JFP **15** (2): 179–217, 2005. © 2005 Cambridge University Press
DOI: 10.1017/S0956796804005465 Printed in the United Kingdom

Interfaces for stack inspection

FRÉDÉRIC BESSON, THOMAS DE GRENIER DE LATOUR
and THOMAS JENSEN

IRISA/CNRS and INRIA-Rennes, Campus de Beaulieu, F-35042 Rennes, France
(*e-mail:* fbesson@irisa.fr, degrenie@irisa.fr, jensen@irisa.fr)

Abstract

Stack inspection is a mechanism for programming secure applications in the presence of code from various protection domains. Run-time checks of the call stack allow a method to obtain information about the code that (directly or indirectly) invoked it in order to make access control decisions. This mechanism is part of 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. A further problem is how such verification can be carried out in an incremental fashion. Incremental analysis is important for avoiding re-analysis of library code every time it is used, and permits the library developer to reason about the code without knowing its context of deployment. We propose a technique for inferring interfaces for stack-inspecting libraries in the form of *secure calling context* for methods. 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. The technique is a constraint-based static program analysis implemented via fixed point iteration over an abstract domain of linear temporal logic properties.

Capsule Review

This paper presents a novel static program analysis for reasoning about, and implementing, stack-based security policies in programming languages. A general stack-based model is considered, where security contexts are defined by the stack as in the JDK stack inpection model, but checks other than JDK stack inspection may be defined. In addition to local checks, global security policies may be specified, with the analysis ensuring that local checks enforce global policies. The analysis is modular, in that properties of code that combines codebases can be composed of separate analysis of the latter. These features render the analysis useful and flexible, with a verification mechanism that significantly improves reliability of stack-based security policies for programming languages.

1 Introduction

Programs that load code dynamically from sites of various degrees of trust pose a challenge in terms of security. Access to resources (confidential data, computing time, printers, etc.) must be controlled such that only code that is authorised to perform a certain operation can do so. The programmer is faced with the task of enforcing such security requirements by combining a number of programming language and operating system features such as strong typing, scope reduction of variables, sandboxing, run-time checks on the state of execution, etc. The options are varied and it may be difficult to estimate the consequences of a particular choice:

- has the desired level of security been attained?
- has security been implemented efficiently, without (too much) redundancy in the protection mechanisms?
- can we certify the security so obtained in a formal manner?

In this article we present a static program analysis technique that can assist the programmer in implementing access control using the *stack inspection* mechanism found in the Java and the .NET security architecture.

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 (Gong, 1997) as well as that of .NET (LaMacchia *et al*., 2002).

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 call 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 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. The logic would also be useful for eliminating such redundant checks but we do not pursue this issue further here (see Skalka & Smith (2000) and Pottier *et al.* (2001)).

The security properties that we consider here are properties on the control flow of the program which can be expressed in terms of the execution stack. Examples include access control properties like “method A never accesses resource B”, “method A only accesses resource B if it holds permission P” or “all access to resource B has first been cleared by authority C" where A, B and C are identifiable entities in a code source. Certain control-flow properties such as "if A at some point calls B then A cannot call C later on in the execution, and vice versa" are not taken into account by our verification because they are not expressed as a global invariant of the execution stacks.

To address the above-stated problem, verification mechanisms based on static program analysis and model checking have been proposed (Besson *et al.*, 2001; Jensen *et al.*, 1999). 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 program modifications implies a long re-analysis of the program. Also, it may be that the source code of a library is not meant to be public; hence, it is necessary to have a means of describing the part of its behaviour that is relevant for security. For these reasons, it is desirable to render the verification technique of Besson *et al.* (2001) more modular by developing an analysis that for each method calculates an interface describing under what circumstances the method will execute safely. Such an interface will take the form of 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 these stacks as current call stack.

This article describes a solution to this problem in the setting of closed libraries whose methods can be called from outside but where the methods of the library only can call other methods within the library – in particular, there is no provision for call-backs via objects passed to library methods from outside. Making this restriction implies that available control flow analysis techniques can be used to build the control flow graph representation of the library on which we base our analysis. We propose an analysis that will derive weakest pre-conditions to be attached to the entry points of a library. Under these weakest preconditions, no security violation will happen inside the library.

The technical contributions of the article 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.

These contributions provide a theoretical framework for validating Java or CLR libraries that use stack inspection. Additional language features have to be taken into account in order to build a full-scale verification tool, in particular certain types of data flow (e.g. strings for building file names or permissions) and the handling of security exceptions. Finally, there are a number of security properties pertaining to the execution history that stack inspection cannot enforce and that cannot be validated within our framework. We return to this point in section 9.

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 defines 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 shows how the verification technique can be used to reason about the security of an (idealized) bank account application. Section 9 compares with related work and section 10 concludes and outlines further work.

A preliminary version of this article appeared in the PPDP 2002 proceedings (Besson *et al*., 2002). In this article we have added certain proofs missing in Besson *et al.* (2002), in particular the proof of exactness of our resolution method. Proofs are now done using a proof technique based on the notion of transitions in contexts, defined in section 2.2, We have also included a worked example in section 8 and considerably expanded the section on related work to take into account the multitude of new developments that have occurred in the rapidly developing area of verification for stack inspection.

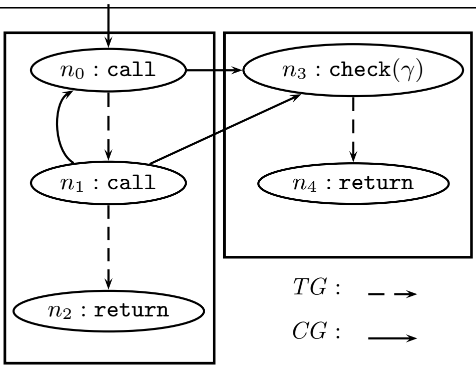
2 Program model

As an abstract program model we will use the particular kind of *control-flow graph* (CFG) that was used by Jensen *et al.* (1999) to represent stack-inspecting programs. This model is adequate for sequential procedural programming languages. 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 types of edges in order to distinguish between two types of control flow. Sequential composition of code is represented by a transfer edge (labeled with TG) between nodes. Method calls are modeled by call edges (labeled with CG) that bind call sites to their potential entry points.

Our model is also equipped with a labeling function AT 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.

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

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



$$\begin{array}{lll} AT: & & \\ n_0, n_1 & \mapsto & \{System\} \\ n_2 & \mapsto & \{System, Crit\} \\ n_3, n_4 & \mapsto & \{Manager\} \end{array}$$

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

Fig. 1. A control flow graph.

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,

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

Control-flow graphs are subject to the following well-formedness constraints. All check nodes and call nodes 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.

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 labeled 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 data flow 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 (Grove *et al.*, 1997). 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 (Bacon & Sweeney, 1996). A constraint based data flow analysis, as proposed by Palsberg & Schwartzbach (1994) 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 & Ryder (1996). This is the only place in the analysis where data flow is taken into account. In the remainder of the paper we will work on a representation of the program that was built using data flow information but which only retains information about the control flow.

2.2 Semantics of a CFG

In previous works (Besson *et al.*, 2001), 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. The logic of interest here is linear temporal logic that will be formally presented in section 3.

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 $n, n', m \in NO$. 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.

Definition 2.3 (*Small-step operational semantics*)

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

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

Notice that there is no transition for a $\texttt{check}(\gamma)$ node if the property γ is not satisfied. Thus we model failure of a check by stopping the execution. This is a simplification with respect to what happens in the JVM and the CLR where a security exception is thrown. This behaviour can be taken into account (see section 9 for related work that deals with this issue) but we have chosen not to do so since a proper treatment of exceptions would complicate the presentation considerably.

The following relation $\xrightarrow{[ctx]}$ describes the behaviour of a piece of code in a particular calling context (call stack) ctx. It is a convenient means for characterising all those states (*i.e.*, call stacks) that are reachable within a specific method invocation.

Definition 2.4 (*Transitions in context*)

$$\frac{ctx{:}s \rhd ctx{:}s' \quad |s|{>}0 \quad |s'|{>}0}{ctx{:}s \xrightarrow{[ctx]} ctx{:}s'}$$

We furthermore define the relation $\xrightarrow{[ctx]}{}^{+}$ to be the standard transitive (but not necessarily reflexive) closure of the relation $\xrightarrow{[ctx]}$. The relation $\xrightarrow{[ctx]}{}^{*}$ is a transitive, partially reflexive closure of $\xrightarrow{[ctx]}$ that relates a state s to itself only if s has ctx as a prefix. This excludes all states reachable outside the particular call identified by ctx. These relations are defined inductively as follows.

Definition 2.5 (*Reachable states in context*)

$$\frac{s \xrightarrow{[ctx]} s' \quad s' \xrightarrow{[ctx]}{}^{*} s''}{s \xrightarrow{[ctx]}{}^{+} s''} \qquad \frac{s = ctx{:}s' \quad |s'|{>}0}{s \xrightarrow{[ctx]}{}^{*} s} \qquad \frac{s \xrightarrow{[ctx]}{}^{+} s'}{s \xrightarrow{[ctx]}{}^{*} s'}$$

We also define $\xrightarrow{[ctx]}{}^{i}$ as the composition of exactly i $\xrightarrow{[ctx]}$ relations, and $\xrightarrow{[ctx]}{}^{i..j}$ as the relation between some stacks s and s' such that $\exists(k{:}[i..j]).s \xrightarrow{[ctx]}{}^{k} s'$.

A first property of $\xrightarrow{[ctx]}$ is that it is strictly equivalent to $\rhd$ if ctx is ε, the zero length stack. In particular, this means that all reachable states from an execution beginning at the main node n_0 is in relation with the stack n_0:

Property 2.6
$\forall(s{:}Stacks).\quad n_0 \xrightarrow{[\varepsilon]}{}^{*} s \quad\Leftrightarrow\quad n_0 \rhd^{*} s$

Another property is that any transition that holds in a given context $ctx{:}ctx'$ also holds for any of its prefixes ctx.

Property 2.7
$\forall(s, s', ctx, ctx'{:}Stacks).$

$$ctx{:}ctx'{:}s \xrightarrow{[ctx{:}ctx']} ctx{:}ctx'{:}s' \quad\Rightarrow\quad ctx{:}ctx'{:}s \xrightarrow{[ctx]} ctx{:}ctx'{:}s'$$

Both properties can be proved by inspection of the rules; we omit the details.

2.3 Collecting semantics

Our main goal is to ensure safety properties that are invariant properties of the reachable stacks. To this end, it suffices and is more convenient to work with a more abstract *collecting semantics* (Cousot & Cousot, 1977a; Nielson *et al.*, 1999) that collects the set of reachable states in a given context. This is exactly what the relations of "reachable states in context" from Definition 2.5 do. We can define collecting semantics as follows:

$$\begin{array}{rclrcl} \{\!|s{:}n|\!\}^{i} & = & \{s' \mid s{:}n \xrightarrow{[s]}{}^{i} s'\} & \{\!|s{:}n|\!\}^{i..j} & = & \{s' \mid s{:}n \xrightarrow{[s]}{}^{i..j} s'\} \\ \{\!|s{:}n|\!\}^{+} & = & \{s' \mid s{:}n \xrightarrow{[s]}{}^{+} s'\} & \{\!|s{:}n|\!\}^{*} & = & \{s' \mid s{:}n \xrightarrow{[s]}{}^{*} s'\} \end{array}$$

By definition of the reachable states in context s, the stacks collected by the semantics are all prefixed by s. This means that the collection process will happen only within the method that the node n belongs to and the methods that are called from n or other nodes reachable sequentially from n. As we show in the next section, this limitation of the scope of the semantics is well adapted to the definition of a notion of safety at library level, independently of the code that uses it.

When applied to a stack reduced to a single node (entry point of a program), our contextual collecting semantics reduces to a classical collecting semantics:

Property 2.8
$\forall(s{:}Stacks,\ n_0{:}NO).\quad n_0 \rhd^{*} s \quad\Leftrightarrow\quad s \in \{\!|n_0|\!\}^{*}$

This semantics also has some good properties in terms of compositionality.

Property 2.9
$\forall(s, s'{:}Stacks,\ i, j, k, l{:}\mathbb{N}).$

$$s' \in \{\!|s|\!\}^{i..j} \quad\Rightarrow\quad \{\!|s'|\!\}^{k..l} \subseteq \{\!|s|\!\}^{i+k..j+l}$$

In particular, all elements that can be collected from a stack s' are also reachable from a stack s if s' is reachable from s, i.e. $s' \in \{\!|s|\!\}^{+} \Rightarrow \{\!|s'|\!\}^{*} \subseteq \{\!|s|\!\}^{+}$.

The following three lemmas make explicit the set of stacks that can be collected by this semantics, depending of the type of a node n.

For a `return` node, as expected, we can't collect anything.

Lemma 2.10
$\forall(s{:}Stacks,\ n{:}NO,\ i{:}\mathbb{N})$. if $IS\,(n) = \mathtt{return}$, then

$$\{\!|s{:}n|\!\}^{1..i} = \emptyset$$

Proof
The proof is immediate since there is no rule for a derivation from a `return` node with context s. $\square$

For a `check` node, we can only collect stacks reachable from the nodes that follow n in sequence, and this only if the check formula is satisfied.

Lemma 2.11
$\forall(s, s'{:}Stacks,\ n{:}NO,\ i{:}\mathbb{N})$. if $IS\,(n) = \mathtt{check}(\gamma)$, then

$$s' \in \{\!|s{:}n|\!\}^{1..i} \text{ iff } \bigwedge \left\{ \begin{array}{l} s{:}n \vDash \gamma \\ \exists(n'{:}NO).n \overset{TG}{\to} n' \ \wedge\ s' \in \{\!|s{:}n'|\!\}^{0..i-1} \end{array} \right.$$

Proof
The left-to-right implication follows from the fact that $\triangleright_{check}$ is the only semantics rule which applies for a `check` node. Hence, a derivation $s{:}n \xrightarrow{[s]}{}^{1..i} s'$ is of the form $s{:}n \xrightarrow{[s]} s{:}n' \xrightarrow{[s]}{}^{0..i-1} s'$ and only exists if $s{:}n \vDash \gamma$ and $n \overset{TG}{\to} n'$.

The right-to-left implication is also immediate because $\triangleright_{check}$ can always be applied under this condition. $\square$

Finally, for a `call` node, the reachable stacks in context s are those reachable in the called methods together with those reachable after the return from one of these calls. To establish this we prove two lemmas. Lemma 2.12 describes the possible behaviour of a stack in a given context after one or several method calls. This rather technical lemma is used to prove the following Lemma 2.13 which is a reformulation of Lemma 2.12 in terms of the collecting semantics. Readers might want to skip directly to the more readable Lemma 2.13.

Lemma 2.12
$\forall(i{:}\mathbb{N},\ s, s', s''{:}Stacks,\ n, m{:}NO)$.

$$s{:}n{:}s'{:}m \xrightarrow{[s]}{}^{i} s'' \Leftrightarrow \bigvee \left| \begin{array}{ll} s{:}n{:}s'{:}m \xrightarrow{[s:n]}{}^{i} s'' & (H_1) \\ \exists(r, n'{:}NO,\ j,k,l{:}\mathbb{N}). & \\ \bigwedge \left| \begin{array}{l} IS\,(r) = \mathtt{return} \wedge n \overset{TG}{\to} n' \wedge i = j+k+l \\ s{:}n{:}s'{:}m \xrightarrow{[s:n]}{}^{j} s{:}n{:}s'{:}r \xrightarrow{[s]}{}^{k} s{:}n' \xrightarrow{[s]}{}^{l} s'' \end{array} \right. & (H_2) \end{array} \right.$$

Proof
Proof of this lemma 2.12 for the left-right implication is obvious because of property 2.7. For the right-left implication, the proof is by induction on the derivation lenght. We call *second segment* the s' part of the stack.

- *Base case*: $s{:}n{:}s'{:}m \xrightarrow{[s]}{}^{0} s''$ is only verified if $s'' = s{:}n{:}s'{:}m$, which implies the H_1 alternative of the Lemma.
- *Inductive case*: A $i{+}1$ steps long derivation is of this form:

$$s{:}n{:}s'{:}m \xrightarrow{[s]} s_1 \xrightarrow{[s]}{}^{i} s''$$

One step of the proof is by case on the type of the node m:

— $IS(n){=}\texttt{check}(\gamma)$: By definition of $\triangleright_{check}$, there exists a node m' such that $s_1 = s{:}n{:}s'{:}m'$. Because the derivation exists also in context $s{:}n$, the property hold by induction.

— $IS(n){=}\texttt{call}$: By definition of $\triangleright_{call}$, there exists a node m' such that $s_1 = s{:}n{:}s'{:}m{:}m'$. We use the induction hypothesis with $s'{:}m$ being the stack second segment.

If H_1 holds at step i, then it also holds at step $i{+}1$ ($s{:}n{:}s'{:}m \xrightarrow{[s:n]} s{:}n{:}s'{:}m{:}m' \xrightarrow{[s:n]}{}^{i} s''$).

If H_2 holds at step i, then there exists a node r such that $IS(r){=}\texttt{return}$ and a node n' such that $n \overset{TG}{\rightarrow} n'$ and a derivation

$$s{:}n{:}s'{:}m{:}m' \xrightarrow{[s:n]}{}^{j} s{:}n{:}s'{:}m{:}r \xrightarrow{[s]}{}^{k} s{:}n' \xrightarrow{[s]}{}^{l} s''.$$

Stack lengths indicate that $k{>}0$. Consequently, and by $\triangleright_{return}$ definition, there exists a node m'' such that $m \overset{TG}{\rightarrow} m''$ and the full derivation can be rewritten:

$$s{:}n{:}s'{:}m \xrightarrow{[s:n]}{}^{j+2} s{:}n{:}s'{:}m'' \xrightarrow{[s]}{}^{k-1+l} s''$$

We can now apply again our induction hypothesis on the second part, and the property immediatly follows in both alternatives.

— $IS(n){=}\texttt{return}$: If s' is an empty stack, then there exists a node n' such that $n \overset{TG}{\rightarrow} n'$, and $s_1 = s{:}n'$. The full derivation is $s{:}n{:}m \xrightarrow{[s]} s{:}n' \xrightarrow{[s]}{}^{i} s''$, which match the H_2 alternative ($j = 0$ and $k = 1$).

If s' is not empty, then it is of form $s_2{:}m_2$, and there exists a node m_2' such that $m_2 \overset{TG}{\rightarrow} m_2'$, and the first derivation step is $s{:}n{:}s_2{:}m_2{:}m \xrightarrow{[s]} s{:}n{:}s_2{:}m_2' \xrightarrow{[s]}{}^{i} s''$. We can again use our induction hypothesis on the next i steps ($s{:}n{:}s_2{:}m_2' \xrightarrow{[s]}{}^{i} s''$). If H_1 holds, then it also holds on the full derivation. If it is H_2 which is verified, then there exist a return node r and a node n' such that $n \overset{TG}{\rightarrow} n'$, and a derivation

$$s{:}n{:}s_2{:}m_2' \xrightarrow{[s:n]}{}^{j} s{:}n{:}s_2{:}r \xrightarrow{[s]}{}^{k} s{:}n' \xrightarrow{[s]}{}^{l} s''.$$

Hence, H_2 is also verified by the full derivation:

$$s{:}n{:}s'{:}m \xrightarrow{[s:n]}{}^{0} s{:}n{:}s'{:}m \xrightarrow{[s]}{}^{j+k+1} s{:}n' \xrightarrow{[s]}{}^{l} s''.$$

□

Lemma 2.13
$\forall(s, s' {:} Stacks,\ n{:}NO,\ i{:}\mathbb{N})$. if $IS(n) = \texttt{call}$, then :

$$s' \in \{\!|s{:}n|\!\}^{1..i} \text{ iff } \exists(m{:}NO).$$
$$n \overset{CG}{\rightarrow} m \wedge \bigvee \begin{cases} s' \in \{\!|s{:}n{:}m|\!\}^{0..i-1} \\ \bigwedge \begin{cases} \exists(k < i,\ r{:}NO). \\ \quad IS(r) = \texttt{return} \wedge s{:}n{:}r \in \{\!|s{:}n{:}m|\!\}^{k} \\ \exists(n'{:}NO).n \overset{TG}{\rightarrow} n' \wedge s' \in \{\!|s{:}n'|\!\}^{0..i-k-1} \end{cases} \end{cases}$$

Proof
The left-right implication comes from the fact that $\triangleright_{call}$ is the only semantics rule which applies for a `call` node. Hence, a derivation $s{:}n \xrightarrow{[s]}{}^{1..i} s'$ is of the form $s{:}n \xrightarrow{[s]} s{:}n{:}m \xrightarrow{[s]}{}^{0..i-1} s'$ and only exists if $n \overset{CG}{\rightarrow} m$. Application of Lemma 2.12 to $s{:}n{:}m$ let us conclude.

For the right-left implication, the proof is immediate since $\triangleright_{call}$ rule can be applied for any node m provided $n \overset{TG}{\rightarrow} m$. again, we conclude by applying Lemma 2.12 to $s{:}n{:}m$. □

The next lemma states that if a transfer transition is possible from a given stack $s{:}n$, then transfer transitions are also possible through all other $\overset{TG}{\rightarrow}$ edge of the same origin n. This is a consequence of the non-determinism in our program model.

Lemma 2.14
$\forall(i{:}\mathbb{N}^+,\ s{:}Stacks,\ n, n', n''{:}NO)$.

$$s{:}n' \in \{\!|s{:}n|\!\}^{1..i} \wedge n \overset{TG}{\rightarrow} n'' \quad \Rightarrow \quad s{:}n'' \in \{\!|s{:}n|\!\}^{1..i}$$

Proof
The proof is by case analysis of the type of node n and relies on the above three lemmas.

- $IS(n) = \texttt{return}$: the lemma is vacuously true.
- $IS(n) = \texttt{check}(\gamma)$: by lemma 2.11, $s{:}n' \in \{\!|s{:}n|\!\}^{1..i} \Rightarrow s \models \gamma$. Since $s{:}n'' \in \{\!|s{:}n''|\!\}^{0..i-1}$, we can use this lemma again to conclude that $s{:}n'' \in \{\!|s{:}n|\!\}^{1..i}$.
- $IS(n) = \texttt{call}$: by lemma 2.13, and because there is no j such that $s{:}n' \in \{\!|s{:}n{:}m|\!\}^{0..j}$, we have $s{:}n' \in \{\!|s{:}n|\!\}^{1..i} \Rightarrow \exists(k{<}i,\ r{:}NO).IS(r) = \texttt{return} \wedge s{:}n{:}r \in \{\!|s{:}n{:}m|\!\}^{k}$. Since $s{:}n'' \in \{\!|s{:}n''|\!\}^{0..i-1}$, we can use this lemma again to conclude that $s{:}n'' \in \{\!|s{:}n|\!\}^{1..i}$.

□

3 Security properties

In this section we define formally the set of security properties that we aim at verifying. These are invariant properties of the form "all call stacks satisfy a given property ϕ". The stack property ϕ is a property on finite sequences of nodes and a number of formalisms for expressing such properties are available. We have chosen linear temporal logic (Emerson, 1990), $\mathcal{LTL}$, that will be interpreted over the set *Stacks* of finite sequences of nodes that correspond to call stacks. There are several reasons for choosing this logic. First, it is a standard, well-studied formalism of verification. Second, 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. We give a couple of examples in Section 3.1.

It can be argued that $\mathcal{LTL}$ in certain cases is too expressive and that a fragment (e.g. without nesting of temporal operators) would suffice. While this might be a fruitful topic to investigate, we have chosen to stick with the full formalism to develop the more general theory.

$\mathcal{LTL}$ 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:

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

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. Recall that stacks grow from left to right. With the labelling function AT that gives the attributes of each node n, the semantics of the core $\mathcal{LTL}$ operators is defined as follows:

$$\begin{array}{lll}
s \models \mathbf{True} & & \\
s \models p & \text{iff} & |s| > 0 \text{ and } p \in AT(s_0) \\
s \models \neg\phi & \text{iff} & \text{not } (s \models \phi) \\
s \models \phi_1 \vee \phi_2 & \text{iff} & s \models \phi_1 \text{ or } s \models \phi_2 \\
s \models \mathbf{X}_\exists\phi & \text{iff} & |s| > 1 \text{ and } s^1 \models \phi \\
s \models \phi_1\mathbf{U}_\exists\phi_2 & \text{iff} & \exists(k{:}[0..|s|-1]).\ s^k \models \phi_2 \\
& & \quad \text{and } \forall(i{:}[0..k-1]).\ s^i \models \phi_1
\end{array}$$

Informally, a stack always satisfies **True** and satisfies 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 satisfies $\mathbf{X}_\exists\phi$ if the stack deprived from its top is non-empty and satisfies ϕ. Finally, a $\phi_1\mathbf{U}_\exists\phi_2$ formula is satisfied by stacks such that their exists a sub-stack modelling ϕ_2 and all the previous sub-stacks model ϕ_1.

From the core syntax, usual propositional syntactic sugar (**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 (**G** and **F**) and an emptiness property (ε):

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

We can informally explain the 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 satisfied at least one time", $\mathbf{G}\phi$ for "ϕ is always satisfied", and $\phi_1 \mathbf{U}_\forall \phi_2$ for "ϕ_1 is either always satisfied or satisfied until ϕ_2 is". The *Weak Next* ($\mathbf{X}_\forall$) is a *Next* variant which is always satisfied by stacks of one or zero element, and ε is only satisfied by the empty stack.

Finally, we introduce the *concretisation* function

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

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 (Besson *et al.*, 2001) 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 7.8) 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 *secure calling contexts*, relative to a global security property. A secure calling context for a node n is a stack s such that all executions starting from $s{:}n$ in the sub-graph rooted at n will respect the global security property. The set of secure calling contexts for a node n relative to a global security property φ is formalized by the function $sec : NO \to \mathcal{P}(Stacks)$.

$$sec_n = \{s \mid \{\!|s{:}n|\!\}^* \subseteq concr(\varphi)\}$$

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. To take this effect of stack inspection into account, 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* contains all the stacks that pass stack inspection. Formally, $trans : NO \to \mathcal{P}(Stacks)$ is defined by:

$$trans_n = \{s \mid \exists (n'{:}NO).\ s{:}n' \in \{\!|s{:}n|\!\}^+\}$$

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 s then this stack belongs to $returns(n)$. Formally, $returns : NO \to \mathcal{P}(Stacks)$ is given by:

$$returns_n = \{s \mid \exists (r{:}NO).IS(r) = \texttt{return} \wedge s{:}r \in \{\!|s{:}n|\!\}^*\}$$

We observe that it is always safe to discard elements from secure calling contexts in *sec*. Indeed, it is a conservative approximation only to keep a subset of those contexts that do not trigger a security violation. Unlike *sec*, elements cannot be removed from *trans* and *returns*: this would mean forgetting about certain execution paths and, perhaps, the execution path that made the predicate valid. In this case, it is a conservative to consider a superset of those calling contexts that pass or return from a node. Thus, in general, we are interested in finding the greatest set of secure calling contexts and the smallest sets that pass or return from a node. 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, $\mathcal{P}(Stacks)$, 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 set of secure calling contexts for each node in G.
2. The system $[\![G]\!]^C$ is formulated using an extensional representation of sets which may be infinite and thus impossible to compute with directly. We therefore 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 $\mathcal{LTL}$.

We stress that this division of the analysis into two phases is done in order to ease the proof of correctness (which now just operates on sets) and in order not to tie the analysis to a particular representation of sets such as the LTL representation given in section 6.

For each node n in a given CFG G we introduce a triple of sets, $(\rho_n, \sigma_n, \tau_n) \in \mathcal{P}(Stacks)^3$ and generate a number of set constraints. The intention that the least solution to the set of constraints generated by a CFG is such that $\rho_n = returns_n$, $\sigma_n = sec_n$ and $\tau_n = trans_n$ for all nodes n in the CFG.

In addition to the standard set-theoretic operators $\cup$ and $\cap$, the constraints are constructed using a complement operator such that $\overline{S} = Stacks \setminus S$. We also use an projection 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 $n \in NO$ be a node. Define $\delta_n : \mathcal{P}(Stacks) \to \mathcal{P}(Stacks)$ 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. Notice that a variable can be constrained (*i.e.*, appear to the left) in several constraints. It is thus the joint effect of these

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

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

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

$$\sigma_{transfer}\ \frac{n \overset{TG}{\to} n'}{\sigma_n \subseteq \overline{\tau_n} \cup \sigma_{n'}}$$

Fig. 2. Syntax-directed definition of the set constraints $[\![G]\!]^C$.

constraints that determines the value of the variable. Also, the variables are constrained differently: ρ and τ are over-approximations, because we can safely consider more control flows than the actual ones, whereas σ must be an under-approximation that only contains secure calling context, at the price of possibly missing some.

We here provide an informal justification of the rules. The first one, τ_{check}, is a direct application of the δ operator: if a stack s leads to a $\texttt{check}(\gamma)$ node n, then execution will continue if $s{:}n$ satisfies γ, i.e. if $s \in \delta_n(concr(\gamma))$. Similarly, rule τ_{call} states that execution can continue through a `call` node n if the called method m returns. 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_{global} ensures that for a stack s to be a secure calling context for a node n, $s{:}n$ must at least satisfy the global security invariant φ. 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 projection 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 starting from n' with the same stack is also secure *or* the execution never tranfers from n to n' with this stack (i.e. s belongs to $\overline{\tau_n}$).

Example 5.2

Continuing our running example, the nodes n_3 and n_4 from Figure 1 give rise to the following set constraints.

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

5.1 Existence of a solution

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 (Cousot & Cousot, 1995) 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 projection operator δ_n is monotone over the lattice $(\mathcal{P}(Stacks), \subseteq)$ and, hence, also over its dual $(\mathcal{P}(Stacks), \supseteq)$.

As argued in section 4, we are interested in the *smallest* sets of stacks satisfying the *trans* and *return* predicates and in the *greatest* set of stacks satisfying the *sec* predicate. To be able to characterise the desired information as the least solution to a set of lattice constraints, we define the following lattice of solutions.

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

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

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

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 use operators like δ_n which are monotone. The only non-trivial case are the constraints of the form $\sigma_n \supseteq \overline{\tau_n} \cup \sigma_{n'}$ 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

$$\overline{\tau_n^1} \cup \sigma_{n'}^1 \supseteq \overline{\tau_n^2} \cup \sigma_{n'}^2$$

which implies monotonicity since the σ 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 $[\![G]\!]^C$.

5.2 Correctness

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 only contains secure calling context (*i.e.*, $\sigma_n \subseteq sec_n$);
- τ_n contains an over-approximations of the call contexts that pass n (*i.e.*, $\tau_n \supseteq trans_n$);
- ρ_n contains an over-approximations of the call contexts that reach a return of the method n belongs to (*i.e.*, $\rho_n \supseteq returns_n$).

In particular, correctness implies that when analysing a full program, safety of its execution can be checked by verifying that the empty stack is a secure calling context of its `main` entry node:

$$\varepsilon \in \sigma_n \quad \Rightarrow \quad \nexists (s{:}Stacks).n_0 \rhd^* s \ \wedge \ s \nvDash \varphi$$

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:

$$s \in \{\!|s'|\!\}^+ \ \text{ iff } \ \exists (i{:}\mathbb{N}).\, s \in \{\!|s'|\!\}^i$$

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

Property 5.7

$$\begin{array}{lcl} sec_n & = & \bigcap_{i \in \mathbb{N}} \{s \mid \sigma^i(s,n)\} \\ trans_n & = & \bigcup_{i \in \mathbb{N}} \{s \mid \tau^i(s,n)\} \\ returns_n & = & \bigcup_{i \in \mathbb{N}} \{s \mid \rho^i(s,n)\} \end{array}$$

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

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

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.

Lemma 5.8
Let G be a CFG and let (ρ,σ,τ) be a solution to $[\![G]\!]^C$. For all integer $i \in \mathbb{N}$, node n and stack s, the following holds:

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

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:

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

To prove the property for the rank $i{+}1$, we assume $\tau^{i+1}(s,n)$ (resp. ρ^{i+1}), and prove $s \in \tau_n$ (resp. $s \in \rho_n$).

In case n is a `return` node, the proof is immediate by Lemma 2.10 which raises a contradiction (resp. by the ρ_{return} rule which always implies $s \in \rho_n$).

For the two other types of nodes, a proof step will consist in proving the existence of a stack $s{:}n' \in \{\!|s{:}n|\!\}^{1..i+1}$ (resp. $s{:}r \in \{\!|s{:}n|\!\}^{0..i+1}$ where $IS(r) = \texttt{return}$). Depending on the type of nodes, this step will rely on Lemmas 2.11 or 2.13.

We consider each case in turn:

- $IS(n) = \texttt{check}(\gamma)$:
  — τ: According to Lemma 2.11, our assumption on $\tau^{i+1}(s,n)$ implies $s \in \delta_n(concr(\gamma))$. By τ_{check}, it immediatly follows that $s \in \tau_n$.
  — ρ: By assuming $\rho^{i+1}(s,n)$, we assume that $\exists(r{:}NO).IS(r){=}\texttt{return}$ such that $s{:}r$ is reachable from $s{:}n$ in less than $i{+}1$ steps. According to Lemma 2.11, and because n is not a `return` node, this also implies that $\exists(n'{:}NO).n \xrightarrow{TG} n'$ such that $s{:}r \in \{\!|s{:}n'|\!\}^{0..i}$. By induction hypothesis, it follows that $s \in \rho'_n$. Since the same lemma and τ_{check} also implies that $s \in \tau_n$, the $\rho_{transfer}$ rule allows to us to conclude that $s \in \rho_n$.
- $IS(n) = \texttt{call}$:
  — τ: Our assumption on $\tau^{i+1}(s,n)$ implies that we can use Lemma 2.13 in its second alternative (since $\not\exists s{:}n' \in \{\!|s{:}n{:}m|\!\}^{0..i}$). Hence, for some node m such that $n \xrightarrow{CG} m$, $\exists(k{<}i, r{:}NO).IS(r) = \texttt{return} \wedge s{:}n{:}r \in \{\!|s{:}n{:}m|\!\}^{k}$. By induction hypothesis, this implies $s{:}n \in \rho_m$, and by δ definition, $s \in \delta_n\rho_m$. Rule τ_{call} allows use to conclude.

— ρ: Again, we use Lemma 2.13 in its second alternative to prove that $s \in \tau_n$, and that $\exists(n'{:}NO).n \overset{TG}{\rightarrow} n' \wedge s{:}r \in \{\!|s{:}n'|\!\}^{0..i-k-1}$. By induction hypothesis, this gives us $s \in \rho'_n$, and we conclude with rule $\rho_{transfer}$.

□

Lemma 5.9
Let G be a CFG and let (ρ, σ, τ) be a solution to $[\![G]\!]^C$. 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: $\forall(n{:}NO,\ s{:}Stacks,\ j \leqslant i)$.

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

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 type of the n node.

In case n is a `return` node, the proof is immediate since Lemma 2.10 reduce it to the base case.

For the two other types of nodes, a proof step will consist in proving that all stacks $s' \in \{\!|s{:}n|\!\}^{1..i+1}$ are in $concr(\varphi)$. Depending on the type of nodes, this step will rely on Lemmas 2.11 or 2.13.

- $IS(n) =$ `check` According to Lemma 2.11, all stacks from $\{\!|s{:}n|\!\}^{1..i+1}$ belong to $\{\!|s{:}n'|\!\}^{0..i}$ for some n' such that $n' \overset{TG}{\rightarrow} n$. By induction hypothesis, this stacks are in $concr(\varphi)$ if $s \in \sigma'_n$. And this is ensured by rule $\sigma_{transfer}$.
- $IS(n) =$ `call` According to Lemma 2.13, all stacks from $\{\!|s{:}n|\!\}^{1..i+1}$ belong to either $\{\!|s{:}n{:}m|\!\}^{0..i}$ for some m such that $n \overset{CG}{\rightarrow} m$, or to $\{\!|s{:}n'|\!\}^{0..i-k-1}$ for some n' such that $n' \overset{TG}{\rightarrow} n$.
  In the first case, induction hypothesis state that this stacks are in $concr(\varphi)$ if $s{:}n \in \sigma_m$. This is ensured by the σ_{call} rule and δ definition.
  In the second case, $\sigma_{transfer}$ rule and τ_n correction imply that $s \in \sigma'_n$. Hence we can conclude by induction hypothesis.

5.3 Completeness

In the following we prove that our analysis is also complete, meaning that if a stack s is not recognized as being a secure calling stack for a node n ($s \notin \sigma_n$), then there exist some illegal stacks reachable from $s{:}n$, that is $\exists(s'{:}Stacks).s' \in \{\!|s{:}n|\!\}^{*} \wedge s' \not\models \varphi$.

Theorem 5.10
Let G be a CFG and let (ρ, σ, τ) be the smallest solution to $[\![G]\!]^C$. Then, for all nodes $n \in NO$, all secure calling contexts are subsets of σ_n:

$$\forall(n{:}NO, s{:}Stacks).s \in sec_n \quad \Rightarrow \quad s \in \sigma_n$$

In particular, completeness implies that if execution of a full program is safe, then the empty stack is recognized as a secure calling context for its `main` entry node:

$$\varepsilon \in \sigma_n \quad \Leftarrow \quad \nexists(s{:}Stacks).n_0 \rhd^* s \ \wedge \ s \not\models \varphi$$

As in the case of the correctness theorem, the completeness of σ relies on a similar property for τ and ρ. Hence, will prove a more general result, expressed this terms of two lemmas.

Lemma 5.11 states the completeness of the τ and ρ sets when computed as the least solution of a system $[\![G]\!]^C$.

Lemma 5.11
Let G be a CFG and let (ρ, σ, τ) be the smallest solution to $[\![G]\!]^C$. For every node n, stack s, the following holds:

$$\begin{array}{lcl}\exists(i{:}\mathbb{N}).\rho^i(s,n) & \Leftarrow & s \in \rho_n \\ \exists(i{:}\mathbb{N}).\tau^i(s,n) & \Leftarrow & s \in \tau_n\end{array}$$

Proof
The proof is by least fixpoint induction. We first show that the lemma holds for the first iterate in which $\rho_n = \tau_n = \emptyset$ for all nodes n. Then, an inductive step consists in proving that if in an environment the lemma holds, then it will still hold at the next step after application of one of the constraints.

The base case is vacuously verified ($\nexists(s{:}Stacks).s \in \emptyset$). The inductive step is by case analysis on the applied constraints:

- τ_{check} : we assume $s \in \delta_n(concr(\phi))$. By graph wellformedness, $\exists(n'{:}NO).n \overset{TG}{\rightarrow} n'$. Since $s{:}n' \in \{\!|s{:}n'|\!\}^{0..i-1}$, Lemma 2.11 implies that $s{:}n' \in \{\!|s{:}n|\!\}^{0..i}$ and $\tau^i(s,n)$ is verified.
- τ_{call} : we assume some node m such that $n \overset{CG}{\rightarrow} m$ and $s \in \delta_n(\rho_m)$. By δ definition and induction hypothesis, this implies $\rho^i(s{:}n, m)$ ie., $\exists(k<i,\ r : NO).IS(r) =$ `return` $\wedge\ s{:}n{:}r \in \{\!|s{:}n{:}m|\!\}^{k}$. Again, by graph wellformedness, $\exists(n'{:}NO).n \overset{TG}{\rightarrow} n'$. Since $s{:}n' \in \{\!|s{:}n'|\!\}^{0..i-k-1}$, Lemma 2.13 implies that $s{:}n' \in \{\!|s{:}n|\!\}^{0..i}$ and $\tau^i(s,n)$ is verified.
- $\rho_{transfer}$: we assume some node n' such that $n \overset{TG}{\rightarrow} n'$ and $s \in \tau_n \cap \rho'_n$. By induction hypothesis, we have $\tau^i(s,n)$ and $\rho^j(s,n')$. By $\tau^i(s,n)$ definition and Lemma 2.14, $s{:}n' \in \{\!|s{:}n|\!\}^{1..i}$. By the definition of $\rho^j(s,n')$, we have that $\exists s{:}r \in \{\!|s{:}n'|\!\}^{0..j}$ where $IS(r) =$ `return`. Hence, by Property 2.9, $\exists s{:}r \in \{\!|s{:}n|\!\}^{1..i+j}$ and $\rho^{i+j}(s,n)$.
- ρ_{return} : for any i, $\rho^i(s,n)$ is verified.

It remains to argue the admissibility of the predicates. This follows from the fact that the least upper bound of the iterates is the set union of the iterates. Hence, if a stack belongs to the least upper bound it also belongs to one of the iterates; this guarantees the existence of the relevant i. □

Lemma 5.12 states the completeness of the σ set:

Lemma 5.12
Let G be a CFG and let (ρ, σ, τ) be the smallest solution to $[\![G]\!]^C$. For all node n and stack s, the following holds:

$$s \in \sigma_n \;\Leftarrow\; \forall(i{:}\mathbb{N}).\sigma^i(s,n)$$

Proof
The proof is by least fixpoint induction. The property is obviously verified for the first iterate ($\sigma_n = \mathit{Stacks}$). The inductive step is by case analysis on the applied constraint:

- σ_{global}: we assume $s \notin \delta_n(\mathit{concr}(\varphi))$. This obviously implies $\neg\sigma^0(s,n)$, hence the property is verified.
- σ_{call}: we assume $n \overset{CG}{\to} m$ and $s \notin \delta_n(\sigma_m)$. By the definition of δ and the induction hypothesis, this means that $\exists(k{:}\mathbb{N}).\neg\sigma^k(s{:}n,m)$ (or $\exists(s'{:}\mathit{Stacks},\ k{:}\mathbb{N}).s' \in \{\!|s{:}n{:}m|\!\}^{0..k} \wedge s' \nvDash \varphi$). By Lemma 2.13, this also implies $\exists(i{:}\mathbb{N}).\neg\sigma^i(s,n)$ ($i = k+1$ for instance), and the property is verified.
- $\sigma_{transfer}$: we assume $n \overset{TG}{\to} n'$, $s \in \tau_n$ and $s \notin \sigma'_n$. By τ completeness, we have $\exists(j{:}\mathbb{N}).\tau^j(s,n)$. By Lemma 2.14, it gives us $s{:}n' \in \{\!|s{:}n'|\!\}^{1..j}$. The induction hypothesis also states that $\exists(k{:}\mathbb{N}).\neg\sigma^k(s,n')$, hence, with property 2.9, we prove $\exists(i{:}\mathbb{N}).\neg\sigma^i(s,n)$ ($i = j+k$ for instance).

Admissibility follows from the fact that the limit of the iterates is the set intersection of the iterates; hence, if a stack belong to all iterates it belongs to their limit. □

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. To obtain a system of constraints whose least solution is computable, we interpret concrete, extensional sets of stacks by an intensional representation based on $\mathcal{LTL}$ formulae. This transformation does not incur any loss of precision and can be understood as a correct and complete abstract interpretation of the concrete constraints. In the following section we will then show how to solve these constraints by a least fixpoint calculation over an abstract domain built from $\mathcal{LTL}$ formulae.

6.1 Abstract constraints

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). The abstraction of the subset ordering is the ordering $\Rightarrow$ of $\mathcal{LTL}$ formulae, satisfying $\phi \Rightarrow \phi'$ if and only if $\mathit{concr}(\phi) \subseteq \mathit{concr}(\phi')$ (Lichtenstein & Pnueli, 2000). We recast the set-based contraints from the analysis in Figure 2 as constraints over $\mathcal{LTL}$ formulae, by replacing the set-based operators ($\subseteq, \cup, \cap, \bar{\cdot}, \delta_n$) with their abstract counterpart ($\Rightarrow, \vee, \wedge, \neg, \delta_n^{\#}$). The result is a similar syntax-directed constraint generation scheme, which is shown in Figure 3.

$$\tau^{\#}_{check}\ \frac{IS(n) = \texttt{check}(\gamma)}{\tau^{\#}_n \Leftarrow \delta^{\#}_n(\gamma)} \qquad \tau^{\#}_{call}\ \frac{n \overset{CG}{\longrightarrow} m}{\tau^{\#}_n \Leftarrow \delta^{\#}_n(\rho^{\#}_m)}$$

$$\rho^{\#}_{transfer}\ \frac{n \overset{TG}{\longrightarrow} n'}{\rho^{\#}_n \Leftarrow (\tau^{\#}_n \wedge \rho^{\#}_{n'})} \qquad \rho^{\#}_{return}\ \frac{IS(n) = \texttt{return}}{\rho^{\#}_n \Leftarrow \mathbf{True}}$$

$$\sigma^{\#}_{global}\ \frac{}{\sigma^{\#}_n \Rightarrow \delta^{\#}_n(\varphi)} \qquad \sigma^{\#}_{call}\ \frac{n \overset{CG}{\longrightarrow} m}{\sigma^{\#}_n \Rightarrow \delta^{\#}_n(\sigma^{\#}_m)}$$

$$\sigma^{\#}_{transfer}\ \frac{n \overset{TG}{\longrightarrow} n'}{\sigma^{\#}_n \Rightarrow (\neg\tau^{\#}_n \vee \sigma^{\#}_{n'})}$$

Fig. 3. Constraint specification of $\tau^{\#}$, $\rho^{\#}$, $\sigma^{\#}$.

Among the abstract operators, $\delta^{\#}$ is of particular interest. Intuitively, given a formula that denotes the calling contexts, it computes the weakest precondition such that the property is satisfied before a call. The operator is the temporal logic version of the *Brzozowski derivative* (Brzozowski, 1964) on regular expressions. This operator was defined by Drissi-Kaitouni *et al.* (1988, 1989), who also stated the result equivalent to our Lemma 6.2. The proof corresponding to the proof of our Lemma 6.2 can be found in the technical report by Drissi-Kaitouni & Jard (1988). We here give a proof in our notation of the most complicated cases.

Definition 6.1
Let $n \in NO$ be a node, with $AT(n)$ the set of attributes of node n (*cf.* Definition 2.1). The abstract *weakest calling context operator* $\delta^{\#}_n : \mathcal{LTL} \rightarrow \mathcal{LTL}$ is inductively defined over the structure of the formula.

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

A similar operator for LTL on infinite words is used by Vardi to construct the alternating Büchi automaton of a formula (Vardi, 1996). The main difference here is the $\neg\varepsilon$ introduced here by formulae of type $\mathbf{X}_{\exists}\phi$. This is necessary in our finite stacks domain, to ensure that the empty stack will not satisfy $\delta^{\#}_n(\mathbf{X}_{\exists}\phi)$ even if it satisfies ϕ.

The following lemma states that the $\delta^{\#}$ operator calculates the most precise precondition for a property ϕ to hold at a given node n.

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

$$s \models \delta^{\#}_n(\phi) \iff s{:}n \models \phi$$

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 cases for the $\mathbf{X}_\exists$ and $\mathbf{U}_\exists$ operators.

- We prove that $s{:}n \vDash \mathbf{X}_\exists\phi$ iff $s \vDash \phi \wedge \neg\varepsilon$. By definition, $\varepsilon \equiv \neg(\mathbf{True}\mathbf{U}_\exists\mathbf{True})$, which by the semantics of $\mathbf{U}_\exists$ gives that $s \vDash \varepsilon$ iff $|s|=0$. Thus, $s \vDash \phi \wedge \neg\varepsilon$ iff $s \vDash \phi$ and $|s|>0$ (*i.e.*, $|s{:}n|>1$). This is exactly the semantics the $\mathbf{X}_\exists$ operator.
- We prove that $s{:}n \vDash \phi_1\mathbf{U}_\exists\phi_2$ iff $s \vDash \delta_n^\#(\phi_1\mathbf{U}_\exists\phi_2)$. By definition of $\mathbf{U}_\exists$, we have: $s{:}n \vDash \phi_1\mathbf{U}_\exists\phi_2$ iff $\exists(k<|s{:}n|).\ s{:}n^k \vDash \phi_2$ and $\forall(i<k).\ s{:}n^i \vDash \phi_1$. We split this formula depending on whether k and i are strictly positive or null. We simplify further this expression by using the fact that $|s{:}n|-1 = |s|$ and that $s{:}n^{j+1} = s^j$ to get the following expression:

$$s{:}n \vDash \phi_1\mathbf{U}_\exists\phi_2 \text{ iff } (s{:}n \vDash \phi_2) \text{ or } (s{:}n \vDash \phi_1 \text{ and } s \vDash \phi_1\mathbf{U}_\exists\phi_2)$$

  By induction hypothesis, we have $s{:}n \vDash \phi_2$ iff $s \vDash \delta_n^\#(\phi_2)$ and $s{:}n \vDash \delta_n^\#(\phi_1)$. Moreover, propositional operators $\vee$/ or and $\wedge$/ and distribute with respect to $\vDash$. As a result, we have $s{:}n \vDash \phi_1\mathbf{U}_\exists\phi_2$ iff $s \vDash \delta_n^\#(\phi_2) \vee (\delta_n^\#(\phi_1) \wedge \phi_1\mathbf{U}_\exists\phi_2)$ By definition of $\delta^\#$, this concludes the proof.

□

6.2 *Exactness of the abstraction*

Let $[\![G]\!]^\#$ denote the abstraction of $[\![G]\!]^C$. In this section, we prove that the least solution to $[\![G]\!]^\#$ exactly models the secure calling contexts as defined implicitly by $[\![G]\!]^C$.

Theorem 6.3
Let $(\sigma^\#, \rho^\#, \tau^\#)$ be the least abstract solutions to $[\![G]\!]^\#$, then the following holds: $\forall(n{:}NO\ , s{:}Stacks)$.

$$\begin{array}{lll} s \vDash \sigma_n^\# & \text{iff} & s \in sec_n \\ s \vDash \tau_n^\# & \text{iff} & s \in trans_n \\ s \vDash \rho_n^\# & \text{iff} & s \in returns_n \end{array}$$

We stress that the theorem does not guarantee that such a solution exists. Here, we only argue that the abstract iteration sequence of LTL formulae exactly corresponds to the set-based iteration sequence. Existence will only be proved in section 7 where we show that the abstract iteration sequence of LTL formulae stabilises at a solution after a finite number of iteration.

One consequence of the theorem is that execution of a full program is safe if and only if the empty stack satifies the abstract secure calling context of its `main` entry node:

$$\varepsilon \vDash \sigma_n^\# \quad \Leftrightarrow \quad \nexists(s{:}Stacks).n_0 \rhd^* s\ \wedge\ s \nvDash \varphi$$

Proof of this theorem relies on the exactness of the least solution of the concrete constraints system $[\![G]\!]^C$. In addition, there is by construction a one-to-one correspondence between the contraints in $[\![G]\!]^C$ and $[\![G]\!]^\#$. Formally, the concretisation *concr* of the abstract least solution is the least solution of the concrete constraints (Lemma 6.4).

Lemma 6.4
$\forall(n{:}NO).$

$$concr(\sigma_n^\#) = \sigma_n \quad \wedge \quad concr(\rho_n^\#) = \rho_n \quad \wedge \quad concr(\tau_n^\#) = \tau_n$$

Proof
To prove this result, we show that computing in the abstract domain does not incur a loss of precision. To do so, we show that the least $\mathcal{LTL}$ formula exactly models the least set of stacks and that each abstract operator exactly models its concrete counterpart.

- We have defined $concr(\mathbf{True}) = Stacks$. Furthermore, it is a straightforward consequence of the $\mathcal{LTL}$ semantics that $concr(\mathbf{False}) = \emptyset$. This concludes the first part of the proof.
- Formally, the exactness of abstract operators is stated by showing that concretisation commutes with respect to abstract/concrete operators. By definition of $\mathcal{LTL}$ semantics, this property holds for propositional connectors and we have :

$$\begin{array}{lcl} concr(\phi \wedge \phi') & = & concr(\phi) \cap concr(\phi') \\ concr(\phi \vee \phi') & = & concr(\phi) \cup concr(\phi') \\ concr(\neg\phi) & = & \overline{concr(\phi)} \end{array}$$

  To finish the proof, we have to show a similar result for the δ operator. This is a direct corollary of Lemma 6.2. Indeed, by this lemma, we have $s \models \delta_n^\#(\phi) \iff s{:}n \models \phi$. By reinterpreting the previous equivalence in terms sets, we have that $concr(\delta_n^\#(\phi)) = \{s \mid s{:}n \in concr(\phi)\}$. By definition of the concrete operator δ, we obtain the desired equality:

$$concr(\delta_n^\#(\phi)) = \delta_n(concr(\phi))$$

Hence, each concrete computation is exactly modelled by an abstract one and Lemma 6.4 holds. □

Example 6.5
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 security invariant formula $Crit \Rightarrow \mathbf{F}(Accountant) \wedge \mathbf{F}(Manager)$, and γ for the *check* formula $\mathbf{F}(Manager) \wedge \mathbf{F}(Accountant)$.

$$\begin{array}{llll}
\tau_{n_0}^\# \Leftarrow \delta_{n_0}^\#(\rho_{n_3}^\#) & \tau_{n_1}^\# \Leftarrow \delta_{n_1}^\#(\rho_{n_0}^\#) & \tau_{n_1}^\# \Leftarrow \delta_{n_1}^\#(\rho_{n_3}^\#) & \tau_{n_3}^\# \Leftarrow \delta_{n_3}^\#(\gamma) \\
\rho_{n_0}^\# \Leftarrow \tau_{n_0}^\# \wedge \rho_{n_1}^\# & \rho_{n_1}^\# \Leftarrow \tau_{n_1}^\# \wedge \rho_{n_2}^\# & \rho_{n_2}^\# \Leftarrow \mathbf{True} & \rho_{n_3}^\# \Leftarrow \tau_{n_3}^\# \wedge \rho_{n_4}^\# \\
\rho_{n_4}^\# \Leftarrow \mathbf{True} & \sigma_{n_0}^\# \Rightarrow \delta_{n_0}^\#(\varphi) & \sigma_{n_0}^\# \Rightarrow \delta_{n_0}^\#(\sigma_{n_3}^\#) & \sigma_{n_0}^\# \Rightarrow (\neg\tau_{n_0}^\# \vee \sigma_{n_1}^\#) \\
\sigma_{n_1}^\# \Rightarrow \delta_{n_1}^\#(\varphi) & \sigma_{n_1}^\# \Rightarrow \delta_{n_1}^\#(\sigma_{n_0}^\#) & \sigma_{n_1}^\# \Rightarrow \delta_{n_1}^\#(\sigma_{n_3}^\#) & \sigma_{n_1}^\# \Rightarrow (\neg\tau_{n_1}^\# \vee \sigma_{n_2}^\#) \\
\sigma_{n_2}^\# \Rightarrow \delta_{n_2}^\#(\varphi) & \sigma_{n_3}^\# \Rightarrow \delta_{n_3}^\#(\varphi) & \sigma_{n_3}^\# \Rightarrow (\neg\tau_{n_3}^\# \vee \sigma_{n_4}^\#) & \sigma_{n_4}^\# \Rightarrow \delta_{n_4}^\#(\varphi)
\end{array}$$

A solution to this constraints system is shown in the next section.

7 Iterative constraint solving

In this section we show how to compute a solution to the set of constraints $[\![G]\!]^{\#}$ obtained by analysing the control flow graph G. Our resolution scheme rephrases constraint solving in terms of a least fixed point problem (Cousot & Cousot, 1995; Nielson *et al.*, 1999). For a complete lattice $(D, \sqsubseteq, \sqcup, \sqcap)$, a set of constraints C induces an *iterator* $F : (Var \rightarrow D) \rightarrow (Var \rightarrow D)$ obtained 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 (Cousot & Cousot, 1977b) since *if* the ascending Kleene chain $\bot, F(\bot), F^2(\bot), \ldots$ stabilises at an element e then e is the least fixed point of F.

However, a potential problems arise when trying to solve the constraints by an iterative fixed point calculation. The abstract domain of $\mathcal{LTL}$ formulae contains infinite chains which means that termination of the iteration is not guaranteed. We address this problem together by solving the constraints in an abstract domain of $\mathcal{LTL}$ formulae where the implication relation between temporal formulae has been replaced by the weaker and less complex propositional implication. Formally, we work within the boolean lattice obtained by propositional completion of the set of temporal formulae occurring in the set of constraints. The fact that this approach addresses the issue of infinite chains in the abstract domain has to be proved in detail. In this section will prove that all iteration sequences arising during the verification of a particular property will be stationary after a finite number of iterations. One essential observation (Lemma 7.3) 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.

Definition 7.1

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

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

Definition 7.2

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

The following lemma is the important closure property of $Prop(Sub(\phi))$, *viz.*, that applying $\delta^{\#}$ to a formula in $Prop(Sub(\phi))$ results in a property that still belongs to the set $Prop(Sub(\phi))$.

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 $AT(n)$. 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 $Prop$. $\square$

The Sub (resp. $\delta^{\#}$) operator extends to sets Φ of $\mathcal{LTL}$ formulae in the obvious element-wise fashion, by stipulating that

$$Sub(\Phi) = \bigcup_{\phi \in \Phi} Sub(\phi) \quad \text{resp.} \quad \delta^{\#}_n(\Phi) = \bigcup_{\phi \in \Phi} \delta^{\#}_n(\phi).$$

The following Corollary is an immediate consequence of Lemma 7.3.

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

$$\delta^{\#}_n(Prop(Sub(\Phi))) \subseteq Prop(Sub(\Phi))$$

To obtain the termination result for the iterative fixed point computation, we then instantiate these results to the constraint systems arising from the analysis of a CFG G.

Definition 7.5
Given a global property φ and a CFG G, let $\{\gamma_1, \ldots, \gamma_n\}$ be the set of $\mathcal{LTL}$ formulae occurring in the check nodes of G. Define the set

$$Const = Const(G, \varphi) \equiv \{\gamma_1, \ldots, \gamma_n\} \cup \{\varphi\}$$

Theorem 7.6
Given CFG G and $\mathcal{LTL}$ formula φ, let $F : (Var \to \mathcal{LTL}) \to (Var \to \mathcal{LTL})$ defined by $F(\rho)(x_i) = E_i(\rho)$ be the iterator obtained from the constraint system $[\![G]\!]^{\#}$. 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 within the lattice $Prop(Sub(Const))$

Proof
By construction, a lattice of the form $Prop(Sub(Const))$ is closed under the standard logical operations. This, together with Lemma 7.4 implies that the fixed point iteration induced by the 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 observe that $Prop(Sub(Const))$ is finite because $Const$ and hence $Sub(Const))$ is finite. This and the monotonicity of operators $\delta_n^{\#}, \wedge, \vee$ (which ensures monotonicity of F) implies that the fixed point iteration stabilises. □

Example 7.7
In section 3.1 we argued that the Java stack inspection mechanism could be formalised via the predicate $JDK(perm)$ defined by $perm\ \mathbf{U}_{\forall}\ (perm \wedge Priv)$. 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 $AT(n) = \{p\}$. 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 $\delta_n^{\#}(JDK(p))$:

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

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 7.8
We now return to the analysis of the code in Figure 1. The code gave rise to the fixed point system in Example 6.5, the solution of which is given below. The solution is computed by fixed point iteration over the domain

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

Recall that $\mathbf{F}(\phi)$ stands for $\mathbf{True}\mathrm{U}_\exists\phi$. 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 **F**.

$$\begin{aligned}
&\tau^\#_{n_0} = \tau^\#_{n_1} = \tau^\#_{n_3} = \mathbf{F}(Accountant)\\
&\tau^\#_{n_4} = \tau^\#_{n_2} = \mathbf{False}\\
&\rho^\#_{n_0} = \rho^\#_{n_1} = \rho^\#_{n_3} = \mathbf{F}(Accountant)\\
&\rho^\#_{n_2} = \rho^\#_{n_4} = \mathbf{True}\\
&\sigma^\#_{n_0} = \sigma^\#_{n1} = \mathbf{F}(Accountant) \Rightarrow \mathbf{F}(Manager)\\
&\sigma^\#_{n_2} = \mathbf{F}(Accountant) \wedge \mathbf{F}(Manager)\\
&\sigma^\#_{n_3} = \sigma^\#_{n_4} = \mathbf{True}
\end{aligned}$$

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 $s \models \neg\mathbf{F}(Accountant)$ (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 $s \models \mathbf{F}(Manager)$ (i.e. there is a node with the *Manager* attribute on the call stack) then the dynamic stack inspection ensures that $s \models \mathbf{F}(Accountant)$. In this case, the critical action is executed in a secure fashion.

8 An example verification

The analysis presented in the preceding sections has several uses in the verification of stack-inspecting programs. In this section, we illustrate:

- how it serves to infer interfaces for library routines (this was also illustrated in the previous section),
- how this interface is integrated into the analysis of code using this library, and
- the type of interfaces inferred for correct and for malicious code.

We will use an example from Jensen *et al.* (1999) of an idealized bank account.

```
public class ControlledVar {
  private int var;
  void write(int new) {
    AccessController.checkPermission(Write);
    var = new;
  }
  int read() {
    AccessController.checkPermission(Read);
    return var;
  }
}
```

Fig. 4. The system code (*System* domain).

```
public class AccountMan {
  private ControlledVar balance;
  public boolean canpay?(int amount) {
    AccessController.checkPermission(Canpay);
    boolean res = false;
    try {
      AccessController.beginPrivileged();
      res = balance.read() > amount;
    } finally {
      AccessController.endPrivileged();
    }
    return res;
  }
  public void debit(int amount) {
    AccessController.checkPermission(Debit);
    if (this.canpay?(amount)) {
      try {
        AccessController.beginPrivileged();
        balance.write(balance.read() - amount);
      } finally {
        AccessController.endPrivileged();
      }
    } else ...
  }
}
```

Fig. 5. The account manager code (*Provider* domain).

8.1 The library code

Two protection domains (corresponding to two principals) are involved. They are called *System* and *Provider*:

- We assume the system (Figure 4) supplies code to implement a controlled integer variable holding the balance of the account. This variable has entry points for `read` and `write` operations, protected with a `check` for the respective permissions.
- Using the controlled variable, the service provider builds an account manager (Figure 5) with a `debit` transaction and a boolean query method `canpay?`.

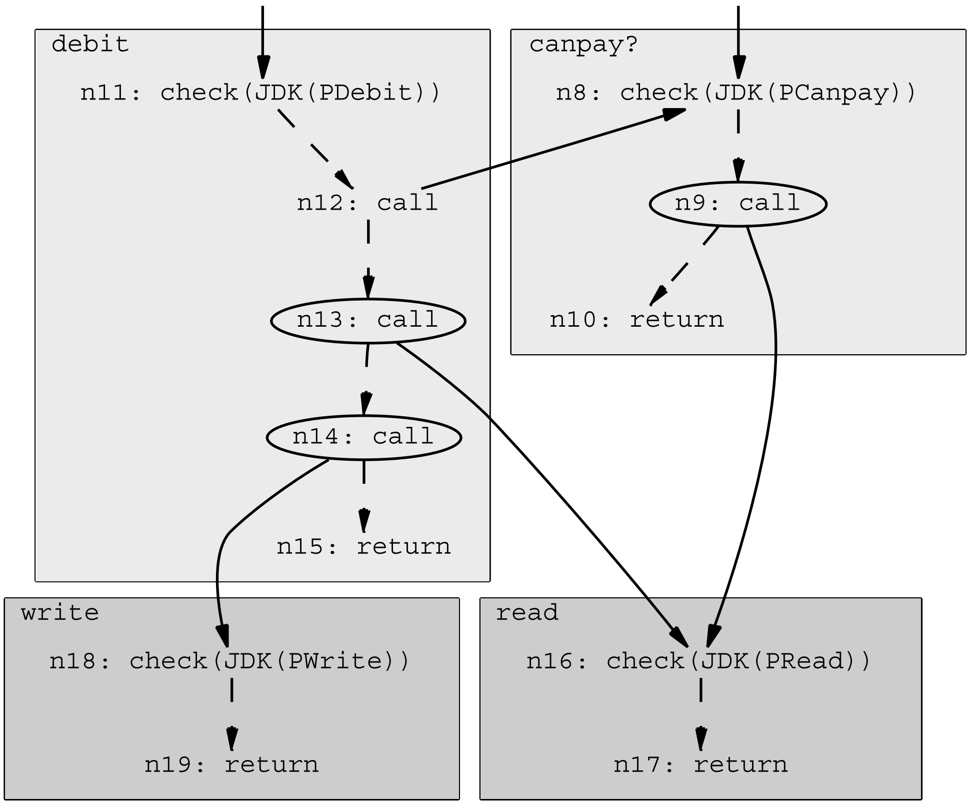


Fig. 6. The derived graph G_{EC}.

For this to work, we assume that the provider code is granted the `Write`, `Read`, `Debit`, and `Canpay` permissions. The `debit` and `canpay?` methods call the methods `read` and `write` in privileged mode because they can be called by clients that do not have the permission to call `read` and `write` directly (i.e. which are not granted the `Read` and `Write` permissions).

8.2 Translation of the example into our model

From the code for the above example, we derive the graph G_{EC} as outlined in section 2.1. The result is shown in Figure 6. Although the methods have no representation in the graph, we have clustered the nodes in boxes according to their method of origin. Furthermore, boxes are coloured according to the protection domains to which its nodes belong. Like the example in section 2, the dashed edges are transfer edges (TG), while the solid edges are call edges (CG), obtained through a class analysis. The three encircled nodes correspond to code executed as privileged.

The two protection domains partition the set of nodes as follows:

$$\begin{array}{lcl} \textit{Provider} & = & n_8, \ldots, n_{15} \\ \textit{System} & = & n_{16},\ n_{17},\ n_{18},\ n_{19} \end{array}$$

Attributes of each node in the graph G_{EC} include the permissions corresponding to its protection domain, plus the attribute *Priv* if it appears within a privileged section. Furthermore, to describe that a node belongs to a Java method `foo`, we introduce the attribute E_{foo} which is given to each node in the method `foo`.

The attributes associated to each protection domain are the following:

$$\begin{array}{lcl} D_{Provider} & = & \{P_{Read}, P_{Write}\} \\ D_{System} & = & \{P_{Debit}, P_{Canpay}, P_{Read}, P_{Write}\} \end{array}$$

In addition, the nodes n_9, n_{13}, and n_{14} have the attribute *Priv*.

8.3 Verification of security properties

As a global statement about the security of the system, we state that all the calls leading to a modification of the balance must possess the P_{Debit} permission and all the calls leading to disclosure of the balance must possess the P_{Canpay} permission. This property is expressed as follows in our language:

$$\varphi = \bigwedge \left\{ \begin{array}{l} E_{Write} \Rightarrow \mathbf{G}(P_{Debit}) \\ E_{Read} \Rightarrow \mathbf{G}(P_{Canpay}) \end{array} \right.$$

We apply our analysis in two steps, corresponding to the two classes. The results of the first analysis (`ControlledVar`) are reused to analyse the whole system composed of `ControlledVar` and `AccountMan`.

During such an incremental analysis, we do not need the full solution of an analysed library. Constraints for a given node only reference $\tau^{\#}$, $\rho^{\#}$ or $\sigma^{\#}$-properties of the nodes that it is directly linked to (by transfer or call edges), so the only properties of interest are those of the node to which a method can link (i.e. the entry points). Furthermore, the $\tau^{\#}$ property is internal to a method and does not have to be exported. The remaining $\sigma^{\#}$ and $\rho^{\#}$ properties of the entry points of the methods of a library constitute its security interface.

Analysis of `read` and `write` The secure calling contexts σ deduced by analysing `read` and `write` conforms to what would intuitively be expected: because the security invariant requires that every execution stack satisfy $E_{Write} \Rightarrow \mathbf{G}(P_{Debit})$ (resp. $E_{Read} \Rightarrow \mathbf{G}(P_{Canpay})$), and because E_{Write} (resp. E_{Read}) is satisfied by every node of the method `write` (resp. `read`), the analysis concludes that calling contexts for node n_{18} (resp. n_{16}) are safe only if they satisfy $\mathbf{G}(P_{Debit})$ (resp. $\mathbf{G}(P_{Canpay})$). Furthermore, because of the `check` nodes which may cut some executions, `return` nodes are only reachable from calling contexts satisfying the right *JDK* property.

$$\begin{array}{lcl} \sigma^{\#}_{n_{16}} & = & \mathbf{G}(P_{Canpay}) \\ \sigma^{\#}_{n_{18}} & = & \mathbf{G}(P_{Debit}) \\ \rho^{\#}_{n_{16}} & = & JDK(P_{Read}) \\ \rho^{\#}_{n_{18}} & = & JDK(P_{Write}) \end{array}$$

Analysis of `debit` and `credit` The second part of this analysis is done by introducing the above results in the constraints system. This is done by adding the four equations from above to the set of constraints generated by the analysis of `debit` and `credit`.

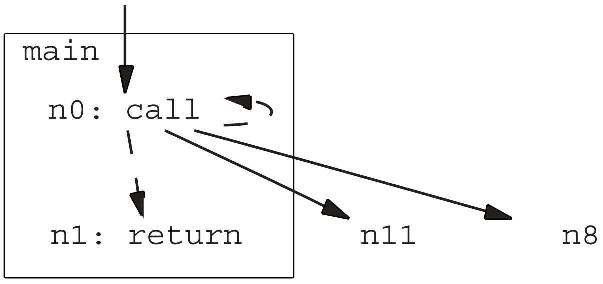


Fig. 7. Simulating a client application.

The results we obtain for the entry points of `debit` and `credit` are the following:

$$\begin{aligned}
\sigma^{\#}_{n_8} &= \mathbf{G}(P_{Canpay}) \vee \neg JDK(P_{Canpay})\\
\sigma^{\#}_{n_{11}} &= (\mathbf{G}(P_{Canpay}) \wedge \mathbf{G}(P_{Debit})) \vee \neg JDK(P_{Canpay}) \vee \neg JDK(P_{Debit})\\
\rho^{\#}_{n_8} &= JDK(P_{Canpay})\\
\rho^{\#}_{n_{11}} &= JDK(P_{Canpay}) \wedge JDK(P_{Debit})
\end{aligned}$$

We examine $\sigma^{\#}_{n_8}$ in more detail (similar reasoning applies to $\sigma^{\#}_{n_{11}}$). Recalling the *JDK* definition, $\sigma^{\#}_{n_{16}}$ can be rewritten as:

$$\begin{aligned}
\sigma^{\#}_{n_8} &= \mathbf{G}(P_{Canpay}) \vee \neg(\mathbf{G}(P_{Canpay})) \vee (P_{Canpay}\mathbf{U}_{\exists}(P_{Canpay} \wedge Priv))\\
&= \mathbf{G}(P_{Canpay}) \vee \neg(P_{Canpay}\mathbf{U}_{\exists}(P_{Canpay} \wedge Priv))
\end{aligned}$$

A first observation we can make is that this precondition for a calling context to be safe is more permissive than the one for `read` ($\sigma^{\#}_{n_{16}}$). That means that the additional `check` nodes we have introduce to protect the critical code help cutting illegal executions.

The programmer of the library may have expected a better result, e.g. a **True** precondition meaning that any calling context would be safe. However, this is not possible to achieve with the security invariant we have required and the rigid structure of the Java stack inspection checks. Indeed, the stack inspection cannot prevent callers from acquiring permissions they do not have through calls to privileged code which have the required permissions. Hence, if we want all callers to have a given permission, this condition will remain in the interface.

This precondition also offer a way to reason about the permissions to give to the applications that will use it. A trusted application should be granted both P_{Debit} and P_{Canpay} permissions on its whole code, so that all the calling contexts that will reach entry points of the library will verify $\mathbf{G}(P_{Debit}) \wedge \mathbf{G}(P_{Canpay})$, and hence the security preconditions of the library. Of course, this permission should be refused to any other application, so that they don't have access to our critical methods.

Analysis of an application To illustrate the above discussion about permissions and privileged code, we reuse previous results in the analysis of the small application illustrated in Figure 7. Its entry point (the program `main`) is node n_0. We apply our analysis with various hypothesis:

- the application is trusted, and granted permissions P_{Debit} and P_{Canpay}. Analysis result gives us

$$\sigma^{\#}_{n_0} = (\mathbf{G}(P_{Canpay}) \wedge \mathbf{G}(P_{Debit})) \vee \neg JDK(P_{Canpay}) \vee \neg JDK(P_{Debit}).$$

  We can easily verify that $\varepsilon \models \sigma^{\#}_{n_0}$, and hence that any execution starting on this `main` method is safe.
- the application is untrusted, and has no granted permissions. The result of the analysis is $\sigma^{\#}_{n_0} = \mathbf{True}$, hence execution is obviously safe (execution will be cut by both of the `check` nodes n_8 or n_{11}.
- the application is trusted, and granted permissions P_{Debit} and P_{Canpay}, but the `call` node n_0 is marked as privileged. The result of the analysis gives us $\sigma^{\#}_{n_0} = \mathbf{G}(P_{Canpay}) \wedge \mathbf{G}(P_{Debit})$. We conclude that it is still safe to execute this code starting from n_0, but if this method can be called from some hostile code (even code without any permissions) then our security invariant is violated.

9 Related work

The concept of stack inspection has been formalised in various ways. Wallach & (1998) 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 & Gordon (2002) 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 most general model of stack inspection has been proposed by Esparza *et al.* (2001). In this model, pushdown automata are extended with transition rules that depend on the content of the pushdown stack. This model has the advantage of integrating exceptions in a straightforward manner, and has allowed to obtain precise complexity results for the model checking problem for such automata. Our semantics in section 2.2 could be cast in this model but we have opted for a lighter operational semantics presentation that leads to rather simple proofs.

The present work builds on the verification techniques developed by Besson, Jensen and others (Jensen *et al.*, 1999; Besson *et al.*, 2001), 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 by Besson, Jensen and others 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.* (2002) 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 (2000) 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. See Bauer *et al.* (2002) for a general class of properties that can be enforced. Erlingsson & Schneider (2000), Colcombet & Fradet (2000) and Walker (2000) all 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.

Analysing and verifying stack-inspecting code has been the object of a number of recent studies. Skalka & Smith (2000) 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 *et al.* (2001) 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 (Wallach, 1999) to higher-order functions. Higuchi & Ohori (2003) present a similar type system for the Java Virtual Machine byte code language. Bartoletti *et al.* (2001) 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 linear temporal logic.

Our work focuses on control flow properties. Banerjee & Naumann (2003) study the complementary problem of how stack inspection can be used to guarantee information flow properties relative to a set of security levels, e.g. to ensure that data classified as confidential does not flow into publicly accessible variables. To this end they have devised a type system that assigns types of the form $L_1 \xrightarrow{P} L_2$ to methods. The intended meaning of such a type is that if the method is called with data with security level L_1 and from a method that *cannot* enable the permissions P then the result will have security level at most L_2. Preliminary work by Blanc *et al.* (2002) has similar goals, but is placed in the setting of the intermediate language of Microsoft's Common Language Runtime. In this work a control-flow analysis extracts a control-flow graph on which properties on the control-flow graph can be checked. Properties of interest include verifying that every granting of permissions serves a purpose (may affect the computation) and that all sensitive operations are preceded by a proper access control. Finally, the work by Koved *et al.* (2002) deals with the somewhat dual problem of analysing a piece of Java code in order to infer the *set of least privileges* in terms of access rights requirements that a code requires to execute without security exceptions. This type of analysis seems particular useful for providing a developer with a means of understanding and refining an access control security policy for an application.

Our security properties are global invariants of the trace of call-stack. Abadi & Fournet (2003) have considered another security model in which access control is based on the entire execution history and not just the current call stack. Consequently, they can enforce more refined trace properties. In particular, the "Chinese Wall"-style properties (Brewer & Nash, 1989) in which certain accesses are authorised only if other access have not been performed earlier on in the execution. These properties are beyond what we can verify with our framewrok since there might be no trace of these earlier accesses on the control stack.

10 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 (Skalka & Smith, 2000; Pottier *et al.*, 2001) that are concerned with inferring the permissions required for all stack inspections to succeed. The verification works for closed code libraries for which it infers pre-conditions under which execution of a library method is secure.

The analysis has been proved correct with respect to a formal semantics. We have furthermore developed the theory necessary for implementing the analysis by fixed-point iteration over an abstract domain built of temporal properties. While the construction $Prop(A)$ of the boolean completion of a set of atoms A of temporal formulae has been studied before (Drissi-Kaitouni & Jard, 1988), it is to the best of our knowledge the first time that this approach has been used to build an abstract domain of temporal properites and to argue the termination of an iteration-based program verifier.

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 size of the constraint system $[\![G]\!]^{\#}$ (and therefore the number of expressions to be evaluated within one iteration) is linear in the size of the control flow graph G. On the other hand, the number of iterations is only bounded by the height of the lattice $Prop(Sub(\gamma_1,\ldots,\gamma_n,\varphi))$ where $\gamma_1,\ldots,\gamma_n$ are the different $\mathcal{LTL}$ formulae used in the `check` nodes and φ is the global property. The height of this lattice is exponential in the size $\mid \gamma_1 \mid + \ldots, + \mid \gamma_n \mid + \mid \varphi \mid$ of the formulae. However, in general we would expect to have

$$\mid \gamma_1 \mid + \ldots, + \mid \gamma_n \mid + \mid \varphi \mid \ll \mid G \mid.$$

because the global security property φ is fixed and that there are generally few different check instructions compared to the number of call and return instructions in a library routine.

The most challenging further step is to extend the analysis to arbitrary *fragments* of control flow graphs in order to deal with software components in which methods make calls to virtual methods that are unavailable 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. This would firstly require a modular Control Flow Analysis, as the one presented by Besson & Jensen (2003). Next, it will be necessary 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. The theoretical foundations of such an extension is proposed in Besson's PhD thesis (Besson, 2002) but the question of how such relational properties can be expressed in a compact manner that enable automated compositional reasoning remains to be settled.

Acknowledgements

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

References

Abadi, M. and Fournet, C. (2003) Access control based on execution history. *Proc. 10th Annual Network and Distributed System Security Symposium (NDSS'03)*, pp. 107–121. Internet Society.

Bacon, D. F. and Sweeney, P. F. (1996) Fast Static Analysis of C++ Virtual Function Calls. *Proc. of OOPSLA'96*, pp. 324–341. *ACM SIGPLAN Notices*, **31**(10).

Banerjee, A. and Naumann, D. A. (2003) Using access control for secure information flow in a java-like language. *Proc. Sixteenth IEEE Computer Security Foundations Workshop (CSFW)*, pp. 155–169. IEEE Press.

Barthe, G., Gurov, D. and Huisman, M. (2002) Compositional verification of secure applet interactions. *Proc. Foundations of Software Engineering (FASE'02)*.

Bartoletti, M., Degano, P. and Ferrari, G. (2001) Static analysis for stack inspection. *Proc. Int. Workshop on Concurrency and Coordination (Concoord 2001)*. Electronic Notes in Theoretical Computer Science vol. 54. Elsevier.

Bauer, L., Ligatti, J. and Walker, D. (2002) *More enforceable security policies*. Technical report TR-649-02, Princeton University.

Besson, F. (2002) *Résolution modulaire d'analyses de programmes : application à la sécurité logicielle*. PhD thesis, University of Rennes I.

Besson, F., Jensen, T., Le Métayer, D. and Thorn, T. (2001) Model ckecking security properties of control flow graphs. *J. Comput. Security*, **9**, 217–250.

Besson, F., de Grenier de Latour, T. and Jensen, T. (2002) Secure calling contexts for stack inspection. *Proc. 4th Int Conf. on Principles and Practice of Declarative Programming (PPDP 2002)*, pp. 76–87. ACM Press.

Besson, F. and Jensen, T. (2003) Modular class analysis with datalog. In: Cousot, R. (ed.), *Proc. 10th Static Analysis Symposium (SAS 2003): Lecture Notes in Computer Science 2694*, pp. 19–36. Springer-Verlag.

Blanc, T., Fournet, C. and Gordon, A. (2002) *From stack inspection to access control: A security analysis for libraries*. Microsoft Research.

Brewer, D. and Nash, M. (1989) The Chinese wall security policy. *Proc. IEEE Symp. on Security and Privacy*, pp. 206–211. IEEE Press.

Brzozowski, J. (1964) Derivatives of regular expressions. *J. ACM*, **11**(4).

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

Cousot, P. and Cousot, R. (1977a) Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. *Proc. 4th ACM Symposium on Principles of Programming Languages*, pp. 238–252. ACM Press.

Cousot, P. and Cousot, R. (1977b) Automatic synthesis of optimal invariant assertions: Mathematical foundations. *Proc. ACM Symposium on Artificial Intelligence and Programming Languages, SIGPLAN Notices*, (8), 1–12.

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

Drissi-Kaitouni, O. and Jard, C. (1988) *Compiling temporal logic specifications into observers*. Technical report INRIA Rapports de Recherche No. 881, Institut National de Recherche en Informatique et en Automatique.

Emerson, E. A. (1990) Temporal and Modal Logic. In: van Leeuwen, J. (ed.), *Handbook of Theoretical Computer Science*, vol. B, pp. 996–1072. Elsevier.

Erlingsson, U. and Schneider, F. (2000) SASI enforcement of security policies: A retrospective. *New Security Paradigms Workshop*. ACM Press.

Esparza, J., ín Kučera, A. and Schwoon, S. (2001) Model-checking LTL with regular valuations for pushdown systems. *Proc. Theoretical Aspects of Computer Science (TACS'2001): Lecture Notes in Computer Science 2215*, pp. 306–339.

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

Gong, L. (1997) Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. *Proc. Usenix Symposium on Internet Technologies and Systems*.

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

Higuchi, T. and Ohori, A. (2003) A static type system for jvm access control. *Proceedings 8th ACM SIGPLAN International Conference on Functional Programming*, pp. 227–237. ACM Press.

Jard, C. and Jeron, T. (1989) On-line model-checking for finite linear temporal logic specifications. In: Sifakis, J. (ed.), *Proc. International Workshop on Automatic Verification Methods for Finite States Systems: Lecture Notes in Computer Science 407*, pp. 189–196. Springer-Verlag.

Jensen, T., Le Métayer, D. and Thorn, T. (1999) Verification of control flow based security properties. *Proc. 20th IEEE Symp. on Security and Privacy*, pp. 89–103. IEEE Computer Society.

Koved, L., Pistoia, M. and Kershenbaum, A. (2002) Access rights analysis for java. *Proceedings 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 359–372. ACM Press.

LaMacchia, B. A., Lange, S., Lyons, M., Martin, R. and Price, K. T. (2002) *.NET framework security*. Addison-Wesley.

Lichtenstein, O. and Pnueli, A. (2000) Propositional temporal logics: Decidability and completeness. *Logic J. IGPL*, **8**(1), 55–85.

Nielson, F., Nielson, H. R. and Hankin, C. L. (1999) *Principles of Program Analysis*. Springer.

Palsberg, J. and Schwartzbach, M. I. (1994) *Object-oriented Type Systems*. John Wiley & Sons.

Pande, H. and Ryder, B. (1996) Data-flow-based virtual function resolution. *Proc. 3rd Static Analysis Symposium (SAS'96): Lecture Notes in Computer Science 1145*. Springer-Verlag.

Pottier, F., Skalka, C. and Smith, S. (2001) A systematic approach to static access control. In: Sands, D. (ed.), *Proc. 10th European Symposium on Programming (ESOP'01): Lecture Notes in Computer Science 2028*, pp. 30–45. Springer-Verlag.

Schneider, F. (2000) Enforceable security policies. *ACM Trans. on Information & System Security*, **3**(1), 30–50.

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

Vardi, M. Y. (1996) An automata-theoretic approach to linear temporal logic. *Lecture Notes in Computer Science 1043*, pp. 238–266. Springer-Verlag.

Walker, D. (2000) A type system for expressive security policies. *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pp. 254–267. ACM Press.

Wallach, D. S. (1999) *A new approach to mobile code security*. PhD thesis, Department of Computer Science, Princeton University.

Wallach, D. S. and Felten, E. W. (1998) Understanding Java stack inspection. *Proc. 19th IEEE Symposium on Security and Privacy*. IEEE Computer Society.