A Simple Static Analysis for Java Security

Anindya Banerjee and David A. Naumann [a,1,2]

[a] *Stevens Institute of Technology, Hoboken, NJ 07030 USA*

Abstract

Security in Java depends on an access control mechanism specified operationally in terms of run-time stack inspection. We give a static analysis of safety, *i.e.*, the absence of security errors, and justify it using a more abstract, denotational semantics which can also be used to justify optimizing transformations.

1 Introduction

System security depends in part on protecting resources through access control. Typically, an access control list associates with each user name n a set $\mathcal{A}(n)$ of privileges, or resources, for which the user is authorized. For example, a user might be authorized to change their password but not to directly write the password file. A running program has an associated user, normally the one who invoked it, and the host system checks before performing a protected action that the user has the requisite authorization. In order for the host to export safe access to protected resources, certain trusted programs run under the auspices of the host rather than the user ("setuid" programs in UNIX). For example, the user can invoke the *passwd* program to change their password; this program runs under the auspices of *root* which is granted the privilege to write the password file.

The Java security system provides a mechanism of this kind [3]. It handles typical object-oriented patterns with non-hierarchical control flows, such as callbacks, where host code may call untrusted code. To prevent unintended propagation of privileges, each function activation (method call) is associated with a *principal* n which can be the user, *root*, or an authenticated remote site from which the code was obtained. Privileges must be explicitly enabled by

[1] Partially supported by the NSF under Grant No. EIA-9806835.
[2] Partially supported by the NSF under Grant No. INT-9813854.

an operation called `doPrivileged`. Before executing a protected operation, a check (`checkPermission`) is made that the associated privilege has been enabled and is authorized for all of the principals associated with activations currently on the stack. The intent is that a program only enables its privileges when they are needed, and cannot inadvertently use privileges not authorized for its callers. This "principle of least privilege" [3] may help isolate the effect of security bugs and facilitate safe optimization.

The Java system was specified operationally, in terms of the *stack inspection* algorithm used for checking. More abstract descriptions are needed to analyze properties of the system and to assess alternative implementations. An obvious optimization is the elimination of checks that can be shown by static analysis to be unnecessary. Concentrating on the most relevant language features —function definitions and calls, `doPrivileged`, `checkPermission`— several authors give formal semantics and static analyses that characterize program expressions that are *safe* in the sense that security checks never fail. As discussed in Section 5, these studies are complicated due to use of operational semantics and reliance on non-standard logics and type systems [6,5,7].

Our contribution is two-fold. We give a simple denotational semantics which is equivalent to stack inspection, but which uses available permissions directly rather than as encoded in the stack (Section 3). Unlike the cited work, our language includes recursion; without it, static analysis could be accomplished by brute-force methods. Our second contribution is to specify a static analysis derived from the semantics (Section 4). Thanks to the simplicity and compositionality of the semantics, we can give a detailed, self-contained proof of the safety property guaranteed by the static analysis.

2 Overview and language

We consider a call-by-value typed functional language extended with the security constructs involving given sets $\mathsf{Principals}$ and Privs along with mapping $\mathcal{A}$ from principals to sets of privileges. The body of a function is an expression signed by a principal. We write $\texttt{signs}\ n\ e$ for an expression e signed by n. When a function is applied, a stack frame is pushed with its arguments along with its signer. Ignoring arguments, a frame is a pair $\langle n, P\rangle$ where P is the set of privileges that have been enabled. This set is initially empty; privileges are enabled by the `doPrivileged` construct which we write $\texttt{dopriv}\ p\ \texttt{in}\ e$. Its effect is to add privilege p to the stack frame, evaluate e, and then remove p from the frame. This is done regardless of whether p is authorized for n.

To model Java's `checkPermission`, our construct is $\texttt{check}\ p\ \texttt{for}\ e$. Its effect is to raise a security error, which we denote by $\star$, unless the current stack contains some frame $\langle n, P\rangle$ with $p \in P$; moreover, p must be authorized for n and for the principals in all frames above this one. More precisely, $\texttt{check}\ p\ \texttt{for}\ e$ evaluates e provided $\mathsf{chk}(p, S)$ is true in the current stack S; otherwise it returns $\star$. Predicate chk is defined as follows, where we write $\langle n, P\rangle :: S$ for $\langle n, P\rangle$ on top of stack S. For the empty stack, $\mathsf{chk}(p, \mathit{nil}) \Leftrightarrow \mathsf{false}$. For nonempty stacks, define $\mathsf{chk}(p, (\langle n, P\rangle :: S)) \Leftrightarrow p \in \mathcal{A}(n) \wedge (p \in P \vee \mathsf{chk}(p, S))$.

As a running example, suppose p is the privilege for changing password and w is the privilege to write the password file; let $\mathcal{A}(\mathit{user}) = \{p\}$ and $\mathcal{A}(\mathit{root}) = \{w, p\}$. Suppose write is the low-level operation that is to be protected from direct user access. The host provides passwd, which guards write with the privilege w, and which can be called by user code.

$$
\begin{aligned}
\mathit{writepass} &= \texttt{fun}\ x.\ \texttt{signs}\ \mathit{root}\ \texttt{check}\ w\ \texttt{for}\ \mathit{write}(x, \text{“passfile”})\\
\mathit{passwd} &= \texttt{fun}\ x.\ \texttt{signs}\ \mathit{root}\ \texttt{check}\ p\ \texttt{for}\ \texttt{dopriv}\ w\ \texttt{in}\ \mathit{writepass}(x)\\
\mathit{use} &= \texttt{signs}\ \mathit{user}\ \texttt{dopriv}\ p\ \texttt{in}\ \mathit{passwd}(\text{“mypass”})\\
\mathit{bad} &= \texttt{signs}\ \mathit{user}\ \texttt{dopriv}\ w\ \texttt{in}\ \mathit{writepass}(\text{“mypass”})
\end{aligned}
$$

Here bad yields $\star$ because $\mathit{writepass}$ checks for privilege w which is not possessed by user, although bad enables it. By contrast, use does not raise an error. Function passwd checks for privilege p which is possessed by user and enabled by use; and passwd enables the privilege w needed by $\mathit{writepass}$.

As described, the stack inspection semantics is *lazy* in that tests $p \in \mathcal{A}(n)$ are only performed when p is actually needed to perform a guarded operation. But checks take time linear in the stack depth. A stack S determines a set $\mathsf{privs}\ S$ of enabled, authorized privileges: $p \in \mathsf{privs}\ S \Leftrightarrow \mathsf{chk}(p, S)$. This gives rise to an *eager* [3] semantics: Instead of evaluating an expression in the context of a stack S, we use $\mathsf{privs}\ S$, along with the current principal which appears on top of S. The eager semantics is given in Section 3. For implementation, the lazy strategy is believed to be more efficient than the eager one, but the latter facilitates analyses that support optimizations.

The constructs described so far are strict in $\star$: if a subexpression raises an error, so does the entire expression. In Java, however, security errors are exceptions that can be caught. So we include a non-strict construct, $\texttt{test}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, which evaluates e_1 if $\mathsf{chk}(p, S)$ succeeds in the current stack S, and evaluates e_2 otherwise. We include abstractions, $\texttt{fun}\ x.\ e$, even though they can be expressed using **letrec**, because they have a simpler semantics and safety proof. Types are generated by $t ::= \texttt{bool} \mid (t \to t)$ and expressions by the following:

$$
\begin{aligned}
e ::= {} & \texttt{true} \mid x \mid \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{fun}\ x.\ e \mid e_1\ e_2 \mid \texttt{letrec}\ f(x) = e_1\ \texttt{in}\ e_2) \mid \\
& \texttt{signs}\ n\ e \mid \texttt{dopriv}\ p\ \texttt{in}\ e \mid \texttt{check}\ p\ \texttt{for}\ e \mid \texttt{test}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2
\end{aligned}
$$

The $\texttt{signs}$ construct is separate from function definitions because this disen-

$$D \vdash \texttt{true} : \texttt{bool} \qquad D, x : t \vdash x : t \qquad \frac{D \vdash e : \texttt{bool} \quad D \vdash e_1 : t \quad D \vdash e_2 : t}{D \vdash \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 : t}$$

$$\frac{D, x : t_1 \vdash e : t_2}{D \vdash \texttt{fun } x.\, e : t_1 \to t_2} \qquad \frac{D \vdash e_1 : t_1 \to t_2 \qquad D \vdash e_2 : t_1}{D \vdash e_1\ e_2 : t_2}$$

$$\frac{D, f : t_1 \to t_2, x : t_1 \vdash e_1 : t_2 \qquad D, f : t_1 \to t_2 \vdash e_2 : t}{D \vdash \texttt{letrec } f(x) = e_1 \texttt{ in } e_2 : t}$$

$$\frac{D \vdash e : t}{D \vdash \texttt{signs } n\ e : t} \qquad \frac{D \vdash e : t}{D \vdash \texttt{dopriv } p \texttt{ in } e : t}$$

$$\frac{D \vdash e : t}{D \vdash \texttt{check } p \texttt{ for } e : t} \qquad \frac{D \vdash e_1 : t \qquad D \vdash e_2 : t}{D \vdash \texttt{test } p \texttt{ then } e_1 \texttt{ else } e_2 : t}$$

Fig. 1. Typing rules.

tangles definitions and proofs: these constructs are treated independently in our safety result. But unsigned abstractions do not properly model the Java mechanism. In the companion technical report [2], we show that our semantics is equivalent to stack inspection for standard expressions. An expression is *standard* if for every subexpression $\texttt{fun } x.\ e$ or $\texttt{letrec } f(x) = e \texttt{ in } e_1$ we have that e is $\texttt{signs } n\ e'$ for some n, e'.

3 Typing and semantics

Well-formed expressions are characterized by typing judgements $D \vdash e : t$ which express that e has type t where free identifiers are declared by D. A typing context D is a labeled tuple of declarations $\{x_1 : t_1, \ldots, x_k : t_k\}$. We write $D.x_i$ for the type of x_i, and $D, x : t$ for the extended context $\{x_1 : t_1, \ldots, x_k : t_k, x : t\}$. Complete typing rules are given in Figure 1.

The denotational semantics uses ordered sets. A *cpo* is a partially ordered set with least upper bounds (*lubs*) of ascending chains; it need not have a least element. We shall define, for each type t, a cpo $[\![t]\!]$. We assume that $\bot$ (divergence) and $\star$ are two values that are not elements of any $[\![t]\!]$. For cpo C, define $C_{\bot\star} = C \cup \{\bot, \star\}$, ordered as the disjoint union of C with $\{\star\}$, lifted with $\bot$. That is, for any $u, v \in C_{\bot\star}$, define $u \leq v$ iff $u = \bot$, $u = v$, or u and v are in C and $u \leq v$ in C.

Given a principal n, a set $P \in \mathcal{P}(\mathsf{Privs})$ of privileges, and environment h giving values to the free identifiers declared by D, the semantics $[\![D \vdash e : t]\!] nPh$ is either $\bot$ or $\star$ or an element of $[\![t]\!]$. This embodies the idea of eager evaluation in the context of enabled, authorized privileges, and it leads to the semantics

$$
\begin{aligned}
&[\![D \vdash \texttt{true} : \texttt{bool}]\!] nPh &&= \mathsf{true}\\
&[\![D, x : t \vdash x : t]\!] nPh &&= h.x\\
&[\![D \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t]\!] nPh\\
&\quad = \textbf{let}\ b\ =\ [\![D \vdash e : \texttt{bool}]\!] nPh\ \textbf{in}\\
&\qquad\quad \textbf{if}\ b\ \textbf{then}\ [\![D \vdash e_1 : t]\!] nPh\ \textbf{else}\ [\![D \vdash e_2 : t]\!] nPh\\
&[\![D \vdash \texttt{fun}\ x.\ e : t_1 \to t_2]\!] nPh &&= \lambda P' \in \mathcal{P}(\mathsf{Privs}).\ \lambda d \in [\![t_1]\!].\\
& && \quad [\![D, x : t_1 \vdash e : t_2]\!] nP'[h \mid x \mapsto d]\\
&[\![D \vdash e_1\ e_2 : t_2]\!] nPh &&= \textbf{let}\ f\ =\ [\![D \vdash e_1 : t_1 \to t_2]\!] nPh\ \textbf{in}\\
& && \quad \textbf{let}\ d = [\![D \vdash e_2 : t_1]\!] nPh\ \textbf{in}\ f P d\\
&[\![D \vdash \texttt{letrec}\ f(x) = e_1\ \texttt{in}\ e_2 : t]\!] nPh\\
&\quad = \textbf{let}\ G(g)\ =\ \lambda P'.\ \lambda d.\ [\![D, f : t_1 \to t_2, x : t_1 \vdash e_1 : t_2]\!] nP'[h \mid f \mapsto g, x \mapsto d]\ \textbf{in}\\
&\qquad [\![D, f : t_1 \to t_2 \vdash e_2 : t]\!] nP[h \mid f \mapsto \mathit{fix}\, G]\\
&[\![D \vdash \texttt{signs}\ n'\ e : t]\!] nPh &&= [\![D \vdash e : t]\!] n'(P \cap \mathcal{A}(n'))h\\
&[\![D \vdash \texttt{dopriv}\ p\ \texttt{in}\ e : t]\!] nPh &&= [\![D \vdash e : t]\!] n(P \sqcup_n \{p\})h\\
&[\![D \vdash \texttt{check}\ p\ \texttt{for}\ e : t]\!] nPh &&= \textbf{if}\ p \in P\ \textbf{then}\ [\![D \vdash e : t]\!] nPh\ \textbf{else}\ \star\\
&[\![D \vdash \texttt{test}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t]\!] nPh\\
&\quad = \textbf{if}\ p \in P\ \textbf{then}\ [\![D \vdash e_1 : t]\!] nPh\ \textbf{else}\ [\![D \vdash e_2 : t]\!] nPh
\end{aligned}
$$

Fig. 2. Denotational semantics

of types. We define $[\![\texttt{bool}]\!] = \{\mathsf{true}, \mathsf{false}\}$, ordered by equality. The semantics of arrow types reflects dependence on privilege sets:

$$[\![t_1 \to t_2]\!] = \mathcal{P}(\mathsf{Privs}) \to [\![t_1]\!] \to [\![t_2]\!]_{\bot\star}$$

where $\to$ denotes continuous function space, ordered pointwise. We let $\to$ associate to the right. We take the powerset $\mathcal{P}(\mathsf{Privs})$ to be a cpo ordered by equality. Although $[\![t_1 \to t_2]\!]$ does not contain $\bot$, it has a least element: the constant function $\lambda P.\ \lambda d.\ \bot$. For context $D = \{x_1 : t_1, \ldots\}$ we define $[\![D]\!]$ to be the set $\{x_1 : [\![t_1]\!], \ldots\}$ of labeled tuples of appropriate type (and write $\{\}$ for the empty tuple). Thus a judgement $D \vdash e : t$ denotes a function $[\![D \vdash e : t]\!]$ in $\mathsf{Principals} \to \mathcal{P}(\mathsf{Privs}) \to [\![D]\!] \to [\![t]\!]_{\bot\star}$.

The complete denotational semantics is in Figure 2. We use the metalanguage construct, **let** $d = E_1$ **in** E_2, with the following meaning: If the value of E_1 is either $\bot$ or $\star$ then that is the value of the entire let expression; otherwise, its value is the value of E_2 with d bound to the value of E_1. We write $P \sqcup_n \{p\}$ to abbreviate **if** $p \in \mathcal{A}(n)$ **then** $P \cup \{p\}$ **else** P. Finally, for $h \in [\![D]\!]$ we write $h.x_i$ for the value in $[\![t_i]\!]$ of field x_i. For $h \in [\![D]\!]$ and $d \in [\![t]\!]$ we write $[h \mid x \mapsto d]$ for the extended record in $[\![D, x : t]\!]$.

$$\Delta;\ n \vdash \texttt{true} : \texttt{bool},\ \varnothing \qquad \Delta, x : \theta;\ n \vdash x : \theta,\ \varnothing \qquad \frac{\Delta, x : \theta_1;\ n \vdash e : \theta_2,\ \Pi}{\Delta;\ n \vdash \texttt{fun}\ x.\ e : \theta_1 \xrightarrow{\Pi} \theta_2,\ \varnothing}$$

$$\frac{\Delta;\ n \vdash e_1 : \theta_1 \xrightarrow{\Pi} \theta_2,\ \Pi_1 \qquad \Delta;\ n \vdash e_2 : \theta_1',\ \Pi_2 \qquad \theta_1' \leq \theta_1}{\Delta;\ n \vdash e_1\ e_2 : \theta_2,\ \Pi \cup \Pi_1 \cup \Pi_2}$$

$$\frac{\Delta;\ n \vdash e : \texttt{bool},\ \Pi_1 \qquad \Delta;\ n \vdash e_1 : \theta,\ \Pi_2 \qquad \Delta;\ n \vdash e_2 : \theta,\ \Pi_3}{\Delta;\ n \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \theta,\ \Pi_1 \cup \Pi_2 \cup \Pi_3}$$

$$\frac{\Delta, f : \theta_1 \xrightarrow{\Pi} \theta_2, x : \theta_1;\ n \vdash e_1 : \theta_2,\ \Pi \qquad \Delta, f : \theta_1 \xrightarrow{\Pi} \theta_2;\ n \vdash e_2 : \theta,\ \Pi_1}{\Delta;\ n \vdash \texttt{letrec}\ f(x) = e_1\ \texttt{in}\ e_2 : \theta,\ \Pi \cup \Pi_1}$$

$$\frac{\Delta;\ n \vdash e : \theta,\ \Pi}{\Delta;\ n \vdash \texttt{check}\ p\ \texttt{for}\ e : \theta,\ \Pi \cup \{p\}} \qquad \frac{\Delta;\ n \vdash e : \theta,\ (\Pi \sqcup_n \{p\})}{\Delta;\ n \vdash \texttt{dopriv}\ p\ \texttt{in}\ e : \theta,\ \Pi}$$

$$\frac{\Delta;\ n' \vdash e : \theta,\ \Pi \qquad \Pi \subseteq \mathcal{A}(n')}{\Delta;\ n \vdash \texttt{signs}\ n'\ e : \theta,\ \Pi} \qquad \frac{\Delta;\ n \vdash e_1 : \theta,\ \Pi_1 \qquad \Delta;\ n \vdash e_2 : \theta,\ \Pi_2}{\Delta;\ n \vdash \texttt{test}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \theta,\ \Pi_1 \cup \Pi_2}$$

Fig. 3. Static analysis

4 Static Analysis

The semantics gives rise to a notion of safety dependent on principal and privilege: e is safe for n, P if $[\![\varnothing \vdash e : t]\!]nP\{\} \neq \star$. The static analysis is specified by a form of typing judgement which gives not only the type of an expression, but also n, P for which the expression is safe. Because an arrow type $t_1 \to t_2$ denotes functions dependent on a set of privileges, the static analysis uses annotated types to track sets of privileges adequate for safety. We adopt a Greek notational style for types in the static analysis. Letting Π range over sets of privileges, annotated types, θ, are defined by $\theta ::= \texttt{bool} \mid (\theta_1 \xrightarrow{\Pi} \theta_2)$. If $x : \theta_1 \vdash e : \theta_2$ depends on Π for safety, then the typing of its abstraction, $\texttt{fun}\ x.\ e : \theta_1 \xrightarrow{\Pi} \theta_2$, records this latent requirement. Writing Δ for a context assigning annotated types, the judgement $\Delta;\ n \vdash e : \theta,\ \Pi$ expresses that expression e, signed by principal n and typed in context Δ, has annotated type θ and is safe provided at least the set Π of privileges are enabled and authorized. Figure 3 specifies the analysis.

As an example, suppose the primitive *write* returns type `bool`. Then the expressions in the password example can be analyzed as follows, for any n.

$$\varnothing;\ n \vdash \mathit{writepass} : \texttt{string} \xrightarrow{\{w\}} \texttt{bool},\ \varnothing \qquad \varnothing;\ n \vdash \mathit{use} : \texttt{bool},\ \varnothing$$
$$\varnothing;\ n \vdash \mathit{passwd} : \texttt{string} \xrightarrow{\{p\}} \texttt{bool},\ \varnothing \qquad \varnothing;\ n \vdash \mathit{bad} : \texttt{bool},\ \{w\}$$

Thus *use* is unconditionally safe, but *bad* is not, although it could be used by *root* in the context of `dopriv` w.

An abstraction $\mathtt{fun}\ x.\ e$ does not require any privileges to be enabled for safe evaluation. However, the body e may require privileges Π. This is manifest in the type $\theta_1 \xrightarrow{\Pi} \theta_2$. The latent privileges, Π, get exposed during an application, $e_1 e_2$. Say e_1 has type $\theta_1 \xrightarrow{\Pi} \theta_2$; if Π_1 may be enabled during e_1's evaluation, and Π_2 may be enabled during e_2's evaluation, then application itself may require Π to be enabled; hence $\Pi \cup \Pi_1 \cup \Pi_2$ may be enabled during the evaluation of $e_1 e_2$.

The application rule also uses subtyping, which is needed at higher types because the privileges required by an argument need not exactly match those of a function's domain type. Subtyping is defined as the least relation $\leq$ with $\mathtt{bool} \leq \mathtt{bool}$ and, for arrow types, $\theta_1 \xrightarrow{\Pi_1} \theta_1' \leq \theta_2 \xrightarrow{\Pi_2} \theta_2'$ provided $\theta_2 \leq \theta_1$, $\theta_1' \leq \theta_2'$, and $\Pi_1 \subseteq \Pi_2$. To relate the semantics to the static analysis, we need the ordinary type θ^* obtained by erasing annotations. This is defined by induction on θ, to wit: $\mathtt{bool}^* = \mathtt{bool}$ and $(\theta_1 \xrightarrow{\Pi} \theta_2)^* = \theta_1^* \rightarrow \theta_2^*$. It is easy to show that if $\theta_1 \leq \theta_2$, then $\theta_1^* = \theta_2^*$. Due to subtyping, an expression can have more than one annotated type and satisfy more than one judgement. But a derivable judgement $\Delta;\ n \vdash e : \theta,\ \Pi$ has only one derivation, which is dictated by the structure of e.

Theorem 4.1 Suppose $\varnothing;\ n \vdash e : \theta,\ \Pi$ is derivable. Then for all $P \in \mathcal{P}(\mathsf{Privs})$ and $\Pi \subseteq P$, it is the case that $[\![\varnothing \vdash e : \theta^*]\!] nP\{\} \neq \star$.

The theorem is an immediate consequence of Lemma 4.3 below. In order to serve as an adequate induction hypothesis, the lemma strengthens the theorem by allowing judgements with non-empty contexts. But this is not enough. Values at arrow types are functions that depend on privilege sets. As induction hypothesis for the case of application we require these functions be safe with respect to the privilege set Π annotating their type.

Definition 4.2 For each annotated type θ the predicate $\mathsf{safe}\ \theta$ on $[\![\theta^*]\!]_{\perp\star}$ is defined as follows: $\mathsf{safe}\ \theta(\perp) \Leftrightarrow \mathsf{true}$ and $\mathsf{safe}\ \theta(\star) \Leftrightarrow \mathsf{false}$ for all θ. For values other than $\perp$ and $\star$, the definition is by induction on structure of θ.

$$\begin{array}{lcl} \mathsf{safe}\ \mathtt{bool}(b) & \Leftrightarrow & \mathsf{true} \\ \mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(f) & \Leftrightarrow & \forall d \in [\![\theta_1^*]\!].\ \forall P.\ \ \Pi \subseteq P \wedge \mathsf{safe}\ \theta_1(d) \ \Rightarrow\ \mathsf{safe}\ \theta_2(fPd) \end{array}$$

For annotated type environment Δ, the predicate $\mathsf{safe}\ \Delta$ on $[\![\Delta^*]\!]$ is defined by $\mathsf{safe}\ \Delta(h) \Leftrightarrow \forall x \in dom(h).\mathsf{safe}(\Delta.x)(h.x)$.

Lemma 4.3 Suppose $\Delta;\ n \vdash e : \theta,\ \Pi$ is derivable. Then for all $P \in \mathcal{P}(\mathsf{Privs})$, for all $h \in \Delta^*$, if $\mathsf{safe}\ \Delta(h)$ and $\Pi \subseteq P$ then $\mathsf{safe}\ \theta\ ([\![\Delta^* \vdash e : \theta^*]\!] nPh)$.

The Theorem follows because $\mathsf{safe}\ \varnothing\{\}$ and $\mathsf{safe}\ \theta([\![\varnothing \vdash e : \theta^*]\!] nP\{\})$ implies $[\![\varnothing \vdash e : \theta^*]\!] nP\{\} \neq \star$. The Lemma is proved by induction on derivation of

$\Delta;\ n \vdash e : \theta,\ \Pi$. We say "induction on e" for brevity, and use identifiers from the hypotheses of the applicable rules. Throughout the proof, we assume $P \in \mathcal{P}(\mathsf{Privs})$ and $h \in [\![\Delta^*]\!]$ and $\mathsf{safe}\ \Delta(h)$ and $\Pi \subseteq P$, and also let $u = [\![\Delta^* \vdash e : \theta^*]\!] nPh$ for each case of e.

Case $\mathtt{true}$: Then, $u = \mathsf{true}$, hence $\mathsf{safe}\ \mathtt{bool}(u)$ by definition safe.

Case x: Then, $u = h.x$ and $\mathsf{safe}\ \theta(h.x)$ follows, by definition safe, from the assumption $\mathsf{safe}\ \Delta(h)$.

Case $\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$: Then, $\Pi_1 \cup \Pi_2 \cup \Pi_3 \subseteq P$, and

$$u = \textbf{if}\ b\ \textbf{then}\ [\![\Delta^* \vdash e_1 : \theta^*]\!] nPh\ \textbf{else}\ [\![\Delta^* \vdash e_2 : \theta^*]\!] nPh$$

where $b = [\![\Delta^* \vdash e : \mathtt{bool}]\!] nPh$. By induction on e, using that $\Pi_1 \subseteq P$, we have $\mathsf{safe}\ \mathtt{bool}(b)$ and hence $b \neq \star$. If $b = \bot$ then $u = \bot$, and $\bot$ is safe. Otherwise, $b = \mathsf{true}$ or $b = \mathsf{false}$. If $b = \mathsf{true}$, we use induction on e_1, noting that $\Pi_2 \subseteq P$, to obtain $\mathsf{safe}\ \theta(u)$. The case of $b = \mathsf{false}$ is symmetric.

Case $\mathtt{fun}\ x.\ e$: Then, $u = \lambda P'.\ \lambda d.\ [\![\Delta^*, x : \theta_1^* \vdash e : \theta_2^*]\!] nP'[h \mid x \mapsto d]$. Thus $u \neq \star$. To prove $\mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(u)$, consider any $P' \in \mathcal{P}(\mathsf{Privs})$ and any $d \in [\![\theta_1^*]\!]$ such that $\Pi \subseteq P'$ and $\mathsf{safe}\ \theta_1(d)$, to show $\mathsf{safe}\ \theta_2(uP'd)$. By semantics, $uP'd = [\![\Delta^*, x : \theta_1^* \vdash e : \theta_2^*]\!] nP'[h \mid x \mapsto d]$, so the induction hypothesis for e yields $\mathsf{safe}\ \theta_2(uP'd)$ provided that $\Pi \subseteq P'$ and $\mathsf{safe}\ (\Delta, x : \theta_1)[h \mid x \mapsto d]$. We have $\Pi \subseteq P'$ by assumption, and $\mathsf{safe}\ (\Delta, x : \theta_1)[h \mid x \mapsto d]$ follows from $\mathsf{safe}\ \Delta(h)$ and $\mathsf{safe}\ \theta_1(d)$.

Case $e_1\ e_2$: Let $f = [\![\Delta^* \vdash e_1 : \theta_1^* \to \theta_2^*]\!] nPh$ and $d = [\![\Delta^* \vdash e_2 : {\theta'_1}^*]\!] nPh$, so that $u = fPd$. (Recall that $\theta'_1 \leq \theta_1$ implies ${\theta'_1}^* = \theta_1^*$ so the application fPd makes sense.) Using safety of h and the assumption $\Pi \cup \Pi_1 \cup \Pi_2 \subseteq P$, induction on e_1 gives $\mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(f)$, and induction on e_2 gives $\mathsf{safe}\ \theta'_1(d)$. Now we need the fact that for any θ, θ', $\theta \leq \theta'$ and $\mathsf{safe}\ \theta\ d$ imply $\mathsf{safe}\ \theta'\ d$. (This can be shown by a straightforward induction on derivation of $\theta \leq \theta'$.) Thus, by $\theta'_1 \leq \theta_1$, we have $\mathsf{safe}\ \theta'_1(d) \Rightarrow \mathsf{safe}\ \theta_1(d)$. Then by definition $\mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(f)$ we get $\mathsf{safe}\ \theta_2(fPd)$.

Case $\mathtt{signs}\ n'\ e$: Then, $\Pi \subseteq P$ and $u = [\![\Delta^* \vdash e : \theta^*]\!] n'(P \cap \mathcal{A}(n'))h$. The induction hypothesis on e can be used to obtain $\mathsf{safe}\ \theta(u)$, because $\Pi \subseteq (P \cap \mathcal{A}(n'))$ which follows from assumption $\Pi \subseteq P$ and side condition $\Pi \subseteq \mathcal{A}(n')$ in the hypothesis $\Delta;\ n' \vdash e : \theta,\ \Pi$ for the typing of $\mathtt{signs}\ n'\ e$.

Case $\mathtt{dopriv}\ p\ \mathtt{in}\ e$: Then, $\Pi \subseteq P$ and $u = [\![\Delta^* \vdash e : \theta^*]\!] n(P \sqcup_n \{p\})h$. By induction on e, noting that $(\Pi \sqcup_n \{p\}) \subseteq (P \sqcup_n \{p\})$, we have $\mathsf{safe}\ \theta(u)$.

Case $\mathtt{check}\ p\ \mathtt{for}\ e$: Then, $\Pi \cup \{p\} \subseteq P$, hence $p \in P$. Now $u = \textbf{if}\ p \in P\ \textbf{then}\ [\![\Delta^* \vdash e : \theta^*]\!] nPh\ \textbf{else}\ \star$. Since $p \in P$, we have, $u = [\![\Delta^* \vdash e : \theta^*]\!] nPh$ and, by induction on e, we have $\mathsf{safe}\ \theta(u)$.

Case $\mathtt{test}\ p\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$: Then, $\Pi_1 \cup \Pi_2 \subseteq P$ and $u = \textbf{if}\ p \in P\ \textbf{then}\ [\![\Delta^* \vdash e_1 : \theta^*]\!] nPh\ \textbf{else}\ [\![\Delta^* \vdash e_2 : \theta^*]\!] nPh$. In the case $p \in P$, by induction e_1 and using that $\Pi_1 \subseteq P$, we have $u = [\![\Delta^* \vdash e_1 : \theta^*]\!] nPh$

and $\mathsf{safe}\ \theta(u)$. The case of $p \notin P$ is symmetric.

Case `letrec` $f(x) = e_1$ `in` e_2: Then, $\Pi \cup \Pi_1 \subseteq P$. Now $u = [\![\Delta^*, f : \theta_1^* \to \theta_2^* \vdash e_2 : \theta^*]\!] nP[h \mid f \mapsto \mathit{fix}\ G]$, where $G(g) = \lambda P'.\ \lambda d.\ [\![\Delta^*, f : \theta_1^* \to \theta_2^*, x : \theta_1^* \vdash e_1 : \theta_2^*]\!] nP'[h \mid f \mapsto g, x \mapsto d]$. To get $\mathsf{safe}\ \theta(u)$ by induction for e_2, we need $\mathsf{safe}\ (\Delta, f : \theta_1 \xrightarrow{\Pi} \theta_2)[h \mid f \mapsto \mathit{fix}\ G]$ and $\Pi_1 \subseteq P$. The latter follows from the assumption $\Pi \cup \Pi_1 \subseteq P$. The former follows from $\mathsf{safe}\ \Delta(h)$, which is assumed, and $\mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(\mathit{fix}\ G)$, which we proceed to show. Now $\mathit{fix}\ G = \bigsqcup_i g_i$, where $g_0 = \lambda P'.\ \lambda d.\ \bot$ and $g_{i+1} = G(g_i)$. A crucial fact is that, for any θ, $\mathsf{safe}\ \theta$ preserves lubs, *i.e.*, $\forall i.\mathsf{safe}\ \theta(u_i)$ implies $\mathsf{safe}\ \theta(\bigsqcup_i u_i)$ for any ascending chain $u : \mathbb{N} \to [\![\theta^*]\!]_{\bot\star}$. This can be shown by a structural induction on θ. Thus, to show $\mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(\mathit{fix}\ G)$ it remains to show $\forall i.\ \mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(g_i)$. This we do by induction on i. For the base case, take $P' \in \mathcal{P}(\mathsf{Privs})$ and any $d \in [\![\theta_1^*]\!]$, such that $\Pi \subseteq P'$ and $\mathsf{safe}\ \theta_1(d)$. Then $g_0 P' d = \bot \neq \star$ and $\mathsf{safe}\ \theta_2(g_0 P' d)$ holds. For the induction step, assume $\mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(g_i)$. Now $g_{i+1} = G(g_i) = \lambda P'.\ \lambda d.\ [\![\Delta^*, f : \theta_1^* \to \theta_2^*, x : \theta_1^* \vdash e_1 : \theta_2^*]\!] nP[h \mid f \mapsto g_i, x \mapsto d]$. Take any $P' \in \mathcal{P}(\mathsf{Privs})$ and $d \in [\![\theta_1^*]\!]$, such that $\Pi \subseteq P'$ and $\mathsf{safe}\ \theta_1(d)$. Then $g_{i+1} P'(d) = [\![\Delta^*, f, x \vdash e_1 : \theta_2^*]\!] nP'[h \mid f \mapsto g_i, x \mapsto d]$. Note that $\mathsf{safe}\ (\Delta, f : \theta_1 \xrightarrow{\Pi} \theta_2, x : \theta_1)[h \mid f \mapsto g_i, x \mapsto d]$. Therefore, by the main induction hypothesis on the typing derivation $\Delta, f : \theta_1 \xrightarrow{\Pi} \theta_2, x : \theta_1;\ p \vdash e_1 : \theta_2,\ \Pi$, since $\Pi \subseteq P$, we obtain $\mathsf{safe}\ \theta_2(g_{i+1} P' d)$.

5 Discussion

We have studied the Java access control system using a simple denotational semantics. The semantics led to design and justification of a syntax-directed static analysis that shows certain expressions, including recommended patterns of use for the Java system, do not raise security errors. By itself, such an analysis does not ensure intrinsic security properties such as "user programs do not write the password file". Suitable use of encapsulation facilities can ensure that, *e.g.*, the *write* operation is only called by *writepass*. Full assurance depends on correctness of the compiler and Java runtime system, to ensure encapsulation, and on the underlying operating system, to ensure that the file cannot be accessed except by *write*. To reduce the size of the trusted computing base in the face of burgeoning use of mobile code, the trend is towards more link-time checking (*e.g.*, Java bytecode verification and proof-carrying code [4]). For such purposes, compositional analyses and proofs like ours have many advantages.

Wallach et al [7] study the Java system using a logic of authentication [1]. Under the name "security passing style", the eager strategy is described in terms of beliefs and inferences; using automata-theoretic notions they sketch an argument for equivalence of stack inspection and their eager semantics. This approach has the potential to address the full spectrum of reasoning about intrinsic security properties, but it would be quite indirect as a means for justifying a static analysis or program transformations. Skalka et al give operational semantics for both lazy [6] and eager [5] strategies; they use the semantics to justify static analyses that are similar to ours, and which inspired our work. The analysis in [6] is complicated by use of a constraint system, which entangles implementation with specification of the analysis. The results in [5] depend on a very general theory of typing systems. In contrast, our analysis is syntax directed and thus admits standard implementation techniques; furthermore, we give a detailed, self-contained justification for our analysis.

The main use for static analysis is performance optimization, but optimizing transformations for Java security have not previously appeared. Owing to the simplicity of our semantics, we have been able to show how, for safe expressions, `check` and `dopriv` can be eliminated without changing program behavior [2]. For example, if $p \in \mathcal{A}(n)$ and e is closed we have

$$\texttt{signs}\ n\ \texttt{dopriv}\ p\ \texttt{in}\ \texttt{check}\ p\ \texttt{for}\ e = \texttt{signs}\ n\ e$$

provided e contains no `check` p or `test` p. Transformations can move checks outward, *e.g.*, if $p \in \mathcal{A}(n)$ then $\texttt{signs}\ n\ \texttt{check}\ p\ \texttt{for}\ e = \texttt{check}\ p\ \texttt{for}\ \texttt{signs}\ n\ e$. Applying these laws and idempotence of `signs` n , we obtain

$$passwd(\text{“mypass”}) = \texttt{check}\ p\ \texttt{for}\ \texttt{signs}\ root\ write(\text{“mypass”}, \text{“passfile”})$$

and hence $use = \texttt{signs}\ user\ \texttt{signs}\ root\ write(\text{“mypass”}, \text{“passfile”})$.

References

[1] M. Abadi, M. Burrows, B. Lampson and G. Plotkin. A calculus for access control in distributed systems. *ACM Trans. Programming Languages and Systems* 15 (4), 1993.

[2] A. Banerjee and D.A. Naumann A simple semantics and static analysis for Java security. Computer Science Technical Report 2001-1, Stevens Institute of Technology, 2001. http://www.cs.stevens-tech.edu/˜naumann/tr2001.ps.

[3] Li Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.

[4] G. C. Necula. Proof-Carrying Code. *Proceedings of POPL*, 1997.

[5] F. Pottier, C. Skalka and S. Smith. A systematic approach to static access control. *Proceedings of ESOP*, 2001.

[6] C. Skalka and S. Smith. Static enforcement of security with types. *Proceedings of the fifth ACM International Conference on Functional Programming*, 2000.

[7] D. Wallach, A. Appel and E. Felten. SAFKASI: a security mechanism for language-based systems. *ACM Trans. Software Engineering and Methodology*, 2000.