URL: http://www.elsevier.nl/locate/entcs/volume54.html 12 pages

Static Analysis for Stack Inspection

Massimo Bartoletti, Pierpaolo Degano, GianLuigi Ferrari

Dipartimento di Informatica — Università di Pisa, Italy
email: {bartolet, degano, giangi}@di.unipi.it

Abstract

We propose two control flow analyses for the Java bytecode. They safely approximate the set of permissions granted/denied to code at run-time. This static information helps optimizing the implementation of the stack inspection algorithm.

1 Introduction

A main innovation of the Java platform concerns its approach to security: the language comes equipped with constructs and mechanisms for expressing and enforcing security policies. Since the code actually executed is on the form of an intermediate object-oriented language – the bytecode – *bytecode verification* is the basic building block of Java security.

Over the past few years, there has been considerable effort in developing formal models of the Java bytecode verifier. Some authors showed that the problem of bytecode verification can be formally understood and described at static time using type systems [3,4,14]. All the proposals are proved to enjoy the type soundness properties (on the bytecode fragments they consider). Also, the type inference algorithm can be turned into a correct bytecode verifier, see e.g. [2,5,10].

Another crucial aspect of the Java security architecture is the dynamic check of the permissions granted to running code. Roughly, one has to make sure that whenever a principal invokes a certain method, it has the rights to. At run-time, permissions are enforced by *stack inspection*: a permission is granted, provided that it belongs to *all* principals on the call stack. An exception are the so-called *privileged operations*, which are allowed to execute any code granted to their principal, regardless of the calling sequence.

Since the analysis of stack frames may be expensive, the run-time overhead due to stack inspection may grow very high: effective techniques which improve and optimize stack inspection are therefore in order.

In this paper we develop a static analysis which improves run-time checking of permissions. We reduce the number of frames to be examined, while maintaining the same accuracy of the plain stack inspection algorithm. Also,

our analysis may be used for optimizing bytecode, by moving checks where they are actually needed, and by removing redundant ones.

Our approach is based on Control Flow Analysis (CFA) [9], a static technique for predicting safe and computable approximations to the set of values that the objects of a program may assume during its execution. These approximations are then used to analyze properties of programs in a safe manner: if a property holds at static time, then it will always hold at run-time. The vice-versa may not be true: the analysis may "err on the safe side". CFA and other static program analysis techniques are generally more efficient than program verification, and for that reason more approximate, because the focus is on the fully automatic processing of large programs.

Our main technical contribution is the formulation of a couple of control flow analyses over an abstract representation of Java programs. This abstract representation specializes the usual call graph, focussing on permission checks and method invocations (and protection domains), similarly to [8]. Call graphs are given an operational semantics. Essentially, the states that a program can pass through are represented by stacks σ, made of nodes of the call graph, each interpreted as an abstraction of the actual stack frames. The control point is the top n of the stack $\sigma : n$, and a computation step is represented by a transition between stacks, written as $\sigma \rhd \sigma'$.

For each node n our first analysis computes an approximation, i.e. a subset $\delta(n)$ of those permissions that are *denied* to n, in every run leading to n. Similarly, our second analysis computes a subset $\gamma(n)$ of the permissions *granted* to n in every run leading to n. Both analyses are correct with respect to the operational semantics. Suppose that n is a security check of permission P, and that $P \in \delta(n)$ (resp. $P \in \gamma(n)$). Then, whenever there is a computation $[] \rhd \ldots \rhd \sigma : n$, the security check *always* fails (resp. succeeds). The approximations computed by our analyses are then used to reduce the depth at which the stack inspection algorithm stops. When checking privileges towards a permission P, it suffices to reach a frame m such that $P \in \delta(m)$ or $P \in \gamma(m)$. In the first case an `AccessControlException` is raised, while in the second one the check succeeds.

2 Program model

We represent bytecode programs as oriented graphs where only security checks and control flow are made explicit. On them, we base our analyses.

A *call graph* is a triple $G = (N, E, S)$, where:

- N is the set of nodes, including a distinguished element $\bot_N$. Each node $n \in N \setminus \{\bot_N\}$ is associated with a label $\ell(n)$, describing the control flow primitive represented by the node. Labels give rise to three kinds of nodes: `call` nodes, representing method invocation, `return` nodes, which represent return from a method, and `check` nodes, which enforce the access control

policy. Roughly, we can think of a node labelled `check(P)` has having the same meaning of an `AccessController.checkPermission(P)` instruction in the Java language. The distinguished node $\bot_N$ plays the technical role of a single, isolated entry point.

- $E = E_{call} \uplus E_{trans} \uplus E_{entry} \subseteq N \times (N \setminus \{\bot_N\})$ is the set of edges. Edges are split into *call edges* $n \longrightarrow n' \in E_{call}$, modelling inter-procedural flow, and *transfer edges* $n \dashrightarrow n' \in E_{trans}$, which instead correspond to intra-procedural flow. Moreover, we have the set of *entry edges* $\bullet\!\!\longrightarrow n \in E_{entry}$, containing all pairs $(\bot_N, n)$ for $n \in S$. The $\bot_N$ element only appears in entry edges.
- $S \subseteq N \setminus \{\bot_N\}$ is the non-empty set of entry nodes. We assume that a program may have many entry points, as it actually happens with programs designed to be launched both as applets and as stand-alone applications.

In order to give a specification of the access control policy being consistent with the one introduced by the JDK 1.2, we endow each node $n \in N \setminus \{\bot_N\}$ with the following additional information:

- *Permissions(n)*, the set of permissions associated with n. The Java security architecture bounds permissions to whole *protection domains*, that our model does not handle explicitly. We only require that, whenever $n \dashrightarrow n'$, both n and n' carry the same permission set.
- *Priv(n)*, a boolean predicate indicating whether n represents *privileged* code.

In what follows, we assume that all the information above is extracted from the bytecode, e.g. by the constructions presented in [7,8,9].

Throughout the paper we will make use of an example taken from [8], that describes a small e-commerce application. The call graph extracted from the Java program is shown in figure 1 (for more details, we refer the reader to [8]). Circled nodes represent blocks of privileged code. The mapping between protection domains and nodes is illustrated in Fig. 2.

The operational semantics of call graphs is defined by a transition system whose configurations are sequences of nodes, modelling call stacks. The transition relation is defined in Fig. 3 (the definition of the JDK predicate is discussed later on).
We also need a reachability relation $\vdash$ stating when the execution of a program G can lead to a given state:

$$\frac{}{G \vdash []} \qquad\qquad \frac{G \vdash \sigma \qquad \sigma \rhd \sigma'}{G \vdash \sigma'}$$

We say that a state σ is *reachable* by G if and only if $G \vdash \sigma$.

Here, we use a slightly simplified version of the full access control algorithm presented in [6], as we let privileged frames to exploit all of their own permissions. The simplified algorithm performs a top-down scan of the call stack. Each frame in the stack refers to the protection domain containing the class to

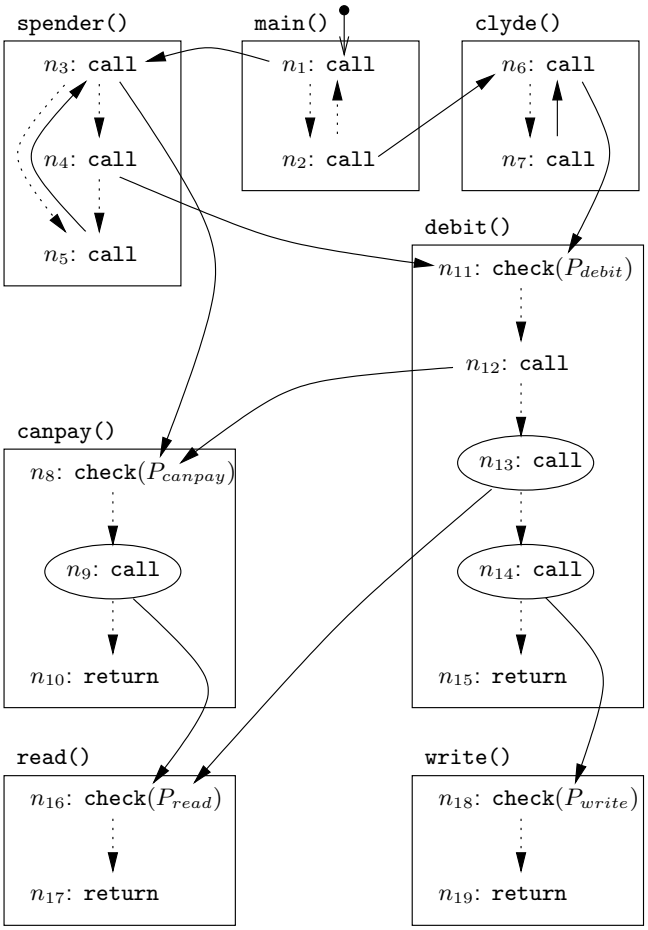


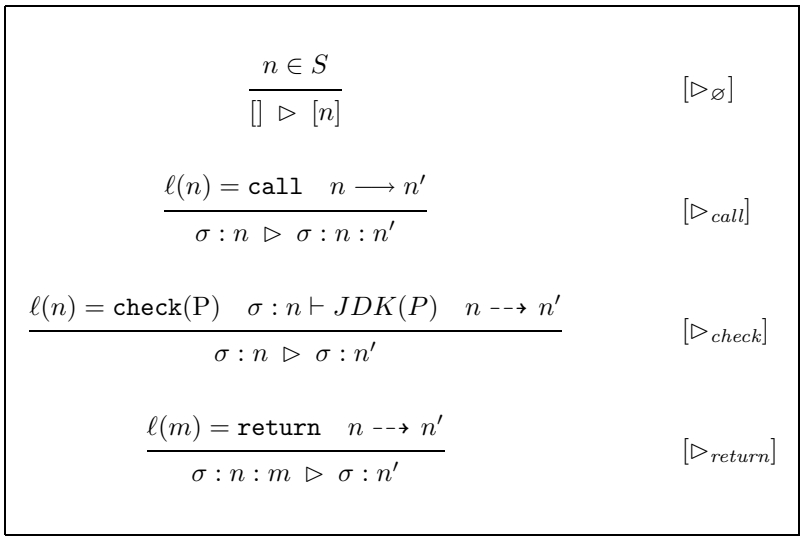
Fig. 1. A call graph.

which the called method belongs. As soon as a frame is found whose protection domain has not the required permission, an `AccessControlException` is raised. The algorithm succeeds when a privileged frame is found that carries the required permission, or when all frames have been visited. A formal specification of this algorithm is given in Fig. 4, that defines JDK.

We stress an important point here. In the JDK 1.2 security architecture, a permission P may be granted to a piece of code, lying inside a protection domain $\mathcal{D}$, even if P does not belong to the permissions explicitly associated

Protection Domain	Methods	Permissions
Client	`spender()`	$\{P_{debit}, P_{canpay}\}$
Unknown	`clyde()`	$\emptyset$
Provider	`canpay()`, `debit()`	$\{P_{debit}, P_{canpay}, P_{read}, P_{write}\}$
System	`main(), read(), write()`	**Permission**

Fig. 2. Protection domains.



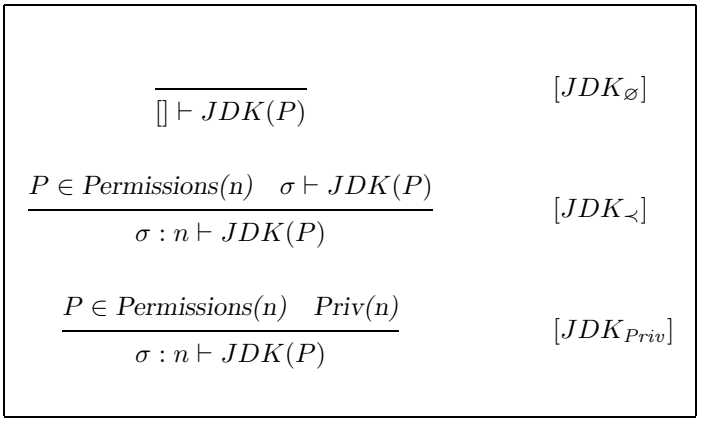
$$\frac{n \in S}{[] \rhd [n]} \quad [\rhd_{\varnothing}]$$

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n'}{\sigma : n \rhd \sigma : n : n'} \quad [\rhd_{call}]$$

$$\frac{\ell(n) = \texttt{check(P)} \quad \sigma : n \vdash JDK(P) \quad n \dashrightarrow n'}{\sigma : n \rhd \sigma : n'} \quad [\rhd_{check}]$$

$$\frac{\ell(m) = \texttt{return} \quad n \dashrightarrow n'}{\sigma : n : m \rhd \sigma : n'} \quad [\rhd_{return}]$$

Fig. 3. Operational semantics.



$$\frac{}{[] \vdash JDK(P)} \quad [JDK_{\varnothing}]$$

$$\frac{P \in \mathit{Permissions(n)} \quad \sigma \vdash JDK(P)}{\sigma : n \vdash JDK(P)} \quad [JDK_{\prec}]$$

$$\frac{P \in \mathit{Permissions(n)} \quad \mathit{Priv(n)}}{\sigma : n \vdash JDK(P)} \quad [JDK_{Priv}]$$

Fig. 4. Specification of the access control policy.

with $\mathcal{D}$. [1] Our model prevents this behaviour, because the JDK rules ensure that:

$$\forall n \in N, \sigma \in N^*. \quad P \notin \mathit{Permissions(n)} \implies \sigma : n \nvdash JDK(P)$$

Note also that our inference rules for JDK are *fixed*, as well as those for $\rhd$. So we are prevented from modelling permissions like `AllPermission` and `FilePermission("*","write")`, as they may breach security by altering the Java system binaries.

In the following, we will say that a permission P is *denied* (resp. *granted*) to a state σ if $\sigma \nvdash JDK(P)$ (resp. $\sigma \vdash JDK(P)$). Also, the *finite* set of all permissions referenced to in a given call graph will be denoted by **Permission**.

Back to our example, consider node n_{16}: both callers n_9 and n_{13} are privileged and have the permission P_{read}. Hence, the security check at n_{16} will

[1] This may happen through the `implies()` method.

$$\overline{DP}_{in}(n) = \bigcup_{(m,n)\in E} \{\overline{DP}_{out}(m,n)\}$$

$$\overline{DP}_{out}(m,n) = \begin{cases} \mathit{Permissions(n)} & \text{if } \bullet\!\!\longrightarrow n \\ \overline{DP}_{call}(m) \cap \mathit{Permissions(n)} & \text{if } m \longrightarrow n \\ \overline{DP}_{trans}(m) & \text{if } m \dashrightarrow n \end{cases}$$

$$\overline{DP}_{call}(n) = \begin{cases} \mathit{Permissions(n)} & \text{if } \mathit{Priv(n)} \\ \overline{DP}_{in}(n) & \text{otherwise} \end{cases}$$

$$\overline{DP}_{trans}(n) = \begin{cases} \emptyset & \text{if } \ell(n) = \texttt{check}(\mathrm{P}) \text{ and } \mathit{kill}(n,P) \\ \bigcup\limits_{\substack{(m,n)\in E \\ P\in\overline{DP}_{out}(m,n)}} \{\overline{DP}_{out}(m,n)\} & \text{if } \ell(n) = \texttt{check}(\mathrm{P}) \text{ and } \neg\mathit{kill}(n,P) \text{ and } \neg\mathit{Priv(n)} \\ \overline{DP}_{in}(n) & \text{otherwise} \end{cases}$$

$$\mathit{kill}(n,P) =_{def} \forall (m,n) \in E.\ P \notin \overline{DP}_{out}(m,n)$$

Fig. 5. The Denied Permissions Analysis.

always pass. The same holds for n_{18}, as its only caller is the privileged n_{14}. Now consider n_{11}: one of its callers (n_4) has permission P_{debit}, while the other (n_6) has not. Indeed, the security check at n_{11} is necessary. Also, note that no execution involving `clyde` will ever pass the check in n_{11}: then the permission P_{canpay} is always granted to both callers of n_8 (n_3 and n_{12}), and the check at n_8 turns out to be redundant, too.

Our static analyses aim at discovering the redundant checks, i.e. those that always succeed, as well as those that always fail.

3 Static analyses

Our first analysis is called *Denied Permissions Analysis* (DP for short). It computes, for each program node n, a safe approximation, i.e. a subset of the set of permissions that are denied to any state $\sigma : n$. The analysis is defined by the system of control flow equations $DP(G)$ in Fig. 5 (actually it defines the complement $\overline{DP}$ of DP w.r.t. **Permission**). Note that DP is a *forward* analysis, and that we are interested in the *largest* sets satisfying the equalities.

The control flow information is represented through a finite *property space*

$\mathcal{L} = \mathcal{L}_{in} \times \mathcal{L}_{out} \times \mathcal{L}_{call} \times \mathcal{L}_{trans}$, where $\mathcal{L}_{in}, \mathcal{L}_{call}, \mathcal{L}_{trans}$ are total function spaces from N to $\mathcal{P}(\textbf{Permission})$, while $\mathcal{L}_{out}$ is a total function space from E to $\mathcal{P}(\textbf{Permission})$. Assuming that $\mathcal{P}(\textbf{Permission})$ is partially ordered by $\supseteq$, a standard construction equips each of these spaces with a pointwise order. As an example, the set $\mathcal{L}_{in}$ is partially ordered by the relation $\sqsubseteq_{in}$ given by:

$$l_{in} \sqsubseteq_{in} l'_{in} \quad =_{def} \quad \forall n \in N.\ l_{in}(n) \supseteq l'_{in}(n)$$

Similarly, we define a join operator on these spaces. Back to our example:

$$l_{in} \sqcup_{in} l'_{in} \quad =_{def} \quad \lambda n : N.\ l_{in}(n) \cap l'_{in}(n)$$

With the above, our function spaces turn out to be finite complete lattices. Thus, also $\mathcal{L}$ is a finite complete lattice.

The equation system in Fig. 5 defines a *transfer function* $\mathcal{F}_{DP}$ between elements of this lattice, i.e. $\mathcal{F}_{DP} : \mathcal{L} \to \mathcal{L}$. Any solution $\delta \in \mathcal{L}$ of the control flow equations must satisfy $\delta = \mathcal{F}_{DP}(\delta)$: in this case, we write $\delta \models DP(G)$. Actually, $\mathcal{F}_{DP}$ is a monotonic (and continuous) function, therefore the chain $\bot_{\mathcal{L}} \sqsubseteq \mathcal{F}(\bot_{\mathcal{L}}) \sqsubseteq \mathcal{F}^2(\bot_{\mathcal{L}}) \sqsubseteq \cdots$ eventually stabilises to the *largest* solution of the equation system.

We can now state the correctness of our DP analysis. For every reachable state $\sigma : n$, the permissions denied to n are a superset of the $\delta_{call}(n)$ component of any solution.

Theorem 3.1 (Correctness of DP Analysis)
Let G be a call graph, $G \vdash \sigma : n$ and $\delta \models DP(G)$. Then:

$$P \in \delta_{call}(n) \quad \Longrightarrow \quad \sigma : n \not\vdash JDK(P)$$

The intuition follows on how a solution is built. The permissions non-denied at the entry of a node are the union of those (non-denied) at the exit of all its callers. Call nodes generate non-denied permissions only if they are privileged; otherwise they propagate the non-denied permissions of their entry points. A check node propagates the permissions of the callers that may pass the check. Return nodes have no outgoing edges, so they are irrelevant here. Note that permissions can be discarded when crossing the boundaries of protection domains. As an example, the $\overline{\delta}_{out}$ component for the edge $n_6 \to n_{11}$ in Fig. 1 is:

$$\begin{aligned} \overline{\delta}_{out}(n_6, n_{11}) &= \overline{\delta}_{call}(n_6) \ \cap \ \mathit{Permissions}(n_{11}) \ \subseteq \ \overline{\delta}_{call}(n_6) \ = \ \overline{\delta}_{in}(n_6) \\ &= \overline{\delta}_{out}(n_2, n_6) \ \cup \ \overline{\delta}_{out}(n_7, n_6) \\ &= (\overline{\delta}_{call}(n_2) \ \cup \ \overline{\delta}_{call}(n_7)) \ \cap \ \mathit{Permissions}(n_6) \ = \ \emptyset. \end{aligned} \tag{1}$$

Our second analysis is called *Granted Permissions Analysis* (GP for short). Similarly to DP, it gives, for every node n, a safe approximation of the set of

$$GP_{in}(n) = \bigcap_{(m,n)\in E} \{GP_{out}(m,n)\}$$

$$GP_{out}(m,n) = \begin{cases} \mathit{Permissions}(n) & \text{if } \bullet \longrightarrow n \\ GP_{call}(m) \cap \mathit{Permissions}(n) & \text{if } m \longrightarrow n \\ GP_{trans}(m) & \text{if } m \dashrightarrow n \end{cases}$$

$$GP_{call}(n) = \begin{cases} \mathit{Permissions}(n) & \text{if } \mathit{Priv}(n) \\ GP_{in}(n) & \text{otherwise} \end{cases}$$

$$GP_{trans}(n) = \begin{cases} \emptyset & \text{if } \ell(n) = \texttt{check}(\mathrm{P}) \text{ and } \mathit{kill}(n,P) \\ \bigcap\limits_{\substack{(m,n)\in E \\ P\in \overline{DP}_{out}(m,n)}} \{GP_{out}(m,n)\} \cup \{P\} & \text{if } \ell(n) = \texttt{check}(\mathrm{P}) \text{ and } \neg\mathit{kill}(n,P) \text{ and } \neg\mathit{Priv}(n) \\ GP_{in}(n) & \text{otherwise} \end{cases}$$

Fig. 6. The Granted Permissions Analysis.

permissions that are granted to any state with top n. The analysis is defined by the system of equations $GP(G)$ in Fig. 6. Also GP is a forward analysis, and we look for the largest sets satisfying the equalities.

The permissions granted at the entry of a node are those granted at the exit of *all* its callers. Call nodes generate granted permissions only if they are privileged; otherwise they propagate those at their entry points. A check node generates both the permissions it enforces and those granted to *all* of the callers that *may* pass the check. As an example of the GP analysis, we compute the set of permissions granted to node n_{16}:

$$\begin{aligned} \gamma_{call}(n_{16}) &= \gamma_{in}(n_{16}) \;=\; \gamma_{out}(n_9, n_{16}) \;\cap\; \gamma_{out}(n_{13}, n_{16}) \\ &= (\gamma_{call}(n_9) \;\cap\; \gamma_{call}(n_{13})) \;\cap\; \mathit{Permissions}(n_{16}) \\ &= \mathit{Permissions}(n_9) \;\cap\; \mathit{Permissions}(n_{13}) \\ &= \{P_{debit}, P_{canpay}, P_{read}, P_{write}\} \end{aligned} \tag{2}$$

We can now state the correctness of our GP analysis. For every reachable state $\sigma : n$, the permissions granted to n are a superset of the $\gamma_{call}(n)$ component of any solution.

Theorem 3.2 (Correctness of GP Analysis)
Let G be a call graph, $G \vdash \sigma : n$ and $\gamma \models GP(G)$. Then:

$$P \in \gamma_{call}(n) \quad \Longrightarrow \quad \sigma : n \vdash JDK(P).$$

Back to our example, the correctness theorem for GP ensures that any state whose top node is n_{16} will pass the security check, because $P_{read} \in \gamma_{call}(n_{16})$ (see Eq. 2). Thus, the GP analysis statically captures the redundancy of this check, which however is dynamically tested, as intuitively discussed in Section 2. This is an example of how our analysis can be used to optimize stack inspection by removing redundant checks from the code.

Figure 7 displays the largest solutions of the DP and GP analyses for the e-commerce example.

The largest solutions of the DP and the GP analyses can be computed by a slight adaptation of a standard worklist algorithm (see [9]). Our basic operations are the binary set union and intersection. Their computation requires a number of steps linear on $|\mathbf{Permission}|$, i.e. the size of the permissions set. Then a (coarse) upper bound on the number of basic operations performed by the a naive implementation of the worklist algorithm is $O(|E|^2 \cdot |\mathbf{Permission}|^2)$.

n	$\delta_{call}(n)$	$\gamma_{call}(n)$
$n_1 - n_2$	$\emptyset$	**Permission**
$n_3 - n_5$	$\{P_{read}, P_{write}\}$	$\{P_{debit}, P_{canpay}\}$
$n_6 - n_7$	$\{P_{debit}, P_{canpay}, P_{read}, P_{write}\}$	$\emptyset$
n_8	$\{P_{read}, P_{write}\}$	$\{P_{debit}, P_{canpay}\}$
n_9	$\emptyset$	$\{P_{debit}, P_{canpay}, P_{read}, P_{write}\}$
n_{10}	$\{P_{read}, P_{write}\}$	$\{P_{debit}, P_{canpay}\}$
n_{11}	$\{P_{read}, P_{write}\}$	$\emptyset$
n_{12}	$\{P_{read}, P_{write}\}$	$\{P_{debit}, P_{canpay}\}$
$n_{13} - n_{14}$	$\emptyset$	$\{P_{debit}, P_{canpay}, P_{read}, P_{write}\}$
n_{15}	$\{P_{read}, P_{write}\}$	$\{P_{debit}, P_{canpay}\}$
$n_{16} - n_{19}$	$\emptyset$	$\{P_{debit}, P_{canpay}, P_{read}, P_{write}\}$

Fig. 7. The largest solutions for DP and GP.

4 Optimized stack inspection

The correctness results of the previous section shed light on a possible optimization of the stack inspection algorithm. When an access control decision has to be made towards a permission P, the call stack (with nodes in place of protection domains) is examined top-down as follows. Assume n to be the

currently scanned node. If $P \in \delta_{call}(n)$, then an `AccessControlException` is thrown. Otherwise, if $P \in \gamma_{call}(n)$ the algorithm succeeds. If neither case occurs, the search goes on.

This optimized stack inspection algorithm is specified in Fig. 8 and is proved to yield the same results of the standard JDK.

$$\frac{}{[] \vdash JDK^\star(P)} \quad [JDK^\star_\varnothing]$$

$$\frac{P \notin \delta_{call}(n) \quad \sigma \vdash JDK^\star(P)}{\sigma : n \vdash JDK^\star(P)} \quad [JDK^\star_\delta]$$

$$\frac{P \in \gamma_{call}(n)}{\sigma : n \vdash JDK^\star(P)} \quad [JDK^\star_\gamma]$$

Fig. 8. Specification of the optimized access control policy.

Theorem 4.1 (Correctness of $JDK^\star$)
Let G be a call graph and $G \vdash \sigma$. Then, for any permission P:

$$\sigma \vdash JDK(P) \iff \sigma \vdash JDK^\star(P)$$

5 Concluding Remarks

In this paper we developed two Control Flow Analyses for the Java bytecode. The first analysis yields a safe approximation of the set of permissions granted to bytecode at run-time, and the second one approximates the denied permissions. The analyses provide us with the basis for reducing the run-time overhead due to stack inspection. Here, we focussed on Java bytecode, however the same static techniques can be applied to deal with programming languages or systems whose security architecture provides dynamic checking of permissions through stack inspection (e.g. C♯ [19]).

Many authors advocated the use of static techniques to optimize the check of security properties. Walker [16] develops a notion of typed compilation schemata: types encode assertions about program security ensuring that no run-time violation of the security properties will occur. Another approach is proposed by Jensen, Le Mètayer and Thorn [8]. They formalize classes of security properties (including stack inspection) through a linear time temporal logic. Then, model checking is used to prove that local security checks enforce a given global security policy.

The problem of optimizing stack inspection is addressed by Wallach and Felten in [17,18] by exploiting a belief logic [1] and a technique called *security-passing style*. Pottier, Skalka and Smith [11] introduce a type system to model

a simplified version of Java stack inspection. These two approaches implicitly characterize the checks thar are redundant, while our control flow analyses do it directly.

The extension of our proposal to the full access control policy requires the call graph construction algorithm to single out the program points where new threads can be generated. This step seems to be the hard part of the job. Indeed, we feel then that our analyses only require slight modifications.

Our program model does not handle *dynamic linking* features of Java. Actually, the whole program is available prior the construction of its call graph. The extension of our approach to cope with dynamic linking requires substantial efforts. The first step consists in linking dynamically the relevant call graphs. Then the available solutions for the various program fragments have to be combined. Some preliminary work on data flow analysis taking care of dynamic linking can be found in [13,15].

6 Acknowledgments

The last two authors are partially supported by the MURST project *TOSCA*, and the second author also by the MURST project *Interpretazione Astratta, Sistemi di Tipo e Analisi Control Flow*.

References

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM ToPLAS*, 706–734, 1993.

[2] A. Coglio, A. Goldberg, and Z. Qian. Toward a provably-correct implementation of the JVM bytecode verifier. T.R., Kestrel Institute, 1998.

[3] S. N. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language", In *ACM OOPSLA '98*, pp. 310–327.

[4] S. N. Freund and J. C. Mitchell. A formal framework for the Java Bytecode Language and Verifier. In *ACM OOPSLA '99*, pp. 147–166.

[5] A. Goldberg. A specification of Java loading and bytecode verification. In *5th ACM Conference on Computer and Communications Security*, pp. 49–58, 1998.

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

[7] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *ACM OOPSLA '97*, pp. 108–124.

[8] T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security policies. T.R., IRISA, 1998.

[9] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[10] T. Nipkow. Verified bytecode verifiers. In *FOSSACS 2001*, LNCS 2030.

[11] F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. In *ESOP '01*, LNCS 2028, pp. 30–45

[12] Z. Qian. Formal specification of a large subset of Java^{TM} virtual machine instructions for objects, methods and subroutines. In *Formal Syntax and Semantics of Java*, LNCS 1523, pp. 271-311, 1998.

[13] A. Rountev, B. G. Ryder and W. Landi. Data-Flow Analysis of Program Fragments. In *ESEC / SIGSOFT FSE*, 1999.

[14] R. Stata and M. Abadi. A type system for Java Bytecode Subroutines. In *ACM POPL '98*, pp. 149–160.

[15] V. Sreedhar, M. Burke and J. D. Choi A framework for interprocedural optimization in the presence of dynamic class loading. In SIGPLAN Conference on Programming Language Design and Implementation, 2000.

[16] D. Walker. A type system for expressive security policies. In *ACM POPL 2000*.

[17] D. S. Wallach, A. W. Appel and E. W. Felten. SAFKASI: a security mechanism for language-based systems. In *ACM TOSEM 2000*.

[18] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *Proc. of the 1998 IEEE Symposium on Security and Privacy*.

[19] C. Wille, Presenting C♯, SAMS Publishing, 2000.