

Model checking security properties of control flow graphs

Frédéric Besson Thomas Jensen
IRISA/CNRS
Campus de Beaulieu
F-35042 Rennes Cedex
France

Daniel Le Métayer
Trusted Logic
5, rue du Bailliage
F-78000 Versailles
France

Tommy Thorn
BRICS
University of Aarhus
DK-8000 Aarhus C
Denmark

26th February 2001

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. This article introduces a formalism based on a linear-time temporal logic for specifying global security properties pertaining to the control flow of the program, and illustrates 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. We then show how to instantiate the framework to the security architecture of Java 2 based on stack inspection and privileged method calls.

1 Introduction

A number of recently proposed programming languages provide language constructs for enforcing security requirements. Examples include Telescript [23] with its facilities for controlling permissions and resource consumption, and the recent versions of Java [12] 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 [20] who proved that well-typing can be used to guarantee that a program will not corrupt memory. Volpano et al. [35, 36] have devised type systems for ensuring secure information flow within programs. Dean [10], Jensen et al. [16] and Liang and Bracha [21] 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.*, [4, 28, 29, 31]).

Each of these contributions focuses 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 [13] 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 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 article 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 article 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 control flow 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 model checking technique for the verification of this class of security properties. This technique takes as input the control flow graph and the property to verify and produces as output a finite state transition system such that this finite system satisfies the property in question *if and only if* the original program satisfies the property.

The construction of a control flow graph for a given program involves static analysis [2, 14, 25] and will in general only yield a conservative approximation of the real control flow of the program. However, as stated above, the verification method in itself is complete. All verifications can be carried out on the finite state system without running the risk of rejecting a control flow graph whose (possibly infinite) behaviour satisfies the given property. This is useful information for the user to understand why the verification of a property fails. In that case the only option for improvement is increasing the precision of the control flow graph. This might then either lead to the verification succeeding (given that the property holds) or provide control flow information that is sufficiently precise to understand why the property does not hold.

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 abstracted 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):

$$G = (NO, IS, n_0, TG, CG).$$

So *e.g.* a code sequence such as $m_1() ; m_2()$ will result in two nodes n_1 and n_2 , representing the calls $m_1()$ and $m_2()$, 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 m_1 and from n_2 to the node(s) at the beginning of method m_2 . See figure 9 in Section 6 for an example of such a graph.

The components of the flow graph have the following signatures:

$$\begin{aligned} IS & : NO \rightarrow \{\text{call}, \text{return}, \text{check}(\phi)\} \\ n_0 & : NO \\ TG & : NO \rightarrow \mathcal{P}(NO) \\ CG & : NO \rightarrow \mathcal{P}(NO) \end{aligned}$$

The nodes (*NO*) can be seen as program points and n_0 is the entry point of the whole program. If $m \in TG(n)$ (respectively $m \in CG(n)$) we write $n \xrightarrow{TG} m$ (respectively $n \xrightarrow{CG} m$).

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

- An ordinary procedure (or method, or function) call.
- A **return** node.
- A check node, **check** (ϕ), where ϕ is a property on the state of the machine. 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.

This definition of flow graph is liberal and a reasonable translation from programs to graphs would only yield graphs that satisfy some further well-formedness properties. In particular, there will be no transfer edges (\xrightarrow{TG}) coming out of return nodes and it would be reasonable to eliminate a node from the graph if it is not reachable. Similarly, for call edges $n \xrightarrow{CG} n'$, the source n must be a call node for the edge to make sense. However, these well-formedness properties are not required for the verification method to be correct and hence are not imposed here.

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

$$m_1(); \text{ if } \dots \text{ then } m_2() \text{ else } m_3()$$

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 generally not possible to determine statically what method is invoked in a virtual method call (or function call). The call edges CG describe for each call node n a safe approximation of the possible actual methods (or functions) that might be invoked by the virtual method call (or function call). The approximation is safe in the sense that if at any point during execution the call at node n will result in control being transferred to node m then there will be an edge $n \xrightarrow{CG} m$. However, there might also be edges that do not correspond to a call at execution. Such superfluous edges degrades the precision of the control flow graph but do not jeopardise the correctness of the verification method that we describe in the following.

For object oriented languages such as Java (that we will consider in Section 5) the techniques for constructing the approximate control flow graph are based on data flow analysis that for each variable determines the classes of those objects that will be stored in that variable. These techniques are by now well understood and their correctness has been proved (see *e.g.*, the text book by Palsberg and Schwartzbach [26]). The verification technique described in this paper is independent of the particular analysis chosen. For now we just assume that a control flow graph is available and return to the issue of constructing such a graph when discussing the application to Java in Section 5.

2.1 Trace semantics for control flow graphs

The operational semantics of a flow graph is defined as a transition system with a state consisting of a control stack. Formally, the state of the system is an element of the set

$$Stack = NO^*$$

We use the variables s, s_1, s_2, \dots to range over such stacks.

Nodes model 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, m_3 say, whose body contains a node n_3 which is the current point of control. Thus, the top element of the stack is the “current program point” of our execution model, indicating which node to execute next. 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 (cf. the semantic rule for `return` below).

The operational semantics of a graph G is defined by a transition relation \triangleright . Two stacks s_1 and s_2 are related by \triangleright , written $s_1 \triangleright s_2$, if execution can lead directly from s_1 to s_2 . The following three rules constitute an inductive definition [41] of this relation.

Definition 2.1 *The transition relation $\triangleright \subseteq \text{Stack} \times \text{Stack}$. In the following, $s \in \text{Stack}$ and $n, n', m \in \text{NO}$.*

$$\frac{IS(n) = \mathbf{call} \quad n \xrightarrow{CG} m}{s : n \triangleright s : n : m}$$

$$\frac{IS(m) = \mathbf{return} \quad n \xrightarrow{TG} n'}{s : n : m \triangleright s : n'}$$

$$\frac{IS(n) = \mathbf{check}(\phi) \quad s : n \vdash \phi \quad n \xrightarrow{TG} n'}{s : n \triangleright s : n'}$$

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

For verification purposes we are interested in the set of stacks that execution can lead to. Given a program G with a designated start node n_0 , the operational semantics gives rise to the following transition system:

Definition 2.2 *The trace semantics of $G = (\text{NO}, IS, n_0, TG, CG)$ is the set of sequences of stacks (or “execution traces”) reachable from the initial configuration. Formally, the set of execution traces $\llbracket G \rrbracket \subseteq \text{Stack}^*$ is inductively defined by:*

$$\frac{n_0 \xrightarrow{CG} n}{\langle n_0 : n \rangle \in \llbracket G \rrbracket}$$

$$\frac{\langle s_0, \dots, s_n \rangle \in \llbracket G \rrbracket \quad s_n \triangleright s_{n+1}}{\langle s_0, \dots, s_n, s_{n+1} \rangle \in \llbracket G \rrbracket}$$

Note that the elements of $\llbracket G \rrbracket$ are sequences of stacks *i.e.*, sequences of finite sequences of nodes from the flow graph. For technical reasons it is convenient always to operate on stacks with two or more nodes. For this reason, we assume that the graph G is constrained so that $CG(n_0)$ is a singleton $\{n\}$ where n corresponds to the `main` method of the program and we let the initial stack in the trace be the stack $s_0 = n_0 : n$. It is possible to dispense with the artificial entry node n_0 at the expense of slightly complicating the correctness proof of the static analysis defined in Section 4.

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.

Syntax:

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

Semantics: (NP ranges over node predicates, $s \in \mathit{Stack}$, $T \in \mathit{Stack}^*$, $S \subseteq \mathit{Stack}^*$).

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

Figure 1: Language for the definition of security properties

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 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 3.3).

3.1 A formalism for defining security properties

Since the semantics of a graph 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 to be stacks, *i.e.*, finite sequences of nodes. We write predicates on such sequences 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, negation,

a next operator \mathbf{X} and an until operator \mathbf{U} . We lift node predicates NP to predicates on sequences of nodes by stipulating that NP holds of all those sequences for which the first node satisfies NP . Negation means that the negated predicate does not hold of the sequence; in particular $\neg NP$ means that NP is not a property of the first node in the sequence. The formal semantics of the temporal logic formulae is given in Figure 1 and explained in Section 3.2 below.

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 $\mathbf{G}\phi \equiv \phi \mathbf{U} \textit{False}$ 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.

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 [30] 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

$$\phi \text{ holds for all stacks in the trace.}$$

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 .

3.2 Semantics of security properties

In Figure 1 we define the satisfaction relation $s \vdash \phi$ that formalises when a sequence s satisfies a predicate ϕ . In the 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 . Thus s^i is the empty sequence if $|s| = i$ and s^i is undefined if $|s| < i$. 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). The negation of a formula ϕ means that ϕ cannot be proved with the rules defining the semantics of the formulae. In particular, the negation of a node predicate, $\neg NP$, holds of all those sequences where NP applied to the first node of the sequence is false. 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 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 (modelled by a graph) satisfies a security property”.

Definition 3.1 *A graph G satisfies a security property ϕ , which is denoted by $G \models \phi$, if and only if $\llbracket G \rrbracket \vdash \phi$.*

This definition is only useful for verifying properties of a *program* P if the graph G_P modelling the program correctly reflects the control flow of P . Formally, we must

require that $\llbracket G \rrbracket$ contains all the possible execution traces of P . This is essentially the correctness criterion of the control flow analysis used for constructing the graph G_P from the program P . We do not consider the issue of proving correctness of control flow analyses in this article but refer the reader to articles specifically related to this topic [2, 14, 17, 25].

3.3 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 [13]. 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:

$$SD = (\neg Critical \ \mathbf{U} \ Manager) \wedge (\neg Critical \ \mathbf{U} \ Accountant).$$

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:

$$SB = \mathbf{G} (Site_S \Rightarrow \mathbf{X} (Site_S \vee Local)).$$

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. In this setting, each piece of code is granted a set of *permissions* to execute certain operations (for example, reading from and writing to a file). If a critical operation op is executed by method m_1 , then m_1 must have permission to do so. Furthermore, if m_1 itself was invoked by method m_2 , then m_2 should *also* have permission to execute op . In general, the stack inspection policy imposes that an operation op can be executed only if all the code that leads to the execution of op has the corresponding right. Operationally speaking, this amounts to examining the call stack to check that all the methods on the call stack have permission to perform the operation in question. This policy prevents code from performing an operation on behalf of an unauthorised code.

Requiring that all callers have a specific permission is in certain cases considered too restrictive and can be circumvented by designating certain parts of the code as “privileged”. Marking a method call, $m_2()$ say, in the body of method m_1 as privileged means that all callers (direct or indirect) of m_1 will be given the permissions held by m_1 as long as the call $m_2()$ is executing. For example, an operation in m_2 ’s body is executed as soon as the methods m_1 and m_2 have the permission to do so—the callers of m_1 do not need to have this permission. In a sense, m_1 takes sole responsibility for what happens when the method call $m_2()$ is executed.

Stack inspection for a particular permission P in the presence of permissions and privileged code can then be described operationally as follows. Examine the stack starting from its top (which corresponds to the method currently executing), performing the following checks:

1. if the stack top does not have the permission P , the stack inspection stops and returns failure,
2. if the stack top has the permission P *and* is marked as privileged then stack inspection stops and returns success,
3. otherwise, if the stack top has the permission P but is not privileged, pop the top element of the stack and continue the stack inspection on the remaining stack.

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

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

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 . 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 . 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 .

There is a subtle difference in the stack inspection used by JDK and Internet Explorer (IE) on one hand and Netscape on the other [40]. The difference only manifests itself on stacks in which all stack elements satisfy the property ϕ but none of them are privileged. JDK and IE accepts such a stack (and $JDK(\phi)$ is true because ϕ holds globally). Netscape rejects such a stack. Thus, the Netscape stack inspection policy can be reformulated as: there must exist a privileged call such that the code containing the call and all methods invoked (directly or indirectly) by that call have the property ϕ . In terms of stacks this means that there must exist a privileged node in the stack such that that node and all nodes higher up in the control stack have permission ϕ . In our formalism, this can be expressed in two ways. One solution is to add to the JDK policy the extra requirement that there exists a privileged node in the stack. A more compact formula expressing the same is:

$$Netscape(\phi) = \mathbf{F} (Priv \wedge \mathbf{G} \phi).$$

which states that there must exist a node satisfying $Priv$ such that the node itself and all the following nodes satisfy the property 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 Verification

In this section we present a method for verifying that a program (abstracted by a control flow graph, as defined in Section 2) satisfies a given security property. Designing a mechanical verification method is complicated by the fact that the operational semantics of a control flow graph 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(γ_i)` nodes 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 system with respect to a given property.

The core of the verification technique is a calculation of the set of reachable states of an abstraction of the infinite transition system. The abstract transition system is obtained by partitioning the infinite state-space into a finite number of equivalence classes according to the global security property ϕ to verify and each of the properties γ_i from the check nodes in the program's control flow graph. The partitioning is defined by an equivalence relation on stacks that equates two stacks if they satisfy the same set of properties among the properties $\phi, \gamma_1, \dots, \gamma_n$. Thus, by construction, this results in a finite number of equivalence classes. The main theorem to be proved here states that a property holds of the original, unabstracted system if and only if it holds of the finite, abstracted system. Hence reasoning with the finite system suffices to decide whether the property holds.

4.1 Finite automata representation of properties

The verification technique is based on a result due to Vardi and Wolper (see *e.g.*, [34, 33]) which states that there exists a translation from formulae of linear temporal logic to deterministic finite-state automata such that a string satisfies a formula if and only if the string is accepted by the corresponding automaton. The Vardi-Wolper translation deals with temporal logics over infinite strings which are translated into Büchi automata. Here, we interpret the formulae over finite strings. This changes the acceptance conditions for the automata slightly but the translation technique remains the same. Since the following results only depend on the existence of such a translation, and not on how it is defined, we omit a more detailed description and refer to [33]. Examples of formulae and their translation into automata can be found in Figure 10 and Figure 11.

The automata play a central role in the verification algorithm. The algorithm translates the global property ϕ and all locally checked properties γ_i into automata, written A_ϕ, A_{γ_i} . It then proceeds by following all possible paths in the control flow graph, letting the automata evolve simultaneously. When reaching a `check`(γ_i) node it is then immediate to decide whether γ_i holds by checking whether the corresponding automaton A_{γ_i} is in an accepting state. Similarly, the property ϕ holds globally if the automaton A_ϕ is in an accepting state all the time.

The following definitions fix the notation used for automata.

Definition 4.1 *A deterministic finite state automaton A is a quintuple $(Q, \Sigma, F, \delta, q_0)$ where Q is a finite set of states ranged over by $q, q_i \dots$, Σ is a finite alphabet ranged over by a, a_i, \dots , $F \subseteq Q$ is the set of final states, δ_ϕ the transition function, and q_0 the initial state.*

A is overloaded to denote the function

$$A : \Sigma^* \rightarrow Q$$

that maps each string $w \in \Sigma^$ to the state reached by the the automaton after reading w . Let a_i denote the i th element of a string σ . The function A is defined by*

$$\begin{aligned} A(a_0 : \dots : a_j) &= \delta(A(a_0 : \dots : a_{j-1}), a_j) \\ A(a_0) &= \delta(q_0, a_0) \end{aligned}$$

Notice that the function A is well defined because the automaton A is deterministic.

An automaton recognising the set of strings satisfying a linear temporal logic formula ψ will be written A_ψ . We stress that this automaton is not uniquely determined but we assume for the rest of the article that a particular translation for each property is chosen.

Definition 4.2 *Let ψ be a linear time temporal logic formula as defined in Figure 1. We use $A_\psi = (Q^\psi, \Sigma, F^\psi, \delta^\psi, q_0^\psi)$ to denote a deterministic finite state automaton that accepts the set of strings satisfying the property ψ . Formally, A_ψ must satisfy that for all $w \in \Sigma^*$,*

$$w \vdash \psi \quad \text{iff} \quad A_\psi(w) \in F^\psi$$

We extend the notation to tuples $\Gamma = (\psi_1, \dots, \psi_k, \phi)$ of properties. Formally, A_Γ is defined to be the product automaton $A_{\psi_1} \times \dots \times A_{\psi_k} \times A_\phi$. For w a finite string we obtain the associated function

$$\begin{aligned} A_\Gamma & : \Sigma^* \rightarrow (Q_1 \times \dots \times Q_n) \\ A_\Gamma(w) & = (A_{\psi_1}(w), \dots, A_{\psi_n}(w)). \end{aligned}$$

where Q_i is the set of states of the automaton A_{ψ_i} .

4.2 The equivalence relation on stacks

Next, we define an equivalence relation on stacks that partitions the set of stacks into a finite number of equivalence classes. The equivalence relation is based on the automata representation of security properties and the finiteness of the set of equivalence classes will form the basis of a decision algorithm for security properties, outlined in Section 4.3. The key idea behind the equivalence relation is that equivalent stacks will have the same behaviour against $\text{check}(\gamma_i)$ statements and the global property ϕ . Concretely, equivalent stacks will take each of the automata corresponding to γ_i and ψ into the same state.

This equivalence is refined in order to incorporate some additional control flow information into the equivalence classes. In order to be able to match `call` and `return` statements, we require that the two top elements of the stacks must be identical in order for two stacks to be equivalent.

Definition 4.3 *Let $\Gamma = (\gamma_1, \dots, \gamma_k, \phi)$ be a finite tuple of properties where ϕ is the global safety property to ensure and γ_i the safety property associated with the i^{th} check statement. The equivalence relation $\sim \subseteq \text{Stack} \times \text{Stack}$ is defined as follows:*

$$s:m:n \sim s':m:n \quad \text{iff} \quad A_\Gamma(s:m) = A_\Gamma(s':m)$$

An essential property of the equivalence relation is the finite symbolic representation of an equivalence class as a triple

$$(M, m, n) \in (Q_{\gamma_1} \times \dots \times Q_{\gamma_k} \times Q_\phi) \times \text{Node} \times \text{Node}.$$

The number of equivalence classes is finite because:

- the number of nodes of the flow graph is finite;
- the number of safety properties (properties in check nodes and the global security property to verify) is finite;
- and the number of states of each corresponding automaton is finite.

Definition 4.4 *The stack $s = s':m:n$ belongs to the equivalence class $(A_\Gamma(s' : m), m, n)$. We write $[s]$ for the equivalence class of s .*

Intuitively, a triple (M, m, n) contains the following information. Execution is at node n which belongs to a method invoked at node m . At the moment of invocation at node m , the control stack (of form $s':m$) satisfied that

$$A_\Gamma(s':m) = M.$$

M is a tuple of automata states where a component is an accepting state if and only if the stack $s:m$ satisfies the corresponding security property. Hence, an equivalence class (M, m, n) satisfies the global property ϕ if the component of M corresponding to the A_ϕ automaton reaches a final state after executing n . We shall overload the symbol \vdash to mean both that a stack and an equivalence class satisfy a property.

Definition 4.5 For a given graph G and a global property ϕ (with associated automaton A_ϕ), we write

$$(M, m, n) \vdash \phi$$

when

$$M = (q^{\gamma_1}, \dots, q^{\gamma_k}, q^\phi) \quad \text{and} \quad \delta_\phi(q^\phi, n) \in F_\phi.$$

Similarly, an equivalence class (M, m, n) satisfies a property γ_i if the i^{th} component of M corresponding to the A_{γ_i} automaton reaches a final state after executing n .

Lemma 4.6

$$[s] \vdash \phi \Leftrightarrow s \vdash \phi.$$

Proof Let $s = s':m:n$ be a string and $[s] = (M, m, n)$ its equivalence class. By definition, $M = A_\Gamma(s':m) = (q^{\gamma_1}, \dots, q^{\gamma_k}, q^\phi)$. We have

$$A_\phi(s':m:n) = \delta_\phi(A_\phi(s':m), n) = \delta_\phi(q^\phi, n).$$

It follows that $s \vdash \phi$ if and only if $A_\phi(s':m:n) \in F_\phi$ if and only if $\delta_\phi(q^\phi, n) \in F_\phi$ if and only if $[s] \vdash \phi$ and Lemma 4.6 is verified.

4.3 Reachability Analysis

We now define a set $\llbracket G \rrbracket^\#$ of “reachable” equivalence classes that is used to decide whether all the reachable stacks in the program modelled by G satisfy the global property ϕ . The set $\llbracket G \rrbracket^\#$ is defined inductively by the rules described in Fig 2. Here, n_0 is the initial node of the graph G and n the unique node called from the initial node n_0 . Since $n_0:n$ is the first reachable stack, its equivalence class $[n_0:n]$ is the initially reachable equivalence class. Its symbolic representation is:

$$(\delta((q_0^{\gamma_1}, \dots, q_0^{\gamma_k}, q_0^\phi), n_0), n_0, n)$$

Because the set of equivalence classes is finite, it is decidable whether an equivalence class belongs to $\llbracket G \rrbracket^\#$. Together with the following theorem this provides a procedure for deciding whether a program satisfies a global property.

Theorem 4.7

$$\llbracket G \rrbracket \vdash \phi \Leftrightarrow \forall E \in \llbracket G \rrbracket^\#. E \vdash \phi$$

$$\begin{array}{c}
\frac{n_0 \xrightarrow{CG} n}{(\delta((q_0^{\gamma_1}, \dots, q_0^{\gamma_k}, q_0^\phi), n_0), n_0, n) \in \llbracket G \rrbracket^\#} \\
\\
\frac{IS(n) = \text{call } n \xrightarrow{CG} m \quad (P, p, n) \in \llbracket G \rrbracket^\#}{(\delta(P, n), n, m) \in \llbracket G \rrbracket^\#} \\
\\
\frac{IS(m) = \text{return } n \xrightarrow{TG} n' \quad (P, p, n) \in \llbracket G \rrbracket^\# \quad \delta(P, n) = N \quad (N, n, m) \in \llbracket G \rrbracket^\#}{(P, p, n') \in \llbracket G \rrbracket^\#} \\
\\
\frac{IS(n) = \text{check}(\gamma_i) \quad n \xrightarrow{TG} n' \quad (P, p, n) \in \llbracket G \rrbracket^\# \quad (P, p, n) \vdash \gamma_i}{(P, p, n') \in \llbracket G \rrbracket^\#}
\end{array}$$

Figure 2: Inductive definition of the set of reachable equivalence classes

Proof: The proof is divided into two parts that can be viewed as a correctness (the “ \Leftarrow ”) and a completeness (the “ \Rightarrow ”) part of the analysis. Section 4.4 shows that the “abstract” transition system on equivalence classes is a safe approximation of the concrete transition system while Section 4.5 shows that the set of reachable equivalence classes covers sufficiently many stacks to account for all possible behaviour of the program with respect to the global property ϕ . Formally, we prove the following two properties:

$$s_0 \triangleright^* s \Rightarrow [s] \in \llbracket G \rrbracket^\# \quad (1)$$

$$E \in \llbracket G \rrbracket^\# \Rightarrow \exists s' \in E. s_0 \triangleright^* s' \quad (2)$$

We can then prove the theorem as follows. For correctness, assume that all equivalence classes $E \in \llbracket G \rrbracket^\#$ satisfies $E \vdash \phi$ and let $s \in \llbracket G \rrbracket$ i.e., $s_0 \triangleright^* s$. From (1) we get that $[s] \in \llbracket G \rrbracket^\#$ and hence that $[s] \vdash \phi$. Lemma 4.6 then implies that $s \vdash \phi$. For completeness, assume that $\llbracket G \rrbracket \vdash \phi$ i.e., that for all s such that $s_0 \triangleright^* s$ we have $s \vdash \phi$. Let E be an equivalence class in $\llbracket G \rrbracket^\#$. By (2) there exists a $s' \in E$ such that $s_0 \triangleright^* s'$. By assumption, $s' \vdash \phi$ and Lemma 4.6 then implies that $E = [s'] \vdash \phi$.

4.4 Correctness

We prove that the abstract transition system is a safe approximation of the concrete one in the sense that for all reachable stacks s , the corresponding equivalence class $[s]$ belongs to $\llbracket G \rrbracket^\#$.

Lemma 4.8

$$s_0 \triangleright^* s \Rightarrow [s] \in \llbracket G \rrbracket^\#$$

We prove the lemma by induction over the derivation length. Suppose that $s_0 \triangleright^k s_k \Rightarrow [s] \in \llbracket G \rrbracket^\#$; we show that if $s_0 \triangleright^k s_k \triangleright s_{k+1}$ then $[s_{k+1}] \in \llbracket G \rrbracket^\#$.

The transition $s_k \triangleright s_{k+1}$ is determined by the type of node on top of s_k . We consider each possibility in turn.

call node

$$IS(n) = \mathbf{call} \quad \text{and} \quad n \xrightarrow{CG} m.$$

Suppose that $s_0 \triangleright^k s_k = s':p:n$. By induction hypothesis, $[s_k] = (P, p, n) \in \llbracket G \rrbracket^\#$ where $P = A_\Gamma(s':p)$. By the rule for **call** nodes in the operational semantics (Definition 2.1), $s_k \triangleright s':p:n:m$. Similarly, by the rule for **call** nodes in the definition of $\llbracket G \rrbracket^\#$ (Figure 2) we have $E = (\delta(P, n), n, m) \in \llbracket G \rrbracket^\#$. Furthermore,

$$E = (\delta(A_\Gamma(s':p), n), n, m) = (A_\Gamma(s':p:n), n, m) = [s':p:n:m]$$

Hence, $[s':p:n:m] \in \llbracket G \rrbracket^\#$ and the property holds for **call** nodes.

return node

$$IS(m) = \mathbf{return} \quad n \xrightarrow{TG} n'$$

Suppose that $s_0 \triangleright^k s_k = s:n:m$. It is straightforward to see that the transition relation \triangleright is *prefix-closed* in the sense that for all prefixes w of s_k there exists $l \leq k$ such that $s_0 \triangleright^l w$. It follows that $s_0 \triangleright^l r = s:n = s':p:n$ with $l \leq k$. By applying twice the induction hypothesis, we obtain $[s_k] = (N, n, m) \in \llbracket G \rrbracket^\#$ and $[r] = (P, p, n) \in \llbracket G \rrbracket^\#$ where $N = A_\Gamma(s':p:n)$ and $P = A_\Gamma(s':p)$. By the **return** rule, $s_k \triangleright s':p:n'$. Moreover, the precondition $\delta(P, n) = A_\Gamma(s':p:n) = N$ holds and by the **return** rule, $E = (P, p, n') \in \llbracket G \rrbracket^\#$. By definition of the equivalence relation,

$$[s':p:n'] = (A_\Gamma(s':p), p, n') = (P, p, n') = E.$$

Hence, $[s':p:n'] \in \llbracket G \rrbracket^\#$ and the property holds for **return** nodes.

check node

$$IS(n) = \mathbf{check}(\gamma_i) \quad s:n \vdash \gamma_i \quad n \xrightarrow{TG} n'$$

Suppose that $s_0 \triangleright^k s_k = s:n = s':p:n$. By induction hypothesis, $[s_k] = (P, p, n) \in \llbracket G \rrbracket^\#$ where $P = A_\Gamma(s':p)$. From Lemma 4.6, we have $[s_k] \vdash \gamma_i$. By the **check** rules $s \triangleright s':p:n'$ and $(P, p, n') \in \llbracket G \rrbracket^\#$. Since $(P, p, n') = [s':p:n']$ we have $[s':p:n'] \in \llbracket G \rrbracket^\#$ and the property holds for **check** nodes.

4.5 Completeness

The completeness part of Theorem 4.7 states that for each reachable equivalence class, there exists at least one reachable representative:

$$E \in \llbracket G \rrbracket^\# \Rightarrow \exists t \in E. s_0 \triangleright^* t$$

The proof is given in Section 4.5.2. In order to prove this result, we need the following lemma. Intuitively, it states that method calls present in the set of abstract states correspond to calls at the concrete level.

Lemma 4.9

$$\left. \begin{array}{l} (P, p, n) \in \llbracket G \rrbracket^\# \\ (\delta(P, n), n, m) \in \llbracket G \rrbracket^\# \\ s_0 \triangleright^* s:p:n \in (P, p, n) \end{array} \right\} \Rightarrow s_0 \triangleright^* s:p:n:m$$

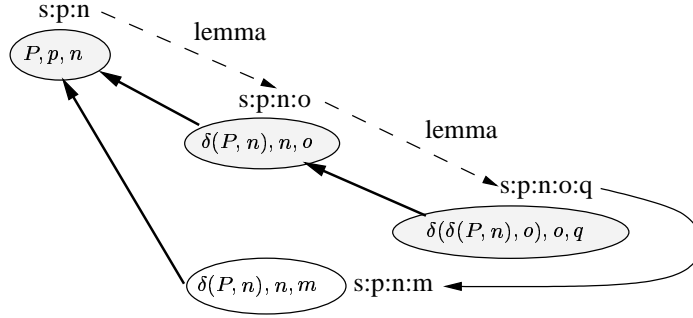


Figure 3: Lemma 4.9 and `return` nodes

4.5.1 Proof

The proof of the lemma is by induction over the deduction of $(\delta(P, n), n, m) \in \llbracket G \rrbracket^\#$. In the following, a figure describes the situation for each of the derivation rules. The conventions are:

- Filled nodes represent equivalence classes given by the induction hypothesis.
- The white node is a newly deduced equivalence class.
- Dashed arrows, labelled by `lemma`, figure out use of induction hypotheses.
- Bold arrows link equivalence classes (P, p, n) and $(\delta(P, n), n, m)$.
- Simple arrows are labelled by transition rules `call`, `return`, `check`.

Base case The proof of the property for the rule concerning the initial node n_0 can be reformulated as follows.

$$\left. \begin{array}{l} (P, p, n_0) \in \llbracket G \rrbracket^\# \\ (\delta(P, n_0), n_0, m) \in \llbracket G \rrbracket^\# \\ s_0 \triangleright^* s:p:n_0 \in (P, p, n_0) \end{array} \right\} \Rightarrow s_0 \triangleright^* s:p:n_0:m$$

This is vacuously true since there is no p such that $(P, p, n_0) \in \llbracket G \rrbracket^\#$. This latter fact is seen by inspecting the rules defining the set $\llbracket G \rrbracket^\#$.

Induction step

call node Let $(\delta(P, n), n, m) \in \llbracket G \rrbracket^\#$ be deduced from a `call` rule. As a result, there exists a node in the flow graph such that:

$$IS(n) = \text{call } n \xrightarrow{CG} m$$

Suppose that $(P, p, n) \in \llbracket G \rrbracket^\#$ such that $s_0 \triangleright^* s:p:n \in (P, p, n)$. By `call` rule, $s_0 \triangleright^* s:p:n:m$. Hence lemma 4.9 holds for `call` nodes.

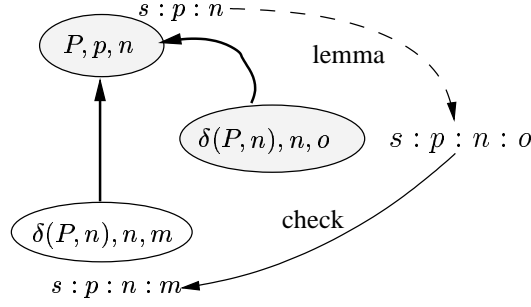


Figure 4: Lemma 4.9 and check nodes

return node Let $(\delta(P, n), n, m) \in \llbracket G \rrbracket^\#$ be deduced from a **return** rule. The hypotheses of this rule yields that there exists a node $IS(q) = \mathbf{return}$, an edge $o \xrightarrow{TG} m$ in the flow-graph and two deductions:

$$(\delta(P, n), n, o) \in \llbracket G \rrbracket^\# \quad (0)$$

$$(\delta(\delta(P, n), o), o, q) \in \llbracket G \rrbracket^\# \quad (1)$$

The other hypotheses from the lemma are:

$$(P, p, n) \in \llbracket G \rrbracket^\# \quad (2)$$

$$s_0 \triangleright^* s:p:n \in (P, p, n) \quad (3)$$

Lemma 4.9 is now applied twice in order to prove the property. First, from (2),(3),(0) we deduce that $s_0 \triangleright^* s:p:n:o$. Second, this fact together with the facts (0) and (1) imply that $s_0 \triangleright^* s:p:n:o:q$. By the **return** rule, $s_0 \triangleright^* s:p:n:m$. Hence Lemma 4.9 holds for **return** nodes.

check node Let $(\delta(P, n), n, m) \in \llbracket G \rrbracket^\#$ be deduced from a **check** rule. Thus there exists a node $IS(o) = \mathbf{check}(\gamma_i)$, a transfer edge $o \xrightarrow{TG} m$ and a deduction $(\delta(P, n), n, o) \in \llbracket G \rrbracket^\#(0)$ such that $(\delta(P, n), n, o) \vdash \gamma_i$. The other hypotheses of the lemma are

$$(P, p, n) \in \llbracket G \rrbracket^\# \quad (1)$$

$$s_0 \triangleright^* s:p:n \in (P, p, n) \quad (2).$$

By Lemma 4.9, from hypotheses (1), (2), (0), we deduce that

$$s_0 \triangleright^* s:p:n:o \in (\delta(P, n), n, o).$$

By the construction of the equivalence classes and the fact that $(\delta(P, n), n, o) \vdash \gamma_i$ we get that $s:p:n:o \vdash \gamma_i$. As a result, by the **check** rule, $s:p:n:o \triangleright^* s:p:n:m$. Hence, Lemma 4.9 holds for **check** rules.

4.5.2 Completeness Proof

The completeness proof now proceeds by induction on the derivation of $E \in \llbracket G \rrbracket^\#$.

call node Let $(\delta(P, n), n, m) \in \llbracket G \rrbracket^\#$ be deduced by **call** rule. Hence, there exists a node $IS(n) = \mathbf{call}$ and an edge $n \xrightarrow{CG} m$ belonging to the flow-graph. and a deduction $(P, p, n) \in \llbracket G \rrbracket^\#$. By induction hypothesis, there exists a reachable stack $s_0 \triangleright^* s:p:n \in (P, p, n)$. By the **call** rule, $s:p:n \triangleright^* s:p:n:m$. By definition of the equivalence relation

$$[s:p:n:m] = (A_\Gamma(s:p:n), n, m) = (\delta(P, n), n, m)$$

We conclude that the property holds for **call** nodes.

return node Let $(P, p, n') \in \llbracket G \rrbracket^\#$ be deduced by the **return** rule. Then, the following node and edge belong to the flow-graph:

$$IS(m) = \mathbf{return} \quad n \xrightarrow{TG} n'$$

Moreover, there exist two deductions

$$\begin{aligned} (P, p, n) &\in \llbracket G \rrbracket^\# \\ (\delta(P, n), n, m) &\in \llbracket G \rrbracket^\# \end{aligned}$$

and by induction hypothesis a reachable stack $s_0 \triangleright^* s:p:n \in (P, p, n)$. By lemma 4.9, there exists $s_0 \triangleright^* s:p:n:m$ representative of $(\delta(P, n), n, m)$. By the **return** rule $s:p:n:m \triangleright s:p:n'$. By definition of the equivalence relation, $[s:p:n] = (P, p, n)$ and the property holds for **return** nodes.

check node Let $(P, p, n') \in \llbracket G \rrbracket^\#$ be deduced by the **check** rule. The following node and edge belong to the flow-graph:

$$IS(n) = \mathbf{check}(\gamma_i) \quad n \xrightarrow{TG} n'$$

Moreover, there exists a deduction $(P, p, n) \in \llbracket G \rrbracket^\#$ such that $(P, p, n) \vdash \gamma_i$. By induction hypothesis, there exists a reachable stack $s_0 \triangleright^* s:p:n \in (P, p, n)$. Since $(P, p, n) \vdash \gamma_i$ it follows from Lemma 4.6 that $s:p:n \vdash \gamma_i$. As a result, by **check** rule $s:p:n \triangleright s:p:n'$ and by equivalence definition, $[s:p:n'] = (P, p, n')$. We conclude that the property holds for **check** nodes. This concludes the completeness proof.

4.6 Complexity of the verification method

Definition 2 directly translates into an iterative algorithm that calculates $\llbracket G \rrbracket^\#$ by repeatedly applying the inference rules until no new abstract state can be added. The number of iteration steps for constructing $\llbracket G \rrbracket^\#$ is bounded by the size of the set of abstract states. An upper bound on the size of this set can be determined as follows.

Let *Node* be the number of nodes in the control flow graph, *Call* be the number of call nodes in the flow graph and *Check* be the number of check nodes in the program. Let furthermore *GP* be the size of the automaton describing the global property to verify. Then the number of possible (abstract) states is bounded by

$$Calls \times Nodes \times 2^{Check} \times GP.$$

The exponential factor comes from the fact that there are two states in the automaton corresponding to a check node in the program and that each check node gives rise to an automaton that tells whether the given property at that check node is satisfied or not. A first reduction of the state space would be to have one automaton for each check property such that check nodes with the same property share the same automaton. Furthermore, it should be noted that the number *Check* of check nodes is usually much smaller than the size of the program. The example that we present in Section 6 shows that the upper bound given above is not very accurate. It predicts in the order of ten thousand states for the example whereas the real number is twenty-six. Thus even a relatively security-intensive program as the one used in the example (see Figure 9) only explores a relatively small part of the state space.

The number of states in $\llbracket G \rrbracket^\#$ reflects quite accurately the number of different combinations of permissions that different parts of the program have. In the extreme case of all code having the same set of permissions, the exponential factor in the formula above can be replaced by 1 since none of the automata change state. As the program traverses more and more protection domains (and hence the set of held permissions changes more frequently) more and more of the abstract state space will be explored. Thus a program with a rich security structure will be more complex to verify, as one would expect.

5 Application to the Java Development Kit

The Java Development Kit JDK 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 protection 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 [12, 15], enabled with the method call `beginPrivileged` and disabled with the method call `endPrivileged` 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` disap-

pear in our model, but the nodes delimited by the two are recorded as satisfying the property *Priv*).

In the latest version of the JDK (2.0), the `begin/endPrivileged` pair has been deprecated and replaced by the method `doPrivileged`. The `doPrivileged` method is a safer way of designating parts of code as privileged compared to the earlier `begin/endPrivileged` blocks where special care had to be taken to make sure that `endPrivileged` was always called appropriately (notably in the presence of exceptions). A call to `doPrivileged` takes as argument an object of a class that implements the interface `PrivilegedAction`. This interface contains one method, `run`, which is responsible for calling those methods that are to be executed in privileged mode. In order to handle a call `doPrivileged(O)` where `O` is of class `C` which implements the interface `PrivilegedAction`, the initial control flow analysis must represent such a call by a privileged call node (*i.e.*, a call node having the permission *Priv*) that has a call edge to the `run` method of class `C`.

A call to `checkPermission(Perm)` from class `AccessController` can then be modelled by the instruction `check(JDK(Perm))` with *JDK* as defined at the end of Section 3.3.

5.1 Constructing the control flow graph

To obtain the graph corresponding to a Java program, its code is transformed into basic blocks and everything but method calls is abstracted away. As indicated in Section 2, the construction of the call edges in the control flow graph uses a data flow analysis that for each variable finds an over-approximation of the classes of the objects that are being stored in the variable. For each call node n corresponding to a virtual method call of the form $X.m()$ and for each possible class C of the objects stored in X we introduce an edge $n \xrightarrow{CG} m_C$ to the entry nodes of the method named m in C . A number of such analyses have been proposed. The simplest analysis approximates the set of possible classes by all the subclasses of the declared type for the variable. A simple improvement, called rapid type analysis [2] was proposed by Bacon and Sweeney. It consists in intersecting the set of subclasses with an approximation of the set of classes that are actually being instantiated during execution. This analysis can deal with large programs and is generally considered to give acceptable results. These analyses do not consider the data flow of the program. This aspect is taken into account in the constraint based analysis proposed by Palsberg and Schwartzbach [26, 25]. In its basic formulation, the analysis does not take the sequential control flow of the program into account since it only calculates one global approximation for each variable. Thus its precision can be further improved by distinguishing between different *occurrences* of a variable, rendering the analysis flow-sensitive as proposed by Pande and Ryder [27]. A prototype implementation of the verification technique [32] has been developed using the flow-insensitive constraint based analysis, adapted to take the visibility modifiers of Java into account. The next section describes a small example that was analysed automatically by this prototype.

```

1  public class ControlledVar {
2      private float var;
3      void write(float new) {
4          AccessController.checkPermission(Write);
5          var = new;
6      }
7      float read() {
8          AccessController.checkPermission(Read);
9          return var;
10     }
11 }

```

Figure 5: The system code (*System* domain)

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 5) 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 6) 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 7). 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 8) 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 9. 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

```

12 public class AccountMan {
13     private ControlledVar balance;
14     public boolean canpay(float amount) {
15         AccessController.checkPermission(Canpay);
16         boolean res = false;
17         try {
18             AccessController.beginPrivileged();
19             res = balance.read() > amount;
20         } finally {
21             AccessController.endPrivileged();
22         }
23         return res;
24     }
25     public void debit(float amount) {
26         AccessController.checkPermission(Debit);
27         if (this.canpay(amount)) {
28             try {
29                 AccessController.beginPrivileged();
30                 balance.write(balance.read() - amount);
31             } finally {
32                 AccessController.endPrivileged();
33             }
34         } else ...
35     }
36 }

```

Figure 6: The account manager code (*Provider* domain)

```

37   public void spender() {
38       float spend = ...;
39       if (account.canpay(spend)) {
40           account.debit(spend);
41       }
42       spender();
43   }

```

Figure 7: The application code (*Client* domain)

```

44   public void clyde() {
45       account.debit(50000000);
46       clyde();
47   }

```

Figure 8: An uncertified application (*Unknown* domain)

solid edges are call edges (*CG*), obtained through a class analysis. The three encircled nodes correspond to code executed as privileged.

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

$$\begin{aligned}
 \textit{Client} &= n_3, n_4, n_5 \\
 \textit{Provider} &= n_8, \dots, n_{15} \\
 \textit{System} &= n_0, n_1, n_2, n_{16}, n_{17}, n_{18}, n_{19} \\
 \textit{Unknown} &= n_6, n_7
 \end{aligned}$$

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:

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

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

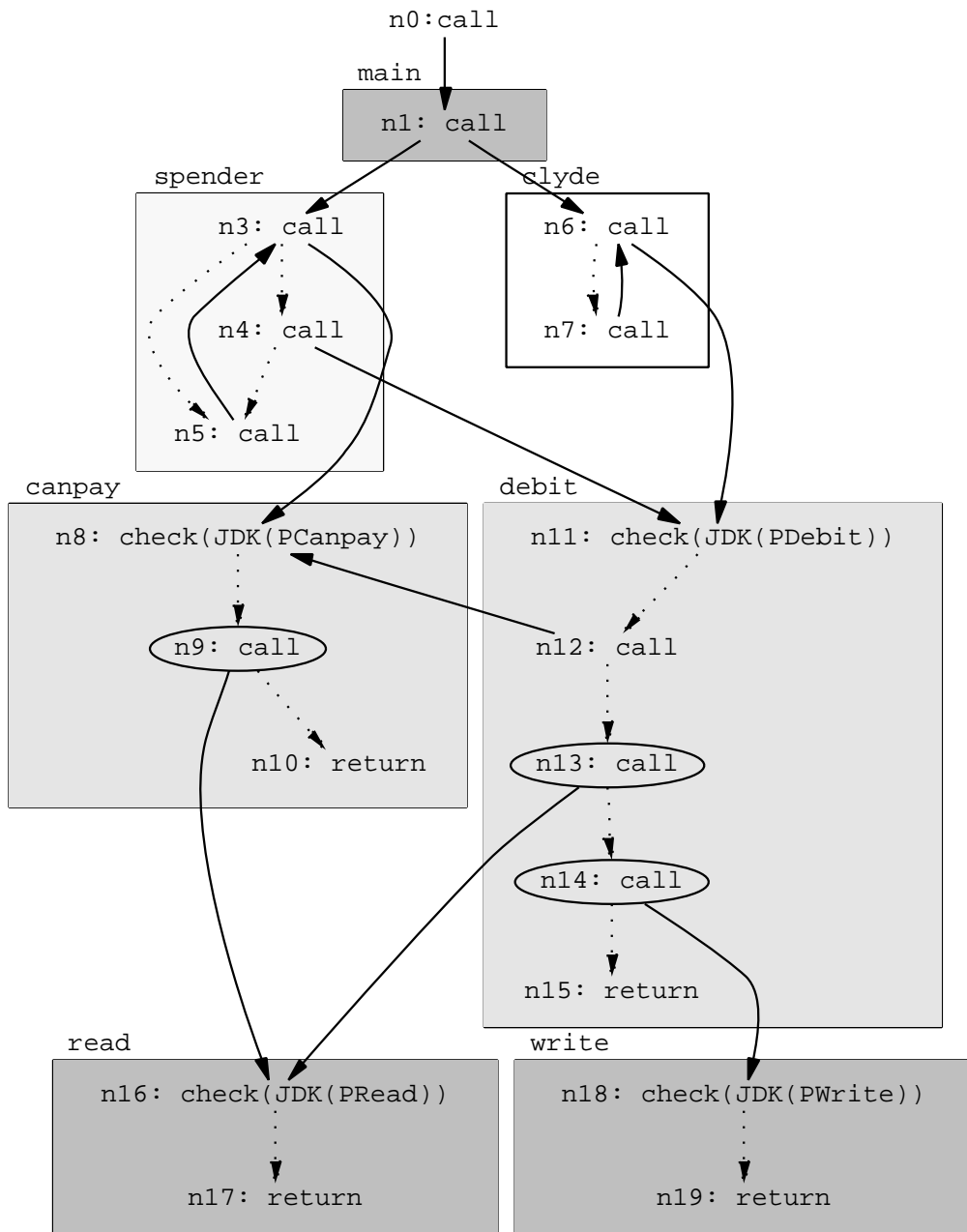


Figure 9: The Derived Graph G_{EC}

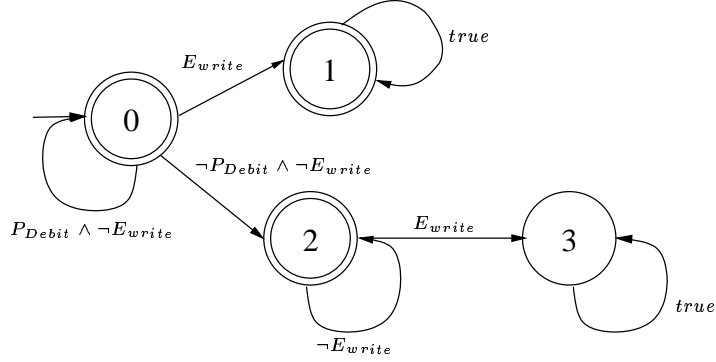


Figure 10: Automaton associated with $(\mathbf{G}(\neg E_{write}) \vee (P_{Debit} \mathbf{U} E_{write}))$

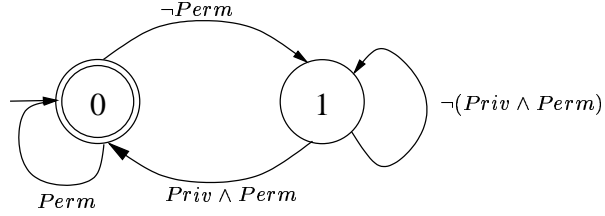


Figure 11: Automaton associated with $JDK(Perm) = \mathbf{G}((\mathbf{X}(\mathbf{F} Priv)) \vee Perm)$

property is expressed as follows in our language:

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

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. The first conjunction of the property is modelled by the four-state deterministic automaton given in Figure 10. All the checks occurring in G_{EC} are of the form $\text{check}(JDK(Perm))$ with $Perm$ a permission. $JDK(Perm)$ was defined in Section 3 as

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

Such a property is modelled by the two-state deterministic automaton in Figure 11.

The equivalence classes in the automaton $\llbracket G_{EC} \rrbracket^\#$ are of the form

$$((s_{CanPay}, s_{Debit}, s_{Read}, s_{Write}, s_{\phi_{write}}, s_{\phi_{read}}), m, n)$$

where *e.g.*, the component s_{CanPay} is a state in the automaton modelling the property $JDK(P_{CanPay})$. For example, the equivalence class

$$((1, 1, 1, 1, 2, 2), n_6, n_{11})$$

belongs to the set $\llbracket G_{EC} \rrbracket^\#$, meaning that it is possible to reach node n_{11} via node n_6 with the automata corresponding to the check nodes in the states (1,1,1,1) and the

automata corresponding to each of the conjuncts of the global security property both in the state 2. All in all, the set $\llbracket G_{EC} \rrbracket^\#$ contains 26 equivalence classes that all satisfy the property ϕ according to Definition 4.5. These equivalence classes are listed in Appendix A.

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)

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

Now, the condition checked at n_{18} is $JDK(P_{Write}) = \mathbf{G}((\mathbf{X}(\mathbf{F} Priv)) \vee P_{Write})$. P_{Write} is trivially satisfied for all the above nodes, except n_6 and n_7 , but since the node n_{14} is privileged $\mathbf{X}(\mathbf{F} Priv)$ is satisfied for node n_6 and n_7 in the possible traces above. Furthermore, an inspection of the equivalence classes reaching node n_{18} shows that this check does not cut any execution path (the $JDK(P_{Write})$ automaton remains in an accepting state). Thus it is safe to optimise the program by removing this check.

Finally, the analysis can show that the check in `debit` is necessary since the unauthorised user possesses a link to the method and may potentially try to use it. Formally, analysing the program without this check will find an equivalence class

$$((0, 2, 2), n_9, n_{16})$$

which violates the property. The stack $n_0; n_1 : n_6 : n_{12} : n_9 : n_{16}$ is a representative of this equivalence class and corresponds to the illegal path $n_0 : n_1 : n_6 : n_{11} : n_{12} : n_8 : n_9 : n_{16}$ arising from the unauthorised user trying to invoke the `debit` method.

7 Related work

7.1 Specifying security properties

Schneider [30] 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. In the work cited above, 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. This is a different usage of the security automata compared to the work reported here since Schneider uses the security automata for the dynamic monitoring of programs whereas we are interested in proving statically that a property is verified.

Two recent articles deal with incorporating security automata into programs. Colcombet and Fradet [7] consider the problem of inserting operations on a security automaton into a program such that the automaton monitors the execution and hence can enforce that the program respects the policy described by the automaton. In particular, they show how to optimise these operations in order to minimise the overhead associated with enforcing the security policy.

Walker [37] proposes a program logic in the shape of a dependent type system for proving that a program instrumented with security automaton-operations and run-time checks respects a given policy. The primary purpose of this logic is to certify programs (in the sense of proof-carrying code [24]) by providing sufficiently many type annotations for a type checker at a foreign host to be able to verify that an imported program respects a given policy. Such a type-based approach is likely to be better suited for the kind of modular verification that is needed for verifying mobile code although some initial work on annotating programs with information for directing a flow analyser indicates that this is a feasible approach to “analysis-carrying code”. [29]. It should be noted, though, that the type inference problem for the type system proposed by Walker still requires to be solved (Walker doesn’t provide an algorithm). In contrast, our flow analysis is implemented by standard iterative fixpoint algorithms.

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 [39] provides an informal but thorough treatment of the security architectures proposed for Java. In a later paper Wallach and Felten [40] 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** $C_2 \dots C_n$ **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 [40] 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 investigate whether it is possible to use an abstract operational semantics in the style that we have presented here to prove that this models exactly Java stack inspection. Using an abstract semantics should make the proof easier compared to using a concrete, complete semantics. A step towards this is taken in chapter 7 of Wallach’s thesis [38]. Wallach’s thesis contains a formal proof of equivalence but it is still with respect to an informal operational semantics.

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 [42] 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 [18] show that bisimulation between infinite-state CCS terms is decidable if these terms are data-independent and have finite-state control components. Dam [8] 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 abstracted to a finite set on which the property can be verified without loss of precision. 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. [22] 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 [6] 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 [9] that present an abstract interpretation-based framework for reducing transition systems while preserving validity of the full mu-calculus.

Compared to the approach reported in this article, the main difference is that these works consider abstractions that preserve whole fragments of formulae. Furthermore, the main emphasis is to provide *principles* for the safe abstraction of transition systems in model checking. Our method, on the other hand, provides a concrete abstraction that is dependent both on the program to be analysed and the property to be verified. While less general, the advantage of this is that we obtain a method that is both sound and complete, whereas the above cited works only obtain abstractions that are sound. It should be noted that the logics studied in the works cited above are more expressive than the logic considered in this article. The extension of our technique to these stronger logics remains to be done.

8 Conclusions

We have presented a security-related program model and a formalism for specifying control flow related 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 fully automatic method for verifying that a particular program satisfies a given security property. The method is based on a reachable-states analysis of an abstraction of the program. The distinctive feature of this verification method is that it is complete for the set of properties considered here. The soundness and completeness of the verification method has been proved formally. Combined with a safe control flow analysis such as rapid type analysis [2] or the constraint based analysis of Palsberg and Schwartzbach [26] this provides a complete proof of the verification technique. We have then shown how this verification method can be applied to analysing programs written using the new Java 2 security architecture. Our work thus contributes to 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 [11, 36] and the detection of covert channels [19]. 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. For Java programs, the control flow analysis is essentially a data flow analysis, so some of this information is already available in that case. This would allow us to lift the restriction to purely control flow related 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 [5] that dynamically prohibits certain accesses depending on accesses performed in the past. It would be worth 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 [3] 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).

Acknowledgements: Thanks are due to Thomas Colcombet for numerous discussions about automata-based model checking.

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] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proc. of OOPSLA'96*, volume 31(10) of *ACM SIGPLAN Notices*, pages 324–341, New York, 1996. ACM Press.
- [3] 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.
- [4] P. Bertelsen. Semantics of Java byte code. Technical report, Department of Information Technology, Technical University of Denmark, March 1997.
- [5] 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.
- [6] 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.
- [7] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. of 27 ACM Symp. on Principles of Programming Languages (POPL'00)*, pages 54–66. ACM Press, 2000.
- [8] M. Dam. Model checking mobile processes. *Inf. and Comp.*, 129:35–51, 1996.
- [9] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. on Prog. Lang. and Systems*, 19:253–291, 1997.
- [10] 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.
- [11] D. Denning and P. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, 1977.
- [12] 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.
- [13] I. Gray and S. Manson. *The Audit Process*. Van Nostrand Reinhold (International), London, 1989.
- [14] 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.

- [15] JavaSoft. Java development kit 1.2. URL: <http://www.javasoft.com/products/jdk/1.2>, 1998.
- [16] 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.
- [17] T. Jensen and F. Spoto. Class analysis of object-oriented programs through abstract interpretation. In F. Honsell, editor, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS 2001)*. Springer LNCS, 2001.
- [18] B. Jonsson and J. Parrow. Deciding bisimulation equivalence for a class of non-finite-state programs. *Inf. and Comp.*, 107:272–302, 1993.
- [19] B. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10), 1973.
- [20] 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.
- [21] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proc. of 13th Annual ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*. ACM Press, 1998.
- [22] 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.
- [23] General Magic. The Telescript home page. URL: <http://www.genmagic.com/Telescript>, 1996.
- [24] G. Necula. Proof-carrying code. In *Proc. of the 24th Symp. on Principles of Programming Languages (POPL'97)*, pages 106–119. ACM Press, 1997.
- [25] J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proc. of OOPSLA'91*, volume 26(11) of *ACM SIGPLAN Notices*, pages 146–161. ACM Press, November 1991.
- [26] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.
- [27] H. Pande and B. Ryder. Data-flow-based virtual function resolution. In *Proc. of 3rd Static Analysis Symposium (SAS'96)*. Springer LNCS vol. 1145, 1996.
- [28] Z. Qian. A formal specification of Java virtual machine instructions. Technical report, Universitt Bremen, 1997. URL: <http://www.informatik.uni-bremen.de/~qian/pub-list.html>.
- [29] E. Rose and K. H. Rose. Lightweight bytecode verification. In *Formal Underpinnings of Java (an OOPSLA workshop)*, Vancouver, BC, Canada, 1998. ACM.

- [30] F. Schneider. Enforceable security policies. Technical Report TR98-1664, Dept. of Computer Science, Cornell University, 1998.
- [31] 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.
- [32] T. Thorn. *Vérification de politiques de sécurité par analyse de programmes*. PhD thesis, Université de Rennes I, February 1999.
- [33] M. Vardi and P. Wolper. Reasoning about infinite computation. *Information and Computation*, 115(1):1–37, 1994.
- [34] M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logic of programs. *Journal of Computer and System Sciences*, 32:183–221, 1986.
- [35] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. of TAPSOFT'97, Colloquium on Formal Approaches in Software Engineering*, 1997.
- [36] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. of Computer Security*, 4(3):1–21, 1996.
- [37] D. Walker. A type system for expressive security policies. In *Proc. of 27 ACM Symp. on Principles of Programming Languages (POPL'00)*, pages 254–267. ACM Press, 2000.
- [38] D. S. Wallach. *A new approach to mobile code security*. PhD thesis, Dept. of Computer Science, Princeton University, January 1999.
- [39] 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.
- [40] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *1998 IEEE Symposium on Security and Privacy*, May 1998.
- [41] G. Winskel. *The formal semantics of programming languages*. Foundations of Computing Series. The MIT Press, 1993.
- [42] 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.

A Analysis of the electronic wallet

The analysis of the electronic wallet example calculates the following set $\llbracket G_{EC} \rrbracket^\#$ of equivalence classes of reachable states:

$((0, 0, 0, 0, 0, 0), n_0, n_1)$	$((0, 0, 0, 0, 0, 0), n_1, n_6),$	$((0, 0, 0, 0, 0, 0), n_1, n_3),$
$((1, 1, 1, 1, 2, 2), n_6, n_{11}),$	$((0, 0, 1, 1, 0, 0), n_3, n_8),$	$((0, 0, 1, 1, 0, 0), n_3, n_9),$
$((0, 0, 0, 0, 0, 0), n_9, n_{16}),$	$((0, 0, 0, 0, 0, 0), n_9, n_{17}),$	$((0, 0, 1, 1, 0, 0), n_3, n_{10}),$
$((0, 0, 0, 0, 0, 0), n_1, n_5),$	$((0, 0, 0, 0, 0, 0), n_1, n_4),$	$((0, 0, 1, 1, 0, 0), n_5, n_3),$
$((0, 0, 1, 1, 0, 0), n_4, n_{11}),$	$((0, 0, 1, 1, 0, 0), n_5, n_5),$	$((0, 0, 1, 1, 0, 0), n_5, n_4),$
$((0, 0, 1, 1, 0, 0), n_4, n_{12}),$	$((0, 0, 1, 1, 0, 0), n_{12}, n_8),$	$((0, 0, 1, 1, 0, 0), n_{12}, n_9),$
$((0, 0, 1, 1, 0, 0), n_{12}, n_{10}),$	$((0, 0, 1, 1, 0, 0), n_4, n_{13}),$	$((0, 0, 0, 0, 0, 0), n_{13}, n_{16}),$
$((0, 0, 0, 0, 0, 0), n_{13}, n_{17}),$	$((0, 0, 1, 1, 0, 0), n_4, n_{14}),$	$((0, 0, 0, 0, 0, 0), n_{14}, n_{18}),$
$((0, 0, 0, 0, 0, 0), n_{14}, n_{19}),$	$((0, 0, 1, 1, 0, 0), n_4, n_{15}),$	