For each policy which is possibly violated, the model-checker can be used to over-approximate the set of methods leading to the violation, so allowing us to guard them with suitable run-time checks. An open-source implementation of our framework is available [31]. It features the full-fledged runtime environment and model-checker, and a prototype of the static analyser.

We now discuss some possible extensions to our proposal. A significant improvement consists in extending the language of policies by allowing for further logical operators in guards. As a simple extension, we can add the “or” connective. This can be done either in a native fashion, or by exploiting the existing operators. The latter requires to transform logical conditions into disjunctive normal form, and to create an edge for each disjunct. As a further extension, we can add a string matching operator, similarly to the policy specification language of Java 2 [26]. This would allow our policies to exploit the hierarchical structure of file paths and URLs, e.g. to require that new files can only be created under a given directory. The expressive power can be further increased by including the usage of JML boolean expressions, like e.g. the evaluation of pure methods without side effects. This would allow to directly specify policies that depend on implicit counters (e.g. no more than N kilobytes of data can be transmitted). The impact of such a refinement on the static analysis requires further investigation. Another improvement would be to allow our policies to mention the values *returned* by methods. This can be done by exposing these values when generating the “return” events.



Acknowledgments. This research has been partially supported by EU-FETPI Global Computing Project IST-2005-16004 Sensoria (Software Engineering for Service-Oriented Overlay Computers) and by EU project IST-033817 GridTrust – Trust and Security for Next Generation Grids.

REFERENCES

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium, San Diego, California, USA*. The Internet Society, 2003.

[2] F. Baiardi, F. Martinelli, P. Mori, and A. Vaccarelli. Improving grid services security with fine grain policies. In *OTM Workshops*, 2004.

[3] A. Banerjee and D. A. Naumann. History-based access control and secure information flow. In *Proceedings of the Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Cards, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers*, volume 3362 of *Lecture Notes in Computer Science*, pages 27–48, Berlin, 2005. Springer.

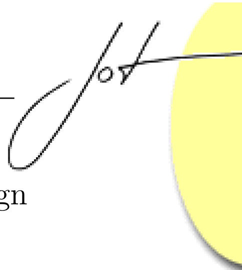
[4] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. Jack - a tool for validation of security and behaviour of Java applications. In *Formal Methods for Components and Objects, 5th International Symposium, Amsterdam, NL, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 152–174. Springer, 2007.

[5] M. Bartoletti. Usage automata. In *Workshop on Issues in the Theory of Security, York, UK*, volume 5511 of *Lecture Notes in Computer Science*, Berlin, 2009. Springer. to appear.

[6] M. Bartoletti, P. Degano, and G.-L. Ferrari. Static analysis for stack inspection. In *Proceedings of the International Workshop on Concurrency and Coordination, Electr. Notes Theor. Comput. Sci.*, volume 54, 2001.

[7] M. Bartoletti, P. Degano, and G.-L. Ferrari. History based access control with local policies. In *Proceedings of the 8th Foundations of Software Science and Computational Structures, FOSSACS 2005, Edinburgh, UK, April 4-8, 2005*, volume 3441 of *Lecture Notes in Computer Science*, pages 316–332, Berlin, 2005. Springer.

[8] M. Bartoletti, P. Degano, G.-L. Ferrari, and R. Zunino. Types and effects for resource usage analysis. In *Proceedings of the 10th Foundations of Software Science and Computational Structures, FOSSACS 2007, Braga, Portugal, March 24-April 1, 2007*, volume 4423 of *Lecture Notes in Computer Science*, pages 32–47, Berlin, 2007. Springer.



[9] M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Semantics-based design for secure web services. *IEEE Trans. Software Eng.*, 34(1), 2008.

[10] M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Model checking usage policies. In *Proceedings of the 4th Trustworthy Global Computing, Barcelona, Spain, November 3-4, 2008*, Lecture Notes in Computer Science, Berlin, 2009. Springer. to appear.

[11] M. Bartoletti and R. Zunino. LocUsT: a tool for checking usage policies. Technical Report TR-08-07, Dip. Informatica, Univ. Pisa, 2008.

[12] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Proceedings of the Workshop on Foundations of Computer Security (FCS)*, 2002.

[13] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI), Chicago, IL, USA, June 12-15, 2005*, pages 305–314, New York, 2005. ACM.

[14] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985.

[15] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2008.

[16] J. Brzosowski and J. E. McCluskey. Signal flow graph techniques for sequential circuit state diagrams. *IEEE Trans. on Electronic Computers*, 1963.

[17] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures http://www-compsci.swan.ac.uk/~csfm/Pubs/handbook01.pdf.

[18] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proceedings of the 27th Annual Symposium on Principles of Programming Languages, POPL*, pages 54–66, New York, 2000. ACM.

[19] K. Donnelly, J. J. Hallett, and A. Kfoury. Formal semantics of weak references. In *ISMM '06: Proceedings of the 5th International Symposium on Memory Management*, 2006.

[20] N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *EuroPKI*, volume 4582 of *Lecture Notes in Computer Science*, pages 297–312. Springer, 2007.

[21] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, pages 413–431, Berlin, 1999. Springer.

[22] J. Esparza. On the decidability of model checking for several μ-calculi and Petri nets. In *Proceedings of the 19th Int. Colloquium on Trees in Algebra and Programming (CAAP'94), Edinburgh, U.K., April 11-13, 1994*, volume 787 of *Lecture Notes in Computer Science*, pages 115–129, Berlin, 1994. Springer.

[23] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2000.

[24] P. W. Fong. Access control by tracking shallow execution history. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2004), 9-12 May 2004, Berkeley, CA, USA*, pages 43–55, Los Alamitos, 2004. IEEE Computer Society.

[25] C. Fournet and A. D. Gordon. Stack inspection: theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, 2003.

[26] L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley, 1999.

[27] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001.

[28] K. W. Hamlen, J. G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.

[29] A. Igarashi and N. Kobayashi. Resource usage analysis. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages, POPL*, pages 331–342, New York, 2002. ACM.

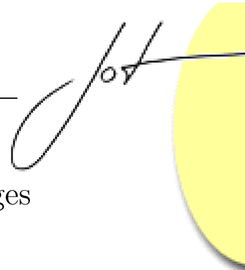
[30] I. Ion, B. Dragovic, and B. Crispo. Extending the Java Virtual Machine to enforce fine-grained security policies in mobile devices. In *ACSAC*, 2007.

[31] Jalapa: Securing Java with Local Policies http://jalapa.sourceforge.net/.

[32] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[33] K. Marriott, P. J. Stuckey, and M. Sulzmann. Resource usage verification. In *Proceedings of the First Asian Symposium on Programming Languages and Systems, APLAS 2003, Beijing, China, November 27-29, 2003*, volume 2895 of *Lecture Notes in Computer Science*, pages 212–229, Berlin, 2003. Springer.

[34] F. Martinelli and P. Mori. Enhancing java security with history based access control. In *Foundations of Security Analysis and Design IV, FOSAD 2006/2007*



Tutorial Lectures, volume 4677 of *Lecture Notes in Computer Science*, pages 135–159. Springer, 2007.

[35] F. Martinelli, P. Mori, and A. Vaccarelli. Towards continuous usage control on grid computational services. In *ICAS/ICNS*, 2005.

[36] F. Massacci and I. Siahaan. Matching midlet's security claims with a platform security policy using automata modulo theory. In *12th Nordic Workshop on Secure IT Systems (NordSec'07)*, 2007.

[37] Microsoft Corp. *.NET Framework Developer's Guide: Securing Applications.*

[38] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[39] R. Pandey and B. Hashii. Providing fine-grained access control for java programs. In *ECCOP'99 - Object-Oriented Programming, 13th European Conference, Lisbon, Portugal, June 14-18, 1999, Proceedings*, volume 1628 of *Lecture Notes in Computer Science*, pages 449–473. Springer, 1999.

[40] K. V. Renaud. Experience with statically-generated proxies for facilitating Java runtime specialisation. *IEEE Proc. Software*, 149(6), Dec 2002.

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

[42] C. Skalka and S. Smith. History effects and verification. In *Proceedings of the Second Asian Symposium on Programming Languages and Systems, APLAS 2004, Taipei, Taiwan, November 4-6, 2004*, volume 3302 of *Lecture Notes in Computer Science*, pages 107–128, Berlin, 2004. Springer.

[43] Soot: a Java optimization framework http://www.sable.mcgill.ca/soot/.

[44] P. Thiemann. Enforcing safety properties using type specialization. In *Proc. ESOP*, 2001.

[45] Úlfar Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *Proceedings of the 1999 Workshop on New security paradigms*, pages 87–95, New York, 1999. ACM.

[46] J. Wang, Y. Takata, and H. Seki. HBAC: A model for history-based access control and its model checking. In *11th European Symposium on Research in Computer Security – ESORICS 2006, Hamburg, Germany, September 18-20, 2006*, volume 4189 of *Lecture Notes in Computer Science*, pages 263–278, Berlin, 2006. Springer.

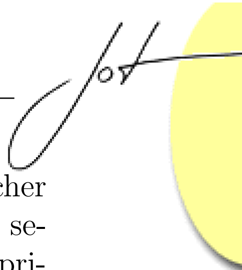
[47] I. Welch and R. J. Stroud. Kava - using byte code rewriting to add behavioural reflection to Java. In *USENIX Conference on Object-Oriented Technology*, 2001.

ABOUT THE AUTHORS

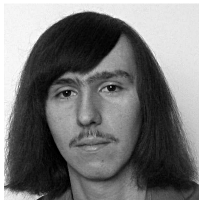
Massimo Bartoletti received the PhD degree in Computer Science from the University of Pisa in 2005, and he is now researcher at the Computer Science Department of the University of Cagliari, Italy. His current research interests are language-based security and security issues in service-oriented computing. Other research interests include control flow analysis and type systems for functional and object-oriented languages. He can be reached at bart@unica.it

Pierpaolo Degano has been full professor of Computer Science since 1990 and, since 1993, he has been at the Department of Computer Science, University of Pisa, being head from 1993 to 1996; since 2006 he is the chairman of the Ph.D. programme in Computer Science; from 1999 to 2003 he chaired the Italian Association of Professors of Computer Science. Pierpaolo Degano served as program chair of many international conferences and as guest editor of many international journals; he served as member of the steering Committees of TAPSOFT, ETAPS, EATCS, and co-founded the IFIP TC1 WG 1.7 on Theoretical Foundations of Security Analysis and Design; since 2005 he is member of the Board of Directors of the Microsoft Research – University of Trento Center for Computational and Systems Biology. His main areas of interest have been, or are, security of concurrent and mobile systems, computational systems biology, semantics and concurrency, methods and tools for program verification and evaluation, and programming tools. He can be reached at degano@di.unipi.it

Gabriele Costa is a Ph.D. student in Computer Science at University of Pisa and a researcher of the security group of the National Research Council (CNR). His research interests include foundational and practical aspects of programming language security. He can be reached at gabriele.costa@iit.cnr.it



Fabio Martinelli (M.Sc. 1994, Ph.D. 1999) is a senior researcher of IIT-CNR, Pisa, where he is the scientific coordinator of the security group. His main research interests involve security and privacy in distributed and mobile systems and foundations of security and trust. He serves as PC-chair/organizer in several international conferences/workshops. He is the co-initiator of the International Workshop series on Formal Aspects in Security and Trust (FAST). He is serving as scientific co-director of the international research school on Foundations of Security Analysis and Design (FOSAD) since 2004 edition. He chairs the WG on security and trust management (STM) of the European Research Consortium in Informatics and Mathematics (ERCIM). He usually manages R&D projects on information and communication security and he is involved in several FP6/7 EU projects. He can be reached at Fabio.Martinelli@iit.cnr.it



Roberto Zunino (M.Sc. 2002, Ph.D. 2006) is assistant professor at the Department of Information Engineering and Computer Science of the University of Trento, Italy. His current research topics include computer security, crypto-protocol verification techiniques, and bioinformatics. Other research interests include language-based security and type systems. He can be reached at zunino@disi.unitn.it