A Static Type System for JVM Access Control

Tomoyuki Higuchi and Atsushi Ohori
Japan Advanced Institute of Science and Technology

This paper presents a static type system for the Java Virtual Machine (JVM) code that enforces an access control mechanism similar to that found in a Java implementation. In addition to verifying type consistency of a given JVM code, the type system statically verifies whether the code accesses only those resources that are granted by the prescribed access policy. The type system is proved to be sound with respect to an operational semantics that enforces access control dynamically, similar to Java stack inspection. This result ensures that "well typed code cannot violate access policy." The authors then develop a type inference algorithm and show that it is sound with respect to the type system. These results allows us to develop a static system for JVM access control without resorting to costly runtime stack inspection.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.2 [**Programming Languages**]: Language Classifications—*Macro and assembly languages, Object-oriented languages*; D.4.6 [**Operating Systems**]: Security and Protection—*access controls, authentication*

General Terms: Languages, Security, Theory, Verification
Additional Key Words and Phrases: JVM, access control, stack inspection, type system, type inference

1. INTRODUCTION

Access control is a mechanism to prevent an unauthorized agent (or *principal*) from accessing protected resources. This has traditionally been enforced by monitoring each user's resource access requests dynamically in a resource server, typically an operating system. This simple strategy has been based on the assumption that the user knows the semantics of a program code; thus, resource access requests issued by the code reflect the user's intention. This assumption no longer holds in the emerging network computing environment, where a program code to be executed

A preliminary summary of this paper appeared in Proceedings of ACM International Conference on Functional Programming, pages 227–237, Uppsala, Sweden, 2003.
The authors were partially supported by Grant-in-aid for scientific research on the priority area of "informatics" A01-08, grants no:15017239, 16016240. The second author was also partially supported by Grant-in-aid for scientific research (B), grant no:15300006, and by the Japan MEXT (Ministry of Education, Culture, Sports, Science and Technologies) leading project of "Comprehensive Development of Foundation Software for E-Society" under the title "dependable software development technology based on static program analysis."
Authors current address. Tomoyuki Higuchi: Insite Corporation, 2-14 Sinsenrinishimachi 1, Toyonaka, 560-0083, Japan. higuchi528@insite-corp.co.jp. Atsushi Ohori: Research Institute of Electrical Communication, Tohoku University, Katahira 2-1-1, Aobaku, Sendai 980-8577, Japan ohori@riec.tohoku.ac.jp.

is dynamically composed using various pieces downloaded from foreign sites. To cope with this situation, we need to develop *code-level access control*, in which an unauthorized principal is not a foreign user but an owner of untrusted piece of code, and the access privilege granted to a principal is a property of each code fragment.

This problem has recently attracted the attention of researchers and developers, and several verification systems have been proposed and developed. The most notable among them is perhaps the Java access control system [Gong and Schemers 1998; Gong 1999] (implemented in JDK1.2 and later.) In this system, each class (consisting of a set of methods) is owned by a principal, and each principal is assigned a set of granted privileges. In order to enforce access control, the code implementer explicitly inserts an instruction to invoke a special static method `checkPermission`, every time when accessing protected resources. This static method ensures that the principal that owns the current code has the required privilege under the current execution environment. Since method calls are typically nested and the access requests issued by a method should be regarded as those of the calling methods, `checkPermission` traverses the current call stack to ensure that all the calling methods have the required access privileges. This process is known as *stack inspection*. As reviewed in the subsequent sections, the Java access control system also provides a mechanism for a trusted code to gain a privilege, irrespective of the privileges granted to the calling methods.

Various formal properties of this approach have been studied and efficient implementation methods have been proposed. Karjoth [2000] has presented its operational semantics as a transition relation on abstract machine states. Wallach [1998] and Wallach et al. [2000] have provided denotational semantics and have proposed an alternative access control method. Banerjee and Naumann [2002] have provided a logical account. Fournet and Gordon [2002] have studied semantic properties of a program performing stack inspection. Clements and Felleisen [2003] developed an efficient implementation of stack inspection based on tail recursive semantics.

There are however some weaknesses in this approach due to its dynamic nature. One is the runtime overhead due to dynamic inspection of the call stack. Furthermore, under this approach, application of program optimization is difficult or impossible due to the requirement of maintaining the call stack. A *security-passing* method has been proposed in [Wallach 1998; Wallach et al. 2000] to reduce the runtime overhead. Instead of inspecting the call stack, this method explicitly passes security information as extra parameters so that the called method can verify access privileges. As demonstrated in [Wallach et al. 2000], this yields a more efficient alternative to stack inspection. Since this method does not assume any runtime architecture, it would also be more amenable to various optimizations. Despite these advantages, this approach is known to incur non-trivial runtime overhead for passing extra security information, which could be potentially large, as reported by Erlingsson and Shneider [2000]. Another and perhaps more fundamental limitation of this approach is that it only checks the conformance to assertions made by the programmer through explicit calls of `checkPermission`, which detects access violation only when the problematic portion of the code is executed. This simple model would be suitable for an expert to implement shared reliable code. Properly decorating the program with `checkPermission` calls prevent illegal access, and they would also be useful for documentation. However, some programmers might

fail to insert all the appropriate calls as noted in [Koved et al. 2002] or those with malicious intent would deliberately omit some of the necessary calls. Also, in many cases, runtime detection of access violation would be too late, resulting in a disastrous consequence. Since access violations may be due to simple programming errors, their early detection is highly desirable.

An alternative approach to overcome these weaknesses is to develop code-level access control as a proof system that statically verifies the desired properties of a code from the given code. Based on this insight, Skalka and Smith [2000] have developed a static type system of code-level access control for a variant of the lambda calculus by refining the type system of the lambda calculus with access privilege information. They have proved the soundness of the type system, which ensures that a well typed program will not cause security violation. Pottier et al. [2005] further refined this type system. Banerjee and Naumann [2001] have defined a denotational semantics for a language similar to the calculus considered in [Skalka and Smith 2000; Pottier et al. 2005], which provides an additional assurance of the safety of this type-based approach. As a static verification system, this approach does not incur any runtime overhead, and detects all the access violations at compile time. However, since the target language is the lambda calculus and its development relies on the properties of a static type system of the lambda calculus, their framework is not directly applicable to low-level code languages including the Java bytecode language [Lindholm and Yellin 1999].

Current standard approach to static verification of Java bytecode language is to perform type-level abstract interpretation using a type system such as [Stata and Abadi 1998; Freund and Mitchell 2003]. This general strategy has been combined with various techniques including model checking [Posegga and Vogt 1998; Klein and Wildmoser 2003] and tool support of automated theorem proving [Barthe and Dufay 2004] (see [Nipkow 2003] for other related works), and several algorithms and formalisms have been proposed. The main target of these static verification systems so far proposed are type and memory safety (see [Leroy 2003] for a survey in this area), and there does not seem to exist static type system for code level access control. Since underlying type systems only check type consistency of each instruction against the machine state and do not reflect static semantics of the code, it is not immediately obvious that the idea of static access effect as exploited in [Skalka and Smith 2000; Pottier et al. 2005] can be smoothly integrated.

The aim of the present study is to establish a static type system for code-level access control of the Java bytecode language and to show that it is correct with respect to a standard model of JVM execution. We base our development on our earlier work [Higuchi and Ohori 2002], in which we have shown that the JVM language can be regarded as a typed term calculus based on a proof theory for low-level code [Ohori 1999]. In this formalism, the type system represents static semantics of a code language, as the type system of the lambda calculus does. This property allows us to transfer the concepts and methods developed for the lambda calculus in [Skalka and Smith 2000] to the Java bytecode language.

On the basis of this general strategy, we define a type system for access control of the Java bytecode language and establish its soundness with respect to an operational semantics that closely models the Java access control via stack inspection. For this type system, we develop a type inference algorithm. These results yield a

static access control system that automatically detects all the possible access violations statically from a given code and infers a minimal set of privileges required to execute the code. Since the type system directly examines each resource access through method invocation instructions, explicit insertion of `checkPermission` is unnecessary. As being an inference system for bytecode, our system does not rely on any property of a Java compiler or other that generate the target bytecode. Due to this property, our system can be used for verifying un-trustworthy and potentially malicious code.

As we shall discuss in Section 9, our static approach can be considered weaker than dynamic stack inspection in the following points. Firstly, the inferred access violation information is static approximation, and it does not necessary imply that the code will actually cause access violation when executed. Secondly, it cannot accurately infer target objects of privileged operations. Despite these limitation, the benefit of static verification mentioned above would potentially be significant. We believe that the type system presented in this paper will serve as a viable approach that complements currently popular dynamic approach.

The rest of the paper is organized as follows. Section 2 outlines our approach. Section 3 defines the target language and the type system. Section 4 defines an operational semantics of the target language. Section 5 shows that the type system is sound with respect to the semantics. Section 6 develops a type inference algorithm and proves its soundness. These results yield a static verification system for access control. However, compared with JVM stack inspection, it is still weaker in that it does not allow specification of target resources and dynamic class loading. Sections 7 and 8 describe possible extensions for dealing with these features. Section 9 discusses several extensions and implementation. Section 10 concludes the paper.

2. OUR APPROACH

In the framework proposed by Skalka and Smith [2000], a function has a type of the form

$$\tau_1 \xrightarrow{\Pi} \tau_2$$

indicating the fact that the function takes an argument of type τ_1 and computes a result of type τ_2 *using privileges* Π. To apply this idea to a low-level language, we need to define a type system that deduces a static semantics of a given code. In [Higuchi and Ohori 2002], we have developed a type-theoretical framework for the Java bytecode language, in which a JVM block B is represented as a static judgment of the form $\Delta \rhd B : \tau$ indicating the property that B computes a value of type τ using a stack of type Δ. For simplicity of presentation, we ignore local variable environments, which are irrelevant to the approach presented here. The type system is constructed based on a proof-theoretical interpretation of low-level machine instructions [Ohori 1999]. In this formalism, an ordinary instruction I, which changes a machine state Δ to a new state Δ', is regarded as a left-rule of the form

$$\frac{\Delta' \rhd B : \tau}{\Delta \rhd I{\cdot}B : \tau}$$

in a sequent-style proof system. A `return` statement corresponds to an initial sequent (the axiom in the proof system) of the form

$$\tau{\cdot}\Delta \ \rhd \texttt{return} \ : \ \tau.$$

A jump instruction refers to an existing code block (proof) through a label, and it can be represented as a meta-level rule of the form

$$\Delta \ \rhd \texttt{goto}(l) \ : \ \tau \quad (\text{if } \mathcal{L}(l) = \Delta \rhd \tau)$$

where $\mathcal{L}$ is an environment describing the typing of each entry point of the existing blocks (i,e., the end sequent of existing proofs.)

We refine this formalism to introduce access control information by interpreting a code block B as a judgment of the form

$$\Pi; \Delta; p \ \rhd B \ : \ \tau$$

indicating the fact that B is *owned by principal* p, and it computes a value of type τ from a stack of type Δ, *using a privilege set* Π.

In order to define a type system for deducing a sequent of the above form, we must describe the relationship between the privilege set Π and the block B according to its behavior with respect to resource access. For this purpose, we associate each method with the set of privileges Π' required to execute it and consider Π in the sequent as the capability to invoke a method whose required privileges are included in Π. A method then has a type of the form $\Delta \xrightarrow{\Pi} \tau$, similar to function type in [Skalka and Smith 2000]. In order to properly grant some privilege to a code through a special instruction similar to `doPrivileged` in Java, the type system maintains an access policy, which describes the maximum set of privileges for each principal.

The type system checks all the method invocations in a given bytecode statically. This mechanism frees the programmer from the responsibility of checking access privilege using `checkPermission`. The only thing required is to declare a type of the form $\Delta \xrightarrow{\Pi} \tau$ for each trusted method so that Π represents the set of privileges used by the method. These trusted methods are typically native methods in a system library, for which we can safely assume that the correct type has already been declared.

3. THE JVM ACCESS CONTROL CALCULUS

To present our method, we define a calculus, JVM_{sec}, and its static type system. JVM_{sec} is a language similar to the Java bytecode language, but it contains minimal features that are sufficient to present our method. In this section, we do not consider *dynamic class loading* and assume that the set of all class files of a program is given statically. We discuss dynamic class loading in Section 8.

3.1 Notations

In the subsequent development, we shall use the following notations. Let S be a sequence and e be an element. $|S|$ is the length of S. $e{\cdot}S$ is the sequence obtained by adding e at the front of S. $S.i$ is the i-th element of S. We also write $S_1{\cdot}S_2$ for the concatenation of S_1 and S_2. $S\{i \leftarrow e\}$ is the sequence obtained from S by replacing

$$
\begin{array}{rcl}
\Sigma & ::= & \{c_1 = (p_1, C_1), \cdots, c_n = (p_n, C_n)\} \\
C & ::= & \{m_1 = M_1, \cdots, m_n = M_n\} \\
M & ::= & \{l_1 : B_1, \cdots, l_n : B_n\} \quad \text{(where the set } \{l_1, \cdots, l_n\} \text{ contains the special label "}\mathit{entry}\text{")} \\
B & ::= & \texttt{goto}(l) \mid \texttt{return} \mid I \cdot B \\
I & ::= & \texttt{acc}(n) \mid \texttt{iconst}(n) \mid \texttt{dup} \mid \texttt{ifeq}(l) \mid \texttt{priv}(\pi) \mid \texttt{new}(c) \mid \texttt{invoke}(c, m)
\end{array}
$$

Fig. 1. Syntax of programs

the i-th element of S with e. We use a similar notation for finite functions, i.e., $f\{x \leftarrow v\}$ denotes the function f' such that $Dom(f') = Dom(f) \cup \{x\}$, $f'(x) = v$ and $f'(y) = f(y)$ for any $y \neq x$.

3.2 The Syntax of JVM$_{sec}$

We assume that we are given sets of *privileges* (ranged over by π), *principals* (ranged over by p), *class names* (ranged over by c), *method names* (ranged over by m), and *block labels* (ranged over by l). A privilege π is an atom representing some privileged operation. In the JDK access control architecture, a privilege (or "permission" in JDK parlance) consists of an operation and its target object. In Section 7, we shall describe a mechanism to extend our formalism to include target specification in π.

In the actual Java bytecode language, a program consists of a set of class files, and each class file contains method definitions and their type declarations. For simplicity of presentation, we distinguish method definitions from type declarations and refer to the former as a *program* and the latter as a *type specification.*

Figure 1 defines the syntax for programs (ranged over by Σ) through *class definitions* (ranged over by C), *method bodies* (ranged over by M), *code blocks* (ranged over by B), and *instructions* (ranged over by I). A program Σ is a set of classes, represented by a finite function on a set of class names. $\Sigma(c) = (p, C)$ indicates that class c is owned by principal p and consists of a set of methods described in C. Each method M in a class is implicitly owned by the principal that owns the class. We occasionally write M^p for a method M owned by p. We also write M_c for a method M in class c. For simplicity, we use the notation $mbody(\Sigma, c, m)$ for a method M^p such that $\Sigma(c) = (p, C)$ and $C(m) = M^p$. A method M is a collection of labeled code blocks. A code block B is a sequence of instructions terminated with `return` or `goto`. `acc(`n`)` pushes the n-th value of the stack onto the top of the stack. This instruction is added to make the set of instructions non trivial though it does not have much significance in our type system. `priv` corresponds to `doPrivileged` method in the Java access control architecture. The effect of `priv(`π`)` in a block of the form `priv(`π`)`$\cdot B$ is to check whether or nor the access policy allows the current principal to use π, and if this is the case then to extend the current privilege set with π so that B can use it.

A *type specification* consists of a *class specification* (ranged over by Θ), a *native method specification* (ranged over by $\mathcal{N}$), and an *access policy* (ranged over by $\mathcal{A}$). Their syntax is given in Figure 2. A class specification Θ specifies for each class name its method typings including those for native methods. This structure models explicit type information specified in JVM class files. Δ is a sequence of types and

$$
\begin{array}{rcl}
\Theta & ::= & \{c_1 = spec_1, \cdots, c_n = spec_n\} \\
spec & ::= & \{m_1 = \Delta_1 \rightarrow \tau_1, \cdots, m_n = \Delta_n \rightarrow \tau_n\} \\
\tau & ::= & \mathtt{int} \mid c \\
\Delta & ::= & \emptyset \mid \tau{\cdot}\Delta \\
\mathcal{N} & ::= & \{(c_1, m_1) \mapsto \Pi_1, \cdots, (c_n, m_n) \mapsto \Pi_n\} \\
\mathcal{A} & ::= & \{p_1 \mapsto \Pi_1, \cdots, p_n \mapsto \Pi_n\}
\end{array}
$$

Fig. 2. Syntax of type specification

is used for argument types of a method and for runtime stack types. We adopt the convention that the leftmost element corresponds to the top of the stack. $\Delta \rightarrow \tau$ represents the type of a method that takes arguments of type Δ and returns a value of type τ. A native method specification $\mathcal{N}$ and an access policy $\mathcal{A}$ are introduced for specifying access control information. $\mathcal{N}(c, m) = \Pi$ represents the fact that a native method m in a class c uses privileges in Π. $\mathcal{A}(p) = \Pi$ indicates that the principal p can use the privileges Π.

Actual Java class files also include subclass declarations that determine a subclass relation. Here we assume that a (well-formed) set of subclass declarations is implicitly given, and it determines a subclass relation $c_1 <: c_2$. We further extend this relation to τ by setting $\mathtt{int} <: \mathtt{int}$ as well as to stack types by setting $\Delta <: \Delta'$ if $\Delta.i <: \Delta'.i$ for each i.

We assume that the user sets up a native method specification $\mathcal{N}$ and an access policy $\mathcal{A}$ statically and globally. It must be noted that Θ, $\mathcal{N}$, and $\mathcal{A}$ are extra inputs used as constraints in specifying typing relations and not parts of the static environment of typing derivation.

3.3 The Type System

The type system is constructed in such a manner that it type-checks all classes of a program simultaneously. To deal with the mutually recursive nature of classes and methods, typing relations are defined relative to a *privilege environment* $\mathcal{P}$ of the form

$$\mathcal{P} ::= \{(c_1, m_i) \mapsto \Pi_1, \cdots, (c_n, m_k) \mapsto \Pi_l\}.$$

$\mathcal{P}(c, m) = \Pi$ denotes the fact that method m defined in class c requires privileges Π. The domain of $\mathcal{P}$ and that of $\mathcal{N}$ are disjoint, and the following properties hold: for each class c in Θ, for each method m in c, $(c, m) \in Dom(\mathcal{P})$ or $(c, m) \in Dom(\mathcal{N})$.

Corresponding to the structure of JVM_{sec} program, the type system consists of typing relations for code blocks, methods, and programs. We define them in this order.

Block typing. As outlined in Section 2, the type system for blocks is defined as a proof system to derive a judgment of the form

$$\Pi; \Delta; p \ \rhd B \ : \ \tau$$

relative to a privilege environment $\mathcal{P}$. In addition to $\mathcal{P}$, we need to introduce a *label environment* $\mathcal{L}$ to describe the types of the set of code blocks of a method to which B belongs. This is necessary since block definitions are mutually recursive

$$\Pi;\tau\cdot\Delta;p\ \rhd \texttt{return} : \tau \qquad \Pi;\Delta;p\ \rhd \texttt{goto}(l)\ :\ \tau \quad (\text{if } \mathcal{L}(l) = \Pi';\Delta;p \rhd \tau \text{ and } \Pi' \subseteq \Pi)$$

$$\frac{\Pi;\Delta;p\ \rhd B\ :\ \tau}{\Pi;\texttt{int}\cdot\Delta;p\ \rhd \texttt{ifeq}(l)\cdot B\ :\ \tau} \quad (\text{if } \mathcal{L}(l) = \Pi';\Delta;p \rhd \tau \text{ and } \Pi' \subseteq \Pi)$$

$$\frac{\Pi;(\Delta.n)\cdot\Delta;p\ \rhd B\ :\ \tau}{\Pi;\Delta;p\ \rhd \texttt{acc}(n)\cdot B\ :\ \tau}\ (\text{if } n < |\Delta|) \qquad \frac{\Pi;\texttt{int}\cdot\Delta;p\ \rhd B\ :\ \tau}{\Pi;\Delta;p\ \rhd \texttt{iconst}(n)\cdot B\ :\ \tau}$$

$$\frac{\Pi;\tau\cdot\tau\cdot\Delta;p\ \rhd B\ :\ \tau}{\Pi;\tau\cdot\Delta;p\ \rhd \texttt{dup}\cdot B\ :\ \tau} \qquad \frac{\Pi;c\cdot\Delta;p\ \rhd B\ :\ \tau}{\Pi;\Delta;p\ \rhd \texttt{new}(c)\cdot B\ :\ \tau}$$

$$\frac{\pi\cdot\Pi;\Delta;p\ \rhd B\ :\ \tau}{\Pi;\Delta;p\ \rhd \texttt{priv}(\pi)\cdot B\ :\ \tau}\ (\text{if } \pi \in \mathcal{A}(p)) \qquad \frac{\Pi;\Delta;p\ \rhd B\ :\ \tau}{\Pi;\Delta;p\ \rhd \texttt{priv}(\pi)\cdot B\ :\ \tau}\ (\text{if } \pi \notin \mathcal{A}(p))$$

$$\frac{\Pi;\tau_1\cdot\Delta;p\ \rhd B\ :\ \tau}{\Pi;\Delta_0\cdot c_0\cdot\Delta;p\ \rhd \texttt{invoke}(c,m)\cdot B\ :\ \tau}\ \left(\text{if}\ \begin{array}{l} \mathit{mtype}(\Theta,c,m) = \Delta_1 \to \tau_1, \\ \Delta_0 <: \Delta_1, \\ c_0 <: c, \text{ and} \\ \mathit{AllPrivs}(\mathcal{P},c,m) \subseteq \Pi \end{array}\right)$$

Fig. 3. Typing rules for code block B

through label references. A label environment is a finite function of the form $\{l_1 \mapsto (\Pi_1;\Delta_1;p \rhd \tau), \cdots, l_n \mapsto (\Pi_n;\Delta_n;p \rhd \tau)\}$ on the set $\{l_1,\cdots,l_n\}$ of labels used in a method. We write $\mathcal{P};\mathcal{L} \vdash \Pi;\Delta;p\ \rhd B\ :\ \tau$ if $\Pi;\Delta;p\ \rhd B\ :\ \tau$ is derivable under $\mathcal{P}$ and $\mathcal{L}$. Since there is no rule that changes $\mathcal{P}$ or $\mathcal{L}$, we treat $\mathcal{P}$ and $\mathcal{L}$ as global variables in defining each typing rule.

The set of typing rules is given in Figure 3. The rules other than those for `priv` and `invoke` are essentially the same as in [Higuchi and Ohori 2002]. The rule for `priv`(π) adds π to the current privilege set if the current principal has access privilege π under the access policy $\mathcal{A}$. The rule for `invoke` first verifies the type consistency using an auxiliary function $\mathit{mtype}(\Theta,c,m)$, which returns the type specification of m for c. It also verifies that the privileges used by the method are included in the current privilege set Π. This implies that the method being invoked must have a method of type $\Delta \xrightarrow{\Pi'} \tau$ such that $\Pi' \subseteq \Pi$. For a simple type discipline, such as that of the lambda calculus, this can be done by computing the type of m for c and checking the inclusion. However, since in JVM_{sec}, the method actually called is determined at runtime based on the runtime class of the receiver object, this constraint must be checked against all the possible methods that may be called. $\mathit{AllPrivs}(\mathcal{P},c,m) \subseteq \Pi$ guarantees this constraint. $\mathit{AllPrivs}(\mathcal{P},c,m)$ looks up all the methods that can be invoked by `invoke`(c,m) and returns the union of their privilege sets. The definitions for $\mathit{AllPrivs}$ and mtype are given in Figure 4. They use an auxiliary function $\mathit{mclass}(\Theta,c,m)$, which returns the closest superclass of c that defines a method m. We note that this check is possible only if all the classes of a program are statically given prior to the type checking of the program. To support dynamic class loading, we need some mechanism to delay this check, which we shall discuss in Section 8.

$mtype(\Theta, c, m) =$ let $c_1 = mclass(\Theta, c, m)$ in $\Theta(c_1)(m)$

$mclass(\Theta, c, m) =$ the least class c_0 such that $m \in Dom(\Theta(c_0))$ and $c <: c_0$.

$AllPrivs(\mathcal{P}, c, m) =$
  let $\Pi_1 = \bigcup\{\Pi \mid \mathcal{P}(c', m) = \Pi$ for some c' such that $c' <: c\}$
    $\Pi_2 = \bigcup\{\Pi \mid \mathcal{N}(c', m) = \Pi$ for some c' such that $c' <: c\}$
  in if $(c, m) \in Dom(\mathcal{P})$ or $(c, m) \in Dom(\mathcal{N})$ then $\Pi_1 \cup \Pi_2$
    else let $c_1 = mclass(\Theta, c, m)$
      in if $(c_1, m) \in Dom(\mathcal{P})$ then $\Pi_1 \cup \Pi_2 \cup \mathcal{P}(c_1, m)$
        else $\Pi_1 \cup \Pi_2 \cup \mathcal{N}(c_1, m)$

Fig. 4. The definitions for the auxiliary functions used in typing rules

$\mathcal{A}; \Theta; \mathcal{N} \vdash \Sigma : \mathcal{P}$
  $\iff \vdash \Theta$ and for each $c \in Dom(\Sigma)$, for each m in c, the following typing holds:
    $\mathcal{P} \vdash mbody(\Sigma, c, m) : \Delta \xrightarrow{\Pi} \tau$ such that $\Theta(c)(m) = \Delta \to \tau$ and $\mathcal{P}(c, m) = \Pi$.

$\vdash \Theta \iff$ if $c_0 <: c_1$, $m \in Dom(\Theta(c_0))$, and $m \in Dom(\Theta(c_1))$ then $\Theta(c_1)(m) = \Theta(c_2)(m)$.

Fig. 5. Typing rules for programs

Method typing. A method is a set of labeled code blocks for which we first define the following typing relation:

$$\mathcal{P} \vdash M^p : \mathcal{L} \iff \text{for each } l \in Dom(M^p), \text{ the following conditions hold:}$$
$$\mathcal{L}(l) = \Pi; \Delta; p \rhd \tau\,,$$
$$\mathcal{P}; \mathcal{L} \vdash \Pi; \Delta; p \ \rhd M^p(l) \ : \ \tau\,, \text{ and } \Pi \subseteq \mathcal{A}(p)$$

indicating the fact that the set of labeled blocks in M is typed by the label environment $\mathcal{L}$. The condition $\Pi \subseteq \mathcal{A}(p)$ enforces the constraint that each code block only uses the privileges granted by the access policy $\mathcal{A}$.

Since the type of a method is the type of the entry block, we define method typing as follows.

$$\mathcal{P} \vdash M^p_c : \Delta \xrightarrow{\Pi} \tau \iff \text{there exists some } \mathcal{L} \text{ such that } \mathcal{P} \vdash M^p : \mathcal{L}, \text{ and}$$
$$\mathcal{P}; \mathcal{L} \vdash \Pi; \Delta{\cdot}c{\cdot}\emptyset; p \ \rhd M^p(entry) \ : \ \tau$$

Program typing. Using these definitions, we define the type correctness of a program as the following property. A program Σ with its type specifications $\mathcal{A}$, Θ, and $\mathcal{N}$ is well typed with a privilege environment $\mathcal{P}$, written as $\mathcal{A}; \Theta; \mathcal{N} \vdash \Sigma : \mathcal{P}$ if Θ is well formed (denoted by $\vdash \Theta$), and for each class in the program, all methods of the class are well typed. These relations are given in Figure 5.

3.4 Examples

In this subsection, we show a few typing derivations of sample programs. We assume that the system's access policy $\mathcal{A}$ is declared as follows:

$$\mathcal{A}(Applet) \ = \ \emptyset$$
$$\mathcal{A}(System) \ = \ \{FRead\},$$

where *Applet* is an untrusted principal, *System* is a trusted principal, and $FRead$ is a privilege for file read access.

Let `IO` be a system class whose principal is *System* and let us assume that it includes a native method `readFile` that accepts a file name and returns the contents of the file as a string. To prevent untrusted code from reading arbitrary files using the `readFile` method, suppose that a native method specification $\mathcal{N}$ is declared as $\mathcal{N}(IO, readFile) = \{FRead\}$.

Consider the following untrusted method in some class of *Applet* principal:

```
peekPassword() {
   new(IO);                  // make an instance of IO
   sconst("/etc/password"); // push a file name onto the stack
   invoke(IO,readFile);     // invoke readFile with the file name
   return;     }             // return the contents of the file
```

which attempts to read a password file by directly invoking the `readFile` method. `sconst(`s`)` is an instruction for pushing string s onto the stack, whose typing rule is similar to that of `iconst(`n`)`. For this method, our type system statically detects a security violation in the following steps. The type system attempts to construct a typing derivation tree for a block under a privilege environment $\mathcal{P}$ by traversing the instructions of the block in order, starting from `return`. Since $FRead \notin \mathcal{A}(Applet)$, in each inference step, it should be the case that the current privilege set Π does not include $FRead$. At the `invoke(IO,readFile)` instruction, the type system computes $AllPrivs(\mathcal{P}, IO, readFile) = \{FRead\}$ form $\mathcal{N}(IO, readFile) = \{FRead\}$. Since $\{FRead\} \not\subseteq \Pi$, it detects the violation of the typing condition for `invoke` instruction. Hence, the typing derivation for the method fails.

Next, we consider the following example involving inheritance:

```
peekPassword() {
   new(Dummy);               // make an instance of Dummy
   sconst("/etc/password"); // push a file name to stack
   invoke(Dummy,readFile)   // invoke readFile with the file name
   return;     }             // return the contents of the file
```

where `Dummy` is defined as a subclass of `IO` as follows:

```
class Dummy extends IO {   // no methods are defined  }
```

Since the `invoke(Dummy,readFile)` instruction in the method `peekPassword` actually calls `readFile` of class `IO` at runtime, this code also causes a security violation. Our type system also detects a security violation of this method, when it checks instruction `invoke(Dummy,readFile)`. The class `Dummy` does not contain method `readFile` and it is declared as a subclass of `IO`. Using these facts, the type system correctly computes $AllPrivs(\mathcal{P}, Dummy, readFile) = \{FRead\}$ and detects the security violation, as in the previous example.

The next example illustrates the use of `priv` by a trusted method to override the current access policy. Let `System` be a *System* class and `Applet` be an *Applet* class, and consider the following method defined in `System`:

```
readMe() {
```

Typing derivation for `readMe` :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\texttt{FRead}\};\texttt{str}\cdot\emptyset;\mathit{System}\ \triangleright \texttt{return : str}}}{\{\texttt{FRead}\};\texttt{str}\cdot\texttt{IO}\cdot\emptyset;\mathit{System}\ \triangleright \texttt{invoke(IO ,readFile) : str}}}{\emptyset;\texttt{str}\cdot\texttt{IO}\cdot\emptyset;\mathit{System}\ \triangleright \texttt{priv(fileOpen) : str}}}{\emptyset;\texttt{IO}\cdot\emptyset;\mathit{System}\ \triangleright \texttt{sconst(``README!'') : str}}}{\emptyset;\emptyset;\mathit{System}\ \triangleright \texttt{new(IO) : str}}$$

Typing derivation for `getFile` :

$$\dfrac{\dfrac{\dfrac{}{\emptyset;\texttt{str}\cdot\emptyset;\mathit{Applet}\ \triangleright \texttt{return : str}}}{\emptyset;\texttt{str}\cdot\texttt{System}\cdot\emptyset;\mathit{Applet}\ \triangleright \texttt{invoke(System,readMe) : str}}}{\emptyset;\emptyset;\mathit{Applet}\ \triangleright \texttt{new(System) : str}}$$

Fig. 6. Examples of typing derivation

```
   new(IO);                  // make an instance of IO
   sconst("README!");        // push a file name onto the stack
   priv(FRead);              // enable a privilege FRead
   invoke(IO,readFile);      // invoke readFile with the file name
   return;     }             // return the contents of the file
```

Any method can invoke the above method, even if it does not have a privilege FRead. Using this method, `Applet` class can define the following method for reading the `README!` file.

```
getFile() {
   new(System);              // make an instance of System
   invoke(System,readMe);    // invoke readMe to access README! file
   return;     }             // return the contents of the file
```

These methods are indeed type correct, as shown in Figure 6. This figure only shows the first instruction of code block in each sequent.

4. OPERATIONAL SEMANTICS

An operational semantics of JVM_{sec} is defined by specifying the effect of each instruction as a transition rule on machine states of the form

$$(P, S, M^p\{B\}, D); h$$

consisting of a *dynamic privilege set* P, a *runtime stack* S, a code block $M^p\{B\}$ in a method, a *dump* D, and a *heap* h. P is a finite set of privileges that the block can currently use. $M^p\{B\}$ is the current code block B belonging to method M^p. Figure 7 gives the definitions for runtime values (ranged over by v), stacks (S), dumps (D), and heaps (h). Following [Skalka and Smith 2000], we model access violation by a special value `secfail`. If an access violation occurs at runtime, the machine terminates with this special value. Introducing this special value is necessary for distinguishing access violation from type error, which is modeled by the value `wrong`. An ordinary runtime value is either an integer n or a heap address r. $\langle\rangle_c$ is an instance object of class c. Since we omitted object fields, an object

$$
\begin{array}{rcl}
v & ::= & n \mid r \mid \texttt{secfail} \mid \texttt{wrong} \\
S & ::= & \emptyset \mid v{\cdot}S \\
D & ::= & \emptyset \mid (P, S, M^p\{B\}){\cdot}D \\
h & ::= & \{r_1 \mapsto \langle\rangle_{c_1}, \ldots, r_n \mapsto \langle\rangle_{c_n}\}
\end{array}
$$

Fig. 7. Runtime objects

instance is empty. As we shall discuss in Section 9.2, there is no difficulty in extending our formalism to objects with field values.

A native method is an external function whose dynamic semantics is already given. We treat each of those native methods as an *opaque function* f that accepts argument values and a heap and returns a computed value and a modified heap. We assume that each opaque function satisfies its type specification defined in Θ. A precise definition of the satisfaction relation will be given when we prove the type soundness theorem. These opaque functions are given in a *native method environment* Ω of the following form.

$$\Omega ::= \{(c_1, m_1) \mapsto f_1, \cdots, (c_n, m_n) \mapsto f_n\}$$

We now define for each instruction a state transition rule of the form

$$\mathcal{A}; \Theta; \mathcal{N}; \Omega; \Sigma \vdash (P, S, M^p\{I{\cdot}B\}, D); h \longrightarrow (P', S', M'^{p'}\{B'\}, D'); h'$$

describing the fact that I transforms a machine state $(P, S, M^p\{I{\cdot}B\}, D); h$ to another state $(P', S', M'^{p'}\{B'\}, D'); h'$ under $\mathcal{A}$, Θ, $\mathcal{N}$, Ω, and Σ. The set of transition rules is given in Figure 8. (The contexts $\mathcal{A}$, Θ, $\mathcal{N}$, Ω, and Σ, which are not changed during the execution, are omitted.) The function $ArgSize(\Theta,\mathrm{c},\mathrm{m})$ used in the figure returns the number of the arguments of a method m in a class c.

The only instruction that may cause a security error is `invoke`. There are three transitions for `invoke`. In the case of a non-native method, the machine first locates the method M' and its class c_1 and principal p' using the dynamic class c_0 of the given object. It then updates the dynamic privilege set to the intersection of the current set P and the set $\mathcal{A}(p')$ of privileges granted to the method code, and it transfers control to the method code. This corresponds to the Java Security Architecture where protection domain is associated with source code. Note that, for type correct code, it is always the case that the dynamic class c_0 of the object is a subclass of the class c specified in the instruction. This is guaranteed by the type system and the type soundness theorem. In the case of a native method, if the set of privileges required by the method is included in the current dynamic privilege set, then the method succeeds with a value of appropriate type; otherwise the method invocation is aborted and the system returns the special value `secfail` indicating access violation.

An observant reader may have noted that the operational semantics is based on *eager* semantics for security checking, i.e., a new privilege set P is calculated at every method call by `invoke`; therefore, security verification can be performed by checking the current frame without traversing the entire frame stack. Since a method call occurs frequently, eager semantics may incur high runtime overhead

$(P, v{\cdot}S, M^p\{\texttt{return}\}, \emptyset); h \longrightarrow (\emptyset, v, \emptyset, \emptyset); h$

$(P, v{\cdot}S, M^p\{\texttt{return}\}, (P_0, S_0, M_0^{p_0}\{B_0\}){\cdot}D); h \longrightarrow (P_0, v{\cdot}S_0, M_0^{p_0}\{B_0\}, D); h$

$(P, S, M^p\{\texttt{goto}(l)\}, D); h \longrightarrow (P, S, M^p\{M^p(l)\}, D); h$

$(P, S, M^p\{\texttt{acc}(n){\cdot}B\}, D); h \longrightarrow (P, (S.n){\cdot}S, M^p\{B\}, D); h \quad (\text{if } n < |S|)$

$(P, S, M^p\{\texttt{iconst}(n){\cdot}B\}, D); h \longrightarrow (P, n{\cdot}S, M^p\{B\}, D); h$

$(P, v{\cdot}S, M^p\{\texttt{dup}{\cdot}B\}, D); h \longrightarrow (P, v{\cdot}v{\cdot}S, M^p\{B\}, D); h$

$(P, 0{\cdot}S, M^p\{\texttt{ifeq}(l){\cdot}B\}, D); h \longrightarrow (P, S, M^p\{M(l)\}, D); h$

$(P, n{\cdot}S, M^p\{\texttt{ifeq}(l){\cdot}B\}, D); h \longrightarrow (P, S, M^p\{B\}, D); h \quad \text{if } n \neq 0$

$(P, S, M^p\{\texttt{new}(c){\cdot}B\}, D); h \longrightarrow (P, r{\cdot}S, M^p\{B\}, D); h\{r \leftarrow \langle\rangle_c\} \quad \text{where } r \notin Dom(h)$

$$(P, S_1{\cdot}r{\cdot}S, M^p\{\texttt{invoke}(c, m){\cdot}B\}, D); h \longrightarrow (P', S_1{\cdot}r{\cdot}\emptyset, M'^{p'}\{M'^{p'}(entry)\}, (P, S, M^p\{B\}){\cdot}D); h$$
$$\text{if} \begin{cases} h(r) = \langle\rangle_{c_0}, \quad c_1 = mclass(\Theta, c_0, m), \quad ArgSize(\Theta, c_1, m) = |S_1|, \quad (c_1, m) \notin Dom(\mathcal{N}), \\ mbody(\Sigma, c_1, m) = M'^{p'}, \text{ and } P' = P \cap \mathcal{A}(p') \end{cases}$$

$$(P, S_1{\cdot}r{\cdot}S, M^p\{\texttt{invoke}(c, m){\cdot}B\}, D); h \longrightarrow (P, v{\cdot}S, M^p\{B\}, D); h'$$
$$\text{if} \begin{cases} h(r) = \langle\rangle_{c_0}, \quad c_1 = mclass(\Theta, c_0, m), \quad ArgSize(\Theta, c_1, m) = |S_1|, \quad (c_1, m) \in Dom(\mathcal{N}), \\ \mathcal{N}(c_1, m) \subseteq P, \quad \Omega(c, m) = f, \quad \text{and} \quad (v, h') = f(S_1, h) \end{cases}$$

$$(P, S_1{\cdot}r{\cdot}S, M^p\{\texttt{invoke}(c, m){\cdot}B\}, D); h \longrightarrow (\emptyset, \texttt{secfail}, \emptyset, \emptyset); h$$
$$\text{if } h(r) = \langle\rangle_{c_0}, \quad c_1 = mclass(\Theta, c_0, m), \quad (c_1, m) \in Dom(\mathcal{N}), \quad \text{and} \quad \mathcal{N}(c_1, m) \not\subseteq P$$

$$(P, S, M^p\{\texttt{priv}(\pi){\cdot}B\}, D); h \longrightarrow (P', S, M^p\{B\}, D); h$$
$$\text{if } \pi \in \mathcal{A}(p) \text{ then } P' = \{\pi\} \cup P \text{ else } P' = P$$

$(P, S, M^p\{B\}, D); h \longrightarrow (\emptyset, \texttt{wrong}, \emptyset, \emptyset); h \quad \text{if no other rule applies}$

Fig. 8. Transition rules for instructions

due to computation of P. For this reason, most implementations including JDK [Gong and Schemers 1998; Gong 1999] adopt *lazy* semantics, in which calculation of the effective privilege set is performed by stack inspection only when the security check is actually required. The trade-off between eager and lazy semantics may be important for access-control systems based on dynamic checking of privilege information. It should be noted, however, that this issue is not relevant to the present study. The operational semantics is only defined to prove the soundness of the type system. The type soundness theorem shown the following section guarantees that a type-checked program will never cause security violation. We therefore use the operational semantics that does not perform any runtime access check for actual execution. Since it is proved that eager and lazy semantics are equivalent [Banerjee and Naumann 2001; Fournet and Gordon 2002], we have chosen an eager semantics as it yields a simpler proof of the soundness theorem.

$\models h : H$ if $Dom(h) = Dom(H)$ and for any $r \in Dom(h)$ if $h(r) = \langle\rangle_c$ then $c <: H(r)$.

$H \models n : \texttt{int}$

$H \models r : \tau$ if $H(r) <: \tau$.

$H \models S : \Delta$ if $Dom(S) = Dom(\Delta)$ and $H \models S.i : \Delta.i$ for each i.

$H; \mathcal{P} \models \emptyset : \tau$ (for any τ)

$H; \mathcal{P} \models (P, S, M^p\{B\})\cdot D : \tau$ if there are some $\mathcal{L}$, Δ, Π, τ' such that $\mathcal{P} \vdash M^p : \mathcal{L}$, $H \models S : \Delta$, $\Pi \subseteq P$, $\mathcal{P}; \mathcal{L} \vdash \Pi; \tau\cdot\Delta; p \rhd B : \tau'$, and $H; \mathcal{P} \models D : \tau'$.

Fig. 9. Typing relations for runtime structures

5. TYPE SOUNDNESS

In order to show that our type system enforces the desired access control, we show the soundness of the type system with respect to the operational semantics defined above.

We first define typing relations for the following runtime structures.

$\models h : H$	heap h has heap type H
$H \models v : \tau$	value v has type τ under H
$H \models S : \Delta$	stack S has type Δ under H
$H; \mathcal{P} \models D : \tau$	dump D has type τ under H and $\mathcal{P}$

A heap type H is a mapping from heap addresses to types. This is similar to store type [Leroy 1992] and is needed for the soundness theorem we shall establish below to scale to heaps containing cyclic structures [Higuchi and Ohori 2002]. We say that a heap type H' is an extension of H if $H \subseteq H'$ (regarded as graphs.) The last relation means that a dump D accepts a value of τ and resumes the saved computation. Figure 9 shows the definitions of these relations.

Using these definitions, we define a machine state typing as follows:

$$
\begin{aligned}
&H; \mathcal{P} \models (P, S, M^p\{B\}, D); h : \tau \\
&\iff \text{there are some } \mathcal{L},\ \Delta,\ \Pi \text{ such that } \models h : H,\ \mathcal{P} \vdash M^p : \mathcal{L},\\
&\quad \Pi \subseteq P,\ \ H \models S : \Delta,\ \ \mathcal{P}; \mathcal{L} \vdash \Pi; \Delta; p \rhd B : \tau, \text{ and}\\
&\quad H; \mathcal{P} \models D : \tau.
\end{aligned}
$$

This states that a machine state $(P, S, M^p\{B\}, D); h$ is well typed with τ if each component is well typed, the return type of a method M is τ, and the privilege set Π statically deduced for B is contained in the dynamic privilege set P.

We also define the relation $\vdash \Omega : \Theta$ indicating that a native method environment Ω *satisfies* a class specification Θ as follows:

$$
\begin{aligned}
\vdash \Omega : \Theta \iff &\text{For each } (c, m) \in Dom(\Omega), \text{ the following conditions hold.}\\
&\text{Let } f = \Omega(c, m) \text{ and } \Delta \to \tau = \Theta(c)(m). \text{ For any } S, h, \text{ if}\\
&\models h : H \text{ and } H \models S : \Delta \text{ for some } H \text{ then } f(S, h) = (h', v),\\
&\models h' : H', \text{ and } H' \models v : \tau \text{ for some extension } H' \text{ of } H.
\end{aligned}
$$

To state the type soundness of JVM_{sec}, we define top-level evaluation relation $\mathcal{A}; \Theta; \mathcal{N}; \Omega; \Sigma \vdash (P, S, M^p\{B\}, D); h \Downarrow v$ as $\mathcal{A}; \Theta; \mathcal{N}; \Omega; \Sigma \vdash (P, S, M^p\{B\}, D); h \xrightarrow{*} (\emptyset, v, \emptyset, \emptyset); h'$ where $\xrightarrow{*}$ is the reflexive transitive closure of $\rightarrow$. We also need to

assume some convention to start a JVM_{sec} program. In the JVM, the system starts a program by invoking a special static method `main` in some class. Since JVM_{sec} does not have a static method, following [Freund and Mitchell 1999], we assume that a single object of some class is initially in a heap and a program begins by invoking a method (that takes no argument) on the object. The soundness theorem for JVM_{sec} is then formally stated as follows.

THEOREM TYPE SOUNDNESS. *Let Σ be a given program and $\mathcal{A}$, Θ, $\mathcal{N}$ be given type specifications for Σ such that there exists some $\mathcal{P}$ satisfying $\mathcal{A};\Theta;\mathcal{N} \vdash \Sigma : \mathcal{P}$. Suppose $mbody(\Sigma, c, m) = M^p$ and $\Theta(c)(m) = \emptyset \to \tau$. Let Ω and P be a native method environment and a dynamic privilege set, respectively. If $\vdash \Omega : \Theta$, $\mathcal{P}(c, m) \subseteq P$, and*

$$\mathcal{A};\Theta;\mathcal{N};\Omega;\Sigma \vdash (P, r{\cdot}\emptyset, M^p\{M(entry)\}, \emptyset); \emptyset\{r \leftarrow \langle\rangle_c\} \Downarrow v$$

then v is neither `secfail` *nor* `wrong`.

This is a direct consequence of the following two properties. First, a well typed machine state transforms into another well typed machine state. Second, machine state typing is preserved during the execution of a method. The following two theorems show these two properties.

THEOREM 2. *Let Σ be a given program and $\mathcal{A}$, Θ, $\mathcal{N}$ be given type specifications for Σ such that there is some $\mathcal{P}$ satisfying $\mathcal{A};\Theta;\mathcal{N} \vdash \Sigma : \mathcal{P}$. Let M^p be any method in Σ. Let Ω be a native method environment satisfying $\vdash \Omega : \Theta$. Let $(P, S, M^p\{B\}, D); h$ be any machine state. If $H;\mathcal{P} \models (P, S, M^p\{B\}, D); h : \tau$ for some heap type H then either*

—*$B =$ `return` and $D = \emptyset$, or*

—*there exist some P', S', M', p', B', D', h', τ' such that*

$$\mathcal{A};\Theta;\mathcal{N};\Omega;\Sigma \vdash (P, S, M^p\{B\}, D); h \longrightarrow (P', S', M'^{p'}\{B'\}, D'); h'$$

and $H';\mathcal{P} \vdash (P', S', M'^{p'}\{B'\}, D'); h' : \tau'$ for some extension H' of H.

PROOF. The proof uses the following simple lemmas which are easily shown.

LEMMA 1. *Suppose H' is an extension of H. Then the following three properties hold. (1) If $H \models v : \tau$ then $H' \models v : \tau$. (2) If $H \models S : \Delta$ then $H' \models S : \Delta$. (3) If $H;\mathcal{P} \models D : \tau$ then $H';\mathcal{P} \models D : \tau$.*

LEMMA 2. *If $mclass(\Theta, c_1, m) = c_2$, and either $\mathcal{P}(c_2, m) = \Pi$ or $\mathcal{N}(c_2, m) = \Pi$ then for any c such that $c_1 <: c$, $\Pi \subseteq AllPrivs(\mathcal{P}, c, m)$.*

The proof of the theorem proceeds by case analysis in terms of the first instruction of block B.

Suppose $H;\mathcal{P} \vdash (P, S, M^p\{B\}, D); h : \tau$. Then there are some $\mathcal{L}, \Delta, \Pi$ such that $\models h : H$, $\mathcal{P} \vdash M^p : \mathcal{L}$, $H \models S : \Delta$, $\Pi \subseteq P$, $\mathcal{P};\mathcal{L} \vdash \Pi;\Delta; p \rhd B : \tau$, and $H;\mathcal{P} \models D : \tau$.

Case $B =$ `return`. By the definition of the typing rule for `return`, there are some Δ_0, v, S_0 such that $\Delta = \tau{\cdot}\Delta_0$. Then $S = v{\cdot}S_0$. The case for $D = \emptyset$ is trivial.

Suppose there are some $P_1, S_1, M_1, p_1, B_1, D_1$ such that $D = (P_1, S_1, M_1^{p_1}\{B_1\})\cdot D_1$. By the definition of the transition rule,

$$(P, v\cdot S_0, M^p\{\texttt{return}\}, (P_1, S_1, M_1^{p_1}\{B_1\})\cdot D_1); h \longrightarrow (P_1, v\cdot S_1, M_1^{p_1}\{B_1\}, D_1); h$$

Since $H; \mathcal{P} \models D : \tau$, there are some $\mathcal{L}_1, \Delta_1, \Pi_1, \tau_1$ such that $\mathcal{P} \vdash M_1^{p_1} : \mathcal{L}_1$, $H \models S_1 : \Delta_1$, $\Pi_1 \subseteq P_1$, $\mathcal{P}; \mathcal{L} \vdash \Pi_1; \tau\cdot\Delta_1; p_1 \rhd B_1 : \tau_1$, and $H; \mathcal{P} \models D_1 : \tau_1$. Since $H \models v : \tau$, $H; \mathcal{P} \models (P_1, v\cdot S_1, M_1^{p_1}\{B_1\}, D_1); h : \tau_1$.

Case $B = \texttt{goto}(l)$. By the definition of the transition rule,

$$(P, S, M^p\{\texttt{goto}(l)\}, D); h \longrightarrow (P, S, M^p\{M(l)\}, D); h$$

By the definition of the typing rule for goto, $\mathcal{L}(l) = \Pi'; \Delta; p \rhd \tau$ and $\Pi' \subseteq \Pi$ for some Π'. Since $\mathcal{P} \vdash M^p : \mathcal{L}$, $\mathcal{P}; \mathcal{L} \vdash \Pi'; \Delta; p \rhd M(l) : \tau$. Also, $\Pi' \subseteq P$ follows from $\Pi \subseteq P$. Therefore we have $H; \mathcal{P} \models (P, S, M^p\{M(l)\}, D); h : \tau$.

Case $B = \texttt{ifeq}(l)\cdot B_1$. There are some Δ', n, S' such that $\Delta = \texttt{int}\cdot\Delta'$ and $S = n\cdot S'$. We only show the case for $n = 0$. The other case is simpler. By the definition of the transition rule,

$$(P, 0\cdot S', M^p\{\texttt{ifeq}(l)\cdot B_1\}, D); h \longrightarrow (P, S', M^p\{M(l)\}, D); h$$

By the definition of the typing rule for ifeq, $\mathcal{L}(l) = \Pi'; \Delta'; p \rhd \tau$ and $\Pi' \subseteq \Pi$ for some Π'. Thus, $\mathcal{P}; \mathcal{L} \vdash \Pi'; \Delta'; p \rhd M(l) : \tau$ and $\Pi' \subseteq P$. Therefore $H; \mathcal{P} \models (P, S', M^p\{M(l)\}, D); h : \tau$.

Case $B = \texttt{new}(c)\cdot B_1$. By the definition of the transition rule,

$$(P, S, M^p\{\texttt{new}(c)\cdot B_1\}, D); h \longrightarrow (P, r\cdot S, M^p\{B_1\}, D); h\{r \leftarrow \langle\rangle_c\}$$

and $r \notin Dom(h)$. By the definition of the typing rule for new, $\mathcal{P}; \mathcal{L} \vdash \Pi; c\cdot\Delta; p \rhd B_1 : \tau$. Let $H' = H\{r \leftarrow c\}$. Then $H' \models r : c$ and $\models h\{r \leftarrow \langle\rangle_c\} : H'$. By Lemma 1, we have $H' \models S : \Delta$ and $H' \models D : \tau$. Thus, $H'; \mathcal{P} \models (P, r\cdot S, M^p\{B_1\}, D); h\{r \leftarrow \langle\rangle_c\} : \tau$.

Case $B = \texttt{invoke}(c, m)\cdot B_1$. By the definition of the typing rule for the invoke instruction, Δ must be of the form $\Delta_1\cdot c_0\cdot\Delta_0$ such that $\mathcal{P}; \mathcal{L} \vdash \Pi; \tau'\cdot\Delta_0; p \rhd B_1 : \tau$, $mtype(\Theta, c, m) = \Delta' \to \tau'$, $\Delta_1 <: \Delta'$, $c_0 <: c$, and $AllPrivs(\mathcal{P}, c, m) \subseteq \Pi$. Since $H \models S : \Delta$, S must be of the form S_1, r_0, S_0 such that $H \models S_1 : \Delta_1$, $H \models r_0 : c_0$, and $H \models S_0 : \Delta_0$. Suppose $h(r_0) = \langle\rangle_{c_1}$ and $mclass(\Theta, c_1, m) = c_2$. We first show the case for $(c_2, m) \notin Dom(\mathcal{N})$. By the definition of the transition rule,

$$(P, S_1\cdot r_0\cdot S_0, M^p\{\texttt{invoke}(c, m)\cdot B_1\}, D); h \longrightarrow (P', S_1\cdot r_0\cdot\emptyset, M'^{p'}\{M'(entry)\}, D'); h$$

where $P' = P \cap \mathcal{A}(p')$, $D' = (P, S_0, M^p\{B_1\})\cdot D$ and $M'^{p'} = mbody(\Sigma, c_2, m)$. From the typing of B_1 and the definition of dump typing, $H; \mathcal{P} \models (P, S_0, M^p\{B_1\})\cdot D : \tau'$. Since $H \models r_0 : c_0$, $H(r_0) <: c_0$. Moreover, $h(r_0) = \langle\rangle_{c_1}$ and $\models h : H$ implies $c_1 <: H(r_0)$. Thus, $c_1 <: c$ follows from $c_0 <: c$. Since Σ is well-typed with $\mathcal{P}$, we have $\mathcal{P} \vdash mbody(\Sigma, c_2, m) : \Delta' \xrightarrow{\Pi'} \tau'$ such that $\mathcal{P}(c_2, m) = \Pi'$. By Lemma 2, $\Pi' \subseteq AllPrivs(\mathcal{P}, c, m)$. Thus $\Pi' \subseteq P$ follows from $AllPrivs(\mathcal{P}, c, m) \subseteq \Pi$ and $\Pi \subseteq P$. By the definition of method typing, $\mathcal{P}; \mathcal{L}' \vdash \Pi'; \Delta'\cdot c_2\cdot\emptyset; p' \rhd M'(entry) : \tau'$ for some $\mathcal{L}'$ and $\Pi' \subseteq \mathcal{A}(p')$. Then we have $\Pi' \subseteq P \cap \mathcal{A}(p')$. Since $\Delta_1 <: \Delta'$, $H \models S_1 : \Delta'$.

Thus $H; \mathcal{P} \models (P', S_1 \cdot r_0 \cdot \emptyset, M'^{p'}\{M'(entry)\}, (P, S, M^p\{B_1\}) \cdot D); h : \tau'$. Next, we show the case for $(c_2, m) \in Dom(\mathcal{N})$. Suppose $\mathcal{N}(c_2, m) = \Pi'$. Similarly to the case above, we have $c_1 <: c$. By Lemma 2, $\Pi' \subseteq AllPrivs(\mathcal{P}, c, m)$. Thus we have $\Pi' \subseteq P$. Then we have the the following transition:

$$(P, S_1 \cdot r \cdot S_0, M^p\{\texttt{invoke}(c, m) \cdot B_1\}, D); h \longrightarrow (P, v \cdot S_0, M^p\{B_1\}, D); h'$$

where $\Omega(c, m) = f$ and $f(S_1, h) = (v, h')$. Since $\vdash \Omega : \Theta$, there exists some H' such that H' is an extension of H, $\models h' : H'$, and $H' \models v : \tau'$. Therefore $H'; \mathcal{P} \models (P, v \cdot S, M^p\{B_1\}, D); h' : \tau$ follows from Lemma 1.

Case $B = \texttt{priv}(\pi) \cdot B_1$. We only show the case for $\pi \in \mathcal{A}(p)$. The case for $\pi \notin \mathcal{A}(p)$ is similar. By the transition rule,

$$(P, S, M^p\{\texttt{priv}(\pi) \cdot B_1\}, D); h \longrightarrow (\{\pi\} \cup P, S, M^p\{B_1\}, D); h$$

By the definition of the type system, $\mathcal{P}; \mathcal{L} \vdash \{\pi\} \cup \Pi; \Delta; p \rhd B_1 : \tau$. Since $\{\pi\} \cup \Pi \subseteq \{\pi\} \cup P$, $H; \mathcal{P} \models (\{\pi\} \cup P, S, M^p\{B_1\}, D); h : \tau$.

Cases for `acc`, `iconst`, and `dup` are similarly shown. □

The next theorem states that a machine state typing is preserved during the execution of a method.

THEOREM 3. *Let Σ be a given program and $\mathcal{A}$, Θ, $\mathcal{N}$ be given type specifications for Σ such that there exists some $\mathcal{P}$ satisfying $\mathcal{A}; \Theta; \mathcal{N} \vdash \Sigma : \mathcal{P}$. Suppose B and B' are code blocks which belong to a method M^p in Σ. Let Ω be a given native method environment satisfying $\vdash \Omega : \Theta$.*

If $H; \mathcal{P} \models (P, S, M^p\{B\}, D); h : \tau$, $B \neq \texttt{return}$ and

$$\mathcal{A}; \Theta; \mathcal{N}; \Omega; \Sigma \vdash (P, S, M^p\{B\}, D); h \xrightarrow{*} (P', S', M^p\{B'\}, D); h'$$

then $H'; \mathcal{P} \models (P', S', M^p\{B'\}, D); h' : \tau$ for some extension H' of H.

PROOF. This is proved by induction on the length of the reduction sequence. The base case is trivial. For induction steps, the proof proceeds by case analysis in terms of the first instruction of block B. We only show the cases for `goto` and `invoke`. The case for `ifeq` is similar to `goto`. The other cases follow from the induction hypothesis and the typing rule.

Case $B = \texttt{goto}(l)$. We have the following reduction sequence.

$$\begin{aligned} (P, S, M\{\texttt{goto}(l)\}, D); h &\longrightarrow (P, S, M\{M(l)\}, D), h && (1) \\ &\xrightarrow{*} (P', S', M\{B'\}, D); h' && (2) \end{aligned}$$

By applying Theorem 2 to reduction step (1), there exists some extension H_1 of H such that $H_1; \mathcal{P} \models (P, (S.n) \cdot S, M\{B_1\}, D); h : \tau_1$ for some τ_1. Since B and $M(l)$ belongs to the same method, $\tau_1 = \tau$. By induction hypothesis applied to reduction sequence (2), there exists some extension H' of H_1 such that $H'; \mathcal{P} \models (P', S', M\{B'\}, D); h' : \tau_1$, as desired.

Case $B = \texttt{invoke(c,m)} \cdot B_1$. There exist some S_0, r_0, S_1 such that $S = S_0 \cdot r_0 \cdot S_1$. The case for a native method is trivial. We consider the case for a non-native method. Since the only instruction that decreases the dump D is `return` and a

method can only terminate with `return`, the reduction sequence must have been as follows for some P_1, S_2, M_0, p_0, h_1, v.

$$
\begin{aligned}
&(P, S_0{\cdot}r_0{\cdot}S_1, M^p\{\texttt{invoke}(c, m){\cdot}B_1\}, D); h \\
&\qquad \longrightarrow (P_0, S_0{\cdot}r_0{\cdot}\emptyset, M_0^{p_0}\{M_0(\mathit{entry})\}, (P, S_1, M^p\{B_1\}){\cdot}D); h && (1)\\
&\qquad \stackrel{*}{\longrightarrow} (P_1, v{\cdot}S_2, M_0^{p_0}\{\texttt{return}\}, (P, S_1, M^p\{B_1\}){\cdot}D); h_1 && (2)\\
&\qquad \longrightarrow (P_1, v{\cdot}S_1, M^p\{B_1\}, D); h_1 && (3)\\
&\qquad \stackrel{*}{\longrightarrow} (P', S', M^p\{B'\}, D); h' && (4)
\end{aligned}
$$

By Theorem 2 applied to reduction step (1), there exists some extension H_1 of H such that $H_1; \mathcal{P} \models (P_0, S_0{\cdot}r_0{\cdot}\emptyset, M_0^{p_0}\{M_0(\mathit{entry})\}, (P, S_1, M^p\{B_1\}){\cdot}D); h : \tau_1$ for some τ_1. By induction hypothesis for reduction sequence (2), there exists some H_1' such that $H_1'; \mathcal{P} \models (P_1, v{\cdot}S_2, M_0^{p_0}\{\texttt{return}\}, (P, S_1, M^p\{B_1\}){\cdot}D); h_1 : \tau_1$, and H_1' is an extension of H_1. By applying Theorem 2 to reduction step (3), there exists some extension H_2 of H_1' such that $H_2; \mathcal{P} \models (P_1, v{\cdot}S, M^p\{B_1\}, D); h_1 : \tau_1'$ for some τ_1'. Since B and B_1 belong to the same method, $\tau = \tau_1'$. By induction hypothesis applied to reduction sequence (4), there exists some extension H' of H_2 such that $H'; \mathcal{P} \models (P', S', M^p\{B'\}, D); h' : \tau_1'$. □

As a result of the above two theorems, we have the following.

COROLLARY 1. *Suppose a program Σ and type specifications $\mathcal{A}$, Θ, $\mathcal{N}$, satisfy $\mathcal{A}; \Theta; \mathcal{N} \vdash \Sigma : \mathcal{P}$ for some $\mathcal{P}$. Let Ω be a native method environment satisfying $\vdash \Omega : \Theta$. If $H; \mathcal{P} \vdash (P, S, M^p\{B\}, \emptyset); h : \tau$ and $\mathcal{A}; \Theta; \mathcal{N}; \Omega; \Sigma \vdash (P, S, M^p\{B\}, \emptyset); h \Downarrow v$ then $H' \models v : \tau$ for some extension H' of H.*

PROOF. Since neither `wrong` nor `secfail` has a type, by Theorem 2, there exist some P_1, S_1, h_1 such that

$$(P, S, M^p\{B\}, \emptyset); h \stackrel{*}{\longrightarrow} (P_1, S_1, M^p\{\texttt{return}\}, \emptyset); h_1.$$

By theorem 3, $H_1; \mathcal{P} \models (P_1, S_1, M^p\{\texttt{return}\}, \emptyset); h_1 : \tau$ for some extension H_1 of H such that $\models h_1 : H_1$. Then there exist some $\mathcal{L}, \Pi_1, \Delta_1$ such that $\mathcal{P} \vdash M^p : \mathcal{L}$, $\mathcal{P}; \mathcal{L} \vdash \Pi_1; \Delta_1; p \ \triangleright \texttt{return} : \tau$, and $H_1 \models S_1 : \Delta_1$. By the typing rule for the `return` instruction, there exists some Δ' such that $\Delta = \tau{\cdot}\Delta'$. Therefore $S_1 = v{\cdot}S_1'$ and $H_1 \models v : \tau$. By the transition rule, we have $(P_1, v{\cdot}S_1', M^p\{\texttt{return}\}, \emptyset); h_1 \longrightarrow (\emptyset, v, \emptyset, \emptyset); h_1$. □

We now complete the type soundness theorem.

PROOF OF TYPE SOUNDNESS. By the definition of the method typing, $\mathcal{P}; \mathcal{L} \vdash \Pi, c{\cdot}\emptyset, p \ \triangleright M(\mathit{entry}) : \tau$ for some $\mathcal{L}$. Suppose H is a heap type such that $\models h\{r \leftarrow c\} : H$. Then $H \models r : c$. Thus, $H; \mathcal{P} \models (P, r{\cdot}\emptyset, M^p\{M(\mathit{entry})\}, \emptyset); h\{r \leftarrow \langle\rangle_c\} : \tau$. By Corollary 1, we have $H' \models v : \tau$ for some extension H' of H. The proof concludes with the fact that neither `secfail` nor `wrong` has a type. □

6. TYPE INFERENCE

In this section, we develop a type inference algorithm that infers a privilege environment for a given program.

The function of the type inference algorithm is twofold: it checks the type consistency of each method, and it also checks the conformance of privilege restrictions imposed by a given access policy. The former is achieved by inferring a type of a method using a unification algorithm and checking it against a given type specification. The latter is achieved by inferring a minimal set of privileges for each method by generating a set of inclusion constraints on privilege sets and solving the constraint set. This section first develops algorithms for unification and constraint solving and then defines the type inference algorithm.

6.1 The Unification Algorithm

We firstly extend types and stack types by introducing *type variables* (ranged over by t) and *stack type variables* (ranged over by δ), respectively as follows.

$$\tau ::= t \mid \mathtt{int} \mid c$$
$$\Delta ::= \delta \mid \emptyset \mid \tau{\cdot}\Delta$$

Type variables are bounded by a bound environment $\mathcal{K}$, which is a mapping from a finite set of type variables to class names or $*$. $\mathcal{K}(t) = c$ indicates that t only ranges over subclasses of a class c, and $\mathcal{K}(t) = *$ indicates that t has no bound. We write $\mathcal{K} \vdash \tau <: \tau'$ if τ is a subclass of τ' under the bound environment $\mathcal{K}$. This relation is given below.

$$\mathcal{K} \vdash t <: c \quad \text{if } \mathcal{K}(t) <: c$$
$$\mathcal{K} \vdash \tau_1 <: \tau_2 \quad \text{if } \tau_1 <: \tau_2$$

This is also extended to stack types as $\mathcal{K} \vdash \Delta <: \Delta'$ if $\mathcal{K} \vdash \Delta.i <: \Delta'.i$ for each i.

A *substitution* (ranged over by $\mathcal{S}$) is a function from type variables to types and stack type variables to stack types. We write $[v_1/x_1, \cdots, v_n/x_n]$ for the substitution that maps each x_i to v_i. A substitution $\mathcal{S}$ is extended to the set of all type variables by setting $\mathcal{S}(x) = x$ for all $x \notin \mathit{Dom}(\mathcal{S})$. In what follows, we further identify $\mathcal{S}$ with its homomorphic extension to any syntactic structure containing type variables and stack type variables. Specifically, we write $\mathcal{S}(\Pi; \Delta; p \rhd \tau)$ for $\Pi; \mathcal{S}(\Delta); p \rhd \mathcal{S}(\tau)$ and write $\mathcal{S}(\mathcal{L})$ for $\{l : \mathcal{S}(\Pi; \Delta; p \rhd \tau) \mid l \in \mathit{Dom}(\mathcal{L}) \text{ and } \mathcal{L}(l) = \Pi; \Delta; p \rhd \tau\}$. If $\mathcal{S}_1$ and $\mathcal{S}_2$ are substitutions, $\mathcal{S}_1 \circ \mathcal{S}_2$ represents their composition, i.e., $(\mathcal{S}_1 \circ \mathcal{S}_2)(x) = \mathcal{S}_1(\mathcal{S}_2(x))$.

A *kinded substitution* is a pair $(\mathcal{S}, \mathcal{K})$ consisting of a substitution $\mathcal{S}$ and a bound environment $\mathcal{K}$ such that for all $t \in \mathit{Dom}(\mathcal{S})$, if $\mathcal{S}(t) = t'$ then $t' \in \mathit{Dom}(\mathcal{K})$. We say that a kinded substitution $(\mathcal{S}, \mathcal{K})$ *respects* $\mathcal{K}'$ if for all $t \in \mathit{Dom}(\mathcal{K}')$, $\mathcal{K} \vdash \mathcal{S}(t) <: \mathcal{K}'(t)$ or $\mathcal{K}'(t) = *$. We also say that a substitution $\mathcal{S}$ respects $\mathcal{K}$ if $(\mathcal{S}, \mathcal{K})$ respects $\mathcal{K}$. The following lemmas can be easily verified.

LEMMA 3. *If* $\mathcal{K} \vdash \tau_1 <: \tau_2$ *and* $(\mathcal{S}, \mathcal{K}')$ *respects* $\mathcal{K}$ *then* $\mathcal{K}' \vdash \mathcal{S}(\tau_1) <: \mathcal{S}(\tau_2)$

LEMMA 4. *If* $(\mathcal{S}_1, \mathcal{K}_1)$ *respects* $\mathcal{K}$ *and* $(\mathcal{S}_2, \mathcal{K}_2)$ *respects* $\mathcal{K}_1$ *then* $(\mathcal{S}_2 \circ \mathcal{S}_1, \mathcal{K}_2)$ *respects* $\mathcal{K}$.

As a consequence of Lemma 3, if $\mathcal{K} \vdash \Delta_1 <: \Delta_2$ and $(\mathcal{S}, \mathcal{K}')$ respects $\mathcal{K}$ then $\mathcal{K}' \vdash \mathcal{S}(\Delta_1) <: \mathcal{S}(\Delta_2)$.

In inferring a typing for $\mathtt{invoke}(c, m)$, the stack type must be a subclass of an argument type of method m. In order to check this subclass relation, we define the algorithms *SubStack* and *SubType* given in Figure 10. For these algorithms, the following properties hold.

$$
\begin{array}{l}
\mathit{SubStack}(\mathcal{K}, \tau_1{\cdot}\Delta_1, \tau_2{\cdot}\Delta_2) = \text{let } (\mathcal{S}_1, \mathcal{K}_1) = \mathit{SubType}(\mathcal{K}, \tau_1, \tau_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathcal{S}_2, \mathcal{K}_2) = \mathit{SubStack}(\mathcal{K}_1, \mathcal{S}_1(\Delta_1), \mathcal{S}_1(\Delta_2)) \\
\qquad\qquad\qquad\qquad\qquad \text{in } (\mathcal{S}_2 \circ \mathcal{S}_1, \mathcal{K}_2) \\
\mathit{SubStack}(\mathcal{K}, \emptyset, \emptyset) = (\emptyset, \mathcal{K}) \\
\\
\mathit{SubType}(\mathcal{K}, t, c) = \text{if } \mathcal{K}(t) = * \text{ or } c <: \mathcal{K}(t) \text{ then } ([t'/t], \mathcal{K} \cup \{t' = c\}) \quad (t' \text{ fresh}) \\
\qquad\qquad\qquad\qquad \text{else if } \mathcal{K}(t) <: c \text{ then } (\emptyset, \mathcal{K}) \\
\mathit{SubType}(\mathcal{K}, c_1, c_2) = \text{if } c_1 <: c_2 \text{ then } (\emptyset, \mathcal{K}) \\
\mathit{SubType}(\mathcal{K}, t, \mathtt{int}) = \text{if } \mathcal{K}(t) = * \text{ then } ([\mathtt{int}/t], \mathcal{K}) \\
\mathit{SubType}(\mathcal{K}, \mathtt{int}, \mathtt{int}) = (\emptyset, \mathcal{K}) \\
\mathit{SubType}(\mathcal{K}, \tau, \tau') = \mathit{Failure}
\end{array}
$$

Fig. 10. The algorithm *SubStack* and *SubType*

LEMMA 5. *(1) If* $\mathit{SubType}(\mathcal{K}, \tau, \tau') = (\mathcal{S}, \mathcal{K}')$ *then* $(\mathcal{S}, \mathcal{K}')$ *respects* $\mathcal{K}$ *and* $\mathcal{K}' \vdash \mathcal{S}(\tau) <: \mathcal{S}(\tau')$. *(2) If* $\mathit{SubStack}(\mathcal{K}, \Delta, \Delta') = (\mathcal{S}, \mathcal{K}')$ *then* $(\mathcal{S}, \mathcal{K}')$ *respects* $\mathcal{K}$ *and* $\mathcal{K}' \vdash \mathcal{S}(\Delta) <: \mathcal{S}(\Delta')$.

PROOF. The proof of (1) is by case analysis of type τ. Property (2) is shown by induction on the numbers of recursive calls of *SubStack*, using Lemma 3, Lemma 4, and Property (1). □

Unification is refined to incorporate bound environments. In the style of Gallier and Snyder [1989], a unification algorithm *Unify* for types is defined by transformation rules on 3-tuples of the form $(E, \mathcal{S}, \mathcal{K})$ consisting of a set E of type equations, a substitution $\mathcal{S}$, and a bound environment $\mathcal{K}$. The set of rules is given as follows.

$$
\begin{array}{rcll}
(E \cup \{(\tau, \tau)\}, S, \mathcal{K}) & \Longrightarrow & (E, S, \mathcal{K}) & \\
(E \cup \{(t, \tau)\}, S, \mathcal{K}) & \Longrightarrow & ([\tau/t]E, \{(t, \tau)\} \cup [\tau/t]S, \mathcal{K}) & (\text{if } \mathcal{K}(t) = * \text{ or } \tau <: \mathcal{K}(t)) \\
(E \cup \{(\tau, t)\}, S, \mathcal{K}) & \Longrightarrow & ([\tau/t]E, \{(t, \tau)\} \cup [\tau/t]S, \mathcal{K}) & (\text{if } \mathcal{K}(t) = * \text{ or } \tau <: \mathcal{K}(t)) \\
(E \cup \{(t_1, t_2)\}, S, \mathcal{K}) & \Longrightarrow & ([t_2/t_1]E, \{(t_1, t_2)\} \cup [t_2/t_1]S, \mathcal{K}) & (\text{if } \mathcal{K}(t_2) <: \mathcal{K}(t_1)) \\
(E \cup \{(t_1, t_2)\}, S, \mathcal{K}) & \Longrightarrow & ([t_1/t_2]E, \{(t_2, t_1)\} \cup [t_1/t_2]S, \mathcal{K}) & (\text{if } \mathcal{K}(t_1) <: \mathcal{K}(t_2))
\end{array}
$$

Using these transition rules, *Unify* is defined as

$$
\mathit{Unify}(\mathcal{K}, E) = \begin{cases} \mathcal{S} & \text{if } (E, \emptyset, \mathcal{K}) \stackrel{*}{\Longrightarrow} (\emptyset, \mathcal{S}, \mathcal{K}) \\ \mathit{Failure} & \text{otherwise} \end{cases}
$$

where $\stackrel{*}{\Longrightarrow}$ is the reflexive transitive closure of the relation $\Longrightarrow$.

In the JVM, the subclass relation $<:$ has the property that if two classes are incomparable then they have no common subclass. On the basis of this property, we can prove the following lemma, which establishes the correctness of *Unify*.

LEMMA 6. *Let* E *be a set of type equations and* $\mathcal{K}$ *be a bound environment for the set of type variables in* E. *If* $\mathit{Unify}(E, \mathcal{K}) = \mathcal{S}$ *then* S *respects* $\mathcal{K}$ *and* S *is a unifier for* E.

PROOF. It can be easily verified by simple inspection of each transformation that if $(E_1, \mathcal{S}_1, \mathcal{K}) \Longrightarrow (E_2, \mathcal{S}_2, \mathcal{K})$ then the following two properties hold:

(1) if $\mathcal{S}_1$ respects $\mathcal{K}$ then $\mathcal{S}_2$ respects $\mathcal{K}$, and

(2) $(E_1 \cup \mathcal{S}_1)$ and $(E_2 \cup \mathcal{S}_2)$ have the same set of unifiers.

Suppose $\mathit{Unify}(\mathcal{K}, E)$ computes a substitution $\mathcal{S}$. Then $(E, \emptyset, \mathcal{K}) \stackrel{*}{\Longrightarrow} (\emptyset, \mathcal{S}, \mathcal{K})$. Since $\emptyset$ respects $\mathcal{K}$, by property 2, $\mathcal{S}$ respects $\mathcal{K}$. It can be easily verified that $\mathcal{S}$ is a most general unifier of itself. Therefore, by property 1, $\mathcal{S}$ is a most general unifier of E. □

Using *Unify*, we also define a unification algorithm *UnifyStack* for stack types. It accepts a bound environment $\mathcal{K}$ and two stack types Δ_1, Δ_2 and computes a substitution $\mathcal{S}$ such that $\mathcal{S}(\Delta_1) = \mathcal{S}(\Delta_2)$ and $\mathcal{S}$ respects $\mathcal{K}$. *UnifyStack* is given as follows.

$$
\begin{array}{l}
\mathit{UnifyStack}(\mathcal{K}, \tau{\cdot}\Delta_1, \tau{\cdot}\Delta_2) = \mathit{UnifyStack}(\mathcal{K}, \Delta_1, \Delta_2) \\
\mathit{UnifyStack}(\mathcal{K}, \tau_1{\cdot}\Delta_1, \tau_2{\cdot}\Delta_2) = \text{let } S_1 = \mathit{Unify}(\mathcal{K}, \{(\tau_1, \tau_2)\}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad S_2 = \mathit{UnifyStack}(\mathcal{K}, S_1(\Delta_1), S_1(\Delta_2)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{in } S_2 \circ S_1 \\
\mathit{UnifyStack}(\mathcal{K}, \delta, \Delta) = [\Delta/\delta] \\
\mathit{UnifyStack}(\mathcal{K}, \Delta, \delta) = \mathit{UnifyStack}(\mathcal{K}, \delta, \Delta)
\end{array}
$$

With respect to the above, we have the following.

LEMMA 7. *If* $\mathit{UnifyStack}(\mathcal{K}, \Delta_1, \Delta_2) = \mathcal{S}$ *then* $\mathcal{S}$ *respects* $\mathcal{K}$ *and* $\mathcal{S}(\Delta_1) = \mathcal{S}(\Delta_2)$.

PROOF. This can be shown by induction on the number of recursive calls of *UnifyStack*, using Lemma 4 and Lemma 6. □

6.2 The Algorithm for Solving Inclusion Constraints

We extend the language of privilege sets to include *privilege set variables* ρ as

$$\Pi ::= R \mid R{\cdot}\rho$$

where R is a meta variable for a *ground* set of privileges, i.e., those that do not contain a set variable. $R{\cdot}\rho$ intuitively denotes the union of the (known) set R and the (undetermined) set represented by variable ρ. We occasionally write $R \cup \Pi$ for the privilege set expression obtained by adding R to the known part of Π , i.e., if $\Pi = R'$ then $R \cup \Pi = R \cup R'$ and if $\Pi = R'{\cdot}\rho$ then $R \cup \Pi = (R \cup R'){\cdot}\rho$. We also simply write ρ for $\emptyset{\cdot}\rho$. A *privilege set variable substitution* (ranged over by φ) is a function from a finite set of set variables to privilege sets. Its application to privilege sets is defined as follows.

$$
\begin{array}{rcl}
\varphi(R) & ::= & R \\
\varphi(R{\cdot}\rho) & ::= & R \cup \varphi(\rho)
\end{array}
$$

In the following, we identify φ with its homomorphic extension to any syntactic structure containing privilege set variables.

As explained, the type inference algorithm generates a *privilege inclusion constraint set* of the form $\{\Pi_1 \sqsubseteq \Pi'_1, \cdots, \Pi_n \sqsubseteq \Pi'_n\}$ and then solves the constraint set to compute the minimal set of privileges for each method. Each element of the constraint set represents a constraint to be satisfied by each method invocation. We use PC as a meta variable for privilege inclusion constraint sets. We say that a privilege set variable substitution φ *satisfies* PC if $\varphi(\Pi) \subseteq \varphi(\Pi')$ for any $\Pi \sqsubseteq \Pi' \in PC$. Figure 11 gives the algorithm *SolvePC* that accepts PC and

$$
\begin{array}{l}
\mathit{SolvePC}(PC) = \text{if } \mathit{Ground}(\Pi_1) \subseteq \mathit{Ground}(\Pi_2) \text{ for all } \Pi_1 \sqsubseteq \Pi_2 \in PC \text{ then} \\
\qquad\qquad \text{the substitution } \varphi_0 \text{ such that } \varphi_0(\rho) = \emptyset \text{ for each } \rho \text{ in } PC \\
\qquad\quad \text{else let } \{\Pi_1 \sqsubseteq \Pi_2\} \cup PC_0 = PC \text{ such that } \mathit{Ground}(\Pi_1) \not\subseteq \mathit{Ground}(\Pi_2) \\
\qquad\qquad (\varphi_1, \Pi_1' \sqsubseteq \Pi_2') = \mathit{SolveOne}(\Pi_1 \sqsubseteq \Pi_2) \\
\qquad\qquad \varphi_2 = \mathit{SolvePC}(\{\Pi_1' \sqsubseteq \Pi_2'\} \cup \varphi_1(PC_0)) \\
\qquad\quad \text{in } \ \varphi_2 \circ \varphi_1 \\
\\
\mathit{SolveOne}(R_1 \sqsubseteq R_2) = \mathit{Failure} \\
\mathit{SolveOne}(R_1 \sqsubseteq R_2{\cdot}\rho) = \text{let } R_3 = R_1 \setminus R_2 \\
\qquad\qquad\qquad\qquad \varphi = [R_3{\cdot}\rho'/\rho] \quad (\rho' \text{ fresh}) \\
\qquad\qquad\qquad \text{in } \ (\varphi, R_1 \sqsubseteq (R_2 \cup R_3){\cdot}\rho_3) \\
\mathit{SolveOne}(R_1{\cdot}\rho \sqsubseteq R_2) = \mathit{Failure} \\
\mathit{SolveOne}(R_1{\cdot}\rho_1 \sqsubseteq R_2{\cdot}\rho_2) = \text{let } R_3 = R_1 \setminus R_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad \varphi = [R_3{\cdot}\rho_3/\rho_2] \quad (\rho_3 \text{ fresh}) \\
\qquad\qquad\qquad\qquad\qquad \text{in } \ (\varphi, R_1{\cdot}\rho_1 \sqsubseteq (R_2 \cup R_3){\cdot}\rho_3)
\end{array}
$$

Fig. 11. Algorithm *SolvePC*

computes φ such that φ *satisfies* PC. An auxiliary function *Ground* used in this definition is defined as $\mathit{Ground}(R) = R$ and $\mathit{Ground}(R{\cdot}\rho) = R$.

The following lemma shows that *SolvePC* computes the minimal solution of a given privilege inclusion constraint set.

LEMMA 8. *(1) SolvePC terminates on all inputs.*

(2) If $\mathit{SolvePC}(PC) = \varphi$ *then* φ *satisfies* PC.

(3) If φ *satisfies* PC *then* $\mathit{SolvePC}(PC) = \varphi'$ *such that for each* ρ *occurring in* PC, $\varphi'(\rho) \subseteq \varphi(\rho)$.

PROOF. The first property follows from the facts that *SolvePC* monotonically increases the size of constraints and that there are only finitely many privilege atoms. The second and the third properties can then be shown by induction on the number of recursive calls of *SolvePC*. □

We note that R appearing in a privilege inclusion constraint set is a ground set that does not contain a set variable, and expressions $R_1 \setminus R_2$ and $R_1 \cup R_2$ used in this algorithm are ordinary set-theoretic operations. In contrast to the constrained type system developed in [Skalka and Smith 2000], our language of privilege sets (Π) only contains a limited form of union $R{\cdot}\rho$ and does not contain expressions to denote set difference due to the simpler nature of JVM_{sec} compared to the lambda calculus.

6.3 The Type Inference Algorithm

Using *Unify*, *UnifyStack*, and *SolvePC*, a type inference algorithm is developed. We assume that a class specification Θ, a native method specification $\mathcal{N}$, and an access policy $\mathcal{A}$ are globally given. Figure 12 gives the main algorithm $\mathcal{W}$ that takes a program Σ and infers a privilege environment $\mathcal{P}$ of the program. $\mathcal{W}$ first generates a skeleton of a privilege environment of a program using an auxiliary function *makeSkeleton*. It then collects a privilege inclusion constraint set by invoking the sub-algorithm $\mathcal{WC}$ for each class. $\mathcal{WC}$ in turn invokes $\mathcal{WM}$ on each method. $\mathcal{WM}$,

$$
\begin{array}{l}
\mathcal{W}(\Sigma) = \text{let } \{c_1 = (p_1, C_1), \cdots, c_n = (p_n, C_n)\} = \Sigma \\
\qquad\qquad \mathcal{P}_i = \mathit{makeSkeleton}(c_i, C_i, p_i) \text{ (for each } i\text{)} \\
\qquad\qquad \mathcal{P} = \bigcup_i \mathcal{P}_i \\
\qquad\qquad PC_0 = \emptyset \\
\qquad\qquad PC_i = PC_{i-1} \cup \mathcal{WC}(\mathcal{P}, c_i, C_i, p_i) \text{ (for each } i\text{)} \\
\qquad\qquad \varphi = \mathit{SolvePC}(PC_n) \\
\qquad\quad \text{in } \varphi(\mathcal{P})
\end{array}
$$

$$
\mathit{makeSkeleton}(c, C, p) = \{(c, m_i) = \rho_{i\ (1 \le i \le n)} \mid m_i \in \mathit{Dom}(C),\ \rho_i \text{ fresh}\}
$$

$$
\begin{array}{l}
\mathcal{WC}(\mathcal{P}, c, C, p) = \text{let } \{m_1 = M_1, \cdots, m_n = M_n\} = C \\
\qquad\qquad\qquad\qquad PC_0 = \emptyset \\
\qquad\qquad\qquad\qquad PC_i = PC_{i-1} \cup \mathcal{WM}(\mathcal{P}, c, m_i, M_i, p) \text{ (for each } i\text{)} \\
\qquad\qquad\qquad \text{in } PC_n
\end{array}
$$

Fig. 12. Type inference algorithm for programs

$$
\begin{array}{l}
\mathcal{WM}(\mathcal{P}, c, m, M, p) = \\
\quad \text{let } \{l_1 = B_1, \cdots, l_n = B_n\} = M \\
\qquad \mathcal{B}_k = \Pi; \Delta; p \rhd \tau \ (l_k = \mathit{entry},\ \Theta(c)(m) = \Delta \to \tau,\ \mathcal{P}(c, m) = \Pi) \\
\qquad \mathcal{B}_i = \rho_i; \delta_i; p \rhd \tau \ (\rho_i, \delta_i \text{ fresh}, 1 \le i \le n, l_i \ne \mathit{entry}) \\
\qquad \mathcal{L} = \{l_1 = \mathcal{B}_1, \cdots, l_n = \mathcal{B}_n\} \\
\qquad \mathcal{S}_0 = \emptyset,\ \mathcal{K}_0 = \emptyset \\
\qquad PC_0 = \{\rho_1 \sqsubseteq \mathcal{A}(p), \cdots, \rho_n \sqsubseteq \mathcal{A}(p), \Pi \sqsubseteq \mathcal{A}(p)\} \\
\qquad \text{for each i do} \\
\qquad\quad (\mathcal{S}, \mathcal{K}_i, PC_i) = \mathcal{WB}(\mathcal{P}, \mathcal{S}_{i-1}(\mathcal{L}), \mathcal{S}_{i-1}(\mathcal{B}_i), B_i, \mathcal{K}_{i-1}, PC_{i-1}) \\
\qquad\quad \mathcal{S}_i = \mathcal{S} \circ \mathcal{S}_{i-1} \\
\qquad \text{end} \\
\quad \text{in } PC_n
\end{array}
$$

Fig. 13. Type inference algorithm for methods

given in Figure 13, first sets up a skeleton of a label environment of a method. It then infers a typing for each code block of the method and collects a privilege inclusion constraint set by invoking $\mathcal{WB}$ on each code block. Figure 14 shows $\mathcal{WB}$ which infers a typing of a code block using *Unify* and *UnifyStack* and collects a privilege inclusion constraint set in which each element is a constraint of a method invocation of the block. The auxiliary function $\mathit{AllClasses}(\Theta, c, m)$ used in $\mathcal{WB}$ looks up all the methods that can be invoked by `invoke(`c, m`)` and returns the set of classes that includes them. It is defined as follows.

$$
\begin{array}{l}
\mathit{AllClasses}(\Theta, c, m) = \text{let } \mathit{classes} = \{c' \mid c' <: c,\ m \in \mathit{Dom}(\Theta(c))\} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{in if } m \in \mathit{Dom}(\Theta(c)) \text{ then } \mathit{classes} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \text{else } \mathit{mclass}(\Theta, c, m) \cup \mathit{classes}
\end{array}
$$

After obtaining a privilege inclusion constraint set PC for all classes, the main algorithm $\mathcal{W}$ solves PC using *SolvePC* and obtains a substitution for PC. Finally, it applies the substitution to the skeleton of the privilege environment and returns the obtained privilege environment.

$$\mathcal{WB}(\mathcal{P},\mathcal{L},(\Pi;\Delta;p\rhd\tau),\texttt{return},\mathcal{K},PC) = \text{let}\ \ \mathcal{S}_1 = \mathit{UnifyStack}(\mathcal{K},\Delta,\tau{\cdot}\delta)\ \ (\delta\text{ fresh})\ \ \text{in}\ \ (\mathcal{S}_1,\mathcal{K},PC)$$

$$\begin{array}{l}\mathcal{WB}(\mathcal{P},\mathcal{L},(\Pi;\Delta;p\rhd\tau),\texttt{goto}(l),\mathcal{K},PC) = \text{let}\ \ \Pi';\Delta';p\rhd\tau' = \mathcal{L}(l)\\ \qquad\qquad \mathcal{S}_1 = \mathit{Unify}(\mathcal{K},\{(\tau,\tau')\})\\ \qquad\qquad \mathcal{S}_2 = \mathit{UnifyStack}(\mathcal{K},\mathcal{S}_1(\Delta),\mathcal{S}_1(\Delta'))\\ \qquad\qquad PC_1 = PC\cup\{\Pi'\sqsubseteq\Pi\}\\ \qquad\qquad \text{in}\ \ (\mathcal{S}_2\circ\mathcal{S}_1,\mathcal{K},PC_1)\end{array}$$

$$\begin{array}{l}\mathcal{WB}(\mathcal{P},\mathcal{L},(\Pi;\Delta;p\rhd\tau),\texttt{acc}(n){\cdot}B,\mathcal{K},PC) =\\ \quad\text{let}\ \ \mathcal{K}_1 = \mathcal{K}\cup\{t_1=*,\cdots,t_n=*\}\ \ (t_i\text{ fresh})\\ \qquad \mathcal{S}_1 = \mathit{UnifyStack}(\mathcal{K}_1,\Delta,t_1\cdot\ldots\cdot t_n{\cdot}\delta)\ \ (\delta\text{ fresh})\\ \qquad (\mathcal{S}_2,\mathcal{K}_2,PC') = \mathcal{WB}(\mathcal{P},\mathcal{S}_1(\mathcal{L}),\mathcal{S}_1(\Pi;t_n{\cdot}\Delta;p\rhd\tau),B,\mathcal{K}_1,PC)\\ \quad\text{in}\ \ (\mathcal{S}_2\circ\mathcal{S}_1,\mathcal{K}_2,PC')\end{array}$$

$$\mathcal{WB}(\mathcal{P},\mathcal{L},(\Pi;\Delta;p\rhd\tau),\texttt{iconst}(n){\cdot}B,\mathcal{K},PC) = \mathcal{WB}(\mathcal{P},\mathcal{L},(\Pi;\texttt{int}{\cdot}\Delta;p\rhd\tau),B,\mathcal{K},PC)$$

$$\begin{array}{l}\mathcal{WB}(\mathcal{P},\mathcal{L},(\Pi;\Delta;p\rhd\tau),\texttt{dup}{\cdot}B,\mathcal{K},PC) =\\ \quad\text{let}\ \ \mathcal{K}_1 = \mathcal{K}\cup\{t=*\}\ \ (t\text{ fresh})\\ \qquad \mathcal{S}_1 = \mathit{UnifyStack}(\mathcal{K}_1,\Delta,t{\cdot}\delta)\ \ (\delta\text{ fresh})\\ \qquad (\mathcal{S}_2,\mathcal{K}_2,PC_1) = \mathcal{WB}(\mathcal{P},\mathcal{S}_1(\mathcal{L}),\mathcal{S}_1(\Pi;t{\cdot}t{\cdot}\delta;p\rhd\tau),B,\mathcal{K}_1,PC)\\ \quad\text{in}\ \ (\mathcal{S}_2\circ\mathcal{S}_1,\mathcal{K}_2,PC_1)\end{array}$$

$$\begin{array}{l}\mathcal{WB}(\mathcal{P},\mathcal{L},(\Pi;\Delta;p\rhd\tau),\texttt{ifeq}(l){\cdot}B,\mathcal{K},PC) =\\ \quad\text{let}\ \ \Pi';\Delta';p\rhd\tau' = \mathcal{L}(l)\\ \qquad \mathcal{S}_1 = \mathit{Unify}(\mathcal{K},\{(\tau,\tau')\})\\ \qquad \mathcal{S}_2 = \mathit{UnifyStack}(\mathcal{K},\mathcal{S}_1(\Delta),\mathcal{S}_1(\texttt{int}{\cdot}\delta))\ \ (\delta\text{ fresh})\\ \qquad \mathcal{S}_3 = \mathit{UnifyStack}(\mathcal{K},\mathcal{S}_2\circ\mathcal{S}_1(\Delta),\mathcal{S}_2\circ\mathcal{S}_1(\Delta'))\\ \qquad (\mathcal{S}_4,\mathcal{K}_1,PC_1) = \mathcal{WB}(\mathcal{P},\mathcal{S}_3\circ\mathcal{S}_2\circ\mathcal{S}_1(\mathcal{L}),\mathcal{S}_3\circ\mathcal{S}_2\circ\mathcal{S}_1(\Pi;\delta;p\rhd\tau),B,\mathcal{K},PC)\\ \qquad PC_2 = PC_1\cup\{\Pi'\sqsubseteq\Pi\}\\ \quad\text{in}\ \ (\mathcal{S}_4\circ\mathcal{S}_3\circ\mathcal{S}_2\circ\mathcal{S}_1,\mathcal{K}_1,PC_2)\end{array}$$

$$\mathcal{WB}(\mathcal{P},\mathcal{L},(\Pi;\Delta;p\rhd\tau),\texttt{new}(c){\cdot}B,\mathcal{K},PC) = \mathcal{WB}(\mathcal{P},\mathcal{L},(\Pi;c{\cdot}\Delta;p\rhd\tau),B,\mathcal{K},PC)$$

$$\begin{array}{l}\mathcal{WB}(\mathcal{P},\mathcal{L},(\Pi;\Delta;p\rhd\tau),\texttt{invoke}(c,m){\cdot}B,\mathcal{K},PC) =\\ \quad\text{let}\ \ \Delta'\to\tau' = \mathit{mtype}(\Theta,c,m)\\ \qquad \mathcal{K}_1 = \mathcal{K}\cup\{t_1=*,\cdots,t_n=*,t_{n+1}=*\}\ \ (|\Delta'|=n\text{ and }t_i\text{ fresh})\\ \qquad \mathcal{S}_1 = \mathit{UnifyStack}(\mathcal{K}_1,\Delta,t_1{\cdot}\ldots{\cdot}t_n{\cdot}t_{n+1}\cdot\delta)\\ \qquad (\mathcal{S}_2,\mathcal{K}_2) = \mathit{SubType}(\mathcal{K}_1,\mathcal{S}_1(t_{n+1}),c)\\ \qquad (\mathcal{S}_3,\mathcal{K}_3) = \mathit{SubStack}(\mathcal{K}_2,\mathcal{S}_2\circ\mathcal{S}_1(t_1{\cdot}\ldots{\cdot}t_n{\cdot}\emptyset),\Delta')\\ \qquad (\mathcal{S}_4,\mathcal{K}_4,PC_1) = \mathcal{WB}(\mathcal{P},\mathcal{S}_3\circ\mathcal{S}_2\circ\mathcal{S}_1(\mathcal{L}),\mathcal{S}_3\circ\mathcal{S}_2\circ\mathcal{S}_1(\Pi;\tau'{\cdot}\delta;p\rhd\tau),B,\mathcal{K}_3,PC)\\ \qquad \{c_1,\cdots,c_n\} = \mathit{AllClasses}(\Theta,c,m)\\ \qquad PC_2 = PC_1\cup\{\Pi_1\sqsubseteq\Pi,\cdots,\Pi_n\sqsubseteq\Pi\}\ (\mathcal{P}(c_i,m)=\Pi_i\text{ or }\mathcal{N}(c_i,m)=\Pi_i)\\ \quad\text{in}\ \ (\mathcal{S}_4\circ\mathcal{S}_3\circ\mathcal{S}_2\circ\mathcal{S}_1,\mathcal{K}_4,PC_2)\end{array}$$

$$\mathcal{WB}(\mathcal{P},\mathcal{L},(\Pi;\Delta;p\rhd\tau),\texttt{priv}(\pi){\cdot}B,\mathcal{K},PC) = \text{let}\ \ \Pi' = \text{if}\ \pi\in\mathcal{A}(p)\ \text{then}\ \{\pi\}\cup\Pi\ \text{else}\ \Pi\ \ \text{in}\ \ \mathcal{WB}(\mathcal{P},\mathcal{L},(\Pi';\Delta;p\rhd\tau),B,\mathcal{K},PC)$$

Fig. 14. Type inference algorithm for blocks

6.4 An Example of Type Inference

We show how the algorithm computes a privilege environment for the sample program in Section 3.4 consisting of `Applet`, `System`, and `IO` classes. In what follows, we write A for `Applet` and S for `System`.

The main inference algorithm $\mathcal{W}$ first creates a skeleton of a privilege environment $\mathcal{P}$ using set variables ρ_1 and ρ_2 as

$$\mathcal{P} ::= \{(A, \texttt{getFile}) = \rho_1,\ (S, \texttt{readMe}) = \rho_2\}.$$

Under this privilege environment, $\mathcal{W}$ infers a type of each method using $\mathcal{WC}$ and obtains the following privilege inclusion constraint set PC.

$$PC ::= \{\rho_2 \sqsubseteq \rho_1,\ \{FRead\}\cdot\rho_2 \sqsubseteq \{FRead\},\ \rho_1 \sqsubseteq \emptyset,\ \rho_2 \sqsubseteq \{FRead\}\}$$

The first and second elements are the constraints required for invoking `readMe` and `readFile`, respectively. The third and forth elements represent the maximum privilege sets ρ_1 and ρ_2 can have under the access policy. $\mathcal{W}$ then solves PC using *SolvePC* and returns the following substitution.

$$\varphi = [\emptyset/\rho_1,\ \emptyset/\rho_2]$$

$\mathcal{W}$ finally applies φ to the skeleton of a privilege environment and obtains the following privilege environment.

$$\mathcal{P} = \{(A, \texttt{getFile}) = \emptyset,\ (S, \texttt{readMe}) = \emptyset\}$$

6.5 Soundness of the Type Inference Algorithm

In order for the type inference algorithm to serve as a static verification system for code-level access control, it must be sound with respect to the type system – a typing judgment inferred by the type inference algorithm must be deducible by the type system. In this section, we establish this desired property by showing the soundness of each sub-algorithm.

We say that $\mathcal{S}$ *is ground for* $\mathcal{K}$ if for all $t \in Dom(\mathcal{K})$, $\mathcal{S}(t)$ is not a type variable. We say that φ is *ground for* PC if for all ρ in PC, $\varphi(\rho)$ is ground. The following lemma shows the soundness of $\mathcal{WB}$.

LEMMA 9. *If* $\mathcal{WB}(\mathcal{P}, \mathcal{L}, (\Pi; \Delta; p \rhd \tau), B, \mathcal{K}, PC) = (\mathcal{S}, \mathcal{K}', PC')$ *then* $(\mathcal{S}, \mathcal{K}')$ *respects* $\mathcal{K}$*; for any* φ_0*,* $\mathcal{S}_0$ *such that* φ_0 *is ground for* PC'*,* φ_0 *satisfies* PC'*,* $\mathcal{S}_0$ *is ground for* $\mathcal{K}'$*, and* $\mathcal{S}_0$ *respects* $\mathcal{K}'$*, the following is derivable.*

$$\varphi_0(\mathcal{P}); \varphi_0(\mathcal{S}_0(\mathcal{S}(\mathcal{L}))) \vdash \varphi_0(\Pi); \mathcal{S}_0(\mathcal{S}(\Delta)); p\ \rhd B\ :\ \mathcal{S}_0(\mathcal{S}(\tau))$$

PROOF. This is proved by induction on B. The proof proceeds by case analysis in terms of the first instruction of B. We show some of the cases. Other cases are similarly shown. Suppose $\mathcal{WB}(\mathcal{P}, \mathcal{L}, (\Pi; \Delta; p \rhd \tau), B, \mathcal{K}, PC) = (\mathcal{S}, \mathcal{K}', PC')$ and $\mathcal{S}_0$, φ_0 are substitutions such that $\mathcal{S}_0$ is ground for $\mathcal{K}'$, $\mathcal{S}_0$ respects $\mathcal{K}'$, φ_0 is ground for PC', and φ_0 satisfies PC'.

Case $B = \texttt{return}$. By the definition of $\mathcal{WB}$, for a fresh δ, we have

$$\begin{aligned}&\mathcal{S} = \mathit{UnifyStack}(\mathcal{K}, \Delta, \tau\cdot\delta),\\ &\mathcal{K}' = \mathcal{K},\ \text{and}\\ &PC' = PC.\end{aligned}$$

By Lemma 7, $\mathcal{S}(\Delta) = \mathcal{S}(\tau{\cdot}\delta)$ and $\mathcal{S}$ respects $\mathcal{K}$. Thus $\mathcal{S}_0(\mathcal{S}(\Delta)) = \mathcal{S}_0(\mathcal{S}(\tau)){\cdot}\mathcal{S}_0(\mathcal{S}(\delta))$. Then by the typing rule for `return`, we have

$$\varphi_0(\mathcal{P}); \varphi_0(\mathcal{S}_0(\mathcal{S}(\mathcal{L}))) \vdash \varphi_0(\Pi); \mathcal{S}_0(\mathcal{S}(\Delta)); p \rhd \texttt{return} : \mathcal{S}_0(\mathcal{S}(\tau)).$$

Case $B = \texttt{goto(}l\texttt{)}$. By the definition of $\mathcal{WB}$, there exist some Π', Δ', τ', $\mathcal{S}_1$, $\mathcal{S}_2$ such that

$$\begin{aligned}
&\Pi'; \Delta'; p \rhd \tau' = \mathcal{L}(l),\\
&\mathcal{S}_1 = \mathit{Unify}(\mathcal{K}, \{(\tau, \tau')\}),\\
&\mathcal{S}_2 = \mathit{UnifyStack}(\mathcal{K}, \mathcal{S}_1(\Delta), \mathcal{S}_1(\Delta')),\\
&PC' = PC \cup \{\Pi' \sqsubseteq \Pi\},\\
&\mathcal{K}' = \mathcal{K}, \text{ and}\\
&\mathcal{S} = \mathcal{S}_2 \circ \mathcal{S}_1.
\end{aligned}$$

By Lemma 6, $\mathcal{S}_1(\tau) = \mathcal{S}_1(\tau')$ and $\mathcal{S}_1$ respects $\mathcal{K}$. By Lemma 7, $\mathcal{S}_2$ respects $\mathcal{K}$ and $\mathcal{S}_2(\mathcal{S}_1(\Delta)) = \mathcal{S}_2(\mathcal{S}_1(\Delta'))$ Thus by definition $\mathcal{S}_0(\mathcal{S}_2(\mathcal{S}_1(\tau))) = \varphi(\mathcal{S}_0(\mathcal{S}_2(\mathcal{S}_1(\tau'))))$ and $\mathcal{S}_0(\mathcal{S}_2(\mathcal{S}_1(\Delta))) = \varphi(\mathcal{S}_0(\mathcal{S}_2(\mathcal{S}_1(\Delta'))))$. By Lemma 4, $\mathcal{S}_2 \circ \mathcal{S}_1$ respects $\mathcal{K}$. Since φ_0 satisfies PC', $\varphi_0(\Pi') \subseteq \varphi_0(\Pi)$. Then by the typing rule for `goto`, we have

$$\varphi_0(\mathcal{P}); \varphi_0(\mathcal{S}_0(\mathcal{S}_2(\mathcal{S}_1(\mathcal{L})))) \vdash \varphi_0(\Pi); \mathcal{S}_0(\mathcal{S}_2(\mathcal{S}_1(\Delta))); p \rhd \texttt{goto(l)} : \mathcal{S}_0(\mathcal{S}_2(\mathcal{S}_1(\tau))).$$

Case $B = \texttt{priv}(\pi){\cdot}B_1$. We only show the case for $\pi \in \mathcal{A}(p)$. The other case is similar. By the definition of $\mathcal{WB}$, $(\mathcal{S}, \mathcal{K}', PC') = \mathcal{WB}(\mathcal{P}, \mathcal{L}, (\{\pi\} \cup \Pi; \Delta; p \rhd \tau), B, \mathcal{K}, PC)$. By the induction hypothesis,

$$\varphi_0(\mathcal{P}); \varphi_0(\mathcal{S}_0(\mathcal{S}(\mathcal{L}))) \vdash \varphi_0(\{\pi\} \cup \Pi); \mathcal{S}_0(\mathcal{S}(\Delta)); p \ \rhd B \ : \ \mathcal{S}_0(\mathcal{S}(\tau))$$

and $(\mathcal{S}, \mathcal{K}')$ respects $\mathcal{K}$. By the typing rule for `priv`, we have

$$\varphi_0(\mathcal{P}); \varphi_0(\mathcal{S}_0(\mathcal{S}(\mathcal{L}))) \vdash \varphi_0(\Pi); \mathcal{S}_0(\mathcal{S}(\Delta)); p \ \rhd \texttt{priv}(\pi){\cdot}B_1 \ : \ \mathcal{S}_0(\mathcal{S}(\tau)).$$

Case $B = \texttt{invoke(}c, m\texttt{)}{\cdot}B_1$. By the definition of $\mathcal{WB}$, there are some Δ', τ', $t_1, \cdots, t_{n+1}$, $\mathcal{K}_1, \cdots, \mathcal{K}_3$, $\mathcal{S}_1, \cdots, \mathcal{S}_4$, PC_1, $c_1 \cdots c_k$, $\Pi_1, \cdots, \Pi_k$ such that

$$\begin{aligned}
&\Delta' \to \tau' = \Theta(c)(m),\\
&\mathcal{K}_1 = \mathcal{K} \cup \{t_1 = *, \cdots, t_n = *, t_{n+1} = *\}\\
&\mathcal{S}_1 = \mathit{UnifyStack}(\mathcal{K}_1, \Delta, t_1{\cdot}\ldots{\cdot}t_n{\cdot}t_{n+1} \cdot \delta),\\
&(\mathcal{S}_2, \mathcal{K}_2) = \mathit{SubType}(\mathcal{K}_1, \mathcal{S}_1(t_{n+1}), c)\\
&(\mathcal{S}_3, \mathcal{K}_3) = \mathit{SubStack}(\mathcal{K}_2, \mathcal{S}_2 \circ \mathcal{S}_1(t_1{\cdot}\ldots{\cdot}t_n{\cdot}\emptyset), \Delta')\\
&(\mathcal{S}_4, \mathcal{K}', PC_1) = \mathcal{WB}(\mathcal{P}, \mathcal{S}_3 \circ \mathcal{S}_2 \circ \mathcal{S}_1(\mathcal{L}), \mathcal{S}_3 \circ \mathcal{S}_2 \circ \mathcal{S}_1(\Delta; \tau'{\cdot}\delta; p \rhd \tau), B_1, \mathcal{K}_3, PC),\\
&\{c_1, \cdots, c_k\} = \mathit{AllClasses}(\Theta, c, m),\\
&PC' = PC_1 \cup \{\Pi_1 \sqsubseteq \Pi, \cdots \Pi_k \sqsubseteq \Pi\} \ \ (\Pi_i = \mathcal{P}(c_i, m) \text{ or } \mathcal{N}(c_i, m)), \text{ and}\\
&\mathcal{S} = \mathcal{S}_4 \circ \mathcal{S}_3 \circ \mathcal{S}_2 \circ \mathcal{S}_1.
\end{aligned}$$

By Lemma 7, $\mathcal{S}_1$ respects $\mathcal{K}_1$ and $\mathcal{S}_1(\Delta) = \mathcal{S}_1(t_1{\cdot}\ldots{\cdot}t_{n+1}{\cdot}\delta)$. By Lemma 5, $(\mathcal{S}_2, \mathcal{K}_2)$ respects $\mathcal{K}_1$ and $\mathcal{K}_2 \vdash \mathcal{S}_2(\mathcal{S}_1(t_{n+1})) <: c$. Also, $(\mathcal{S}_3, \mathcal{K}_3)$ respects $\mathcal{K}_2$ and $\mathcal{K}_3 \vdash \mathcal{S}_3(\mathcal{S}_2(\mathcal{S}_1(t_1{\cdot}\ldots{\cdot}t_n{\cdot}\emptyset))) <: \Delta'$. Since φ_0 is ground for PC' and satisfies PC', φ_0 is ground for PC_1 and satisfies PC_1. By induction hypothesis for B_1,

$$\varphi_0(\mathcal{P}); \varphi_0(\mathcal{S}_0(\mathcal{S}(\mathcal{L}))) \vdash \varphi_0(\Pi); \mathcal{S}_0(\mathcal{S}(\tau'{\cdot}\delta)); p \ \rhd B_1 \ : \ \mathcal{S}_0(\mathcal{S}(\tau)),$$

and $(\mathcal{S}_4, \mathcal{K}')$ respects $\mathcal{K}_3$. By Lemma 4, $(\mathcal{S}_4 \circ \mathcal{S}_3 \circ \mathcal{S}_2 \circ \mathcal{S}_1, \mathcal{K}')$ respects $\mathcal{K}_1$. Thus, by the construction of $\mathcal{K}_1$, $(\mathcal{S}, \mathcal{K}')$ respects $\mathcal{K}$. By Lemma 3, $\mathcal{K}' \vdash \mathcal{S}(t_{n+1}) <: c$ and $\mathcal{K}' \vdash \mathcal{S}(t_1 \cdot \ldots \cdot t_n \cdot \emptyset) <: \Delta'$. Since $\mathcal{S}_0$ is ground for $\mathcal{K}'$, $\mathcal{S}_0(\mathcal{S}(t_{n+1})) <: c$ and $\mathcal{S}_0(\mathcal{S}(t_1 \cdot \ldots \cdot t_n \cdot \emptyset)) <: \Delta'$. Since φ_0 satisfies PC', $\varphi_0(\Pi_i) \subseteq \varphi_0(\Pi)$ for each i. Then by the definition of *AllPrivs* and *AllClasses*, it can be easily verified that $AllPrivs(\varphi_0(\mathcal{P}), c, m) \subseteq \varphi_0(\Pi)$. By the typing rule for `invoke`, we have

$$\varphi_0(\mathcal{S}_0(\mathcal{S}(\mathcal{P})); \varphi_0(\mathcal{S}_0(\mathcal{S}(\mathcal{L})) \vdash \varphi_0(\Pi); \mathcal{S}_0(\mathcal{S}(\Delta)); p \ \rhd \texttt{invoke}(c, m) \cdot B_1 \ : \ \mathcal{S}_0(\mathcal{S}(\tau)).$$

□

LEMMA 10. *If* $\mathcal{WM}(\mathcal{P}, c, m, M, p) = PC$ *then* $\varphi(\mathcal{P}) \vdash M^p : \Delta \stackrel{\varphi(\Pi)}{\rightarrow} \tau$ *such that* $\Theta(c)(m) = \Delta \rightarrow \tau$ *and* $\mathcal{P}(c, m) = \Pi$ *for any* φ *that is ground for* PC *and satisfies* PC.

PROOF. $\mathcal{WM}$ simply calls $\mathcal{WB}$ for each block in the method. So the result follows from Lemma 9 and the following simple property: if $\mathcal{WB}(\mathcal{P}, \mathcal{L}, (\Pi; \Delta; p \rhd \tau), B, \mathcal{K}, PC) = (\mathcal{S}, \mathcal{K}', PC')$, then if $\mathcal{S}'$ is ground for $\mathcal{K}'$ and respects $\mathcal{K}'$ then $\mathcal{S}' \circ \mathcal{S}$ is ground for $\mathcal{K}$ and respects $\mathcal{K}$, and if φ satisfies PC' then φ' also satisfies PC. □

We now prove the soundness of the main algorithm $\mathcal{W}$.

THEOREM SOUNDNESS OF $\mathcal{W}$. *Let* Θ *be a class specification such that* $\vdash \Theta$ *and* $\mathcal{N}$ *be a native method specification. If* $\mathcal{W}(\Sigma) = \mathcal{P}$ *under* $\mathcal{A}$, Θ, *and* $\mathcal{N}$ *then* $\mathcal{A}; \Theta; \mathcal{N} \vdash \Sigma : \mathcal{P}$.

PROOF. It is easily shown that the following properties hold:

(1) If $makeSkeleton(c, C, p) = \mathcal{P}$ then for each $m \in Dom(C)$, there is some Π such that $\mathcal{P}(c, m) = \Pi$.
(2) If $\mathcal{WC}(\mathcal{P}, c, C, p) = PC$ then for any φ that is ground for PC and satisfies PC, the following holds: for each $m \in Dom(C)$, $\varphi(\mathcal{P}) \vdash C(c) : \Delta \stackrel{\varphi(\Pi)}{\rightarrow} \tau$ such that $\Theta(c)(m) = \Delta \rightarrow \tau$ and $\mathcal{P}(c, m) = \Pi$.

Suppose $\mathcal{W}(\Sigma) = \mathcal{P}$ under $\mathcal{A}$, Θ, and $\mathcal{N}$. Using Properties (1) and (2) as well as Lemma 8, it can be verified that for each $c \in Dom(\Sigma)$ and for each method m in a class c, $\mathcal{P} \vdash \Sigma(c)(m) : \Delta \stackrel{\Pi}{\rightarrow} \tau$ such that $\Theta(c)(m) = \Delta \rightarrow \tau$ and $\mathcal{P}(c, m) = \Pi$. Therefore, $\mathcal{A}; \Theta; \mathcal{N} \vdash \Sigma : \mathcal{P}$. □

7. INCLUSION OF TARGET OBJECTS

One simplification we have made in the previous development is that a privilege π is an atom representing some privileged operation. In the JDK access control architecture, a privilege (permission) consists of an operation together with its *target* object set. This feature is desirable for finer grained access control. Since a target object set is computed at runtime, a static type system cannot infer its precise information. In [Pottier et al. 2005], this problem has been discussed and suggested the use of soft typing [Cartwright and Fagan 1991] to approximate target information statically with complementary dynamic check. In [Skalka and Smith 2004], the authors showed that their trace based verification system can deals with Java-style stack inspection with target objects. Instead of introducing those elaborate

mechanisms as in these approaches, we outline a simpler solution below by directly extending the type system we have developed in the previous sections.

Since a target object set is specified by character strings, such as file names or URLs, and, in many cases, they are directly given to a privileged method through string literals, our strategy is to extend the type system with a minimal mechanism to trace these literal constants appearing as arguments of some privileged native methods. For this purpose, we introduce a type of the form: $\mathtt{str}(\{s_1, \ldots, s_n\})$ for possible string values, and to give each native method m on string values as a type scheme of the form:

$$m : \forall\alpha.\Delta(\mathtt{str}(\alpha)) \overset{\{\mathsf{F}(\alpha)\}}{\rightarrow} \tau$$

where $\Delta(\mathtt{str}(\alpha))$ is type Δ containing $\mathtt{str}(\alpha)$ as one of its element types. This typing judgment describes the fact that the method m accepts arguments of type $\Delta(\mathtt{str}(\alpha))$ including a string value of type $\mathtt{str}(\alpha)$ and performs a privileged operation F on the string value. Whenever such a privileged native method is invoked with some arguments, the type system instantiates its type scheme, unifies it with the actual argument type, and deduces the required privileges. Suppose m above is invoked with arguments containing a string literal s in the target position. The type system infers a type of the form $\Delta(\mathtt{str}(\{\mathtt{s}\}))$ for the arguments, unifies it with $\Delta(\mathtt{str}(\alpha))$, and deduces the privilege $F(\{s\})$ required for this invocation. If the target expression is not a string literal, then the type system infers $\mathtt{str}(\top)$ and deduces that the invoking expression requires the privilege $F(\top)$, i.e., the ability to perform F on all possible target values.

The inferred privilege is either of the forms: $F(\{s_1, \ldots, s_n\})$ or $F(\top)$. The former case contains a set of literals. This is due to multiple control flows, and any member of the set can be an actual target of F. We consider this form of privilege to be "known" and generate a privilege constraint of the form $F(\{s_1, \ldots, s_n\}) \subseteq \Pi$ for this invocation. $F(\top)$ indicates that a target object set is statically unknown. In this case, the type system does not generate any privilege constraint for this method invocation; instead, it inserts a special instruction `check` that performs dynamic access check on the actual target at runtime.

On the basis of this strategy, we describe below the refinements necessary for our type system.

7.1 The Refined Language

For the refined language, we assume that we are given the following set of privileges π including target object sets (ranged over by v).

$$\begin{aligned} \pi &::= F(v) \\ v &::= \{s_1, \ldots, s_n\} \mid \top \end{aligned}$$

Target object sets are ordered by set inclusion with $\top$ being the largest element. This ordering is extended to π, i.e., $F(v_1) \subseteq F(v_2)$ if $v_1 \subseteq v_2$. We write $\Pi_1 \subseteq \Pi_2$ if for any $\pi_1 \in \Pi_1$, there is some $\pi_2 \in \Pi_2$ such that $\pi_1 \subseteq \pi_2$.

The required refinements for the syntax of JVM_{sec} involve incorporating these refined privileges in an access policy $\mathcal{A}$ and in a native method specification $\mathcal{N}$ and extending the set of instructions to support complementary dynamic access check.

As before, the syntax of access policy $\mathcal{A}$ is a function from a set of principals p to a privilege set Π. In the following development, we assume that Π is in canonical form satisfying the property: if $F(v) \in \Pi$ and $F'(v') \in \Pi$ for some $v \neq v'$, then $F \neq F'$. We also assume that a dynamic privilege set P is in canonical form.

A native method specification $\mathcal{N}$ specifies privileges used by each native method. For simplicity of presentation, we only consider the cases where native methods perform a maximum of one privileged operation. Generalization to multiple privileged operations can be done by refining the notation of type scheme without much difficulty. Under this assumption, a native method has a type scheme of the form $\forall\alpha.\Delta(\mathtt{str}(\alpha)) \stackrel{\mathsf{F}(\alpha)}{\rightarrow} \tau$, where α is a *target set variable*. Since type information is specified by Θ, $\mathcal{N}$ needs to specify only the privilege information represented by such a type scheme, i.e., the operation name F and the position indicated by the target set variable α. The required information is represented by $F(n)$, where n is the relative index of $\mathtt{str}(\alpha)$ in Δ. On the basis of this strategy, we refine the syntax of native method specifications as follows.

$$\mathcal{N} ::= \{(c_1, m_1) \mapsto F(n_1), \cdots, (c_k, m_k) \mapsto F(n_k)\}$$

We assume that a native method specification is correctly declared.

The set of instructions is extended as follows.

$$I ::= \cdots \mid \mathtt{sconst}(s) \mid \mathtt{check}(F, n)$$

`sconst(`s`)` pushes string constant s onto the stack. `check(`F, n`)` retrieves the value v stored at the n-th position of the stack and checks whether or not the current dynamic privilege set P contains $F(v)$; it fails with `secfail` if this condition is not satisfied. The n in `check(`F, n`)` indicates that the target of F is given as the n-th argument of the method being checked. Transition rules for these instructions are shown in Figure 15, which also shows the refined rules for `invoke`. The rules for other instructions are the same as before.

7.2 The Refined Type System

The syntax of types is extended as follows.

$$\tau ::= \mathtt{int} \mid c \mid \mathtt{str}(v)$$

$\mathtt{str}(v)$ represents the subset of strings denoted by v, which is used to statically approximate the runtime value of an expression. For example, $\mathtt{str}(\{s_1, s_2\})$ denotes the set $\{s_1, s_2\}$ and indicates that the runtime value is either s_1 or s_2. $\mathtt{str}(\top)$ denotes the set of all strings and corresponds to the ordinary string type in the conventional type system of JVM. We simply write `str` for $\mathtt{str}(\top)$. To represent the set inclusion relation on target object sets, we define a subtype relation $\preceq$ as follows.

$$\frac{}{\tau \preceq \tau} \qquad \frac{v_1 \subseteq v_2}{\mathtt{str}(v_1) \preceq \mathtt{str}(v_2)} \qquad \frac{\tau_1 \preceq \tau_2 \quad \tau_2 \preceq \tau_3}{\tau_1 \preceq \tau_3}$$

From this definition, $\mathtt{str}(v) \preceq \mathtt{str}$ for any v. This relation is extended to stack types as follows.

$$\Delta_1 \preceq \Delta_2 \iff \text{if } \Delta_1.i \preceq \Delta_2.i \text{ for each } i$$

$$(P, S, M^p\{\texttt{sconst}(s)\cdot B\}, D); h \longrightarrow (P, s\cdot S, M^p\{B\}, D); h$$

$$(P, S, M^p\{\texttt{check}(F, n)\cdot B\}, D); h \longrightarrow (P, S, M^p\{B\}, D); h \quad \text{if } F(\{S.n\}) \in P$$

$$P, S, M^p\{\texttt{check}(F, n)\cdot B\}, D); h \longrightarrow (\emptyset, \texttt{secfail}, \emptyset, \emptyset); h \quad \text{if } F(\{S.n\}) \notin P$$

$$(P, S_1\cdot r\cdot S, M^p\{\texttt{invoke}(c, m)\cdot B\}, D); h \longrightarrow (P', S_1\cdot r\cdot\emptyset, M'^{p'}\{M'(entry)\}, (P, S, M^p\{B\})\cdot D); h$$
$$\text{if} \begin{cases} h(r) = \langle\rangle_{c_0}, \quad c_1 = mclass(\Theta, c_0, m), \quad ArgSize(\Theta, c_1, m) = |S_1|, \\ (c_1, m) \notin Dom(\mathcal{N}), \quad mbody(\Sigma, c_1, m) = M'^{p'} \text{ and } P' = P \cap \mathcal{A}(p') \end{cases}$$

$$(P, S_1\cdot r\cdot S, M^p\{\texttt{invoke}(c, m)\cdot B\}, D); h \longrightarrow (P, v\cdot S, M^p\{B\}, D); h'$$
$$\text{if} \begin{cases} h(r) = \langle\rangle_{c_0}, \quad c_1 = mclass(\Theta, c_0, m), \quad ArgSize(\Theta, c_1, m) = |S_1|, \quad (c_1, m) \in Dom(\mathcal{N}), \\ \mathcal{N}(c_1, m) = F(n), \quad F(\{S_1.n\}) \in P, \quad \Omega(c, m) = f \text{ and } \quad f(S_1, h) = (v, h') \end{cases}$$

$$(P, S_1\cdot r\cdot S, M^p\{\texttt{invoke}(c, m)\cdot B\}, D); h \longrightarrow (\emptyset, \texttt{secfail}, \emptyset, \emptyset); h$$
$$\text{if} \begin{cases} h(r) = \langle\rangle_{c_0}, \quad c_1 = mclass(\Theta, c_0, m), \quad (c_1, m) \in Dom(\mathcal{N}), \\ \mathcal{N}(c_1, m) = F(n) \text{ and } F(\{S_1.n\}) \notin P \end{cases}$$

Fig. 15. Transition rules for some instructions

$$\Pi; \Delta; p \rhd \texttt{goto}(l) : \tau \quad (\text{if } \mathcal{L}(l) = \Pi'; \Delta'; p \rhd \tau', \ \Pi' \subseteq \Pi, \text{ and } \Delta \rhd \tau \preceq \Delta' \rhd \tau')$$

$$\frac{\Pi; \Delta; p \rhd B : \tau}{\Pi; \texttt{int}\cdot\Delta; p \rhd \texttt{ifeq}(l)\cdot B : \tau} \quad (\text{if } \mathcal{L}(l) = \Pi'; \Delta'; p \rhd \tau', \ \Pi' \subseteq \Pi, \text{ and } \Delta \rhd \tau \preceq \Delta' \rhd \tau')$$

$$\frac{\Pi; \texttt{str}(\{\texttt{s}\})\cdot\Delta; p \rhd B : \tau}{\Pi; \Delta; p \rhd \texttt{sconst}(s)\cdot B : \tau}$$

$$\frac{\Pi; \tau_1\cdot\Delta; p \rhd B : \tau}{\Pi; \Delta_0\cdot c_0\cdot\Delta; p \rhd \texttt{invoke}(c, m)\cdot B : \tau} \quad \left(\text{if} \begin{array}{l} mtype(\Theta, c, m) = \Delta_1 \to \tau_1, \\ \Delta_0 <: \Delta_1, \quad c_0 <: c, \text{ and} \\ AllPrivs(\mathcal{P}, c, m, \Delta_0) \subseteq \Pi \end{array} \right)$$

$$\frac{\Pi; \tau_1\cdot\Delta; p \rhd B : \tau}{\Pi; \Delta_0\cdot c_0\cdot\Delta; p \rhd \texttt{check}(F, n)\cdot\texttt{invoke}(c, m)\cdot B : \tau} \quad \left(\text{if} \begin{array}{l} mtype(\Theta, c, m) = \Delta_1 \to \tau_1, \\ \Delta_0 <: \Delta_1, \quad c_0 <: c, \\ \Delta_0.n = \texttt{str}(v), \text{ and} \\ (AllPrivs(\mathcal{P}, c, m, \Delta_0) \setminus \{F(v)\}) \subseteq \Pi \end{array} \right)$$

Fig. 16. Typing rules for blocks (excerpts)

A target object set will be promoted through this relation when control flow merges. The subclass relation $<:$ is also extended to string types as $\texttt{str}(v) <: \texttt{str}$. This is necessary to verify the type constraint specified in a class specification Θ.

The type system is constructed relative to a privilege environment. The typing rules for programs and methods are the same as before. Typing rules for blocks are refined to check inclusion constraints on target object sets. Figure 16 shows some of the refined typing rules for code blocks. Typing rules for other instructions are the same as before.

The typing rule for `goto` checks the subtype relation $\Delta \rhd \tau \preceq \Delta' \rhd \tau'$, which is an abbreviation for $\Delta \preceq \Delta'$ and $\tau \preceq \tau'$. This enforces that each target object set at the entry point of the code block labeled l must include the corresponding

$$
\begin{array}{l}
\mathit{AllPrivs}(\mathcal{P}, c, m, \Delta) = \\
\quad \text{let } \Pi_1 = \bigcup\{\Pi \mid \mathcal{P}(c', m) = \Pi \text{ for some } c' \text{ such that } c' <: c\} \\
\qquad \Pi_2 = \{F(v) \mid \mathcal{N}(c', m) = F(n),\ \mathtt{str}(v) = \Delta.n \text{ for some } c' \text{ such that } c' <: c\} \\
\quad \text{in if } (c, m) \in \mathit{Dom}(\mathcal{P}) \text{ or } (c, m) \in \mathit{Dom}(\mathcal{N}) \text{ then } \Pi_1 \cup \Pi_2 \\
\qquad \text{else let } c_1 = \mathit{mclass}(\Theta, c, m) \\
\qquad\quad \text{in if } (c_1, m) \in \mathit{Dom}(\mathcal{P}) \text{ then } \Pi_1 \cup \Pi_2 \cup \mathcal{P}(c_1, m) \\
\qquad\qquad \text{else let } F(n) = \mathcal{N}(c_1, m) \\
\qquad\qquad\qquad\quad \mathtt{str}(v) = \Delta.n \\
\qquad\qquad\quad \text{in } \Pi_1 \cup \Pi_2 \cup \{F(v)\}
\end{array}
$$

Fig. 17. The definition of the refined *AllPrivs* algorithm

target object set at the current program point. The typing rule for `ifeq` also enforces this relation. The rule for `invoke` computes the union of privilege sets of the methods, which can be invoked at runtime using the refined auxiliary function *AllPrivs* given in Figure 17, and checks whether it is included in the current privilege set. *AllPrivs* is now defined relative to a given context Δ, which is used to determine the instance privilege for each native method invocation. The typing rule for `check` is defined together with the accompanying `invoke` since the `check` instruction is complementary to `invoke`. It removes the specified privilege from the set of required privileges for the accompanying `invoke` instruction. In the type system (static semantics), a `check` instruction and the subsequent `invoke` instruction together act as one constructor. In the dynamic semantics, a `check` instruction performs dynamic check using the arguments for the subsequent `invoke` instruction without consuming them. Although we do not require it in the typing rule, the only interesting usage of `check` is the case where $\mathcal{N}(c, m) = F(n)$.

We note that refined string types of the form $\mathtt{str}(v)(v \neq \top)$ are introduced only through constant literals and explicitly annotated native methods. String types specified in ordinary methods remain `str`, i.e. $\mathtt{str}(\top)$. So for example stack entries corresponding to formal parameters are typed with $\mathtt{str}(\top)$.

7.3 Refined Type Inference

In this subsection, we describe a type inference algorithm for the refined type system. In addition to inferring a privilege environment, the type inference algorithm collects a set of *check points* where the `check` instruction is inserted. The part for inferring a privilege environment is essentially the same as before; it infers a type of each method in a program using a unification algorithm and computes a minimal privilege set for each method by generating a privilege inclusion constraint set PC and solving it. Collecting a set of check points is performed in the following two steps. First, for each native method invocation in a program, its target object set is computed by generating a set of constraints on target object set variables and solving the constraints. Second, the algorithm identifies the native method invocations whose target object set is $\top$ and marks them as check points. In the following subsections, we first describe the refinements that are necessary for the unification algorithm. We then outline the procedure for collecting check points and describe the type inference algorithm.

$$
\begin{array}{l}
\mathit{SubStack}(\tau_1{\cdot}\Delta_1, \tau_2{\cdot}\Delta_2, \mathcal{K}) = \text{let } (\mathcal{S}_1, \mathcal{K}_1) = \mathit{SubType}(\tau_1, \tau_2, \mathcal{K}) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathcal{S}_2, \mathcal{K}_2) = \mathit{SubStack}(\mathcal{S}_1(\Delta_1), \mathcal{S}_1(\Delta_2), \mathcal{K}_1) \\
\qquad\qquad\qquad\qquad\qquad \text{in } (\mathcal{S}_2 \circ \mathcal{S}_1, \mathcal{K}_2) \\
\mathit{SubStack}(\emptyset, \emptyset, \mathcal{K}) = (\emptyset, \mathcal{K}) \\
\\
\mathit{SubType}(t, \mathtt{str}, \mathcal{K}) = ([\mathtt{str}(\alpha)/t], \mathcal{K}) \quad (\mathcal{K}(t) = *,\ \alpha \text{ fresh}) \\
\mathit{SubType}(\mathtt{str}(v), \mathtt{str}, \mathcal{K}) = (\emptyset, \mathcal{K}) \\
\mathit{SubType}(t, c, \mathcal{K}) = \text{if } \mathcal{K}(t) = * \text{ or } c <: \mathcal{K}(t) \text{ then } ([t'/t], \mathcal{K} \cup \{t' = c\}) \quad (t' \text{ fresh}) \\
\qquad\qquad\qquad\qquad \text{else if } \mathcal{K}(t) <: c \text{ then } (\emptyset, \mathcal{K}) \\
\mathit{SubType}(c_1, c_2, \mathcal{K}) = \text{if } (c_1 <: c_2) \text{ then } (\emptyset, \mathcal{K}) \\
\mathit{SubType}(\mathtt{int}, \mathtt{int}, \mathcal{K}) = (\emptyset, \mathcal{K}) \\
\mathit{SubType}(t, \mathtt{int}, \mathcal{K}) = ([\mathtt{int}/t], \mathcal{K}) \\
\mathit{SubType}(\tau, \tau') = \mathit{Failure}
\end{array}
$$

Fig. 18. The extended *SubStack* and *SubType* algorithm

7.3.1 *Refined subtype checking algorithms.* In order to infer a target object set, we refine the language of target object sets v to include target set variable α as follows.

$$v ::= \{s_1, \ldots, s_n\} \mid \{s_1, \ldots, s_n\}{\cdot}\alpha \mid \top$$

We simply write α for $\emptyset{\cdot}\alpha$. A *target set variable substitution* (ranged over by ψ) is a function from a finite set of target set variables to target object sets. We identify ψ with its homomorphic extension to any syntactic structure containing target set variables.

We extend the subclass checking algorithms to include type $\mathtt{str}(v)$. The extended algorithms *SubStack* and *SubType* are given in Figure 18. $\mathit{SubStack}(\Delta, \Delta', \mathcal{K})$ returns $(\mathcal{S}, \mathcal{K}')$ such that $(\mathcal{S}, \mathcal{K}')$ respects $\mathcal{K}$ and $\mathcal{K}' \vdash \psi(\mathcal{S}(\Delta)) <: \Delta'$ for any ψ.

The refined type inference algorithm given below infers a typing for each code block in a method independently using a unification algorithm and collects a set of inclusion constraints to be satisfied by these typings without unifying them. Since we use unification only to check the structural equality and treat subtype relation involving target set variables through constraint solving, we can use the previous unification algorithm for terms with type variables.

7.3.2 *Collecting check points.* The type inference algorithm first assigns a fresh target set variable to each native method invocation as its target object set. It then generates a *target inclusion constraint set* of the form:

$$\{v_1 \ll v_1', \cdots, v_n \ll v_n'\}$$

and infers a ground set of target objects for each native method invocation by solving the constraint set. We use OC as a meta variable for target inclusion constraint sets. We say that ψ satisfies OC if $\psi(v) \subseteq \psi(v')$ for each $v \ll v' \in OC$. The algorithm *SolveOC* takes OC and computes ψ that satisfies OC. Its definition is similar to that of *SolvePC*.

A target inclusion constraint set OC is extracted from a *type inclusion constraint set* of the form

$$\{\Delta_1 \rhd \tau_1 \ll \Delta_1' \rhd \tau_1', \cdots, \Delta_n \rhd \tau_n \ll \Delta_n' \rhd \tau_n'\}.$$

$$
\begin{array}{l}
\mathit{SolveTC}(TC,\mathcal{K}) = \text{if } TC = \emptyset \text{ then } (\emptyset,\emptyset) \\
\qquad\qquad \text{else let } \{\Delta \rhd \tau \ll \Delta' \rhd \tau'\} \cup TC' = TC \\
\qquad\qquad\qquad \text{such that } \Delta \text{ is doesn't contain a stack variable.} \\
\qquad\qquad\qquad (\mathcal{S}_1, OC_1) = \mathit{SolveType}(\tau,\tau',\mathcal{K}) \\
\qquad\qquad\qquad (\mathcal{S}_2, OC_2) = \mathit{SolveStack}(\mathcal{S}_1(\Delta), \mathcal{S}_1(\Delta'), \mathcal{K}) \\
\qquad\qquad\qquad \mathcal{S}_3 = \mathcal{S}_2 \circ \mathcal{S}_1 \\
\qquad\qquad\qquad (\mathcal{S}_4, OC_3) = \mathit{SolveTC}(\mathcal{S}_3(TC'),\mathcal{K}) \\
\qquad\qquad\quad \text{in } (\mathcal{S}_4 \circ \mathcal{S}_3, OC_1 \cup OC_2 \cup OC_3) \\
\\
\mathit{SolveStack}(\tau{\cdot}\Delta, \delta, \mathcal{K}) = \text{let } \mathcal{K}' = \mathcal{K} \cup \{t = *\} \quad (t \text{ fresh}) \\
\qquad\qquad\qquad\qquad (\mathcal{S}_1, OC_1) = \mathit{SolveType}(\tau, t, \mathcal{K}') \\
\qquad\qquad\qquad\qquad \mathcal{S}_2 = [\mathcal{S}_1(t){\cdot}\delta'/\delta] \quad (\delta' \text{ fresh}) \\
\qquad\qquad\qquad\qquad \mathcal{S}_3 = \mathcal{S}_2 \circ \mathcal{S}_1 \\
\qquad\qquad\qquad\qquad (\mathcal{S}_4, OC_2) = \mathit{SolveStack}(\mathcal{S}_3(\Delta), \delta', \mathcal{K}') \\
\qquad\qquad\qquad\quad \text{in } (\mathcal{S}_4 \circ \mathcal{S}_3, OC_1 \cup OC_2) \\
\mathit{SolveStack}(\tau{\cdot}\Delta, \tau'{\cdot}\Delta', \mathcal{K}) = \text{let } (\mathcal{S}_1, OC_1) = \mathit{SolveType}(\tau,\tau',\mathcal{K}) \\
\qquad\qquad\qquad\qquad\qquad (\mathcal{S}_2, OC_2) = \mathit{SolveStack}(\mathcal{S}_1(\Delta), \mathcal{S}_1(\Delta'), \mathcal{K}) \\
\qquad\qquad\qquad\qquad\quad \text{in } (\mathcal{S}_2 \circ \mathcal{S}_1, OC_1 \cup OC_2) \\
\mathit{SolveStack}(\Delta, \Delta', \mathcal{K}) = \text{let } \mathcal{S} = \mathit{UnifyStack}(\mathcal{K}, \Delta, \Delta') \text{ in } (\mathcal{S}, \emptyset) \\
\\
\mathit{SolveType}(\mathtt{str}(v), t, \mathcal{K}) = ([\mathtt{str}(\alpha)/t], \{v \ll \alpha\}) \quad (\alpha \text{ fresh}) \\
\mathit{SolveType}(\mathtt{str}(v_1), \mathtt{str}(v_2), \mathcal{K}) = (\emptyset, \{v_1 \ll v_2\}) \\
\mathit{SolveType}(\mathtt{str}(v), \tau, \mathcal{K}) = \mathit{Failure} \\
\mathit{SolveType}(t, \mathtt{str}(v), \mathcal{K}) = ([\mathtt{str}(\alpha)/t], \{\alpha \ll v\}) \quad (\alpha \text{ fresh}) \\
\mathit{SolveType}(\tau, \mathtt{str}(v), \mathcal{K}) = \mathit{Failure} \\
\mathit{SolveType}(\tau_1, \tau_2, \mathcal{K}) = \text{let } \mathcal{S} = \mathit{Unify}(\mathcal{K}, \{(\tau_1, \tau_2)\}) \text{ in } (\mathcal{S}, \emptyset)
\end{array}
$$

Fig. 19. Algorithm *SolveTC*

We use TC as a meta variable for type inclusion constraint sets. The type inference algorithm generates a type inclusion constraint set TC for each method body. Each element of TC represents a constraint to be satisfied by some branch instruction between two code blocks. We say that a pair $(\mathcal{S}, \psi)$ consisting of a substitution $\mathcal{S}$ and a target variable substitution ψ *satisfies* TC if $\psi(\mathcal{S}(\Delta \rhd \tau)) \preceq \psi(\mathcal{S}(\Delta' \rhd \tau'))$ for each $\Delta \rhd \tau \ll \Delta' \rhd \tau' \in TC$. Figure 19 shows the algorithm *SolveTC* which accepts TC and $\mathcal{K}$ and generates a pair $(\mathcal{S}, OC)$ such that $\mathcal{S}$ respects $\mathcal{K}$ and $(\mathcal{S}, \psi)$ satisfies TC for any ψ satisfying OC.

After inferring a type of each method in a program and obtaining a privilege inclusion constraint PC, the type inference algorithm collects check points. This is done by checking PC. Since each element in PC is a constraint of the form $\Pi \sqsubseteq \Pi'$ to be satisfied by some specific method invocation in a program, it is sufficient to find elements of the form $\{F(\top)\} \sqsubseteq \Pi$ that correspond to native method invocations whose target object set is statically unknown. To achieve this, we mark a constraint $\Pi \sqsubseteq \Pi'$ in PC that corresponds to a native method invocation with the instruction address of the method invocation. We write $\langle \Pi \sqsubseteq \Pi' \rangle_a$ for a marked constraint with address a. Figure 20 gives the algorithm *FindChecks* that collects a set of check points using this strategy. It accepts PC and returns a pair (Φ, PC') consisting of the set Φ of instruction addresses at which the `check` instruction must be inserted and the privilege inclusion constraint set PC' obtained from PC by removing the

$$
\begin{array}{l}
\mathit{FindChecks}(PC) = \text{if there is no } a \text{ such that } \langle \Pi \sqsubseteq \Pi' \rangle_a \in PC \text{ then } (PC, \emptyset) \\
\qquad \text{else let } \{\langle \Pi \sqsubseteq \Pi' \rangle_a\} \cup PC_1 = PC \ \text{ for some } a \\
\qquad\qquad (\Phi, PC_2) = \mathit{FindChecks}(PC_1) \\
\qquad\qquad F(v) = \Pi \\
\qquad\quad \text{in } \ \text{if } v = \top \text{ then } (\Phi \cup \{a\}, PC_2) \\
\qquad\qquad \text{else } (\Phi, PC_2 \cup \{\Pi \sqsubseteq \Pi'\})
\end{array}
$$

Fig. 20. Algorithm *FindChecks*

elements that are checked by `check` at runtime.

7.3.3 *The type inference algorithm.* The algorithm consists of sub-algorithms $\mathcal{WB}_t$ for blocks, $\mathcal{WM}_t$ for methods, and $\mathcal{W}_t$.

Figure 21 gives the definition of $\mathcal{WB}_t$ for some instructions. The cases for other instructions are defined as before. $\mathcal{WB}_t$ infers a typing of a code block and generates a type inclusion constraint set TC in addition to PC. The auxiliary function *GetIA* used in the case of `invoke` returns the address of the current `invoke` instruction.

Figure 22 gives the refined algorithm $\mathcal{WM}_t$ for methods. Its overall structure is the same as before except that it solves TC using *SolveTC* and obtains a target inclusion constraint set OC, and then solves OC using *SolveOC* to obtain a target set variable substitution. The main algorithm $\mathcal{W}_t$, shown in Figure 23, is refined to collect check points using *FindChecks*.

8. DYNAMIC CLASS LOADING

The system we have developed is based on the assumption that the set of all class files of a program is given statically. This assumption is incompatible with dynamic class loading [Liang and Bracha 1998] – one of important features of the JVM. In order for our type system to scale up to the full-fledged JVM system, it must be further refined so that it can also type-check dynamically loaded classes. This section describes refinements of the type system for dynamic class loading and outlines a type inference algorithm for the refined type system.

8.1 Refined Typing for Dynamic Class Loading

In the following development, we assume that a class specification Θ, a native method specification $\mathcal{N}$ and an access policy $\mathcal{A}$ are implicitly given and fixed. We also assume that a subclass relation $<:$ is implicitly given.

In our original type system, the main typing judgment is of the form $\vdash \Sigma : \mathcal{P}$ where Σ is a complete program consisting of all its classes and $\mathcal{P}$ is a complete privilege environment for the program. Our approach is to extend the type system so that it type-checks classes *incrementally*.

To model dynamic class loading, we consider a JVM program as a set of classes Λ_i inductively defined as follows:

$$
\begin{aligned}
\Lambda_0 &= \Sigma_0 \cup \Psi_0 \\
\Lambda_{i+1} &= \Lambda_i \cup \Psi_{i+1}
\end{aligned}
$$

$$\mathcal{WB}_t(\mathcal{P},\mathcal{L},(\Pi;\Delta;p\rhd\tau),\texttt{goto}(l),\mathcal{K},PC,TC) = \text{let}\ \ \Pi';\Delta';p\rhd\tau' = \mathcal{L}(l)\ \ PC_1 = PC\cup\{\Pi'\sqsubseteq\Pi\}\ \ TC_1 = TC\cup\{\Delta\rhd\tau\ll\Delta'\rhd\tau'\}\ \ \text{in}\ \ (\emptyset,\mathcal{K},PC_1,TC_1)$$

$$\begin{array}{l}\mathcal{WB}_t(\mathcal{P},\mathcal{L},(\Pi;\Delta;p\rhd\tau),\texttt{ifeq}(l)\cdot B,\mathcal{K},PC,TC) =\\ \quad\text{let}\ \ \Pi';\Delta';p\rhd\tau' = \mathcal{L}(l)\\ \qquad \mathcal{S}_1 = \mathit{UnifyStack}(\mathcal{K},\Delta,\texttt{int}\cdot\delta)\quad(\delta\ \text{fresh})\\ \qquad (\mathcal{S}_2,\mathcal{K}_1,PC_1,TC_1) = \mathcal{WB}_t(\mathcal{P},\mathcal{S}_1(\mathcal{L}),\mathcal{S}_1(\Pi;\delta;p\rhd\tau),B,\mathcal{K},PC,TC)\\ \qquad PC_2 = PC_1\cup\{\Pi'\sqsubseteq\Pi\}\\ \qquad TC_2 = TC_1\cup\{\delta\rhd\tau\ll\Delta'\rhd\tau'\}\\ \quad\text{in}\ \ (\mathcal{S}_2\circ\mathcal{S}_1,\mathcal{K}_3,PC_2,TC_2)\end{array}$$

$$\mathcal{WB}_t(\mathcal{P},\mathcal{L},(\Pi,\Delta,p\rhd\tau),\texttt{sconst}(s)\cdot B,\mathcal{K},PC,TC) = \mathcal{WB}_t(\mathcal{P},\mathcal{L},(\Pi,\texttt{str}(\{s\})\cdot\Delta,p\rhd\tau),B,\mathcal{K},PC,TC)$$

$$\begin{array}{l}\mathcal{WB}_t(\mathcal{P},\mathcal{L},(\Pi;\Delta;p\rhd\tau),\texttt{invoke}(c,m)\cdot B,\mathcal{K},PC,TC) =\\ \quad\text{let}\ \ \Delta'\to\tau' = \Theta(c)(m)\\ \qquad \mathcal{K}_1 = \mathcal{K}\cup\{t_1=*,\cdots,t_n=*,t_{n+1}=*\}\ (n=|\Delta'|)\\ \qquad \mathcal{S}_1 = \mathit{UnifyStack}(\mathcal{K}_1,t_1\cdot\ldots\cdot t_n\cdot t_{n+1}\cdot\delta,\Delta)\\ \qquad (\mathcal{S}_2,\mathcal{K}_2) = \mathit{SubType}(\mathcal{K}_1,\mathcal{S}_1(t_{n+1}),c)\\ \qquad (\mathcal{S}_3,\mathcal{K}_3) = \mathit{SubStack}(\mathcal{K}_3,\mathcal{S}_2\circ\mathcal{S}_1(t_1\cdot\ldots\cdot t_n\cdot\emptyset),\Delta')\\ \qquad \{c_1,\cdots,c_k\} = \mathit{AllClasses}(\Theta,c,m)\\ \qquad PC_0 = PC\\ \qquad \text{for each } i \text{ do}\\ \qquad\quad \text{if } (c_i,m)\in\mathcal{N}\ \wedge\ \mathcal{N}(c_i,m) = F(j) \text{ then}\\ \qquad\qquad \text{if } \mathcal{S}_3\circ\mathcal{S}_2\circ\mathcal{S}_1(t_j) = \texttt{str}(v) \text{ then } PC_i = PC_{i-1}\cup\{\langle F(v)\sqsubseteq\Pi\rangle_a\}\quad(a=\mathit{GetIA}())\\ \qquad\qquad \text{else } \mathit{Failure}\\ \qquad\quad \text{else}\ \ PC_i = PC_{i-1}\cup\{\Pi'\sqsubseteq\Pi\}\quad(\Pi' = \mathcal{P}(c_i)(m))\\ \qquad \text{end}\\ \qquad (\mathcal{S}_3,\mathcal{K}_3,PC',TC') = \mathcal{WB}_t(\mathcal{P},\mathcal{S}_2\circ\mathcal{S}_1(\mathcal{L}),\mathcal{S}_2\circ\mathcal{S}_1(\Pi;\tau'\cdot\delta;p\rhd\tau;p),B,\mathcal{K}_2,PC_k,TC)\\ \quad\text{in}\ \ (\mathcal{S}_3\circ\mathcal{S}_2\circ\mathcal{S}_1,\mathcal{K}_3,PC',TC')\end{array}$$

$$\mathcal{WB}_t(\mathcal{P},\mathcal{L},(\Pi;\Delta;p\rhd\tau),\texttt{priv}(\pi)\cdot B,\mathcal{K},PC,TC) = \text{let}\ \ \Pi' = \text{if } \pi\in\mathcal{A}(p) \text{ then } \{\pi\}\cup\Pi \text{ else } \Pi\ \ \text{in}\ \ \mathcal{WB}_t(\mathcal{P},\mathcal{L},(\Pi';\Delta;p\rhd\tau),B,\mathcal{K},PC,TC)$$

Fig. 21. Type inference algorithm for blocks (excerpts)

where Σ_0 is the stable part of a program typically containing library classes and Ψ_i is the set of classes dynamically loaded at time i. Ψ_0 corresponds to the user's main program and Λ_i corresponds to the program state at time i. Ψ_i (and therefore Λ_i) may refer to unloaded classes.

We assume that Σ_0 is closed, i.e., it does not refer to any unloaded classes. For this part, we use our original type system since it is more accurate than the incremental version for Λ described below.

For Λ and Ψ, we deduce typing judgments of the forms

$$\mathcal{P}^F\ \vdash\ \Lambda:\mathcal{P}$$
$$\mathcal{P}^F;\mathcal{P}^E\ \vdash\ \Psi:\mathcal{P}^I$$

where $\mathcal{P}^F$ is a privilege environment of the classes that will be loaded later and $\mathcal{P}^E$ is a privilege environment of those that have been already loaded.

The typing relation $\mathcal{P}^F\vdash\Lambda:\mathcal{P}$ is given by the following simple induction:

$$
\begin{array}{l}
\mathcal{WM}_t(\mathcal{P}, c, m, M, p) = \\
\quad \text{let} \;\; \{l_1 = B_1, \cdots, l_n = B_n\} = M \\
\qquad \mathcal{B}_k = \Pi; \Delta; p \rhd \tau \quad (l_k = \text{entry},\ \Theta(c)(m) = \Delta \to \tau,\ \mathcal{P}(c,m) = \Pi) \\
\qquad \mathcal{B}_i = \rho_i; \delta_i; p \rhd \tau \quad (\rho_i, \delta_i \text{ fresh},\ 1 \le i \le n, l_i \neq \mathit{entry}) \\
\qquad \mathcal{L} = \{l_1 = \mathcal{B}_1, \cdots, l_n = \mathcal{B}_n\} \\
\qquad \mathcal{S}_0 = \emptyset,\ \mathcal{K}_0 = \emptyset \\
\qquad PC_0 = \{\rho_1 \sqsubseteq \mathcal{A}(p), \cdots, \rho_n \sqsubseteq \mathcal{A}(p), \Pi \sqsubseteq \mathcal{A}(p)\} \\
\qquad TC_0 = \emptyset \\
\qquad \text{for each i do} \\
\qquad\quad (\mathcal{S}, \mathcal{K}_i, PC_i, TC_i) = \mathcal{WB}_t(\mathcal{P}, \mathcal{S}_{i-1}(\mathcal{L}), \mathcal{S}_{i-1}(\mathcal{B}_i), B_i, \mathcal{K}_{i-1}, PC_{i-1}, TC_{i-1}) \\
\qquad\quad \mathcal{S}_i = \mathcal{S} \circ \mathcal{S}_{i-1} \\
\qquad \text{end} \\
\qquad (\mathcal{S}, OC) = \mathit{SolveTC}(\mathcal{S}_n(TC_n)) \\
\qquad \psi = \mathit{SolveOC}(OC) \\
\quad \text{in} \;\; (\psi(PC_n))
\end{array}
$$

Fig. 22. Type inference algorithm for methods

$$
\begin{array}{l}
\mathcal{W}_t(\Sigma) = \text{let} \;\; \{c_1 = (p_1, C_1), \cdots, c_n = (p_n, C_n)\} = \Sigma \\
\qquad\qquad\quad \mathcal{P}_i = \mathit{makeSkeleton}(c_i, C_i, p_i) \text{ (for each } i) \\
\qquad\qquad\quad \mathcal{P} = \bigcup_i \mathcal{P}_i \\
\qquad\qquad\quad PC_0 = \emptyset \\
\qquad\qquad\quad PC_i = PC_{i-1} \cup \mathcal{WC}_t(\mathcal{P}, c_i, C_i, p_i) \text{ (for each } i) \\
\qquad\qquad\quad (\Phi, PC') = \mathit{FindChecks}(PC_n) \\
\qquad\qquad\quad \varphi = \mathit{SolvePC}(PC') \\
\qquad\qquad \text{in} \;\; (\varphi(\mathcal{P}), \Phi)
\end{array}
$$

$$
\mathit{makeSkeleton}(c, C, p) = \{(c, m_i) = \rho_{i\ (1 \le i \le n)} \mid m_i \in \mathit{Dom}(C),\ \rho_i \text{ fresh}\}
$$

$$
\begin{array}{l}
\mathcal{WC}_t(\mathcal{P}, c, C, p) = \text{let} \;\; \{m_1 = M_1, \cdots, m_n = M_n\} = C^p \\
\qquad\qquad\qquad\qquad\quad PC_0 = \emptyset \\
\qquad\qquad\qquad\qquad\quad PC_i = PC_{i-1} \cup \mathcal{WM}_t(\mathcal{P}, c, m_i, M_i, p_i) \\
\qquad\qquad\qquad\quad \text{in} \;\; PC_n
\end{array}
$$

Fig. 23. Type inference algorithm for a program

$$
\frac{\vdash \Sigma_0 : \mathcal{P}_0 \qquad \mathcal{P}_0^F; \mathcal{P}_0 \vdash \Psi_0 : \mathcal{P}_0^I \qquad \mathcal{P}_1^F = \mathcal{P}_0^F \setminus \mathcal{P}_0^I}{\mathcal{P}_1^F \vdash \Lambda_0 : \mathcal{P}_0 \cup \mathcal{P}_0^I}
$$

$$
\frac{\mathcal{P}_{i-1}^F \vdash \Lambda_{i-1} : \mathcal{P}_{i-1}^E \qquad \mathcal{P}_{i-1}^F; \mathcal{P}_{i-1}^E \vdash \Psi_i : \mathcal{P}_i^I \qquad \mathcal{P}_i^F = \mathcal{P}_{i-1}^F \setminus \mathcal{P}_i^I}{\mathcal{P}_i^F \vdash \Lambda_i : \mathcal{P}_{i-1}^E \cup \mathcal{P}_i^I}
$$

In order to define the relation $\mathcal{P}^F; \mathcal{P}^E \vdash \Psi : \mathcal{P}^I$, we need to make some assumptions on privilege sets Π required by methods that will be loaded in future. In our original type system, the typing rule for the `invoke(`c, m`)` instruction needs to identify the set of all methods that may be invoked at runtime and verify the constraint $\Pi \subseteq \Pi'$ for the type $\Delta \xrightarrow{\Pi} \tau$ of each method in the set. This is impossible when dynamic class loading is supported. To solve this problem, we require that if a dynamically loaded class overrides some method of an existing class, then the overriding method must not use more privileges than the existing one. We refer to

this restriction as the *subclass restriction*.

The typing rules for $\mathcal{P}^F; \mathcal{P}^E \vdash \Psi : \mathcal{P}^I$ are obtained by refining our original typing rules for $\vdash \Sigma : \mathcal{P}$ with the privilege environment $\mathcal{P}^F$ and $\mathcal{P}^E$. Typing judgments for code blocks are refined to have the form $\mathcal{P}^F; \mathcal{P}^E; \mathcal{L} \vdash \Pi; \Delta; p \ \triangleright B \ : \ \tau$. The rules other than that for `invoke` are obtained by simply adding the assumptions $\mathcal{P}^F; \mathcal{P}^E$. The rule for `invoke` is refined to the following two rules.

$$\frac{\Pi; \tau_1 \cdot \Delta; p \ \triangleright B \ : \ \tau}{\Pi; \Delta_0 \cdot c_0 \cdot \Delta; p \ \triangleright \texttt{invoke(c,m)} \cdot B \ : \ \tau} \quad \left(\begin{array}{l} \mathit{mclass}(\Theta, c, m) = c', \\ c' \in \mathit{Dom}(\mathcal{P}^E), \\ \text{if } \ \mathcal{P}^E(c')(m) \subseteq \Pi, \\ \Theta(c')(m) = \Delta_1 \to \tau_1, \\ c_0 <: c, \text{ and } \Delta_0 <: \Delta_1 \end{array} \right)$$

$$\frac{\Pi; \tau_1 \cdot \Delta; p \ \triangleright B \ : \ \tau}{\Pi; \Delta_0 \cdot c_0 \cdot \Delta; p \ \triangleright \texttt{invoke(c,m)} \cdot B \ : \ \tau} \quad \left(\begin{array}{l} \mathit{mclass}(\Theta, c, m) = c', \\ c' \notin \mathit{Dom}(\mathcal{P}^E), \\ \text{if } \ \mathcal{P}^F(c')(m) \subseteq \Pi, \\ \Theta(c')(m) = \Delta_1 \to \tau_1, \\ c_0 <: c, \text{ and } \Delta_0 <: \Delta_1 \end{array} \right)$$

The first rule is for the case in which the closest super class c' of a class c that defines a method m is already loaded. In this case, it checks whether a privilege set Π' of the method m in the class c' specified in $\mathcal{P}^E$ satisfies the constraint $\Pi' \subseteq \Pi$. The second rule is for the case in which the closest super class of a class c that defines a method m is still not loaded. In this case, it checks whether a privilege set Π' of the method m in the class c' specified in $\mathcal{P}^F$ satisfies the constraint $\Pi' \subseteq \Pi$. Under the subclass restriction, these rules are sufficient to guarantee that no access violation occurs at runtime.

8.2 Outline of Type Inference with Dynamic Class Loading

Corresponding to the structure of a program, the top-level algorithm performs the following steps. First, it infers a privilege environment $\mathcal{P}_0$ for the closed base Σ_0 of the program using our original type inference algorithm. The algorithm then iterates the following inference steps for each time a new set Ψ_i is loaded.

(1) It infers $\mathcal{P}^I$ and $\mathcal{P}^F$ for Ψ_i under $\mathcal{P}^E_{i-1}$ using the extended main algorithm $\mathcal{W}$ (described below).

(2) It checks that $\mathcal{P}^I$ *satisfies* $\mathcal{P}^F_{i-1}$ and $\mathcal{P}^E_{i-1}$, i.e., it checks the following property for each $(c, m) \in \mathit{Dom}(\mathcal{P}^I)$. Let $\Pi = \mathcal{P}^I(c, m)$. If $(c, m) \in \mathit{Dom}(\mathcal{P}^F_{i-1})$ then $\Pi \subseteq \mathcal{P}^F_{i-1}(c, m)$. Otherwise $\Pi \subseteq (\mathcal{P}^I \cup \mathcal{P}^E_{i-1})(c', m)$ for the class c' such that $\mathit{mclass}(\Theta, c, m) = c'$.

(3) It assigns $\mathcal{P}^E_i = \mathcal{P}^E_{i-1} \cup \mathcal{P}^I$ and $\mathcal{P}^F_i = (\mathcal{P}^F_{i-1} \setminus \mathcal{P}^I) \cup \mathcal{P}^F$.

The main algorithm $\mathcal{W}$ is extended to infer a typing judgment of the form $\mathcal{P}^F; \mathcal{P}^E \vdash \Psi : \mathcal{P}^I$. To achieve this, in addition to the inference steps in the original $\mathcal{W}$, the algorithm also generates a template C^F for $\mathcal{P}^F$ consisting of entries of the form $(c, m) \mapsto \rho$ and adds a constrain $\rho \sqsubseteq \Pi$ to the constraint set PC for each invocation of method m on a non-loaded class c. In the final step, the algorithm applies the solution substitution of PC to the template C^F to obtain a ground privilege environment $\mathcal{P}^F$.

Although we have not developed the details of the extended type inference algorithm, the required machinery seems to be the same as that for the original $\mathcal{W}$. Therefore, it would not be difficult to develop an extended type inference algorithm based on the strategy outlined above.

9. DISCUSSION

This study is the first step toward a type-based access control system for JVM-style bytecode languages, and a number of limitations and further issues remain to be investigated. In this section, we review some of them and suggest further work.

9.1 Limitations of Static Approximation

Our method is based on inferring static typing of a bytecode program. As we have discussed in Introduction, this static approach has the advantage of detecting possible access violations before the execution of a given code. Its limitation, compared with stack inspection and other dynamic approaches, is that it can only approximate dynamic behavior. As a result, it may reject programs that do not actually cause access violation. Typical cases include the following.

—*Dynamically dispatched methods.* The `invoke`(c, m) instruction of JVM_{sec} performs dynamic method dispatch based on the runtime class of the receiver object. This means that the actual method code invoked by this instruction is determined at runtime among those in all the subclasses of c. This is an essential feature of a JVM like bytecode language that supports object oriented method inheritance, but creates an apparent conflict with static verification of access property of this instruction. Our solution is to regard the privilege set needed for this instruction to be the union of all the privilege sets of methods that may be invoked. Under the assumption that the class hierarchy is statically given, this yields a simple and sound strategy. However, the resulting privilege set might become too coarse approximation when the access privilege sets of possible methods vary. It also makes modular verification difficult due to the requirement that all the privileges of all the methods must be known before typechecking.

—*Method overriding by dynamically loaded classes.* The above strategy become unsound if the system allow dynamically loaded classes to override existing methods with those having larger privilege sets. The solution we adopted is to place the *subclass restriction*, which requires that the privilege set of overriding method should be a subset of the original set. This restriction may sometimes be too strong in practice.

—*Statically unknown objects.* In Section 7, we have outlined a method to include target object specification. However, it is rather weak in that the type system regards a target expression to be an arbitrary object whenever it involves method invocation or primitive operations.

These cases show inherent weakness of static typing, and overcoming these limitations requires some new machinery in addition to standard static typing. As we have mentioned, soft typing and trace based approach have been considered in literature to infer target object specification. Skalka [2005] addresses the problem of dynamically dispatched methods and have proposed a solution based on trace

effects. Extending our static type system with some of these mechanisms is one important future research direction.

In a more practical perspective, one can use our static verification system as a complementary tool to those based on stack inspection and other dynamic approaches. Type based static approach can be regarded as a system to verify *safety property*, i.e. the code is free of access violation. On the other hand, stack inspection and other dynamic systems detect *liveness property*, i.e. the code actually causes access violation. These two aspects are complementary – safety property may sometimes be too approximate and detection of liveness property may be too late. Designing an access control system that combines these two approaches can also be an important future research.

9.2 Adding Other JVM Features

We have developed our access control system for JVM_{sec}, which is a small subset of the Java bytecode language. In order to develop an actual access control system, we need to extend the type system to the full set of the Java bytecode language.

Since the type system is based on the logical presentation of the Java bytecode language [Higuchi and Ohori 2002], we believe that the set of instructions considered there can be added without much difficulty. These include instructions for local variable access and for object field manipulation. Furthermore, the type system developed in [Higuchi and Ohori 2002] supports polymorphic subroutines, whose treatment is orthogonal to the typing mechanism for access control presented here. Therefore, our type system should extend smoothly to include JVM subroutines without the use of any additional machinery.

In Java, `checkPermission` can be used to protect object fields. This feature is also easily added by extending field types to include privilege annotation similar to method types as in

$$\{f_1 : (\Pi_1, \tau_1), \ldots, f_n : (\Pi_n, \tau_n)\}$$

and by defining a typing rule for field manipulation as follows.

$$\frac{\Pi; \tau' \cdot \Delta; p \ \rhd B \ : \ \tau}{\Pi; c_0 \cdot \Delta; p \ \rhd \texttt{getfield}(c, f) \cdot B \ : \ \tau} \quad (\text{if } c_0 <: c,\ \Theta(c)(f) = (\Pi', \tau') \text{ and } \Pi' \subseteq \Pi)$$

The rule for `putfield` can similarly be defined.

9.3 Implementation Issues

We also need to consider a number of implementation issues including the following.

—*Compatibility with existing programs.* The current practice in Java access control is dynamically invoking static methods `checkPermission` and `doPrivileged`, supplied as a JDK security package. A static access control system should work with existing programs using these methods. One of the approaches for this is to replace these two methods with those whose intended effect is represented by their types but whose runtime effect is nil and to consider `doPrivileged` invocation as `priv` instruction.

—*Relationship with JVM runtime system.* In JVM, a bytecode verifier checks the type consistency of a class file. Type-based access control involves static type-checking similar to bytecode verification; therefore it is desirable to unite these

two verification systems. Since our access control system is based on a type theory [Higuchi and Ohori 2002] for bytecode verification, it is not difficult to develop a static system which checks type consistency and access control simultaneously. Since the Java bytecode verifier is closely coupled with the JVM runtime system, development of an integrated system requires us to modify a major part of the JVM runtime system. An alternative strategy is to check all the class files in a program independently of the JVM runtime system before executing the program. By adopting this approach, we should be able to design a verifier as a combination of a stand-alone core verifier that verifies the stable part (including libraries) of a program off-line and a plug-in to a JVM runtime system that performs security verification on-line each time a new class is loaded.

—*Specifying privilege requirements.* We need to declare a privilege set for each native method. A possible approach is to directly write it in a class file that declares a native method. Another approach is to describe it in an external file corresponding to the class file. In this approach, a coding technique, such as digital signature used in the current JDK to sign a code, may be required to guarantee the credibility of the file.

—*Efficiency of type inference.* We must also investigate efficient type inference algorithm. The most time consuming step in our type inference algorithm is *SolvePC* which searches a substitution that satisfies a set of inequality constraints. The current definition is a naive iteration until the set of inequations become solved form. Since this can be regarded as a problem to determine set inclusion, some techniques and results in algorithmic study on set inclusion problems can be used to develop an efficient algorithm for our type inference.

10. CONCLUSIONS

We have developed a static access control system for the Java bytecode language. We have extended our earlier work of presenting the JVM code language as a typed term calculus to incorporate privilege attributes in a method type. We have then defined an operational semantics that simulates JDK style runtime stack inspection and have shown that the type system is sound with respect to the operational semantics. This result guarantees that we can safely omit costly runtime stack inspection. All the possible access violation is statically detected. Another advantage of our approach is that the user can directly verify whether or not a code conforms to a given access policy, without relying on explicit insertion of `checkPermission`. This approach can therefore be used as a security verification system for foreign and possibly malicious code. For this type system, we have developed a type inference algorithm that enables automatic verification for code-level access control. We have also described strategies for incorporating specification of target resources and dynamic class loading into our type system.

With the extensions and further works we have discussed in the previous section, we believe that the presented type-based approach to access control will become a viable alternative to stack inspection.

Acknowledgments

The authors would like to thank the anonymous reviewers for their thorough and careful reading and for their detailed constructive comments.

REFERENCES

Banerjee, A. and Naumann, D. 2001. A simple semantics and static analysis for Java security. CS Report AI-068-85, Stevens Institute of Technology.

Banerjee, A. and Naumann, D. 2002. Representation independence, confinement, and access control. In *Proc. Symposium on Principles of Programming Languages*. ACM Press, New York, 166–177.

Barthe, G. and Dufay, G. 2004. Certified bytecode verification. In *Proc. International Conference on Fundamental Approaches to Software Engineering*. Lecture Notes in Computer Science, vol. 2984. Springer, Berlin, 99–113.

Cartwright, R. and Fagan, M. 1991. Soft typing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 278–292.

Clements, J. and Felleisen, M. 2003. A tail-recursive semantics for stack inspections. In *European Symposium on Programming*. Lecture Notes in Computer Science, vol. 2618. Springer, Berlin, 22–37.

Erlingsson, U. and Shneider, F. 2000. RM enforcement of Java stack inspection. In *Proc. IEEE Symposium on Security and Privacy*. IEEE, New York, 246–255.

Fournet, C. and Gordon, A. 2002. Stack inspection: theory and variants. In *Proc. Symposium on Principles of Programming Languages*. ACM Press, New York, 307–318.

Freund, S. and Mitchell, J. 1999. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems 21,* 6, 1196–1250.

Freund, S. and Mitchell, J. 2003. A type system for the Java bytecode language and verifier. *Journal of Automated Reasoning 30,* 3–4, 271–321.

Gallier, J. and Snyder, W. 1989. Complete sets of transformations for general E-unification. *Theoretical Computer Science 67,* 2, 203–260.

Gong, L. 1999. *Inside Java^{TM} 2 Platform Security*. Addison-Wesley, Reading, Massachusetts.

Gong, L. and Schemers, R. 1998. Implementing protection domains in the Java Development Kit 1.2. In *Internet Society Symposium on Network and Distributed System Security*. The Internet Society, Reston, VA, 125–134.

Higuchi, T. and Ohori, A. 2002. Java bytecode as a typed term calculus. In *Proc. International Conference on Principles and Practice of Declarative Programming*. ACM Press, New York.

Karjoth, G. 2000. An operational semantics of Java 2 access control. In *IEEE Computer Security Foundations Workshop*. IEEE, New York, 224–232.

Klein, C. and Wildmoser, M. 2003. Verified bytecode subroutines. *Journal of Automated Reasoning 30,* 3–4, 363–398.

Koved, L., Pistoia, M., and Kershenbaum, A. 2002. Access rights analysis for Java. In *Proc. Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM, New York.

Leroy, X. 1992. Polymorphic typing of an algorithmic language. Ph.D. thesis, University of Paris VII.

Leroy, X. 2003. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning 30,* 3–4, 235–269.

Liang, S. and Bracha, G. 1998. Dynamic class loading in the Java virtual machine. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM, New York, 36–44.

Lindholm, T. and Yellin, F. 1999. *The Java virtual machine specification*, Second edition ed. Addison Wesley, Reading, Massachusetts.

Nipkow, T. 2003. Java bytecode verification, editorial introduction to the special issue. *Journal of Automated Reasoning 30,* 3-4, 233–233.

Ohori, A. 1999. The logical abstract machine: a Curry-Howard isomorphism for machine code. In *Proceedings of International Symposium on Functional and Logic Programming*. Lecture Notes in Computer Science, vol. 1722. Springer, Berlin, 300–318.

Posegga, J. and Vogt, H. 1998. Byte code verification for Java smart card based on model checking. In *Proc. European Symposium on Research in Computer Security*. Lecture Notes in Computer Science, vol. 1485. Springer, Berlin, 175–190.

Pottier, F., Skalka, C., and Smith, S. 2005. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems 27,* 2, 344–382.

Skalka, C. 2005. Trace effects and object orientation. In *Proc. ACM Conference on Principles and Practice of Declarative Programming*. ACM, New York, 139–150.

Skalka, C. and Smith, S. 2004. History effects and verification. In *Proc. Asian Symposium on Programming Languages and Systems*. Lecture Notes in Computer Science. Springer, Berlin, 107–128.

Skalka, S. and Smith, S. 2000. Static enforcement of security with types. In *Proc. International Conference on Functional Programming*. ACM Press, New York, 34–45.

Stata, R. and Abadi, M. 1998. A type system for Java bytecode subroutines. In *Proc. Symposium on Principles of Programming Languages*. ACM Press, New York, 149–160.

Wallach, D. 1998. Understanding Java stack inspection. In *Proc. IEEE Symposium on Security and Privacy*. IEEE, New York, 52–63.

Wallach, D., Appel, A., and Felten, E. 2000. Safkasi:a security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology 9*, 341–378.