Stack inspection and secure program transformations [*]

Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari
Dipartimento di Informatica, Università di Pisa, Italy
`{bartolet, degano, giangi}@di.unipi.it`

Abstract

The paper focuses on stack inspection, the access control mechanism implemented in Java and the CLR. We introduce a static analysis which safely approximates the set of access rights granted to code at run-time. This analysis provides us with the basis to reduce the run-time overhead of stack inspection, also in combination with other program transformations.

1 Introduction

The growing use of network technologies in distributed computing has made *security* critical in the design, development and distribution of applications. Indeed, both final users and application designers put special emphasis on security issues. For final users, the *awareness* of security mechanisms is crucial for choosing the best network services that match their requirements. Designers wish to control resource usage and access in order to ensure and maintain adequate security levels. Designing and implementing security policies at the programming language level help in handling security [22]. Here, we consider an authorization-based model where a security policy is enforced by inserting appropriate checks in a program.

Java is perhaps the most well-known example of a language with a comprehensive architecture offering linguistic constructs for security. Java applications run components with different levels of trust, e.g. components originated from different, possibly unknown, administration domains. In the Java security model, access control decisions are taken by examining the call stack at run-time. A permission is granted, provided that it belongs to *all* principals on the call stack. The so-called *privileged operations* are an exception. These are allowed to execute any code granted to their principal, regardless of the calling sequence. This access control mechanism is known as *stack inspection.* Beyond Java, other run-time environments (e.g. the .NET Common Language Runtime [18]) adopt stack inspection as basic authorization mechanism.

Stack inspection may be expensive. First, the run-time overhead due to the analysis of stack frames may grow very high. Second, stack inspection deeply affects standard program transformations, such as method inlining and tail call

[*]The first two authors have been partially supported by EU project DEGAS (IST-2001-32072) and MIUR project MEFISTO. The third author has been partially supported by MIUR project NAPOLI and FET project PROFUNDIS (IST-2001-33100)

elimination. These optimizations may in fact alter the structure of the call stack. Hence, understanding the semantics of program transformations in a setting with stack inspection is a research (and technological) challenge.

Our contribution aims at developing semantic-driven mechanisms as an aid to improve efficiency of architectures for language-based security. The problem we face is not merely ensuring that certain program transformations preserve program semantics, but also to guarantee that the optimized code is *security safe*, that is, the optimized code will never violate the security constraints.

We build over control flow analysis [19], the main goal of which is to efficiently obtain computable approximations of the set of values or behaviours that may arise during the execution of a program. These approximations are then used to analyze program properties in a safe way: if a property holds at static time, then it will always hold at run-time. The vice-versa may not be true: the analysis may "err on the safe side".

In this paper, we represent Java programs by *control flow graphs*, an idealized model not tied to any particular language. Control flow graphs feature primitive constructs for method invocations, exceptions, and access control based on stack inspection. These graphs are equipped with a formal operational semantics.

Our first contribution is the definition of a static analysis over control flow graphs, called *Trace Permissions Analysis* (TP analysis for short). Intuitively, this analysis computes, for each program point and each execution reaching that point, the set of permissions granted at run-time. The TP analysis is sound and complete with respect to the operational semantics of our idealized language, i.e. it computes all and only the permissions that are granted at run-time.

However, the execution traces of control flow graphs over-approximate the traces of the Java code they are extracted from. Therefore, if a permission is granted to an execution in a control flow graph, then it will also be granted to the corresponding execution – if any – of the Java code. In this sense we mean that our analysis is safe.

The ability of identifying statically the set of permissions granted at run-time offers a support for security-aware code optimizations. As a first application, we detect and remove the redundant checks in a program, i.e. the checks which always pass. Dead code elimination is a program optimization which detects and removes the code unused or unreached in executions. Security restrictions may cause more fragments of code to become unreachable, e.g. because a security check protecting it is never passed. Our technique permits to discard such dead code in the linking phase. We also cope with *method inlining*, an optimization that replaces a method invocation with a copy of the called method code. In presence of stack inspection, method inlining may break security, because the protection domain of the inlined method is ignored. The TP analysis provides us with the basis to efficiently construct the set of method invocations which can be safely inlined. Our analysis also allows for fast implementations of stack inspection based on an (hyper-) eager strategy.

The paper is organized as follows. The next section surveys the Java security model. Section 3 introduces our program model and its operational semantics; the adequacy of our model is discussed in Section 7. Our static analysis is in Section 4. Section 5 presents the secure program transformations. As an example, we analyze in Section 6 a small e-commerce application. All the proofs, the Java code of our example and the actual algorithm implementing our analysis are in the Appendixes.

2 Background on Java security

The Java language advocates code mobility: bytecode may come from different places, either trusted (e.g. files on the local disk) or untrusted (e.g. applets downloaded across the network). Java provides a customizable environment, called the *sandbox*, in which code is placed: the sandbox prevents untrusted code from performing security-critical operations, according to a fine-grained policy.

The *class loader architecture* brings bytecode from the outside into the JVM, so it is the first line of defense against security attacks, for the following reasons:

- it enforces *name space separation* of the loaded classes. Roughly, classes downloaded by different sources are placed into different name spaces. This mechanism constitutes a main contribution to the security of the Java platform. First, since duplicated class names are forbidden within a name space, a malicious class cannot pretend to be a system class. Second, since classes belonging to different name spaces cannot even detect each other's presence (unless class loaders explicitly allow them to), programmers are relieved from worries about class name collisions.

- it assigns each loaded class to a *protection domain* [12], based on the class signers and location origin. Each protection domain is in turn associated with a set of permissions, according to a global security policy.

A verification step is performed on each loaded class before linking it to the rest of the system. This task is accomplished by the *bytecode verifier*, which statically analyses the bytecode, to guarantee that it satisfies some safety properties, e.g. (1) access to classes, methods and variables is done according to their respective visibility rules (access modifiers); (2) methods are invoked with the correct number and types of arguments.

While both the class loader and the bytecode verifier are mainly concerned with the safety facet of security, the *security manager* more directly addresses the problem of protecting critical resources from leakage and tampering threats. This is done by invoking the *access controller* before the security-critical methods of the protected resource are called.

The security policy is enforced by the access controller each time the method `checkPermission()` is invoked. The access controller decides whether granting access to the protected resource or not by performing *stack inspection* [11].

Stack inspection checks the sequence of method invocations. Each method in the sequence belongs to a class, which in turn belongs to a protection domain. A resource access is granted if and only if all protection domains in the sequence have the required permission. This mechanism is slightly complicated by the presence of *privileged actions*. Technically speaking, a method `M` performs a privileged action `A` by invoking `AccessController.doPrivileged(A)`; this involves invoking method `A.run()` with all the permissions of `M` enabled. This can be seen as marking the method frame of `M` as privileged: stack inspection will then stop as soon as a privileged frame (starting from top) is found.

There are at least two strategies for implementing this algorithm:

- the *eager evaluation* strategy states that the set of effective permissions is updated at each method call (and return).

- in the *lazy evaluation* strategy, the call stack is retrieved and inspected only when access control is performed.

3 The program model

The stack inspection mechanism implements access control by looking solely into the call stack: then, we base our analyses on an abstraction of object-oriented languages that only takes into account security checks, method invocations and returns, and a very basic exception handling mechanism.

More precisely, we model a program as a *control flow graph* (CFG for short) whose nodes represent the activities mentioned above (checks, method invocations and returns) and whose arcs represent the flow of control. We do not define how CFGs are extracted from an actual program. This construction is well understood and algorithms and tools exist for it; see for example [13, 19, 26, 27].

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

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

3.1 Syntax

Let $\mathcal{D}$ be a finite set of protection domains, and $\mathcal{P}$ be a finite set of permissions. CFGs are defined as follows.

Definition 3.1. A CFG $\langle N \cup \{n_\varepsilon\}, E, \mathrm{Priv}, \mathrm{Dom}\rangle$ is an oriented graph, where:

- N is the set of nodes. Each node $n \in N$ is associated with a label $\ell(n)$, describing the control flow primitive it represents. Labels partition nodes in three kinds: `call` nodes, that stand for method invocation, `return` nodes, which represent return from a method, and `check` nodes, which enforce the access control policy. For each $P \in \mathcal{P}$, we can think of a node labelled $\texttt{check}(P)$ as the abstract representation of an `AccessController.checkPermission(P)` instruction in the Java language. The distinguished element $n_\varepsilon \notin N$ plays the technical role of a single, isolated entry point.

- $E \subseteq (N \cup \{n_\varepsilon\}) \times N$ is the set of edges. Edges are partitioned into four sets: *entry edges* $\bullet\!\!\longrightarrow n$, that represent the entry points of a program; *call edges* $n \longrightarrow n'$, which model *interprocedural* flow; *transfer edges* $n \dashrightarrow n'$, which correspond to sequencing; and *catch edges* $n \dashrightarrow_{\lightning} n'$, which correspond to exception handling. The last two kinds of edges represent *intraprocedural* flow. The set of entry edges contains all pairs (n_ε, n) where n is a program entry point. The n_ε element is the source of entry edges, only.

- Priv : $N \to \mathit{Bool}$ tells whether a node enables its privileges or not.
- Dom : $N \to \mathcal{D}$ is a mapping from nodes to protection domains.

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

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

It is convenient to introduce some terminology and notation.

Definition 3.2. The *methods* of a CFG $\langle N, E \rangle$ are the connected components of the graph $\langle N, E' \rangle$, where E' is the set of intraprocedural edges in E, with no orientation. We call $\mu(n)$ the method to which node n belongs. The *entry points* of $\mu(n)$ are defined as:

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

Definition 3.3. The set $\rho(n)$ of return nodes associated to a node n is:

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

Definition 3.4. The set $\xi(n)$ of nodes that may throw an exception catchable by n is defined as the smallest set satisfying:

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

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

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

Our CFGs obey some mild well-formedness constraints to reflect more appropriately some peculiarities of Java-like bytecode, as shown below. Therefore, in what follows we shall always assume that CFGs are well-formed.

Constraint 1. It makes little sense breaking a check in several parts, or merging it with other activities, e.g., with a method call. Therefore, all and only the code of a `checkPermission` method is represented as a *single* check node. Formally, check nodes do not admit outgoing call edges:

$$\ell(n) = \texttt{check}(P) \qquad \Longrightarrow \qquad \neg \exists n' \in N.\, n \longrightarrow n'$$

Constraint 2. Return nodes have no outgoing edges:

$$\ell(n) = \texttt{return} \qquad \Longrightarrow \qquad \neg \exists n' \in N.\, (n, n') \in E$$

Return nodes model exit points of methods, e.g. the `ireturn` and `areturn` instructions. They cannot have outgoing call edges, because this kind of flow only originates from method invocations (e.g. from `invokevirtual` or `invokespecial` instructions). Return nodes cannot have outgoing transfer or catch edges, either, as such edges only represent intraprocedural flow. Then, this constraint is satisfied by any CFG derived from actual Java bytecode.

Constraint 3. Each method has a single entry point, i.e. for each $n \in N$:

$$|\,\varepsilon(\mu(n))\,| \quad = \quad 1$$

Upon method invocation, the virtual machine creates a new stack frame for the called method, and sets the program counter to the first instruction in the new frame: this instruction is just the entry point of the called method. After this constraint, we abbreviate $n \longrightarrow \varepsilon(\mu(m))$ with $n \longrightarrow \mu(m)$.

Constraint 4. Nodes in the same method are in the same protection domain:

$$\mu(n) = \mu(n') \quad \Longrightarrow \quad \mathrm{Dom}(n) = \mathrm{Dom}(n')$$

Indeed, in Java the granularity of a security policy is the class level: classes in the same code source are in the same protection domain.

Constraint 5. Only call nodes can be privileged:

$$\mathrm{Priv}(n) \quad \Longrightarrow \quad \ell(n) = \texttt{call}$$

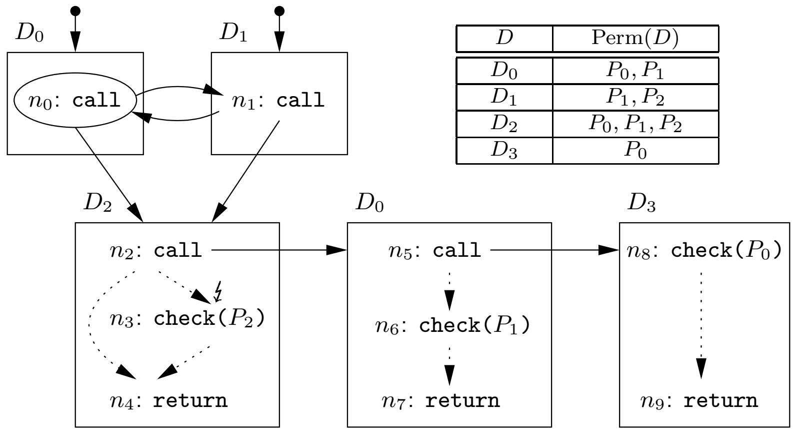
In general, also security checks can occur within privileged actions: however, privileged check nodes make little sense, because it is always possible to determine whether a privileged check will succeed or not, during the construction of the CFG. Similarly, there is no point in enabling returns to be privileged, because a return node will never be on the call stack when inspecting it. As a matter of fact, constraint 5 can easily be removed, at the price of a slightly more involved analysis.

A more detailed discussion on the adequacy of our proposal, along with some hints on possible extensions, is in Section 7.

Example 1. We use the CFG in Fig. 1 as a working example through the paper. Circled calls are privileged. Solid boxes enclose nodes belonging to the same methods, and they are labelled with the protection domain to which the enclosed method belongs. Table 1 displays the methods, their entry points, the return nodes, the nodes that may throw exceptions and those that may propagate them, according to definitions 3.2, 3.3 and 3.4.

n	$\mu(n)$	$\varepsilon(\mu(n))$	$\rho(n)$	$\xi(n)$	$\xi_1(n)$
n_0	$\{n_0\}$	$\{n_0\}$	$\{n_4\}$	$\{n_3\}$	$\{n_3\}$
n_1	$\{n_1\}$	$\{n_1\}$	$\{n_4\}$	$\{n_3\}$	$\{n_3\}$
n_2	$\{n_2, n_3, n_4\}$	$\{n_2\}$	$\{n_7\}$	$\{n_6, n_8\}$	$\{n_5, n_6\}$
n_3	$\{n_2, n_3, n_4\}$	$\{n_2\}$	$\emptyset$	$\{n_3\}$	$\emptyset$
n_4	$\{n_2, n_3, n_4\}$	$\{n_2\}$	$\emptyset$	$\emptyset$	$\emptyset$
n_5	$\{n_5, n_6, n_7\}$	$\{n_5\}$	$\{n_9\}$	$\{n_8\}$	$\{n_8\}$
n_6	$\{n_5, n_6, n_7\}$	$\{n_5\}$	$\emptyset$	$\{n_6\}$	$\emptyset$
n_7	$\{n_5, n_6, n_7\}$	$\{n_5\}$	$\emptyset$	$\emptyset$	$\emptyset$
n_8	$\{n_8, n_9\}$	$\{n_8\}$	$\emptyset$	$\{n_8\}$	$\emptyset$
n_9	$\{n_8, n_9\}$	$\{n_8\}$	$\emptyset$	$\emptyset$	$\emptyset$

Table 1: Methods, return nodes and exceptions for the CFG in Fig. 1



D	Perm(D)
D_0	P_0, P_1
D_1	P_1, P_2
D_2	P_0, P_1, P_2
D_3	P_0

Figure 1: A CFG and its security policy.

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

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

$$\frac{n \dashrightarrow_{\lightning} n'}{\sigma : n\lightning \rhd \sigma : n'}\ [\rhd_{catch}] \qquad \frac{n \not\dashrightarrow_{\lightning}}{\sigma : n\lightning \rhd \sigma\lightning}\ [\rhd_{throw}]$$

$$\frac{}{[\,] \vdash P}\ [\vdash_1] \qquad \frac{P \in \mathrm{Perm}(n) \quad \sigma \vdash P}{\sigma : n \vdash P}\ [\vdash_2] \qquad \frac{P \in \mathrm{Perm}(n) \quad \mathrm{Priv}(n)}{\sigma : n \vdash P}\ [\vdash_3]$$

Table 2: Operational semantics of CFGs.

3.2 Semantics

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

If no exception is active, a state is represented as sequence of nodes enclosed in square brackets: for example, $\sigma = [n_0, \ldots, n_k]$ is a state whose top node is n_k. If an exception is active, we append the symbol $\lightning$ to the sequence of nodes, i.e. $\sigma\lightning$ abbreviates $\langle \sigma, \mathit{true} \rangle$. Pushing a node n on a stack σ is written as $\sigma : n$ (the infix operator : associates to the left).

The transition relation $\rhd$ between states is the minimal relation induced by the inference rules in Table 2. A *trace* of G leading to $\langle \sigma_k, x_k \rangle$ is a derivation $\langle \sigma_0, x_0 \rangle \ \rhd \ \cdots \ \rhd \ \langle \sigma_k, x_k \rangle$ where $\sigma_0 = [\,]$ and $x_0 = \mathit{false}$. By overloading the notation, we also denote with $\rhd$ the relation:

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

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

In our formalization, we use a slightly simplified version of the full access control algorithm presented in [11]. A discussion of the differences between the two models is in Section 7.

The simplified algorithm scans the call stack top-down. Each frame in the stack refers to the protection domain containing the class to which the called method belongs. As soon as a frame is found whose protection domain has not the required permission, an `AccessControlException` is raised. The algorithm succeeds when a privileged frame is found that carries the required permission, or when all frames have been visited. We formally specify this behavior by the minimal relation induced by the inference rules for $\vdash$ in Table 2. We say that a permission P is *granted* to a state σ if $\sigma \vdash P$.

Example 2. The following traces illustrate the behaviour of the CFG in Fig. 1. We also show that a check on a permission P is passed by writing $\vdash P$ and $\not\vdash P$ otherwise. The state σ in (b) and (c) can be any sequence of alternating n_0 and n_1, ending in n_1.

(a) $[] \rhd [n_1] \rhd [n_1, n_2] \rhd [n_1, n_2, n_5] \rhd [n_1, n_2, n_5, n_8] \not\vdash P_0$ (because $P_0 \notin \mathrm{Perm}(n_1)$)
$\rhd [n_1, n_2, n_5, n_8]\lightning \rhd [n_1, n_2, n_5]\lightning \rhd [n_1, n_2]\lightning \rhd [n_1, n_3] \vdash P_2 \rhd [n_1, n_4]$

(b) $[] \rhd \cdots \rhd \sigma : n_0 \rhd \sigma : n_0 : n_2 \rhd \sigma : n_0 : n_2 : n_5 \rhd \sigma : n_0 : n_2 : n_5 : n_8 \vdash P_0$
$\rhd \sigma : n_0 : n_2 : n_5 : n_9 \rhd \sigma : n_0 : n_2 : n_6 \vdash P_1 \rhd \sigma : n_0 : n_2 : n_7 \rhd \sigma : n_0 : n_4$

(c) $[] \rhd \cdots \rhd \sigma : n_0 : n_1 \rhd \sigma : n_0 : n_1 : n_2 \rhd \sigma : n_0 : n_1 : n_2 : n_5$
$\rhd \sigma : n_0 : n_1 : n_2 : n_5 : n_8 \not\vdash P_0 \rhd \sigma : n_0 : n_1 : n_2 : n_5 : n_8\lightning$
$\rhd \sigma : n_0 : n_1 : n_2 : n_5\lightning \rhd \sigma : n_0 : n_1 : n_2\lightning \rhd \sigma : n_0 : n_1 : n_3 \not\vdash P_2$
$\rhd \sigma : n_0 : n_1 : n_3\lightning \rhd \sigma : n_0 : n_1\lightning \rhd \sigma : n_0\lightning \rhd \sigma\lightning \rhd \cdots \rhd []\lightning$

4 The Trace Permissions Analysis

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

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

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

The set of permissions granted to a security context γ is $\Pi(\gamma) = \bigcap_{D \in \gamma} \mathrm{Perm}(D)$.

The permissions granted to the security context of a state σ are exactly the permissions granted to σ, as established by the following:

$$
\begin{aligned}
TP_{in}(n) &= \bigcup_{(m,n)\in E} TP_{out}(m,n) \\[1em]
TP_{out}(m,n) &= \begin{cases} \{\{\mathrm{Dom}(n)\}\} & \text{if } \bullet\!\!\longrightarrow n \\ \{\, \gamma \cup \{\mathrm{Dom}(n)\} \mid \gamma \in TP_{call}(m) \,\} & \text{if } m \longrightarrow n \\ TP_{trans}(m) & \text{if } m \dashrightarrow n \\ TP_{catch}(m) & \text{if } m \dashrightarrow_{\text{↯}} n \end{cases} \\[1em]
TP_{call}(n) &= \begin{cases} \{\{\mathrm{Dom}(n)\}\} & \text{if } \mathrm{Priv}(n) \text{ and } TP_{in}(n) \neq \emptyset \\ \quad TP_{in}(n) & \text{otherwise} \end{cases} \\[1em]
TP_{trans}(n) &= \begin{cases} \{\, \gamma \in TP_{in}(n) \mid P \in \Pi(\gamma) \,\} & \text{if } \ell(n) = \texttt{check}(P) \\ \{\, \gamma \in TP_{in}(n) \mid \mathrm{Trans}(n, \{\mathrm{Dom}(n)\}) \,\} & \text{if } \ell(n) = \texttt{call}, \mathrm{Priv}(n) \\ \{\, \gamma \in TP_{in}(n) \mid \mathrm{Trans}(n, \gamma) \,\} & \text{otherwise} \end{cases} \\[1em]
TP_{catch}(n) &= \begin{cases} \{\, \gamma \in TP_{in}(n) \mid P \notin \Pi(\gamma) \,\} & \text{if } \ell(n) = \texttt{check}(P) \\ \{\, \gamma \in TP_{in}(n) \mid \mathrm{Catch}(n, \{\mathrm{Dom}(n)\}) \,\} & \text{if } \ell(n) = \texttt{call}, \mathrm{Priv}(n) \\ \{\, \gamma \in TP_{in}(n) \mid \mathrm{Catch}(n, \gamma) \,\} & \text{otherwise} \end{cases} \\[1em]
\mathrm{Trans}(n,\gamma) &\overset{def}{=} \exists m \in \rho(n).\ \gamma \cup \{\mathrm{Dom}(m)\} \in TP_{in}(m) \\
\mathrm{Catch}(n,\gamma) &\overset{def}{=} \exists m \in \xi_1(n).\ \gamma \cup \{\mathrm{Dom}(m)\} \in TP_{catch}(m)
\end{aligned}
$$

Table 3: Flow equations for the TP analysis.

Theorem 4.1. For all $\sigma \in N^*$, $P \in \mathcal{P}$:

$$\sigma \vdash P \quad\iff\quad P \in \Pi(\Gamma(\sigma))$$

Given a CFG G and a security policy *Perm*, the analysis is specified by the set of equations $TP^{=}(G, \mathit{Perm})$ in Table 3. A solution $\tau \models TP^{=}(G, \mathit{Perm})$ is a 5-tuple $\tau = \langle \tau_{in}, \tau_{call}, \tau_{trans}, \tau_{catch}, \tau_{out} \rangle$ which satisfies all the equations. The purpose of the analysis is to find, for each node n, the set of security contexts:

$$\{\, \Gamma(\sigma : n) \mid G \,\triangleright\, \sigma : n \,\}$$

The reason why we ask the TP analysis to compute security contexts – instead of plain sets of permissions – is that the knowledge on which protection domains have been visited is needed for some optimizations (e.g. for method inlining).

Technically, TP is a forward, monotone control flow analysis with values in $2^{2^{\mathcal{D}}}$. Since both G and $\mathcal{D}$ are finite, the least solution to the analysis does exist and is finitely computable, e.g. through the worklist algorithm in Appendix A.

For each node n, edge (n, n'), and reachable state $\sigma : n$, the analysis comprises the following sets:

- $\tau_{in}(n)$ contains the context $\Gamma(\sigma) \cup \{\mathrm{Dom}(n)\}$, which is equal to the context of $\sigma : n$ if n is not privileged, and it is $\{\mathrm{Dom}(n)\}$ otherwise.
- $\tau_{call}(n)$ contains the context of $\sigma : n$, i.e. $\tau_{call}(n) \supseteq \Gamma(\sigma : n)$.

- if $\sigma : n \rhd \sigma : n'$, i.e. the control of execution flows sequentially from n to n', then $\tau_{trans}(n)$ contains the context $\Gamma(\sigma) \cup \{\mathrm{Dom}(n)\}$.
- if $\sigma : n \lightning \rhd \sigma : n'$, i.e. an exception active at n is caught by n', then $\tau_{catch}(n)$ contains the context $\Gamma(\sigma) \cup \{\mathrm{Dom}(n)\}$.
- if $\sigma : n \rhd \sigma : n : n'$, then $\tau_{out}(n, n')$ contains $\tau_{call}(n) \cup \{\mathrm{Dom}(n')\}$. If $\sigma : n \rhd \sigma : n'$, then $\tau_{out}(n, n')$ contains $\tau_{trans}(n)$. If $\sigma : n \lightning \rhd \sigma : n'$, then $\tau_{out}(n, n')$ contains $\tau_{catch}(n)$.

The only equation for TP_{in} says that $\tau_{in}(n)$ consists of the union of the contexts $\tau_{out}(m, n)$ for each edge (m, n). When (m, n) represents interprocedural flow, then $\tau_{out}(m, n)$ consists of the context at m, plus the context $\mathrm{Dom}(n)$ (second equation for TP_{out}). Otherwise, $\tau_{out}(m, n)$ comprises just the contexts at m. The first equation for TP_{call} lifts to $\mathrm{Dom}(n)$ the context of n when n is privileged. If n is a check for permission P, then $\tau_{trans}(n)$ contains all the contexts in $\tau_{in}(n)$ that pass the check (first equation for TP_{trans}). The predicate $\mathrm{Trans}(n, \gamma)$ is true for all the contexts γ such that an execution starting from n with context γ can reach a return node in some method called by n. If n is a privileged call, then $\tau_{trans}(n)$ equals to $\tau_{in}(n)$ if $\mathrm{Trans}(n, \{\mathrm{Dom}(n)\})$ is true, otherwise it is empty (second equation for TP_{trans}). If n is not privileged, then $\tau_{trans}(n)$ contains all the contexts $\gamma \in \tau_{in}(n)$ such that $\mathrm{Trans}(n, \gamma)$ is true (third equation for TP_{trans}). The predicate $\mathrm{Catch}(n, \gamma)$ is true for all the contexts γ that suffice to reach some node m that can propagate an exception catchable by n. If n is a privileged call, then $\tau_{catch}(n)$ equals to $\tau_{in}(n)$ if is $\mathrm{Catch}(n, \{\mathrm{Dom}(n)\})$ is true, otherwise it is empty (second equation for TP_{catch}). If n is not privileged, then $\tau_{catch}(n)$ contains all the contexts $\gamma \in \tau_{in}(n)$ such that $\mathrm{Catch}(n, \gamma)$ is true (third equation for TP_{catch}).

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

Theorem 4.2. *Let $\tau \models TP^{=}(G, Perm)$. Then:*

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

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

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

The worklist algorithm in Appendix A which actually computes the (unique) minimal solution to the analysis has computational complexity $\mathcal{O}(c \cdot |N|) = \mathcal{O}(|N|)$. The constant c depends on the number of protection domains occurring in G: in the worst case, $c = 2^{3 \cdot |\mathcal{D}_G|}$, where $\mathcal{D}_G = \bigcup_{n \in N} \mathrm{Dom}(n)$. However, the exponential factor only occurs when the number of protection domains is proportional to the number of nodes. Actually, the number of protection domains can be considered as a constant, because it depends on the security policy, rather than on the size of the program (for more details, see Appendix A).

Example 3. The least solution to the analysis for the CFG in Fig. 1 comprises:

$$
\begin{array}{rcl}
\tau_{in}(n_1) & = & \tau_{out}(n_\varepsilon, n_1) \cup \tau_{out}(n_0, n_1) \\
& = & \{\{\mathrm{Dom}(n_1)\}\} \cup \{\, \gamma \cup \{\mathrm{Dom}(n_1)\} \mid \gamma \in \tau_{call}(n_0)\,\} \\
& = & \{\{D_1\}\} \cup \{\{\mathrm{Dom}(n_0), \mathrm{Dom}(n_1)\}\} = \{\{D_1\}, \{D_0, D_1\}\} \\
\tau_{in}(n_0) & = & \tau_{out}(n_\varepsilon, n_0) \cup \tau_{out}(n_1, n_0) \\
& = & \{\{\mathrm{Dom}(n_0)\}\} \cup \{\, \gamma \cup \{\mathrm{Dom}(n_0)\} \mid \gamma \in \tau_{in}(n_1)\,\} = \{\{D_0\}, \{D_0, D_1\}\} \\
\tau_{in}(n_2) & = & \tau_{out}(n_0, n_2) \cup \tau_{out}(n_1, n_2) = \{\{D_0, D_2\}, \{D_1, D_2\}, \{D_0, D_1, D_2\}\} \\
\tau_{in}(n_5) & = & \{\, \gamma \cup \{\mathrm{Dom}(n_5)\} \mid \gamma \in \tau_{in}(n_2)\,\} = \{\{D_0, D_2\}, \{D_0, D_1, D_2\}\} \\
\tau_{in}(n_8) & = & \{\, \gamma \cup \{\mathrm{Dom}(n_8)\} \mid \gamma \in \tau_{in}(n_5)\,\} = \{\{D_0, D_2, D_3\}, \{D_0, D_1, D_2, D_3\}\} \\
\tau_{trans}(n_8) & = & \{\, \gamma \in \tau_{in}(n_8) \mid P_0 \in \Pi(\gamma)\,\} = \{\{D_0, D_2, D_3\}\} \\
\tau_{catch}(n_8) & = & \{\, \gamma \in \tau_{in}(n_8) \mid P_0 \notin \Pi(\gamma)\,\} = \{\{D_0, D_1, D_2, D_3\}\} \\
\tau_{in}(n_9) & = & \tau_{trans}(n_8) = \{\{D_0, D_2, D_3\}\} \\
\tau_{trans}(n_5) & = & \{\, \gamma \in \tau_{in}(n_5) \mid \mathrm{Trans}(n_5, \gamma)\,\} = \{\{D_0, D_2\}\} \\
\tau_{catch}(n_5) & = & \{\, \gamma \in \tau_{in}(n_5) \mid \mathrm{Catch}(n_5, \gamma)\,\} = \{\{D_0, D_1, D_2\}\} \\
\tau_{in}(n_6) & = & \tau_{trans}(n_5) = \{\{D_0, D_2\}\} \\
\tau_{trans}(n_6) & = & \{\, \gamma \in \tau_{in}(n_6) \mid P_1 \in \Pi(\gamma)\,\} = \{\{D_0, D_2\}\} \\
\tau_{in}(n_7) & = & \tau_{trans}(n_6) = \{\{D_0, D_2\}\} \\
\tau_{trans}(n_2) & = & \{\, \gamma \in \tau_{in}(n_2) \mid \mathrm{Trans}(n_2, \gamma)\,\} = \{\{D_0, D_2\}\} \\
\tau_{in}(n_3) & = & \tau_{catch}(n_2) = \{\, \gamma \in \tau_{in}(n_2) \mid \mathrm{Catch}(n_2, \gamma)\,\} = \{\{D_1, D_2\}, \{D_0, D_1, D_2\}\} \\
\tau_{trans}(n_3) & = & \{\, \gamma \in \tau_{in}(n_3) \mid P_2 \in \Pi(\gamma)\,\} = \{\{D_1, D_2\}\} \\
\tau_{in}(n_4) & = & \tau_{trans}(n_2) \cup \tau_{trans}(n_3) = \{\{D_0, D_2\}, \{D_1, D_2\}\}
\end{array}
$$

A feature not discussed above concerns *dynamic linking.* This mechanism allows a program to be extended on demand, e.g. with code downloaded from the network. Our program model does not directly support this feature; however, the TP analysis (though not in a fully compositional way) supports a form of incremental computation (see [2] for details).

5 Program transformations

In this section we show that the TP analysis provides us with an effective basis for several code optimizations. This is not a trivial task, because performing interprocedural optimizations in presence of stack inspection may break security. Indeed, stack inspection deeply relies on the structure of the call stack, which may be altered by such optimizations.

5.1 Elimination of redundant checks

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

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

Theorem 5.1. Let $\tau \models TP^{=}(G, Perm)$. For each node n, let $\Pi(n)$ be the set of permissions (statically) granted to n:

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

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

$$G \rhd \sigma : n \quad \Longrightarrow \quad \sigma : n \vdash P$$

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

5.2 Dead code elimination

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

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

Theorem 5.2. Let $\tau \models TP^{=}(G, Perm)$. Then:

$$\tau_{call}(n) = \emptyset \quad \Longrightarrow \quad \neg\exists\sigma.\, G \rhd \sigma : n$$

5.3 Method inlining

Method inlining is a general program optimization that replaces a method invocation with a copy of the called method code. As a side effect, the protection domain of the inlined method is ignored when performing stack inspection, which then may become unsafe.

The TP analysis can be exploited to compute the set of method invocations that can be safely inlined. Intuitively, a method invocation may be inlined if the outcome of the security checks is not affected by ignoring the protection domain of the inlined method.

We adopt the so-called *original version inlining* approach [15], which always considers the original version of the callee and the current version of the caller when performing inlinings. This can be obtained by duplicating the original code of the inlined method.

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

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

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

We formally specify in definition 5.3 when a method invocation can be safely inlined. The condition (1a) guarantees static dispatching of $\dot{n}$, as well as that $\dot{n}$ is not a recursive call (otherwise inlining makes little sense). The condition (1b) rephrases the original version inlining approach. The condition (1c) ensures that the protection domain of $\dot{n}$ is isolated. These conditions, apart from $\dot{n}$ being not recursive, can easily be satisfied, as noted above. The key condition is (1d): it guarantees that the security checks passed after inlining are exactly those passed before inlining.

Definition 5.3. We say that $\dot{n} \longrightarrow n'$ is *inlineable* in G iff, for each $n \in N$:

$$\dot{n} \longrightarrow n \implies n = n' \ \wedge \ n \notin \mu(\dot{n}) \tag{1a}$$

$$n \longrightarrow n' \implies n = \dot{n} \tag{1b}$$

$$n \notin \mu(n') \implies \mathrm{Dom}(n) \neq \mathrm{Dom}(n') \tag{1c}$$

$$\ell(n) = \mathtt{check}(P) \implies \forall \gamma \in \tau_{in}(n).\ P \in \Pi(\gamma) \iff P \in \Pi(\mathit{Inl}_{\dot{n}}(\gamma)) \tag{1d}$$

We say that $\dot{n}$ is inlineable iff there exists some n' such that $\dot{n} \longrightarrow n'$ is inlineable.

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

The effect of the inlining of $\dot{n}$ on states is specified by the function $\mathit{inl}_{\dot{n}}$ in Table 4. Given a state σ, $\mathit{inl}_{\dot{n}}(\sigma)$ is obtained by removing all the occurrences of $\dot{n}$ in σ (except when $\dot{n}$ is in top position).

The operational semantics of a CFG after the inlining of $\dot{n}$ is defined by the transition relation $\rhd^{\dot{n}}$ in Table 4. For instance, the rules $\rhd_{icall1}$ and $\rhd_{icall2}$ state, respectively, that method invocation proceeds as usual when the calling node is not $\dot{n}$, otherwise $\dot{n}$ is removed from the call stack.

Definition 5.4. Let G be the CFG $\langle N \cup \{n_\varepsilon\}, E, \mathrm{Priv}_G, \mathrm{Dom}_G \rangle$. The *$\dot{n}$-inlined version of* G is the CFG $\dot{G} = \langle N \cup \{n_\varepsilon\}, E, \mathrm{Priv}_{\dot{G}}, \mathrm{Dom}_{\dot{G}} \rangle$, where:

$$\mathrm{Priv}_{\dot{G}}(n) \quad = \quad \begin{cases} \mathit{true} & \text{if } \mathrm{Priv}_G(\dot{n}) \text{ and } \dot{n} \longrightarrow \mu(n) \\ \mathrm{Priv}_G(n) & \text{otherwise} \end{cases}$$

$$\mathrm{Dom}_{\dot{G}}(n) \quad = \quad \begin{cases} \mathrm{Dom}_G(\dot{n}) & \text{if } \dot{n} \longrightarrow \mu(n) \\ \mathrm{Dom}_G(n) & \text{otherwise} \end{cases}$$

According to the above definitions, we may end up with privileged checks and returns. We already pointed out that this would only require a more involved definition of the TP analysis (see the discussion on constraint 5).

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

$$\frac{}{\mathit{inl}_{\dot n}([]) = []}\;[\mathit{inl}_1] \qquad \frac{\mathit{inl}_{\dot n}(\sigma) = \dot\sigma \quad \mathit{top}(\sigma) \neq \dot n}{\mathit{inl}_{\dot n}(\sigma : n') = \dot\sigma : n'}\;[\mathit{inl}_2] \qquad \frac{\mathit{inl}_{\dot n}(\sigma) = \dot\sigma}{\mathit{inl}_{\dot n}(\sigma : \dot n : n') = \dot\sigma : n'}\;[\mathit{inl}_3]$$

$$\frac{}{\mathit{Inl}_{\dot n}(\emptyset) = \emptyset}\;[\mathit{Inl}_1] \qquad \frac{\mathit{Inl}_{\dot n}(\gamma) = \dot\gamma \quad \mathit{Inl}_{\dot n}(\gamma') = \dot\gamma'}{\mathit{Inl}_{\dot n}(\gamma \cup \gamma') = \dot\gamma \cup \dot\gamma'}\;[\mathit{Inl}_2]$$

$$\frac{\dot n \not\longrightarrow \mu(n')}{\mathit{Inl}_{\dot n}(\{\mathrm{Dom}(n')\}) = \{\mathrm{Dom}(n')\}}\;[\mathit{Inl}_3] \qquad \frac{\dot n \longrightarrow \mu(n')}{\mathit{Inl}_{\dot n}(\{\mathrm{Dom}(n')\}) = \{\mathrm{Dom}(\dot n)\}}\;[\mathit{Inl}_4]$$

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n' \quad n \neq \dot n}{\sigma : n \rhd^{\dot n}_{\mathit{inl}} \sigma : n : n'}\;[\rhd_{\mathit{icall1}}] \qquad \frac{\ell(\dot n) = \texttt{call} \quad \dot n \longrightarrow n'}{\sigma : \dot n \rhd^{\dot n}_{\mathit{inl}} \sigma : n'}\;[\rhd_{\mathit{icall2}}]$$

$$\frac{\ell(n') = \texttt{return} \quad \begin{array}{l} n \dashrightarrow m \\ \dot n \not\longrightarrow \mu(n') \end{array}}{\sigma : n : n' \rhd^{\dot n}_{\mathit{inl}} \sigma : m}\;[\rhd_{\mathit{iret1}}] \qquad \frac{\ell(n') = \texttt{return} \quad \begin{array}{l} \dot n \dashrightarrow m \\ \dot n \longrightarrow \mu(n') \end{array}}{\sigma : n' \rhd^{\dot n}_{\mathit{inl}} \sigma : m}\;[\rhd_{\mathit{iret2}}]$$

$$\frac{n \not\dashrightarrow_{\lightning} \quad \dot n \not\longrightarrow \mu(n')}{\sigma : n\lightning \rhd^{\dot n}_{\mathit{inl}} \sigma\lightning}\;[\rhd_{\mathit{ithrow1}}] \qquad \frac{n \not\dashrightarrow_{\lightning} \quad \dot n \longrightarrow \mu(n')}{\sigma : n\lightning \rhd^{\dot n}_{\mathit{inl}} \sigma : \dot n\lightning}\;[\rhd_{\mathit{ithrow2}}]$$

Table 4: Specification of method inlining.

Theorem 5.5. If $\dot n$ is inlineable in G and $\dot G$ is the $\dot n$-inlined version of G, then:

$$\langle \sigma_0, x_0 \rangle \ \rhd \ \cdots \ \rhd \ \langle \sigma_k, x_k \rangle \qquad \Longleftrightarrow \qquad \langle \dot\sigma_0, x_0 \rangle \ \rhd^{\dot n}_{\mathit{inl}} \ \cdots \ \rhd^{\dot n}_{\mathit{inl}} \ \langle \dot\sigma_k, x_k \rangle$$

where $\sigma_0 = []$, $x_0 = \mathit{false}$, and $\dot\sigma_i = \mathit{inl}_{\dot n}(\sigma_i)$ for each $i \in 0..k$.

5.4 Eager stack inspection

The eager evaluation strategy for stack inspection allows security checks to be performed very efficiently; there is however an overhead at each cross-domain method invocation (and return), because the security context must be keep updated. Since security checks are statistically less frequent than cross-domain calls, actual implementations of the JDK adopt the lazy strategy. However, the eager strategy is still worth of consideration: indeed, in combination with the *security passing style* of [29] it allows for interprocedural optimizations, which are instead prevented by the lazy strategy. Actual implementations of the eager evaluation strategy [30] have nevertheless showed worse performance than implementations of the lazy strategy.

Our TP analysis helps in improving the performance of eager stack inspection. Below, we specify a novel implementation technique for it, which exploits our analysis to efficiently update the security contexts.

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

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

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

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

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

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

Formally, the context updating operations are implemented as follows:

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

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

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

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

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

Compared with the lazy evaluation strategy, our technique involves an overhead at each method invocation: besides the cost of passing an additional parameter, we have to perform either a "bitwise or" operation (or a hash table lookup). Security checks require a "bitwise and" (or a hash table lookup), and are independent of the size of the call stack. As a matter of fact, a good choice of the set $\mathcal{D}_0$ is crucial, and possibly requires a statistical estimate of the frequency of contexts.

6 An e-commerce example

To illustrate our analysis, we consider small e-commerce application written in Java, and displayed in Appendix B. As a simple optimization we then detect the redundant checks. Four actors are involved:

- an application server (`Bank`), with a remote interface to perform queries and transactions over bank accounts.
- an e-commerce provider (`Shop`), which interacts with users and the bank to manage the acquisition and payment of goods.
- a fraudulent entity (`Robber`), which spoofs for a trusted e-commerce provider.
- a user agent (`Client`), which exploits the e-commerce facilities offered by `Shop` and `Robber`.

We consider below only the security properties of the code executed on the client and on the application server.

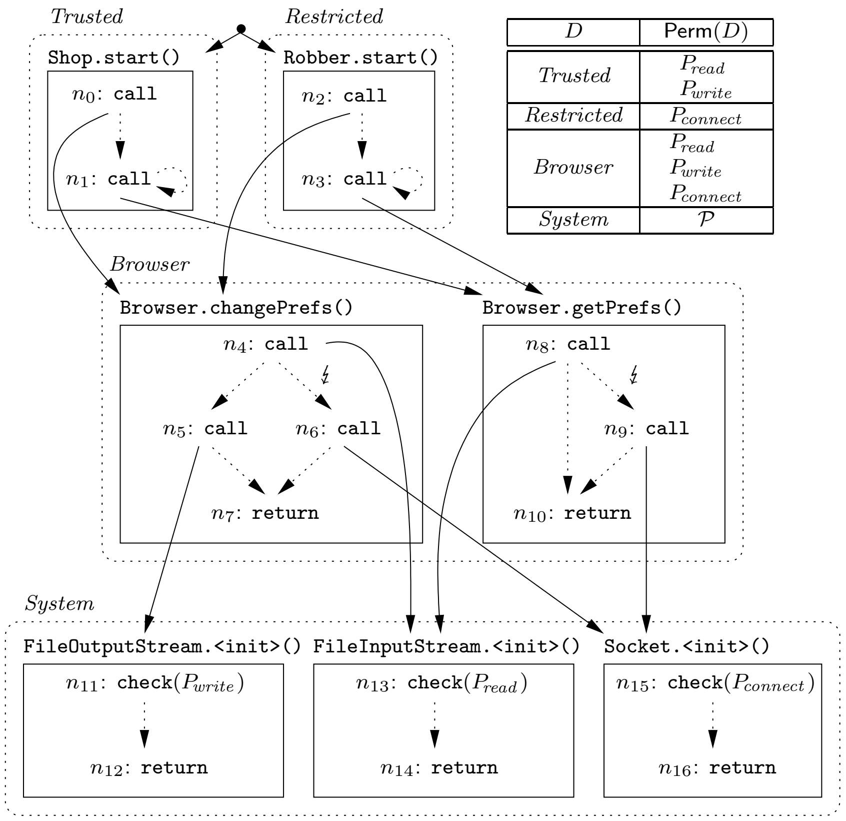
The user agent runs a Java-enabled Web browser, which has the rights to access the local filesystem (in both read and write mode), and to open a socket connection. The client-tier components of the trusted and untrusted e-commerce providers are implemented as Java applets; so, they are executed by the Java virtual machine embedded in the browser.

The class `Browser` provides the applets with some facilities to manage the user preferences: the `getPrefs()` method tries to retrieve the preferences (e.g. payment settings) from a local file, if the applet has the rights to. Otherwise, it opens a socket connection with the remote server where the applet was downloaded. The remote server is then delegated to send back the preferences to the applet. The `changePrefs()` method first looks for the old preferences (either they are in the local disk or on the remote server); then, it asks the user for the new preferences, which are thereafter saved on the local disk (if the applet has the rights to) or sent to the remote server.

The `Shop` applet is basically a two-stage agent. First, the applet registers the user profile. The user is asked to fill a form, containing e.g. its bank account number; then, the registration data is stored by the `changePrefs()` method. Once the user has been registered, it can start purchasing items from the provider. When the user orders an item, the applet looks for its registration data by calling the `getPrefs()` method. The data is sent back to the e-commerce server, which performs the transaction on the bank server.

The `Robber` applet acts similarly. Indeed, the two applets cannot be distinguished by looking solely at their control flow. However, this applet may still perform some fraudulent operations by acting on the data flow. For instance, it can exploit the user account number to steal the user money. Moreover, the robber can corrupt some sensitive information on the user local disk by a malicious use of the `changePrefs()` method.

The `Bank` class features a defensive implementation of a simple account manager. Its public interface consists of one boolean query (`canpay()`) and three transactions (`debit()`, `credit()` and `transfer()`). Each of these methods is protected by an appropriate security check. The `canpay()` method tests if at least a given amount of money can be withdrawn from a bank account. The `debit()` method withdraws a sum from that account, provided there is enough



D	$\mathsf{Perm}(D)$
Trusted	P_{read} P_{write}
Restricted	$P_{connect}$
Browser	P_{read} P_{write} $P_{connect}$
System	$\mathcal{P}$

Figure 2: CFG and security policy for the e-commerce application (client side).

money. The `credit()` method deposits a sum in an account. The `transfer()` method withdraws (if possible) a sum from an account, which it then deposited into another account. The local file system is always accessed in privileged mode.

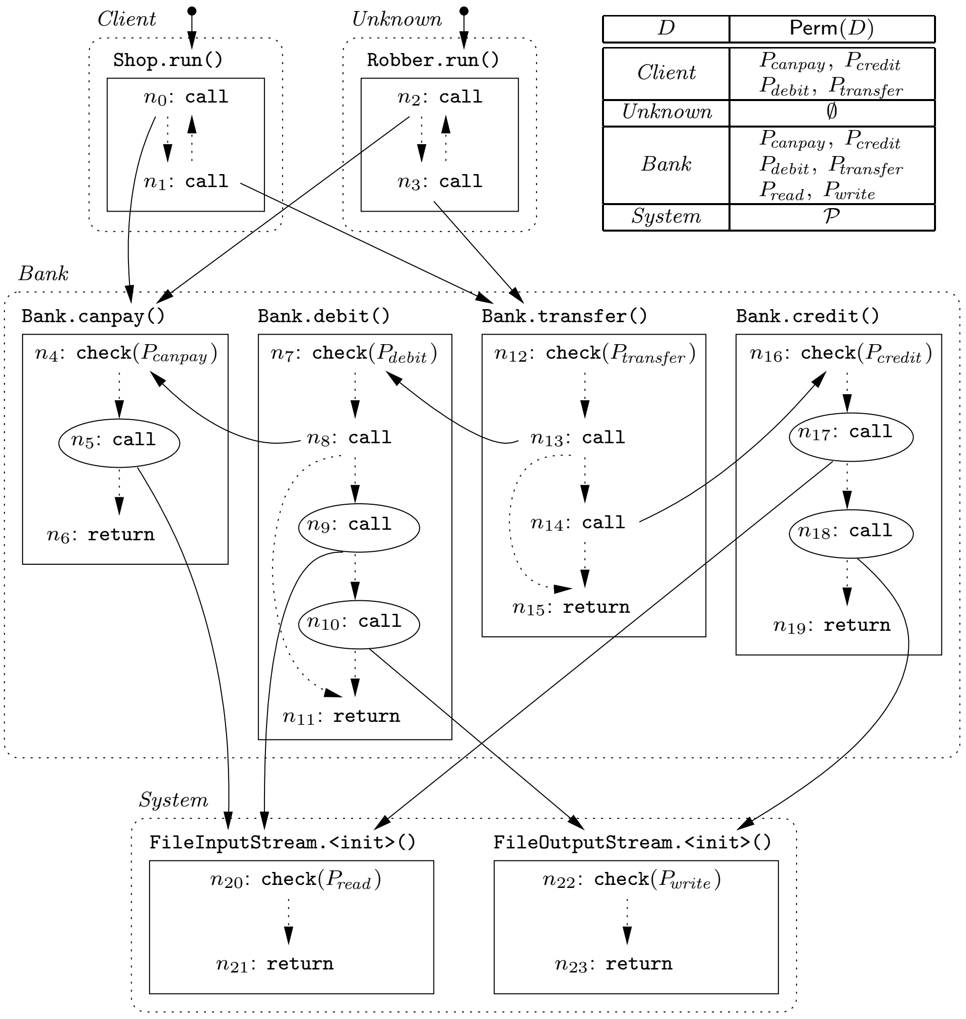
Two clients aim at exploiting the services offered by the bank: `Shop`, which is trusted, and `Robber`, which is not granted any permission.

The `Shop` performs the money transfer from the user account, having first checked that the user has enough money. This is done by invoking the methods `canpay()` and `transfer()` in sequence.

The `Robber` acts in the same manner. Again, its behavior cannot be taken apart from that of `Shop` by looking at the control flow, only. The `Robber` tries to steal money from the user account, but it has not the rights to (even if it has knowledge of the user account number).

The CFGs and the security policies extracted from the Java programs in Appendix B are shown in Figures 2 and 3, respectively. Dashed boxes enclose the methods belonging to the same protection domain. Notice that the conditional constructs are modeled by non-determinism in the intraprocedural flow.

The results of the iterations of the worklist algorithm for the client-side and for the server-side are shown respectively in Tables 6 and 7 in Appendix B. We abbreviate the names of protection domains using their initial letters; for instance, "UB" stands for the security context $\{\mathit{Unknown}, \mathit{Bank}\}$. Notice that,



D	Perm(D)
Client	P_{canpay}, P_{credit} P_{debit}, $P_{transfer}$
Unknown	$\emptyset$
Bank	P_{canpay}, P_{credit} P_{debit}, $P_{transfer}$ P_{read}, P_{write}
System	$\mathcal{P}$

Figure 3: CFG and security policy for the e-commerce application (server side).

in Table 6, $\tau_{in}(n) = \tau_{call}(n)$, because no call is privileged in the CFG in Fig. 2.

We exploit the TP analysis to detect the redundant checks. Consider the client-side in Fig. 2 first. By Table 6, we have that, at node n_{11}:

$$\tau_{call}(n_{11}) \quad = \quad \{\{\mathit{Trusted}, \mathit{Browser}, \mathit{System}\}\}$$

By definition of Π, we have that $\Pi(n_{11}) = \mathrm{Perm}(\mathit{Trusted})$. Then, the check at n_{11} is redundant, because $P_{write} \in \mathrm{Perm}(\mathit{Trusted})$. For node n_{13}, we have:

$$\tau_{call}(n_{13}) \quad = \quad \{\{\mathit{Trusted}, \mathit{Browser}, \mathit{System}\}, \{\mathit{Restricted}, \mathit{Browser}, \mathit{System}\}\}$$

Here $\Pi(n_{13}) = \mathrm{Perm}(\mathit{Trusted}) \cap \mathrm{Perm}(\mathit{Restricted}) = \emptyset$, so the check at n_{13} is indeed necessary. For node n_{15} we have:

$$\tau_{call}(n_{15}) \quad = \quad \{\{\mathit{Restricted}, \mathit{Browser}, \mathit{System}\}\}$$

Then, $\Pi(n_{15}) = \mathrm{Perm}(\mathit{Restricted})$. Since $P_{connect} \in \mathrm{Perm}(\mathit{Restricted})$, the check at n_{15} is redundant.

We now focus on the server-side (Fig. 3). By Table 7, we have that, at n_4:

$$\tau_{call}(n_4) \quad = \quad \{\{\mathit{Client}, \mathit{Bank}\}, \{\mathit{Unknown}, \mathit{Bank}\}\}$$

Therefore, the check at n_4 is indeed necessary, because $P_{canpay} \notin \Pi(n_4) = \emptyset$. For nodes n_7, n_{12} and n_{16}, we have:

$$\tau_{call}(n_7) = \tau_{call}(n_{12}) = \tau_{call}(n_{16}) \quad = \quad \{\{\mathit{Client}, \mathit{Bank}, \mathit{System}\}\}$$

Then, $\Pi(n_7) = \Pi(n_{12}) = \Pi(n_{16}) = \mathrm{Perm}(\mathit{Client})$. Since $P_{debit}, P_{transfer}, P_{credit}$ are granted to *Client*, it turns out that all the checks in `Bank`, except n_4, are redundant. Finally, for nodes n_{20} and n_{22}, we have:

$$\tau_{call}(n_{20}) = \tau_{call}(n_{22}) \quad = \quad \{\{\mathit{Bank}, \mathit{System}\}\}$$

Then, both the checks are redundant, because $\Pi(n_{20}) = \Pi(n_{22}) = \mathrm{Perm}(\mathit{Bank})$ and $P_{read}, P_{write} \in \mathrm{Perm}(\mathit{Bank})$.

7 Model adequacy and future work

There are some differences between our model and the Java security model [11]:

- our model prevents a permission P to be granted to a state $\sigma : n$ if P does not belong to the permissions granted to $\mathrm{Dom}(n)$, i.e. $P \notin \mathrm{Perm}(n)$ implies $\sigma : n \not\vdash P$. Instead, in the Java security model, P may be *implied* by some permission $P' \in \mathrm{Perm}(n)$. For example, `FilePermission("/-","read")` implies the permission to read any file on the local disk. We can easily extend our program model by introducing a partial order on permissions to encompass permission implications. The inclusion test $P \in \mathrm{Perm}(n)$ in the rules for $\vdash$ should be replaced by $\mathrm{Perm}(n) \Rightarrow P$, which tests if P is implied by some permission $P' \in \mathrm{Perm}(n)$.

- although the Java security model allows for the dynamic instantiation of permissions (e.g. an application that asks the user for a file name and then tries to open that file), we only consider the permissions that can be determined statically. We are investigating an extension of our present approach to deal with such *parametric permissions* on the form $P(x)$, where x ranges over the set of possible targets for the permissions of class P.

- starting from version 1.4.1, the Java SDK has added support for dynamic security policies. This means that the binding between a class and its permissions can be deferred until the class protection domain is involved in an access control test (rather than when the class is loaded). With our current approach, the whole analysis has to be recomputed from scratch each time the security policy is changed. A control flow analysis which is parametrized with respect to the security policy allows for more incremental behaviour. This subject is currently under investigation.

- in the Java security model, a new thread upon creation inherits the *access control context* (i.e. the set of protection domains for the classes on the call stack) from its parent. When stack inspection is performed, both the context of the current thread and the contexts of all its ancestors are examined. In this way, a child thread cannot obtain a resource access which is not granted to its ancestors. We do not model threads. To consider them, we should first single out the program points where new threads can be created (and started) while constructing the CFG (as done in [16]).

- here, we consider a "skeletal" exception handling mechanism, where exceptions are all of the same type, and neither nested `try` blocks nor `finally` clauses are featured. A full treatment of exceptions requires a tailored construction of the CFG, e.g. by the techniques presented in [6, 24], that also suggest how to adjust interprocedural analyses to exceptions.
- in our model, only *code-centric* security policies are allowed: permissions are granted to code according to its code source, regardless of who is running it. The Java Authentication and Authorization Service [17], extends the Java security model by enabling *user-centric* access control policies, based on the principal who actually runs the code. Permissions can be granted to principals, and the `doAs` method allows a piece of code to be executed on behalf of a given subject. This is done by associating the (authenticated) subject running the code with the current access control context. Stack inspection ensures that subjects are taken into account when access control is performed (see e.g. [14] for a formal specification).

There are some features of the Java security architecture we think difficult to cope with: they are reflection, native methods, and some "dangerous" permissions implications (e.g. `AllPermission` may even breach the whole security system by replacing the JVM system binaries). Besides deeply affecting security, these features reduce the effectiveness of any analysis which aims at determining statically the permissions granted to running code.

8 Conclusions and related work

We have developed a technique to perform program transformations in presence of stack inspection. The technique relies on the definition of our Trace Permissions Analysis. It is a control flow analysis, and computes a safe approximation to the set of permissions which are *always* granted to bytecode at run-time. The analysis is sound and complete w.r.t. the control flow graphs derived from the bytecode (however, these graphs only approximate the actual behaviour). Our analysis makes various optimizations possible. We focussed here on elimination of redundant checks and of dead code, on method inlining, and on eager stack inspection. A similar approach also applies to general tail call elimination. Although we restricted our attention to Java, the same techniques work as well with other programming languages whose authorization mechanisms rely on stack inspection (e.g. C$\sharp$ [31]). It is worthwhile noting that our analysis can take advantage of the control flow graphs generated by the HotSpot optimizers embedded in the latest JVMs [25]. This would also make our technique directly exploitable by these tools, e.g. to produce larger methods by inlining, so allowing for further optimizations.

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

Besson, Jensen, Le Mètayer and Thorn [5] were among the first to apply static techniques to the verification of global security properties. They formalize classes of security properties through a linear-time temporal logic. They show that a large class of policies (including stack inspection) can be expressed in this formalism, while more sophisticated ones (like the Chinese Wall policy) are not. Model checking is then used to prove that local security checks enforce a given global security policy. Their verification method is based on the translation from linear-time temporal formulae to deterministic finite-state automata, and it can be used to optimize stack inspection. For each node n, the analysis in [5] can compute the set $\{ P \in \mathcal{P}_{check} \mid G \rhd \sigma : n \ \wedge \ \sigma : n \vdash P \}$, where $\mathcal{P}_{check}$ is the set of permissions checked in G. The computational complexity of the method is $\mathcal{O}(c \cdot |N|)$, where the constant c depends on the cardinality of $\mathcal{P}_{check}$ (in the worst case, $c = 2^{|\mathcal{P}_{check}|}$). Therefore, our Trace Permissions analysis performs better when there are few protection domains, while [5] is more efficient when there are few security checks. Note that our analysis is at least as precise as [5], because $\mathcal{P}_{check} \subseteq \mathcal{P}$. Also, the analysis in [5] does not seem to scale up smoothly to handle dynamic linking, because it must be recomputed each time a new permission is discovered.

Based on the same program model of [5], Besson, de Grenier de Latour and Jensen [4] develop a static analysis that computes, for each method, the set of its *secure calling contexts* with respect to a given global security property. When a method is invoked from a secure calling context, it is guaranteed that its execution will not violate the global property. For some optimizations, e.g. for method inlining, this technique is even too powerful, as the information about calling contexts is unnecessary.

Esparza, Kučera and Schwoon [9] tackle stack inspection in terms of model checking pushdown systems (PDSs). Obdržálek [20] uses the same technique to accurately model Java exception handling. A suitable combination of the two will then be an alternative approach to ours. Since our model is specifically tailored on stack inspection, we think that our analysis may be implemented and exploited more efficiently than a general method such as model checking PDSs. Like in [5], the latter approach seem to suffer from dynamic linking, in particular when some program transformations (e.g. method inlinings) have to be revalidated at run-time.

Walker [28] explores an alternative approach. When a security-unaware program is compiled, a centralized security policy dictates where to insert run-time checks, in order to obtain provably-secure compiled code. An optimization phase follows: whenever a security check is removed, it is replaced by a proof that the optimized code is still safe. Before executing a piece of code, a certified verification software ensures that it respects the centralized security policy. Security properties are specified by *security automata* [23, 3]. This mechanism is rather powerful (it subsumes linear temporal logic), but it does not handle some interesting properties, e.g. information flow, resource availability, liveness, performance. Actually, security automata can only express a proper subset of the class of *safety* properties. The policies specified by security automata can be enforced by *inlined reference monitors*, which control the execution steps of target applications, terminating those about to violate the security policy in force. Inlined reference monitors have been used to enforce Java stack inspection [8].

One of the main drawbacks of [5, 28] is the difficulty of mechanically determining a "suitable" global security property for an arbitrary program, i.e. a property that, if enforced, guarantees safe execution.

Wallach, Appel and Felten [30] formalize stack inspection by exploiting the access control logic of [1]. The authors show that their decision procedure is equivalent to Java stack inspection, according to an informal operational semantics. Moreover, they propose an alternative semantics of eager stack inspection, called *security-passing style*. This technique consists of tracking the security state of an execution as an additional parameter of each method invocation. This allows for interprocedural compiler optimizations that do not interfere with stack inspection. The security-passing style allows each security operation to be performed in constant time, but it involves an overhead, because the security state must be computed at each method invocation. Dynamic caching techniques are adopted to reduce this overhead: therefore, in the optimal case, the additional cost of each method invocation is that of a hash lookup. The same technique allows for an implementation of security checks which requires a hash lookup in the optimal case. Instead, in our approach, each security operation costs as a hash lookup *in the worst case*, while, in the optimal case, it is as cheap as a bitwise operation. A further difference w.r.t. our approach is that [30] assumes that the whole program is available at compilation time.

Pottier, Skalka and Smith [21] address the problem of stack inspection in λ_{sec}, a typed lambda calculus enriched with primitive constructs for enforcing security checks and managing permissions. They have polymorphic types on the form $\tau_1 \rightarrow \varsigma \rightarrow \tau_2$, where τ_1, τ_2 are types and ς is a set of permissions. Intuitively, the type $\tau_1 \rightarrow \varsigma \rightarrow \tau_2$ details the security context necessary to execute a function of type $\tau_1 \rightarrow \tau_2$. Stack inspection never fails on a well typed program, because the set of permissions granted at run-time always includes the security context. These types are very powerful and can deal with several issues (e.g. security policy overriding and dependencies from untrusted code). Moreover, they can be smoothly extended to deal with objects by standard type-theoretic techniques. This analysis supports all-or-nothing optimizations that remove the security manager when all the checks are redundant. Instead, we can single out and remove individual redundant checks.

The problem of establishing the correctness of program transformations in presence of stack inspection is investigated by Fournet and Gordon in [10]. They present an equational theory, together with a coinductive proof technique, for the λ_{sec} calculus. They study how stack inspection affects program behavior, proving that certain function inlinings and tail-call eliminations are correct. The equational theory is used to reason about the (somewhat limited) security properties actually guaranteed by stack inspection. Here, we are more concerned with efficient (semantically-based) optimization procedures to be used on the field, rather than with a general reasoning framework. Indeed, it is unclear how to (mechanically) derive a procedure (e.g. a confluent terminating rewriting system) to ensure correctness of program transformations under security constraints.

Clemens and Felleisen [7] present a different semantics of (eager) stack inspection on continuation CESK machines, which allows for tail-call optimizing implementations.

Compared with our approach, [7, 10, 21] consider more basic programming primitives (e.g. there is no exception mechanisms). Also, static typing appears to be more difficult than control flow analysis when permissions can be dynamically instantiated. Indeed, we argue that typing and control flow analysis are complementary static techniques. Approaches based on types focus more on defining safe programming disciplines; control flow analysis, instead, seems more accurate in efficiently determining effective program optimizations.

Koved, Pistoia and Kershenbaum [16] address the problem of computing the set of permissions a class needs in order to execute without throwing security exceptions. Also this analysis suffers from allowing only all-or-nothing optimizations, as in [21]. The analysis is built over *access rights invocation graphs*. These flow graphs are context-sensitive: each node is associated also with its *calling context*, i.e. with its target method, receiver and parameters values. In this way, the analysis in [16] can deal with parametric permissions and multi-threading. Our approach can gain precision through the exploitation of these graphs. We plan to study this issue in future work.

Acknowledgements

We thank anonymous referees for their valuable comments.

References

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 4(15):706–734, Sept. 1993.

[2] M. Bartoletti, P. Degano, and G. Ferrari. Security-aware program transformations. In *Proc. 8th Italian Conference on Theoretical Computer Science*, 2003.

[3] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Foundations of Computer Security (FCS '02)*, July 2002.

[4] F. Besson, T. de Grenier de Latour, and T. Jensen. Secure calling contexts for stack inspection. In *Proc. 4th Conference on Principles and Practice of Declarative Programming*. ACM Press, 2002.

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

[6] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Workshop on Program Analysis For Software Tools and Engineering*, 1999.

[7] J. Clemens and M. Felleisen. A tail-recursive semantics for stack inspections. In P. Degano, editor, *Proc. 12th European Symposium on Programming*, volume 2618 of *LNCS*. Springer-Verlag, 2003.

[8] U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, 2000.

[9] J. Esparza, A. Kučera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. In *Proc. 4th International Symposium on Theoretical Aspects of Computer Software*, 2001.

[10] C. Fournet and A. D. Gordon. Stack inspection: theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, 2003.

[11] L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation.* Addison-Wesley, 1999.

[12] L. Gong and R. Schemers. Implementing protection domains in the Java Development Kit 1.2. In *Proc. Internet Society Symposium on Network and Distributed System Security*, Mar. 1998.

[13] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6), 2001.

[14] G. Karjoth. An operational semantics for Java 2 access control. In *Proc. 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

[15] O. Kaser and C. R. Ramakrishnan. Evaluating inlining techniques. *Computer Languages*, 24(2):55–72, 1998.

[16] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. In *Proc. 17th ACM conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM Press, 2002.

[17] C. Lai, L. Gong, L. Koved, A. Nadalin, and R. Schemers. User authentication and authorization in the Java platform. In *Proc. 15th Annual Computer Security Application Reference*. IEEE Computer Society Press, 1999.

[18] Microsoft Corp. *.NET Framework Developer's Guide: Securing Applications*.

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

[20] J. Obdržálek. Model checking java using pushdown systems. In *Workshop on Formal Techniques for Java-like Programs*, 2002.

[21] F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. In D. Sands, editor, *Proc. 10th European Symposium on Programming*, volume 2028 of *LNCS*. Springer-Verlag, Apr. 2001.

[22] F. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead*. Springer-Verlag, 2001.

[23] F. B. Schneider. Enforceable security policies. Technical Report TR98-1664, Cornell University, Jan. 1998.

[24] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *Software Engineering*, 26(9):849–871, 2000.

[25] Sun Microsystems. *The Java HotSpot Virtual Machine (Technical White Paper)*.

[26] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Proc. 15th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, volume 35(10). ACM Press, Oct. 2000.

[27] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proc. 15th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 2000.

[28] D. Walker. A type system for expressive security policies. In *Conference record of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2000.

[29] D. S. Wallach. *A New Approach to Mobile Code Security*. PhD thesis, Princeton University, Jan. 1999.

[30] D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM TOSEM*, 9(4):341–378, Oct. 2001.

[31] C. Wille. *Presenting* C$\sharp$. SAMS Publishing, 2000.

A Proofs

A.1 Properties of CFGs

Lemma A.1. Let:

$$\langle \sigma_0, x_0 \rangle \ \rhd \ \cdots \ \rhd \ \langle \sigma_k, x_k \rangle$$

be a trace on G, where $\sigma_k = \sigma : n : m$. Then:

$$n \longrightarrow \mu(m) \tag{1a}$$

$$\exists i \in 1..k-1.\ \langle \sigma_i, x_i \rangle = \langle \sigma : n, \mathit{false} \rangle \tag{1b}$$

Proof. We proceed by induction on the length of the trace. The base case $k = 0$ holds trivially, because $\sigma_0 = []$.

For the inductive case, assume (1a) and (1b) are true for all traces of length lower than k. By case analysis on the rule used to deduce $\langle \sigma_{k-1}, x_{k-1} \rangle \rhd \langle \sigma_k, x_k \rangle$, we have:

- case [*call*]:

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow m}{\sigma : n \rhd \sigma : n : m}$$

  Here (1a) follows by the fact that $n \longrightarrow m$, and the index i which satisfies (1b) is just $k-1$. Notice that it must be $k - 1 > 0$, because any derivation for $\sigma : n$ requires at least one step.

- case [*ret*]:

$$\frac{\ell(n') = \texttt{return} \quad m' \dashrightarrow m}{\sigma : n : m' : n' \rhd \sigma : n : m}$$

  By the induction hypothesis, we have that:

$$\exists j \in 1..k-2.\ \langle \sigma_j, x_j \rangle = \langle \sigma : n : m', \mathit{false} \rangle \tag{2}$$

  Since any derivation for $\sigma : n : m'$ requires at least two steps, it must be $j > 1$. Then, the induction hypothesis on $j - 1$ gives $n \longrightarrow \mu(m')$, and:

$$\exists i \in 1..j-1.\ \langle \sigma_i, x_i \rangle = \langle \sigma : n, \mathit{false} \rangle \tag{3}$$

  Since $m' \dashrightarrow m$, we have $\mu(m') = \mu(m)$: then, $n \longrightarrow \mu(m')$ implies $n \longrightarrow \mu(m)$, and this proves (1a). Moreover, (1b) is satisfied with the index i given by (3).

- case [*pass*]:

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

  By the induction hypothesis, $n \longrightarrow \mu(m')$, and:

$$\exists i \in 1..k-2.\ \langle \sigma_i, x_i \rangle = \langle \sigma : n, \mathit{false} \rangle \tag{4}$$

  Since $m' \dashrightarrow m$, we have $\mu(m') = \mu(m)$: then, $n \longrightarrow \mu(m')$ implies $n \longrightarrow \mu(m)$, and this proves (1a). Moreover, (1b) is satisfied with the index i given by (4).

- case [*fail*]:

$$\frac{\ell(m) = \texttt{check}(P) \quad \sigma : n : m \not\vdash P}{\sigma : n : m \rhd \sigma : n : m\lightning}$$

  Here (1a) and (1b) follow directly by the induction hypothesis.

- case $[catch]$:
$$\frac{m' \dashrightarrow_{\lightning} m}{\sigma : n : m' \lightning \rhd \sigma : n : m}$$
By the induction hypothesis, we have $n \longrightarrow \mu(m')$, and:
$$\exists i \in 1..k-2.\ \langle \sigma_i, x_i \rangle = \langle \sigma : n, \mathit{false} \rangle \tag{5}$$
Since $m' \dashrightarrow_{\lightning} m$, we have $\mu(m') = \mu(m)$: then, $n \longrightarrow \mu(m')$ implies $n \longrightarrow \mu(m)$, and this proves (1a). Moreover, (1b) is satisfied with the index i given by (5).
- case $[throw]$:
$$\frac{m' \not\dashrightarrow_{\lightning}}{\sigma : n : m : m' \lightning \rhd \sigma : n : m \lightning}$$
By the induction hypothesis, we have that:
$$\exists j \in 1..k-2.\ \langle \sigma_j, x_j \rangle = \langle \sigma : n : m, \mathit{false} \rangle \tag{6}$$
Since any derivation for $\sigma : n : m$ requires at least two steps, it must be $j > 1$. Then, the induction hypothesis on $j-1$ gives $n \longrightarrow \mu(m)$, and:
$$\exists i \in 1..j-1.\ \langle \sigma_i, x_i \rangle = \langle \sigma : n, \mathit{false} \rangle \qquad \square$$

Lemma A.2.
$$G \rhd \sigma : n : m \ \wedge \ \ell(m) = \texttt{return} \quad \Longrightarrow \quad m \in \rho(n)$$

Proof. By lemma A.1, $G \rhd \sigma : n : m$ implies that $n \longrightarrow \mu(m)$. Since $\ell(m) = \texttt{return}$, by definition 3.3 it follows that $m \in \rho(n)$. $\square$

Lemma A.3.
$$G \rhd \sigma : n \lightning \quad \Longrightarrow \quad \xi(n) \neq \emptyset$$

Proof. Let $\langle \sigma_0, x_0 \rangle \rhd \cdots \rhd \langle \sigma_k, x_k \rangle$ be a trace leading to $\sigma : n \lightning$, and define:
$$i^\star = \max \{ i \in 0..k-1 \mid x_i = \mathit{false} \}$$
Since the only rule that raises an exception is $\rhd_{\mathit{fail}}$, the transition $\sigma_{i^\star} \ \rhd \ \sigma_{i^\star+1} \lightning$ must be on the form:
$$\frac{\ell(n') = \texttt{check}(P) \quad \sigma' : n' \not\vdash P}{\sigma' : n' \rhd \sigma' : n' \lightning}$$
where $\sigma_{i^\star} = \sigma_{i^\star+1} = \sigma' : n'$. By definition 3.4, we have $\xi(n') = \{n'\} \neq \emptyset$. Now, let $i \in i^\star + 1..k-1$. Since the only rule that propagates an exception is $\rhd_{\mathit{throw}}$, the transition $\sigma_i \lightning \rhd \sigma_{i+1} \lightning$ must be on the form:
$$\frac{m \not\dashrightarrow_{\lightning}}{\sigma'' : n'' : m \lightning \rhd \sigma'' : n'' \lightning}$$
where $\sigma_i = \sigma'' : n'' : m$ and $\sigma_{i+1} = \sigma'' : n''$. By an inductive argument, it follows that $\xi(m) \neq \emptyset$. Moreover, $m \not\dashrightarrow_{\lightning}$ holds, and $n'' \longrightarrow \mu(m)$ follows by lemma A.1: by definition 3.4, we then have $\xi(n'') \supseteq \xi(m)$. Since $\xi(m) \neq \emptyset$, this implies $\xi(n'') \neq \emptyset$. $\square$

A.2 Soundness of the TP analysis

Theorem A.4. For all $\sigma \in N^*$, $P \in \mathcal{P}$:

$$\sigma \vdash P \quad \iff \quad P \in \Pi(\Gamma(\sigma))$$

Proof. We proceed by induction on the number of nodes in σ. For the base case $\sigma = []$, we have $\Pi(\Gamma([])) = \Pi(\emptyset) = \mathcal{P}$ (the last equality holds by convention), and $[] \vdash P$ by $[\vdash_1]$. For the inductive case, let $\sigma = \sigma' : n$. There two subcases, according n being privileged or not.

If $\text{Priv}(n)$, then $\Pi(\Gamma(\sigma' : n)) = \Pi(\{\text{Dom}(n)\}) = \text{Perm}(n)$. Thus, if $P \in \text{Perm}(n)$ then $\sigma' : n \vdash P$ by $[\vdash_3]$. On the other hand, $\sigma' : n \vdash P$ implies that $P \in \text{Perm}(n)$, regardless of n being privileged or not.

If $\neg\text{Priv}(n)$, then $\Pi(\Gamma(\sigma' : n)) = \Pi(\Gamma(\sigma') \cup \{\text{Dom}(n)\}) = \Pi(\Gamma(\sigma')) \cap \text{Perm}(n)$. By definition of Π and Γ, $P \in \Pi(\Gamma(\sigma))$ implies $P \in \Pi(\Gamma(\sigma'))$ and $P \in \text{Perm}(n)$. Then, $\sigma' \vdash P$ follows by the induction hypothesis, and $\sigma' : n \vdash P$ by $[\vdash_2]$. On the other hand, $\sigma' : n \vdash P$ requires that $\sigma' \vdash P$ and $P \in \text{Perm}(n)$. The induction hypothesis implies $P \in \Pi(\Gamma(\sigma'))$, and $P \in \Pi(\Gamma(\sigma' : n))$ follows by definition of Π and Γ. □

Theorem A.5. Let $\tau \models TP^{=}(G, Perm)$. Then:

$$G \rhd \sigma : n \quad \Longrightarrow \quad \exists \gamma \in \tau_{call}(n).\ \gamma = \Gamma(\sigma : n) \tag{7}$$

Proof. We prove the following, stronger, result:

$$G \rhd \sigma : n \implies \exists \gamma \in \tau_{in}(n).\ \ \gamma = \Gamma(\sigma) \cup \{\text{Dom}(n)\} \tag{8a}$$
$$G \rhd \sigma : n \lightning \implies \exists \gamma \in \tau_{catch}(n).\ \ \gamma = \Gamma(\sigma) \cup \{\text{Dom}(n)\} \tag{8b}$$

Equation (7) follows by (8a) by noticing that:

- if $\neg\text{Priv}(n)$, then $\tau_{call}(n) = \tau_{in}(n)$, and $\Gamma(\sigma : n) = \Gamma(\sigma) \cup \{\text{Dom}(n)\}$;
- if $\text{Priv}(n)$, then $\tau_{call}(n) = \{\{\text{Dom}(n)\}\}$ (because $\tau_{in}(n) \neq \emptyset$ when (8a) is true), and $\Gamma(\sigma : n) = \{\text{Dom}(n)\}$.

We prove (8a) and (8b) simultaneously by induction on the length of the derivation $[] \rhd \cdots \rhd \langle \sigma : n, x\rangle$. The base case corresponds to our single axiom:

$$\frac{\bullet\!\!\longrightarrow n}{[] \rhd [n]}$$

We show that $\gamma = \{\text{Dom}(n)\}$ satisfies (8a). Since $\bullet\!\!\longrightarrow n$, we have that $\{\text{Dom}(n)\} \in \tau_{out}(n_\varepsilon, n) \subseteq \tau_{in}(n)$. Then, $\Gamma([]) \cup \{\text{Dom}(n)\} = \emptyset \cup \{\text{Dom}(n)\} = \gamma$.

For the inductive case, we proceed by case analysis on rule used to deduce the last step of the derivation for $\langle \sigma : n, x\rangle$.

- case $[call]$:

$$\frac{\ell(n') = \texttt{call} \quad n' \longrightarrow n}{\sigma' : n' \rhd \sigma' : n' : n} \qquad \text{where } \ \sigma = \sigma' : n'$$

By the induction hypothesis (8a), we have:

$$\exists \gamma' \in \tau_{in}(n').\ \gamma' = \Gamma(\sigma') \cup \{\text{Dom}(n')\} \tag{9}$$

If $\neg\text{Priv}(n')$, we show that $\gamma = \gamma' \cup \{\text{Dom}(n)\}$ satisfies (8a). Since:

$$\tau_{in}(n) \ \supseteq \ \tau_{out}(n', n) \ = \ \{\, \gamma \cup \{\text{Dom}(n)\} \mid \gamma \in \tau_{in}(n')\,\}$$

and $\gamma' \in \tau_{in}(n')$ by (9), it follows that $\gamma \in \tau_{in}(n)$. Moreover:

$$\begin{aligned} \gamma &= \gamma' \cup \{\mathrm{Dom}(n)\} && \text{by def. } \gamma \\ &= \Gamma(\sigma') \cup \{\mathrm{Dom}(n')\} \cup \{\mathrm{Dom}(n)\} && \text{by (9)} \\ &= \Gamma(\sigma' : n') \cup \{\mathrm{Dom}(n)\} && \text{as } \neg\mathrm{Priv}(n) \end{aligned}$$

If $\mathrm{Priv}(n')$, we show that $\gamma = \{\mathrm{Dom}(n')\} \cup \{\mathrm{Dom}(n)\}$ satisfies (8a). Since:

$$\tau_{in}(n) \ \supseteq \ \tau_{out}(n', n) \ = \ \{\{\mathrm{Dom}(n')\} \cup \{\mathrm{Dom}(n)\}\}$$

then $\gamma \in \tau_{in}(n)$, and:

$$\gamma \ = \ \{\mathrm{Dom}(n')\} \cup \{\mathrm{Dom}(n)\} \ = \ \Gamma(\sigma' : n') \cup \{\mathrm{Dom}(n)\}$$

- case $[ret]$:

$$\frac{\ell(m) = \texttt{return} \quad n' \dashrightarrow n}{\sigma : n' : m \rhd \sigma : n}$$

By lemma A.1, any trace leading to $\sigma : n' : m$ is on the form:

$$[] \ \rhd \ \cdots \ \rhd \ \sigma : n' \ \rhd \ \cdots \ \rhd \ \sigma : n' : m$$

where $\ell(n') = \texttt{call}$, $n' \longrightarrow \mu(m)$, and $m \in \rho(n')$ by lemma A.2. Therefore, we can apply the induction hypothesis on $\sigma : n'$ as well as on $\sigma : n' : m$, obtaining:

$$\exists \gamma' \in \tau_{in}(n').\ \gamma' = \Gamma(\sigma) \cup \{\mathrm{Dom}(n')\} \tag{10a}$$

$$\exists \gamma'' \in \tau_{in}(m).\ \gamma'' = \Gamma(\sigma : n') \cup \{\mathrm{Dom}(m)\} \tag{10b}$$

We show that $\gamma = \gamma'$ satisfies (8a). First, notice that $\gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\}$, because $\mathrm{Dom}(n) = \mathrm{Dom}(n')$ by constraint 4. Second, we prove that $\gamma \in \tau_{in}(n)$. If $\neg\mathrm{Priv}(n')$, then:

$$\tau_{in}(n) \ \supseteq \ \tau_{trans}(n') \ = \ \{\, \gamma \in \tau_{in}(n') \mid \mathrm{Trans}(n', \gamma) \,\}$$

To show that $\mathrm{Trans}(n', \gamma')$, i.e.:

$$\exists m \in \rho(n'),\ \gamma'' \in \tau_{in}(m).\ \gamma'' = \gamma' \cup \{\mathrm{Dom}(m)\}$$

observe that:

$$\begin{aligned} \gamma'' &= \Gamma(\sigma : n') \cup \{\mathrm{Dom}(m)\} && \text{by (10b)} \\ &= \Gamma(\sigma) \cup \{\mathrm{Dom}(n')\} \cup \{\mathrm{Dom}(m)\} && \text{as } \neg\mathrm{Priv}(n') \\ &= \gamma' \cup \{\mathrm{Dom}(m)\} && \text{by (10a)} \end{aligned}$$

In the case $\mathrm{Priv}(n')$, we have:

$$\tau_{in}(n) \ \supseteq \ \tau_{trans}(n') \ = \ \{\, \gamma \in \tau_{in}(n') \mid \mathrm{Trans}(n', \{\mathrm{Dom}(n')\}) \,\}$$

To show that $\mathrm{Trans}(n', \{\mathrm{Dom}(n')\})$, i.e.:

$$\exists m \in \rho(n'),\ \gamma'' \in \tau_{in}(m).\ \gamma'' = \{\mathrm{Dom}(n')\} \cup \{\mathrm{Dom}(m)\}$$

observe that:

$$\begin{aligned} \gamma'' &= \Gamma(\sigma : n') \cup \{\mathrm{Dom}(m)\} && \text{by (10b)} \\ &= \{\mathrm{Dom}(n')\} \cup \{\mathrm{Dom}(m)\} && \text{as } \mathrm{Priv}(n') \end{aligned}$$

- case [*pass*]:

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

By the induction hypothesis (8a), we have:

$$\exists \gamma' \in \tau_{in}(n').\ \gamma' = \Gamma(\sigma) \cup \{\mathrm{Dom}(n')\} \tag{11}$$

We show that $\gamma = \gamma'$ satisfies (8a). First, notice that $\gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\}$ by constraint 4. Second, we prove that $\gamma \in \tau_{in}(n)$. Since $n' \dashrightarrow n$, we have:

$$\tau_{in}(n) \ \supseteq\ \tau_{trans}(n') \ =\ \{\, \gamma \in \tau_{in}(n') \mid P \in \Pi(\gamma) \,\}$$

Since $\gamma = \gamma'$, by (11) it follows that $\gamma \in \tau_{in}(n')$. By constraint 5, we have $\gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n')\} = \Gamma(\sigma : n')$. Since $\sigma : n' \vdash P$ (premise of the $\rhd_{pass}$ rule), theorem A.4 implies that $P \in \Pi(\gamma)$. Then, $\gamma \in \tau_{in}(n)$.

- case [*fail*]:

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

By the induction hypothesis (8a), we have:

$$\exists \gamma' \in \tau_{in}(n).\ \gamma' = \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\} \tag{12}$$

We show that $\gamma = \gamma'$ satisfies (8b). By definition of TP_{catch}, we have:

$$\tau_{catch}(n) \ =\ \{\, \gamma \in \tau_{in}(n) \mid P \notin \Pi(\gamma) \,\}$$

By (12) and constraint 5, it follows that $\gamma = \Gamma(\sigma : n) \in \tau_{in}(n)$. Since $\sigma : n \nvdash P$ (premise of the $\rhd_{fail}$ rule), theorem A.4 implies $P \notin \Pi(\gamma)$. Then, $\gamma \in \tau_{catch}(n)$.

- case [*catch*]:

$$\frac{n' \dashrightarrow_{\lightning} n}{\sigma : n'\lightning \rhd \sigma : n}$$

By the induction hypothesis (8b), we have:

$$\exists \gamma' \in \tau_{catch}(n').\ \gamma' = \Gamma(\sigma) \cup \{\mathrm{Dom}(n')\} \tag{13}$$

We show that $\gamma = \gamma'$ satisfies (8a). Since $n' \dashrightarrow_{\lightning} n$, we have:

$$\tau_{in}(n) \ \supseteq\ \tau_{out}(n', n) \ =\ \tau_{catch}(n')$$

Then, $\gamma \in \tau_{in}(n)$ follows by (13), and $\gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\}$ by constraint 4.

- case [*throw*]:

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

By lemma A.1, any derivation of $\sigma : n : m\lightning$ is on the form:

$$[] \ \rhd\ \cdots\ \rhd\ \sigma : n\ \rhd\ \cdots\ \rhd\ \sigma : n : m\lightning$$

We apply the induction hypothesis on $\sigma : n$ as well as on $\sigma : n : m\lightning$, obtaining:

$$\exists \gamma' \in \tau_{in}(n).\ \gamma' = \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\} \tag{14a}$$
$$\exists \gamma'' \in \tau_{catch}(m).\ \gamma'' = \Gamma(\sigma : n) \cup \{\mathrm{Dom}(m)\} \tag{14b}$$

By definition 3.4, $m \in \xi_1(n)$: indeed, $n \longrightarrow \mu(m)$ follows by lemma A.1, $m \not\dashrightarrow_{\lightning}$ is a premise of the $\rhd_{throw}$ rule, and $\xi(m) \neq \emptyset$ by lemma A.3. We show that $\gamma = \gamma'$ satisfies (8b). There two subcases, according n being privileged or not.

If $\neg\mathrm{Priv}(n)$, then:

$$\tau_{catch}(n) \quad = \quad \{\, \gamma \in \tau_{in}(n) \mid \mathrm{Catch}(n, \gamma) \,\}$$

where $\mathrm{Catch}(n, \gamma)$ holds if:

$$\exists m \in \xi_1(n),\ \gamma'' \in \tau_{catch}(m).\ \gamma'' = \gamma' \cup \{\mathrm{Dom}(m)\}$$

Since $m \in \xi_1(n)$, we have:

$$\begin{aligned} \gamma'' &= \Gamma(\sigma : n) \cup \{\mathrm{Dom}(m)\} && \text{by (14b)} \\ &= \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\} \cup \{\mathrm{Dom}(m)\} && \text{as } \neg\mathrm{Priv}(n) \\ &= \gamma' \cup \{\mathrm{Dom}(m)\} && \text{by (14a)} \end{aligned}$$

Then, $\gamma \in \tau_{catch}(n)$. If $\mathrm{Priv}(n)$, then:

$$\tau_{catch}(n) \quad = \quad \{\, \gamma \in \tau_{in}(n) \mid \mathrm{Catch}(n, \{\mathrm{Dom}(n)\}) \,\}$$

To show $\mathrm{Catch}(n, \{\mathrm{Dom}(n)\})$, observe that:

$$\begin{aligned} \gamma'' &= \Gamma(\sigma : n) \cup \{\mathrm{Dom}(m)\} && \text{by (14b)} \\ &= \{\mathrm{Dom}(n)\} \cup \{\mathrm{Dom}(m)\} && \text{as } \mathrm{Priv}(n) \end{aligned}$$

□

A.3 Completeness of the TP analysis

Theorem A.6. Let $\sqsubseteq$ be the following partial order:

$$\tau \sqsubseteq \tau' \quad \overset{def}{=} \quad \forall n \in N.\ \tau_{in}(n) \subseteq \tau'_{in}(n) \tag{15}$$

Then, there exists a unique $\tau \models TP^{=}(G, \mathit{Perm})$ minimal w.r.t. $\sqsubseteq$, and:

$$\gamma \in \tau_{call}(n) \quad \Longrightarrow \quad \exists \sigma.\ G \triangleright \sigma : n \ \wedge\ \gamma = \Gamma(\sigma : n) \tag{16}$$

Proof. The proof is split in four parts:

1. lemma A.7 formalizes the intuitive concept that, in our model, reachability is only a matter of access control. This enables us to prove the following item.
2. lemma A.8, showing that:

$$\gamma \in \mathsf{call}[n] \quad \Longrightarrow \quad \exists \sigma.\ G \triangleright \sigma : n \ \wedge\ \gamma = \Gamma(\sigma : n) \tag{17}$$

   is indeed an invariant for the worklist-iteration algorithm.
3. lemma A.9 shows that, whenever the algorithm terminates, it computes a solution to the TP analysis. Moreover, the solution is minimal w.r.t. (15).
4. finally, lemma A.10 shows that the algorithm always terminates.

□

Lemma A.7. Let $G \triangleright \sigma : n$, $G \triangleright \langle \sigma' : m, x \rangle$, and $n = \varepsilon(\mu(m))$. Then:

$$\Gamma(\sigma) \cup \{\mathrm{Dom}(n)\} \ = \ \Gamma(\sigma') \cup \{\mathrm{Dom}(m)\} \quad \Longrightarrow \quad G \triangleright \langle \sigma : m, x \rangle$$

Proof. Since $n = \varepsilon(\mu(m))$, by constraint 3 and lemma A.1, it follows that any trace leading to $\langle \sigma' : m, x \rangle$ is on the form:

$$[] = \langle \sigma_0, x_0 \rangle \ \triangleright\ \cdots\ \triangleright\ \langle \sigma_h, x_h \rangle = \sigma' : n \ \triangleright\ \cdots\ \triangleright\ \langle \sigma_k, x_k \rangle = \langle \sigma' : m, x \rangle$$

We can always determine h in such a way that, for each $i \in h..k$:

```
TP-ANALYSIS($G$)
W $\leftarrow$ CONS(NIL, $n_\varepsilon$)
for each $n$ in $N$ do
    in$[n]$ $\leftarrow$ call$[n]$ $\leftarrow$ trans$[n]$ $\leftarrow$ catch$[n]$ $\leftarrow \emptyset$
for each $(n, n')$ in $E$ do
    out$[n, n']$ $\leftarrow \emptyset$
while W $\neq$ NIL do
    $n$ $\leftarrow$ HEAD(W)
    W $\leftarrow$ TAIL(W)
    old-catch$[n]$ $\leftarrow$ catch$[n]$
    switch
      case $\ell(n) = \texttt{call}$ :
            if Priv$(n)$
              then call$[n]$  $\leftarrow \{\{\mathrm{Dom}(n)\}\}$
                   trans$[n]$ $\leftarrow \{\gamma \in$ in$[n]$ $\mid \exists m \in \rho(n).\ \{\mathrm{Dom}(n)\} \cup \{\mathrm{Dom}(m)\} \in$ in$[m]\}$
                   catch$[n]$ $\leftarrow \{\gamma \in$ in$[n]$ $\mid \exists m \in \xi_1(n).\{\mathrm{Dom}(n)\} \cup \{\mathrm{Dom}(m)\} \in$ catch$[m]\}$
              else call$[n]$  $\leftarrow$ in$[n]$
                   trans$[n]$ $\leftarrow \{\gamma \in$ in$[n]$ $\mid \exists m \in \rho(n).\ \gamma \cup \{\mathrm{Dom}(m)\} \in$ in$[m]\}$
                   catch$[n]$ $\leftarrow \{\gamma \in$ in$[n]$ $\mid \exists m \in \xi_1(n).\gamma \cup \{\mathrm{Dom}(m)\} \in$ catch$[m]\}$
      case $\ell(n) = \texttt{check}(P)$ :
            trans$[n]$ $\leftarrow \{\gamma \in$ in$[n]$ $\mid P \in \Pi(\gamma)\}$
            catch$[n]$ $\leftarrow \{\gamma \in$ in$[n]$ $\mid P \notin \Pi(\gamma)\}$
    for each $(n, n')$ in $E$ do
        switch
          case $\bullet\!\!\longrightarrow n'$ :
                out$[n, n']$ $\leftarrow \{\{\mathrm{Dom}(n)\}\}$
          case $n \longrightarrow n'$ :
                out$[n, n']$ $\leftarrow \{\gamma \cup \{\mathrm{Dom}(n')\} \mid \gamma \in$ call$[n]\}$
          case $n \dashrightarrow n'$ :
                out$[n, n']$ $\leftarrow$ trans$[n]$
          case $n \dashrightarrow_{\lightning} n'$ :
                out$[n, n']$ $\leftarrow$ catch$[n]$
        if out$[n, n']$ $\not\subseteq$ in$[n']$
          then in$[n']$ $\leftarrow$ in$[n']$ $\cup$ out$[n, n']$
               W $\leftarrow$ CONS(W, $n'$)
    for each $m$ such that $n \in \rho(m)$ do
        if in$[m] \neq \emptyset$
          then W $\leftarrow$ CONS(W, $m$)
    for each $m$ such that $n \in \xi_1(m)$ do
        if in$[m] \neq \emptyset$ and catch$[n]$ $\not\subseteq$ old-catch$[n]$
          then W $\leftarrow$ CONS(W, $m$)
```

Table 5: Worklist-Iteration for the Trace Permissions Analysis

(a) $\sigma_i = \sigma' : \sigma'_i$ for some $\sigma'_i \neq []$, and

(b) the bottommost node of σ'_i belongs to $\mu(m)$.

This is because there always exists a trace leading to $\langle \sigma' : m, x \rangle$ that never returns from $\mu(m)$. Therefore, if we prove that, for any $i \in h..k-1$:

$$\langle \sigma' : \sigma'_i, x_i \rangle \ \rhd\ \langle \sigma' : \sigma'_{i+1}, x_{i+1} \rangle \quad \Longleftrightarrow \quad \langle \sigma : \sigma'_i, x_i \rangle \ \rhd\ \langle \sigma : \sigma'_{i+1}, x_{i+1} \rangle$$

then we can conclude $G \rhd \langle \sigma : m, x \rangle$. The proof is by cases on the rule applied to deduce the transition $\langle \sigma' : \sigma'_i, x_i \rangle \rhd \langle \sigma' : \sigma'_{i+1}, x_{i+1} \rangle$. We work out only the most relevant cases. First, consider the $\rhd_{pass}$ rule, which instances to:

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

where $\sigma'_i = \sigma''_i : n'$ and $\sigma'_{i+1} = \sigma''_i : m'$. Note that $\sigma : \sigma'_i \rhd \sigma : \sigma'_{i+1}$ holds if $\sigma : \sigma'_i \vdash P$. Let $\sigma'_i = [n_0, \ldots, n_h]$, where $n_h = n'$ and $n_0 \in \mu(m)$ by the hypotheses (a) and (b).

Assume first there are no privileged nodes in σ'_i. Then, using the facts that n_0, n, m are in the same method (so they are in the same protection domain by constraint 4), and by hypothesis $\Gamma(\sigma) \cup \{\mathrm{Dom}(n)\} = \Gamma(\sigma') \cup \{\mathrm{Dom}(m)\}$, we obtain:

$$\begin{aligned}\Gamma(\sigma' : \sigma'_i) \;&=\; \Gamma(\sigma' : \sigma'_i) \cup \{\mathrm{Dom}(m)\} \;=\; \Gamma(\sigma') \cup \{\mathrm{Dom}(m)\} \cup \Gamma(\sigma'_i) \\ &=\; \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\} \cup \Gamma(\sigma'_i) \;=\; \Gamma(\sigma : \sigma'_i) \cup \{\mathrm{Dom}(n)\}\end{aligned}$$

Otherwise, let $i^\star$ be the index of the topmost privileged node in σ'_i. Then:

$$\Gamma(\sigma' : \sigma'_i) \;=\; \bigcup_{j \in i^\star..h} \mathrm{Dom}(n_j) \;=\; \Gamma(\sigma : \sigma'_i)$$

In both cases, we have shown $\Gamma(\sigma' : \sigma'_i) = \Gamma(\sigma : \sigma'_i)$. Since $\sigma' : \sigma'_i \vdash P$, by theorem A.4 it follows that $\sigma : \sigma'_i \vdash P$. So, the $\rhd_{pass}$ rule gives $\langle \sigma' : \sigma'_i, x_i \rangle \rhd \langle \sigma' : \sigma'_{i+1}, x_{i+1} \rangle$.

Consider now the $\rhd_{fail}$ rule, which instances to:

$$\frac{\ell(n') = \mathtt{check}(P) \quad \sigma' : \sigma''_i : n' \not\vdash P}{\sigma' : \sigma''_i : n' \rhd \sigma' : \sigma''_i : n' \lightning}$$

where $\sigma'_i = \sigma'_{i+1} = \sigma''_i : n'$. As in the previous case, $\Gamma(\sigma' : \sigma'_i) = \Gamma(\sigma : \sigma'_i)$. It follows that $\sigma : \sigma'_i \not\vdash P$, and the $\rhd_{fail}$ rule gives $\sigma : \sigma'_i \rhd \sigma : \sigma'_{i+1} \lightning$.

For the remaining rules, the transition $\sigma' : \sigma'_i \rhd \sigma' : \sigma'_{i+1}$ depends only on the edges of G and on the topmost node of σ'_i (for rule $\rhd_{ret}$, also on the next-topmost node). Therefore, if $\langle \sigma' : \sigma'_i, x_i \rangle \rhd \langle \sigma' : \sigma'_{i+1}, x_{i+1} \rangle$, then also $\langle \sigma : \sigma'_i, x_i \rangle \rhd \langle \sigma : \sigma'_{i+1}, x_{i+1} \rangle$. □

Lemma A.8. For each node $n \in N$, the statements:

$$\gamma \in \mathsf{in}[n] \implies \exists \sigma.\, G \rhd \sigma : n \;\wedge\; \gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\} \tag{18a}$$
$$\gamma \in \mathsf{catch}[n] \implies \exists \sigma.\, G \rhd \sigma : n\lightning \;\wedge\; \gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\} \tag{18b}$$

are invariant for the worklist-iteration algorithm.

Proof. We first prove that (17) follows by (18a). Let $\gamma \in \mathsf{call}[n]$. Then:

- if $\neg\mathrm{Priv}(n)$, then $\gamma \in \mathsf{in}[n]$ by the assignment at line 16. By (18a), there exists some σ such that $G \rhd \sigma : n$ and $\gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\} = \Gamma(\sigma : n)$.
- if $\mathrm{Priv}(n)$, then $\gamma = \{\mathrm{Dom}(n)\}$ by the assignment at line 13. By (18a), there exists some σ such that $G \rhd \sigma : n$ and $\gamma = \{\mathrm{Dom}(n)\} = \Gamma(\sigma : n)$.

The invariants hold when the **while** loop at lines 6-40 is entered: indeed, after the initialization loop at lines 2-3, $\mathsf{in}[n] = \mathsf{catch}[n] = \emptyset$ for each $n \in N$.

Next, assume (18a) and (18b) are satisfied for all iterations up to the i-th. We show that the invariants are preserved after the $(i+1)$-th iteration.

Consider (18b) first: inside the **switch** statement at lines 10-21, $\mathsf{catch}[n]$ is updated when n is a call or a check.

If $\ell(n) = \mathtt{check}(P)$ then, by the assignment at line 21:

$$\mathsf{catch}[n] \;=\; \{\, \gamma \in \mathsf{in}[n] \mid P \notin \Pi(\gamma) \,\}$$

Let $\gamma \in \mathsf{catch}[n]$. Since $\gamma \in \mathsf{in}[n]$ and (18a) holds at the i-th iteration, then:

$$\exists \sigma.\, G \rhd \sigma : n \;\wedge\; \gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\}$$

Since check nodes cannot be privileged (constraint 5), $\gamma = \Gamma(\sigma : n)$, and $\sigma : n \not\Vdash P$ follows by theorem A.4. Then, by the $\rhd_{fail}$ rule:

$$\frac{\ell(n) = \texttt{check}(P) \quad \sigma : n \not\Vdash P}{\sigma : n \rhd \sigma : n\lightning}$$

In conclusion, we have shown that $G \rhd \sigma : n\lightning$ and $\gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\}$. So, (18b) is preserved by the assignment at line 21.

If $\ell(n) = \texttt{call}$, we have two subcases, depending on n being privileged or not.
If $\neg\mathrm{Priv}(n)$, then, by the assignment at line 18:

$$\mathsf{catch}[n] \quad = \quad \{\, \gamma \in \mathsf{in}[n] \mid \exists m \in \xi_1(n).\ \gamma \cup \{\mathrm{Dom}(m)\} \in \mathsf{catch}[m] \,\}$$

Let $\gamma \in \mathsf{catch}[n]$: then, there is an $m \in \xi_1(n)$ and a $\gamma' \in \mathsf{catch}[m]$ such that $\gamma' = \gamma \cup \{\mathrm{Dom}(m)\}$. Since $\gamma \in \mathsf{in}[n]$, then $\gamma' \in \mathsf{catch}[m]$; moreover, since both the invariants (18a) and (18b) hold at the i-th iteration, we have:

$$\exists \sigma.\ G \rhd \sigma : n \ \wedge\ \gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\} \tag{19a}$$
$$\exists \sigma'.\ G \rhd \sigma' : m\lightning \ \wedge\ \gamma' = \Gamma(\sigma') \cup \{\mathrm{Dom}(m)\} \tag{19b}$$

Let $n' = \varepsilon(\mu(m))$. Since $m \in \xi_1(n)$, by definition 3.4 it follows that $n \longrightarrow \mu(m)$ and $m \not\dashrightarrow$. Then, by the $\rhd_{call}$ rule, we have:

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n'}{\sigma : n \rhd \sigma : n : n'}$$

Moreover, we have that:

$$\begin{aligned}
\Gamma(\sigma : n) \cup \{\mathrm{Dom}(n')\} &= \Gamma(\sigma : n) \cup \{\mathrm{Dom}(m)\} && \text{as } \mu(n') = \mu(m)\\
&= \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\} \cup \{\mathrm{Dom}(m)\} && \text{as } \neg\mathrm{Priv}(n)\\
&= \gamma \cup \{\mathrm{Dom}(m)\} && \text{by (19a)}\\
&= \gamma' && \text{by def. } \gamma'\\
&= \Gamma(\sigma') \cup \{\mathrm{Dom}(m)\} && \text{by (19b)}
\end{aligned}$$

This enables us to apply lemma A.7 to deduce $G \rhd \sigma : n : m\lightning$. Then, by $\rhd_{throw}$:

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

If $\mathrm{Priv}(n)$, then, by the assignment at line 15:

$$\mathsf{catch}[n] \quad = \quad \{\, \gamma \in \mathsf{in}[n] \mid \exists m \in \xi_1(n).\ \{\mathrm{Dom}(n)\} \cup \{\mathrm{Dom}(m)\} \in \mathsf{catch}[m] \,\}$$

Let $\gamma \in \mathsf{catch}[n]$. Then, there exist $m \in \xi_1(n)$ and $\gamma' \in \mathsf{catch}[m]$ such that $\gamma' = \{\mathrm{Dom}(n)\} \cup \{\mathrm{Dom}(m)\}$. Similarly to the case $\neg\mathrm{Priv}(n)$, we have $G \rhd \sigma : n : n'$. Then:

$$\Gamma(\sigma : n) \cup \{\mathrm{Dom}(n')\} \quad = \quad \{\mathrm{Dom}(n)\} \cup \{\mathrm{Dom}(n')\} \quad = \quad \gamma'$$

and $G \rhd \sigma : n\lightning$ follows by $\rhd_{throw}$. This concludes the proof of (18b).

To prove that also the invariant (18a) holds, consider the **for** loop at lines 22-34. At each iteration, $\mathsf{out}[n, n']$ is computed for some $(n, n') \in E$. When the condition of the **if** statement at line 32 is false, the array in is left unchanged. Otherwise, only the value of $\mathsf{in}[n']$ is updated, and we have:

$$\mathsf{in}[n'] \quad \leftarrow \quad \mathsf{in}[n'] \cup \mathsf{out}[n, n']$$

Then, it suffices to show that (18a) holds for $\gamma \in \mathsf{out}[n, n']$. There are only the following cases, according to the kind of the edge (n, n'):

- case [*entry*]: if $\bullet\!\!\longrightarrow n'$, then, after the assignment at line 25:

$$\mathsf{out}[n, n'] \;=\; \{\{\mathrm{Dom}(n')\}\}$$

Let $\gamma = \{\mathrm{Dom}(n')\}$ and $\sigma = []$. Then, $G \rhd \sigma : n'$ follows by rule $\rhd_{entry}$, and $\gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n')\}$.

- case [*call*]: if $n \longrightarrow n'$, then, by constraints 1 and 2, it must be $\ell(n) = \mathtt{call}$. Being n privileged or not, since it has been extracted from the worklist, the guards at lines 32, 36, 39 make sure that $\mathsf{in}[n] \neq \emptyset$. Since (18a) holds at the i-th iteration of the **while** loop, it follows that $G \rhd \sigma' : n$ for some σ'. Let $\sigma = \sigma' : n$. Then, $G \rhd \sigma : n'$ follows by:

$$\frac{\ell(n) = \mathtt{call} \quad n \longrightarrow n'}{\sigma' : n \rhd \sigma' : n : n'}$$

By the assignment at line 27, we have:

$$\mathsf{out}[n, n'] \;=\; \{\, \gamma \cup \{\mathrm{Dom}(n')\} \mid \gamma \in \mathsf{call}[n] \,\}$$

If $\neg\mathrm{Priv}(n)$, then $\mathsf{call}[n] = \mathsf{in}[n]$ by the assignment at line 16. Then, each $\gamma \in \mathsf{call}[n]$ in on the form $\gamma = \gamma' \cup \{\mathrm{Dom}(n')\}$, for some $\gamma' \in \mathsf{in}[n]$. Since (18a) did hold when γ' was inserted in $\mathsf{in}[n]$, we have:

$$G \rhd \sigma' : n \;\;\wedge\;\; \gamma' = \Gamma(\sigma') \cup \{\mathrm{Dom}(n)\} \tag{20}$$

Then, (18a) is preserved, because:

$$\begin{aligned}
\gamma &= \gamma' \cup \{\mathrm{Dom}(n')\} && \text{by def. } \gamma \\
&= \Gamma(\sigma') \cup \{\mathrm{Dom}(n)\} \cup \{\mathrm{Dom}(n')\} && \text{by (20)} \\
&= \Gamma(\sigma' : n) \cup \{\mathrm{Dom}(n')\} && \text{as } \neg\mathrm{Priv}(n) \\
&= \Gamma(\sigma) \cup \{\mathrm{Dom}(n')\} && \text{by def. } \sigma
\end{aligned}$$

If $\mathrm{Priv}(n)$, then, by the assignment at line 13, $\mathsf{call}[n] = \{\{\mathrm{Dom}(n)\}\}$. Let $\gamma = \{\mathrm{Dom}(n)\} \cup \{\mathrm{Dom}(n')\}$. Then:

$$\Gamma(\sigma' : n) \cup \{\mathrm{Dom}(n')\} \;=\; \{\mathrm{Dom}(n)\} \cup \{\mathrm{Dom}(n')\} \;=\; \gamma$$

- case [*transfer*]: if $n \dashrightarrow n'$, then, by the assignment at line 29:

$$\mathsf{out}[n, n'] \;=\; \mathsf{trans}[n]$$

By constraint 2, n cannot be a return node, so we have to consider two subcases, i.e. that n is a check or a call node. If $\ell(n) = \mathtt{check}(P)$, then, by the assignment at line 20, we have:

$$\mathsf{trans}[n] \;=\; \{\, \gamma \in \mathsf{in}[n] \mid P \in \Pi(\gamma) \,\}$$

Take $\gamma \in \mathsf{in}[n]$ such that $P \in \Pi(\gamma)$. By (18a) and by constraint 5, we have:

$$\exists \sigma.\, G \rhd \sigma : n \;\;\wedge\;\; \gamma = \Gamma(\sigma : n) \tag{21}$$

Theorem A.4 implies that $\sigma : n \vdash P$. Therefore, $G \rhd \sigma : n'$ follows by the rule:

$$\frac{\ell(n) = \mathtt{check}(P) \quad \sigma : n \vdash P \quad n \dashrightarrow n'}{\sigma : n \rhd \sigma : n'}$$

Since $\mu(n) = \mu(n')$, constraint 4 implies that $\gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n')\}$.

If $\ell(n) = \texttt{call}$, we must consider two subcases, depending on n being privileged or not. If $\neg\mathrm{Priv}(n)$, by the assignment at line 17:

$$\mathsf{trans}[n] \quad = \quad \{\, \gamma \in \mathsf{in}[n] \mid \exists m \in \rho(n).\ \gamma \cup \{\mathrm{Dom}(m)\} \in \mathsf{in}[m] \,\}$$

Let $\gamma \in \mathsf{trans}[n]$: then, there exist $m \in \rho(n)$ and $\gamma' \in \mathsf{in}[m]$ such that $\gamma' = \gamma \cup \{\mathrm{Dom}(m)\}$. By applying (18a) for n as well as for m:

$$\exists\sigma.\ G \rhd \sigma : n \quad\wedge\quad \gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\} \tag{22a}$$
$$\exists\sigma'.\ G \rhd \sigma' : m \quad\wedge\quad \gamma' = \Gamma(\sigma') \cup \{\mathrm{Dom}(m)\} \tag{22b}$$

Observe that, since $m \in \rho(n)$, it must be $n \longrightarrow m'$ for $m' = \varepsilon(\mu(m))$. Then, $G \rhd \sigma : n : m'$ follows by the rule:

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow m'}{\sigma : n \rhd \sigma : n : m'}$$

Moreover, we have that:

$$\begin{aligned}
\Gamma(\sigma : n) \cup \{\mathrm{Dom}(m')\} &= \Gamma(\sigma : n) \cup \{\mathrm{Dom}(m)\} && \text{as } \mu(m') = \mu(m)\\
&= \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\} \cup \{\mathrm{Dom}(m)\} && \text{as } \neg\mathrm{Priv}(n)\\
&= \gamma \cup \{\mathrm{Dom}(m)\} && \text{by (22a)}\\
&= \gamma' && \text{by def. } \gamma'\\
&= \Gamma(\sigma') \cup \{\mathrm{Dom}(m)\} && \text{by (22b)}
\end{aligned}$$

This enables us to apply lemma A.7 to deduce $G \rhd \sigma : n : m$. Then:

$$\frac{\ell(m) = \texttt{return} \quad n \dashrightarrow n'}{\sigma : n : m \rhd \sigma : n'}$$

Again, $\gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n')\}$ follows because $\mu(n) = \mu(n')$.

If $\mathrm{Priv}(n)$, then $\mathsf{out}[n, n'] = \mathsf{in}[n]$ and $\{\mathrm{Dom}(n)\} \cup \{\mathrm{Dom}(m)\} \in \mathsf{in}[m]$ for some $m \in \rho(n)$, because line 32 ensures that $\mathsf{out}[n, n'] \neq \emptyset$. Let $\gamma \in \mathsf{in}[n]$ and $\gamma' = \{\mathrm{Dom}(n)\} \cup \{\mathrm{Dom}(m)\}$. Then, (22a) and (22b) are still valid, and lemma A.7 can be applied by noticing that:

$$\begin{aligned}
\Gamma(\sigma : n) \cup \{\mathrm{Dom}(m')\} &= \Gamma(\sigma : n) \cup \{\mathrm{Dom}(m)\} && \text{as } \mu(m') = \mu(m)\\
&= \{\mathrm{Dom}(n)\} \cup \{\mathrm{Dom}(m)\} && \text{as } \mathrm{Priv}(n)\\
&= \gamma' && \text{by def. } \gamma'\\
&= \Gamma(\sigma') \cup \{\mathrm{Dom}(m)\} && \text{by (22b)}
\end{aligned}$$

Therefore, by arguments similar to those used for the case $\neg\mathrm{Priv}(n)$, we find that $G \rhd \sigma : n'$ and that $\gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n')\}$.

- case [*catch*]: if $n \dashrightarrow_{\lightning} n'$, then, by the assignment at line 31:

  $$\mathsf{out}[n, n'] \quad = \quad \mathsf{catch}[n]$$

  Let $\gamma \in \mathsf{catch}[n]$. Since we have already proved that (18b) is valid at the $(i{+}1)$-th iteration of the **while** loop, we have:

  $$\exists\sigma.\ G \rhd \sigma : n\lightning \quad\wedge\quad \gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n)\}$$

  Then, $G \rhd \sigma : n'$ follows by the rule:

  $$\frac{n \dashrightarrow_{\lightning} n'}{\sigma : n\lightning \rhd \sigma : n'}$$

  and $\gamma = \Gamma(\sigma) \cup \{\mathrm{Dom}(n')\}$ immediately follows from $\mu(n') = \mu(n)$.

$\square$

Lemma A.9. If the worklist-iteration algorithm terminates, then it computes the *least* solution to the TP analysis. More formally, if we define:

$$\mathsf{TP} = \langle \mathsf{in}, \mathsf{out}, \mathsf{trans}, \mathsf{call}, \mathsf{catch} \rangle$$

then $\mathsf{TP} \models TP^{=}(G)$, and TP is minimal w.r.t. the partial order (15), that is:

$$\tau \models TP^{=}(G) \quad \Longrightarrow \quad \forall n \in N.\ \mathsf{in}[n] \subseteq \tau_{in}(n)$$

Proof. We start by showing the minimality of TP w.r.t. (15). Consider an arbitrary solution $\tau \models TP^{=}(G)$. We prove that:

$$\mathsf{in}[n] \subseteq \tau_{in}(n) \tag{23a}$$
$$\mathsf{out}[m,n] \subseteq \tau_{out}(m,n) \tag{23b}$$
$$\mathsf{call}[n] \subseteq \tau_{call}(n) \tag{23c}$$
$$\mathsf{trans}[n] \subseteq \tau_{trans}(n) \tag{23d}$$
$$\mathsf{catch}[n] \subseteq \tau_{catch}(n) \tag{23e}$$

is invariant for the **while** loop, for each $n \in N$ and $(m,n) \in E$. Define in^0 as the value of the array in at the entry of the **while** loop, and in^i as the value of the array after the assignment at line 33 of the i-th iteration. Similarly for the arrays out, call, trans, catch and for the the worklist W.

After the initialization **for** loops at lines 2-3 and 4-5, we have:

$$\begin{array}{rcccl} \mathsf{in}^0[n] & = & \emptyset & \subseteq & \tau_{in}(n) \\ \mathsf{out}^0[m,n] & = & \emptyset & \subseteq & \tau_{out}(m,n) \\ \mathsf{call}^0[n] & = & \emptyset & \subseteq & \tau_{call}(n) \\ \mathsf{trans}^0[n] & = & \emptyset & \subseteq & \tau_{trans}(n) \\ \mathsf{catch}^0[n] & = & \emptyset & \subseteq & \tau_{catch}(n) \end{array}$$

for each $n \in N$ and $(m,n) \in E$, so all equations (23) are trivially satisfied.

Next, we show that the inclusion relations are preserved after each iteration of the **while** loop. Let $i > 0$, and assume the equations (23) are satisfied for all iterations up to the i-th, for each $n \in N$ and $(m,n) \in E$.

We first prove the invariants (23c)-(23e). At the $(i+1)$-th iteration of the **while**, a node n is extracted from the worklist (lines 7-8), and the values $\mathsf{call}^{i+1}[n]$, $\mathsf{trans}^{i+1}[n]$ and $\mathsf{catch}^{i+1}[n]$ can be updated, according to one of the following cases:

- case [*call*]: if $\ell(n) = \texttt{call}$, we have two cases, depending on m being privileged or not. If $\neg\mathrm{Priv}(n)$, then, by the assignments at lines 16-18:

$$\begin{array}{rcl} \mathsf{call}^{i+1}[n] & \leftarrow & \mathsf{in}^i[n] \\ & \subseteq & \tau_{in}(n) \\ & = & \tau_{call}(n) \end{array}$$

$$\begin{array}{rcl} \mathsf{trans}^{i+1}[n] & \leftarrow & \{\, \gamma \in \mathsf{in}^i[n] \mid \exists m \in \rho(n).\ \gamma \cup \{\mathrm{Dom}(m)\} \in \mathsf{in}^i[m] \,\} \\ & \subseteq & \{\, \gamma \in \tau_{in}(n) \mid \exists m \in \rho(n).\ \gamma \cup \{\mathrm{Dom}(m)\} \in \tau_{in}(m) \,\} \\ & = & \tau_{trans}(n) \end{array}$$

$$\begin{array}{rcl} \mathsf{catch}^{i+1}[n] & \leftarrow & \{\, \gamma \in \mathsf{in}^i[n] \mid \exists m \in \xi_1(n).\ \gamma \cup \{\mathrm{Dom}(m)\} \in \mathsf{catch}^i[m] \,\} \\ & \subseteq & \{\, \gamma \in \tau_{in}(n) \mid \exists m \in \xi_1(n).\ \gamma \cup \{\mathrm{Dom}(m)\} \in \tau_{catch}(m) \,\} \\ & = & \tau_{catch}(n) \end{array}$$

If $\mathrm{Priv}(n)$, the assignment at line 13 reads as:

$$\mathsf{call}^{i+1}[n] \quad \leftarrow \quad \{\{\mathrm{Dom}(n)\}\}$$

Indeed, the **if** statements at lines 32, 36 and 39 ensure that an assignment to $\mathsf{in}[n]$ took place before n was inserted in the worklist, say at the j-th iteration of the **while**, with $j < i$. Therefore, $\mathsf{in}^j[n] \neq \emptyset$, and $\mathsf{in}^j[n] \subseteq \mathsf{in}^i[n]$ because of monotonicity. By hypothesis $\mathsf{in}^i[n] \subseteq \tau_{in}(n)$, it follows $\tau_{in}(n) \neq \emptyset$, and thus:

$$\mathsf{call}^{i+1}[n] \quad = \quad \{\{\mathrm{Dom}(n)\}\} \quad = \quad \tau_{call}(n)$$

By arguments similar to those used for the case $\neg\mathrm{Priv}(n)$, $\mathsf{trans}^{i+1}[n] \subseteq \tau_{trans}(n)$ and $\mathsf{catch}^{i+1}[n] \subseteq \tau_{catch}(n)$.

- case [*check*]: if $\ell(n) = \mathtt{check}(P)$, by the assignments at lines 20-21:

$$\begin{aligned} \mathsf{trans}^{i+1}[n] \quad &\leftarrow \quad \{\, \gamma \in \mathsf{in}^i[n] \mid P \in \Pi(\gamma)\, \} \\ &\subseteq \quad \{\, \gamma \in \tau_{in}(n) \mid P \in \Pi(\gamma)\, \} \\ &= \quad \tau_{trans}(n) \end{aligned}$$

$$\begin{aligned} \mathsf{catch}^{i+1}[n] \quad &\leftarrow \quad \{\, \gamma \in \mathsf{in}^i[n] \mid P \notin \Pi(\gamma)\, \} \\ &\subseteq \quad \{\, \gamma \in \tau_{in}(n) \mid P \notin \Pi(\gamma)\, \} \\ &= \quad \tau_{catch}(n) \end{aligned}$$

Now we prove that (23b) is an invariant, as well, using what we have just established above. Inside the **switch** statement at lines 23-31, $\mathsf{out}^{i+1}[n, n']$ is updated for each edge $(n, n') \in E$, according to one of the following cases:

- case [*entry*]: if $\bullet\!\!\longrightarrow n'$, then, by the assignment at line 25:

$$\mathsf{out}^{i+1}[n, n'] \quad \leftarrow \quad \{\{\mathrm{Dom}(n')\}\} \quad = \quad \tau_{out}(n, n')$$

- case [*call*]: if $n \longrightarrow n'$, then, by the assignment at line 27:

$$\begin{aligned} \mathsf{out}^{i+1}[n, n'] \quad &\leftarrow \quad \{\, \gamma \cup \{\mathrm{Dom}(n')\} \mid \gamma \in \mathsf{call}^{i+1}[n]\, \} \\ &\subseteq \quad \{\, \gamma \cup \{\mathrm{Dom}(n')\} \mid \gamma \in \tau_{call}(n)\, \} \\ &= \quad \tau_{out}(n, n') \end{aligned}$$

- case [*transfer*]: if $n \dashrightarrow n'$, then, by the assignment at line 29:

$$\mathsf{out}^{i+1}[n, n'] \quad \leftarrow \quad \mathsf{trans}^{i+1}[n] \quad \subseteq \quad \tau_{trans}(n) \quad = \quad \tau_{out}(n, n')$$

- case [*catch*]: if $n \dashrightarrow_{\lightning} n'$, then, by the assignment at line 31:

$$\mathsf{out}^{i+1}[n, n'] \quad \leftarrow \quad \mathsf{catch}^{i+1}[n] \quad \subseteq \quad \tau_{catch}(n) \quad = \quad \tau_{out}(n, n')$$

To prove (23a) is an invariant, observe that, by the assignment at line 33, and using the fact that $\tau \models TP^{=}(G)$:

$$\begin{aligned} \mathsf{in}^{i+1}[n'] \quad &\leftarrow \quad \mathsf{in}^i[n'] \,\cup\, \mathsf{out}^{i+1}[n, n'] \\ &\subseteq \quad \tau_{in}(n') \,\cup\, \tau_{out}(n, n') \\ &= \quad \tau_{in}(n') \end{aligned}$$

So we conclude that, if we eventually obtain a solution, then is is minimal. Now, we show that the arrays in, out, call, trans and catch satisfy the equations of Table 3. Note first that, for each $n \in N$, by construction of the arrays in and out:

$$\mathsf{in}[n] \quad = \quad \bigcup_{(m,n) \in E} \mathsf{out}[m, n]$$

is invariant in the worklist-iteration algorithm. This is ensured by the assignment at line 33, because, for each $(m, n) \in E$, the value $\mathsf{out}[m, n]$ grows monotonically. Therefore, the array in satisfies the equation for TP_{in}.

To show that call satisfies the equation for TP_{call}, consider a call node n. Two cases arise, depending on n being privileged or not.
If $\neg\text{Priv}(n)$, consider first the case that $\mathsf{in}[n]$ is never assigned at line 33. As $\mathsf{in}[n] = \emptyset$, neither the assignment at line 34 nor those at lines 37 and 40 can take place: indeed, the first assignment is always executed after a non-empty assignment to $\mathsf{in}[n]$, while the others are prevented by the **if** statements at lines 36 and 39, respectively. Thus, n is never inserted into W, and:

$$\mathsf{call}[n] \quad = \quad \emptyset \quad = \quad \mathsf{in}[n]$$

If $\mathsf{in}[n] \neq \emptyset$, let $\mathsf{call}[n]$ bet assigned for the last time at the i-th iteration of the **while** loop (recall that we assume the algorithm to terminate). In this case, $\mathsf{in}[n]$ never changes its value after the i-th iteration; otherwise, n would be inserted into W again at line 34, and $\mathsf{call}[n]$ would be updated in a subsequent iteration. Therefore, by the assignment at line 16:

$$\mathsf{call}[n] \quad = \quad \mathsf{call}^i[n] \quad = \quad \mathsf{in}^i[n] \quad = \quad \mathsf{in}[n]$$

If $\text{Priv}(n)$, consider first the case that $\mathsf{in}[n]$ is never assigned inside the **while** loop. By the same arguments used above, we have:

$$\mathsf{call}[n] \quad = \quad \emptyset$$

Otherwise, if $\mathsf{in}[n] \neq \emptyset$, then, by the assignment at line 13, we have:

$$\mathsf{call}[n] \quad = \quad \{\{\text{Dom}(n)\}\}$$

and the value $\mathsf{call}[n]$ is no longer changed inside the **while** loop. This concludes the proof that call satisfies the equation for TP_{call}.

To show that trans satisfies the equation for TP_{trans}, take a node n. There are two cases, depending on n being a check or a call node. Consider first the case $\ell(n) = \texttt{check}(P)$. As seen above, if $\mathsf{in}[n]$ is never assigned, then:

$$\mathsf{trans}[n] \quad = \quad \emptyset \quad = \quad \{\, \gamma \in \mathsf{in}[n] \mid P \in \Pi(\gamma) \,\}$$

Otherwise, if $\mathsf{trans}[n]$ is lastly updated at the i-th iteration of the **while** loop, then $\mathsf{in}[n]$ cannot change its value after that iteration, and we have:

$$\mathsf{trans}[n] \quad = \quad \mathsf{trans}^i[n] \quad = \quad \{\, \gamma \in \mathsf{in}^i[n] \mid P \in \Pi(\gamma) \,\} \quad = \quad \{\, \gamma \in \mathsf{in}[n] \mid P \in \Pi(\gamma) \,\}$$

If $\ell(n) = \texttt{call}$, consider first the case that n is never inserted in the worklist, that is $\mathsf{in}[n] = \emptyset$. If $\neg\text{Priv}(n)$, this implies:

$$\mathsf{trans}[n] \quad = \quad \emptyset \quad = \quad \{\, \gamma \in \mathsf{in}[n] \mid \text{Trans}(n, \gamma) \,\}$$

Similarly, if $\text{Priv}(n)$:

$$\mathsf{trans}[n] \quad = \quad \emptyset \quad = \quad \{\, \gamma \in \mathsf{in}[n] \mid \text{Trans}(n, \{\text{Dom}(n)\}) \,\}$$

Otherwise, say n is extracted for the last time from W at the i-th iteration. Then, it cannot happen that $\mathsf{in}[n]$ changes its value after that iteration (as seen above), nor it can $\mathsf{in}[m]$ for any $m \in \rho(n)$. The latter statement is proved by contradiction. Assume that, for some $m \in \rho(n)$ and $j > i$:

$$\mathsf{in}[m] \quad = \quad \mathsf{in}^j[m] \quad \supset \quad \mathsf{in}^{j-1}[m]$$

Then, by the assignment at line 34, $m \in \mathsf{W}^j$. Since W is empty at the exit of the algorithm, there is an index $h > j$ such that $m = \textsc{Head}(\mathsf{W}^h)$. Thus, m is extracted from the worklist at the h-th iteration, and, after the **for** loop at lines 35-37, $n \in \mathsf{W}^h$. Since $h > j > i$, this contradicts our assumption that n is extracted for the last time from W at the i-th iteration.

If $\neg\mathrm{Priv}(n)$, then, by the assignment at line 17 of the i-th iteration:

$$\begin{aligned}\mathsf{trans}[n] &= \mathsf{trans}^i[n] \\ &= \{\, \gamma \in \mathsf{in}^i[n] \mid \exists m \in \rho(n).\ \gamma \cup \{\mathrm{Dom}(m)\} \in \mathsf{in}^i[m] \,\} \\ &= \{\, \gamma \in \mathsf{in}[n] \mid \exists m \in \rho(n).\ \gamma \cup \{\mathrm{Dom}(m)\} \in \mathsf{in}[m] \,\} \\ &= \{\, \gamma \in \mathsf{in}[n] \mid \mathrm{Trans}(n, \gamma) \,\}\end{aligned}$$

Similarly, if $\mathrm{Priv}(n)$, at the exit of the **while**:

$$\mathsf{trans}[n] \;=\; \{\, \gamma \in \mathsf{in}[n] \mid \mathrm{Trans}(n, \{\mathrm{Dom}(n)\}) \,\}$$

This concludes the proof that trans satisfies the equation for TP_{trans}.

To show that catch satisfies the equation for TP_{catch}, take a node n. There are two cases, depending on n being a check or a call node. Consider first the case $\ell(n) = \mathtt{check}(P)$. Just as above, if $\mathsf{in}[n]$ is never assigned:

$$\mathsf{catch}[n] \;=\; \emptyset \;=\; \{\, \gamma \in \mathsf{in}[n] \mid P \notin \Pi(\gamma) \,\}$$

Otherwise, if $\mathsf{catch}[n]$ is lastly updated at the i-th iteration of the **while** loop, then $\mathsf{in}[n]$ cannot change its value after that iteration, and we have:

$$\mathsf{catch}[n] \;=\; \mathsf{catch}^i[n] \;=\; \{\, \gamma \in \mathsf{in}^i[n] \mid P \notin \Pi(\gamma) \,\} \;=\; \{\, \gamma \in \mathsf{in}[n] \mid P \notin \Pi(\gamma) \,\}$$

If $\ell(n) = \mathtt{call}$, consider first the case that n is never inserted in the worklist, that is $\mathsf{in}[n] = \emptyset$. If $\neg\mathrm{Priv}(n)$, this implies:

$$\mathsf{catch}[n] \;=\; \emptyset \;=\; \{\, \gamma \in \mathsf{in}[n] \mid \mathrm{Catch}(n, \gamma) \,\}$$

Similarly, if $\mathrm{Priv}(n)$:

$$\mathsf{catch}[n] \;=\; \emptyset \;=\; \{\, \gamma \in \mathsf{in}[n] \mid \mathrm{Catch}(n, \{\mathrm{Dom}(n)\}) \,\}$$

If $\mathsf{in}[n] \neq \emptyset$, let n be extracted for the last time from W at the i-th iteration. Then, it cannot happen that $\mathsf{in}[n]$ changes its value after that iteration (as seen above), nor it can $\mathsf{catch}[m]$ for any $m \in \xi_1(n)$. The latter statement is proved by contradiction. Assume that, for some $m \in \xi_1(n)$ and $j > i$:

$$\mathsf{catch}[m] \;=\; \mathsf{catch}^j[m] \;\supset\; \mathsf{old\text{-}catch}^j[m] \;=\; \mathsf{catch}^{j-1}[m]$$

Then, after the **for** loop at lines 38-40, n would be inserted into W again, i.e. $n \in \mathsf{W}^j$. Since $j > i$, this contradicts our assumption that n is extracted for the last time from W at the i-th iteration.

If $\neg\mathrm{Priv}(n)$, then, by the assignment at line 18 of the i-th iteration:

$$\begin{aligned}\mathsf{catch}[n] &= \mathsf{catch}^i[n] \\ &= \{\, \gamma \in \mathsf{in}^i[n] \mid \exists m \in \xi_1(n).\ \gamma \cup \{\mathrm{Dom}(m)\} \in \mathsf{catch}^i[m] \,\} \\ &= \{\, \gamma \in \mathsf{in}[n] \mid \exists m \in \xi_1(n).\ \gamma \cup \{\mathrm{Dom}(m)\} \in \mathsf{catch}[m] \,\} \\ &= \{\, \gamma \in \mathsf{in}[n] \mid \mathrm{Catch}(n, \gamma) \,\}\end{aligned}$$

Similarly, if $\mathrm{Priv}(n)$, at the exit of the **while**:

$$\mathsf{catch}[n] \;=\; \{\, \gamma \in \mathsf{in}[n] \mid \mathrm{Catch}(n, \{\mathrm{Dom}(n)\}) \,\}$$

We conclude the proof by showing that out satisfies the equation for TP_{out} in Table 3. Consider the **switch** statement at lines 23-31.

First, let $\bullet\!\!\longrightarrow n'$. Since, by the assignment at line 1, n_ε is present in W when the **while** loop is entered, and W is empty at the exit of the algorithm, it follows that n_ε must be extracted from W during an iteration of the loop. According to the assignment at line 25, we have:

$$\mathsf{out}[n_\varepsilon, n'] \quad = \quad \{\{\mathrm{Dom}(n')\}\}$$

For the remaining cases, notice that the assignments made to out at lines 27, 29 and 31 correspond literally to the cases in Table 3. The fact that call, trans and catch satisfy the corresponding equations therefore suffices.

In the end, we have shown that, whenever the worklist-iteration algorithm terminates, then it computes the least solution to the TP analysis. $\square$

Lemma A.10. The worklist-iteration algorithm always terminates.

Proof. The bounded **for** loops at lines 2-3 and 4-5 trivially terminate. The worklist contains just the entry point n_ε when the **while** loop starts. Each iteration of the loop removes one node n from the worklist, and may add a finite number of nodes by the assignments at lines 34, 37 and 40. Consider the following cases:

- if $\ell(n) = \texttt{return}$ or $\ell(n) = \texttt{check}(P)$, then n can be inserted into W only at line 34. Indeed, by definition of ξ_1 and ρ, it follows:

  $$n' \in \xi_1(n) \;\vee\; n' \in \rho(n) \quad\Longrightarrow\quad n \longrightarrow \mu(n') \quad\Longrightarrow\quad \ell(n) = \texttt{call}$$

  so neither n can be appended to W at line 37 nor it can at line 40. Therefore, n can be added to the worklist only if, for some $(m, n) \in E$:

  $$\exists \gamma \in \mathsf{out}[m, n].\; \gamma \notin \mathsf{in}[n]$$

  which corresponds to the condition in the **if** statement at line 32. Thus, the new value assigned to $\mathsf{in}[n]$ at line 33 strictly contains the old one. Since $\mathcal{D}$ is finite, also its subsets are such, and the condition of the **if** statement will eventually become false. Therefore, n cannot be inserted into W infinitely often.

- if $\ell(n) = \texttt{call}$, then n can be inserted into W both at lines 34, 37 and 40. As above, line 34 cannot contribute infinitely often, so we are left to consider lines 37 and 40 only.

  Consider line 37 first. By contradiction, assume that it endlessly inserts n into W: then, there is a return node n' such that $n' \in \rho(n)$ and n' appears in W an infinite number of times. Contradiction: this cannot happen for return nodes.

  Next, consider line 40. By contradiction, assume that it endlessly inserts n into W: then, there is a node n' such that $n' \in \xi_1(n)$ and n' appears in W infinitely often. Moreover, the inequality:

  $$\mathsf{catch}^i[n'] \;\supset\; \mathsf{old\text{-}catch}^i[n'] \;=\; \mathsf{catch}^{i-1}[n']$$

  holds for an infinite number of iterations. Since $\mathcal{D}$ is finite, this is a contradiction.

- finally, if $n = n_\varepsilon$, then n is never inserted in the worklist after line 1. Actually, neither n_ε can be inserted into W by lines 37 or 40 (it is not a call node), nor it can by line 34 (it is never a target node in edges).

Therefore, each node can be inserted into the worklist only a finite number of times: this causes the worklist to be eventually exhausted, and the algorithm to terminate. $\square$

A.4 Computational Complexity

The bounded **for** loops at lines 2-3 and 4-5 are executed $|N|$ and $|E|$ times, respectively. Each iteration of the **while** loop at lines 6-40 extracts a node from the worklist, and may add new nodes to it because of the assignments at lines 34, 37 and 40. The if statement at line 32 ensures that, for each node n, the new value assigned to $\mathsf{in}[n]$ strictly contains the old one. Moreover, we can safely modify our algorithm so that it no longer inserts a node n into W once $\mathsf{in}[n]$ reaches its maximal value $2^{|\mathcal{D}_G|}$, where $\mathcal{D}_G$ is the set of protection domains actually occurring in G. Therefore, the body of **while** loop may be executed at most $|N| \cdot 2^{|\mathcal{D}_G|}$ times.

The most expensive instructions inside the loop are the assignments at lines 17 and 18. Consider line 17 first. To compute $\mathsf{trans}[n]$, we have to check, for each $\gamma \in \mathsf{in}[n]$ and $m \in \rho(n)$, whether $\gamma \cup \{\mathrm{Dom}(m)\} \in \mathsf{in}[m]$ or not. In the case that n has $\mathcal{O}(|N|)$ returns, and $|\mathsf{in}[m]| = \mathcal{O}(2^{|\mathcal{D}_G|})$ for each $m \in \rho(n)$, this instruction requires $\mathcal{O}(|N| \cdot 2^{2\,|\mathcal{D}_G|})$ comparisons. The linear factor can be discarded if we assume that, for each call node n, the set $\bigcup\{\, \mathsf{in[m]} \mid m \in \rho(n) \,\}$ is maintained, e.g. inside the **for** loop at lines 35-37. Similar arguments apply for line 18, so each iteration of the **while** loop can be executed in time proportional to $2^{2\,|\mathcal{D}_G|}$. Therefore, the worst-case complexity of the algorithm is:

$$\mathcal{O}(|N| \cdot 2^{3\,|\mathcal{D}_G|})$$

However, this upper bound is too pessimistic indeed. The exponential factor only occurs when the number of protection domains is proportional to the number of nodes, as the pathological case in Example 4 shows.

Actually, each protection domain corresponds to a grant entry in the policy file that defines the authorization state of the program: thus, the number of protection domains depends on the security policy associated with the program, rather than on the program size.

Thus, the worst-case complexity of the algorithm can be rewritten as:

$$\mathcal{O}(c \cdot |N|) \quad = \quad \mathcal{O}(|N|)$$

where the constant c depends on the number of protection domains in G. An upper bound to the space complexity of the algorithm is:

$$\mathcal{O}(\max\{|N|, |E|\} \cdot 2^{|\mathcal{D}_G|}) \quad = \quad \mathcal{O}(\max\{|N|, |E|\})$$

where the exponential factor has been discarded by the arguments above.

Example 4. For any $k > 0$, let $\mathcal{D}_k = \{D_0, \ldots, D_k\}$, and let G_k be the CFG:

$$\begin{aligned} N &= \{m_1, \ldots, m_k,\, n_1, \ldots, n_k\} \\ E &= \{\, m_i \rightleftarrows n_i \mid i \in 1..k \,\} \cup \{\, m_i \rightleftarrows m_{i+1} \mid i \in 1..k-1 \,\} \cup \{\bullet\!\!\longrightarrow m_1\} \\ \mathrm{Priv}(n) &= \mathit{false} \quad \text{for each } n \in N \\ \mathrm{Dom}(m_i) &= D_0 \quad \text{for each } i \in 1..k \\ \mathrm{Dom}(n_i) &= D_i \quad \text{for each } i \in 1..k \end{aligned}$$

where $\ell(m_i) = \ell(n_i) = \mathtt{call}$ for $i \in 1..k$, and $n \rightleftarrows m$ abbreviates $n \longrightarrow m$ and $m \longrightarrow n$. As an example, the CFG G_3 is depicted in Fig. 4.

For each $i \in 1..k$ and context $\gamma \subseteq \mathcal{D}_k$ containing D_0, we have:

$$\exists \sigma.\, G_k \rhd \sigma : m_i \;\wedge\; \gamma = \Gamma(\sigma : m_i)$$

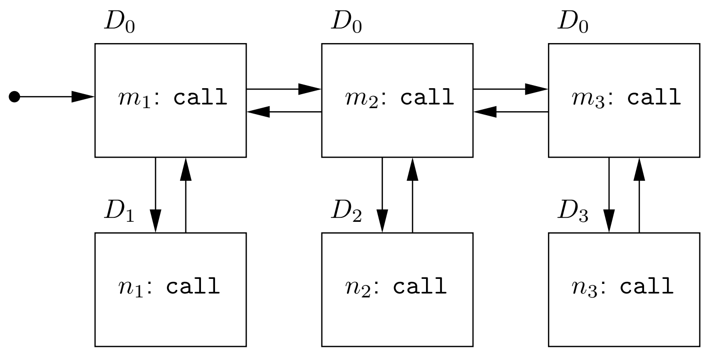


Figure 4: CFG for Example 4 ($k = 3$).

Let $\tau \models TP^{=}(G_k, \mathit{Perm})$ for an arbitrary security policy Perm. By theorem A.5, it follows that $\gamma \in \tau_{in}(m_i)$. Then, $|\tau_{in}(m_i)| = 2^{|\mathcal{D}_k|-1} = 2^k = 2^{|N|/2}$.

A.5 Program transformations

Lemma A.11. Let $\dot{n}$ be inlineable in G. Then, for each state σ,

$$\Gamma_{\dot{G}}(\mathit{inl}_{\dot{n}}(\sigma)) \quad = \quad \mathit{Inl}_{\dot{n}}(\Gamma_G(\sigma))$$

Proof. Let $\gamma = \Gamma_G(\sigma)$, $\dot{\sigma} = \mathit{inl}_{\dot{n}}(\sigma)$ and $\dot{\gamma} = \Gamma_{\dot{G}}(\dot{\sigma})$. We proceed by induction on the size (number of nodes) of σ. The base case is $\sigma = []$. Then, $\dot{\sigma} = []$ by rule inl_1, $\gamma = \dot{\gamma} = \emptyset$, and $\emptyset = \mathit{Inl}_{\dot{n}}(\emptyset)$ by rule Inl_1.

For the inductive case, case analysis on the rule used to deduce $\dot{\sigma} = \mathit{inl}_{\dot{n}}(\sigma)$ gives:

- case $[\mathit{inl}_2]$: if $\sigma = \sigma' : n'$, $\mathit{top}(\sigma') \neq \dot{n}$ and $\dot{\sigma}' = \mathit{inl}_{\dot{n}}(\sigma')$, then $\mathit{inl}_{\dot{n}}(\sigma) = \dot{\sigma}' : n'$. Moreover, by condition (1b), it follows that $\dot{n} \not\longrightarrow \mu(n')$. Let $\gamma' = \Gamma_G(\sigma')$, and $\dot{\gamma}' = \Gamma_{\dot{G}}(\dot{\sigma}')$. There are two subcases, according n' being privileged or not.

  If $\mathrm{Priv}_G(n')$, then $\gamma = \{\mathrm{Dom}_G(n')\}$. By definition 5.4, we have that $\mathrm{Priv}_{\dot{G}}(n')$ and $\mathrm{Dom}_{\dot{G}}(n') = \mathrm{Dom}_G(n')$. Since $\dot{n} \not\longrightarrow \mu(n')$, by rule Inl_3 it follows that:

  $$\mathit{Inl}_{\dot{n}}(\gamma) \quad = \quad \mathit{Inl}_{\dot{n}}(\{\mathrm{Dom}_G(n')\}) \quad = \quad \{\mathrm{Dom}_G(n')\} \quad = \quad \{\mathrm{Dom}_{\dot{G}}(n')\} \quad = \quad \dot{\gamma}$$

  Otherwise, if $\neg\mathrm{Priv}_G(n')$, then:

  $$\begin{aligned} \mathit{Inl}_{\dot{n}}(\gamma) &= \mathit{Inl}_{\dot{n}}(\gamma' \cup \{\mathrm{Dom}_G(n')\}) && \text{as } \neg\mathrm{Priv}_G(n') \\ &= \mathit{Inl}_{\dot{n}}(\gamma') \cup \mathit{inl}_{\dot{n}}(\{\mathrm{Dom}_G(n')\}) && \text{by rule } \mathit{Inl}_2 \\ &= \dot{\gamma}' \cup \mathit{Inl}_{\dot{n}}(\{\mathrm{Dom}_G(n')\}) && \text{by the ind. hyp.} \\ &= \dot{\gamma}' \cup \mathit{Inl}_{\dot{n}}(\{\mathrm{Dom}_{\dot{G}}(n')\}) && \text{by def. 5.4} \\ &= \dot{\gamma}' \cup \{\mathrm{Dom}_{\dot{G}}(n')\} && \text{by rule } \mathit{Inl}_3 \\ &= \dot{\gamma} && \text{as } \neg\mathrm{Priv}_{\dot{G}}(n') \end{aligned}$$

- case $[\mathit{inl}_3]$: if $\sigma = \sigma' : \dot{n} : n'$, then $\mathit{inl}_{\dot{n}}(\sigma) = \dot{\sigma}' : n'$, where $\dot{\sigma}' = \mathit{inl}_{\dot{n}}(\sigma')$. Note that, by lemma A.1 and condition (1a), $\dot{n} \longrightarrow \mu(n')$. Let $\gamma' = \Gamma_G(\sigma')$ and $\dot{\gamma}' = \Gamma_{\dot{G}}(\dot{\sigma}')$. We have to consider the following cases.

  If $\mathrm{Priv}_G(n')$, by definition 5.4, it follows that $\mathrm{Priv}_{\dot{G}}(n')$ and $\mathrm{Dom}_{\dot{G}}(n') = \mathrm{Dom}_G(\dot{n})$. Then, $\gamma = \{\mathrm{Dom}_G(n')\}$ and $\dot{\gamma} = \{\mathrm{Dom}_{\dot{G}}(n')\}$, and, by rule Inl_4:

  $$\mathit{Inl}_{\dot{n}}(\gamma) \quad = \quad \mathit{Inl}_{\dot{n}}(\{\mathrm{Dom}_G(n')\}) \quad = \quad \{\mathrm{Dom}_G(\dot{n})\} \quad = \quad \{\mathrm{Dom}_{\dot{G}}(n')\} \quad = \quad \dot{\gamma}$$

Otherwise, if $\neg\mathrm{Priv}_G(n')$, there are two further subcases, according $\dot{n}$ being privileged or not.
If $\mathrm{Priv}_G(\dot{n})$, then $\mathrm{Priv}_{\dot{G}}(n')$ follows by definition 5.4, and:

$$\begin{aligned}
\mathit{Inl}_{\dot{n}}(\gamma) &= \mathit{Inl}_{\dot{n}}(\{\mathrm{Dom}_G(\dot{n})\} \cup \{\mathrm{Dom}_G(n')\}) && \text{as } \mathrm{Priv}_G(\dot{n}),\ \neg\mathrm{Priv}_G(n') \\
&= \mathit{Inl}_{\dot{n}}(\{\mathrm{Dom}_G(\dot{n})\}) \ \cup\ \mathit{Inl}_{\dot{n}}(\{\mathrm{Dom}_G(n')\}) && \text{by rule } \mathit{Inl_2} \\
&= \{\mathrm{Dom}_G(\dot{n})\} \ \cup\ \{\mathrm{Dom}_G(\dot{n})\} && \text{by rules } \mathit{Inl_3} \text{ and } \mathit{Inl_4} \\
&= \{\mathrm{Dom}_{\dot{G}}(n')\} && \text{by def. 5.4} \\
&= \dot{\gamma} && \text{as } \mathrm{Priv}_{\dot{G}}(n')
\end{aligned}$$

Otherwise, if $\neg\mathrm{Priv}_G(\dot{n})$, then:

$$\begin{aligned}
\mathit{Inl}_{\dot{n}}(\gamma) &= \mathit{Inl}_{\dot{n}}(\gamma' \cup \{\mathrm{Dom}_G(\dot{n})\} \cup \{\mathrm{Dom}_G(n')\}) && \text{as } \neg\mathrm{Priv}_G(\dot{n}),\ \neg\mathrm{Priv}_G(n') \\
&= \mathit{Inl}_{\dot{n}}(\gamma') \cup \mathit{Inl}_{\dot{n}}(\{\mathrm{Dom}_G(\dot{n})\}) \cup \mathit{Inl}_{\dot{n}}(\{\mathrm{Dom}_G(n')\}) && \text{by rule } \mathit{Inl_2} \\
&= \mathit{Inl}_{\dot{n}}(\gamma') \ \cup\ \{\mathrm{Dom}_G(\dot{n})\} && \text{by rule } \mathit{Inl_4} \\
&= \dot{\gamma}' \ \cup\ \{\mathrm{Dom}_G(\dot{n})\} && \text{by the ind. hyp.} \\
&= \dot{\gamma}' \cup \{\mathrm{Dom}_{\dot{G}}(n')\} && \text{by def. 5.4} \\
&= \dot{\gamma} && \text{as } \neg\mathrm{Priv}_{\dot{G}}(n') \quad \square
\end{aligned}$$

Theorem A.12. Let $\dot{n}$ be inlineable in G, and $\dot{G}$ be the $\dot{n}$-inlined version of G. Then, each trace of G corresponds to a trace of $\dot{G}$, i.e.:

$$\langle \sigma_0, x_0 \rangle \ \rhd\ \cdots\ \rhd\ \langle \sigma_k, x_k \rangle \qquad \Longleftrightarrow \qquad \langle \dot{\sigma}_0, x_0 \rangle\ \rhd^{\dot{n}}_{\mathit{inl}}\ \cdots\ \rhd^{\dot{n}}_{\mathit{inl}}\ \langle \dot{\sigma}_k, x_k \rangle$$

where $\sigma_0 = []$, $x_0 = \mathit{false}$, and $\dot{\sigma}_i = \mathit{inl}_{\dot{n}}(\sigma_i)$ for each $i \in 0..k$.

Proof. Consider the forward implication first. We proceed by case analysis on the rule used to deduce $\langle \sigma_i, x_i \rangle \rhd \langle \sigma_{i+1}, x_{i+1} \rangle$. We omit a detailed discussion of the cases $\rhd_{\mathit{fail}}$ and $\rhd_{\mathit{catch}}$, because they are treated similarly to the cases $\rhd_{\mathit{fail}}$ and $\rhd_{\mathit{ret}}$, respectively.

- case $[\mathit{call}]$:

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n'}{\sigma : n \rhd \sigma : n : n'}$$

Here $\sigma_i = \sigma : n$ and $\sigma_{i+1} = \sigma : n : n'$. Let $\sigma' : n = \mathit{inl}_{\dot{n}}(\sigma : n) = \dot{\sigma}_i$.
If $n \neq \dot{n}$, then rule $\rhd_{\mathit{icall1}}$ yields:

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n' \quad n \neq \dot{n}}{\sigma' : n \ \rhd^{\dot{n}}_{\mathit{inl}}\ \sigma' : n : n'}$$

To show that $\dot{\sigma}_{i+i} = \sigma' : n : n' = \mathit{inl}_{\dot{n}}(\sigma : n : n') = \mathit{inl}_{\dot{n}}(\sigma_{i+1})$, it suffices to note that rule $\mathit{inl_2}$ instances to:

$$\frac{\mathit{inl}_{\dot{n}}(\sigma : n) = \sigma' : n \quad \mathit{top}(\sigma : n) \neq \dot{n}}{\mathit{inl}_{\dot{n}}(\sigma : n : n') = \sigma' : n : n'}$$

Otherwise, if $n = \dot{n}$, then rules $\rhd_{\mathit{icall2}}$ and $\mathit{inl_3}$ give:

$$\frac{\ell(\dot{n}) = \texttt{call} \quad \dot{n} \longrightarrow n'}{\sigma' : \dot{n} \ \rhd^{\dot{n}}_{\mathit{inl}}\ \sigma' : n'} \qquad\qquad \frac{\mathit{inl}_{\dot{n}}(\sigma) = \dot{\sigma}}{\mathit{inl}_{\dot{n}}(\sigma : \dot{n} : n') = \dot{\sigma} : n'}$$

To prove that $\sigma' = \dot{\sigma}$, assume first that $\sigma = []$. Then, $\sigma' : \dot{n} = \mathit{inl}_{\dot{n}}([\dot{n}]) = [\dot{n}]$ implies that $\sigma' = []$, and $\dot{\sigma} = \mathit{inl}_{\dot{n}}([]) = []$, Second, assume $\sigma = \sigma'' : n''$. Condition (1a), ensures that $n'' \neq \dot{n}$, because, otherwise, it would be $\dot{n} \longrightarrow \mu(\dot{n})$. Then, rule *inl2* gives:

$$\frac{\mathit{inl}_{\dot{n}}(\sigma'' : n'') = \dot{\sigma} \quad \mathit{top}(\sigma'' : n'') \neq \dot{n}}{\mathit{inl}_{\dot{n}}(\sigma'' : n'' : \dot{n}) = \dot{\sigma} : \dot{n}}$$

On the other hand, it is also $\sigma' : \dot{n} = \mathit{inl}_{\dot{n}}(\sigma : \dot{n})$. Therefore, $\sigma' = \dot{\sigma}$.

- case [*ret*]:

$$\frac{\ell(n') = \texttt{return} \quad n \dashrightarrow m}{\sigma : n : n' \rhd \sigma : m}$$

Let $\sigma' : n' = \mathit{inl}_{\dot{n}}(\sigma : n : n')$. We have to consider two subcases.
If $\dot{n} \not\longrightarrow \mu(n')$, let $\dot{\sigma} : n = \mathit{inl}_{\dot{n}}(\sigma : n)$. Then, lemma A.1 ensures that $n \neq \dot{n}$, hence rules *inl2* and $\rhd_{\mathit{iret1}}$ give:

$$\frac{\mathit{inl}_{\dot{n}}(\sigma : n) = \dot{\sigma} : n \quad \mathit{top}(\sigma : n) \neq \dot{n}}{\mathit{inl}_{\dot{n}}(\sigma : n : n') = \dot{\sigma} : n : n'} \qquad \frac{\ell(n') = \texttt{return} \quad n \dashrightarrow m \quad \dot{n} \not\longrightarrow \mu(n')}{\dot{\sigma} : n : n' \rhd^{\dot{n}}_{\mathit{inl}} \dot{\sigma} : m}$$

Then, $\dot{\sigma} : m = \mathit{inl}_{\dot{n}}(\sigma : m)$ immediately follows by the fact that $\dot{\sigma} : n = \mathit{inl}_{\dot{n}}(\sigma : n)$.
Otherwise, if $\dot{n} \longrightarrow \mu(n')$, let $\dot{\sigma} = \mathit{inl}_{\dot{n}}(\sigma)$. Lemma A.1 and condition (1b) ensure that $n = \dot{n}$. Then, rules *inl3* and $\rhd_{\mathit{iret2}}$ give:

$$\frac{\mathit{inl}_{\dot{n}}(\sigma) = \dot{\sigma}}{\mathit{inl}_{\dot{n}}(\sigma : \dot{n} : n') = \dot{\sigma} : n'} \qquad \frac{\ell(n') = \texttt{return} \quad \dot{n} \dashrightarrow m \quad \dot{n} \longrightarrow \mu(n')}{\dot{\sigma} : n' \rhd^{\dot{n}}_{\mathit{inl}} \dot{\sigma} : m}$$

To prove $\dot{\sigma} : m = \mathit{inl}_{\dot{n}}(\sigma : m)$, observe that, since $\mathit{top}(\sigma) \neq \dot{n}$ is ensured by condition (1a), then rule *inl2* instances to:

$$\frac{\mathit{inl}_{\dot{n}}(\sigma) = \dot{\sigma} \quad \mathit{top}(\sigma) \neq \dot{n}}{\mathit{inl}_{\dot{n}}(\sigma : m) = \dot{\sigma} : m}$$

- case [*pass*]:

$$\frac{\ell(n) = \texttt{check}(P) \quad \sigma : n \vdash P \quad n \dashrightarrow m}{\sigma : n \rhd \sigma : m}$$

Let $\dot{\sigma} : n = \mathit{inl}_{\dot{n}}(\sigma : n)$, and let $\tau \models TP^{=}(G, \mathit{Perm})$. By theorem A.5 and constraint 5, there exist a context $\gamma \in \tau_{in}(n)$ such that $\gamma = \Gamma_G(\sigma : n)$. Then:

$$\begin{array}{rcll}
\sigma : n \vdash_{G,\mathit{Perm}} P & \iff & P \in \Pi(\Gamma_G(\sigma : n)) & \text{by theorem A.4} \\
& \iff & P \in \Pi(\mathit{Inl}_{\dot{n}}(\Gamma_G(\sigma : n))) & \text{by condition (1d)} \\
& \iff & P \in \Pi(\Gamma_{\dot{G}}(\mathit{inl}_{\dot{n}}(\sigma : n))) & \text{by lemma A.11} \\
& \iff & P \in \Pi(\Gamma_{\dot{G}}(\dot{\sigma} : n)) & \text{by def. } \dot{\sigma} : n \\
& \iff & \dot{\sigma} : n \vdash_{\dot{G},\mathit{Perm}} P & \text{by theorem A.4}
\end{array}$$

where $\sigma : n \vdash_{G,\mathit{Perm}} P$ emphasises the fact that the relation $\vdash$ depends on a given CFG G and security policy *Perm*. Then, rule $\rhd_{\mathit{check}}$ gives:

$$\frac{\ell(n) = \texttt{check}(P) \quad \dot{\sigma} : n \vdash P \quad n \dashrightarrow m}{\dot{\sigma} : n \rhd^{\dot{n}}_{\mathit{inl}} \dot{\sigma} : m}$$

and $\dot{\sigma} : m = \mathit{inl}_{\dot{n}}(\sigma : m)$ immediately follows by the fact that $\dot{\sigma} : n = \mathit{inl}_{\dot{n}}(\sigma : n)$.

- case [*throw*]:
$$\frac{n' \not\dashrightarrow_{\lightning}}{\sigma : n'\lightning \rhd \sigma\lightning}$$
If $\dot{n} \not\longrightarrow \mu(n')$, let $\dot{\sigma} = \mathit{inl}_{\dot{n}}(\sigma)$. The condition (1a) ensures that $\mathit{top}(\sigma) \neq \dot{n}$. Then, rules inl_2 and $\rhd_{\mathit{icatch1}}$ give:
$$\frac{\mathit{inl}_{\dot{n}}(\sigma) = \dot{\sigma} \quad \mathit{top}(\sigma) \neq \dot{n}}{\mathit{inl}_{\dot{n}}(\sigma : n') = \dot{\sigma} : n'} \qquad \frac{n' \not\dashrightarrow_{\lightning} \quad \dot{n} \not\longrightarrow \mu(n')}{\dot{\sigma} : n'\lightning \rhd^{\dot{n}}_{\mathit{inl}} \dot{\sigma}\lightning}$$
Otherwise, if $\dot{n} \longrightarrow \mu(n')$, then by condition (1b) and lemma A.1 it follows that $\sigma = \sigma' : \dot{n}$ for some σ'. Let $\dot{\sigma} = \mathit{inl}_{\dot{n}}(\sigma')$. Then, rules inl_3 and $\rhd_{\mathit{icatch2}}$ give:
$$\frac{\mathit{inl}_{\dot{n}}(\sigma') = \dot{\sigma}}{\mathit{inl}_{\dot{n}}(\sigma' : \dot{n} : n') = \dot{\sigma} : n'} \qquad \frac{n' \not\dashrightarrow_{\lightning} \quad \dot{n} \longrightarrow \mu(n')}{\dot{\sigma} : n'\lightning \rhd^{\dot{n}}_{\mathit{inl}} \dot{\sigma} : \dot{n}\lightning}$$
To prove that $\dot{\sigma} : \dot{n} = \mathit{inl}_{\dot{n}}(\sigma)$, observe that, since $\mathit{top}(\sigma') \neq \dot{n}$ is ensured by condition (1a), then rule inl_2 instances to:
$$\frac{\mathit{inl}_{\dot{n}}(\sigma') = \dot{\sigma} \quad \mathit{top}(\sigma') \neq \dot{n}}{\mathit{inl}_{\dot{n}}(\sigma' : \dot{n}) = \dot{\sigma} : \dot{n}}$$

For the backward implication, we proceed by case analysis on the rule used to deduce $\langle \dot{\sigma}_i, x_i \rangle \rhd \langle \dot{\sigma}_{i+1}, x_{i+1} \rangle$. The function *inl* is bijective: given an inlined state $\dot{\sigma}$, the original state can be recovered by inserting $\dot{n}$ before each n' occurring in $\dot{\sigma}$ whenever $\dot{n} \longrightarrow \mu(n')$.

- case [*icall1*]:
$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n' \quad n \neq \dot{n}}{\sigma' : n \rhd^{\dot{n}}_{\mathit{inl}} \sigma' : n : n'}$$
Since *inl* is bijective, let $\sigma : n$ be such that $\mathit{inl}_{\dot{n}}(\sigma : n) = \sigma' : n$. Then:
$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n'}{\sigma : n \rhd \sigma : n : n'} \qquad \frac{\mathit{inl}_{\dot{n}}(\sigma : n) = \sigma' : n \quad \mathit{top}(\sigma : n) \neq \dot{n}}{\mathit{inl}_{\dot{n}}(\sigma : n : n') = \sigma' : n : n'}$$
follow by rules $\rhd_{\mathit{call}}$ and inl_2, respectively.
- case [*icall2*]:
$$\frac{\ell(\dot{n}) = \texttt{call} \quad \dot{n} \longrightarrow n'}{\sigma' : \dot{n} \rhd^{\dot{n}}_{\mathit{inl}} \sigma' : n'}$$
Let $\sigma : \dot{n}$ be such that $\mathit{inl}_{\dot{n}}(\sigma : \dot{n}) = \sigma' : \dot{n}$. Then:
$$\frac{\ell(\dot{n}) = \texttt{call} \quad \dot{n} \longrightarrow n'}{\sigma : \dot{n} \rhd \sigma : \dot{n} : n'} \qquad \frac{\mathit{inl}_{\dot{n}}(\sigma) = \sigma'}{\mathit{inl}_{\dot{n}}(\sigma : \dot{n} : n') = \sigma' : n'}$$
follow by rules $\rhd_{\mathit{call}}$ and inl_3, respectively.
- case [*iret1*]:
$$\frac{\ell(n') = \texttt{return} \quad n \dashrightarrow m \quad \dot{n} \not\longrightarrow \mu(n')}{\dot{\sigma} : n : n' \rhd^{\dot{n}}_{\mathit{inl}} \dot{\sigma} : m}$$
Since $\dot{n} \not\longrightarrow \mu(n')$, by condition (1a) it follows that $n \neq \dot{n}$. So, let $\sigma : n : n'$ be such that $\mathit{inl}_{\dot{n}}(\sigma : n : n') = \dot{\sigma} : n : n'$. Then:
$$\frac{\ell(n') = \texttt{return} \quad n \dashrightarrow m}{\sigma : n : n' \rhd \sigma : m} \qquad \frac{\mathit{inl}_{\dot{n}}(\sigma : n) = \dot{\sigma} : n \quad \mathit{top}(\sigma : n) \neq \dot{n}}{\mathit{inl}_{\dot{n}}(\sigma : n : n') = \dot{\sigma} : n : n'}$$
follow by rules $\rhd_{\mathit{ret}}$ and inl_2, respectively, while $\dot{\sigma} : m = \mathit{inl}_{\dot{n}}(\sigma : m)$ immediately follows by the fact that $\dot{\sigma} : n = \mathit{inl}_{\dot{n}}(\sigma : n)$.

- case [*iret2*]:

$$\frac{\ell(n') = \texttt{return} \quad \dot{n} \dashrightarrow m \quad \dot{n} \longrightarrow \mu(n')}{\dot{\sigma} : n' \rhd^{\dot{n}}_{inl} \dot{\sigma} : m}$$

  Let $\sigma : n'$ be such that $inl_{\dot{n}}(\sigma : n') = \dot{\sigma} : n'$. Since $\dot{n} \longrightarrow \mu(n')$, by lemma A.1 and condition (1b) it follows that $top(\sigma) = \dot{n}$, i.e. $\sigma = \sigma' : \dot{n}$ for some σ'. Then:

$$\frac{\ell(n') = \texttt{return} \quad \dot{n} \dashrightarrow m}{\sigma' : \dot{n} : n' \rhd \sigma' : m} \qquad \frac{inl_{\dot{n}}(\sigma) = \dot{\sigma}}{inl_{\dot{n}}(\sigma : \dot{n} : n') = \dot{\sigma} : n'}$$

  follow by rules $\rhd_{ret}$ and *inl3*, respectively, while $\dot{\sigma} : m = inl_{\dot{n}}(\sigma : m)$ immediately follows by the fact that $\dot{\sigma} = inl_{\dot{n}}(\sigma)$.

- case [*ithrow1*]:

$$\frac{n' \not\dashrightarrow_{\lightning} \quad \dot{n} \not\longrightarrow \mu(n')}{\dot{\sigma} : n' \lightning \rhd^{\dot{n}}_{inl} \dot{\sigma} \lightning}$$

  Let $\sigma : n'$ be such that $inl_{\dot{n}}(\sigma : n') = \dot{\sigma} : n'$.

$$\frac{n' \not\dashrightarrow_{\lightning}}{\sigma : n' \lightning \rhd \sigma \lightning} \qquad \frac{inl_{\dot{n}}(\sigma) = \dot{\sigma} \quad top(\sigma) \neq \dot{n}}{inl_{\dot{n}}(\sigma : n') = \dot{\sigma} : n'}$$

  follow by rules $\rhd_{throw}$ and *inl2*, respectively.

- case [*ithrow2*]:

$$\frac{n' \not\dashrightarrow_{\lightning} \quad \dot{n} \longrightarrow \mu(n')}{\dot{\sigma} : n' \lightning \rhd^{\dot{n}}_{inl} \dot{\sigma} : \dot{n} \lightning}$$

  Let $\sigma : n'$ be such that $inl_{\dot{n}}(\sigma : n') = \dot{\sigma} : n'$. Since $\dot{n} \longrightarrow \mu(n')$, by lemma A.1 and condition (1b) there exists a σ' such that $\sigma = \sigma' : \dot{n}$. Then:

$$\frac{n' \not\dashrightarrow_{\lightning}}{\sigma : n' \lightning \rhd \sigma \lightning} \qquad \frac{inl_{\dot{n}}(\sigma') = \dot{\sigma}}{inl_{\dot{n}}(\sigma' : \dot{n} : n') = \dot{\sigma} : n'}$$

  follow by rules $\rhd_{throw}$ and *inl3*, respectively, while $\dot{\sigma} : \dot{n} = inl_{\dot{n}}(\sigma)$ immediately follows by the fact that $inl_{\dot{n}}(\sigma') = \dot{\sigma}$.

$\square$

B Java code for the e-commerce example

```
public class Browser {

    public String[] getPrefs(String filename) {
        try {
            InputStream in = new FileInputStream(filename);
            ... // read prefs from in
        } catch (Exception ex) {
            Socket s = new Socket(...);
            InputStream in = s.getInputStream();
            ... // read prefs from in
        }
        return prefs;
    }

    public void changePrefs(String filename) {
        try {
            InputStream in = new FileInputStream(filename);
            ... // read prefs from in, then asks the user for the new prefs
            OutputStream out = new FileOutputStream(filename);
            ... // write prefs to out
        } catch (Exception ex) {
            Socket s = new Socket(...);
            InputStream in = s.getInputStream();
            ... // read prefs from in, then asks the user for the new prefs
            OutputStream out = s.getOutputStream();
            ... // write prefs to out
        }
    }
}
```

Figure 5: Java program for the client browser.

```
public class Bank {
    private int balance;
    ...
    public boolean canpay(int account, int amount) {
        AccessController.checkPermission(canpay);
        Object res = AccessController.doPrivileged(new PrivilegedAction() {
            public Object run() {
                return new Boolean(readBalance(account) > amount);
            }
        });
        return ((Boolean) res).booleanValue();
    }

    public boolean debit(int account, int amount) {
        AccessController.checkPermission(debit);
        boolean res = false;
        if (canpay(account, amount)) {
            AccessController.doPrivileged(new PrivilegedAction() {
                public Object run() {
                    writeBalance(account, readBalance(account) - amount);
                    res = true;
                    return null;
                }
            });
        }
        return res;
    }

    public boolean transfer(int srcAccount, int dstAccount, int amount) {
        AccessController.checkPermission(transfer);
        boolean res = debit(srcAccount, amount);
        if (res) credit(dstAccount, amount);
        return res;
    }

    public void credit(int account, int amount) {
        AccessController.checkPermission(credit);
        AccessController.doPrivileged(new PrivilegedAction() {
            public Object run() {
                writeBalance(account, readBalance(account) + amount);
                return null;
            }
        });
    }
}
```

Figure 6: Java program for the bank server.

W	in$[n]$ (= call$[n]$)	trans$[n]$	catch$[n]$
$\underline{0}$,2	T		
$\underline{2}$,4	U		
$\underline{4}$	TB,UB		
$\underline{13}$	TBS,UBS	TBS	UBS
$\underline{14}$	TBS		
$\underline{4}$	TB,UB	TB	UB
$\underline{5}$,6	TB		
$\underline{6}$,11	UB		
$\underline{11}$,15	TBS	TBS	
$\underline{15}$,12	UBS	UBS	
$\underline{12}$,16	TBS		
$\underline{16}$,7	UBS		
$\underline{7}$	TB,UB		
$\underline{1}$,3	T		
$\underline{3}$,8	U		
$\underline{8}$	TB,UB	TB	UB
$\underline{9}$,10	UB	UB	
$\underline{10}$	TB,UB		
$\underline{1}$,3	T	T	
$\underline{3}$	U	U	

Table 6: Iterations of the worklist algorithm (client side).

W	in$[n]$	call$[n]$	trans$[n]$	catch$[n]$
$\underline{0}$,2	C	C		
$\underline{2}$,4	U	U		
$\underline{4}$	CB,UB		CB	UB
$\underline{5}$	CB	B		
$\underline{20}$	BS		BS	
$\underline{21}$	BS			
$\underline{5}$	CB	B	CB	
$\underline{6}$	CB			
$\underline{0}$,2	C	C	C	
$\underline{2}$,1	U	U		U
$\underline{1}$	C	C		
$\underline{12}$	CB		CB	
$\underline{13}$	CB	CB		
$\underline{7}$	CB		CB	
$\underline{8}$	CB	CB	CB	
$\underline{9}$,11	CB	B	CB	
$\underline{11}$,10	CB			
$\underline{10}$,13	CB	B		
$\underline{13}$,22	CB	CB	CB	
$\underline{22}$,14,15	BS		BS	
$\underline{14}$,15,23	CB	CB		
$\underline{15}$,23,16	CB			
$\underline{23}$,16,1,3	BS			
$\underline{16}$,1,3,10	CB		CB	
$\underline{1}$,3,10,17	C	C	C	
$\underline{3}$,10,17	U	U		U
$\underline{10}$,17	CB	B	CB	
$\underline{17}$	CB	B	CB	
$\underline{18}$	CB	B	CB	
$\underline{19}$	CB			
$\underline{14}$	CB	CB	CB	

Table 7: Iterations of the worklist algorithm (server side).