Securing Java with Local Policies

Massimo Bartoletti[1], Gabriele Costa[2], Pierpaolo Degano[1],
Fabio Martinelli[2], and Roberto Zunino[3]

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Italy
[3] Dipartimento di Informatica e Telecomunicazioni, Università di Trento, Italy

Abstract. We propose an extension to the security model of Java. It allows for specifying, analysing and enforcing history-based policies. Policies are defined by finite state automata recognizing the permitted 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, that will only check the program points where some security violation may actually occur.

1 Introduction

A fundamental concern of security is to ensure that resources are used correctly. Devising expressive, flexible and efficient mechanisms to control resource usages 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 [23] and the .NET CLR [31], 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 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, local checks must be explicitly inserted into code by the programmer. Since forgetting even a single check might compromise the safety of the whole application, 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, because a method removed from the call stack no longer affects security. This may be harmful, e.g. when trusted code depends on the results supplied by untrusted code [22].

History-based access control has been receiving major attention as an alternative to stack inspection. Differently from stack inspection, the run-time security

state depends on (a suitable abstraction of) the *whole* execution. History-based policies and mechanisms have been studied at both levels of foundations [3, 21, 35] and of language design and implementation [1, 18]. A common drawback of all these approaches is that the security policy is a *global* invariant, that must hold at any point of the 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 [16, 28]. Furthermore, a large monolithic policy may be hard to understand, and not very flexible either.

Local policies [6], formalise and enhance the concept of *sandbox* [23], while being more flexible than global policies and local checks spread over program code. In the spirit of history-based security [1], local policies can inspect the whole trace of security-relevant events generated by a running program. 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 of an extension to the Java language, so to enhance its security mechanism with local policies. In the spirit of JML [27], policies are orthogonal to Java code and they are specified as comments. Our policies are defined through a special kind of finite state automata (FSA), where the input alphabet comprises the security-relevant events, parametrized over resources. So, policies can express any regular property on execution histories.

The first contribution of this paper is the design of a run-time mechanism for enforcing local policies in Java. Apart from the specification of policies and sandboxes, this requires no intervention by the programmer in the source code.

The second contribution 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 [8]. 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. We have implemented this phase in [10], as a polynomial-time algorithm on the size of the approximation and of the policies. 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.

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 following interface:

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

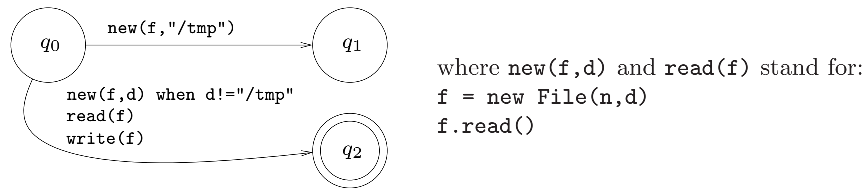


Fig. 1. File confinement policy `file-confine(f,d)`.

The constructor takes as parameters the name of the file and the directory where is is located. A new file is created when no file with the given `name` exists in the given `dir`. The meaning of the other methods is as expected.

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 modelled by the automaton `file-confine(f,d)` in Fig. 1. The edge from q_0 to q_1 represents creating a file `f` in the directory `"/tmp"` (the name of the file is immaterial). The edge from q_0 to q_2 labelled `read(f)` prohibits reading the

file `f` if no `new(f,d)` has occurred beforehand (the double circle means q_2 is *offending*). Similarly for the edge labelled `write(f)`. The edge from q_0 to q_2 labelled `new(f,d) when d!="/tmp"` prohibits creating a file in any directory `d` different from `"/tmp"`.

```
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`. Note also that any attemp 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 Local policies: specification and enforcement

We start by introducing resources, events and policies. The specification of sandboxes and of their run-time enforcement mechanism follows.

2.1 Resources and Events

We model resources $\mathtt{R}_1, \mathtt{R}_2, \ldots$ as objects, and security-relevant events as method calls. For notational convenience, we use *aliases* for events. An alias `ev` for a method signature $(\mathtt{y} : \mathtt{C}).\mathtt{m}(\mathtt{y}_1 : \mathtt{C}_1, ..., \mathtt{y}_\mathtt{n} : \mathtt{C}_\mathtt{n})$ is defined as:

$$\mathtt{alias}\ \mathtt{ev}(\mathtt{x}_1, ..., \mathtt{x}_\mathtt{k}) = (\mathtt{y} : \mathtt{C}).\mathtt{m}(\mathtt{y}_1 : \mathtt{C}_1, ..., \mathtt{y}_\mathtt{n} : \mathtt{C}_\mathtt{n})$$

where $\forall j \in 1..k : \mathtt{x}_\mathtt{j} \in \{\mathtt{y}, \mathtt{y}_1, \ldots, \mathtt{y}_\mathtt{n}\}$. For instance:

```
alias new(f,d) = (f:File).File(string name, string d)
alias read(f)  = (f:File).File.read()
alias write(f) = (f:File).File.write(String t)
```

means `new(f,d)` is an alias for the constructor `File(string name,string d)` of the class `File`, while `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 resource, in this case an object of type `File`. The method parameters not involved in the definition of a policy can be omitted, e.g. we simply write `alias ev = C.m(x1,...,xk)` when all the parameters are immaterial.

2.2 Policies

Usage policies (Def. 1) constrain the usage of resources to obey a regular property on the program *trace*, i.e. the sequence of method calls occurred at run-time. E.g., a file usage policy `file-usage(x)` might require that before reading or writing a file `x`, that file must have been opened, and not yet closed. A usage policy gives rise to an finite state automaton (FSA) when the formal parameters are instantiated to actual resources (see [8] for further details). These automata will be exploited in Sec. 2.4 to implement the execution monitor for usage policies.

Definition 1. Usage policies

Let Ev *be a set of aliases, and let* Res *be the set of resources. A* usage policy $\mathtt{p}(\mathtt{x}_1,\dots,\mathtt{x}_k)$ *is a 5-tuple* $\langle S, Q, q_0, F, E\rangle$*, where:*

- *S is the input alphabet, defined as follows:*

$$S = \{\, \mathtt{ev}(\mathtt{R}_1,\dots,\mathtt{R}_k) \mid \mathtt{ev}(\mathtt{x}_1,\dots,\mathtt{x}_k) \in \mathtt{Ev} \textit{ and } \mathtt{R}_1,\dots,\mathtt{R}_k \in \mathtt{Res}\,\}$$

- *Q is a finite set of states,*
- *$q_0 \in Q \setminus F$ is the start state,*
- *$F \subseteq Q \setminus \{q_0\}$ is the set of final "offending" states,*
- *$E \subseteq Q \times Z \times Q$ is a finite set of labelled edges, where Z is defined as follows:*

$$Z = \{\, \mathtt{ev}(\mathtt{Z}_1,\dots,\mathtt{Z}_k)\ \mathtt{when}\ \mathtt{<cond>} \mid \mathtt{ev}(\mathtt{x}_1,\dots,\mathtt{x}_k) \in \mathtt{Ev} \wedge \mathtt{Z}_i \in \mathtt{Res} \cup \{\mathtt{x}_i\}\,\}$$

where the condition `<cond>` *is defined with the following syntax:*

$$\mathtt{<cond>\ ::=\ true\ \mid\ Z_i\ !=\ Z\ \mid\ <cond>\ and\ <cond>} \quad (\mathtt{Z} \in \mathtt{Res} \vee \mathtt{Z} = \mathtt{x}_j)$$

Usage policies resemble non-deterministic FSA, from which they differ in two points. First, the input alphabet is infinite; second, it does not coincide with the set of labels in the transition relation. Indeed, the parameters $\mathtt{Z}_i$ in the edges of a usage policy can be of two kinds: $\mathtt{Z}_i = \mathtt{R}$ for a static resource R, or $\mathtt{Z}_i = \mathtt{x}_i$. By binding the formal parameters $\mathtt{x}_1,\dots,\mathtt{x}_k$ to actual resources $\mathtt{R}_1,\dots,\mathtt{R}_k$ we obtain a FSA $\mathtt{p}(\mathtt{R}_1,\dots,\mathtt{R}_k)$, to be used in recognizing those traces respecting the policy. Roughly, the transformation into a FSA amounts to: (i) instantiating $\mathtt{x}_i$ to $\mathtt{R}_i$, while respecting the conditions in the `when` clauses, (ii) maintaining $\mathtt{Z}_i = \mathtt{R}$ for R static, and (iii) adding self-loops for all the events not explicitly mentioned in the policy (see [8] for details).

A trace η *respects* a policy $\mathtt{p}(\mathtt{x}_1,\dots,\mathtt{x}_k)$ when, for all the relevant instantiations of the formal parameters $\mathtt{x}_1,\dots,\mathtt{x}_k$ to actual resources $\mathtt{R}_1,\dots,\mathtt{R}_k$ in η, we

have that η is not in the language of the FSA $\texttt{p}(\texttt{R}_1, \ldots, \texttt{R}_\texttt{k})$ – i.e. it is not possible to reach an offending state on η. For instance, consider the following traces:

$$\begin{aligned}
\eta_0 &= \texttt{new(f0,"/tmp") read(f1)} \\
\eta_1 &= \texttt{new(f0,"/tmp") read(f0)} \\
\eta_2 &= \texttt{new(f0,"/tmp") read(f0) new(f1,"/etc")}
\end{aligned}$$

The trace η_0 violates the policy `file-confine`, because it drives the instantiation `file-confine(f1,"/tmp")` to the offending state q_2. The trace η_1 respects the policy, because the `read` event is performed on a newly created file `f` in the directory `"/tmp"` (recall that instantiations have a self-loop labelled `read(f)` on q_1). Instead, η_2 violates the policy, because it drives the instantiation `file-confine(f1,"/etc")` to the state q_2. Indeed, instantiating the `when` clause results in an edge labelled `new(f1,"/etc")` from q_0 to q_2.

We advocate an extension of JML [27, 15] as an instrument for the formal specification of usage policies. The following comment specifies the file confinement policy of Fig. 1. The first part introduces the needed aliases. The usage policy follows, where `states`, `start`, `final`, and `trans` stand respectively for the sets Q, q_0, F and E of Def. 1.

```
\*@ alias new(f,d) = (f:File).File(String name, string d)
  @ alias read(f)  = (f:File).File.read()
  @ alias write(f) = (f:File).File.write(String t)
  @ name:   file-confine
  @ states: q0 q1 q2
  @ start:  q0
  @ final:  q2
  @ trans:  q0 -- new(f,"/tmp") --> q1
  @         q0 -- new(f,d) --> q2 when d != "/tmp"
  @         q0 -- read(f) --> q2
  @         q0 -- write(f) --> q2
  @*/
```

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 policies on statically-known methods. For instance, system resources – which are accessed through the JVM libraries only – can always be protected by policies.

2.3 Sandboxes

The programmer defines the scope of a local policy through the method `sandbox()` of the static class `PolicyPool`. This signature of `sandbox()` is:

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

The string `pol` is the name of 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 set of policies to be checked at run-time is passed as an option to the `java` command, with the following syntax (where $\mathtt{p}_1, \dots, \mathtt{p}_\mathtt{k}$ are policy names):

`java -Dcheck=<value> class` where `value` $::=$ `NONE` $|$ `ALL` $|$ $\mathtt{p}_1; \cdots; \mathtt{p}_\mathtt{k}$

When `value = NONE`, no policy is checked at run-time; when `value = ALL`, all the policies mentioned in the program are checked; in the other case, only the policies $\mathtt{p}_1, \dots, \mathtt{p}_\mathtt{k}$ are checked. Typically, the set of policies that need to be checked at run-time will be provided by the static analysis in Sec. 3.

2.4 Run-time enforcement

The implementation of the execution monitor for local policies goes through the following steps:

- as a preprocessing step, the specification of the policies to be enforced is extracted from the source code and translated into Java code.
- a custom class loader is set up, to act as a proxy for method invocations.
- when starting the execution of a method `sandbox(p,c)`, the policy `p` is activated.
- the proxy dispatches a monitored method call to the actual class, only if the call respects all the active policies.
- when leaving a `sandbox(p,c)`, the policy `p` is deactivated.

The first step is straightforward. For the second step, we use a statically generated proxy for wrapping method calls, similarly to JavaCloak [34]. Before dispatching the call to the actual class, the proxy updates the state of the policy automata. If an active policy is violated, then the proxy throws an exception.

```
public class SecurityProxy implements InvocationHandler {
    private Object obj;
    ...
    public Object invoke(Object proxy, Method meth, Object[] args)
        throws Throwable {
        Object result;
        if(PolicyPool.check(obj, meth, args))    // monitor call
             result = meth.invoke(obj, args);    // method call
        else throw new SecurityException(meth.toString());
    }
}
```

The method `PolicyPool.check()` is the core of the enforcement mechanism. For each active usage policy, it tracks the states of all the needed instantiations. States are modelled as sets of pairs $((\mathtt{R}_1, \ldots, \mathtt{R}_k), \mathtt{q})$, where $(\mathtt{R}_1, \ldots, \mathtt{R}_k)$ is a tuple of weak references [4] to the resources upon which the policy is instantiated, and `q` is the current state of the automaton. 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. For instance, consider the policy `file-confine` of Fig. 1. Assume that the policy is active when the proxy traps a call to the constructor `File("passwd","/tmp")`. The `check()` method looks up the aliases table and finds the event `new(f0,"/tmp")` associated with the constructor. Firing this event updates the states of the policy `file-confine` to:

$$\{((\mathtt{f0}, \texttt{"/tmp"}), \mathtt{q}_1), ((\mathtt{f1}, \texttt{"/tmp"}), \mathtt{q}_0)\}$$

Assume now the method `read()` is invoked on the file `f1`. The state becomes:

$$\{((\mathtt{f0}, \texttt{"/tmp"}), \mathtt{q}_1), ((\mathtt{f1}, \texttt{"/tmp"}), \mathtt{q}_2)\}$$

Since the offending state q_2 has been reached, the call to `read()` is not dispatched by the proxy, which instead throws a `SecurityException`.

3 Static analysis and optimizations

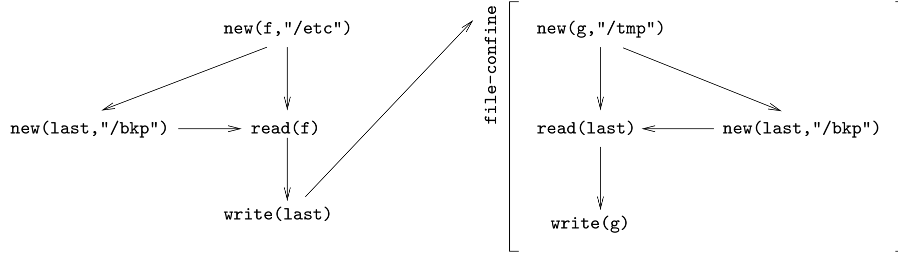
We statically analyse programs to detect those policies that are always respected in all possible executions, so to avoid checking them at run-time. For those policies that may fail, our static analysis finds the method calls that may lead to violations. This allows for optimizing the execution monitor, that will only check the program points where some security violation may actually occur.

The static analysis consists in two phases, briefly described below.

- first, we extract the program *control flow graph* (CFG), and we transform it into a *history expression*.
- then, we model-check the history expression against the usage policies enforced by the sandboxes used in the program.

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, with different tradeoffs between complexity and precision, have been

[4] Weak references [17] are used to avoid interference with the garbage collector. Using standard references would indeed prevent the garbage collector from disposing resources referenced by the `PolicyPool` only, so potentially leading to memory exhaustion. An object referenced only by weak references is considered unreachable, and so it may be disposed by the garbage collector.



$$\nu\texttt{f}.\,\texttt{new(f,"/etc")} \cdot \nu\texttt{last}.\,(\varepsilon + \texttt{new(last,"/bkp")}) \cdot \texttt{read(f)} \cdot \texttt{write(last)} \cdot$$
$$\texttt{file-confine}[\nu\texttt{g}.\,\texttt{new(g,"/tmp")} \cdot \nu\texttt{last}.\,(\varepsilon + \texttt{new(last,"/bkp")}) \cdot \texttt{read(last)} \cdot \texttt{write(g)}]$$

Fig. 2. CFG and history expression of the method `main()`. Sequential composition is modelled by the operator $\cdot$, while $+$ stands for non-deterministic choice. The scope of a dynamically created resource n is defined by the binder νn.

developed [24, 32]. CFGs hide most of the data flow, so approximating the actual behaviour. This approximation 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.

Once a context-sensitive CFG has been extracted, it is transformed into a *history expression* [6], a sort of context-free grammar enriched with special constructs for dealing with policies and resources. To do that, we suitably adapt the classical state-elimination algorithm for FSA [14]. E.g., the CFG and the history expression associated with the `main()` of Sec. 1 are depicted in Fig. 2.

The second phase consists in model-checking history expressions against usage policies. As a first step, history expressions are transformed into Basic Process Algebras (BPAs, [13]), so to enable us to exploit standard model-checking techniques [20]. Roughly, one checks the emptiness of the pushdown automaton resulting from the conjunction of the BPA obtained in the previous step, and the negation of the policy. 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 two main issues are dealing with dynamic creation of resources (not featured by BPAs), and with redundant sandboxes, i.e. nested occurrences of the same sandbox. For the first item, we devised a sort of Skolemization of history expressions, which uses a finite number of witness resources in place of the ν-binders. For the second item, we transformed history expressions to remove the redundant sandboxes therein. Full details about our technique and our model-checking tool can be found in [8, 10].

4 Conclusions

We have presented a proof-of-concept for an extension of the security mechanism of Java. This is based on history-based local policies. These policies are naturally expressed through a sort of finite state automata, the edges of which are parametric over resources. The use of these automata is new in the context of Java. It required extending the formal model of [7] with polyadic events, that model method invocations. We have proposed a programming construct for specifying sandboxes, and designed an execution monitor for enforcing them. We have devised a static analysis that optimizes the run-time enforcement of policies. The analysis exploits call-graph construction and model-checking to predict the policies that will always be obeyed, and to single out the program points where run-time checks are needed. An implementation of our framework is currently under development; only the model-checking tool is already available [10]. It runs in polynomial time in the size of the history expression extracted from the analysed program.

Extensions. A significant improvement to our model consists in extending the language of policies by allowing for more logical operators in conditions. The expressive power can be 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.

Related work. Many authors [16, 19, 28, 36] mix static and dynamic techniques to transform programs and make them obey a given policy. Our model allows for local, polyadic policies and events parameterized over dynamically created resources, while the above-mentioned papers only consider global policies and no parameterized events. Polymer [12] is a language for specifying, composing and enforcing (global) security policies, based on *edit automata* [11]. Run-time monitoring is necessary to enforce policies, while our model-checking tecnique may avoid this overhead. A typed λ-calculus with primitives for creating and accessing resources, and for defining their permitted usages, is presented in [25]. A type system guarantees that well-typed programs are resource-safe, yet no effective algorithm is given to check compliance of the inferred usages with the permitted ones. The policies of [25] can only speak about the usage of *single* resources, while ours can span over many resources, e.g. a policy requiring that no socket connections can be opened after a local file has been read. Wang, Takata and Seki [37] propose a model for history-based access control. They use control-flow graphs enriched with permissions and a primitive to check them, similarly to [5]. 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 local policies, that can enforce any regular policy on traces, the technique of [37] is less general, because there is no way to enforce

a policy unless it is encoded as a suitable assignment of permissions to nodes. Pandey and Hashii [33] enhance the access control model of Java, 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. Since this condition may be the result of calling an arbitrary method, this mechanism is quite general, yet is has some drawbacks. First, the process of deciding if an action must be denied might not terminate. Second, the dynamic conditions in the policy might prevent from static optimizations. Our local policies and static analysis can be smoothly adapted to the case of mobile code. This extension requires analysing bytecode instead of source code when extracting usage policies and when constructing the CFG. Martinelli et al. [2, 29, 30] model security policies as process algebras. They implement a custom JVM, 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. Being interested in efficient run-time enforcement, this framework neglects static optimizations, which however might be unfeasible because of dynamic conditions in the policies. In [26] 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. As before, the presented framework does not feature any static analysis. 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 equi-expressive with first-order logics. 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, although our syntax is not pure JML. The problem of wrapping method calls has been widely studied, and several frameworks have been proposed in the last few years. Some approaches, e.g. the Kava system [38], 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 less invasive solution, adopted e.g. by JavaCloak [34], consists in exploiting the Java core package `java.lang.reflect`. This approach seems particularly appropriate in our framework. Indeed, we can use a custom class loader to substitute a dynamic proxy for the classes involved in the enforcement of security policies. Notably, this solution does not require custom JVMs.

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-funded 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 *Proc. 10th Annual Network and Distributed System Security Symposium*, 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 *Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Cards (CASSIS)*, 2004.
4. G. Barthe et al. JACK - a tool for validation of security and behaviour of Java applications. In *Formal Methods for Components and Objects*, 2007.
5. M. Bartoletti, P. Degano, and G. L. Ferrari. Static analysis for stack inspection. In *Proc. International Workshop on Concurrency and Coordination*, 2001.
6. M. Bartoletti, P. Degano, and G. L. Ferrari. History based access control with local policies. In *Proc. Fossacs*, 2005.
7. M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Types and effects for resource usage analysis. In *Proc. Fossacs*, 2007.
8. M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Model checking usage policies. Technical report, Dip. Informatica, Univ. Pisa, 2008.
9. M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Semantics-based design for secure web services. *IEEE Transactions on Software Engineering*, 34(1), 2008.
10. M. Bartoletti and R. Zunino. LocUsT: a tool for checking usage policies. Technical Report TR-08-07, Dip. Informatica, Univ. Pisa, 2008.
11. L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Foundations of Computer Security (FCS)*, 2002.
12. L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proc. PLDI*, 2005.
13. J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.
14. J. Brzosowski and J. E. McCluskey. Signal flow graph techniques for sequential circuit state diagrams. *IEEE Trans. on Electronic Computers*, 1963.
15. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3), 2005.
16. T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2000.
17. 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.
18. 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*, 1999.
19. Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *Proc. 7th New Security Paradigms Workshop*, 1999.
20. J. Esparza. On the decidability of model checking for several μ-calculi and Petri nets. In *Proc. 19th Int. Colloquium on Trees in Algebra and Programming*, 1994.
21. P. W. Fong. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy*, 2004.

22. C. Fournet and A. D. Gordon. Stack inspection: theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, 2003.
23. L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley, 1999.
24. D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6), 2001.
25. A. Igarashi and N. Kobayashi. Resource usage analysis. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.
26. I. Ion, B. Dragovic, and B. Crispo. Extending the java virtual machine to enforce fine-grained security policies in mobile devices. In *ACSAC*, 2007.
27. 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.
28. K. Marriott, P. J. Stuckey, and M. Sulzmann. Resource usage verification. In *Proc. First Asian Programming Languages Symposium*, 2003.
29. F. Martinelli and P. Mori. Enhancing Java security with history based access control. In *FOSAD*, 2007.
30. F. Martinelli, P. Mori, and A. Vaccarelli. Towards continuous usage control on grid computational services. In *ICAS/ICNS*, 2005.
31. Microsoft Corp. *.NET Framework Developer's Guide: Securing Applications*.
32. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
33. R. Pandey and B. Hashii. Providing fine-grained access control for Java programs. In *Proc. ECOOP*, 1999.
34. K. V. Renaud. Experience with statically-generated proxies for facilitating Java runtime specialisation. *IEEE Proc. Software*, 149(6), Dec 2002.
35. C. Skalka and S. Smith. History effects and verification. In *Asian Programming Languages Symposium*, 2004.
36. P. Thiemann. Enforcing safety properties using type specialization. In *Proc. ESOP*, 2001.
37. J. Wang, Y. Takata, and H. Seki. HBAC: A model for history-based access control and its model checking. In *Proc. ESORICS*, 2006.
38. 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.