Static analysis for eager stack inspection ⋆

Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari

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

Abstract. The paper focuses on stack inspection, a mechanism for the implementation of security-aware programming languages. We introduce a static analysis which safely approximates the set of access rights granted to code at run-time. This analysis allows for fast implementations of stack inspection which based on an (hyper-) eager strategy.

1 Introduction

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*.

There are at least two strategies for implementing stack inspection:

- in the *eager evaluation* strategy, the set of granted permissions is updated at each method call (and return).
- in the *lazy evaluation* strategy, the call stack is retrieved and inspected only when an access control is performed.

In the eager strategy, security checks can be performed very efficiently; on the other hand, there is an overhead at each cross-domain method invocation. Since security checks are statistically less frequent than cross-domain calls, actual implementations of the Java virtual machine adopt the lazy strategy. However, this choice could be expensive. First, the run-time overhead due to the analysis of stack frames may grow very high. Second, lazy stack inspection deeply affects standard program optimizations, such as method inlining and (general) tail call elimination.

We consider here an idealized object-oriented language with primitive constructs for method invocations, exceptions, and access control. We represent programs by *control flow graphs*, an abstract programming model not tied to any particular programming language.

We define a control flow analysis [11] over these graphs, called *Trace Permissions Analysis*. Intuitively, the analysis computes, for any program point and any execution reaching that point, the set of permissions granted at run-time.

⋆ Work partially supported by EU project DEGAS (IST-2001-32072), MIUR project "MEtodi FormalI per la Sicurezza e il TempO" (MEFISTO), and MIUR project "Network Aware Programming: Object, Languages, Implementation" (NAPOLI).

The Trace Permissions Analysis allows for effective implementations of eager stack inspection, on which this paper concentrates. Also, the analysis provides us with the basis for several code optimizations, such as method inlining, tail call elimination, removal of redundant security checks, and dead-code elimination.

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 a detailed description of the model, the proofs of our statements, some illustrative examples, and for code optimizations for the lazy strategy.

2 The program model

We model Java-like 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 e.g. [9, 11, 15, 16].

By construction, CFGs hide any data flow information, and are therefore approximated; typically, switching 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 if 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. CFGs are defined as follows.

Definition 1. *A CFG $\langle N \cup \{n_\varepsilon\}, E, Priv, Dom \rangle$ 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 $n_\varepsilon \notin N$ plays the technical role of a single, isolated entry point.*

- $E \subseteq (N \cup \{n_\varepsilon\}) \times N$ *is the set of edges. Edges are partitioned into four sets:* entry edges $\bullet\!\!\longrightarrow n$, *that represent the entry points of a program;* call edges $n \longrightarrow n'$, *which model* interprocedural *flow;* transfer edges $n \dashrightarrow n'$, *which correspond to sequencing; and* catch edges $n \dashrightarrow_{\natural} n'$, *which correspond to exception handling. The last two kinds of edges represent* intraprocedural *flow. The set of entry edges contains all pairs (n_ε, n) where n is a program entry point. The n_ε element is the source of entry edges, only.*
- *Priv* $: N \to$ *Bool tells whether a node enables its privileges or not.*
- *Dom* $: N \to \mathcal{D}$ *is a mapping from nodes to protection domains.*

When unambiguous, we shall write $\langle N, E\rangle$ instead of $\langle N \cup \{n_\varepsilon\}, E, \mathrm{Priv}, \mathrm{Dom}\rangle$.

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

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:*

$$\varepsilon(\mu(n)) \quad = \quad \{\, n' \in \mu(n) \mid \bullet\!\!\longrightarrow n' \;\vee\; \exists m \in N.\; m \longrightarrow n' \,\}$$

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

$$\rho(n) \quad = \quad \{\, m \in N \mid \ell(m) = \texttt{return} \;\wedge\; n \longrightarrow \varepsilon(\mu(m)) \,\}$$

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

$$\xi(n) \quad = \quad \begin{cases} \{n\} & \text{if } \ell(n) = \texttt{check}(P) \\ \{\, \xi(n') \mid n \longrightarrow \varepsilon(\mu(n')) \;\wedge\; n' \not\dashrightarrow_{\natural} \,\} & \text{otherwise} \end{cases}$$

The set $\xi_1(n)$ of nodes that may propagate an exception to n is defined as:

$$\xi_1(n) \quad = \quad \{\, n' \mid n \longrightarrow \varepsilon(\mu(n')) \;\wedge\; n' \not\dashrightarrow_{\natural} \;\wedge\; \xi(n') \neq \emptyset \,\}$$

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.

$$\frac{\bullet\!\!\longrightarrow n}{[] \rhd [n]} \qquad \frac{\ell(n) = \texttt{call} \quad n \longrightarrow n'}{\sigma : n \rhd \sigma : n : n'} \qquad \frac{\ell(m) = \texttt{return} \quad n \dashrightarrow n'}{\sigma : n : m \rhd \sigma : n'}$$

$$\frac{\ell(n) = \texttt{check}(P) \quad \sigma : n \vdash P \quad n \dashrightarrow n'}{\sigma : n \rhd \sigma : n'} \qquad \frac{\ell(n) = \texttt{check}(P) \quad \sigma : n \nvdash P}{\sigma : n \rhd \sigma : n\lightning}$$

$$\frac{n \dashrightarrow_{\lightning} n'}{\sigma : n\lightning \rhd \sigma : n'} \qquad \frac{n \not\dashrightarrow_{\lightning}}{\sigma : n\lightning \rhd \sigma\lightning}$$

$$\frac{}{[] \vdash P} \qquad \frac{P \in \mathrm{Perm}(n) \quad \sigma \vdash P}{\sigma : n \vdash P} \qquad \frac{P \in \mathrm{Perm}(n) \quad \mathrm{Priv}(n)}{\sigma : n \vdash P}$$

Table 1. Operational semantics of CFGs.

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 $N^* \times \mathit{Bool}$.

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 $\lightning$ to the sequence of nodes, i.e. $\sigma\lightning$ abbreviates $\langle \sigma, \mathit{true} \rangle$. Pushing a node n on a stack σ is written as $\sigma : n$ (the infix operator : associates to the left).

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

$$\frac{}{G \rhd \langle [], \mathit{false} \rangle} \qquad \frac{G \rhd \langle \sigma, x \rangle \quad \langle \sigma, x \rangle \rhd \langle \sigma', x' \rangle}{G \rhd \langle \sigma', x' \rangle}$$

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

$$TP_{in}(n) = \bigcup_{(m,n)\in E} TP_{out}(m,n)$$

$$TP_{out}(m,n) = \begin{cases} \{\{\mathrm{Dom}(n)\}\} & \text{if } \bullet\!\!\longrightarrow n \\ \{\, \gamma \cup \{\mathrm{Dom}(n)\} \mid \gamma \in TP_{call}(m) \,\} & \text{if } m \longrightarrow n \\ TP_{trans}(m) & \text{if } m \dashrightarrow n \\ TP_{catch}(m) & \text{if } m \dashrightarrow_{\not\,} n \end{cases}$$

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

$$TP_{trans}(n) = \begin{cases} \{\, \gamma \in TP_{in}(n) \mid P \in \Pi(\gamma) \,\} & \text{if } \ell(n) = \texttt{check}(P) \\ \{\, \gamma \in TP_{in}(n) \mid \mathrm{Trans}(n, \{\mathrm{Dom}(n)\}) \,\} & \text{if } \ell(n) = \texttt{call}, \mathrm{Priv}(n) \\ \{\, \gamma \in TP_{in}(n) \mid \mathrm{Trans}(n, \gamma) \,\} & \text{otherwise} \end{cases}$$

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

$$\mathrm{Trans}(n,\gamma) \stackrel{def}{=} \exists m \in \rho(n).\ \gamma \cup \{\mathrm{Dom}(m)\} \in TP_{in}(m)$$
$$\mathrm{Catch}(n,\gamma) \stackrel{def}{=} \exists m \in \xi_1(n).\ \gamma \cup \{\mathrm{Dom}(m)\} \in TP_{catch}(m)$$

Table 2. Flow equations for the TP analysis.

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 σ if $\sigma \vdash P$.

3 The Trace Permissions Analysis

We introduce the Trace Permissions Analysis (TP), a static analysis over CFGs which approximates the access rights granted to each reachable state.

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 $\Gamma : N^* \to 2^{\mathcal{D}}$ is defined as follows:

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

The set of permissions granted to a security context γ is:

$$\Pi(\gamma) \quad = \quad \bigcap_{D \in \gamma} \mathrm{Perm}(D)$$

The permissions granted to the security context of a state σ are exactly the permissions granted to σ, 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 *Perm*, the analysis is specified by the set of equations $TP^{=}(G, Perm)$ in Table 2. A solution $\tau \models TP^{=}(G, Perm)$ is a 5-tuple $\tau = \langle \tau_{in}, \tau_{call}, \tau_{trans}, \tau_{catch}, \tau_{out} \rangle$ which satisfies all the equations. The purpose of the analysis is to find, for each node n, the set $\{\, \Gamma(\sigma : n) \mid G \rhd \sigma : n \,\}$.

Technically, TP is a forward, monotone data flow analysis with values in $2^{2^{\mathcal{D}}}$. Since both G and $\mathcal{D}$ 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:*

$$G \rhd \sigma : n \quad \Longrightarrow \quad \exists \gamma \in \tau_{call}(n).\ \gamma = \Gamma(\sigma : n)$$

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

$$\gamma \in \tau_{call}(n) \quad \Longrightarrow \quad \exists \sigma.\ G \rhd \sigma : n \ \wedge \ \gamma = \Gamma(\sigma : n)$$

The worklist algorithm which actually computes the (unique) minimal solution to the analysis has computational complexity $\mathcal{O}(c \cdot |N|) = \mathcal{O}(|N|)$. The constant c depends on the number of protection domains occurring in G: in the worst case, $c = 2^{3 \cdot |\mathcal{D}_G|}$, where $\mathcal{D}_G = \bigcup_{n \in N} \mathrm{Dom}(n)$. 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. The CFG construction algorithm is demanded to correctly perform the dynamic linking of the relevant CFGs, as in [14]. 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 briefly show how the incremental computation of the analysis is performed. Let $G = \langle N, E\rangle$, and assume that a solution τ to $TP^{=}(G, \mathit{Perm})$ is available when the CFG $G' = \langle N', E'\rangle$ is loaded. Through the CFG construction algorithm, we single out the set $E_{\bowtie}$ of *resolved calls* between G and G', i.e. those edges $n \longrightarrow m$ such that n, m do not belong both to the same CFG. Linking G and G' together yields the CFG $G \bowtie G' = \langle N \cup N',\ E \cup E' \cup E_{\bowtie}\rangle$. The analysis $\tau' \models TP^{=}(G \bowtie G', \mathit{Perm})$ is a refinement of τ. To compute it, the worklist algorithm adds to τ 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, $(n, m) \in E_{\bowtie}$. Moreover, the worklist must include all entry points of G', if any. Although our approach is not fully compositional, note that adding new executable paths to a CFG never affects the analysis of the old ones.

4 Eager stack inspection

We now specify an alternative implementation of eager stack inspection, which exploits our TP analysis to efficiently update the security contexts.

We adopt the *security passing style* of [18] to track the security context as an additional parameter of each method invocation. The type of this parameter is assumed to be one of the primitive integral types of the JVM (i.e. `byte`, `short`, `int` and `long`); accordingly, its size (in bits) is then $k \in \{8, 16, 32, 64\}$.

Choose a set $\mathcal{D}_0 = \{D_1, \ldots, D_{k-1}\} \subseteq \mathcal{D}$ of protection domains. If $\mathcal{D}$ has not enough elements, add the needed ones, and assign them arbitrary permissions.

Given a CFG G and a security policy *Perm*, a solution to $TP^{=}(G, \mathit{Perm})$ can be used to enumerate the set $\{\, \Gamma(\sigma) \mid G \triangleright \sigma \,\}$ of the reachable security contexts. Let $\gamma_0, \ldots, \gamma_p$ be such an enumeration. Represent now a security context γ_i as a k-bits array $\boldsymbol{\alpha}_i = \langle \alpha_{i,0}, \ldots, \alpha_{i,k-1}\rangle$, where:

- if $\gamma_i \subseteq \mathcal{D}_0$, then $\alpha_{i,0} = 0$ and, for each $j \in 1..k-1$, $\alpha_{i,j} = 1$ iff $D_j \in \gamma_i$.
- otherwise, $\alpha_{i,0} = 1$ and $\langle \alpha_{i,1}, \ldots, \alpha_{i,k-1}\rangle$ is the binary representation of i.

Security contexts are updated at each method invocation and return. The intuition is that $2^{\mathcal{D}_0}$ contains the contexts that we expect to occur with high probability in executions, and therefore require very efficient updating. These contexts are represented as arrays of bits: the i-th bit is set iff the protection domain D_i has been traversed.

The contexts outside $2^{\mathcal{D}_0}$ are represented by their indexes in the enumeration computed by the TP analysis. The transition function h between security contexts is cached in a hash table; h is computed as a side effect when constructing a solution to the TP analysis. Formally, $h(i, n, n') = j$ whenever γ_j is the context of the state obtained when the control flows from n to n', starting from a state with context γ_i. There is no need to store the entries of h where both γ_i and γ_j in $2^{\mathcal{D}_0}$.

Formally, the context updating operations are implemented as follows:

- on method invocation, let n be a call to $\mu(n')$ and the current security context be γ_i. If $\gamma_i \subseteq \mathcal{D}_0$ and $\mathrm{Dom}(n') = D_j$, then the new context is $\langle 0, \alpha_{i,1}, \ldots, \alpha_{i,k-1}\rangle \vee 2^{k-j-1}$ (bitwise or). That is, $\alpha_{i,j}$ is set to indicate that D_j has been traversed. Otherwise, the new context is $\boldsymbol{\alpha}_{h(i,n,n')}$.
- on method return, the context is retrieved from the popped call stack.

Security checks are performed by looking at the current context, instead of inspecting the call stack. The intuition is that, for the contexts in $2^{\mathcal{D}_0}$, a check for permission P_j succeeds iff no protection domain D_i with $P_j \notin \mathrm{Perm}(D_i)$ has been traversed.

More formally, let $P_1, \ldots, P_q$ be the set of permissions checked in G. For each $j \in 1..q$, we define a k-bits array $\boldsymbol{\beta}_j = \langle 0, \beta_{j,1}, \ldots, \beta_{j,k-1}\rangle$ as follows:

$$\beta_{j,i} \quad = \quad \begin{cases} 0 & \text{if } P_j \in \mathrm{Perm}(D_i) \\ 1 & \text{otherwise} \end{cases}$$

Let n be a check for permission P_j, γ_i be the current context, and let n' follow n sequentially. If $\gamma_i \subseteq \mathcal{D}_0$, then the check succeeds iff $\boldsymbol{\alpha}_i \wedge \boldsymbol{\beta}_j = \mathbf{0}$ (bitwise and). Otherwise, the check succeeds iff $h(i, n, n')$ is defined.

Compared with the lazy evaluation strategy, our technique involves an overhead at each method invocation: besides the cost of passing an additional parameter, we have to perform either a "bitwise or" operation (or a hash table lookup). Security checks require a "bitwise and" (or a hash table lookup), and are independent of the size of the call stack. As a matter of fact, a good choice of the set $\mathcal{D}_0$ is crucial, and possibly requires a statistical estimate of the frequency of contexts. The problem of statically determining the probability of execution traces in CFGs seems to be unsolvable in the general case. However, we hope that a rough approximation of the most frequent states will suffice. Some form of dynamic analysis can possibly be carried on to refine that approximation.

5 Conclusions and related work

We have developed a static analysis for the Java bytecode. The Trace Permissions Analysis 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 CFGs derived from the bytecode (however, these graphs only approximate the actual behavior). Our analysis makes various optimizations possible; here, we only focused on eager stack inspection. We restricted our attention to Java, but the same techniques can be applied to other programming languages whose authorization mechanisms rely on stack inspection (e.g. C♯ [19]).

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

Besson, Jensen, Le Mètayer and Thorn [4] 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. Model checking is then used to prove that local security checks enforce a given global security policy. The verification method can be used to optimize stack inspection by eliminating the redundant checks. This is done by computing, for each node n, the set $\{ P \in \mathcal{P}_{check} \mid G \rhd \sigma : n \ \wedge \ \sigma : n \vdash P \}$, where $\mathcal{P}_{check}$ is the set of permissions checked in G. The computational complexity of the method is $\mathcal{O}(c \cdot |N|)$, where the constant c depends on the cardinality of $\mathcal{P}_{check}$ (in the worst case, $c = 2^{|\mathcal{P}_{check}|}$). Therefore, our TP 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 $\mathcal{P}_{check} \subseteq \mathcal{P}$. 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 [18] 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. Instead, in our approach, each security operation costs as a hash lookup *in the worst case*, while, in the optimal case, it costs as a cheap bitwise operation. A further difference w.r.t our approach is that [18] assumes that the whole program is available at compilation time.

Pottier, Skalka and Smith [12] address the problem of stack inspection in λ_{sec}, a typed lambda calculus enriched with primitive constructs for enforcing security checks and managing permissions. 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 a λ-calculus enriched with access control primitives, as in [12]. They study how stack inspection affects program behavior, proving that certain function inlinings and tail-call eliminations are correct. The theory is used to reason about the (somewhat limited) security properties guaranteed by stack inspection.

References

1. M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 4(15):706–734, Sept. 1993.
2. M. Bartoletti, P. Degano, and G. Ferrari. Stack inspection and program transformations. `http://www.di.unipi.it/~bartolet/static-stack.ps`, 2003.
3. L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Foundations of Computer Security (FCS '02)*, Copenhagen, July 2002.
4. F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *Journal of computer security*, 9:217–250, 2001.
5. J. Clemens and M. Felleisen. A tail-recursive semantics for stack inspections. In P. Degano, editor, *Proc. ESOP'03*, volume 2618 of *LNCS*. Springer-Verlag, 2003.
6. U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.
7. C. Fournet and A. D. Gordon. Stack inspection: theory and variants. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 307–318. ACM Press, 2002.
8. L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley, 1999.
9. D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM TOPLAS*, 23(6), Nov. 2001.
10. Microsoft Corp. *.NET Framework Developer's Guide: Securing Applications*.
11. F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.
12. F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. In D. Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *LNCS*, pages 30–45. Springer-Verlag, Apr. 2001.
13. F. B. Schneider. Enforceable security policies. Technical Report TR98-1664, Cornell University, Jan. 1998.
14. A. Souter and L. Pollack. Incremental call graph reanalysis for object-oriented software maintenance. In *IEEE International Conference on Software Maintenance*, Nov. 2001.
15. V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000)*, volume 35(10) of *SIGPLAN Notices*, pages 264–280, Minneapolis, MN, Oct. 2000. ACM Press.
16. F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '00)*, pages 281–293, 2000.
17. D. Walker. A type system for expressive security policies. In *Conference record of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 254–267. ACM Press, 2000.
18. D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM TOSEM*, 9(4):341–378, Oct. 2001.
19. C. Wille. *Presenting* C♯. SAMS Publishing, 2000.