Secure Calling Contexts for Stack Inspection *

Frédéric Besson
fbesson@irisa.fr

Thomas de Grenier de Latour
degrenie@irisa.fr

Thomas Jensen
jensen@irisa.fr

IRISA
Campus de Beaulieu
F-35042 Rennes Cedex
France

ABSTRACT

Stack inspection is a mechanism for programming secure applications by which a method can obtain information from the call stack about the code that (directly or indirectly) invoked it. This mechanism plays a fundamental role in the security architecture of Java and the .NET Common Language Runtime. A central problem with stack inspection is to determine to what extent the *local* checks inserted into the code are sufficient to guarantee that a *global* security property is enforced. In this paper, we present a technique for inferring a *secure calling context* for a method. By a secure calling context we mean a pre-condition on the call stack sufficient for guaranteeing that execution of the method will not violate a given global property. This is particularly useful for annotating library code in order to avoid having to reanalyse libraries for every new application. The technique is a constraint-based static program analysis implemented via fixed point iteration over an abstract domain of linear temporal logic properties.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security, Verification

Keywords

Language Based Security, Stack Inspection, Linear Temporal Logic, Static Program Analysis

*This work was partially supported by the IST FET/Open project 99-29075 "Secsafe".

PPDP'02, October 6-8, 2002, Pittsburgh, Pennsylvania, USA.

1. INTRODUCTION

Stack inspection has been proposed as a mechanism for programming access control in secure applications in which code components from different protection domains have to co-operate. It enables a component to obtain information about the code that (directly or indirectly) invokes its methods by letting it inspect the call stack of the run-time environment. Based on this information, the component can decide whether or not the callers have the right to access a given resource. Stack inspection plays a fundamental role in the security architecture of Java [12] as well as in the CLR [15].

To get an intuitive understanding of stack inspection we sketch how it is used in Java. Assume that code is given a set of permissions (based on its origin, who signed it, *etc.*), indicating whether the code has been allowed *e.g.*, to write to and read from files, to access peripherals, or to initiate communications with other hosts. The static method `checkPermission`, when called with a particular permission as argument, will inspect the call stack from top to bottom and check that every method on the stack has that permission. If the check fails, a security exception is raised. The only way a component without permission can use such protected resources is by invoking methods that have been marked as *privileged*. Marking a method as privileged means that stack inspection will stop when it is encountered in the call stack, essentially bestowing all its permissions to whoever called it.

As with other kinds of run-time checks (*e.g.*, dynamic typing), a central problem with stack inspection is the following:

> Are the *local* checks inserted into the code sufficient to guarantee that a *global* security property is enforced?

From a certification point of view, it is desirable to develop a program logic with sound semantic foundations that allows to prove such properties formally. Furthermore, stack inspection incurs a performance penalty, so the number of inserted checks should be kept low in order not to slow down execution drastically. Such a logic would also be useful for eliminating such redundant checks but we do not pursue this issue further here (see [21, 19]).

To address the above-stated problem, verification mechanisms based on static program analysis and model checking have been proposed [4, 14]. These verification techniques are whole-application analyses that require the program as well as the libraries to be available for analysis. Having

to re-analyse library functions means that even small programs take long to analyse. It is desirable to render the analysis more modular by developing an analysis that for each method calculates a *secure calling context* that characterizes those call stacks for which we are certain that the global security property is not violated if the method is invoked with one of theses stacks as current call stack. The contributions of this paper can be summarised as follows:

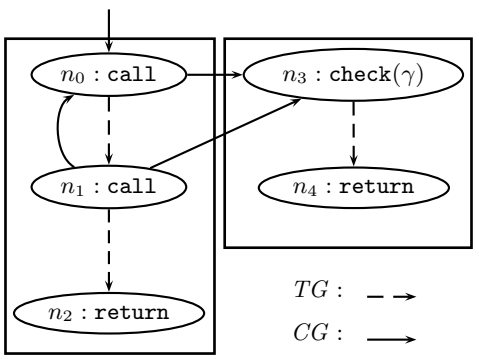
- we provide a semantic definition of *secure calling contexts* based on an operational semantics of control-flow graphs with security checks,
- we derive a constraint-based analysis that characterises the secure calling context of a method described by a control flow graph,
- we show how secure calling contexts can be calculated effectively by symbolic fixed point iteration over a lattice built from linear temporal logic formulae.

The rest of the paper is organised as follows. Our program notion will be a standard control-flow graph extended with *check* nodes indicating those program points where stack inspection is done. Section 2 formalises the notion of such extended control flow graphs (CFGs) and define their operational semantics. Section 3 defines the specification language (a version of linear temporal logic) in which the security properties are expressed. Section 4 introduces an inference system that given a global security property will infer a collection of set constraints whose solution is a valid set of secure calling contexts for the nodes of the CFG. These set constraints are not immediately solvable, so we reinterpret them as constraints over a lattice of temporal logic formulae (Section 6) and show how an iterative fixed point algorithm can be used to solve these constraints (Section 7). Section 8 compares with related work and Section 9 concludes and outlines further work.

2. PROGRAM MODEL

In this section we review the notion of *control-flow graph* (CFG) from [14] that will serve as an abstract program model. This model is not tied to one particular programming language. It abstracts away all data flow and focuses on security checks and control flow *i.e.*, which procedures (or methods, or functions) are called during execution and in what order. Nodes in a CFG correspond to program points and edges model the flow of control. There are three types of nodes: `call`, `return` and `check`(γ). Call nodes represent method calls in the program and return nodes signal the end of a method. A check node `check`(γ) represents stack inspection with respect to property γ: execution will proceed only if the current machine state satisfies γ. In the model we have two type of edges in order to distinguish between two types of control flow. Sequential composition of code is represented by a transfer edge (labelled with TG) between nodes. Method calls are modelled by call edges (labelled with CG) that bind call sites to their potential entry points.

Our model is also equipped with a labelling function $Attr$ that maps nodes to sets of uninterpreted attributes ranged over $attr$. This provides a simple way to formalize security policies that assign each piece of code a protection domain specifying its rights.



$$
\begin{array}{lcl}
Attr: & & \\
n_0, n_1 & \mapsto & \{System\} \\
n_2 & \mapsto & \{System, Crit\} \\
n_3, n_4 & \mapsto & \{Manager\} \\
\gamma & = & \mathbf{F}(Accountant) \wedge \mathbf{F}(Manager)
\end{array}
$$

Figure 1: A control flow graph

DEFINITION 2.1. *A* control-flow graph (CFG) *is a 6-tuple*

$$G = (NO, IS, EN, TG, CG, Attr)$$

where $NO \subseteq Nodes$ is the set of nodes, $EN \subseteq NO$ is the set of nodes designated as entry points, and IS maps a node to its type. Formally,

$$
\begin{array}{lcl}
IS & : & NO \to \{\texttt{call}, \texttt{return}, \texttt{check}(\gamma)\} \\
EN & : & \mathcal{P}(NO) \\
TG & : & NO \to \mathcal{P}(NO) \\
CG & : & NO \to \mathcal{P}(NO) \\
Attr & : & NO \to \mathcal{P}(attr)
\end{array}
$$

Control-flow graphs are subject to various well-formedness constraints, such as all checks and calls must be sequentially followed by another node, no code can follow sequentially after a return node, all calls must have at least one outgoing call edge and the origin of such call edges must be a call node.

EXAMPLE 2.2. *We will use the CFG in Figure 1 as our running example. The unique entry node n_0 is indicated by an arrow. Furthermore, the* `check` *node is labelled by a property*

$$\gamma = \mathbf{F}(Accountant) \wedge \mathbf{F}(Manager)$$

whose precise meaning will be explained in Section 3. Informally, system code (nodes n_0,n_1,n_2) intends to execute a critical operation in node n_2. The global security property to be enforced requires that this operation should only be executed if two actors Manager and Accountant have given their consent. To enforce this property, the `check`(γ) *node performs a dynamic stack inspection. This inspection checks that there will be a node with the Accountant attribute and a node with the Manager attribute in the call stack when control reaches the* `check`(γ) *node.*

2.1 Construction of a CFG

Representing a program by its control flow graph is a standard technique. It is not the main issue of this paper so we only briefly review the various analyses available for this

purpose. To obtain the CFG corresponding to an object-oriented program, its code is transformed into basic blocks and everything but methods calls is abstracted away. The construction of the call edges corresponding to a call `X.foo()` in the program is based on a static analysis that calculates an over-approximation of the classes of the objects that are being stored in variable `X`. Precision of the graph depends on this approximation [13]. The simplest approximations are limited to syntactic scans of the class hierarchy to find classes defining a method called `foo`—possibly improved by taking into account what classes are actually instantiated in the program [1]. A constraint based analysis, as proposed by Palsberg and Schwartzbach [17] takes data flow into account. In its basic formulation, this analysis ignores the sequential control flow of the program since it only calculates one global approximation for each variable. Its precision can be further improved by distinguishing between different *occurrences* of a variable, rendering the analysis flow-sensitive as proposed by Pande and Ryder [18].

2.2 Semantics of a CFG

In previous works [4], the operational semantics of a CFG was defined by a transition relation showing how the call stack evolves at each step in the execution of the program. The semantics is parameterised on the satisfaction relation $\models$ of the logic in which the check properties are expressed. For this paper, the logic in question will be linear temporal logic.

With $Stacks = Nodes^*$ the set of finite sequences of nodes from $Nodes$, the transition relation $\triangleright \subseteq Stacks \times Stacks$ is $\triangleright = \triangleright_{check} \cup \triangleright_{call} \cup \triangleright_{return}$ defined by the following rules where $s \in Stacks$ and n, n', $m \in NO$.

$$\triangleright_{check} \frac{IS(n) = \texttt{check}(\gamma) \quad n \overset{TG}{\rightarrow} n' \quad s{:}n \models \gamma}{s{:}n \triangleright_{check} s{:}n'} \qquad \triangleright_{call} \frac{IS(n) = \texttt{call} \quad n \overset{CG}{\rightarrow} m}{s{:}n \triangleright_{call} s{:}n{:}m}$$

$$\triangleright_{return} \frac{IS(m) = \texttt{return} \quad n \overset{TG}{\rightarrow} n'}{s{:}n{:}m \triangleright_{return} s{:}n'}$$

For this paper, we need a semantics that characterises the transitions made within a particular calling context. In order to achieve this, we generalise the operational semantics $\triangleright$ into a *collecting* semantics [6, 16] that collects the set of states reachable within a given number of transitions. More precisely, for a given starting stack $s{:}n$, our semantics collects the set $<s{:}n>^{1..i}$ of stacks reachable from $s{:}n$ in *at least* one and *at most* i transitions and before execution exits via a `return` node r of the method in which n is found. In other words, collection continues until execution reaches a stack of form $s{:}r$ with r a `return` node. All these stacks will be prefixed by s and will be longer in case of nested method calls.

DEFINITION 2.3. *With $s \in Stacks$ and $i \in \mathbb{N}$, we define the family of sets $<s>^{1..i}$ to be the smallest sets satisfying the following system of constraints:*

$$cs_{check} \frac{IS(n) = \texttt{check}(\gamma) \quad s{:}n \models \gamma \quad n \overset{TG}{\rightarrow} n'}{<s{:}n>^{1..i} \supseteq \{s{:}n'\} \cup <s{:}n'>^{1..i-1}}$$

$$cs_{call} \frac{IS(n) = \texttt{call} \quad n \overset{CG}{\rightarrow} m}{<s{:}n>^{1..i} \supseteq \{s{:}n{:}m\} \cup <s{:}n{:}m>^{1..i-1}}$$

$$cs_{return} \frac{\begin{array}{l} IS(n) = \texttt{call} \\ n \overset{CG}{\rightarrow} m \\ s{:}n{:}r \in \{s{:}n{:}m\} \cup <s{:}n{:}m>^{1..j} \\ IS(r) = \texttt{return} \\ n \overset{TG}{\rightarrow} n' \end{array}}{<s{:}n>^{1..i} \supseteq \{s{:}n'\} \cup <s{:}n'>^{1..i-j-1}}$$

where $n, n', m, r \in NO$ and $s \in Stacks$. Furthermore, we define:

$$\begin{array}{lcl} <s{:}n>^{0..i} & \equiv & \{s{:}n\} \cup <s{:}n>^{1..i} \\ <s{:}n>^{+} & \equiv & \bigcup_{i=1}^{\infty} <s{:}n>^{1..i} \\ <s{:}n>^{*} & \equiv & \bigcup_{i=0}^{\infty} <s{:}n>^{0..i} \end{array}$$

The rule cs_{check} expresses that if n' follows sequentially after a check n then everything that can be reached from n' in $i-1$ steps can be reached from n in i steps, provided that the initial state satisfies the check. The rule cs_{call} can be understood similarly. The rule cs_{return} can be understood as follows. If a return node of a method m can be reached in j steps and if a state s is reachable in $i-j-1$ steps from the node sequentially following a call to m, then s is reachable in i steps from a call node to m.

3. SECURITY PROPERTIES

Properties are specified in a linear temporal logic [9], $\mathcal{LTL}$, that will be interpreted over the set *Stacks* of finite sequences of nodes that corresponds to call stacks. This logic is expressive enough to express both:

- *Check properties* that specify local verifications (the properties in the `check` nodes of the CFGs).
- *Security properties* that specify global invariants of the execution of the program.

$\mathcal{LTL}$ formulae are inductively defined over a set of attributes *attr*. In addition to the propositional logic operators ($\vee$, $\neg$) we introduce the temporal operators *Strong Next* ($\mathbf{X}_\exists$) and *Strong Until* ($\mathbf{U}_\exists$). The set of properties is defined by:

$$\phi ::= \mathbf{True} \mid p \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{X}_\exists \phi \mid \phi \mathbf{U}_\exists \phi \qquad (p \in attr)$$

From this core syntax, usual propositional syntactic sugar ($\mathbf{False}$,$\wedge$,$\Rightarrow$) can be defined, together with the weak variants of the temporal operator ($\mathbf{X}_\forall$ and $\mathbf{U}_\forall$), some universal and existential modalities ($\mathbf{G}$ and $\mathbf{F}$) and an emptiness property (ε):

Weak Next :	$\mathbf{X}_\forall \phi \equiv \neg \mathbf{X}_\exists \neg \phi$
Eventually :	$\mathbf{F}\phi \equiv \mathbf{True}\mathbf{U}_\exists \phi$
Globally :	$\mathbf{G}\phi \equiv \neg \mathbf{F} \neg \phi$
Weak Until :	$\phi_1 \mathbf{U}_\forall \phi_2 \equiv \phi_1 \mathbf{U}_\exists \phi_2 \vee \mathbf{G}\phi_2$
Empty :	$\varepsilon \equiv \neg(\mathbf{True}\mathbf{U}_\exists \mathbf{True})$

The semantics of $\mathcal{LTL}$ is expressed by the satisfaction relation $\models$: $s \models \phi$ stands for "the call stack s is a model of ϕ". Formulae are interpreted from the top of the stack *i.e.*, the

first element taken into account by a formula evaluation is the node that was last pushed on the stack. When writing stacks as sequences we adopt the convention that *stacks grow from left to right*. Hence, in the stack $s = s_n{:}\ldots{:}s_i{:}\ldots{:}s_0$, the node s_n is the initial calling node, s_0 the current program point and s_i is its $(i+1)st$ element from the top. Furthermore, we introduce the notation $s^i = s_n{:}\ldots{:}s_i$ to denote the stack from which the i top elements have been removed, and $|s| = n+1$ to denote its length. With the labelling function *Attr* that gives the attributes of each node n, the semantics of the core $\mathcal{LTL}$ operators is defined as follows:

$$
\begin{array}{lll}
s \vDash \mathbf{True} & & \\
s \vDash p & \text{iff} & |s| > 0 \text{ and } p \in Attr(s_0) \\
s \vDash \neg\phi & \text{iff} & \text{not } (s \vDash \phi) \\
s \vDash \phi_1 \vee \phi_2 & \text{iff} & s \vDash \phi_1 \text{ or } s \vDash \phi_2 \\
s \vDash \mathbf{X}_\exists \phi & \text{iff} & |s| > 1 \text{ and } s^1 \vDash \phi \\
s \vDash \phi_1 \mathbf{U}_\exists \phi_2 & \text{iff} & \exists k, 0 \leq k < |s|.\ s^k \vDash \phi_2 \\
 & & \quad \text{and } \forall i, 0 \leq i < k.\ s^i \vDash \phi_1
\end{array}
$$

Informally, a stack always models **True** and models an attribute p if and only if p is part of the attributes of the top element of the stack. Operators $\neg$ and $\vee$ have their usual meanings. A stack models $\mathbf{X}_\exists\phi$ if the stack deprived from its top is non-empty and verifies ϕ. Finally, a $\phi_1\mathbf{U}_\exists\phi_2$ formula is verified by stacks such that their exists a sub-stack modelling ϕ_2 and all the previous sub-stacks models ϕ_1. We can also informally explain semantics of the syntactic sugar: $\mathbf{F}$, $\mathbf{G}$ and $\mathbf{U}_\forall$ have their usual meanings, *i.e.* $\mathbf{F}\phi$ stands for "ϕ is verified at least one time", $\mathbf{G}\phi$ for "ϕ is always verified", and $\phi_1\mathbf{U}_\forall\phi_2$ for "ϕ_1 is either always verified or verified until ϕ_2 is". The *Weak Next* is a *Next* variant which is always verified by stacks of one or zero element, and ε is only verified by the empty stack.

Finally, we introduce the *concretisation* function

$$
\begin{array}{l}
concr : \mathcal{LTL} \rightarrow \mathcal{P}(Stacks) \\
concr(\phi) = \{s \mid s \vDash \phi\}
\end{array}
$$

that to an $\mathcal{LTL}$ formula assigns the set of stacks that satisfies that formula.

3.1 Examples of properties

Check properties are expressed as $\mathcal{LTL}$ terms. As shown by the $\triangleright_{check}$ rule, the execution stops if the property does not hold for the current call stack. This framework can be instantiated to the Java stack inspection mechanism by only allowing check nodes to be labelled by an instance of the *JDK* formula defined by

$$JDK(perm) = perm\, \mathbf{U}_\forall\, (perm \wedge Priv)$$

which is a direct $\mathcal{LTL}$ formalisation of the property "all nodes must have the permission *perm* until a privileged node with the *perm* permission is encountered" (see [4] for a detailed discussion).

A security property is an invariant over call stacks. We say that a program is secure, with respect to a given security property φ, if and only if all the reachable call stacks, starting execution at an entry node n_0, do model φ. As an example, we might want to verify that critical program points, *i.e.* nodes with the *Crit* attribute, can only be reached from code with a given permission P. This is expressed by the formula

$$Crit \Rightarrow \mathbf{G}(P).$$

For optimisation purposes, we might want to eliminate redundant stack inspections (*i.e.*, stack inspections that always succeed). To prove that a check node n labelled by a property ϕ can be suppressed, the global invariant to verify is

$$N \Rightarrow \phi$$

where N is an attribute that identifies n.

EXAMPLE 3.1. *We return to our running example (Figure 1) and formally state a security property. The code is deemed secured if the critical action Crit in node n_2 can only be executed with the agreement of both Manager and Accountant code. Formally, we require the security property:*

$$Crit \Rightarrow \mathbf{F}(Manager) \wedge \mathbf{F}(Accountant)$$

If the top element of the call stack is a critical node then there exists in the stack nodes with the attributes Manager and Accountant. In order to enforce this property, node n_3 performs a dynamic check:

$$\mathbf{F}(Manager) \wedge \mathbf{F}(Accountant)$$

There are calling contexts for which the code from Figure 1 is not secure with respect to the global property defined in Example 3.1. To exhibit a security violation, it suffices to consider an execution trace starting with a call stack $n{:}n_0$ where n has the *Accountant* attribute.

$$n{:}n_0 \triangleright n{:}n_0{:}n_3 \triangleright n{:}n_0{:}n_4 \triangleright n{:}n_1 \triangleright n{:}n_1{:}n_3 \triangleright n{:}n_1{:}n_4 \triangleright n{:}n_2$$

This execution passes the dynamic `check` in node n_3 twice (successfully) and finally reaches the critical node n_2 with the call stack $n{:}n_2$. This stack does not model the property: none of the nodes has the *Manager* attribute. On the other hand, the code is obviously secure for all calling contexts for which a node has both *Accountant* and *Manager* attributes. An obvious question is whether this requirement is stronger than needed. Our forthcoming analysis will allow to answer this in the affirmative (Example 7.9) because it is specifically designed with the aim of inferring the most liberal pre-condition that prevents security violations.

4. SECURE CALLING CONTEXTS

In this and the following section we develop a constraint system that for each node in a given CFG specifies a *secure calling context*, relative to a global security property. A secure calling context for a node n is a set S of call stacks satisfying that for all stacks $s \in S$, an execution starting from $s{:}n$ in the sub-graph rooted by n will respect the global security property. The notion of secure calling context for a node n relative to a global security property φ is formalized by the predicate $sec : NO \times \mathcal{P}(Stacks) \rightarrow Bool$.

$$sec(n, S) \equiv \forall s \in S.\ <s{:}n>^* \subseteq concr(\varphi)$$

where $concr(\varphi)$ denotes the set of stacks satisfying φ (*cf.* Section 3).

The stack inspection mechanism will stop execution if the stack does not satisfy the property labelling a `check` node. It is essential that the analysis reflects this effect of stack inspection properly, otherwise little code will be deemed secure. To this end, we introduce two auxilliary properties

of nodes: *trans* and *returns*. The *trans* predicate characterises those calling contexts in which execution of node n may *transit* to the nodes following sequentially in the CFG. Thus, for a check node, *trans* is true of a set if it contains a stack that will pass stack inspection. Formally, $\mathit{trans} : \mathit{NO} \times \mathcal{P}(\mathit{Stacks}) \rightarrow \mathit{Bool}$ is defined by:

$$\begin{array}{l} \mathit{trans}(n, S) \equiv \\ \quad \forall s \in \mathit{Stacks}.\ (\exists n' \in \mathit{NO}.\ s{:}n' \in \langle s{:}n\rangle^{+}) \Rightarrow s \in S \end{array}$$

Similarly, the *returns* predicate characterises those calling contexts in which a method call may *return* (because there is an execution in which all stack inspections succeed). This predicate serves to propagate the effect of stack inspection from called methods to the caller. Informally, it states that if a return node r is reachable from a given node n with a given stack then that stack must belong to the calling context S. Formally, $\mathit{returns} : \mathit{NO} \times \mathcal{P}(\mathit{Stacks}) \rightarrow \mathit{Bool}$ is given by:

$$\begin{array}{l} \mathit{returns}(n, S) \equiv \\ \quad \forall s \in \mathit{Stacks}. \left(\begin{array}{c} \exists r \in \mathit{NO}.\ \mathit{IS}(r) = \mathtt{return} \\ \text{and} \\ s{:}r \in \langle s{:}n\rangle^{*} \end{array} \right) \Rightarrow s \in S \end{array}$$

We observe that it is always possible to remove elements from a secure calling context—the result will still be a secure calling context. Indeed, the empty set is a safe calling context albeit not a very interesting one. This is not the case for the sets satisfying *trans* and *returns*. These predicates will continue to hold if we replace a context with a larger context. Thus, in general, we are interested in finding the greatest set satisfying *sec* and the smallest sets satisfying *trans* and *returns*. This is reflected in the following, where we propose a method for finding such sets based on calculating least fixed points over lattices. These lattices will have the same carrier set, $\mathcal{P}(\mathit{Stacks})$, but will be ordered by subset inclusion or by reverse subset inclusion depending on whether we are looking for the greatest or the smallest set.

5. CONSTRAINTS FOR SECURE CALLING CONTEXTS

Having formalised the notion of secure calling context, the goal is now to show how to infer such contexts for a given CFG. We do this in two steps.

1. We show how to derive a system of set contraints $[\![G]\!]^{C}$ from a CFG G and prove that any solution to these constraints will provide a secure calling context for each node in G.
2. The system $[\![G]\!]^{C}$ is formulated using an extensional representation of sets in order to ease the proof of correctness. We derive an abstract version $[\![G]\!]^{\#}$ of the constraints that can be solved over an abstract domain of approximate, intensional representations of sets, which in our case will be formulae of $\mathcal{LTL}$.

5.1 Syntax-directed constraint generation

For each node n in a given CFG G we introduce a triple of set variables, $(\rho_n, \sigma_n, \tau_n) \in \mathcal{P}(\mathit{Stacks})^3$ with the intention that for any valid solution, ρ_n will satisfy the *returns* predicate, σ_n will satisfy *sec* and τ_n will satisfy *trans*.

In addition to the standard set-theoretic operators $\cup$,$\cap$ and $\setminus$ (complement), the constraints are constructed using an operator δ_n, whose effect on a set of stacks is to select those that have n as top element and remove this top element from the stacks. Formally we have a family of operators, one for each node:

DEFINITION 5.1. *Let $n \in \mathit{NO}$ be a node. Define $\delta_n : \mathcal{P}(\mathit{Stacks}) \rightarrow \mathcal{P}(\mathit{Stacks})$ by*

$$\delta_n(S) = \{s \mid s{:}n \in S\}$$

The system of set constraints $[\![G]\!]^{C}$ generated for a CFG G is defined inductively by the set of rules given in Figure 2. We here provide an informal justification of a selection of

$$\tau_{check}\ \frac{\mathit{IS}(n) = \mathtt{check}(\gamma)}{\tau_n \supseteq \delta_n(\mathit{concr}(\gamma))} \qquad \tau_{call}\ \frac{n \overset{CG}{\rightarrow} m}{\tau_n \supseteq \delta_n(\rho_m)}$$

$$\tau_{return}\ \frac{\mathit{IS}(n) = \mathtt{return}}{\tau_n \supseteq \emptyset}$$

$$\rho_{transfer}\ \frac{n \overset{TG}{\rightarrow} n'}{\rho_n \supseteq (\tau_n \cap \rho_{n'})} \qquad \rho_{return}\ \frac{\mathit{IS}(n) = \mathtt{return}}{\rho_n \supseteq \mathit{Stacks}}$$

$$\sigma_{global}\ \frac{}{\sigma_n \subseteq \delta_n(\mathit{concr}(\varphi))} \qquad \sigma_{call}\ \frac{n \overset{CG}{\rightarrow} m}{\sigma_n \subseteq \delta_n(\sigma_m)}$$

$$\sigma_{transfer}\ \frac{n \overset{TG}{\rightarrow} n'}{\sigma_n \subseteq (\mathit{Stacks} \setminus \tau_n) \cup \sigma_{n'}}$$

Figure 2: Syntax-directed definition of the set constraints $[\![G]\!]^{C}$.

rules. First, notice that a variable can be constrained by (*i.e.*, appear to the left) in several constraints. It is thus the joint effect of these constraints that determines the value of the variable. The rule $\rho_{transfer}$ expresses that we can reach a return node from node n if control can transfer to a successor node n' from which we can reach a return node. The rule σ_{call} deals with the case where a node n is a call to a method starting with node m. It uses the δ operator to express that executions starting with call stack s at node n are secure only if the executions emanating from node m with stack $s{:}n$ are also secure. Finally, the rule $\sigma_{transfer}$ formalises that when control can transfer sequentially from n to n', an execution starting from n with stack s is secure only if *either* the execution -7.54(t)(x)-a.9(n')(o)wtihg from-7.54(a)7.8(n)2.4(

$_n$).

EXAMPLE 5.2. *Continuing our running example, the nodes n_3 and n_4 fom figure 1 gi ri to the flox set constraints.*

$$\begin{array}{l} \tau_{n_3} \supseteq \delta_{n_3}(\mathit{concr}(\mathbf{F}(\mathit{Accountant}) \wedge \mathbf{F}(\mathit{Manager}))) \\ \tau_{n_4} \supseteq \emptyset \\ \rho_{n_3} \supseteq \tau_{n_3} \cap \rho_{n_4} \\ \rho_{n_4} \supseteq \mathit{Stacks} \\ \sigma_{n_3} \subseteq \delta_{n_3}(\mathit{concr}(\mathit{Crit} \Rightarrow \mathbf{F}(\mathit{Accountant}) \wedge \mathbf{F}(\mathit{Manager}))) \\ \sigma_{n_3} \subseteq (\mathit{Stacks} \setminus \tau_{n_3}) \cup \sigma_{n_4} \\ \sigma_{n_4} \subseteq \delta_{n_4}(\mathit{concr}(\mathit{Crit} \Rightarrow \mathbf{F}(\mathit{Accountant}) \wedge \mathbf{F}(\mathit{Manager}))) \end{array}$$

5.2 Existence of a solution to $[\![G]\!]^C$

A *solution* to a system of constraints $[\![G]\!]^C$ is a triple

$$(\rho, \sigma, \tau) \in (NO \to \mathcal{P}(Stacks))^3$$

of functions. In the following we will use σ_n to denote both a variable and the value of that variable in a solution (ρ, σ, τ).

The existence of a solution to a constraint system $[\![G]\!]^C$ is argued in the standard way [8] by interpreting the constraints as monotone operators over lattices of subsets and then using the Knaster-Tarski fixed point theorem to assert the existence of a least fixed point and hence a solution to the constraints. We first make the following observation:

OBSERVATION 5.3. *The operator δ_n is monotone over the lattice $(\mathcal{P}(Stacks), \subseteq)$ and also over its dual $(\mathcal{P}(Stacks), \supseteq)$.*

DEFINITION 5.4. *The lattice $(\mathcal{RST}, \sqsubseteq_{\mathcal{RST}})$ of solutions is defined by*

$$\mathcal{RST} = (NO \to \mathcal{P}(Stacks))^3,$$

$$(\rho^1, \sigma^1, \tau^1) \sqsubseteq_{\mathcal{RST}} (\rho^2, \sigma^2, \tau^2) \text{ iff } \forall n \in NO. \bigwedge \left\{ \begin{array}{l} \rho_n^1 \subseteq \rho_n^2 \\ \sigma_n^1 \supseteq \sigma_n^2 \\ \tau_n^1 \subseteq \tau_n^2 \end{array} \right.$$

LEMMA 5.5. *Let $c \in [\![G]\!]^C$ be a constraint whose right-hand side is an expression e in the variables ρ, σ, τ. Then, e, considered as an operator $e : \mathcal{RST} \to \mathcal{P}(Stacks)$, is monotone.*

PROOF. Most right-hand sides are either constants or uses operators like δ_n which are monotone. The only non-trivial case are the constraints of the form $\sigma_{n'} \supseteq (Stacks \setminus \tau_n) \cup \sigma_{n'}$ generated by the rule $\sigma_{transfer}$. Assume $(\rho^1, \sigma^1, \tau^1) \sqsubseteq_{\mathcal{RST}} (\rho^2, \sigma^2, \tau^2)$. Then, $\sigma_n^1 \supseteq \sigma_n^2$ and $\tau_n^1 \subseteq \tau_n^2$, so

$$(Stacks \setminus \tau_n^1) \cup \sigma_{n'}^1 \supseteq (Stacks \setminus \tau_n^2) \cup \sigma_{n'}^2$$

which implies monotonicity since the σ_n are ordered by $\supseteq$. □

It follows that the system $[\![G]\!]^C$ has a least solution with respect to the ordering $\sqsubseteq_{\mathcal{RST}}$. As discussed in Section 4, the least solution is also the most informative in that it will be the largest among all the secure calling context. In Section 6 we use abstract interpretation to derive an abstract system of constraints whose solutions are safe, computable approximations of the least solution to $[\![G]\!]^C$.

5.3 Correctness of $[\![G]\!]^C$

The remainder of this section is devoted to prove the correctness of the constraint system in Figure 2.

THEOREM 5.6. *Let G be a CFG and let (ρ, σ, τ) be a solution to $[\![G]\!]^C$. Then, for all nodes $n \in NO$, σ_n is a secure calling context (*i.e.*, $sec(n, \sigma_n)$ holds).*

We prove the more general result that for all nodes $n \in NO$, we have $returns(n, \rho_n)$, $sec(n, \sigma_n)$ and $trans(n, \tau_n)$. The proofs are by induction over the computation length of the collecting semantics. To make the induction argument explicit, we express directly the *trans*, *returns* and *sec* predicates in terms of the collecting semantics. This transformation is a direct consequence of the property:

$$s \in <s'>^+ \text{ iff } \exists i \in \mathbb{N}.\ s \in <s'>^{1..i}$$

Using predicate logic identities, we obtain equivalent definitions of the *trans*, *returns* and *sec* predicates:

PROPERTY 5.7.

$sec(n, S)$	*iff*	$\forall i \in \mathbb{N}.\ \forall s \in Stacks.\ s \in S \Rightarrow \sigma^i(s, n)$
$trans(n, S)$	*iff*	$\forall i \in \mathbb{N}.\ \forall s \in Stacks.\ \tau^i(s, n) \Rightarrow\ s \in S$
$returns(n, S)$	*iff*	$\forall i \in \mathbb{N}.\ \forall s \in Stacks.\ \rho^i(s, n) \Rightarrow\ s \in S$

where σ^i, τ^i and $\rho^i : Stacks \times NO \to Bool$ are predicates defined as follows:

$$\begin{array}{rcl} \sigma^i(s,n) & = & <s{:}n>^{0..i} \subseteq concr(\varphi) \\ \tau^i(s,n) & = & \exists n' \in NO.\ s{:}n' \in <s{:}n>^{1..i} \\ \rho^i(s,n) & = & \exists r \in NO.\ s{:}r \in <s{:}n>^{0..i} \\ & & \quad \text{and } IS(r) = \texttt{return} \end{array}$$

Correctness proofs will be carried out with respect to these alternative definitions of the *trans*, *returns* and *sec* predicates. Since the ρ_n and τ_n are defined by mutual recursion but without using σ_n we first prove correctness for them. We then prove correctness for σ_n using that of τ_n.

5.3.1 Correctness of the τ_n, ρ_n sets

LEMMA 5.8. *For all integer i, node n and stack s, the following holds:*

$$\begin{array}{rcl} \rho^i(s,n) & \Rightarrow & s \in \rho_n \\ \tau^i(s,n) & \Rightarrow & s \in \tau_n \end{array}$$

PROOF. The proof is by induction over the collecting semantics computation step i.

Base case: $i = 0$

- $<s{:}n>^{1..0}$ is empty. As a result, $\tau^0(s, n)$ cannot be satisfied and the τ part of the lemma is vacuously true.
- $<s{:}n>^{0..0}$ is the singleton $\{s{:}n\}$. Hence, $\rho^0(s, n) \Rightarrow IS(n) = \texttt{return}$. Now, by ρ_{return}, we have $s \in \rho_n$, therefore the ρ part of the lemma is verified.

Inductive step As an induction hypothesis, we assume that the lemma is verified up to a given i:

$$\forall n \in NO.\ \forall s \in Stacks.\ \forall j \leq i.\ \bigwedge \left\{ \begin{array}{rcl} \rho^j(s,n) & \Rightarrow & s \in \rho_n \\ \tau^j(s,n) & \Rightarrow & s \in \tau_n \end{array} \right.$$

To prove the property for the rank $i + 1$, we suppose $\tau^{i+1}(s, n)$ (resp. ρ^{i+1}) and prove $s \in \tau_n$ (resp. $s \in \rho_n$). The proof relies on the fact that $<s>^{i+1}$ is the smallest set satisfying the constraints. It follows that a stack s' belonging to $<s>^{i+1}$ can only be produced by one of the collecting semantics rules. We consider each case in turn.

- cs_{check}: $IS(n) = \texttt{check}(\gamma), s{:}n \vDash \gamma, n \overset{TG}{\to} n'$
  - τ: In this case, we directly prove that $s \in \tau_n$. Since by τ_{check}, $\delta_n(concr(\gamma)) \subseteq \tau_n$, it amounts to prove that $s \in \delta_n(concr(\gamma))$. By definition

of δ and *concr*, we have the following identities: $s \in \delta_n(concr(\gamma))$ iff $s{:}n \in concr(\gamma)$ iff $s{:}n \vDash \gamma$. Furthermore, we have $s{:}n \vDash \gamma$ by hypothesis. As a result, the τ part of the lemma is verified.

– ρ: We suppose a stack $s{:}r \in <s{:}n'>^{0..i}$ such that $IS(r) = \texttt{return}$ and prove that $s \in \rho_n$. By $\rho_{transfer}$, it amounts to prove that $s \in \tau_n \cap \rho_{n'}$. We have just proved that $s \in \tau_n$ under weaker hypotheses. It remains to prove that $s \in \rho_{n'}$. By hypothesis, we have $\rho^i(s, n')$ and by induction hypothesis, $\rho^i(s, n') \Rightarrow s \in \rho_{n'}$. It follows that $s \in \rho_n$, and the ρ part of the lemma is verified.

- cs_{call}: This case is vacuously true. Since the cs_{call} rule collects stacks from nested calls, any stack s' belonging to $<s{:}n{:}m>^{1..i}$ is strictly longer than $s{:}n$. It follows that we cannot pick any stack $s{:}r$ in this set.

- cs_{return}: $IS(n) = \texttt{call}, n \overset{CG}{\rightarrow} m, \rho^j(s{:}n, m), n \overset{TG}{\rightarrow} n'$

  – τ: We directly prove that $s \in \tau_n$. By induction hypothesis, $\rho^j(s{:}n, m) \Rightarrow s{:}n \in \rho_m$, and by definition of δ , $s \in \delta_n(\rho_m)$. Now, by τ_{call}, $\tau_n \supseteq \delta_n(\rho_m)$, therefore $s \in \tau_n$ and the τ part of the lemma is verified.

  – ρ: We suppose a stack $s{:}r \in <s{:}n'>^{0..i-j}$ such that $IS(r) = \texttt{return}$ and prove that $s \in \rho_n$. The arguments are similar to the cs_{check} case except that we use the induction hypothesis at rank $(i-j)$: $\rho^{i-j}(s, n') \Rightarrow s \in \rho_{n'}$. As a result, we have $s \in \rho_n$ and the ρ part of the lemma is verified.

□

5.3.2 Correctness of the σ_n set

LEMMA 5.9. *For all integer i, node n and stack s, the following holds:*

$$s \in \sigma_n \;\Rightarrow\; \sigma^i(s, n)$$

PROOF. The proof is by induction over i.
Base case: $i = 0$ We suppose that $s \in \sigma_n$ and prove $\sigma^0(s, n)$. Since $<s{:}n>^{0..0} = \{s{:}n\}$, it amounts to show that $\{s{:}n\} \subseteq concr(\varphi)$. Now, by σ_{global}, $\sigma_n \subseteq \delta_n(\varphi)$. It follows that $s{:}n \in concr(\varphi)$ *i.e.*, $\{s{:}n\} \subseteq concr(\varphi)$ and the lemma is verified.
Inductive step As an induction hypothesis, we assume that the lemma is verified up to a given i:

$$\forall n \in NO.\, \forall s \in Stacks.\, \forall j \leq i. \quad s \in \sigma_n \;\Rightarrow\; \sigma^j(s, n)$$

To prove the property for rank $i+1$, we assume that $s \in \sigma_n$ and prove $\sigma^{i+1}(n, s)$. The proof is case analysis over the collecting semantics rules. Another time, we rely on the fact that $<s>^{i+1}$ is the smallest set satisfying the constraints.

- cs_{check}: $IS(n) = \texttt{check}(\gamma), s{:}n \vDash concr(\gamma), n \overset{TG}{\rightarrow} n'$. We suppose $s \in \sigma_n$ and prove $<s{:}n'>^{0..i} \subseteq concr(\varphi)$. By $\sigma_{transfer}$, $\sigma_n \subseteq (Stacks \setminus \tau_n) \cup \sigma_{n'}$. From τ correctness, we have $trans(n, \tau_n)$ and $s{:}n' \in <s{:}n>^+$ that is $s \in \tau_n$. Hence, we deduce that $s \in \sigma_{n'}$. By induction hypothesis, we obtain that $\sigma^i(s, n')$ *i.e.*, $<s{:}n'>^{0..i} \subseteq concr(\varphi)$, and the lemma is verified.

- cs_{return}: $IS(n) = \texttt{call}, n \overset{CG}{\rightarrow} m, \rho^j(s{:}n, m), n \overset{TG}{\rightarrow} n'$. We suppose that $s \in \sigma_n$ and prove that $<s{:}n'>^{0..i-j} \subseteq concr(\varphi)$. The arguments are similar to the cs_{check} case except that we use the induction hypothesis at rank $i - j$: $s \in \sigma_{n'} \Rightarrow \sigma^{i-j}(s, n')$.

- cs_{call}: $IS(n) = \texttt{call}, n \overset{CG}{\rightarrow} m$. We suppose that $s \in \sigma_n$ and prove that $<s{:}n{:}m>^{0..i} \subseteq concr(\varphi)$. By σ_{call}, $\sigma_n \subseteq \delta_n(\sigma_m)$, and, by definition of δ, $s \in \sigma_n \Rightarrow s{:}n \in \sigma_m$. Now, by induction hypothesis, $s{:}n \in \sigma_m \Rightarrow \sigma^i(s{:}n, m)$, therefore $<s{:}n{:}m>^{0..i} \subseteq concr(\varphi)$, and the lemma is verified.

□

6. SYMBOLIC CALCULATION OF SECURE CALLING CONTEXTS

The set constraints obtained from a graph G are formulated using extensional definitions of sets that might be infinite. In order to obtain a system of constraints that is more amenable to symbolic manipulations, we replace the concrete, extensional sets of stacks by an abstract, intensional representation based on $\mathcal{LTL}$ formulae and derive a constraint system to be solved over an abstract domain of (equivalence classes) of such formulae.

The abstract constraints are built from usual propositional operators ($\neg$,$\vee$,$\wedge$) and an abstraction of the δ operator from the previous section (Definition 5.1). Intuitively, given a formula that denotes the calling contexts, $\delta^{\#}$ computes the weakest precondition such that the property is verified before a call. This definition is an adaptation of the transition relation given by Vardi in his automata-theoretic approach to $\mathcal{LTL}$ [22].

DEFINITION 6.1. *Let $n \in NO$ be a node. The abstract* weakest calling context operator $\delta_n^{\#} : \mathcal{LTL} \rightarrow \mathcal{LTL}$ *is inductively defined over the structure of the formula.*

$$\begin{array}{ll}
\delta_n^{\#}(p) & = p \in Attr(n) \\
\delta_n^{\#}(\neg\phi) & = \neg\delta_n^{\#}(\phi) \\
\delta_n^{\#}(\phi_1 \vee \phi_2) & = \delta_n^{\#}(\phi_1) \vee \delta_n^{\#}(\phi_2) \\
\delta_n^{\#}(\mathbf{X}_\exists\phi) & = \phi \wedge \neg\varepsilon \\
\delta_n^{\#}(\phi_1 \mathbf{U}_\exists \phi_2) & = \delta_n^{\#}(\phi_2) \vee (\delta_n^{\#}(\phi_1) \wedge \phi_1 \mathbf{U}_\exists \phi_2)
\end{array}$$

The following lemma states that the $\delta^{\#}$ operator calculates the most precise precondition for a property ϕ to hold at a given node n. In other words, there is no loss of precision when calculating with this abstract version of δ.

LEMMA 6.2. *For a given a triple $(s, n, \phi) \in Stack \times NO \times \mathcal{LTL}(P)$, $\delta^{\#}$ satisfies:*

$$s \vDash \delta_n^{\#}(\phi) \iff s{:}n \vDash \phi$$

PROOF. The proof is by induction over the structure of ϕ. Induction hypothesis states the correctness of the lemma on every sub-formulae. Here, we only give the case for the $\mathbf{U}_\exists$ operator.

We first prove that $s{:}n \vDash \phi_1 \mathbf{U}_\exists \phi_2$ implies $s \vDash \delta_n^{\#}(\phi_1 \mathbf{U}_\exists \phi_2)$. By $\mathcal{LTL}$ semantics, $s{:}n \vDash \phi_1 \mathbf{U}_\exists \phi_2 \Leftrightarrow \exists k, 0 \leq k < |s{:}n|.\ s{:}n^k \vDash \phi_2 \wedge \forall i, 0 \leq i < k.\ s{:}n^i \vDash \phi_1$.

- If k=0, then it implies $s{:}n{\models}\phi_2$, and, by induction hypothesis, $s{\models}\delta_n^{\#}(\phi_2)$. By $\delta^{\#}$ definition, this implies $s{\models}\delta_n^{\#}(\phi_1 \mathbf{U}_{\exists} \phi_2)$.
- If k>0, then it implies $s{:}n{\models}\phi_1$ and $s{:}n^k{\models}\phi_2$ and $\forall i$, $0{<}i{<}k$. $s{:}n^i{\models}\phi_1$. By induction hypothesis, we have $s{:}n{\models}\phi_1 \Leftrightarrow s{\models}\delta_n^{\#}(\phi_1)$. Now, by $s{:}n^j{\models}\phi \Leftrightarrow s^{j-1}{\models}\phi$ for j>0 and by $\mathbf{U}_{\exists}$ semantics, it also implies $s^{k-1}{\models}\phi_2$ and $\forall i, 0{\leq}i{<}k{-}1$. $s^i{\models}\phi_1$, *i.e.*, $s{\models}\phi_1 \mathbf{U}_{\exists}\phi_2$. Therefore, $s{:}n{\models}\phi_1 \mathbf{U}_{\exists}\phi_2$ implies $s{\models}(\delta_n^{\#}(\phi_1) \wedge \phi_1 \mathbf{U}_{\exists}\phi_2)$. By $\delta^{\#}$ definition, this also implies $s{\models}\delta_n^{\#}(\phi_1 \mathbf{U}_{\exists}\phi_2)$.

We then prove that $s{\models}\delta_n^{\#}(\phi_1 \mathbf{U}_{\exists}\phi_2)$ implies $s{:}n{\models}\phi_1 \mathbf{U}_{\exists}\phi_2$. By $\delta^{\#}$ definition, $s{\models}\delta_n^{\#}(\phi_1 \mathbf{U}_{\exists}\phi_2) \Leftrightarrow s{\models}\delta_n^{\#}(\phi_2) \vee (\delta_n^{\#}(\phi_1) \wedge \phi_1 \mathbf{U}_{\exists}\phi_2)$.

- If $s{\models}\delta_n^{\#}(\phi_2)$, then, by induction hypothesis, we have $s{:}n{\models}\phi_2$, which leads to $s{:}n{\models}\phi_1 \mathbf{U}_{\exists}\phi_2$.
- Else, if $s{\models}(\delta_n^{\#}(\phi_1) \wedge \phi_1 \mathbf{U}_{\exists}\phi_2)$, we have, by induction hypothesis, $s{:}n{\models}\phi_1$. We also have, by $\mathbf{U}_{\exists}$ semantics, $\exists k, 0{\leq}k{<}|s|$. $s^k{\models}\phi_2 \wedge \forall i, 0{\leq}i{<}k$. $s^i{\models}\phi_1$. Hence, by $s^j{\models}\phi \Leftrightarrow s{:}n^{j+1}{\models}\phi$, $\exists k, 0{<}k{<}|s{:}n|$. $s{:}n^k{\models}\phi_2 \wedge \forall i, 0{\leq}i{<}k$. $s{:}n^i{\models}\phi_1$. According to $\mathbf{U}_{\exists}$ semantics, this implies $s{:}n \models \phi_1 \mathbf{U}_{\exists}\phi_2$.

□

We now recast the set-based contraints from the analysis in Figure 2 as constraints over $\mathcal{LTL}$ formulae, by replacing the set-based operators $(\subseteq, \cup, \cap, \delta_n)$ with their abstract counterpart $(\Rightarrow, \vee, \wedge, \delta_n^{\#})$. The result is a similar syntax-directed constraint generation scheme, which is shown in Figure 3.

$$\tau_{check}\frac{IS(n) = \texttt{check}(\gamma)}{\tau_n \Leftarrow \delta_n^{\#}(\gamma)} \qquad \tau_{call}\frac{n \overset{CG}{\rightarrow} m}{\tau_n \Leftarrow \delta_n^{\#}(\rho_m)}$$

$$\tau_{return}\frac{IS(n) = \texttt{return}}{\tau_n \Leftarrow \mathbf{False}}$$

$$\rho_{transfer}\frac{n \overset{TG}{\rightarrow} n'}{\rho_n \Leftarrow (\tau_n \wedge \rho_{n'})} \qquad \rho_{return}\frac{IS(n) = \texttt{return}}{\rho_n \Leftarrow \mathbf{True}}$$

$$\sigma_{global}\frac{}{\sigma_n \Rightarrow \delta_n^{\#}(\varphi)} \qquad \sigma_{call}\frac{n \overset{CG}{\rightarrow} m}{\sigma_n \Rightarrow \delta_n^{\#}(\sigma_m)}$$

$$\sigma_{transfer}\frac{n \overset{TG}{\rightarrow} n'}{\sigma_n \Rightarrow (\neg\tau_n \vee \sigma_{n'})}$$

Figure 3: Constraint specification of τ,ρ,σ

Let $[\![G]\!]^{\#}$ denote the result of analysing graph G. It is clear that there is a one-to-one correspondence between the contraints in $[\![G]\!]^{C}$ and $[\![G]\!]^{\#}$. In addition, a *solution* to $[\![G]\!]^{\#}$ is defined in a way similar to that of solutions of $[\![G]\!]^{C}$ as a triple of maps from nodes to $\mathcal{LTL}$ formulae. The correctness of $[\![G]\!]^{\#}$ can then amounts to showing that the set of stacks satisfying the LTL formula calculated for $\sigma_n^{\#}$ is a secure calling context for n. This can be stated formally using the concretisation function *concr* that allows to pass from LTL formulae to the set of stacks that satisfy the formula.

PROPOSITION 6.3. *Let $(\rho^{\#}, \sigma^{\#}, \tau^{\#})$ be a solution to $[\![G]\!]^{\#}$ for a given CFG G. Then, for all nodes n in G,*

$$sec(n, concr(\sigma_n^{\#}))$$

PROOF. The proof consists in showing that if an abstract solution $(\rho^{\#}, \sigma^{\#}, \tau^{\#})$ satisfies a constraint in $[\![G]\!]^{\#}$, then its concretisation satisfies the corresponding constraint in $[\![G]\!]^{C}$. Correctness then follows by appealing to the correctness of $[\![G]\!]^{C}$. □

EXAMPLE 6.4. *After $\mathcal{LTL}$ abstraction, here is the set of constraints obtained from the CFG presented in figure 1. We use φ as abbreviation for the formula*

$$Crit \Rightarrow \mathbf{F}(Accountant) \wedge \mathbf{F}(Manager).$$

$$\begin{array}{l}
\tau_{n_0}^{\#} \Leftarrow \delta_{n_0}^{\#}(\rho_{n_3}^{\#}) \\
\tau_{n_1}^{\#} \Leftarrow \delta_{n_1}^{\#}(\rho_{n_0}^{\#}) \qquad \tau_{n_1}^{\#} \Leftarrow \delta_{n_1}^{\#}(\rho_{n_3}^{\#}) \\
\tau_{n_2}^{\#} \Leftarrow \mathbf{False} \\
\tau_{n_3}^{\#} \Leftarrow \delta_{n_3}^{\#}(\mathbf{F}(Manager) \wedge \mathbf{F}(Accountant)) \\
\tau_{n_4}^{\#} \Leftarrow \mathbf{False} \\
\rho_{n_0}^{\#} \Leftarrow \tau_{n_0}^{\#} \wedge \rho_{n_1}^{\#} \\
\rho_{n_1}^{\#} \Leftarrow \tau_{n_1}^{\#} \wedge \rho_{n_2}^{\#} \\
\rho_{n_2}^{\#} \Leftarrow \mathbf{True} \\
\rho_{n_3}^{\#} \Leftarrow \tau_{n_3}^{\#} \wedge \rho_{n_4}^{\#} \\
\rho_{n_4}^{\#} \Leftarrow \mathbf{True} \\
\sigma_{n_0}^{\#} \Rightarrow \delta_{n_0}^{\#}(\varphi) \quad \sigma_{n_0}^{\#} \Rightarrow \delta_{n_0}^{\#}(\sigma_{n_3}^{\#}) \quad \sigma_{n_0}^{\#} \Rightarrow (\neg\tau_{n_0}^{\#} \vee \sigma_{n_1}^{\#}) \\
\sigma_{n_1}^{\#} \Rightarrow \delta_{n_1}^{\#}(\varphi) \quad \sigma_{n_1}^{\#} \Rightarrow \delta_{n_1}^{\#}(\sigma_{n_0}^{\#}) \\
\sigma_{n_1}^{\#} \Rightarrow \delta_{n_1}^{\#}(\sigma_{n_3}^{\#}) \quad \sigma_{n_1}^{\#} \Rightarrow (\neg\tau_{n_1}^{\#} \vee \sigma_{n_2}^{\#}) \\
\sigma_{n_2}^{\#} \Rightarrow \delta_{n_2}^{\#}(\varphi) \\
\sigma_{n_3}^{\#} \Rightarrow \delta_{n_3}^{\#}(\varphi) \quad \sigma_{n_3}^{\#} \Rightarrow (\neg\tau_{n_3}^{\#} \vee \sigma_{n_4}^{\#}) \\
\sigma_{n_4}^{\#} \Rightarrow \delta_{n_4}^{\#}(\varphi)
\end{array}$$

7. ITERATIVE CONSTRAINT SOLVING

The set of constraints $[\![G]\!]^{\#}$ obtained by analysing the control flow graph G can be solved by an iterative fixed point calculation. However, our domain of $\mathcal{LTL}$ formulae does not *a priori* guarantee termination of standard iterative fixed point algorithms. The reasons is that logical implication of $\mathcal{LTL}$ formulae $\Rightarrow$ is a pre-order (syntactically distinct formulae may be equivalent semantically) whose quotient by $\mathcal{LTL}$ equivalence still contains infinitely increasing chains. In our algorithm, rather that explicitly computing over the quotient domain, we rely on $\mathcal{LTL}$ equivalence to check stabilisation of the iteration. This approach is feasible because equivalence of $\mathcal{LTL}$ formulae is decidable. The central part of this section is thus concerned with establishing that all iteration sequences arising during the verification of a particular property will be stationary after a finite number of iterations. The essential observation underlying this result is that the set obtained by iterating the $\delta_n^{\#}$ functions over

a finite set of formulae is again a finite set (modulo $\mathcal{LTL}$ equivalence).

Our resolution scheme rephrases constraint solving in terms of a least fixed point problem [8, 16]. For a complete lattice $(D, \sqsubseteq, \sqcup, \sqcap)$, a set of constraints C induces an *iterator* $F : (Var \to D) \to (Var \to D)$ obtained by by gathering the constraints defining the same variable into a single expression.

$$F(\rho)(x) = \bigsqcup \{e(\rho) \mid x \sqsupseteq e \in C\}$$

where $e(\rho)$ is the evaluation of the expression e in the environment ρ that maps variables to values of D. Monotonicity of the expressions e_i implies monotonicity of the iterator F which, by Tarski's theorem, ensures that it has a least fixed point and hence that a least solution to the original system exists. Furthermore, this fixed point can in certain cases be calculated by a chaotic fixed point iteration [7] since *if* the ascending Kleene chain $\bot, F(\bot), F^2(\bot), \ldots$ stabilises at a least upper bound then this least upper found is the least fixed point of F.

To apply this resolution technique to our analysis and prove termination, we show how to build, for a given property φ and a given program G, a domain of $\mathcal{LTL}$-formulae that is finite modulo $\mathcal{LTL}$ equivalence. Resolution will take place within this domain. This domain, quotiented by $\mathcal{LTL}$ equivalence, induces a finite lattice (thus complete) required by the theory. Let *Var* be a finite set of variables, *Const* the finite set of ground $\mathcal{LTL}$ formulae from the `check` nodes in G together with the global property φ and let *NO* be the nodes of G. The expressions derived from $[\![G]\!]^\#$ belong to the following inductively defined set:

$$E ::= x \mid c \mid \delta_n^\#(E) \mid E \wedge E \mid E \vee E \mid \neg E \mid \mathbf{True} \mid \mathbf{False}$$

where $x \in Var, c \in Const, n \in NO$.

As a result, the iterator derived from such a system of constraints consists of composition of $\delta^\#$ functions and propositional operators $(\wedge, \neg, \vee)$. In order to prove termination of these iterators, we show that it is possible to identify a finite sub-domain (modulo equivalence) of $\mathcal{LTL}$ formulae that contains *Const* and is closed under all $\delta^\#$-compositions and propositional operators. As a consequence, our fixed point iterations are bounded.

Definition 7.1. *For a formula ϕ, the finite set $Sub(\phi)$ is formally defined to be the smallest set of formulae satisfying:*

$$\begin{array}{lcl}
\phi \in Sub(\phi) & & \\
\phi_1 \; op \; \phi_2 \in Sub(\phi) & \Rightarrow & \{\phi_1, \phi_2\} \subseteq Sub(\phi) \\
 & & \quad where \; op \in \{\mathbf{U}_\exists, \vee, \wedge\} \\
\neg\phi' \in Sub(\phi) & \Rightarrow & \phi' \in Sub(\phi) \\
\mathbf{X}_\exists \phi' \in Sub(\phi) & \Rightarrow & \{\phi', \varepsilon\} \subseteq Sub(\phi)
\end{array}$$

Definition 7.2. *Let A be a finite (unordered) set. $Prop(A)$ is the set of propositional formulae built over A.*

The following lemma establishes an important closure property of domain $Prop(Sub(\phi))$, *viz.*, that applying $\delta^\#$ to a formula in $Prop(Sub(\phi))$ results in a property that still belongs to the set $Prop(Sub(\phi))$. This result will be used for arguing the termination of the fixed point iteration.

Lemma 7.3. *Given a pair (ϕ, ψ) of $\mathcal{LTL}$ formulae and a node n, we have*

$$\psi \in Prop(Sub(\phi)) \Rightarrow \delta_n^\#(\psi) \in Prop(Sub(\phi))$$

Proof. The proof is by structural induction over ψ.

Base case: If $\psi = p$ then $\delta_n^\#(\psi)$ is either **True** or **False** depending on whether p belongs (or not) to $Attr(n)$. Obviously, $\{\mathbf{True}, \mathbf{False}\}$ is a subset of $Prop(Sub(\phi))$.

Inductive case: We first consider the case of formulae whose top operator is temporal – either $\mathbf{X}_\exists$ or $\mathbf{U}_\exists$. We rely on the fact that such formulae can only belong to $Sub(\phi)$.

- $\psi = \mathbf{X}_\exists(\psi')$: $\delta_n^\#(\psi) = \psi' \wedge \neg\varepsilon$. Now, by hypothesis, $\{\psi', \varepsilon\} \subseteq Sub(\phi)$, therefore, by definition of *Prop*, $\psi' \wedge \neg\varepsilon \in Prop(Sub(\phi))$.
- If $\psi = \psi_1 \mathbf{U}_\exists \psi_2$ then $\delta_n^\#(\psi) = \delta_n^\#(\psi_2) \vee (\delta_n^\#(\psi_1) \wedge \psi)$. Since both ψ_1 and ψ_2 belongs to $Sub(\phi)$, by applying induction hypothesis, we have that $\{\delta_n^\#(\psi_2), \delta_n^\#(\psi_1)\} \in Prop(Sub(\phi))$. The property follows by definition of *Prop*.

In a second step, we deal with logical operators $\vee, \wedge, \neg$. The proof relies on the fact that $\delta^\#$ simply distributes over those operators. As a result, the case where $\psi = \psi_1 \vee \psi_2$ is representative. By definition of $\delta^\#$, we have $\delta_n^\#(\psi_1 \vee \psi_2) = \delta_n^\#(\psi_1) \vee \delta_n^\#(\psi_2)$. Since $\{\psi_1, \psi_2\} \subseteq Prop(Sub(\phi))$, by applying induction hypothesis, we have that $\{\delta_n^\#(\psi_1), \delta_n^\#(\psi_2)\} \subseteq Prop(Sub(\phi))$. The property follows by definition of *Prop*. □

The *Sub* (resp. $\delta^\#$) operator extends to sets C of $\mathcal{LTL}$ formulae in the obvious element-wise fashion, by stipulating that $Sub(C) = \bigcup_{\phi \in C} Sub(\phi)$ (resp. $\delta_n^\#(C) = \bigcup_{\phi \in C} \delta_n^\#(\phi)$). The following Corollary is an immediate consequence of Lemma 7.3.

Corollary 7.4. *Let Const be a finite set of $\mathcal{LTL}$ formulae. For all nodes $n \in NO$,*

$$\delta_n^\#(Prop(Sub(Const))) \subseteq Prop(Sub(Const))$$

This, together with the fact that $Prop(Sub(Const))$ by construction is closed under the logical operations, allows to conclude that the fixed point iteration induced by the original constraint system will take place entirely inside the domain $Prop(Sub(Const))$. To conclude that the iteration stabilises in a finite number of steps, it remains to prove that $Prop(Sub(Const))$ is finite modulo $\mathcal{LTL}$ equivalence. Lemma 7.5 is the key argument.

Lemma 7.5. *Let A be a finite set, $Prop(A)$ is finite modulo propositional equivalence.*

There are several normal forms for $Prop(A)$ (based on BDDs, conjunctive normal forms, ...) that will have to be considered when implementing the resolution. The actual representation is not important for arguing the termination of the resolution.

Theorem 7.6. *$Prop(Sub(Const))$ is finite modulo $\mathcal{LTL}$ equivalence.*

Proof. By construction, $Sub(Const)$ is finite. By Lemma 7.5 $Prop(Sub(Const))$ is finite modulo propositional equivalence. Furthermore, propositional equivalence implies $\mathcal{LTL}$ equivalence. As a result, Theorem 7.6 holds. □

Theorem 7.6 and the monotonicity of $\delta_n^\#, \wedge, \vee$ implies that the iteration is guaranteed to terminate.

THEOREM 7.7. *Let $F : (Var \to \mathcal{LTL}) \to (Var \to \mathcal{LTL})$ an iterator such that $F(\rho)(x_i) = E_i(\rho)$ and Const the set of ground $\mathcal{LTL}$ formulae occurring in the E_i. Then the iteration sequence*

$$v_i^0 = \mathbf{False}, \ldots, v_i^{k+1} = E_i[x_j \mapsto v_j^k], \ldots$$

stabilises in a finite number of steps.

EXAMPLE 7.8. *Let $\{x \Leftarrow \delta_n^\#(x), x \Leftarrow JDK(p)\}$ be a set of constraints. Its iterator is $F : (\{x\} \to \mathcal{LTL}) \to (\{x\} \to \mathcal{LTL})$ defined by*

$$F(\rho)(x) = (\delta_n^\#(x) \vee JDK(p))(\rho)$$

where $Attr(n) = \{p\}$. We observe that

$$\begin{array}{rcl} JDK(p) & \equiv & p\mathbf{U}_\forall(p \wedge Priv) \\ & \equiv & p\mathbf{U}_\exists(p \wedge Priv) \vee \neg(\mathbf{True}\mathbf{U}_\exists \neg p) \end{array}$$

We first show how to calculate $\delta_n^\#(JDK(p))$:

$$\begin{array}{rcl} & & \delta_n^\#(JDK(p)) \\ & = & \delta_n^\#(p\mathbf{U}_\exists(p \wedge Priv) \vee \neg(\mathbf{True}\mathbf{U}_\exists \neg p)) \\ & = & \delta_n^\#(p\mathbf{U}_\exists(p \wedge Priv)) \vee \neg\delta_n^\#(\mathbf{True}\mathbf{U}_\exists \neg p) \\ & = & \delta_n^\#(p \wedge Priv) \vee (\delta_n^\#(p) \wedge p\mathbf{U}_\exists(p \wedge Priv)) \\ & & \vee\neg(\delta_n^\#(\neg p) \vee (\delta_n^\#(\mathbf{True}) \wedge \mathbf{True}\mathbf{U}_\exists \neg p)) \\ & = & \mathbf{False} \vee (\mathbf{True} \wedge p\mathbf{U}_\exists(p \wedge Priv)) \\ & & \vee\neg(\mathbf{False} \vee (\mathbf{True} \wedge \mathbf{True}\mathbf{U}_\exists \neg p)) \\ & = & p\mathbf{U}_\exists(p \wedge Priv) \vee \neg(\mathbf{True}\mathbf{U}_\exists \neg p) \\ & = & JDK(p) \end{array}$$

We can then find the least fixed point by an iteration that stabilises after two steps:

$$\begin{array}{rcl} x_0 & = & \mathbf{False} \\ x_1 & = & \delta_n^\#(\mathbf{False}) \vee JDK(p) \\ & = & JDK(p) \\ x_2 & = & \delta_n^\#(JDK(p)) \vee JDK(p) \\ & = & JDK(p) \end{array}$$

Example 7.9 gives the properties inferred by our analysis for the code in Figure 1. From a security point of view, we are interested in the context inferred for entry nodes. In our case, the single entry point is n_0 and its secure calling context is characterised by

$$\sigma_{n_0}^\# = \mathbf{F}(Accountant) \Rightarrow \mathbf{F}(Manager)$$

As a consequence, for execution to be secure, node n_0 must be called from a stack s that satisfies $\sigma_{n_0}^\#$. Analysing this result more closely, we see that security is achieved in the following two cases:

- If $s \vDash \neg\mathbf{F}(Accountant)$ (*i.e.* if there is no node with the *Accountant* attribute on the call stack) then the execution is cut by the dynamic stack inspection in node n_3. It follows that the code is secured since the critical action is not executed.
- If $s \vDash \mathbf{F}(Manager)$ (*i.e.* there is a node with the *Manager* attribute on the call stack) then the dynamic stack inspection ensures that $s \vDash \mathbf{F}(Accountant)$. In this case, the critical action is executed in a secure fashion.

EXAMPLE 7.9. *The least solution of the fixed point system from Example 6.4 is given below. The solution is computed by fixed point iteration over the domain*

$$Prop(\{\mathbf{True}\mathbf{U}_\exists Accountant, \mathbf{True}\mathbf{U}_\exists Manager, Crit\})$$

Recall that $\mathbf{F}(\phi)$ stands for $\mathbf{True}\mathbf{U}_\exists\phi$. Each variable is initialised to the least element of its lattice: **False** *for $\tau^\#, \rho^\#$ constraints;* **True** *for $\sigma^\#$ constraints. For readability, the solution is given in terms of the syntactic sugar $\mathbf{F}$.*

$$\begin{array}{l} \tau_{n_0}^\# = \tau_{n_1}^\# = \tau_{n_3}^\# = \mathbf{F}(Accountant) \\ \tau_{n_4}^\# = \tau_{n_2}^\# = \mathbf{False} \end{array}$$

$$\begin{array}{l} \rho_{n_0}^\# = \rho_{n_1}^\# = \rho_{n_3}^\# = \mathbf{F}(Accountant) \\ \rho_{n_2}^\# = \rho_{n_4}^\# = \mathbf{True} \end{array}$$

$$\begin{array}{l} \sigma_{n_0}^\# = \sigma_{n1}^\# = \mathbf{F}(Accountant) \Rightarrow \mathbf{F}(Manager) \\ \sigma_{n_2}^\# = \mathbf{F}(Accountant) \wedge \mathbf{F}(Manager) \\ \sigma_{n_3}^\# = \sigma_{n_4}^\# = \mathbf{True} \end{array}$$

8. RELATED WORK

The concept of stack inspection has been formalised in various ways. Wallach and Felten [24] formalise the Java stack inspection using a belief logic. The paper is based on the security mechanisms as implemented in Netscape, which can be seen as an extension of the JDK 1.2 mechanisms, allowing to grant specifically named permissions to a piece of code. Granting permission P to code C_1 adds the belief statement $Ok(P)$ to the set of beliefs held in the current stack frame, and calling code C_2 records the beliefs of the earlier stack frames by adding the statement C_1 **says** $Ok(P)$ to the belief set for the stack frame for C_2. Fournet and Gordon [11] provides an alternative formalisation of stack inspection based on operational semantics. Their aim is to establish laws for equational reasoning in order to validate program transformations in the presence of stack inspection.

The present work builds on the verification techniques developed by Besson, Jensen and others [14, 4] in which model checking techniques are combined with whole-program static analysis techniques in order to verify global security properties of stack-inspecting code. The methods presented in *op. cit.* differ from what is presented here by providing essentially yes/no answers to a given verification problem, whereas the inference algorithm here must infer (a symbolic representation of) the secure calling contexts of a method. Barthe *et al.* [2] has developed a compositional proof system for verifying temporal properties of control flow graphs. This leads to a compositional analysis of secure applet interaction but it does not deal with stack inspection.

Schneider [20] introduced the idea of *security automata* as a formalism for defining security properties. Security automata are a class of Büchi automata that define what are the legal sequences of actions that a system can take. Erlingsson and Schneider [10] and Colcombet and Fradet [5] both propose to use such automata to monitor an executing system such that an action about to be executed can be prevented if it is deemed illegal by the security automaton. Thus, rather than proving statically that a property is verified by a program as we do in the present work, the corresponding security automaton is *inserted* (using program transformations and optimising analysers) into the program

to dynamically monitor its execution. This approach carries a run-time penalty but allows to use programs in a secure fashion even when their security cannot be proved statically.

There is relatively little work on analysing stack-inspecting code. The closest to ours is that of Skalka and Smith [21] who propose λ_{sec}, a lambda calculus extended with primitives that correspond to the stack inspection primitives in Java. Permissions can be granted and checked for, and code can be marked as privileged. A type system allows to infer function types of the form $\sigma \xrightarrow{P} \tau$ that describe the set P of permissions necessary for executing a function. In a sequel paper, Pottier, Skalka and Smith [19] recast the type system in more standard terms by translating λ_{sec} into a standard lambda calculus by generalising Wallach's security-pasing programming style [23] to higher-order functions. Bartoletti, Degano and Ferrari [3] develop a data flow analysis for control flow graphs that determines the set of permissions that will always or will never be available at a given node in the graph. This information can be used to optimise the stack inspection algorithm in those cases where the analysis determines that a given security will always be thrown or will never be thrown. Compared to our work, these papers are more restricted in scope since they are only concerned with verifying the property that the program "does not go wrong" *i.e.*, that the program does not raise a security exception because a stack inspection failed. In contrast, our analysis can verify arbitrary invariants of call stacks as long as these are expressible in $\mathcal{LTL}$.

9. CONCLUSIONS

We have presented a static program analysis for inferring secure calling contexts for stack-inspecting methods relative to a given global security property. We stress that the method works for arbitrary global properties that can be expressed using our $\mathcal{LTL}$ specification formalism. In this respect our analysis is more general than other analyses for stack-inspecting code [21, 19] that are concerned with inferring the permissions required for all stack inspections to succeed. The analysis is proved correct with respect to a formal semantics and is implemented by a fixed-point iteration over an abstract domain built of temporal properties.

The constraint-based analysis has been prototyped in Caml and experimented on a sample of small control flow graphs. Although the prototype has mainly served to avoid having to calculate the iterations for the examples by hand, its performance indicates that proper use of BDD-representations might allow to treat larger, more realistic applications.

The next step is to extend the analysis to *fragments* of control flow graphs in order to deal with software components in which methods make calls to virtual methods that might not be available for analysis. The current framework is well suited for this because properties of unknown methods can be represented as free variables in the generated constraints. However, we would have to extend the iterative constraint resolution technique to deal with constraints containing free variables, in essence calculating (an intensional representation of) the relation between properties of the "imported" unknown methods and the properties of the methods offered by the component. This would constitute a substantial step towards defining a notion of secure interfaces for stack-inspecting modules.

10. REFERENCES

[1] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proc. of OOPSLA'96*, volume 31(10) of *ACM SIGPLAN Notices*, pages 324–341, New York, 1996. ACM Press.

[2] G. Barthe, D. Gurov, and M. Huisman. Compositional verification of secure applet interactions. In *Proc. of Foundations of Software Engineering (FASE'02)*. To appear in Springer LNCS, 2002.

[3] M. Bartoletti, P. Degano, and G. Ferrari. Static analysis for stack inspection. In *Proc. of Int. workshop on Concurrency and Coordination (ConCoord 2001)*, Electronic Notes in Theoretical Computer Science vol. 54. Elsevier, 2001.

[4] F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model ckecking security properties of control flow graphs. *Journal of Computer Security*, 9:217–250, 2001.

[5] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. of 27 ACM Symp. on Principles of Programming Languages (POPL'00)*, pages 54–66. ACM Press, 2000.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proc. of 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977. ACM Press, New York.

[7] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proc. of the ACM Symposium on Artificial Intelligence and Programming Languages, SIGPLAN Notices, (8)*, pages 1–12, Rochester, NY, 1977.

[8] P. Cousot and R. Cousot. Formal language, grammar and set constraint-based program analysis by abstract interpretation. In *Proc. of the ACM Conf. on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 170–181. ACM Press, 1995.

[9] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072, Amsterdam, 1990. Elsevier Science Publishers.

[10] U. Erlingsson and F. Schneider. SASI enforcement of security policies: A retrospective. In *New Security Paradigms Workshop*. ACM Press, 2000.

[11] C. Fournet and A. Gordon. Stack inspection: Theory and variants. In *Proc. of the 29th ACM Symp. on Principles of Programming Languages (POPL'02)*. ACM Press, 2002.

[12] L. Gong. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, Dec. 1997.

[13] D. Grove, G. Furrow, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proc. of Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, 1997.

[14] T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security properties. In *Proc. of the 20th IEEE Symp. on Security and Privacy*, pages

89–103. New York: IEEE Computer Society, 1999.

[15] Microsoft Corp. *Secure Coding Guidelines for the .NET Framework*. Microsoft Corp., 2002.

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

[17] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

[18] H. Pande and B. Ryder. Data-flow-based virtual function resolution. In *Proc. of 3rd Static Analysis Symposium (SAS'96)*. Springer LNCS vol. 1145, 1996.

[19] F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. In D. Sands, editor, *Proc. of the 10th European Symposium on Programming (ESOP'01)*, pages 30–45. Springer LNCS vol. 2028, 2001.

[20] F. Schneider. Enforceable security policies. *ACM Trans. on Information and System Security*, 3(1):30–50, 2000. Preliminary version appeared as Cornell Univ. Tech. Rep. TR 98–1664,1998.

[21] C. Skalka and S. Smith. Static enforcement of security with types. In *Proc. of 5th International Conference on Functional Programming (ICFP'00)*, pages 34–45. ACM Press, 2000.

[22] M. Y. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag Inc., New York, NY, USA, 1996.

[23] D. S. Wallach. *A new approach to mobile code security*. PhD thesis, Dept. of Computer Science, Princeton University, Jan. 1999.

[24] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *1998 IEEE Symposium on Security and Privacy*, May 1998.