Security-aware program transformations *

Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari

Dipartimento di Informatica, Università di Pisa, Italy
{bartolet, degano, giangl}@di.unipi.it

Abstract. Stack inspection is a basic mechanism for implementing language based security. Stack inspection is time consuming and may prevent from code optimization. A static analysis is presented that safely approximates the access rights granted at run-time. Stack inspection optimizations are then possible, along with program transformations.

1 Introduction

The growing use of network technologies in current distributed computing has made security critical in the design, development and distribution of applications. Indeed, both final users and application designers put special emphasis on security issues. For final users, the awareness of security mechanisms is crucial for choosing the best network services that match their requirements. Designers wish to control resource usage and access in order to ensure and maintain adequate security levels.

Designing and implementing security policies at the programming language level help in handling security. Here, we consider an authorization-based model where a security policy is enforced by inserting appropriate checks in a program. Clearly, writing secure applications is difficult: omitting a single check somewhere in the code may compromise the security of the whole application. There is no general mechanism which identifies what kind of security checks have to be inserted in a program, and where.

The Java programming language features constructs and mechanisms for secure execution of mobile code. Java applications run components with different levels of trust, e.g., components originated from different administration domains. In the Java security model, access control decisions are taken by examining the call stack at run-time. A permission is granted, provided that it belongs to all principals on the call stack. The so-called privileged operations are an exception. These are allowed to execute any code granted to their principal, regardless of the calling sequence. This access control mechanism is known as stack inspection. Beyond Java, other run-time environments (e.g., the .NET Common Language Runtime [15]) adopt stack inspection as basic authorization mechanism.

* Work partially supported by EU project DEGAS (IST-2001-32072), MIUR project “Mlto di FormalI per la Sicurezza e il TempO” (MEFISTO), and MIUR project “Network Aware Programming: Object, Languages, Implementation” (NAPOLI).
Stack inspection has some drawbacks. First, the run-time overhead due to
the analysis of stack frames may grow very high. Second, stack inspection deeply
affects standard program transformations, such as method inlining and tail call
elimination. These optimizations may in fact alter the structure of the call stack.
Hence, understanding the semantics of program transformations in a setting with
stack inspection is a research (and technological) challenge.

Our contribution aims at developing semantic-driven mechanisms as an aid
to improve efficiency of architectures for language-based security. We build over
control flow analysis [16]. The idea of control flow analysis is to efficiently obtain
computable approximations of the set of values that the objects of a program
may assume during its execution. These approximations are then used to analyze
program properties in a safe way: 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”.

In Section 2 we consider an idealized object-oriented language with primitive
constructs for method invocations, exceptions, and access control. The execution
traces of Java code are safely approximated by the traces of their corresponding
abstract programs. We represent these programs by control flow graphs, a pro-
gramming model not tied to any particular language. These graphs are equipped
with a formal operational semantics.

In Section 3 we introduce a static analysis over control flow graphs, called
Trace Permissions Analysis. This analysis computes, for each program point and
each execution reaching that point, the set of permissions granted at run-time.
The analysis is sound and complete with respect to the operational semantics
of our idealized language, i.e. it computes all and only the permissions that are
granted at run-time.

In Section 4 we show that the Trace Permissions analysis provides us with the
basis for some security-aware code optimizations. As a first application, we detect
and remove the redundant checks in a program, i.e. the checks which always pass.
Dead code elimination is a program optimization which detects and removes
the code unused or unreachable in executions. Security restrictions may cause
more fragments of code to become unreachable, e.g. because a security check
protecting it is never passed. Our technique permits to discard such dead code
in the linking phase. We also cope with method inlining, an optimization that
replaces a method invocation with a copy of the called method code. In presence
of stack inspection, method inlining may break security, because the protection
domain of the inlined method is ignored. The Trace Permissions analysis provides
us with the basis to efficiently construct the set of method invocations which can
be safely inlined.

In Section 5 we examine the adequacy of our model, and we propose some
extensions to make it suitable for real-world applications.

Because of space limitations, here we will provide the overall picture of the
technical development of our approach by focusing on the underlying ideas. We
refer to the full paper [2] for the proofs, some illustrative examples, and the
detailed description of the model.
2 The program model

We model programs as control flow graphs (CFGs for short) whose nodes represent the activities relevant for stack inspection (i.e. checks, method invocations and returns) and whose arcs represent the flow of control. We do not define how CFGs are extracted from an actual program. This construction is well understood and algorithms and tools exist for it; see for example [10,16,20,21].

By construction, CFGs hide any data flow information, and are therefore approximated; typically, the conditional construct is rendered as non-deterministic choice. This approximation is safe, in the sense that any actual execution flow is represented by a path in the CFG. However, the converse may not be true: some paths may exist which do not correspond to any actual execution. For instance, both branches of an “if” statement are represented, even in the cases when always the same branch is taken at run-time.

There is a further source of approximation, especially for object-oriented languages with dynamic resolution of method invocations. In Java, for example, when a program invokes an instance method on an object \( O \), the virtual machine may have to choose among various implementations of that method. The decision is not based on the declared type of \( O \), but on the actual class \( O \) belongs to, which is unpredictable at static time. To be safe, CFGs consider a superset of the methods that can be invoked at each program point. This is a main source of approximation for the analyses built over CFGs.

2.1 Syntax

Let \( \mathcal{D} \) be a finite set of protection domains, and \( \mathcal{P} \) be a finite set of permissions.

**Definition 1.** A CFG \((N \cup \{n_e\}, E, \text{Priv}, \text{Dom})\) is an oriented graph, where:

- \( N \) is the set of nodes. Each node \( n \in N \) is associated with a label \( \ell(n) \), describing the control flow primitive it represents. Labels partition nodes in three kinds: call nodes, that stand for method invocation, return nodes, which represent return from a method, and check nodes, which enforce the access control policy. For each \( P \in \mathcal{P} \), a node labeled check\( (P) \) can be seen as the abstract representation of an AccessController.checkPermission\( (P) \) instruction in the Java language. The distinguished element \( n_e \notin N \) plays the technical role of a single, isolated entry point.
- \( E \subseteq (N \cup \{n_e\}) \times N \) is the set of edges. Edges are partitioned into four sets: entry edges \( \bullet \rightarrow n \), that represent the entry points of a program; call edges \( n \rightarrow n' \), which model interprocedural flow; transfer edges \( n \rightarrowtail n' \), which correspond to sequencing; and catch edges \( n \rightarrow_i n' \), which correspond to exception handling. The last two kinds of edges represent intraprocedural flow. The set of entry edges contains all pairs \((n_e, n)\) where \( n \) is a program entry point. The \( n_e \) element is the source of entry edges, only.
- \( \text{Priv}: N \rightarrow \text{Bool} \) tells whether a node enables its privileges or not.
- \( \text{Dom}: N \rightarrow \mathcal{D} \) is a mapping from nodes to protection domains.
When unambiguous, we shall write \( \langle N, E \rangle \) instead of \( \langle N \cup \{n_e\}, E, \text{Priv}, \text{Dom} \rangle \).

Each CFG is associated with a security policy \( \text{Perm} : \mathcal{D} \rightarrow 2^\mathcal{P} \), which grants a set of permissions to each protection domain. Hereafter, we will always abbreviate \( \text{Perm}(\text{Dom}(n)) \) with \( \text{Perm}(n) \).

**Definition 2.** The methods of a CFG \( \langle N, E \rangle \) are the connected components of the graph \( \langle N, E' \rangle \), where \( E' \) is the set of intraprocedural edges in \( E \), with no orientation. We call \( \mu(n) \) the method to which node \( n \) belongs. The entry points of \( \mu(n) \) are defined as:

\[
\varepsilon(\mu(n)) = \{ n' \in \mu(n) \mid \bullet \rightarrow n' \land \exists m \in N. m \rightarrow n' \}
\]

The set \( \rho(n) \) of return nodes associated to a node \( n \) is:

\[
\rho(n) = \{ m \in N \mid \ell(m) = \text{return} \land n \rightarrow \varepsilon(\mu(m)) \}
\]

The set \( \xi(n) \) of nodes that may throw an exception catchable by \( n \) is defined as the smallest set satisfying:

\[
\xi(n) = \begin{cases} 
\{ n \} & \text{if } \ell(n) = \text{check}(P) \\
\{ \xi(n') \mid n \rightarrow \varepsilon(\mu(n')) \land n' \not\rightarrow_i \} & \text{otherwise}
\end{cases}
\]

The set \( \xi_i(n) \) of nodes that may propagate an exception to \( n \) is defined as:

\[
\xi_i(n) = \{ n' \mid n \rightarrow \varepsilon(\mu(n')) \land n' \not\rightarrow_i \land \xi(n') \neq \emptyset \}
\]

As discussed in [2], all the CFGs derived from admissible Java programs satisfy the following well-formedness constraints: (1) check nodes have no outgoing call edges; (2) return nodes have no outgoing edges; (3) each method has a single entry point; (4) nodes in the same method are in the same protection domain. Moreover, we require that only call nodes can be privileged. In general, security checks can also occur within privileged actions: however, privileged check nodes make little sense, because it is always possible to determine whether a privileged check will succeed or not. Similarly, there is no point in enabling return nodes to be privileged, because a return node will never be on the call stack when stack inspection is performed.

### 2.2 Semantics

The operational semantics of CFGs is defined by a transition system whose configurations are sequences of nodes, modeling call stacks. Additionally, each state has a boolean tag which tells whether an exception is active, i.e. thrown and not caught yet. Formally, we define the set of states as \( N^* \times \text{Bool} \).

If no exception is active, a state is represented as sequence of nodes enclosed in square brackets: for example, \( \sigma = [n_0, \ldots, n_k] \) is a state whose top node is \( n_k \). If an exception is active, we append the symbol \( \dagger \) to the sequence of nodes, i.e.
\[
\begin{array}{c|c|c}
\rightarrow n & \ell(n) = \text{call } n \rightarrow n' & \ell(m) = \text{return } n \rightarrow n' \\
\vdash [n] & \sigma : n \vdash \sigma : n' & \sigma : n \vdash \sigma : n' \\
\ell(n) = \text{check}(P) & \sigma : n \vdash P \rightarrow n' & \ell(n) = \text{check}(P) \sigma : n \not\vdash P \\
\vdash [n] & \sigma : n \vdash \sigma : n' & \sigma : n \vdash \sigma : n' \\
\end{array}
\]

\[
\begin{array}{c|c|c}
P \in \text{Perm}(n) & \sigma : n \vdash P & P \in \text{Perm}(n) \; \text{Priv}(n) \\
\vdash [n] & \sigma : n \vdash P & \sigma : n \vdash P \\
\end{array}
\]

Table 1. Operational semantics of CFGs.

\(\sigma\xi\) abbreviates \(\langle\sigma, \text{true}\rangle\). Pushing a node \(n\) on a stack \(\sigma\) is written as \(\sigma : n\) (the infix operator \(\vdash\) associates to the left).

The transition relation \(\vdash\) between states is the minimal relation induced by the inference rules in Table 1. A trace of \(G\) leading to \(\langle\sigma_0, x_0\rangle \vdash \cdots \vdash \langle\sigma_k, x_k\rangle\) is a derivation \(\sigma_0 = [n]\) and \(x_0 = \text{false}\). By overloading the notation, we also denote with \(\vdash\) the relation:

\[
\begin{array}{c}
G \vdash \langle[\mathbf{\null}], \text{false}\rangle \\
G \vdash \langle\sigma, x\rangle \vdash \langle\sigma', x'\rangle \\
G \vdash \langle\sigma', x'\rangle
\end{array}
\]

stating when there is a trace of \(G\) which can lead to a given state. We say that a node \(n\) is reachable iff \(\langle\sigma : n, x\rangle\) is a reachable configuration.

In our formalization, we use a slightly simplified version of the full access control algorithm presented in [8]. The simplified algorithm scans the call stack top-down. Each frame in the stack refers to the protection domain containing the class to 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. We formally specify this behavior by the minimal relation induced by the inference rules for \(\vdash\) in Table 1. We say that a permission \(P\) is granted to a state \(\sigma\) if \(\sigma \vdash P\).

### 3 The Trace Permissions Analysis

In this section we review the static analysis over CFGs called Trace Permissions Analysis (TP). The TP analysis approximates the access rights granted to each reachable state.
\[ TP_{\text{in}}(n) = \bigcup_{(m,n) \in E} TP_{\text{out}}(m, n) \]

\[
TP_{\text{out}}(m, n) = \begin{cases} 
\{ \{ \text{Dom}(n) \} \} & \text{if } \bullet \rightarrow n \\
\{ \gamma \cup \{ \text{Dom}(n) \} \mid \gamma \in TP_{\text{call}}(m) \} & \text{if } m \rightarrow n \\
TP_{\text{regm}}(m) & \text{if } m \rightarrow \gamma n \\
\end{cases}
\]

\[
TP_{\text{call}}(n) = \begin{cases} 
\{ \{ \text{Dom}(n) \} \} & \text{if } \text{Priv}(n) \text{ and } TP_{\text{in}}(n) \neq \emptyset \\
TP_{\text{in}}(n) & \text{otherwise} \\
\end{cases}
\]

\[
TP_{\text{regm}}(n) = \begin{cases} 
\{ \gamma \in TP_{\text{in}}(n) \mid P \in \Pi(\gamma) \} & \text{if } \ell(n) = \text{check}(P) \\
\{ \gamma \in TP_{\text{in}}(n) \mid \text{Trans}(n, \{ \text{Dom}(n) \}) \} & \text{if } \ell(n) = \text{call}, \text{Priv}(n) \\
\{ \gamma \in TP_{\text{in}}(n) \mid \text{Trans}(n, \gamma) \} & \text{otherwise} \\
\end{cases}
\]

\[
TP_{\text{arch}}(n) = \begin{cases} 
\{ \gamma \in TP_{\text{in}}(n) \mid P \notin \Pi(\gamma) \} & \text{if } \ell(n) = \text{check}(P) \\
\{ \gamma \in TP_{\text{in}}(n) \mid \text{Catch}(n, \{ \text{Dom}(n) \}) \} & \text{if } \ell(n) = \text{call}, \text{Priv}(n) \\
\{ \gamma \in TP_{\text{in}}(n) \mid \text{Catch}(n, \gamma) \} & \text{otherwise} \\
\end{cases}
\]

\[
\text{Trans}(n, \gamma) \overset{d}{=} \exists m \in \rho(n). \gamma \cup \{ \text{Dom}(m) \} \in TP_{\text{in}}(m) \\
\text{Catch}(n, \gamma) \overset{d}{=} \exists m \in \xi(n). \gamma \cup \{ \text{Dom}(m) \} \in TP_{\text{arch}}(m)
\]

<table>
<thead>
<tr>
<th>Table 2. Flow equations for the TP analysis.</th>
</tr>
</thead>
</table>

Since the set of permissions granted to a state is just the intersection of the permissions associated to each protection domain traversed after the last privileged frame (if any), we can identify the set \( \{ P \in \mathcal{P} \mid \sigma \vdash P \} \) with the security context \( \Gamma(\sigma) \), where \( \Gamma : N^* \rightarrow 2^D \) is defined as follows:

\[
\Gamma([]) = \emptyset \quad \Gamma(\sigma : n) = \begin{cases} 
\{ \text{Dom}(n) \} & \text{if } \text{Priv}(n) \\
\Gamma(\sigma) \cup \{ \text{Dom}(n) \} & \text{otherwise} \\
\end{cases}
\]

The set of permissions granted to a security context \( \gamma \) is:

\[
\Pi(\gamma) = \bigcap_{D \in \gamma} \text{Perm}(D)
\]

The permissions granted to the security context of a state \( \sigma \) are exactly the permissions granted to \( \sigma \), i.e. \( \sigma \vdash P \iff P \in \Pi(\Gamma(\sigma)) \) for all \( \sigma \in N^* \), \( P \in \mathcal{P} \).

Given a CFG \( G \) and a security policy \( \text{Perm} \), the analysis is specified by the set of equations \( TP^- (G, \text{Perm}) \) in Table 2. A solution \( \tau \models TP^- (G, \text{Perm}) \) is
a 5-tuple $\tau = (\tau_{in}, \tau_{call}, \tau_{trans}, \tau_{catch}, \tau_{out})$ which satisfies all the equations. The purpose of the analysis is to find, for each node $n$, the set $\{ \Gamma(\sigma : n) \mid G \triangleright \sigma : n \}$. 

Technically, TP is a forward, monotone control flow analysis with values in $2^{2^D}$. Since both $G$ and $D$ are finite, the least solution to the analysis does exist and is finitely computable.

The following theorem states the correctness of the TP analysis. The first equation below states that any solution to the analysis is sound w.r.t. the operational semantics. The second equation states that the least solution to the analysis is also complete. This fact should not seem bizarre: indeed, completeness is only up to the precision of the CFG, which is an approximated model of the analyzed program.

**Theorem 1.** Let $\tau \models TP^-(G, Perm)$. Then:

$$G \triangleright \sigma : n \implies \exists \gamma \in \tau_{call}(n). \gamma = \Gamma(\sigma : n)$$

Moreover, the minimal solution w.r.t. the inclusion relation on $2^{2^D}$ is such that:

$$\gamma \in \tau_{call}(n) \implies \exists \sigma. G \triangleright \sigma : n \land \gamma = \Gamma(\sigma : n)$$

The worklist algorithm which actually computes the (unique) minimal solution to the analysis has computational complexity $O(c \cdot |N|) = O(|N|)$. The constant $c$ depends on the number of protection domains occurring in $G$: in the worst case, $c = 2^{|D|}$, where $D = \bigcup_{n \in N} \text{Dom}(n)$. However, the exponential factor only occurs when the number of protection domains is proportional to the number of nodes. Actually, the number of protection domains can be considered as a constant, because it depends on the security policy, rather than on the size of the program.

Dynamic linking is the mechanism which allows a program to be extended on demand, e.g. with code coming from the network. Although our program model does not directly support this feature, the TP analysis can be computed incrementally. An incremental CFG construction algorithm, e.g. the one presented in [19], can be used in order to correctly perform the dynamic linking of the relevant CFGs. Indeed, this operation cannot be performed by looking at the CFGs alone, because CFGs do not carry enough information to restrict the set of targets of dynamically dispatched method invocations. We now outline how the incremental computation of the analysis is performed. Let $G = \langle N, E \rangle$, and assume that a solution $\tau$ to $TP^-(G, Perm)$ is available when the CFG $G' = \langle N', E' \rangle$ is loaded. Through the CFG construction algorithm, we single out the set $E_m$ of resolved calls between $G$ and $G'$, i.e. those edges $n \rightarrow m$ such that $n, m$ do not belong both to the same CFG. Linking $G$ and $G'$ together yields the CFG $G \bowtie G' = \langle N \cup N', E \cup E' \cup E_m \rangle$. The analysis $\tau' \models TP^-(G \bowtie G', Perm)$ is a refinement of $\tau$. To compute it, the worklist algorithm adds to $\tau$ all the contexts associated with the new paths created by the resolved calls. It suffices now to restart the algorithm with the worklist containing all nodes $n$ such that, for some node $m$, $(n, m) \in E_m$. Moreover, the worklist must include all entry points of $G'$, if any. Although this technique is not fully compositional, note that adding new executable paths to a CFG never affects the analysis of the old ones.
4 Program transformations

In this section we show that the Trace Permissions analysis provides us with an effective basis for some security-aware code optimizations.

4.1 Redundant checks elimination

Our first application of the analysis is a code optimization which detects and removes the redundant checks occurring in a program, i.e. those checks which always pass, regardless of the execution trace.

The following theorem states conditions to recognize redundant checks, so enabling the compiler to safely remove them from the code:

**Theorem 2.** Let $\tau \models TP^-(G, Perm)$. For each check node $n$, let $\Pi(n)$ be the set of permissions (statically) granted to $n$:

$$\Pi(n) = \bigcap \{ \Pi(\gamma) \mid \gamma \in \tau_{\text{call}}(n) \}$$

If $\ell(n) = \text{check}(P)$ and $P \in \Pi(n)$, then $n$ is redundant, i.e. for each $\sigma \in N^*$:

$$G \triangleright \sigma : n \implies \sigma : n \vdash P$$

Actually, redundant checks can only be disabled in presence of dynamic linking, because loading a new method may add new traces where the permission is no longer granted. A similar situation also holds for the other optimizations of stack inspection considered below.

4.2 Dead code elimination

Dead code elimination is a program optimization which prevents the compiler from generating bytecode for unreachable or useless pieces of code. Dead code elimination reduces both the size of the generated bytecode and the total application running time (e.g. when code has to be downloaded from the network).

The following theorem allows to detect (and remove) those pieces of code which cannot be reached due to security restrictions:

**Theorem 3.** Let $\tau \models TP^-(G, Perm)$. Then:

$$\tau_{\text{call}}(n) = \emptyset \implies \exists \sigma. G \triangleright \sigma : n$$

4.3 Method inlining

The TP analysis can be exploited to compute the set of method invocations which can be safely inlined. Intuitively, a method invocation may be inlined if the outcome of the security checks is not affected by the the elimination of the protection domain of the inlined method.
We adopt the so-called original version inlining approach [11], which always considers the original version of the callee and the current version of the caller when performing inlinings. This can be obtained by duplicating the original code of the inlined method.

Let \( \hat{n} \) be the node candidate for inlining, and \( \hat{n} \rightarrow n' \). We assume that the method invocation represented by \( \hat{n} \) can be statically dispatched, i.e. it has exactly one callee, represented by \( \mu(n') \).

The decision procedure, which tells whether or not the inlining of \( \hat{n} \) is safe, is outlined below. We first assign a fresh name to the protection domain of \( \mu(n') \), without modifying its granted permissions. Assume that a solution \( \tau \) to the TP analysis is available. We restart the worklist algorithm from \( \hat{n} \), in order to isolate the protection domain of \( \mu(n') \) in the computed security contexts. This allows for the definition of a function \( Inl_\hat{n} \) which simulates the effect of method inlining on security contexts: given a context \( \gamma \), \( Inl_\hat{n}(\gamma) \) is obtained by substituting the protection domain of \( \mu(n') \) for that of \( \mu(\hat{n}) \). Each time a check node \( n \) is reached, we ensure that for each context \( \gamma \in \tau_\mu(n) \), \( \gamma \) and \( Inl_\hat{n}(\gamma) \) agree on the permission \( P \) checked by \( n \), i.e. \( P \in \Pi(\gamma) \iff P \in \Pi(Inl_\hat{n}(\gamma)) \).

The inlining of \( \hat{n} \) is safe if this holds for each check node reached during this procedure. Note that it is possible to deal with the general case of virtual calls with many callees; this only requires some more machinery (all the possible callees must be inlined).

We formally define below when a method invocation can be inlined. The condition (1a) guarantees static dispatching of \( \hat{n} \), as well as that \( \hat{n} \) is not a recursive call (otherwise inlining makes little sense). The condition (1b) rephrases the original version inlining approach. The condition (1c) ensures that the protection domain of \( \hat{n} \) is isolated. These conditions, apart from \( \hat{n} \) being not recursive, can easily be satisfied, as noted above. The key condition is (1d); it guarantees that the security checks passed after inlining are exactly those passed before inlining.

**Definition 3.** We say that \( \hat{n} \) is ininlineable in \( G \) iff:

\[
\exists n' \in N. \hat{n} \rightarrow n' \land n' \notin \mu(\hat{n}) \tag{1a}
\]

\[
\forall n, n' \in N. \forall n' \in N. \hat{n} \rightarrow n' \land m \rightarrow n' \implies m = \hat{n} \tag{1b}
\]

\[
\forall n' \in N. \hat{n} \rightarrow n' \implies \forall m \notin \mu(n'). Dom(m) \neq Dom(n') \tag{1c}
\]

\[
\forall n. \ell(n) = \text{check}(P), \gamma \in \tau_\mu(n), P \in \Pi(\gamma) \iff P \in \Pi(Inl_\hat{n}(\gamma)) \tag{1d}
\]

Next, we define the effect of the method inlining transformation on CFGs. Instead of substituting \( \hat{n} \) for \( \mu(n') \) and adjusting the edges accordingly, we equivalently operate on the semantics of the transformed CFG.

The effect of the inlining of \( \hat{n} \) on states is simulated by the function \( inl_\hat{n} \) in Table 3. Given a state \( \sigma \), \( inl_\hat{n}(\sigma) \) is obtained by removing all the occurrences of \( \hat{n} \) in \( \sigma \) (except when \( \hat{n} \) is in top position).

The operational semantics of a CFG after the inlining of \( \hat{n} \) is defined by the transition relation \( B^{\hat{n}} \) in Table 3.
We define the \( \hat{n} \)-inlined version of a CFG \( G = \langle N \cup \{ \hat{n} \}, E, \text{Priv}_G, \text{Dom}_G \rangle \) as \( \hat{G} = \langle N \cup \{ \hat{n} \}, E, \text{Priv}_{\hat{G}}, \text{Dom}_{\hat{G}} \rangle \), where:

\[
\text{Priv}_{\hat{G}}(n) = \begin{cases} 
\text{true} & \text{if } \text{Priv}_G(\hat{n}) \text{ and } \hat{n} \rightarrow n \\
\text{Priv}_G(n) & \text{otherwise}
\end{cases}
\]

\[
\text{Dom}_{\hat{G}}(n) = \begin{cases} 
\text{Dom}_G(\hat{n}) & \text{if } \hat{n} \rightarrow n \\
\text{Dom}_G(n) & \text{otherwise}
\end{cases}
\]

Note that we may end up with privileged checks and returns, thus violating one of the well-formedness constraints in Section 2. However, this constraint can easily be removed, at the price of a slightly more involved definition of the TP analysis.

The following theorem states the correctness of method inlining: each trace in the original CFG corresponds to a trace in the \( \hat{n} \)-inlined version of the CFG.

**Theorem 4.** If \( \hat{n} \) is inlinable in \( G \), and \( \hat{G} \) is the \( \hat{n} \)-inlined version of \( G \), then:

\[
(\sigma_0, x_0) \triangleright \cdots \triangleright (\sigma_k, x_k) \iff (\hat{\sigma}_0, x_0) \triangleright \hat{n} \triangleright \cdots \triangleright \hat{n} \triangleright (\hat{\sigma}_k, x_k)
\]

where \( \sigma_0 = [], x_0 = \text{false} \), and \( \hat{\sigma}_i = \text{inl}_n(\sigma_i) \) for each \( i \in 0..k \).
5 Adequacy of the model

There are some differences between our model and the Java security model [8]:

- our model prevents a permission $P$ to be granted to a state $\sigma : n$ if $P$ does not belong to the permissions granted to $\text{Dom}(n)$. Instead, in the Java security model, $P$ may be implied by some permission $P' \in \text{Perm}(n)$. For example, FilePermission("/=","read") implies the permission to read any file on the local disk. We can easily extend our program model by introducing a partial order on permissions to encompass permission implications. The inclusion test $P \in \text{Perm}(n)$ in the rules for $\vdash$ should be replaced by $\text{Perm}(n) \Rightarrow P$, which tests if $P$ is implied by some permission $P' \in \text{Perm}(n)$.
- although the Java security model allows for the dynamic instantiation of permissions (e.g. an application that asks the user for a file name and then tries to open that file), we only consider the permissions that can be determined statically. We are currently investigating an extension of our present approach to deal with such parametric permissions on the form $P(x)$, where $x$ ranges over the set of possible targets for the permissions of class $P$.
- in the Java security model, a new thread upon creation inherits the access control context (i.e. the set of protection domains for the classes on the call stack) from its parent. When stack inspection is performed, both the context of the current thread and the contexts of all its ancestors are examined. In this way, a child thread cannot obtain a resource access which is not granted to its ancestors. We do not model threads. To consider them, we should first single out the program points where new threads can be created (and started) while constructing the CFG (as done in [12]).
- in our model, we consider a “skeletal” exception handling mechanism, where exceptions are all of the same type, and neither nested try blocks nor finally clauses are featured. A full treatment of exceptions requires a tailored construction of the CFG, e.g. by the techniques presented in [5,18], that also suggest how to adjust interprocedural analyses to exceptions.
- the Java Authentication and Authorization Service [13] extends the Java security model by allowing for user-centric access control policies, based on the principal who actually runs the code. Permissions can be granted to principals, and the doAs method allows a piece of code to be executed on behalf of a given subject. This is done by associating the (authenticated) subject running the code with the current access control context. Stack inspection ensures that subjects are taken into account when access control is performed. Unfortunately, static analysis techniques are weak in detecting the set of principals which can get authenticated at a given program point.

There are some features of the Java security architecture we think difficult to cope with: they are reflection, native methods, some “dangerous” permissions implications (e.g. AllPermission may breach the whole security system by replacing the JVM system binaries), and dynamic policies (although some recent works, e.g. [9,14], have addressed the formal treatment of this issue). Besides deeply affecting security, these features reduce the effectiveness of any static analysis which aims at determining the permissions granted to running code.
6 Conclusions and related work

We have developed a technique to perform program transformations in presence of stack inspection. The technique relies on the definition of our Trace Permissions Analysis. It is a control flow analysis, and computes a safe approximation to the set of permissions which are always granted to bytecode at run-time. The analysis is sound and complete w.r.t. the control flow graphs derived from the bytecode (however, these graphs only approximate the actual behavior). In this paper, we focused on redundant checks elimination, dead code elimination and method inlining. A similar approach also applies to general tail call elimination (see [2] for details). Although we restricted our attention to Java, the same techniques work with other programming languages whose authorization mechanisms rely on stack inspection (e.g., C# [23]).

Many authors advocated the use of static techniques in order to understand and optimize stack inspection.

Besson, Jensen, Le Métayer and Thorn [4] were among the first to apply static techniques to the verification of global security properties. They formalize classes of security properties through a linear-time temporal logic. They show that a large class of policies (including stack inspection) can be expressed in this formalism, while more sophisticated ones (like the Chinese Wall policy) are not. Model checking is then used to prove that local security checks enforce a given global security policy. Their verification method is based on the translation from linear-time temporal formulae to deterministic finite-state automata, and it can be used to optimize stack inspection. For each node \( n \), the analysis in [4] can compute the set \( \{ P \in P_{\text{check}} \mid G \models \sigma : n \land \sigma : n \vdash P \} \), where \( P_{\text{check}} \) is the set of permissions checked in \( G \). The computational complexity of the method is \( O(c \cdot |N|) \), where the constant \( c \) depends on the cardinality of \( P_{\text{check}} \) (in the worst case, \( c = 2^{|P_{\text{check}}|} \)). Therefore, our Trace Permissions analysis performs better when there are few protection domains, while [4] is more efficient when there are few security checks. Note that our analysis is at least as precise as [4], because \( P_{\text{check}} \subseteq P \). Also, the analysis in [4] does not seem to scale up smoothly to handle dynamic linking, because it must be recomputed each time a new permission is discovered.

Wallach, Appel and Felten [22] formalize stack inspection by exploiting the access control logic of [1]. The authors show that their decision procedure is equivalent to Java stack inspection, according to an informal operational semantics. Moreover, they propose an alternative semantics of eager stack inspection, called security-passing style. This technique consists of tracking the security state of an execution as an additional parameter of each method invocation. This allows for interprocedural compiler optimizations that do not interfere with stack inspection. The security-passing style allows each security operation to be performed in constant time, but it involves an overhead, because the security state must be computed at each method invocation. Dynamic caching techniques are adopted to reduce this overhead: therefore, in the optimal case, the additional cost of each method invocation is that of a hash lookup. The same technique allows for an implementation of security checks which requires a hash lookup in
the optimal case. In [3] we propose an approach to eager stack inspection which allows for security operations that cost as a hash lookup in the worst case, while, in the optimal case, they are as cheap as a bitwise operation. A further difference w.r.t. our approach is that [22] assumes that the whole program is available at compilation time, while we can deal with dynamic linking of code.

Potter, Skalka and Smith [17] address the problem of stack inspection in \( \lambda_{sec} \), a typed lambda calculus enriched with primitive constructs for enforcing security checks and managing permissions. They have polymorphic types on the form \( \tau_1 \rightarrow \varsigma \rightarrow \tau_2 \), where \( \tau_1, \tau_2 \) are types and \( \varsigma \) is a set of permissions. Intuitively, the type \( \tau_1 \rightarrow \varsigma \rightarrow \tau_2 \) details the security context necessary to execute a function of type \( \tau_1 \rightarrow \tau_2 \). Stack inspection never fails on a well typed program, because the set of permissions granted at runtime always includes the security context. These types are very powerful and can deal with several issues (e.g. security policy overriding, dependencies from untrusted code). Moreover, they can be smoothly extended to deal with objects by standard type-theoretic techniques.

The problem of establishing the correctness of program transformations in presence of stack inspection is investigated by Fournet and Gordon in [7]. They present an equational theory, together with a coinductive proof technique, for the \( \lambda_{sec} \) calculus. They study how stack inspection affects program behavior, proving that certain function inlinings and tail-call eliminations are correct. The equational theory is used to reason about the (somewhat limited) security properties actually guaranteed by stack inspection. Some examples are also given of how subtle interaction between trusted and untrusted code may give rise to security breaches. Here, we are more concerned with efficient (semantically-based) optimization procedures, rather than with a general reasoning framework.

Clemens and Felleisen [6] presents a different semantics of (eager) stack inspection on continuation CESK machines, which allows for tail-call optimizing implementations.

Compared with our approach, [6,7,17] consider more basic programming primitives (e.g. there is no exception mechanisms). Also, static typing appears to be more difficult than control flow analysis when permissions can be dynamically instantiated. Indeed, we argue that typing and control flow analysis are complementary static techniques. Approaches based on types focus more on defining safe programming disciplines; control flow analysis, instead, seems more accurate in efficiently determining effective program optimizations.

References
