# Method Inlining in the presence of Stack Inspection ⋆

Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari

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

**Abstract.** We consider languages that use stack inspection as an access control mechanism, and concentrate on a specific optimization technique, namely method inlining. Based on the static analysis of [5], we specify when this optimization is possible, preserving the policy for access control associated with applications. Remarkably, our proposal works even in the presence of dynamic linking.

## 1 Introduction

Security-aware programming languages, such as Java and C♯, have introduced programmable authorization-based models to determine when a principal can access a resource. At run-time, access control decisions are taken by inspecting the call stack. A permission is granted, provided that it belongs to *all* the principals on the call stack. The so-called *privileged operations* are an exception: they are allowed to execute any code granted to their principal, regardless of the calling sequence. This mechanism is known as *stack inspection*.

Recently, just-in-time compilation technologies (e.g. the Java HotSpot Compiler [22]) have been introduced to improve the performance of bytecode execution. However, optimizing bytecode via interprocedural transformations may compromise the security. For instance, consider *method inlining*, a standard program optimization which substitutes a copy of the called method in place of the calling instruction. Method inlining allows for valuable performance improvements: first, it reduces the overhead of dynamic dispatching (indeed, virtual method calls are a well-known bottleneck in the optimization of object oriented languages); second, method inlining allows the just-in-time optimizers to work on larger blocks of code, so making intraprocedural optimizations more effective. However, method inlining has the side effect of altering the call stack, that no longer contains the frame of the inlined method. This may affect the security of a program, which strictly depends on the structure of the run-time stack. Similar considerations apply to library usages. The security of a software library depends on the programs that use it: therefore, standard approaches to method inlining cannot be smoothly exploited to optimize library bytecode.

⋆ Work partially supported by EU project DEGAS and FET project PROFUNDIS.

A large amount of papers [2–7, 9, 10, 12, 16, 20, 23] witnesses the interest towards formally understanding stack inspection. Moreover, some program transformation techniques that preserve programmable access control policies have been introduced. All these approaches share a basic assumption: the binding between a piece of code and its permissions is made at class-loading time, and it cannot be modified at run-time. However, starting from version 1.4.1, the Java security architecture allows for *dynamic security policies*: the binding between a class and its permissions *can* be deferred until the class is involved in an access control test (of course, static binding of permissions is still allowed).

In a recent paper [5], we have introduced a new static analysis for stack inspection, specifically designed to tackle the issues raised by dynamic security policies. In our approach, programs are represented by *control flow graphs*. These graphs are extracted from actual bytecode through available control flow analyses; they feature primitive constructs for method invocation, exceptions, and access control based on stack inspection. Our static analysis takes as input a control flow graph, and computes an abstract graph, whose nodes are pairs of the form $\langle n, \gamma \rangle$ and edges are labelled by $\varphi$, where:

- $n$ is a method invocation or return, or an access control test;
- $\gamma$ is an *access control context*, i.e. the set of protection domains visited after the last privileged call (if any);
- $\varphi$ is a traversability condition for the edge.

The traversability conditions are used to associate each abstract node $\langle n, \gamma \rangle$ with a predicate $\Phi(n, \gamma)$, telling which conditions the security policy must satisfy in order for the node to be reachable.

Our static analysis enjoys a remarkable property: it is *parametric* with respect to the security policy in force, hence there is no need to recompute it each time the access rights are modified. The correctness of the static analysis has been formally proved: if the actual security policy allows for an execution leading to a node $n$ with access control context $\gamma$, then a node $\langle n, \gamma \rangle$ in the abstract graph exists, and it is such that the security policy satisfies the condition $\Phi(n, \gamma)$.

In this paper, we exploit our static analysis to rigorously specify how method inlining transformations can be performed without altering the security of applications. Our main contribution here is the introduction of two strategies to detect the inlineable method calls, i.e. those calls which can be safely replaced by a copy of the called method. The first strategy is complete: it detects all and only the method calls that can be safely inlined. The second strategy is not complete, yet compositional. Most importantly, it can be used to optimize software libraries: if the inlining of a method is predicted to be safe in a library, then that inlining will be such in every program using that library.

This paper will not present in detail the formal machineries of our approach (see [5] for the technical details). With the help of some examples, we point out which kind of security flaws can arise when performing method inlining optimizations. Then, we describe the basic features of our approach, and we outline how abstract graphs are computed. Finally, we explain how our static analysis can effectively detect safe method inlinings.

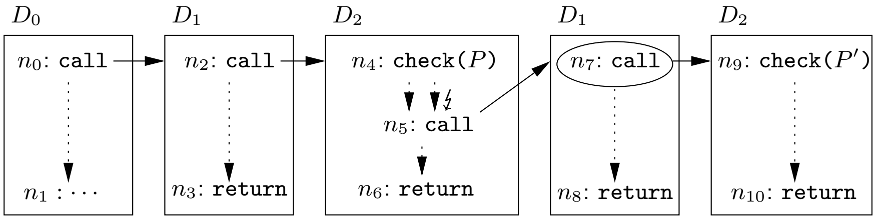


**Fig. 1.** A control flow graph

## 2 A motivating example

We model programs as *control flow graphs* (CFGs for short). Their nodes represent the primitives relevant for stack inspection, i.e. method calls, returns and access control tests, and their arcs represent the flow of control. Consider for example the CFG in Fig. 1: it has five methods, depicted as solid boxes. Each method comprises a set of nodes connected by dotted edges, which model intraprocedural flow. Methods are associated to protection domains (here $D_0, D_1, D_2$). Indeed, each CFG is associated with a security policy, which grants a set of permissions to each protection domain; this mapping may vary over time. There are three kinds of nodes: method calls ($n_0, n_2, n_5, n_7$), returns ($n_3, n_6, n_8, n_{10}$) and check nodes ($n_4, n_9$), testing whether a permission is granted by the security policy in force or not. Solid edges represent interprocedural flow. Dotted edges can be of two kinds: *transfer edges* (e.g. $n_2 \dashrightarrow n_3$), which correspond to sequencing, and *catch edges* (here $n_4 \dashrightarrow_{\lightning} n_5$), which correspond to exception handling. Circled calls (here $n_7$) are privileged, i.e. they enable all of their privileges.

We do not define how CFGs are extracted from actual programs. This construction is well understood and algorithms and tools exist for it; see for example [14, 18]. Note that CFGs hide any data flow information, and are therefore approximated. 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, the "if" construct is rendered in CFGs as non-deterministic choice; safety imposes that both branches are represented, even in the cases when the same branch is always taken at run-time.

Dynamic dispatching in object-oriented languages is another source of approximation. When a program invokes a method on an object $O$, the run-time environment has to choose among the various implementations of that method. The decision is not based on the declared type of $O$, but on the actual class $O$ belongs to, which is unpredictable at static time. To be safe, CFGs over-approximate the set of methods that can be invoked at each program point.

We require our CFGs to obey some well-formedness constraints: (1) check nodes have no outgoing call edges; (2) return nodes do not have outgoing edges; (3) each method has exactly one entry point; (4) nodes in the same method are in the same protection domain (we write $Dom(n)$ for the domain of the method $n$ belongs to); (5) each call has a return; (6) for each exception handler, some

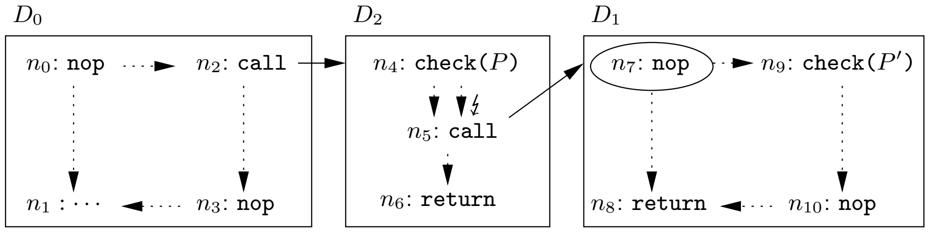


**Fig. 2.** The CFG in Fig. 1 after the inlining of $n_0$ and $n_7$.

node may propagate an exception. These constraints reflect some peculiarities of Java-like bytecode: hence, CFGs extracted from bytecode always satisfy them.

The operational semantics of CFGs is defined in terms of execution traces that are possible under a security policy. Each state in a trace is made of a call stack, plus a boolean tag ($\lightning$) indicating whether an exception has been thrown and not caught yet. Stack inspection is modelled by the predicate $\sigma \vdash_{Perm} P$, which tells whether the permission $P$ is granted to the call stack $\sigma$ under the policy *Perm* or not. We write $\sigma \rhd \sigma'$ when there is a transition from $\sigma$ to $\sigma'$ (see [5] for details). The predicate is true iff *Perm* grants $P$ to each protection domain in the set $\Gamma(\sigma)$ of domains that have been visited after the last privileged call in $\sigma$ (if any). We call $\Gamma(\sigma)$ the *access control context* of $\sigma$.

For example, consider CFG $G$ in Fig. 1 with the following security policy:

$$Perm(D_0) = \{P\} \qquad Perm(D_1) = \{P'\} \qquad Perm(D_2) = \{P, P'\}$$

The execution traces of $G$ under the policy *Perm* are of the form:

$$\begin{array}{l}
\rhd [n_0] \rhd [n_0 : n_2] \rhd [n_0 : n_2 : n_4] \nvdash_{Perm} P \\
\rhd [n_0 : n_2 : n_4]\lightning \rhd [n_0 : n_2 : n_5] \rhd [n_0 : n_2 : n_5 : n_7] \rhd [n_0 : n_2 : n_5 : n_7 : n_9] \vdash_{Perm} P' \\
\rhd [n_0 : n_2 : n_5 : n_7 : n_{10}] \rhd [n_0 : n_2 : n_5 : n_8] \rhd [n_0 : n_2 : n_6] \rhd [n_0 : n_3] \rhd [n_1]
\end{array}$$

Assume that the method invocations at $n_0$ and $n_7$ are our candidates for method inlining. The CFG $G'$ in Fig. 2 displays the effect of the inlining transformation (the `nop` nodes represent operations without effects). The execution traces of $G'$ under the policy *Perm* are of the form:

$$\begin{array}{l}
\rhd [n_0] \rhd [n_2] \rhd [n_2 : n_4] \vdash_{Perm} P \\
\rhd [n_2 : n_5] \rhd [n_2 : n_5 : n_7] \rhd [n_2 : n_5 : n_9] \nvdash_{Perm} P' \\
\rhd [n_2 : n_5 : n_9]\lightning \rhd [n_2 : n_5]\lightning \rhd [n_2]\lightning \rhd []\lightning
\end{array}$$

Note that the behaviour of the two CFGs differs substantially. First, the security check for $P'$ at $n_9$ succeeds in the original CFG (Fig. 1), while it fails after the inlining (Fig. 2). More dangerously, the check for $P$ at $n_4$, which would fail in the original CFG, turns out to succeed after the inlining, thus leading to potentially unsafe behaviour.

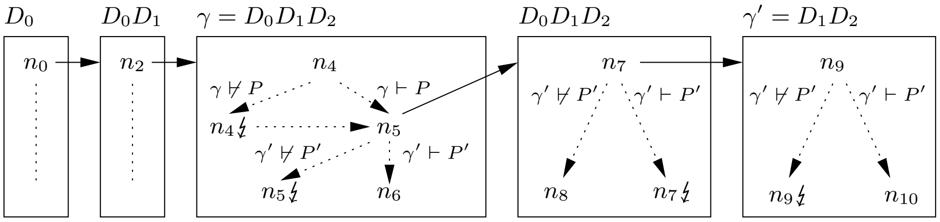


**Fig. 3.** Abstract graph for the CFG in Fig. 1 (portions). Read each node $n$ enclosed in a solid box labelled $\gamma$ as the abstract node $n\gamma$.

## 3 The Security Context Analysis

In [5] we have defined a static analysis over CFGs: given a CFG $G$, it computes an abstract graph $G^\sharp$ that mimicks the evolution of the security contexts in the traces of $G$. An abstract node $n^\sharp$ is a pair $n\gamma$ (plus a possible exception flag ↯). Intuitively, it represents a call stack with top node $n$ and access control context $\gamma$. An abstract edge $n^\sharp \xrightarrow{\varphi} m^\sharp$ models an execution that can flow from $n^\sharp$ to $m^\sharp$ if the security policy in force satisfies the condition $\varphi$. To get the flavour, consider the edge $n_9 \dashrightarrow n_{10}$ in Fig. 1. Its abstract counterpart is the edge $n_9\gamma' \xrightarrow{\gamma' \vdash P'} n_{10}\gamma'$ in Fig. 3, representing that an execution can flow from $n_9$ to $n_{10}$ when the security policy grants the permission $P'$ to the context $\gamma'$.

Each path $\pi$ in the abstract graph represents then a possible execution in the original CFG: an execution is admissible under a security policy *Perm*, provided that *Perm* satisfies $\Phi(\pi)$, i.e. the conjunction of all the traversability conditions labelling the edges of $\pi$. For instance, consider the following path $\pi$ from Fig. 3:

$$n_0D_0 \to n_2D_0D_1 \to n_4\gamma \xrightarrow{\gamma \vdash P} n_5\gamma \xrightarrow{\gamma' \vdash P'} n_6\gamma$$

This path represents all the execution traces on the form:

$$[n_0] \rhd [n_0 : n_2] \rhd [n_0 : n_2 : n_4] \rhd [n_0 : n_2 : n_5] \rhd \cdots \rhd [n_0 : n_2 : n_6]$$

These traces are admissible under any security policy *Perm* that satisfies the condition $\Phi(\pi) = (\gamma \vdash P) \wedge (\gamma' \vdash P')$, i.e. *Perm* grants $P$ to $\gamma$ and $P'$ to $\gamma'$.

The root of the abstract graph is $\langle n_\varepsilon, Dom(n_\varepsilon), \mathit{false}\rangle$, where $n_\varepsilon$ is the unique entry point of $G$. The abstract graph is constructed starting from the root, and then applying the rules in Table 1. Intuitively, they say that:

(a) if there is a non-privileged call from node $n$ to $n'$, and the access control context before the call is $\gamma$, then the context after the call consists of $\gamma$ plus the protection domain of $n'$. Solid edges in $G^\sharp$ represent interprocedural abstract flow, and we assign them unit weight;
(b) if $n$ is a privileged call to $n'$, then the context after the call consists of the protection domains of both $n$ and $n'$;

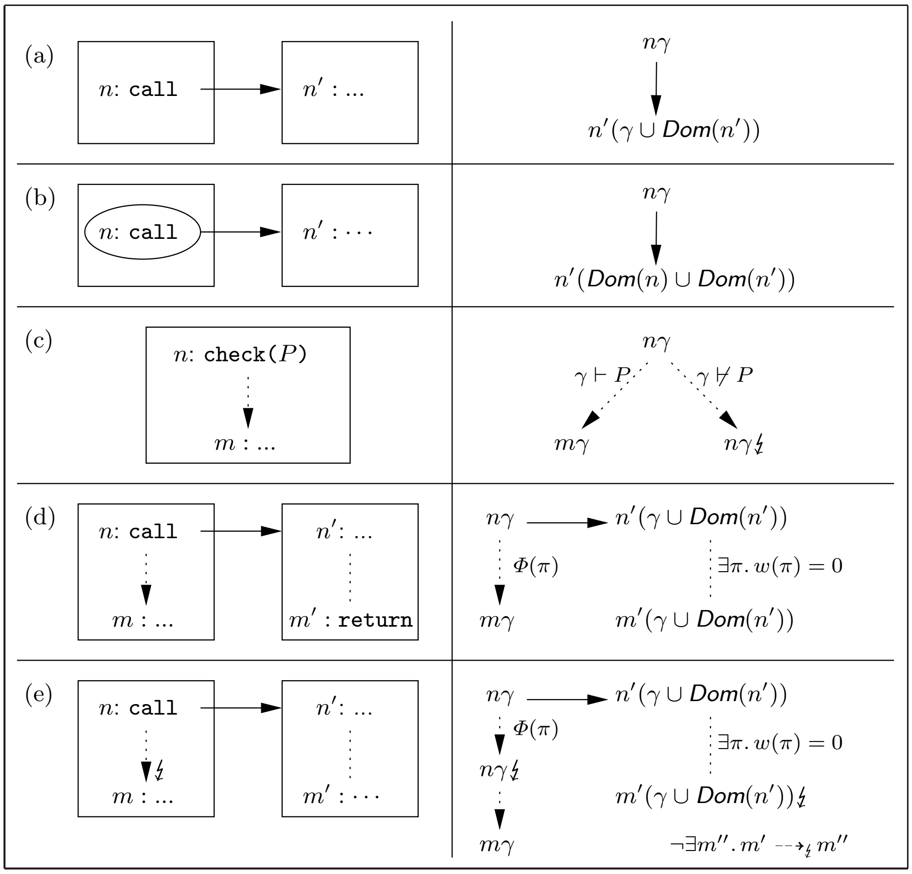


**Table 1.** CFG primitives (left) and their abstract counterparts (right).

(c) if $n$ tests for permission $P$ in the context $\gamma$, then there exist one edge labelled $\gamma \vdash P$ leading to $m$, and one edge labelled $\gamma \nvdash P$ leading to an exception. Both of them preserve the context. Dotted edges model intraprocedural abstract flow, and we assign them null weight;

(d) if $n$ calls $n'$ in the context $\gamma$, and there is a transfer edge from $n$ to $m$, then an abstract edge from $n\gamma$ to $m\gamma$ exists, provided the following holds. There exists a null-weight path $\pi$ in $G^\sharp$ from $n'(\gamma \cup \mathit{Dom}(n'))$ to a node $m'(\gamma \cup \mathit{Dom}(n'))$ where $m'$ is a return in the same method of $n'$.

(e) if $n$ calls $n'$ in the context $\gamma$, then an abstract edge from $n\gamma$ to $n\gamma\lightning$ exists, provided that the following holds. There exists a null-weight path $\pi$ in $G^\sharp$ from $n'(\gamma \cup \mathit{Dom}(n'))$ to a node $m'(\gamma \cup \mathit{Dom}(n'))\lightning$ where $m'$ and $n'$ are in the same method. Moreover, if $m$ is an exception handler for $n$, then there exists an abstract edge from $n\gamma\lightning$ to $m\gamma$.

For the CFGs extracted from actual bytecode, our construction gives abstract graphs whose size is linear in the number of nodes of the original CFG. This is due to some characteristics of actual bytecode. First, the number of protection domains can be considered as a constant, because it depends on static properties of the loaded code (i.e. code origin and digital signatures). Second, the number of security checks in CFGs is usually small: indeed, access control tests are only inserted to guard methods accessing critical resources.

Our analysis is *sound* and *complete* w.r.t. the operational semantics of CFGs. Soundness ensures that, for each execution trace in $G$, there exists a corresponding path in $G^\sharp$ which mimicks the evolution of the access control contexts. Completeness ensures that, for each path in $G^\sharp$ and security policy $Perm$ that satisfies its traversability condition, there exists a corresponding execution trace in $G$ under the policy $Perm$. Note that completeness is only up to the precision of the CFG, which is a safe, approximated model of the analyzed program.

Formally, let $\tau$ be a trace and $\Phi(\tau)$ be the conjunction of the access control tests encountered during its execution. Then, we have the following:

**Theorem 1 (Soundness).** *Let* $\langle G, Perm \rangle \triangleright \tau = [\,] \triangleright \cdots \triangleright \sigma : n$. *Then, there exists a path* $\pi$ *reaching* $n(\Gamma(\sigma) \cup Dom(n))$ *such that* $\Phi(\pi) = \Phi(\tau)$.

**Theorem 2 (Completeness).** *Let* $\pi$ *be a path reaching* $n\gamma$ *and* $Perm \models \Phi(\pi)$. *Then, there exist* $\tau, \sigma$ *s.t.* $\langle G, Perm \rangle \triangleright \tau \triangleright \sigma : n$, *with* $\Gamma(\sigma) \cup Dom(n) = \gamma$ *and* $\Phi(\tau \triangleright \sigma : n) = \Phi(\pi)$.

Our analysis supports a form of incremental computation, though not in a fully compositional way. This is particularly useful for *dynamic linking* of code – the mechanism which allows a program to be extended at run-time. Our program model does not directly support this feature: so we require the CFG construction algorithm to correctly link the relevant CFGs, e.g. as in [21]. 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. The incremental construction of our analysis exploits the fact that adding new executable traces to a CFG never affects the analysis of the old ones.

## 4 Method inlining

Our analysis gives us the means to compute the set of method invocations that can be safely inlined. Intuitively, a method invocation can be inlined if the outcome of the security checks is not affected by ignoring the protection domain of the inlined method. We adopt the so-called *original version inlining* approach [15], 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 $\dot{n}$ be the node candidate for inlining, and $\dot{n} \longrightarrow n'$. We assume that the method invocation represented by $\dot{n}$ can be statically dispatched, i.e. it has exactly one callee ($n'$). As a preprocessing step, we assign a fresh name to the protection domain of the (duplicated) method candidate for inlining.

We define below two notions of inlineability. Some notation helps. We write $\mu(n)$ for the method $n$ belongs to, i.e. the set of nodes that are connected to $n$ intraprocedurally. We write $n\gamma \rightrightarrows_{\varphi} n'\gamma'$ when there is a path $\pi$ in $G^\sharp$ from $n\gamma$ to $n'\gamma'$ beginning with a solid edge, and with $\varphi = \Phi(\pi)$. We write $Inl_{\dot{n}}(\gamma)$ for the context obtained from $\gamma$ by replacing the domain of the (unique) callee of $\dot{n}$ with $Dom(\dot{n})$. We write $Perm \models \Phi(n\gamma)$ when there is a path $\pi$ reaching $n\gamma$ such that the policy $Perm$ satisfies all the traversability conditions on the edges of $\pi$.

Let $m$ be a check for permission $P$. We say that $\dot{n} \longrightarrow n'$ is:

- *weakly inlineable* w.r.t. $Perm$ if, whenever $\dot{n}\gamma \rightrightarrows_{\varphi} m\gamma'$ and $Perm \models \Phi(\dot{n}\gamma) \wedge \varphi$,

$$Perm \models \gamma \vdash P \iff Perm \models Inl_{\dot{n}}(\gamma) \vdash P$$

- *strongly inlineable* w.r.t. $Perm$ if, whenever $\dot{n}\gamma \rightrightarrows m\gamma'$ and $Dom(n') \in \gamma'$,

$$P \in Perm(\dot{n}) \iff P \in Perm(n')$$

Weak inlineability requires that all the security checks *actually* reachable after the inlined call have the same outcome before and after the transformation. Strong inlineability requires that the security policy agrees on the protection domains of $\dot{n}$ and $n'$ for all the permissions *possibly* checked after the inlined call (that include those actually reachable). It is easy to show that strong inlineability implies weak inlineability (the viceversa is not true).

As an example, let the call at $n_0$ in Fig. 1 be our candidate for inlining. Recall the security policy used in the previous examples:

$$Perm(D_0) = \{P\} \qquad Perm(D_1) = \{P'\} \qquad Perm(D_2) = \{P, P'\}$$

By the abstract graph in Fig. 3, we have that $n_0 D_0 \rightrightarrows n_4 \gamma$. The security policy does not grant $P$ to $\gamma = D_0 D_1 D_2$, but it grants $P$ to $Inl_{n_0}(\gamma) = D_0 D_2$: therefore, $n_0$ is not weakly inlineable. Consider now the following security policy:

$$Perm(D_0) = \{P\} \qquad Perm(D_1) = \{P'\} \qquad Perm(D_2) = \{P'\}$$

Since $\gamma = D_0 D_1 D_2 \not\vdash P$ and $Inl_{n_0}(\gamma) = D_0 D_2 \not\vdash P$, $n_0$ is weakly inlineable. However, $n_0$ is not strongly inlineable w.r.t. $Perm$, because $P \in Perm(D_0)$ but $P \notin Perm(D_1)$. Strong inlineability requires security policies that agree on $P$ for both $D_0$ and $D_1$.

To define the effect of the method inlining transformation on CFGs, we act on the operational semantics, instead of replacing the method invocation $\dot{n}$ with a copy of the inlined method (as done in Fig. 2). There are only two effects on the original CFG after the inlining of $\dot{n}$. First, the protection domain of $\dot{n}$ is substituted for that of $n'$. Second, the calls in the method of $n'$ become privileged when $\dot{n}$ is such. We call $inl_{\dot{n}}(G)$ the CFG obtained from $G$ after this procedure. We specify the effect of the inlining of $\dot{n}$ on call stacks by the function $inl_{\dot{n}}$ defined in Table 2. Given a state $\sigma$, $inl_{\dot{n}}(\sigma)$ is obtained by removing all the occurrences

$$\frac{}{inl_{\dot{n}}([]) = []}\,[inl_1] \qquad \frac{inl_{\dot{n}}(\sigma) = \dot{\sigma} \quad top(\sigma) \neq \dot{n}}{inl_{\dot{n}}(\sigma : n') = \dot{\sigma} : n'}\,[inl_2] \qquad \frac{inl_{\dot{n}}(\sigma) = \dot{\sigma}}{inl_{\dot{n}}(\sigma : \dot{n} : n') = \dot{\sigma} : n'}\,[inl_3]$$

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n' \quad n \neq \dot{n}}{\sigma : n \rhd^{\dot{n}} \sigma : n : n'}\,[\rhd_{call1}] \qquad \frac{\ell(\dot{n}) = \texttt{call} \quad \dot{n} \longrightarrow n'}{\sigma : \dot{n} \rhd^{\dot{n}} \sigma : n'}\,[\rhd_{call2}]$$

$$\frac{\ell(n') = \texttt{return} \quad n \dashrightarrow m \quad \dot{n} \not\longrightarrow \mu(n')}{\sigma : n : n' \rhd^{\dot{n}} \sigma : m}\,[\rhd_{return1}] \qquad \frac{n \not\dashrightarrow_{\lightning} \quad \dot{n} \not\longrightarrow \mu(n')}{\sigma : n\lightning \rhd^{\dot{n}} \sigma\lightning}\,[\rhd_{propagate1}]$$

$$\frac{\ell(n') = \texttt{return} \quad \dot{n} \dashrightarrow m \quad \dot{n} \longrightarrow \mu(n')}{\sigma : n' \rhd^{\dot{n}} \sigma : m}\,[\rhd_{return2}] \qquad \frac{n \not\dashrightarrow_{\lightning} \quad \dot{n} \longrightarrow \mu(n')}{\sigma : n\lightning \rhd^{\dot{n}} \sigma : \dot{n}\lightning}\,[\rhd_{propagate2}]$$

**Table 2.** Effect of inlining $\dot{n}$ on call stacks (top) and transitions (bottom).

of $\dot{n}$ in $\sigma$ (except when $\dot{n}$ is in top position). The operational semantics of a CFG after the inlining of $\dot{n}$ is defined by the transition relation $\rhd^{\dot{n}}$ in Table 2. For instance, a method invocation proceeds as usual when the calling node is not $\dot{n}$ (rule $\rhd^{\dot{n}}_{call1}$); otherwise $\dot{n}$ is removed from the call stack ($\rhd^{\dot{n}}_{call2}$).

The following theorem states the soundness of both kinds of inlining: each trace in the original CFG corresponds to an "equivalent" trace in its inlined version (recall that, if $\dot{n}$ is strongly inlineable, then it is also weakly inlineable).

**Theorem 3.** *The call $\dot{n}$ is weakly inlineable in $G$ w.r.t. Perm if and only if:*

$$\langle G, Perm\rangle \rhd \tau \quad\iff\quad \langle inl_{\dot{n}}(G), Perm\rangle \rhd^{\dot{n}} inl_{\dot{n}}(\tau)$$

This theorem also implies that notion of weak inlining is complete: if the traces before and after the inlining of $\dot{n}$ are equivalent, then $\dot{n}$ turns out to be weakly inlineable. This does not holds for strong inlineability, instead.

We now state two compositionality results about method inlining. Note that these results are only valid for strong inlining.

First we define the notion of *linked graph* $G \bowtie_{\tilde{E}} G'$ of two CFGs $G$ and $G'$. Let $\tilde{E}$ be a set of *resolved edges* between $G$ and $G'$, i.e. call edges $n \longrightarrow m$ such that $n, m$ do not belong both to the same CFG (this set is determined by the CFG construction algorithm). Then, $G \bowtie_{\tilde{E}} G'$ contains the nodes and the edges from both $G$ and $G'$, plus the resolved edges $\tilde{E}$.

Our first compositionality result is given by Theorem 4. Suppose we have detected that a method invocation is strongly inlineable in a library (represented by the CFG $G$) under a security policy *Perm*. Then, the inlineability holds in any program using that library, i.e. in any CFG $G \bowtie_{\tilde{E}} G'$, where $\tilde{E}$ are the resolved calls from $G'$ to the methods in the library.

**Theorem 4.** *Let $\dot{n} \longrightarrow n'$ be strongly inlineable in $G$ w.r.t. Perm, and let $\tilde{E}$ be s.t. $(m,n) \in \tilde{E} \implies n$ is the entry point of $G$, $G'$ be s.t. $Dom(n) \neq Dom(n')$ for each $n \in G'$. Then, $\dot{n} \longrightarrow n'$ is strongly inlineable in $G \bowtie_{\tilde{E}} G'$ w.r.t. Perm.*

The following theorem still deals with compositionality, but in the opposite direction. Suppose $\dot{n}$ is strongly inlineable in $G$. Recall that $G$ can have dynamically dispatched method calls, which can be resolved to methods outside $G$ when a new piece of code (say $G'$) is linked to $G$. Theorem 5 states that $\dot{n}$ is inlineable in the linked graph $G \bowtie_{\tilde{E}} G'$, provided that the security policy agrees on the protection domains of $\dot{n}$ and $n'$, for all the permissions checked in a node reachable from the entry point of $G'$.

**Theorem 5.** *Let $\dot{n} \longrightarrow n'$ be strongly inlineable in $G$ w.r.t. Perm, and $\tilde{E}$ be s.t. $(m,n) \in \tilde{E} \implies n = n'_\varepsilon$ (the entry point of $G'$), $G'$ be s.t. $Dom(n) \neq Dom(n')$ for each $n \in G'$. Then, $\dot{n} \longrightarrow n'$ is strongly inlineable in $G \bowtie_{\tilde{E}} G'$ w.r.t. Perm if, whenever $m$ is a check for $P$, $n'_\varepsilon Dom(n'_\varepsilon) \rightrightarrows m\gamma$ and $Dom(n') \in \gamma$, it happens that $P \in Perm(\dot{n}) \iff P \in Perm(n')$.*

## 5 Conclusions and related work

We have developed a technique to perform program transformations in the presence of stack inspection and dynamic security policies. The technique relies on the definition of our Security Context Analysis. The analysis is sound and complete with respect to the control flow graphs derived from the bytecode (recall that these graphs safely approximate the actual behaviour). Our analysis makes various optimizations possible. We focussed here on method inlining; elimination of redundant checks and of dead code are also possible, see [5]. Our formalization of method inlining as a transformation of the operational semantics makes our technique straightforwardly applicable to general tail call elimination. It is worthwhile noting that our analysis can take advantage of the control flow graphs generated by the just-in-time optimizers, e.g. the HotSpot compilers embedded in the latest Java Virtual Machines [22]. This would also make our technique directly exploitable by these tools, e.g. to produce larger methods by inlining, so allowing for further optimizations.

There are some differences between our model and the security models of Java [13] and .NET [17]. In particular, we do not:

- allow 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);
- consider the inheritance of the access control context from a parent thread to its children;
- consider *user-centric* security policies, which grant permissions according to who is actually running the code;
- model some advanced features like reflection and native methods.

Many authors advocated the use of static techniques in order to understand and optimize stack inspection, among them ourselves [2–6]. To the best of our knowledge, our current proposal is the first one that can deal with dynamic security policies and method inlining in the presence of dynamic loading of code.

Besson, Jensen, Le Mètayer and Thorn [8] formalize classes of security properties through a linear-time temporal logic. Model checking is used to prove that local security checks enforce a given global security policy. Based on the same model, [7] develops a static analysis that computes, for each method, the set of its *secure calling contexts* with respect to a given global security property. For method inlining, this technique is even too powerful, as the information about calling contexts is unnecessary.

Esparza, Kučera and Schwoon [11] formalize stack inspection in terms of model checking pushdown systems. Obdržálek [19] uses the same technique to accurately model Java exception handling. A suitable combination of the two will then be an alternative approach to ours. Since our model is specifically tailored on stack inspection, we think that our analysis may be implemented and exploited more efficiently than such a general method. Like in [8], the latter approach seems to suffer from dynamic linking, in particular when some program transformations have to be revalidated at run-time (e.g. method inlinings).

Exploiting the access control logic of [1], Wallach, Appel and Felten [23] 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 (at the cost of more expensive method calls).

Pottier, Skalka and Smith [20] address the problem of stack inspection in $\lambda_{sec}$, a typed lambda calculus enriched with primitive constructs for enforcing security checks and managing permissions, and no exception handling. Stack inspection never fails on a well typed program, because the set of permissions granted at run-time always includes the security context.

Fournet and Gordon [12] investigate the problem of establishing the correctness of program transformations in the presence of stack inspection. 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 on the (somewhat limited) security properties actually guaranteed by stack inspection. Here, we are more concerned with efficient (semantically-based) optimization procedures to be used on the field, rather than with a general reasoning framework.

Koved, Pistoia and Kershenbaum [16] address the problem of computing the set of permissions a class needs in order to execute without throwing security exceptions. The analysis is built over *access rights invocation graphs*. These flow graphs are context-sensitive: each node is associated also with its *calling context*, i.e. with its target method, receiver and parameters values. In this way, the analysis in [16] can deal with parametric permissions and multi-threading. Our approach can gain precision through the exploitation of these graphs.

## References


1. M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Trans. on Programming Languages and Systems*, 1993.
2. M. Bartoletti, P. Degano, and G. L. Ferrari. Static analysis for stack inspection. In *Proc. Concoord*, ENTCS volume 54, 2001.
3. M. Bartoletti, P. Degano, and G. L. Ferrari. Security-aware program transformations. In *Proc. 8th Italian Conference on Theoretical Computer Science*, 2003.
4. M. Bartoletti, P. Degano, and G. L. Ferrari. Static analysis for eager stack inspection. In *Workshop on Formal Techniques for Java-like Programs*, 2003.
5. M. Bartoletti, P. Degano, and G. L. Ferrari. Program transformations under dynamic security policies. *To appear in ENTCS*, 2004.
6. M. Bartoletti, P. Degano, and G. L. Ferrari. Stack inspection and secure program transformations. *To appear in International Journal of Information Security*, 2004.
7. F. Besson, T. de Grenier de Latour, and T. Jensen. Secure calling contexts for stack inspection. In *Proc. 4th Conference on Principles and Practice of Declarative Programming*, 2002.
8. F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security*, 2001.
9. J. Clemens and M. Felleisen. A tail-recursive semantics for stack inspections. In *Proc. 12th European Symposium on Programming*, 2003.
10. U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, 2000.
11. J. Esparza, A. Kučera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. In *Proc. 4th International Symposium on Theoretical Aspects of Computer Software*, 2001.
12. C. Fournet and A. D. Gordon. Stack inspection: theory and variants. *ACM Transactions on Programming Languages and Systems*, 2003.
13. L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation.* Addison-Wesley, 1999.
14. D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 2001.
15. O. Kaser and C. R. Ramakrishnan. Evaluating inlining techniques. *Computer Languages*, 1998.
16. L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. In *Proc. 17th ACM conference on Object-oriented Programming, Systems, Languages, and Applications*, 2002.
17. Microsoft Corp. *.NET Framework Developer's Guide: Securing Applications.*
18. F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.
19. J. Obdržálek. Model checking java using pushdown systems. In *Workshop on Formal Techniques for Java-like Programs*, 2002.
20. F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. In D. Sands, editor, *Proc. 10th European Symposium on Programming*, 2001.
21. A. Souter and L. Pollack. Incremental call graph reanalysis for object-oriented software maintenance. In *Proc. IEEE Int. Conf. on Software Maintenance*, 2001.
22. Sun Microsystems. *The Java HotSpot Virtual Machine (Technical White Paper).*
23. D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM TOSEM*, 2001.