Verification of control flow based security properties

T. Jensen D. Le Métayer T. Thorn
IRISA/CNRS/INRIA*
Campus de Beaulieu
F-35042 Rennes Cedex, France

Email: {jensen,lemetayer,thorn}@irisa.fr

Abstract

A fundamental problem in software-based security is whether *local* security checks inserted into the code are sufficient to implement a *global* security property. We introduce a formalism based on a two-level linear-time temporal logic for specifying global security properties pertaining to the control-flow of the program, and illustrate its expressive power with a number of existing properties. We define a minimalistic, security-dedicated program model that only contains procedure call and run-time security checks and propose an automatic method for verifying that an implementation using local security checks satisfies a global security property. For a given formula in the temporal logic we prove that there exists a bound on the size of the states that have to be considered in order to assure the validity of the formula: this reduces the problem to finite-state model checking. Finally, we instantiate the framework to the security architecture proposed for Java (JDK 1.2).

1 Introduction

A number of recently proposed programming languages provide language constructs for enforcing security requirements. Examples include Telescript [19] with its facilities for controlling permissions and resource consumption, and the latest version of the Java Development Kit (JDK 1.2) [10] that provides constructs for granting permissions to code and for checking the permissions of the code executing.

Various features of these languages have been studied in a formal setting with the aim of providing semantically well-founded methods for verifying that a code is secure. The connections between type systems and security have been investigated by Leroy and Rouaix [17] who proved that well-typing can be used to guarantee that a program will not corrupt memory. Volpano et al. [27, 28] have devised type systems for ensuring secure information flow within programs. Dean [8] and Jensen et al. [14] provide formalisations of the dynamic loading of classes in Java. and its influence on protection of code and data. Semantics and verification of bytecode has been extensively studied by several groups (see *e.g.*, [3, 23, 22, 25]).

Each of these contributions focusses on one specific aspect of a security architecture. What is still missing, however, is the possibility of proving that a given code is secure with respect to a global security property (such as the "segregation of duty" property [11] for example). The programmer can use some of the above-mentioned features to reduce the visibility of members of classes or to make security checks at certain points in the code, but this does not guarantee *per se* that the overall behaviour of the program will be secure. The task is complicated by the multitude of facets of security. In Java for example, features like permission checking, privileged instructions, visibility modifiers for classes and their members, and typing all have an impact on security. It is thus necessary to define a program model that is sufficiently general to accommodate these features and yet simple enough to

*Part of this work has been done in the context of Dyade (R&D joint venture between Bull and Inria).

allow the proof of non-trivial properties. The existing security models that we are aware of are quite different from the models commonly used when defining the semantics of high-level programming languages such as Java. Our goal in this paper is precisely to tackle this problem by showing how a semantic model for modelling control-flow can be related to a formalism for specifying security properties, thereby providing (a step towards) a better integration of abstract specifications of security properties and implementations using lower level security mechanisms.

The contribution of the paper is twofold:

- We provide a formal framework for the definition of a class of programming language based security properties. These are properties that depend only on the control structure and the call graph of the program. We show how this framework can be instantiated to verify security properties of applications built using the security architecture of the Java Development Kit (JDK 1.2).

- We propose a new technique for the verification of this class of security properties which makes it possible to reduce an infinite-state transition system to a finite one exhibiting the same behaviour with respect to a given property.

The abstract model of programs with dynamic security checks and its operational semantics are introduced in Section 2. In Section 3 we present a two-level temporal logic for expressing security properties based on this operational semantics. We proceed (Section 4) with the definition of our technique for reducing an infinite transition system to a finite and complete one (with respect to a given property). This framework is applied to the Java Development Kit (JDK 1.2) in Section 5. In Section 6, we illustrate our model with a small example inspired by an electronic commerce application. The verification technique is applied to prove that a global security property is ensured and to detect redundant dynamic checks. Section 7 is devoted to related work and Section 8 suggests avenues for further research.

2 Program model

In order to define a formal security framework which is not tied to one particular programming language, we introduce in this section an abstract model that will serve as the basis for the definition of security properties in the next section. The model abstracts away all data flow and focuses on security checks and control-flow *i.e.*, which procedures (or methods, or functions) are called during execution and in what order. A program is represented by a flow graph with two kinds of edges: *TG* defines the transfer edges (*i.e.*, the usual intra-procedural control flow) and *CG* the call edges (binding call sites to their potential entry points):

$$G = (NO, IS, IT, TG, CG).$$

So *e.g.* a code sequence such as $\mathtt{m}_1\mathtt{()};\mathtt{m}_2\mathtt{()}$ will result in two nodes n_1 and n_2, representing the calls $\mathtt{m}_1\mathtt{()}$ and $\mathtt{m}_2\mathtt{()}$, and a *TG*-edge from n_1 to n_2. In addition there will be a *CG*-edge from n_1 to the node(s) at the beginning of method $\mathtt{m}_1$ and from n_2 to the node(s) at the beginning of method $\mathtt{m}_2$.

The components of the flow graph have the following signatures:

$$\begin{array}{lcl} IS & : & NO \to \{\mathtt{call}, \mathtt{return}, \mathtt{check}\ (\phi)\} \\ IT & : & NO \\ TG & : & NO \to \mathcal{P}(NO) \\ CG & : & NO \to \mathcal{P}(NO) \end{array}$$

The nodes (*NO*) can be seen as program points and IT is the entry point of the whole program. If $m \in TG(n)$ (respectively $m \in CG(n)$) we write $n \overset{TG}{\to} m$ (respectively $n \overset{CG}{\to} m$).

A node can be of the following three kinds, as indicated by IS:

- An ordinary procedure (or method, or function) call. We assume that $IS(n) = \mathtt{call} \Rightarrow CG(n) \neq \emptyset$.
- A `return` node.
- A check node, $\mathtt{check}\ (\phi)$, where ϕ is a property of the state. The `check` instruction represents the programming language feature for dynamically enforcing security properties : execution reaching a check node will stop if the current state does not satisfy property ϕ. The syntax for defining properties is presented in the next section.

It should be noted that our minimalistic model does not contain common control structure such as `if` and `while` since these can be modelled by non-determinism in the flow relation *TG*. For example, if the calls in the statement

```
m₁(); if..then m₂() else m₃()
```

are represented by nodes n_1, n_2, n_3 then there will be *TG*-edges from n_1 to n_2 and from n_1 to n_3. For languages with dynamic method invocation or higher-order functions it is not possible to determine statically what method is invoked in a virtual method call (or variable call). The call graph *CG* describes for each node n a safe over-approximation of what are the possible actual methods (or functions) corresponding to the virtual method invocation (or function call). The techniques for finding this information are well understood and there exist a number of semantically well-founded static analyses (see *e.g.*, [21, 12]) designed for this purpose. The results reported in this paper are independent of the particular analysis chosen so from here on we shall just assume that a call graph is available.

The operational semantics of the language is defined as a transition system with a state consisting of a control stack.

$$State \quad = \quad NO^\star$$

Nodes are the program points in our model, so the control stack is a stack of nodes. A control stack $n_1 : n_2 : n_3$ means that the call at node n_1 invoked a method during whose execution node n_2 was reached. Node n_2 in turn represents a call to a method ($\mathtt{m}_3$, say) and the body of $\mathtt{m}_3$ contains a node n_3 which is the current point of control. Thus, the "current program point" of a standard operational semantics, indicating which node to execute next, is the top element of the stack. The control stack is used to determine where to go to when executing a `return` node. referring to the stack above, if n_3 is a `return` node then execution of the call that was initiated at node n_2 has terminated, and execution continues with the nodes following n_2 (*c.f.* the semantic rule for `return` below).

The operational semantics of a program G is defined by a transition relation $\rhd$. Two states s_1 and s_2 are related by $\rhd$, written $s_1 \rhd s_2$, if execution can lead directly from s_1 to s_2. The following three rules constitute an inductive definition [32] of this relation.

$$\frac{IS(n) = \mathtt{call} \quad n \stackrel{CG}{\to} m}{cs : n \rhd cs : n : m}$$

$$\frac{IS(m) = \mathtt{return} \quad n \stackrel{TG}{\to} n'}{cs : n : m \rhd cs : n'}$$

$$\frac{IS(n) = \mathtt{check}\ (\phi) \quad cs : n \vdash \phi \quad n \stackrel{TG}{\to} n'}{cs : n \rhd cs : n'}$$

The relation $\vdash$ stating when a state satisfies a property is defined in the next section.

For verification purposes we shall be interested in the set of states that execution can lead to. Given a program G with a designated start node IT, the operational semantics gives rise to a rooted transition system $(State, \rhd, \langle IT \rangle)$.

Definition 2.1 *The trace semantics of* $G = (NO, IS, IT, TG, CG)$ *is the set of sequences of states (or "execution traces") reachable from the initial configuration:*

$$\{T \in State^\star \mid \; T_0 = \langle IT \rangle \;\; and \;\; \forall k < |T| - 1,\ T_k \rhd T_{k+1}\}.$$

We write this set $[\![G]\!]$.

Note that the elements of $[\![G]\!]$ are sequences of stacks *i.e.*, sequences of finite sequences of nodes from the flow graph.

3 Formal definition of security properties

The only way the operational semantics of the previous section is related to security issues is through the `check` instruction. This instruction models a programming language mechanism that enforce a security property at a given point in the execution. A difficult problem however is to be able to relate a collection of such *local* run-time checks with a global security goal. It is well known that very strong security constraints on one software component can be defeated by subtle omissions in another cooperating component. Another facet of the problem is that a defensive implementation, involving security checks before each procedure call for example, would be extremely inefficient. So, it is desirable to be able to determine statically

$$\phi ::= NP \mid \mathbf{X}\phi \mid \phi_1 \,\mathbf{U}\, \phi_2 \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \ True$$

$$\begin{array}{lll}
S \vdash \phi & \Leftrightarrow & \forall T \in S, \forall s \in T, s \vdash \phi \\
s \vdash NP & \Leftrightarrow & |s| \geq 1 \ and \ NP(s_0) \\
s \vdash \mathbf{X}\phi & \Leftrightarrow & |s| \geq 1 \ and \ s^1 \vdash \phi \\
s \vdash \phi_1 \,\mathbf{U}\, \phi_2 & \Leftrightarrow & \forall i < |s|, \ s^i \vdash \phi_1 \quad or \\
 & & \exists k, \quad s^k \vdash \phi_2 \ and \\
 & & \qquad \forall i < k, \ s^i \vdash \phi_1 \\
s \vdash \neg\phi & \Leftrightarrow & s \not\vdash \phi \\
s \vdash \phi_1 \wedge \phi_2 & \Leftrightarrow & s \vdash \phi_1 \ and \ s \vdash \phi_2 \\
s \vdash True & \Leftrightarrow & true
\end{array}$$

Figure 1: Language for the definition of security properties

that certain checks are not necessary to enforce the intended security property. The first step to achieve the above goals is to provide a formalism for defining security properties. In this section, we propose a language to express security properties as properties on sets of execution traces (section 3.1) and we illustrate its expressive power with a collection of well-known security properties (section 3.2).

3.1 A formalism for defining security properties

Since the semantics of a program is defined as a set whose elements are traces of stacks of nodes, there are three different kinds of predicates to consider:

1. Predicates describing single nodes.
2. Predicates describing stacks of nodes.
3. Predicates describing traces of stacks.

A node corresponds to a program point so the predicates on nodes characterise basic security properties attached to a piece of code like its protection domain or site of origin. These are the predicates that are ranged over by *NP* in Figure 1. Exactly what predicates are needed depends on the actual security property to formalise so we do not specify the set of node predicates further.

States were defined in the previous section as stacks, *i.e.*, finite sequences of nodes. We choose to write state predicates using a linear-time temporal logic whose syntax and semantics are defined in Figure 1. The base predicates of the logic are the node predicates and the set of logical connectors include conjunction, a next operator $\mathbf{X}$ and an until operator $\mathbf{U}$. Disjunction $\vee$, implication $\Rightarrow$ and $False$ can be derived from this minimal language, using conjunction, negation, and $True$. As usual, we also introduce the derived operator $\mathbf{G}\phi \equiv \phi \,\mathbf{U}\, False$ to express that ϕ holds of all elements of a sequence and $\mathbf{F}\phi \equiv \neg(\mathbf{G}(\neg\phi))$ to express that ϕ must hold for one element of a sequence. In Figure 1 we define the satisfaction relation $s \vdash \phi$ that formalises when a stack s satisfies a predicate ϕ.

Concerning the trace predicates, we have limited our work to *safety* properties. This restriction is not uncommon in the domain of security properties—see *e.g.* the work by Schneider [24] on enforceable security properties for a discussion on the topic. The consequence of this restriction is that all trace predicates are global invariants on the stacks in the traces *i.e.*, for a stack property ϕ the predicates are of the form

ϕ holds for all stacks in the trace.

Since stack predicates are lifted to trace predicates in a unique way, we overload the satisfaction relation $\vdash$ between stacks and stacks predicates and write

$$S \vdash \phi$$

to mean that the set of traces S satisfies the property induced by the stack predicate ϕ. Formally, this means that all stacks in all traces in S satisfy ϕ.

In the semantic definition of $\vdash$ we use $|s|$ to denote the length of a sequence s, s_i to denote the i^{th} element of s and s^i to represent the suffix of s starting at s_i[1]. Note that, from the definition of $S \vdash \phi$, the semantics of a formula ϕ is defined with respect to a (possibly infinite) set of finite sequences s (each of these sequences will correspond to a possible execution stack of the program). We use a weak version of the *until* operator $\phi_1 \,\mathbf{U}\, \phi_2$ in the sense that the definition does not require that ϕ_2 eventually holds. This choice is not significant however since the

[1] Thus s^i is the empty sequence if $|s| = i$ and s^i is undefined if $|s| < i$.

strong version can be derived from $\mathbf{U}$ and the $\mathbf{F}$ operator defined above.

We are now in a position to define what we mean by "a program satisfies a security property".

Definition 3.1 *A program G satisfies a security property ϕ, which is denoted by $G \models \phi$, if and only if $[\![G]\!] \vdash \phi$.*

3.2 Examples of security properties formalised in our framework

Our approach is dedicated to properties of the control flow of the application. This allows us to formalise a number of commonly used properties as illustrated by the following four examples.

The segregation of duty is often required in financial applications where security is ensured by imposing that a task cannot be completed unless at least two principals are involved [11]. In our framework, a principal (or a subject) is defined by a property which is satisfied only by the nodes corresponding to program points in its code. We can further gather principals into larger groups like *Manager*, *Accountant*, *etc*. The segregation of duty property imposing, for example, that a code of the category *Critical* can only be executed if backed by a manager and an accountant can be expressed as follows:

$$\begin{aligned} SD = \quad & (\neg \mathit{Critical}\ \mathbf{U}\ \mathit{Manager}) \land \\ & (\neg \mathit{Critical}\ \mathbf{U}\ \mathit{Accountant}). \end{aligned}$$

From the semantics of the logic (Figure 1) we have that this property is satisfied if and only if all the possible execution stacks satisfy the two following properties:

1. No node satisfying the property *Critical* occurs before the first node satisfying the property *Manager*.

2. No node satisfying the property *Critical* occurs before the first node satisfying the property *Accountant*.

Resource protection can impose that code from protection domain A can only call code belonging to a domain C via code of protection domain B. Identifying the name of a protection domain D with the node predicate "n belongs to D", this property is specified as follows:

$$RP = \mathbf{G}(\neg A \vee (\neg C\ \mathbf{U}\ B)).$$

In other words, if, for a stack s, A happens to be satisfied by a node n in s , then $\neg C \mathbf{U} B$ must hold at that point, which means that no node from protection domain C can occur after n in the stack s before the first node coming from domain B.

The sandbox model was originally proposed as the security model for Java applications. The model implies that a dynamically loaded method originating from site S can only call methods originating from the same site or local methods. Using the property Local to characterise local nodes and Site_S for nodes belonging to site S, this property can be specified as follows:

$$SB = \mathbf{G}\ (\mathit{Site}_S \Rightarrow \mathbf{X}\ (\mathit{Site}_S \vee \mathit{Local})).$$

Thus, for all nodes in the call stack, if the node called is from a non-local site then the next call should either be to code from the same site or back to some local code (that then can call other sites).

Stack inspection is the basis of the security mechanism of the Java Development Kit JDK 1.2. The security requirement here is that any code that lead to a given method call indeed has the right to perform this call. This prevents that some code execute calls on behalf of another code that is not authorised to do so. The only exception to this is code that has been designated "privileged"—this code can execute all calls regardless of who called it. Operationally speaking, this security policy is enforced by *stack inspection* which means examining the stack from its top (last element pushed), checking for each member of the stack that it has the relevant authorisation, until a member is designated as privileged.

Assuming that privileged code satisfy the property Priv, the stack inspection policy that checks for permission ϕ is characterised by the following formula in our formalism:

$$JDK(\phi) = \mathbf{G}((\mathbf{X}(\mathbf{F}\ \mathit{Priv})) \vee \phi).$$

The property imposes that for any execution stack s and any node n in s, either $Priv$ is satisfied by a node that follows n in s (that is to say $\mathbf{X}(\mathbf{F}\ Priv)$) or n must satisfy ϕ.[2] In other words, it amounts to forgetting the nodes traversed by the code before the last node satisfying $Priv$ occurs (if such a node occurs); this node and all the remaining top nodes must then satisfy ϕ.

The property $JDK(\phi)$ illustrates the fact that our language can be used both to express global security properties and local properties that are checked at run time (through the `check` (ϕ) instruction in our programming model). We describe an application of this in Section 5.

4 Finite state verification

In the previous sections we have defined an operational model for programs and a language for specifying security properties. In this section we present a method for verifying that a program satisfies a given security property. The method is a general verification technique specialised to the particular semantic model and the specification formalism used in this paper. It is applicable to properties other than security properties and, as a consequence, this section is not formulated in a security-specific manner.

Designing a mechanical verification method is complicated by the fact that the operational semantics of a program is a possibly infinite-state transition system. Here, infinity arises from recursion in the program, leading to stacks that grow infinitely. Another source of complexity in this context comes from the `check` instruction whose effect is to cut certain execution traces. In order to get a decision procedure for security properties, we propose a technique for mapping an infinite transition system into a finite system which is equivalent to the original one with respect to a given property. In contrast with previous proposals to tackle this problem, the states of our finite system are not abstractions (or sets) of states of the original system. They form instead a selected subset of the states of the original system. Roughly speaking, they are the states of the original system whose "measure" is less than a given threshold. The key point is that this threshold is derived from the program and the property to be verified. The advantage of the technique is that it is both complete (with respect to the original programming model) and flexible since the size of the finite system depends on a measure of the "complexity" of the program and the property to be verified.

Note that we are not concerned with the problem of analysing the finite system itself in this paper; we assume that we can rely on available model checking algorithms for this task.

We first introduce our definition of complexity for properties and programs.

Definition 4.1 *The complexity $\mathcal{C}(\phi)$ of a property ϕ is defined as:*

$$
\begin{array}{lcl}
\mathcal{C}(NP) & = & 1\\
\mathcal{C}(\mathbf{X}\phi) & = & 1+\mathcal{C}(\phi)\\
\mathcal{C}(\phi_1\ \mathbf{U}\ \phi_2) & = & max(\mathcal{C}(\phi_1),\mathcal{C}(\phi_2))\\
\mathcal{C}(\neg\phi) & = & \mathcal{C}(\phi)\\
\mathcal{C}(\phi_1\wedge\phi_2) & = & max(\mathcal{C}(\phi_1),\mathcal{C}(\phi_2))\\
\mathcal{C}(True) & = & 0
\end{array}
$$

So the complexity of a formula is its maximum number of embedded $\mathbf{X}$ operators (plus one). The complexity of a program is defined as the maximum of the complexities of the properties occurring in its `check` instructions.

Definition 4.2 *The complexity of a program $G = (NO, IS, IT, TG, CG)$ is defined as*

$$\mathcal{C}(G) = max\{\mathcal{C}(\phi) \mid n \in NO,\ IS(n) = \texttt{check}\ (\phi)\}.$$

In order to obtain a finite transition system, we must restrict the set of stacks by imposing a threshold. This threshold does not concern the size (or number of elements) of a stack, but the number of contiguous identical elements on the stack. We call this value the *measure* of the stack.

Definition 4.3 *Let s be a sequence of nodes. The measure of s is defined as*

$$\mathcal{M}(s) = max\{k \mid \exists i, \forall j \in [0, k-1], s_i = s_{i+j}\}.$$

We can now introduce our definition of a *limited transition system* and *limited semantics*.

[2] A technical point: note that the property $JDK(\phi)$ is formulated so that it enforces that the last node satisfying $Priv$ (the node not followed by another $Priv$ node) also satisfies the property ϕ.

Definition 4.4 *The p-limited semantics of a program $G = (NO, IS, IT, TG, CG)$ is defined as:*

$$\llbracket G \rrbracket_p = \{T \in States^\star \mid \; T_0 = \langle IT \rangle \;\; and \;\; \forall k < |T| - 1, \; T_k \rhd_p T_{k+1}\}$$

where the p-limited transition relation $\rhd_p$ is defined by

$$s \rhd_p s' \quad \Leftrightarrow \quad s \rhd s' \text{ and } \mathcal{M}(s') \leq p.$$

So the p-limited semantics of a program is defined like the semantics of Section 2 except that it is based on a p-limited transition system $\rhd_p$ which refuses all transitions that would lead to a stack with more than p contiguous identical elements. The intention is to use the p-limited transition system for the verification of security properties, which motivates the following definition:

Definition 4.5 $G \models_p \phi \quad \Leftrightarrow \quad \llbracket G \rrbracket_p \vdash \phi$.

We are now in a position to state the main result of this section.

Theorem 4.6 *If $p \geq \mathcal{C}(G)$ and $p \geq \mathcal{C}(\phi)$ then $G \models \phi \Leftrightarrow G \models_p \phi$.*

This theorem shows that it is possible to perform the verification of a property on the limited transition system $\rhd_p$ without any loss of accuracy, provided that p is greater than (or equal to) the complexities of the property and the program. The significance of this result follows from the fact that the $\rhd_p$ transition system is finite provided that the program modelled by the graph G does not involve mutual recursion. This is because the length of a stack in $\rhd_p$ is bounded by $N \times p$ with N the number of nodes of G. The restriction concerning mutual recursion is not severe for security properties because it can be circumvented in different ways in this context. The simplest solution is to associate identical properties to the nodes involved in a mutually recursive cycle, which allows us to treat them as identical nodes with respect to security properties (and thus apply the results presented in this section). This solution does not appear to lead to any lack of accuracy in practical applications since mutually recursive calls crossing security protection domains do not seem to be common practice. We emphasise that the restriction still allows a series of calls that switches between two or more protection domains (such as *syscalls* and *opcalls* in operating systems), as long as these calls are not mutually recursive. Ongoing work is considering how this restriction can be lifted by combining our verification method with analysis techniques for graph reduction.

To convey some intuition about Theorem 4.6 and the complexity measure $\mathcal{C}$, let us compare the properties $\phi_1 \; \mathbf{U} \; \phi_2$ and $\phi_1 \; \mathbf{U} \; (\mathbf{X}\phi_2)$. We assume that ϕ_1 and ϕ_2 are basic node properties and that nodes n_1, n_2, and n_3 satisfy respectively the properties $\phi_1 \wedge \neg\phi_2$, $\neg\phi_1 \wedge \neg\phi_2$ and $\phi_1 \wedge \phi_2$. Then the property $\phi_1 \; \mathbf{U} \; \phi_2$ is false for the two sequences $n_1 : n_2 : n_3$ and $n_1 : n_2 : n_2 : n_3$. In contrast, the property $\phi_1 \mathbf{U} (\mathbf{X}\phi_2)$ holds for the sequence $n_1 : n_2 : n_3$ but not for $n_1 : n_2 : n_2 : n_3$. So checking $\phi_1 \; \mathbf{U} \; (\mathbf{X}\phi_2)$ on sequences of measure 1 is not sufficient. This is in line with our definition of complexity since $\mathcal{C}(\phi_1 \; \mathbf{U} \; \phi_2) = 1$ and $\mathcal{C}(\phi_1 \; \mathbf{U} \; (\mathbf{X}\phi_2)) = 2$. Showing that Theorem 4.6 always holds is not straightforward, however, since properties like ϕ_1 and ϕ_2 could in general be sequence properties (rather than just node properties).

In order to prove Theorem 4.6, we have to introduce a notion of p-reduction (noted $\rightsquigarrow_p$) of a sequence. Intuitively, $s \rightsquigarrow_p s'$ if s' can be obtained from s by removing an element from a sequence of $p + 1$ contiguous identical elements:

Definition 4.7

$$s \rightsquigarrow_p s' \Leftrightarrow$$

$$s = s_0 : \ldots : s_k : \ldots s_{k+p} : \ldots : s_n$$

$$s' = s_0 : \ldots : s_k : \ldots s_{k+p-1} : \ldots : s_n$$

with $p > 0$ and $\forall i \in [0, p]$, $s_{k+i} = s_k$

The key lemma for the proof of Theorem 4.6 is the following:

Lemma 4.8

$$(p \geq \mathcal{C}(\phi) \text{ and } s \overset{*}{\rightsquigarrow}_p s') \;\; \Rightarrow \;\; (s \vdash \phi \Leftrightarrow s' \vdash \phi).$$

This lemma shows that proving that a sequence s satisfies a property ϕ can be done by checking the property on a reduced sequence

s' which does not contain any subsequence of more than p contiguous elements (with p the complexity of ϕ, or any greater value).

Lemma 4.8 can be proven by a double induction on the length of the $s \overset{*}{\rightsquigarrow}_p s'$ reduction and on the form of the property ϕ. The induction on the length of the $s \overset{*}{\rightsquigarrow}_p s'$ reduction follows from a straightforward transitivity argument; so we are left with proving:

$$(p \geq \mathcal{C}(\phi) \text{ and } s \rightsquigarrow_p s') \Rightarrow (s \vdash \phi \Leftrightarrow s' \vdash \phi).$$

For the sake of conciseness, we just present the case $\phi = \mathbf{X}\phi_1$ here. We consider first the implication $s \vdash \mathbf{X}\phi_1 \Rightarrow s' \vdash \mathbf{X}\phi_1$. We assume $p \geq \mathcal{C}(\phi)$ *and* $s \rightsquigarrow_p s'$. From the semantics of $\mathbf{X}$ (Figure 1), we have

$$s \vdash \mathbf{X}\phi_1 \quad \Leftrightarrow \quad |s| \geq 1 \text{ and } s^1 \vdash \phi_1.$$

From the definition of $\mathcal{C}$ we have $\mathcal{C}(\mathbf{X}\phi_1) = 1 + \mathcal{C}(\phi_1)$, thus $p \geq \mathcal{C}(\mathbf{X}\phi_1) \Rightarrow p-1 \geq \mathcal{C}(\phi_1)$ and $p \geq 2$. Furthermore, from the definition of $\rightsquigarrow_p$, we can show $s \rightsquigarrow_p s' \Rightarrow s^1 \rightsquigarrow_{p-1} s'^1$ with $p \geq 1$. So we can apply the induction hypothesis on ϕ_1 and get $s'^1 \vdash \phi_1$, which implies $s' \vdash \mathbf{X}\phi_1$. We can prove in a similar way that $s \not\vdash \mathbf{X}\phi_1 \Rightarrow s' \not\vdash \mathbf{X}\phi_1$, which allows us to conclude for the case $\phi = \mathbf{X}\phi_1$. The proof of the other cases can be done along the same lines.

The following results makes it possible to use Lemma 4.8 to relate the standard semantics and a p-limited semantics.

Lemma 4.9 *For all T, s, and s',*

$$T \in [\![G]\!],\ s \in T,\ s \overset{*}{\rightsquigarrow}_p s' \text{ and } p \geq \mathcal{C}(G) \text{ and } p \geq \mathcal{M}(s') \Rightarrow$$

$$\exists T' \in [\![G]\!]_p,\ s' \in T'$$

Lemma 4.10 $[\![G]\!]_p \subseteq [\![G]\!]$.

The proof of Lemma 4.9 is made by induction on the length of the $s \overset{*}{\rightsquigarrow}_p s'$ reduction. The key intuition for the induction case is that the condition $p \geq \mathcal{C}(G)$ guarantees that any `check` (ϕ) instruction occurring in the program will have the same effect on a state s and on a state s' such that $s \overset{*}{\rightsquigarrow}_p s'$ (from Lemma 4.8). Furthermore any transition leading to a stack t whose measure is less than p will be allowed by the transition system $\rhd_p$.

Lemma 4.10 follows from Definitions 2.1 and 4.4.

Theorem 4.6 can now be proved as follows. Assuming $p \geq \mathcal{C}(G)$ and $p \geq \mathcal{C}(\phi)$, we have:

$$
\begin{array}{lll}
G \models \phi & \Leftrightarrow & \forall T \in [\![G]\!],\ \forall s \in T,\ s \vdash \phi \\
 & \Leftrightarrow & \forall T \in [\![G]\!],\ \forall s \in T,\ \forall s', \\
 & & (s \overset{*}{\rightsquigarrow}_p s' \text{ and } p \geq \mathcal{M}(s')) \Rightarrow \\
 & & s' \vdash \phi \\
 & & \textit{from Lemma 4.8 and} \\
 & & p \geq \mathcal{C}(\phi) \\
 & \Leftarrow & \forall T' \in [\![G]\!]_p,\ \forall s' \in T',\ s' \vdash \phi \\
 & & \textit{from Lemma 4.9 and} \\
 & & p \geq \mathcal{C}(G) \\
 & \Leftrightarrow & G \models_p \phi \\
G \models \phi & \Rightarrow & G \models_p \phi \\
 & & \textit{from Lemma 4.10}
\end{array}
$$

5 Application to the Java Development Kit

The Java Development Kit JDK 1.2 is one of the most prominent proposals for language based security management. In this section we show how the JDK 1.2 security mechanisms can be described in our framework. The next section provides an illustration of the model through an electronic commerce example.

The JDK 1.2 security model assigns protections domains to code based on its signature and defines the security property as a global assignment of permissions to protection domains. The virtual machine does not verify the permissions itself, but the standard library provides the special class `AccessController` with a number of security related methods. Of these, `checkPermission` verifies that a given permission is granted in the given context, and throws an exception if not. For a permission to be granted, all the methods on the call stack must have the permission granted. As this is too restrictive in general, the ability is provided to mark certain calls as *privileged*, which temporarily discards all of the previous callers from a permission checking point of view.

The assignment of a protection domain to a given piece of code means that each node in the corresponding graph belongs to a protection domain. We stipulate that a node satisfies the property "has permission P" if it belongs to a protection domain with permission P. Furthermore, a node corresponding to a privileged call is assigned the special property *Priv*. However, being privileged or not is a dynamic property in Java 1.2 [10, 13], enabled with the method call `beginPrivileged` and disabled with the method call `endPrivileged`[3], whereas in our model it is a static property of the code. We make the assumption (true of all the examples we have seen) that we can statically identify the privileged code, thus disallowing calls to `endPrivileged` under dynamic control. Calls to `beginPrivileged` and `endPrivileged` disappear in our model, but the nodes delimited by the two are recorded as satisfying the property *Priv*). A call to `checkPermission(Perm)` from class `AccessController` can then be modelled by the instruction $\texttt{check}(JDK(Perm))$ with JDK as defined at the end of Section 3.2.

To obtain the graph corresponding to a Java program, its code is transformed into basic blocks and everything but method calls is abstracted away. Calls to methods that might throw an exception give rise to an additional edge to the code handling the exception, or the exit node if no handler is present. To improve the quality of the approximated call graph, we apply a standard class hierarchy analysis, also known as a type analysis. The literature is abundant with such analysis; in the current implementation we use a simple constraint based flow-insensitive algorithm [21], taking into account the visibility rules of Java [14]. A prototype embedding the techniques described here has been developed [26]. The next section describes a small example that was analysed automatically by this prototype.

6 Electronic commerce example

In this section we illustrate the concepts involved through an electronic commerce example. Four protection domains (corresponding to four principals) are involved. They are called *System, Provider, Client,* and *Unknown*:

- We assume the system (Figure 2) supplies code to implement a controlled floating point variable. This variable has entry points for `read` and `write` operations, protected with a `check` for the respective permissions. The system also supplies a main method (not shown), serving as an initial entry point to the application.
- Using the controlled variable, the service provider builds an account manager (Figure 3) with a `debit` transaction and a boolean query method `canpay`. For this to work, we assume that the provider is granted the `Write`, `Read`, `Debit`, and the `Canpay` permissions. The `debit` and `canpay` methods call `read` and `write` in a privileged mode because they can be called by clients which do not have the permission to call `read` and `write` directly (*i.e.*, which are not granted the `Read` and `Write` permissions).
- Completing the application, the client builds an application on top of the account manager (Figure 4). We do not detail the application, but the idea is one of an interactive front-end to the account. The client is assumed to be granted the `Debit` and the `Canpay` permission.
- To illustrate the handling of illegal code, trying to execute unauthorised operations, we include an "intruder" (Figure 5) without any permission.

6.1 Translation of the example into our model

From the code for the above example, we derive the graph G_{EC} as outlined in the previous section. The result is shown in Figure 6. Although the methods have no representation in the graph, we have clustered the nodes in boxes according to their method of origin. Furthermore, boxes are coloured according to the protection domains to which its nodes belong. The dotted edges are transfer edges (*TG*), while the solid edges are call edges (*CG*), obtained through a class analysis. The three encircled nodes correspond to code executed as privileged.

[3] In the latest version of JDK 1.2, the begin-/endPrivileged pair has been deprecated and replaced by one `doPrivileged` method, which takes the block of code (in the form of an inner class) to be executed in privileged mode. Our technique faithfully reflects this discipline.

```
public class ControlledVar {
  private float var;
  void write(float new) {
    AccessController.checkPermission(Write);
    var = new;
  }
  float read() {
    AccessController.checkPermission(Read);
    return var;
  }
}
```

Figure 2: The system code (*System* domain)

```
public class AccountMan {
  private ControlledVar balance;
  public boolean canpay(float amount) {
    AccessController.checkPermission(Canpay);
    boolean res = false;
    try {
      AccessController.beginPrivileged();
      res = balance.read() > amount;
    } finally {
      AccessController.endPrivileged();
    }
    return res;
  }
  public void debit(float amount) {
    AccessController.checkPermission(Debit);
    if (this.canpay(amount)) {
      try {
        AccessController.beginPrivileged();
        balance.write(balance.read() - amount);
      } finally {
        AccessController.endPrivileged();
      }
    } else ...
  }
}
```

Figure 3: The account manager code (*Provider* domain)

```
    public void spender() {
      float spend = ...;
      if (account.canpay(spend)) {
        account.debit(spend);
      }
      spender();
    }
```

Figure 4: The application code (*Client* domain)

```
    public void clyde() {
      account.debit(50000000);
      clyde();
    }
```

Figure 5: An uncertified application (*Unknown* domain)

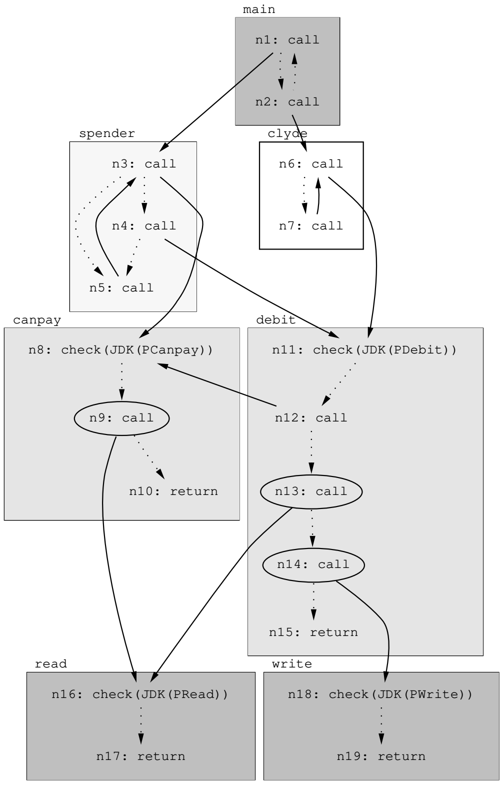

Figure 6: The Derived Graph G_{EC}

The four protection domains partition the set of nodes as follows:

$$\begin{array}{lcl}
\mathit{Client} & = & n_3,\ n_4,\ n_5 \\
\mathit{Provider} & = & n_8, \ldots, n_{15} \\
\mathit{System} & = & n_1,\ n_2,\ n_{16},\ n_{17},\ n_{18},\ n_{19} \\
\mathit{Unknown} & = & n_6,\ n_7
\end{array}$$

Each node in the graph G_{EC} satisfies a property corresponding to its protection domain, plus the property *Priv* if it appears within a privileged section. We use the following conventions for naming node properties: belonging to a Java protection domain *Dom* means satisfying the property D_{Dom}, having a Java permission `Perm` means satisfying P_{Perm}. Furthermore, we write E_{meth} for the property which is true only for nodes of the Java method `meth`. The properties associated with each protection domain are:

$$\begin{array}{lcl}
D_{Client} & = & P_{Debit} \wedge P_{Canpay} \\
D_{Provider} & = & P_{Debit} \wedge P_{Canpay} \wedge \\
 & & P_{Read} \wedge P_{Write} \\
D_{System} & = & P_{Debit} \wedge P_{Canpay} \wedge \\
 & & P_{Read} \wedge P_{Write} \\
D_{Unknown} & = & \mathit{True}
\end{array}$$

In addition, the nodes n_9, n_{13}, and n_{14} are privileged *i.e.*, satisfy the property *Priv*.

6.2 Verification of security properties

As a global statement about the security of the system, we state that all the calls leading to a modification of the balance must possess the `Debit` permission and all the calls leading to disclosure of the balance must possess the `Canpay` permission. This property is expressed as follows in our language:

$$\begin{array}{rl}
\phi = & (\mathbf{G}(\neg E_{write}) \vee (P_{Debit}\ \mathbf{U}\ E_{write})) \ \wedge \\
 & (\mathbf{G}(\neg E_{read}) \vee (P_{Canpay}\ \mathbf{U}\ E_{read})).
\end{array}$$

In other words, either there are no writes (resp. reads) or all the code leading to the write (resp. the read) has the `Debit` (resp. `Canpay`) permission.

It is easy to see that $\mathcal{C}(\phi) = 1$ since there is no **X** in the formula. All the checks occurring in G_{EC} are of the form $\mathtt{check}(\mathit{JDK}(\mathit{Perm}))$ with Perm a permission (thus a node property of complexity 1). $\mathit{JDK}(\phi)$ was defined in Section 3 as:

$$\mathit{JDK}(\phi) = \mathbf{G}((\mathbf{X}(\mathbf{F}\ \mathit{Priv})) \vee \phi)$$

so the complexity of $(\mathit{JDK}(\mathit{Perm}))$ is 2 and $\mathcal{C}(G_{EC}) = 2$. The program can thus be checked against the property ϕ using the associated $\rhd_2$ transition system. Roughly speaking, this means that the loops in the program need to be unrolled only twice to perform the verification[4]. For example, one of the traces that may reach node n_{11} is:

$$
\begin{array}{l}
n_1 \quad n_1 : n_3 \quad n_1 : n_4 \quad n_1 : n_5 \\
n_1 : n_5 : n_3 \quad n_1 : n_5 : n_4 \quad n_1 : n_5 : n_5 \\
n_1 : n_5 : n_5 : n_3 \quad n_1 : n_5 : n_5 : n_4 \\
n_1 : n_5 : n_5 : n_5 : \ldots \\
\qquad : n_1 : n_5 : n_5 : n_5 : n_4 : n_{11}
\end{array}
$$

The consequence of using the 2-limited semantics of G_{EC} (cf. Definition 4.4) is that only the stacks with less than two consecutive n_5-nodes need to be shown to satisfy ϕ. It is straightforward to see that all these remaining stacks satisfy ϕ. For example, the stack $n_1 : n_5 : n_5 : n_4 : n_{11}$ satisfies ϕ, as each of the nodes n_1, n_4, n_5, and n_{11} satisfies both $\neg E_{write}$ and $\neg E_{read}$. Applying the method for each node we obtain that G_{EC}, and thus the program, satisfies the property ϕ.

This program is too defensive though, and can be optimised by eliminating the check in `canpay` because no path leading to node n_8 pass through code without the `Canpay` permission. Likewise, the checks in `write` and `read` are redundant: since all traces leading to n_{18} (the entry node of `write`) must end in a stack of the following form (using regular expressions to express the infinite set)

$$
\begin{array}{l}
n_1 : n_5{}^* : n_4 : n_{14} : n_{18} \mid \\
n_2 : n_7{}^* : n_6 : n_{14} : n_{18}.
\end{array}
$$

Now, the condition checked at n_{18} is $\mathit{JDK}(P_{\mathit{Write}}) = \mathbf{G}((\mathbf{X}(\mathbf{F}\ \mathit{Priv})) \vee P_{\mathit{Write}})$. P_{Write} is trivially satisfied for all the above nodes, except n_6 and n_7, but since the node n_{14} is privileged $\mathbf{X}(\mathbf{F}\ \mathit{Priv})$ is satisfied for node n_6 and n_7 in the possible traces above. Removing the check from the program and reapplying our analysis to the modified program also returns true, which ensures the correctness of the optimisation. A similar analysis shows that the check in `debit` is really necessary since the unauthorised user possesses a link to the method and may potentially try to use it.

[4] In this program, loops occur in the methods `spender` and `clyde`.

7 Related work

In this section we discuss previous work related to the three main aspects of the contribution of this paper: the specification of security properties, the formalisation of Java stack inspection and the reduction of infinite state transition systems.

7.1 Specifying security properties

McLean has proposed to use a trace language for specifying security properties and shows [20] how a simple trace language based on concatenation can be combined with an operational semantics of a first-order procedural language to provide proof of noninterference. Schneider [24] argues that enforceable security properties should be identified with *safety properties* over sets of traces and proposes *security automata* as a formalism for defining security properties. Security automata are a class of Büchi automata that define what are the legal sequences of actions that a system can take. Schneider uses them to monitor an executing system such that an action about to be executed can be prevented if it is deemed illegal by the security automaton. Our linear-time temporal formulae can be represented by automata but the syntactic structuring of the proof of correctness of the reduction would not have been available in the automata setting. It should be emphasised that Schneider employs this formalism in a completely different way since he is interested in dynamic monitoring of executions whereas we are interested in proving statically that a property is verified.

7.2 Formalising Java stack inspection

There is a substantial body of work on formally understanding Java and its virtual machine but the literature on security mechanisms of Java, and especially JDK 1.2 is sparse. Wallach, Balfanz, Dean, and Felten [30] provides an informal but thorough treatment of the security architectures proposed for Java. In a later paper Wallach and Felten [31] formalise the Java stack inspection using a belief logic. The paper is based on the security mechanisms as implemented in Netscape, which can be seen as an extension of the JDK 1.2 mechanisms, allowing to grant specifically named permissions to a piece of code. Granting permission P to code C_1 adds the belief statement $Ok(P)$ to the set of beliefs held in the current stack frame, and calling code C_2 records the beliefs of the earlier stack frames by adding the statement C_1 **says** $Ok(P)$ to the belief set for the stack frame for C_2. In this way, the belief set of a stack frame encodes the control stack together with all the permissions granted to the stack frames in it. Deciding whether $Ok(P)$ holds at a given stack frame is done by a decision procedure that searches the stack frame for a statement C_1 **says** $C_2 \ldots C_n$ **says** $Ok(P)$ where all of the C_i have the permission $Ok(P)$.

The belief logic of Wallach and Felten has its origin in a calculus of principals proposed by Abadi *et al.* [1]. Despite this previous application to the modelling of access control security, it is not evident that this is the ideal formalism for reasoning about stack inspection and [31] does not argue this point. The logic does complicate the correctness proof—indeed the authors can only conjecture that their decision procedure is complete for the logic. Furthermore, its correctness is only argued with respect to an informal operational semantics. It would be interesting to use an operational semantics in the style that we have presented here to prove that this models exactly Java stack inspection. [5]

7.3 Reduction of infinite-state transition systems

We can distinguish two main approaches to the reduction of infinite-state transition systems. The first one was proposed by Wolper [33] who shows that for data-independent reactive programs (*i.e.* programs that only move data around without performing any operations on them) over infinite data domains, it is possible to reduce an infinitary propositional LTL property to a finite property expressed over a finite data domain. The data independence of the program guarantees that validity of the reduced property implies validity of the original property. Jonsson and Parrow [15] show that bisimulation between infinite-state CCS terms is decidable if these terms are data-independent and have finite-state control components. Dam [6] proves that model checking a modal mu-calculus property over an infinite-state pi-calculus agent only needs to explore a finite (property-dependent) part of the state space, *provided* that the agent has finite control (no parallel composition in recursively defined processes). In comparison, when these works *assume* that the programs to be checked have a finite number of control states, we *prove* that for a given property the infinite set of control states (stacks) can be reduced to a finite set without changing the validity of the property to be checked. Furthermore, these works do not apply directly to our framework due to the specific **check** operations of our model and the two-level nature of our logic.

The second approach does not impose restrictions on the programs to be verified but considers the preservation of a whole fragment of the logic used, and provides properties that an abstraction must satisfy in order for model checking on the reduced model to be complete. Loiseaux et al. [18] provide criteria for deciding when an abstraction on the states in a transition system preserves formulae in particular fragments of the mu-calculus. More precisely, they show how an abstraction function on the set of states (formalised as a Galois connection) can be used to obtain an abstract transition system with the property that every formula in the $\Box$-fragment of the modal mu-calculus can be checked on the abstracted system. Cleaveland, Iyer and Yankelevich [5] introduce *democratic Kripke structures* that can be obtained from ordinary Kripke structures by replacing the transition relation by two transition relations: a liberal (overestimating the possible transitions) and a conservative (underestimating the possible transitions). These permit to obtain a safe checking algorithm for CTL*-formulae by using the liberal relation when checking universal path properties and the conservative relation when verifying existential properties. The same idea underlies the work of Dams, Gerth and Grumberg [7] that present an abstract interpretation-based framework for reducing transition systems while preserving validity of the full mu-calculus.

The main departure of our approach with respect to this second approach is that we do not

[5] A step towards this is taken in chapter 7 of Wallach's thesis [29].

perform any abstraction to transform an infinite system into a finite one. We propose instead to limit the original system by considering only a finite subset of states and transitions. We believe that the main advantages of our approach are its simplicity and its flexibility. It is simple because no abstraction and concretisation functions are required. It is flexible because the finite system is tailored to the property to be checked. Furthermore the complexity measure for properties depends on the number of embedded **X** operators, which means that most common properties indeed have a very small complexity.

8 Limitations and further work

We have presented a security-related program model and a formalism for specifying structural security properties. As shown in Section 3, the framework is expressive enough to specify a variety of well-known security properties; in particular, it can express the sandbox model of the original Java security architecture as well as its extension based on stack inspection. Furthermore, we have proposed a method to reduce the semantics of a program to a finite transition system which depends on the property to be checked. To the best of our knowledge, this is the first time that methods from static analysis have been applied to the verification of global security properties (as opposed to specific properties like typing or information-flow). We believe that our framework is a first step towards bridging the gap between abstract specifications of global security properties and their implementations using the security-checking facilities of a programming language. There are several directions in which this work should be extended:

- The results described here are limited to security properties expressed in terms of the control flow and the call graph of the program. Hence, there are a number of properties that cannot be formalised in this setting. Among these are properties concerning the flow of classified information [9, 28] and the detection of covert channels [16]. One direction in which we intend to extend this work is to include data flow as well as control flow information in our abstract model of programs. This would allow us to lift the restriction to structural properties, but the verification algorithm has to be extended to deal with data flow dependencies.

- Some properties are more naturally expressed in terms of the complete execution trace (rather than embedded calls). This is the case for example with the Chinese wall property [4] that dynamically prohibits certain accesses depending on accesses performed in the past. We are examining how to extend the reduction underlying the verification technique presented here to incorporate this kind of properties.

- The program model assumes that all methods are available from the beginning. This means that dynamic class loading can be handled only by re-analysing every time a new class is loaded. It would be desirable to modularise the verification such that it can be done in an incremental fashion. Modular control-flow analysis is needed for this, and some progress has been done in this area [2] but this remains largely an open area of research. The key insight to achieve this goal could be to derive a set of constraints that are required to ensure a given property (rather than a yes/no answer).

References

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Trans. on Prog. Lang. and Systems*, 15:706–734, 1993.

[2] A. Banerjee. A modular, polyvariant and type-based closure analysis. In *Proc. ACM Int. Conf. on Functional Programming (ICFP'97)*, pages 1–10. ACM Press, 1997.

[3] P. Bertelsen. Semantics of Java byte code. Technical report, Department of Information Technology, Technical University of Denmark, March 1997.

[4] D. Brewer and M. Nash. The Chinese wall security policy. In *Proc. of IEEE Symp. on Security and Privacy*, pages 206–211. IEEE Press, 1989.

[5] R. Cleaveland, P. Iyer, and D. Yankelievich. Optimality in abstractions of model checking. In A. Mycroft, editor, *Proc. of Int. Static Analysis Symposium (SAS'95)*, pages 51–63. Springer LNCS vol. 983, 1995.

[6] M. Dam. Model checking mobile processes. *Inf. and Comp.*, 129:35–51, 1996.

[7] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. on Prog. Lang. and Systems*, 19:253–291, 1997.

[8] D. Dean. The security of static typing with dynamic linking. In *Proc. of 4th ACM Conf. on Computer and Communications Security*, pages 18–27. ACM Press, 1997.

[9] D. Denning and P. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, 1977.

[10] L. Gong. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, December 1997.

[11] I. Gray and S. Manson. *The Audit Process*. Van Nostrand Reinhold (International), London, 1989.

[12] D. Grove, G. Furrow, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proc. of Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, 1997.

[13] JavaSoft. Java development kit 1.2. URL: `http://www.javasoft.com/products/jdk/1.2`, 1998.

[14] T. Jensen, D. Le Métayer, and T. Thorn. Security and dynamic class loading in Java: A formalisation. In *Proc. of the 1998 IEEE International Conference on Computer Languages*, pages 4–15, May 1998.

[15] B. Jonsson and J. Parrow. Deciding bisimulation equivalence for a class of non-finite-state programs. *Inf. and Comp.*, 107:272–302, 1993.

[16] B. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10), 1973.

[17] X. Leroy and F. Rouaix. Security properties of typed applets. In *Proc. of 25th ACM Symposium on Principles of Programming Languages*, pages 391–403. ACM Press, 1998.

[18] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, 1995.

[19] General Magic. The Telescript home page. URL: `http://www.genmagic.com/Telescript`, 1996.

[20] J. McLean. Proving noninterference and functional correctness using traces. *J. of Computer Security*, 1:37–57, 1992.

[21] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

[22] Z. Qian. A formal specification of Java virtual machine instructions. Technical report, Universität Bremen, 1997. URL: `http://www.informatik.uni-bremen.de/~qian/pub-list.html`.

[23] E. Rose and K. H. Rose. Lightweight bytecode verification. In *Formal Underpinnings of Java (an OOPSLA workshop)*, Vancouver, BC, Canada, 1998. ACM.

[24] F. Schneider. Enforceable security policies. Technical Report TR98–1664, Dept. of Computer Science, Cornell University, 1998.

[25] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc. of 25th ACM Symp. on Principles of Programming Languages*, pages 149–160. ACM Press, 1998.

[26] T. Thorn. *Vérification de politiques de sécurité par analyse de programmes*. PhD thesis, Université de Rennes I, February 1999.

[27] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. of TAPSOFT'97, Colloqium on Formal Approaches in Software Engineering*, 1997.

[28] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. of Computer Security*, 4(3):1–21, 1996.

[29] D. S. Wallach. *A new approach to mobile code security*. PhD thesis, Dept. of Computer Science, Princeton University, January 1999.

[30] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for java. In *16th Symposium on Operating Systems Principles*, pages 116–128, October 1997.

[31] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *1998 IEEE Symposium on Security and Privacy*, May 1998.

[32] G. Winskel. *The formal semantics of programming languages*. Foundations of Computing Series. The MIT Press, 1993.

[33] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. of 13th ACM Symp. on Principles of Programming Languages (POPL'86)*, pages 184–193, 1986.