A Static Type System for JVM Access Control*†

Tomoyuki Higuchi
School of Information Science
Japan Advanced Institute of
Science and Technology
Tatsunokuchi Ishikawa, 923-1292 Japan
thiguchi@jaist.ac.jp

Atsushi Ohori
School of Information Science
Japan Advanced Institute of
Science and Technology
Tatsunokuchi Ishikawa, 923-1292 Japan
ohori@jaist.ac.jp

Abstract

This paper presents a static type system for JAVA Virtual Machine (JVM) code that enforces an access control mechanism similar to the one found, for example, in a JAVA implementation. In addition to verifying type consistency of a given JVM code, the type system statically verifies that 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, similarly to JAVA stack inspection. This result ensures that "well typed code cannot violate access policy." The paper then develops a type inference algorithm and shows that it is sound with respect to the type system and that it always infers a minimal set of access privileges. 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

*The authors was partially supported by Grant-in-aid for scientific research on priority area "informatics" A01-08, grant no:15017239. The second author was also partially supported by Grant-in-aid for scientific research (B), grant no:15300006.

†**This is the authors' version of the paper to be presented at ACM ICFP Conference, August, 2003.**

ICFP'03, August 25–29, 2003, Uppsala, Sweden.

Keywords

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 in an operating system. This simple strategy has been based on the assumption that the semantics of a program code the user executes is transparent to the user, and therefore resource access requests issued by the code reflect the user's intention. This assumption no longer holds in the recently emerging network computing environment, where a program code to be executed is dynamically composed from various pieces downloaded from not necessarily trusted foreign sites. To deal with this situation, we need to develop *code-level access control*, where an unauthorized principal is not some other user but some untrusted piece of code, and whether a principal has some access privilege or not is a property of some portion of the code. This requires us to develop a verification system for the property of low-level code.

This problem has recently attracted attentions of the researchers and developers, and several verification systems have been proposed and developed. Among them, the most notable one is perhaps the JAVA access control system [8] (implemented in JDK1.2 and later.) In this system, each class (consisting of a set of methods) is owned by some principal, and each principal is assigned a set of privileges that is granted to the principal. In order to enforce access control, the implementer of the code explicitly inserts a call of special static method `checkPermission` before accessing protected resources. This static method checks that the principal associated to the current calling code has the required privilege under the current execution environment. Since method calls are in general nested, and the access requests issued by some method should be regarded as those of calling methods, it traverses the current frame stack to ensure that all the calling methods have the required access privileges. This process is known as *stack inspection*. As we shall review later, the JAVA access control system also provides a mechanism for trusted code to gain privileges even when the calling method does not have these privileges.

Various formal properties of this approach have been studied and efficient implementation methods have been proposed. Karjoth [7] has presented a formal operational semantics as a transition relation on abstract machine states. Wallach, Appel and Felten [16, 15] and Banerjee and Naumann [2] have provided denotational and logical

account, respectively. Fournet and Gordon [3] have studied semantic properties of a program performing stack inspection. With these efforts, this JAVA-style stack inspection approach has become one of most important code-level access control methods.

There are however some weaknesses that are inherent in this approach. One is its high runtime overhead due to dynamic inspection of the entire call stack. This method also makes some program optimization difficult or impossible to apply due to the need of maintaining a call stack. To reduce the run-time overhead, Wallach, Appel and Felten [16, 15] have analyzed the stack inspection semantics, and proposed an alternative method, called a *security-passing style* approach. Instead of inspecting the entire call stack, this method explicitly passes security information as extra parameters so that the called method can verify access privileges. This could be a potentially more efficient alternative to stack inspection. Since this method does not assume any runtime architecture, it would also be more amenable to various optimization. Erlingsson and Shneider [10] have reported implementation experiences including one for security passing style. Despite these advantages, it still incurs non-trivial runtime overhead for passing extra security information, which could potentially be large.

Another and perhaps more serious weakness of the stack inspection approach is that the programmer must explicitly call `checkPermission` method before accessing protected resources. The apparent problem of this strategy is that some programmer might fail to insert all the appropriate calls, or malicious one would intentionally left out some of the necessary calls. The security-passing approach [15] also share this weakness.

As we have noted above, code-level access control is a verification system for security property of code. As such, it should ideally be a proof system that verifies the desired properties of code statically from the given code itself without relying on practice of the programmer. Based on this insight, Skalka and Smith [13] have developed a static type system of code-level access control for a variant of the lambda calculus by refining the static 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, Skalka, and Smith [4] further refined its type system. Banerjee and Naumann [1] have defined a denotational semantics for a language similar to the calculus considered in [13, 4], which provides another assurance of the safety of this type-based approach. However, since the target language is the lambda calculus and their development relies on properties of a static type system of the lambda calculus, it is not clear whether or not this approach can be applicable to low-level code languages including the JVM bytecode language [12].

The goal of the present work is to establish a static type system for code-level access control of the JVM bytecode language, and show that it is correct with respect to a standard model of JVM execution. We base our development on our earlier work [5], where we have shown that the JVM language can be regarded as a typed term calculus based on a proof theory for low-level code [6]. In this formalism, the type system represents static semantics of a code language as a type system of the lambda calculus does. This property allows us to transfer the concepts and methods developed for the lambda calculus in [13] to the JVM bytecode language. We define a type system with access control information for the JVM language, and establish its soundness with respect to an operational semantics which closely models the JAVA access control using stack inspection. For this type system, we develop a type inference algorithm. These results allow us to develop a static access control system that automatically detects all the possible access violation statically from a given code, and infers the minimal set of privileges required to execute the code. Since the type system examines each resource access directly through method invocation instructions, explicit insertion of `checkPermission` is unnecessary. This property makes our system more reliable than those based on explicit library calls by the programmer, and it can also be used for verifying un-trustworthy and potentially malicious code. A prototype type inference algorithm has been implemented, demonstrating the feasibility of our 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 and shows that the type system is sound with respect to the semantics. Section 5 develops a type inference algorithm and shows its soundness and other desirable properties. Section 6 discusses several extensions and implementation. Section 7 concludes the paper.

2 Our Approach

In order to develop a static access control system for a low-level code language like the JVM bytecode language, we need to establish a type theoretical framework for low-level code. We also need to devise a strategy to extract resource access information from a given code. Before going to the technical details, in this section, we outline our approach focusing on these two points.

In the framework of Skalka and Smith [13], a function has a type of the form

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

indicating the fact that it takes an argument of type τ_1, and computes a result of type τ_2 *using the privileges* Π. It is not immediately obvious that this idea can be applied to type systems of JVM bytecode such as the one by Stata and Abadi [14] that only checks consistency of machine states and instructions. To apply the general idea exploited in [13] to a low-level language, we need to define a type system that deduces a static semantics of a given code.

In an earlier work [5], we have developed a type theoretical framework that allows us to regard the JVM bytecode language as a typed term calculus. In this formalism, 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 the form Δ. (For the simplicity of presentation, we ignore local variable environments, which is irrelevant to the approach presented here.) The type system is constructed based on a proof-theoretical interpretation of low-level machine instructions [6], which we outline below.

Each 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. `return` statement corresponds to an initial sequent (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 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 existing blocks (i,e. the end sequent of existing proofs.)

Our strategy is to refine this formalism to introduce access control information. We interpret a code block B as a proof of a judgment of the form

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

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

To define a type system to deduce a sequent of the above form, we must prescribe the relationship between the privilege sets Π and the block B according to its behavior with respect to resource access. Our strategy is to associate each method with a set of privileges Π' required to execute it, and consider Π in the sequent as a constraint $\Pi' \subseteq \Pi$ to invoke the method. A method has then a type of the form $\Delta \xrightarrow{\Pi} \tau$ similar to function type in [13]. In order to properly grant a code to gain some privilege 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 is constructed in such a way that it type-checks a class with respect to an access policy and a given class environment describing existing classes. Each method in an existing class is assumed to be either one that has already been checked by this type system so that it has a correct privilege information, or a trusted method whose privilege information is declared correctly. After type-checking a class under a given class environment, the type system produces a correct type for each new method, so that the system can extend the class environment with the new class.

Since the type system checks all the method invocation statically, verification of conformance of access privileges against a given access policy is thorough and complete. This mechanism frees the programmer from the responsibility for 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 A 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 Java bytecode containing minimal features sufficient to present our method.

In the subsequent development, we shall use the following notations. For a sequence S and an element e, $e{\cdot}S$ is the sequence obtained by adding e at the front of S, and $S.i$ denotes the ith element of S. $S\{i \leftarrow e\}$ is the sequence obtained from S by updating ith element of S to e. $|S|$ is the length of the sequence S. We use similar notations for a finite function f: $f.x$ is the value assigned to x in f, and $f\{x \leftarrow v\}$ is the function f such that $Dom(f') = Dom(f) \cup \{x\}$, $f'(x) = v$ and $f'(y) = f(y)$ for any $y \neq x$.

3.1 The language syntax

We assume that the user sets up an access policy (ranged over by $\mathcal{A}$) statically and globally. It is a function assigning a set of *privileges* (ranged over by π) to each principal. The syntax of access policies is given below.

$$\begin{array}{rcl} \mathcal{A} & := & \{p_1 = \Pi_1, \cdots, p_n = \Pi_n\} \\ \Pi & := & \emptyset \mid \{\pi\} \cup \Pi \\ \pi & := & \mathit{FileRead} \mid \mathit{FileWrite} \mid \mathit{SocketConnect} \mid \cdots \end{array}$$

A privilege π is an atom (constant) modeling some system resource to be protected, and corresponds to a *permission* in JAVA. A permission of JAVA consists of a target and an action. Later in Section 6, we shall discuss a mechanism to extend our formalism to include target specification in π.

For simplicity, we assume that a unit of security verification is one class under a given access policy and a given class environment. An actual JVM program may contain a set of mutually dependent classes. It is routine to extend our framework to allow mutually dependent classes as a unit of verification.

A *class* consists of a set of methods. The syntax of classes is given below.

$$\begin{array}{rcl} C & := & \{m = M, \ldots, m = M\} \\ M & := & \{l_1 : B_1, \cdots, l_n : B_n\} \quad (\mathit{entry} \in \{l_1, \cdots, l_n\}) \\ B & := & \texttt{goto}(l) \mid \texttt{return} \mid I{\cdot}B \\ I & := & \texttt{acc}(n) \mid \texttt{iconst}(n) \mid \texttt{dup} \mid \texttt{ifeq}(l) \\ & \mid & \texttt{priv}(\pi) \mid \texttt{new}(c) \mid \texttt{invoke}(c, m) \end{array}$$

Each class is associated with its principal. We write C^p for a class whose principal is p. Each method M consists of a set of labeled code blocks. We assume that the set of labels of a method contains a special label *entry* to indicate the entry point of the method. Each method M in a class is implicitly owned by the principal associated to the class to which it belongs. We write M^p if M belongs to a class C^p. A code block B is a sequence of instructions terminated with `return` or `goto`. `acc(`n`)` pushes the nth value of a stack on top of the stack. This is added to make the set of instruction non trivial, and does not have much significance to our type system. `priv` corresponds to `doPrivileged` in the JAVA access control architecture. If the block is of the form `priv(`π`)`$\cdot B$ and the access policy allows current principal to use π, then the current privilege set is extended with π so that B can use it. We only consider methods and ignore object fields. A field can be regarded as a special form of a method, and our access control mechanism for method carries over to fields.

In Java, a class file contains explicit type declarations of the methods. We represent these type declaration by *class specification* Θ whose syntax is given blow.

$$\begin{array}{rcl} \Theta & := & \{c_1 = \mathit{methods}, \cdots, c_n = \mathit{methods}\} \\ \mathit{methods} & := & \{m_1 = \Delta_1 \rightarrow \tau_1, \cdots, m_n = \Delta_n \rightarrow \tau_n\} \end{array}$$

This is an extra input to the type system in type-checking a class, and not a static environment for typing derivation.

A static environment used in the type system is a *static class environment* (ranged over by $\mathcal{C}$) describing the static information about (already verified) set of existing classes. It describes for each class its method types (augmented with privileges). The syntax of class environments is defined below.

$$\begin{array}{rcl} \mathcal{C} & := & \{c_1 = \mathcal{M}_1, \cdots, c_n = \mathcal{M}_n\} \\ \mathcal{M} & := & \{m_1 = \Delta_1 \xrightarrow{\Pi_1} \tau_1, \cdots, m_n = \Delta_n \xrightarrow{\Pi_n} \tau_n\} \\ \Delta & := & \emptyset \mid \tau{\cdot}\Delta \\ \tau & := & \texttt{int} \mid c \end{array}$$

Δ is a stack type which is a sequence of types, and the leftmost element corresponds to the stack top.

3.2 The type system

Corresponding to the structure of JVM$_{sec}$, the type system consists of typing relations for code blocks, methods, and classes. We define them in that 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 given class environment $\mathcal{C}$ and an access policy $\mathcal{A}$. $\mathcal{C}$ is of the form $\mathcal{C}_0 \cup \{c = \mathcal{M}\}$, where $\mathcal{C}_0$ is the class environment describing all the existing classes and $\{c = \mathcal{M}\}$ describes the types of methods of the current class. This structure reflects the mutually recursive nature of the set of methods of a class definition. In addition to those two environments, we need to introduce a label environment $\mathcal{L}$ to describe the types of the set of code blocks to which B belongs. This is necessary due to mutually recursive block definitions in a method induced by label references. A label environment has the form $\{l_1 : \Pi_1, \Delta_1, p \rhd \tau, \cdots, l_n : \Pi_n, \Delta_n, p \rhd \tau\}$ where $l_1, \cdots, l_n$ is the set of labels of the blocks. Since bindings of labels and classes are static, there is no rule that changes $\mathcal{C}$ or $\mathcal{L}$. For this reason, in defining each typing rule, we treat $\mathcal{C}$ and $\mathcal{L}$ as global variables. The set of typing rules is given in Figure 1.

The rules for `invoke` and `priv` realize static access control. The other rules are essentially the same as those in [5]. To invoke a method of type $\tau \xrightarrow{\Pi} \tau$, a caller must have at least all the privileges in Π. For a simple type discipline such as that of the lambda calculus, this would be sufficient. 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 therefore be checked against all the possible methods that may be called. The condition "$\Pi_i \subseteq \Pi$ for each i" in the rule for `invoke` guarantees this constraint for each possible method. The auxiliary function `lookupAll(c,m)` traverses the subclass of class c and returns the set of classes that define a method m. The rule for `priv(`π`)` adds π to the current privilege set for B, if the principal of B has access privilege π under the access policy $\mathcal{A}$.

Method typing. A method consists of a set of code blocks. A method M is well typed if each block in M is well typed. This relation is defined below.

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

$\Pi \subseteq \mathcal{A}(p)$ reflects the property that the privileges each code block may use must not surpass the set of allowable 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.

$\mathcal{C} \vdash M^p : \Delta \xrightarrow{\Pi} \tau$
$\iff$ there exists some $\mathcal{L}$ such that $\mathcal{C} \vdash M^p : \mathcal{L}$
and $\mathcal{C}, \mathcal{L} \vdash \Pi, \Delta, p \rhd M^p.entry : \tau$

Class typing. Using these definition, typing of a class C^p with respect to a class environment $\mathcal{C}_0$ of existing classes is defined as follows.

$\mathcal{C}_0, \Theta \vdash C^p : \mathcal{M}$
$\iff$ for each $m \in Dom(C^p)$, following conditions hold:
$\mathcal{C} = \mathcal{C}_0 \cup \{c = \mathcal{M}\}$,
$\mathcal{C} \vdash C^p.c.m : \mathcal{M}.c.m$,
$\mathcal{M}.c.m = \Delta \xrightarrow{\Pi} \tau$ and $\Theta.c.m = \Delta \to \tau$

A well typed class C^p named c with type $\mathcal{M}$ under $\mathcal{C}_0$ yields an extended class environment $\mathcal{C}_0 \cup \{c = \mathcal{M}\}$.

3.3 Example of program and its typing

We show an example of type derivation using a simple program. Let `IO` be an existing system class having privilege `FRead` for reading files, and `readFile` be a native method defined in the class, which takes a file name and returns a content of the file as a string. Suppose the type of `readFile` is declared as `str` $\xrightarrow{\{\texttt{FRead}\}}$ `str`. This type indicates that privilege `FRead` is required to read files through this method. This condition is statically enforced by the typing rule for method invocation.

To enable any user without `FRead` to read a public file, a trusted user with `FRead` can define the following class using `priv`.

```
class safeClass {
      readFooFile() : str -> str = {
        new(IO)
        sconst("/public/foofile")
        priv(FRead)
        invoke(IO,readFile)
        return
      }
}
```

`sconst(`s`)` is an instruction which pushes a string s onto the stack and has a typing rule similar to `iconst(`n`)`. Any user should be able to call `readFooFile`, even if he or she doesn't have `FRead` privilege.

This is achieved by our type system. Let *trusted* be the principal of the class `safeClass` such that `FRead` $\in \mathcal{A}$(*trusted*). The type system deduce the following typing for the body of method `readFooFile`.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\texttt{FRead}\}, \texttt{str}\cdot\emptyset, trusted \rhd \texttt{return} : \texttt{str}}}{\{\texttt{FRead}\}, \texttt{str}\cdot IO\cdot\emptyset, trusted \rhd \texttt{invoke(IO,read)} : \texttt{str}}}{\emptyset, \texttt{str}\cdot IO\cdot\emptyset, trusted \rhd \texttt{priv(Read)} : \texttt{str}}}{\emptyset, IO\cdot\emptyset, trusted \rhd \texttt{sconst(``/public/foofile'')} : \texttt{str}}}{\emptyset, \emptyset, trusted \rhd \texttt{new(IO)} : \texttt{str}}$$

From this type derivation, `readFooFile` is given type `void` $\xrightarrow{\emptyset}$ `str`. This means that no privilege is required to invoke it.

4 Operational Semantics and Type Soundness

In order to show that the type system just defined properly enforces the desired access control, we define an operational semantics that models Java-style dynamic stack inspection, and show the soundness of the type system with respect to the operational semantics.

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

$$\frac{\Pi, (\Delta.n)\cdot\Delta, p \rhd B : \tau}{\Pi, \Delta, p \rhd \texttt{acc}(n)\cdot B : \tau} \qquad \frac{\Pi, \texttt{int}\cdot\Delta, p \rhd B : \tau}{\Pi, \Delta, p \rhd \texttt{iconst}(n)\cdot B : \tau} \qquad \frac{\Pi, \tau\cdot\tau\cdot\Delta, p \rhd B : \tau}{\Pi, \tau\cdot\Delta, p \rhd \texttt{dup}\cdot B : \tau}$$

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

$$\frac{\Pi, \tau_1\cdot\Delta, p \rhd B : \tau \quad \{c_1, \cdots, c_n\} = \texttt{lookupAll}(c, m)}{\Pi, \Delta_0\cdot c_0\cdot\Delta, p \rhd \texttt{invoke}(c, m)\cdot B : \tau} \quad (\text{if } \Delta_1 \xrightarrow{\Pi_i} \tau_1 = \mathcal{C}.c_i.m\,,\, \Delta_0 <: \Delta_1\,,\, c_0 <: c \text{ and } \Pi_i \subseteq \Pi \text{ for each } i)$$

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

Figure 1. Typing rules for code block B

4.1 Operational semantics

Since semantics of a method in a class depends on other methods of the class and those of existing classes, the operational semantics is defined relative to a *dynamic class environment* (ranged over by Ω) of the form:

$$\Omega = \Omega_0 \cup \{c = C^p\}$$

where Ω_0 is a dynamic class environment for existing classes whose static information is described in a static class environment $\mathcal{C}_0$ used in the type system, and c is the current class.

Since $\mathcal{C}_0$ is a class environment for already verified classes, the operational semantics for a class is defined under the assumption that method bodies in Ω_0 satisfies the static constraints described in $\mathcal{C}_0$. To model this situation, we assume that each method in Ω_0 acts as an opaque *function* satisfying the corresponding typing constraint in $\mathcal{C}_0$, whose precise condition will be given when we prove the type soundness theorem. To emphasize the distinction between the method bodies in the current class (for which the semantics is being defined) and those in Ω_0 modeled by opaque functions in existing classes, we write $\Omega.c.m = \texttt{PreChecked}(f)$ if m is a method in an existing class, and write $\Omega.c.m = \texttt{Code}(M)$ if m is a method in the current class.

The operational semantics is defined by specifying for each instruction I, its effect as a transition rule on machine states using a dynamic class environment Ω, a static class environment $\mathcal{C}_0$ for existing classes, and a class specification Θ describing the types of the methods. Θ is necessary to check the type of the current method.

A machine state has the form:

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

P is a dynamic privilege set, which represents the set of privileges that the block B can currently use. S is an operand stack, which is a sequence of runtime value (ranged over by v). Following [13], we model access violation by a special runtime value `secfail`. If an access violation occurs at runtime, the machine terminates with this special value. Introduction of this special value is necessary to distinguish it from type error, which causes the machine to stop prematurely. An ordinary runtime value is either an integer n or a heap address r. $M^p\{B\}$ indicates that the machine is executing the first instruction of block B belonging to method M^p. h is a heap, which is a function from heap address (ranged over by r) to object instances of the form $\langle\rangle_c$ of class c. Since we omitted object fields, the contents of an object instance is empty. D is a *dump*, which is a sequence of saved method frames of the form $(P, S, M^p\{B\})$.

A state transition rule is of the form

$$\mathcal{C}_0, \Omega, \Theta \vdash (P, S, M^p\{I\cdot B\}, D), h \longrightarrow (P', S', M'^{p'}\{B'\}, D'), h'$$

indicating that I transforms the machine state $(P, S, M^p\{I\cdot B\}, D), h$ to $(P', S', M'^{p'}\{B'\}, D'), h'$ under $\mathcal{C}_0, \Omega, \Theta$. The set of transition rules is given in Figure 2 (omitting the context $\mathcal{C}_0, \Omega, \Theta$ which are not changed during the execution.) The function $\texttt{lookup}(c, m)$ used in the rule for `invoke` locates the class which is the closest super class of c that defines a method m. Instruction `new` creates an initialized new object. The only instruction which may causes the security error is `invoke`. There are three cases for `invoke`. If the method to be invoked is defined in the current class, then the method body is called just as in the conventional JVM. If the method to be invoked is one defined in an existing class and the set of privileges required by the method is included in the dynamic privilege set, then the method must succeeds with a value of appropriate type. If the method to be invoked is defined in an existing class and the set of privileges required by the method is not included in the dynamic privilege set, then the method invocation is aborted and the system returns the special value `secfail`. This is the case of runtime access violation. The type soundness theorem we shall show later guarantees that when the current privileges include the statically deduced privileges of the method, then access violation will not happen.

The 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 done by checking the current frame without traversing the entire frame stack. Since a method call occur frequently, eager semantics may incur high runtime overhead due to computation of P. For this reason, most implementations including JDK [8] adopt lazy semantics, where calculation of the effective privilege set is performed by stack inspection only when 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 irrelevant for us. The operational semantics is defined only to show soundness of the type system. The type soundness theorem 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 seman-

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

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

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

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

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

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

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

$$(P, S, M^p\{\texttt{new}(c){\cdot}B\}, D), h \longrightarrow (P, r{\cdot}S, M^p\{B\}, D), h' \quad \text{if } h' = h\{r \leftarrow \langle\rangle_c\} \text{ and } r \notin dom(h)$$

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

$$(P, S_1{\cdot}r{\cdot}S, M^p\{\texttt{invoke}(c,m){\cdot}B\}, D), h \longrightarrow (P', S_1{\cdot}\emptyset, M'^{p'}\{M'.entry\}, (P, S, M^p\{B\}){\cdot}D), h$$
$$\text{if } h(r) = \langle\rangle_{c_0},\ c_1 = \texttt{lookup}(c_0, m),\ \Omega.c_1.m = \texttt{Code}(M'^{p'}),\ \Theta.c_1.m = \Delta \to \tau,\ |S_1| = |\Delta| \text{ and } P' = P \cap \mathcal{A}(p')$$

$$(P, S_1{\cdot}r{\cdot}S, M^p\{\texttt{invoke}(c,m){\cdot}B\}, D), h \longrightarrow (P, v{\cdot}S, M^p\{B\}, D), h'$$
$$\text{if } h(r) = \langle\rangle_{c_0},\ c_1 = \texttt{lookup}(c_0, m),\ \Omega.c_1.m = \texttt{PreChecked}(f),\ \mathcal{C}_0.c_1.m = \Delta \xrightarrow{\Pi'} \tau,\ |S_1| = |\Delta|,\ \Pi' \subseteq P, \text{ and } (v, h') = f(S_1, h)$$

$$(P, S_1{\cdot}r{\cdot}S, M^p\{\texttt{invoke}(c,m){\cdot}B\}, D), h \longrightarrow (\emptyset, \texttt{secfail}, \emptyset, \emptyset), h$$
$$\text{if } h(r) = \langle\rangle_{c_0},\ c_1 = \texttt{lookup}(c_0, m),\ \Omega.c_1.m = \texttt{PreChecked}(f),\ \mathcal{C}_0.c_1.m = \Delta \xrightarrow{\Pi'} \tau,\ |S_1| = |\Delta| \text{ and } \Pi' \not\subseteq P$$

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

Figure 2. Transition rules for instructions

tics are equivalent [1, 3], we choose an eager one, which yields a simpler proof of the soundness theorem.

We define a JVM$_{sec}$ *program* to be a top-level invocation of a method of the current class by the user. To execute a method M^p with arguments S at the top level by the user identified by the principal $p_\top$, the machine state is initialized as follows:

$$(P_\top, S, M^p\{M.entry\}, \emptyset), h$$

where $P_\top = \mathcal{A}(p) \cap \mathcal{A}(p_\top)$. We write $\xrightarrow{*}$ for the reflective transitive closure of the relation $\rightarrow$. We define the top level evaluation relation $p_\top, \mathcal{C}_0, \Theta, \Omega \vdash M^p \Downarrow v$ as

$$p_\top, \mathcal{C}_0, \Theta, \Omega \vdash M^p \Downarrow v \iff \mathcal{C}_0, \Theta, \Omega \vdash (P_\top, S, M^p\{M.entry\}, \emptyset), h \xrightarrow{*} (\emptyset, v, \emptyset, \emptyset), h'$$

indicating the fact that method M^p is executed by the user of principal $p_\top$ and returns the value v.

$$\models h : H \iff Dom(h) = Dom(H) \text{ and } \forall r \in Dom(h) \text{ if } h(r) = \langle\rangle_c \text{ then } c <: H(r)$$

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

$$H \models r : \tau \quad (\text{if } H(r) <: \tau)$$

$$H \models S : \Delta \iff Dom(S) = Dom(\Delta) \text{ and } H \models S.i : \Delta.i \text{ for each } i.$$

$$H, \mathcal{C} \models \emptyset : \tau \quad (\text{for any } \tau)$$

$$H, \mathcal{C} \models (P, S, M^p\{B\}){\cdot}D : \tau \iff \exists \mathcal{L}, \exists \Delta, \exists \Pi, \exists \tau' \text{ such that } \mathcal{C} \vdash M^p : \mathcal{L},\ H \models S : \Delta,\ \Pi \subseteq P,\ \mathcal{C}, \mathcal{L} \vdash \Pi, \tau{\cdot}\Delta, p \rhd B : \tau',\ \text{and } H, \mathcal{C} \models D : \tau'$$

Figure 3. Typing of runtime values

4.2 Type soundness

In order to prove the soundness theorem with respect to the operational semantics defined above, we first define typing relations for runtime structures consisting of the following:

- $\models h : H$ (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)
- $H, \mathcal{C} \models D : \tau$ (dump D has type τ under H and $\mathcal{C}$)

Heap type H is a mapping from heap addresses to types. This is similar to store type [11] and is needed for the soundness theorem we shall establish below to scale to heaps containing cyclic structures [5]. The last relation for dump D means that D accepts a value of τ and resumes the saved computation. Figure 3 shows the definitions of these relations. Using these definitions, we define type correctness of a machine state as follows.

$$H, \mathcal{C} \models (P, S, M^p\{B\}, D), h : \tau \iff \exists \mathcal{L}, \exists \Delta, \exists \Pi \text{ such that } \mathcal{C} \vdash M^p : \mathcal{L},\ H \models S : \Delta,\ \Pi \subseteq P,\ \mathcal{C}, \mathcal{L} \vdash \Pi, \Delta, p \rhd B : \tau', \text{ and } H, \mathcal{C} \models D : \tau'$$

This definition says that for a machine state to be type correct, each component must be well typed and the set Π of privileges statically deduced by the type system must be contained in the set P of privileges the block has at runtime. We also define the relation $\vdash \Omega_0 : \mathcal{C}_0$ denoting the fact that dynamic class environment Ω_0 of existing classes satisfies static class environment $\mathcal{C}_0$ of existing classes as follows.

$\vdash \Omega : \mathcal{C} \iff$
for any c, m, the following conditions hold.
Let $\texttt{PreChecked}(f) = \Omega.c.m$ and $\Delta \xrightarrow{\Pi} \tau = \mathcal{C}.c.m$. For any S, h, if there is some H such that $\models h : H$ and $H \models S : \Delta$ then the application $f(S, h)$ computes (h', v) such that $\models h' : H'$ and $H' \models v : \tau$ for some extension H' of H using only the privileges in Π.

Let $\mathcal{C}_0, \Omega_0, \Theta$ be a given static class environment, a given dynamic class environment, and a given class specification satisfying $\vdash \Omega_0 : \mathcal{C}_0$. Also let C^p be a given class named c such that there is some $\mathcal{M}$ satisfying $\mathcal{C}_0, \Theta \vdash C^p : \mathcal{M}$.

We can now prove the following. Let $\mathcal{C} = \mathcal{C}_0 \cup \{c = \mathcal{M}\}$, $\Omega = \Omega_0 \cup \{c = C^p\}$.

THEOREM 1 (TYPE SOUNDNESS). *For any method M^p in C^p, if $H, \mathcal{C} \vdash (P, S, M^p\{B\}, D), h$ then either (1) $B = \texttt{return}$ and $D = \emptyset$ or (2) there are some P', S', M', p', B', D', h', H' such that*

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

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

PROOF. This is proved by the case analysis of the first instruction of B, using the following simple lemma. □

LEMMA 1. *If $H \models v : \tau$ and H' is an extension of H then $H' \models v : \tau$.*

The theorem implies the following desired property.

COROLLARY 1. *Let Θ be a given class specification; $\mathcal{C}_0, \Omega_0$ be a static class environment and a dynamic class environment of existing classes satisfying $\vdash \Omega_0 : \mathcal{C}_0$; let C^p be a class such that $\mathcal{C}_0, \Theta \vdash C^p : \mathcal{M}$. Also let $p_\top$ be the principal of the user. If $\Pi \subseteq \mathcal{A}(p_\top)$ for any Π such that $\mathcal{M}.m = \Delta \xrightarrow{\Pi} \tau$ then if $p_\top, \mathcal{C}_0, \Theta, \Omega \vdash M^p \Downarrow v$ then v is not* `secfail`.

This is a direct consequence of the definition of top-level execution and our type soundness theorem.

This result says that a well typed program will never cause security violation when executed with the privileges specified in its type. We can therefore safely use a type-checked code without monitoring its resource access at runtime.

5 Type Inference

In JVM, each method is explicitly typed but the privilege set Π is not given. In order to use a type system defined in Section 3 we need to develop a type inference algorithm.

5.1 The type inference algorithm

In order to define a type inference algorithm, we extend types and stack types by introducing *type variables* (ranged over by t) and *sequence variables* (ranged over by δ) respectively as follows.

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

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 ranges only over subclasses of c, and $\mathcal{K}(t) = *$ indicates that t has no bound. We write $\mathcal{K} \vdash \tau <: c$ if τ is a subclass of c under bound environment $\mathcal{K}$. This relation is given as follows.

$$\begin{array}{lll} \mathcal{K} & \vdash & c' <: c \quad (\text{if } c' <: c) \\ \mathcal{K} & \vdash & t <: c \quad (\text{if } \mathcal{K}(t) <: c) \end{array}$$

A (type variable) *substitution* (ranged over by S) is a function from a finite subset of type variables to types. We say that a substitution $\mathcal{S}$ *respects* $\mathcal{K}$ if for all t in $Dom(\mathcal{K})$, $\mathcal{K} \vdash \mathcal{S}(t) <: \mathcal{K}(t)$.

$$(E \cup \{(\tau, \tau)\}, S) \Longrightarrow (E, S)$$
$$(E \cup \{(t, \tau)\}, S) \Longrightarrow ([\tau/t]E, \{(t, \tau)\} \cup [\tau/t]S) \quad (\text{if } \mathcal{K}(t) = *)$$
$$(E \cup \{(t, c)\}, S) \Longrightarrow ([c/t]E, \{(t, c)\} \cup [c/t]S) \quad (\text{if } c <: \mathcal{K}(t))$$
$$(E \cup \{(t_1, t_2)\}, S) \Longrightarrow ([t_2/t_1]E, \{(t_1, t_2)\} \cup [t_2/t_1]S) \quad (\text{if } \mathcal{K}(t_2) <: \mathcal{K}(t_1))$$
$$(E \cup \{(t_1, t_2)\}, S) \Longrightarrow ([t_1/t_2]E, \{(t_2, t_1)\} \cup [t_1/t_2]S) \quad (\text{if } \mathcal{K}(t_1) <: \mathcal{K}(t_2))$$

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

Figure 4. Unification algorithm

A unification algorithm Unify accepts a bound environment $\mathcal{K}$ and a set E of pairs of types, and returns substitution $\mathcal{S}$ that respects $\mathcal{K}$. Algorithm Unify is given in Figure 4. It is easily checked that Unify is a unification algorithm if the subclass relation $<:$ has the property that if two classes are incomparable then they have no common subclass. The set of JVM classes satisfies this property. However, if we extend JAVA-style interfaces, then a more refined algorithm will become necessary.

In what follows, we identify S with its homomorphic extension to any syntactic structures containing type variables and sequence variables.

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

$$\Pi := P \mid P \cdot \rho$$

P is a (closed) set of privileges. $P \cdot \rho$ is an open set of privileges denoting the set consisting of privileges in P and those in the set denoted by set variable ρ. We sometimes write $P \cup \Pi$ to denote the element of this language, i.e., if $\Pi = P'$ then $P \cup \Pi = P \cup P'$ and if $\Pi = P' \cdot \rho$ then $P \cup \Pi = (P \cup P') \cdot \rho$. A *set variable substitution* (ranged over by φ) is a function from a finite set of set variables to sets.

The role of the type inference algorithm is twofold. It infers a most general typing using Unify for each method body and to verify that it satisfies the type specification. It also infers the minimal set of privileges required to execute each method. To perform the latter, the algorithm generates a set of *inclusion constraints* of the form $\Pi \sqsubseteq \Pi'$, and then solves the constraint sets to compute the minimal set of privileges. We use PC as a meta variable for a finite set of inclusion constraints. We say that a set variable substitution φ *satisfies* PC if $\varphi(\Pi) \subseteq \varphi(\Pi')$ for any $\Pi \sqsubseteq \Pi' \in PC$.

Figure 5 shows the main algorithm $\mathcal{WC}$ performing these steps using a sub-algorithm $\mathcal{WM}$ for inferring method typing. The main algorithm takes a static class environment $\mathcal{C}$ for existing classes, class definition C^p, and class name c, and infers the set of method types $\mathcal{M}$ of the class. It first generates a type skeleton containing a privilege set variable for each method. It then infers a type of each method using $\mathcal{WM}$, which returns a set of constraints PC. Its definition is given in Figure 6. $\mathcal{WM}$ first sets up a skeleton of each code block and makes a label environment. It then infers a type of each code block using another sub-algorithm $\mathcal{WB}$, which infers a typing of a code block using Unify. Figure 7 gives its defini-

$$
\begin{array}{l}
\mathcal{WC}(\mathcal{C}, \{m_1 = M_1^p, \cdots, m_n = M_n^p\}, c) = \\
\quad \text{let } \mathcal{M} = \{m_1 = \Delta_1 \overset{\emptyset\cdot\rho_1}{\to} \tau_1, \cdots, m_n = \Delta_n \overset{\emptyset\cdot\rho_n}{\to} \tau_n\} \\
\qquad\qquad (\text{if } \Theta.c.m_i = \Delta_i \to \tau_i \text{ for each } i) \\
\qquad PC_0 = \{\rho_1 \sqsubseteq \mathcal{A}(p), \cdots, \rho_n \sqsubseteq \mathcal{A}(p)\} \\
\qquad \mathcal{C}' = \mathcal{C} \cup \mathcal{M} \\
\qquad \text{for each i do} \\
\qquad\quad PC_i = \mathcal{WM}(\mathcal{C}', \mathcal{M}(m_i), M_i^p, PC_{i-1}) \\
\qquad \text{end} \\
\qquad \varphi = \mathsf{Solve}(PC_n) \\
\quad \text{in } \varphi(\mathcal{M})
\end{array}
$$

Figure 5. Type inference algorithm for classes

$$
\begin{array}{l}
\mathcal{WM}(\mathcal{C}, (\Delta \overset{\Pi}{\to} \tau), M^p, PC) = \\
\quad \text{let } \{l_1 = B_1, \cdots, l_n = B_n\} = M^p \\
\qquad \mathcal{B}_{entry} = \Pi, \Delta, p \rhd \tau \\
\qquad \mathcal{B}_i = \rho_i, \delta_i, p \rhd \tau \quad (1 \le i \le n) \\
\qquad \mathcal{L} = \{l_1 = \mathcal{B}_1, \cdots, l_n = \mathcal{B}_n\} \\
\qquad \mathcal{S}_0 = \emptyset,\ \mathcal{K}_0 = \emptyset \\
\qquad PC_0 = PC \cup \{\rho_1 \sqsubseteq \mathcal{A}(p), \cdots, \rho_n \sqsubseteq \mathcal{A}(p)\} \\
\qquad \text{for each i do} \\
\qquad\quad (\mathcal{S}, \mathcal{K}_i, PC_i) = \mathcal{WB}(\mathcal{C}, \mathcal{S}_{i-1}(\mathcal{L}), \mathcal{S}_{i-1}(\mathcal{B}_i), B_i, \mathcal{K}_{i-1}, PC_{i-1}) \\
\qquad\quad \mathcal{S}_i = \mathcal{S} \circ \mathcal{S}_{i-1} \\
\qquad \text{end} \\
\quad \text{in } PC_n
\end{array}
$$

Figure 6. Type inference algorithm for methods

tion. After obtaining a constraint set for all the methods by $\mathcal{WM}$, the main algorithm solves the constraints by Solve, which returns a substitution φ that satisfies PC. The definition of Solve is given in Figure 8.

Note that P appearing in inclusion constraints is a set not containing set variable, and expressions $P_1 \setminus P_2$ and $P_1 \cup P_2$ used in this algorithm are ordinary set-theoretic operations. In contrast to the constrained type system of [13], introduction of expressions corresponding to these set operations in set inclusion constraints is not required in our formalism due to the simpler nature of JVM$_{sec}$ compared to the lambda calculus.

5.2 Example of type inference

We show how the algorithm computes typing using a simple example. Let `writeFile` be a native method defined in the class *IO*, which writes a string to a file and has type $\texttt{str}\cdot\texttt{str} \overset{\{FWrite\}}{\to} \texttt{void}$. The following program reads a file and writes the contents to other file.

```
class SomeClass {
   updateFoo() : str,str -> str {
      new(IO)
      sconst("/protect/foo.txt")
      new(SafeClass)
      invoke(SafeClass,readPublicFile)
      invoke(IO,writeFile)
   }
}
```

$$
\begin{array}{l}
\mathcal{WB}(\mathcal{C}, \mathcal{L}, (\Pi, \Delta, p \rhd \tau), \texttt{return}, \mathcal{K}, PC) = \\
\quad \text{let } \mathcal{K}_1 = \mathcal{K} \cup \{t = *\} \\
\qquad \mathcal{S} = \mathsf{Unify}(\mathcal{K}_1, \{(\Delta, t\cdot\delta), (t, \tau)\}) \\
\quad \text{in } (\mathcal{S}, \mathcal{K}_1, PC)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{WB}(\mathcal{C}, \mathcal{L}, (\Pi, \Delta, p \rhd \tau), \texttt{goto}(l), \mathcal{K}, PC) = \\
\quad \text{let } \Pi', \Delta', p \rhd \tau' = \mathcal{L}(l) \\
\qquad PC_1 = PC \cup \{\Pi' \sqsubseteq \Pi\} \\
\qquad \mathcal{S} = \mathsf{Unify}(\mathcal{K}, \{(\Delta, \Delta'), (\tau, \tau')\}) \\
\quad \text{in } (\mathcal{S}, \mathcal{K}, PC_1)
\end{array}
$$

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

$$
\begin{array}{l}
\mathcal{WB}(\mathcal{C}, \mathcal{L}, (\Pi, \Delta, p \rhd \tau), \texttt{iconst}(c)\cdot B, \mathcal{K}, PC) = \\
\quad \mathcal{WB}(\mathcal{C}, \mathcal{L}, (\Pi, \texttt{int}\cdot\Delta, p \rhd \tau), B, \mathcal{K}, PC)
\end{array}
$$

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

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

$$
\begin{array}{l}
\mathcal{WB}(\mathcal{C}, \mathcal{L}, (\Pi, \Delta, p \rhd \tau), \texttt{new}(c)\cdot B, \mathcal{K}, PC) = \\
\quad \mathcal{WB}(\mathcal{C}, \mathcal{L}, (\Pi, c\cdot\Delta, p \rhd \tau), B, \mathcal{K}, PC)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{WB}(\mathcal{C}, \mathcal{L}, (\Pi, \Delta, p \rhd \tau), \texttt{invoke}(c, m)\cdot B, \mathcal{K}, PC) = \\
\quad \text{let } \Delta' \to \tau' = \Theta.c.m \\
\qquad \mathcal{K}_1 = \mathcal{K} \cup \{t_1 = \Delta'(1), \cdots, t_n = \Delta'(n), t_0 = c\} \\
\qquad\quad (\text{if } |\Delta'| = n) \\
\qquad \mathcal{S}_1 = \mathsf{Unify}(\mathcal{K}_1, \{(t_1 \cdot \ldots \cdot t_n \cdot t_0 \cdot \delta, \Delta)\} \\
\qquad \{c_1, \cdots, c_n\} = \texttt{lookupAll}(c, m) \\
\qquad \Delta' \overset{\Pi_i}{\to} \tau' = \mathcal{C}.c_i.m \\
\qquad PC_1 = PC \cup \{\Pi_i \sqsubseteq \Pi\} \quad (\text{for each } i) \\
\qquad (\mathcal{S}_2, \mathcal{K}_2, PC_2) = \mathcal{WB}(\mathcal{C}, \mathcal{S}_1(\mathcal{L}), \\
\qquad\qquad\qquad \mathcal{S}_1(\Pi, \tau'\cdot\delta \rhd \tau), B, \mathcal{K}_1, PC_1) \\
\quad \text{in } (\mathcal{S}_2 \circ \mathcal{S}_1, \mathcal{K}_2, PC_2)
\end{array}
$$

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

Figure 7. Type inference algorithm for blocks

$\mathsf{Solve}(PC) =$
  if $\mathsf{isSolved}(s)$ for all $s \in PC$ then φ
    where $\forall \rho \in PC\,.\,\varphi(\rho) = \emptyset$
  else let $\{\Pi_1 \sqsubseteq \Pi_2\} \cup PC_0 = PC$
        such that $\mathsf{isSolved}(\Pi_1 \sqsubseteq \Pi_2)$ is false
      $(\varphi_1, \Pi_1' \sqsubseteq \Pi_2') = \mathsf{Solve1}(\{\Pi_1 \sqsubseteq \Pi_2\})$
      $\varphi_2 = \mathsf{Solve}(\{\Pi_1' \sqsubseteq \Pi_2'\} \cup \varphi_1(PC_0))$
    in $\varphi_2 \circ \varphi_1$

$\mathsf{Solve1}(P_1 \sqsubseteq P_2) = \mathit{Failure}$
$\mathsf{Solve1}(P_1 \sqsubseteq P_2 \cdot \rho) = \text{let } P_3 = P_1 \setminus P_2$
  $\varphi = [P_3 \cdot \rho'/\rho] \quad (\rho' \text{ is a fresh})$
  in $(\varphi, P_1 \sqsubseteq (P_2 \cup P_3) \cdot \rho_3)$
$\mathsf{Solve1}(P_1 \cdot \rho \sqsubseteq P_2) = \mathit{Failure}$
$\mathsf{Solve1}(P_1 \cdot \rho_1 \sqsubseteq P_2 \cdot \rho_2) = \text{let } P_3 = P_1 \setminus P_2$
  $\varphi = [P_3 \cdot \rho_3/\rho_2] \quad (\rho_3 \text{ is a flesh})$
  in $(\varphi, P_1 \sqsubseteq (P_2 \cup P_3) \cdot \rho_3)$

$\mathsf{isSolved}(P_1 \sqsubseteq P_2) = P_1 \subseteq P_2$
$\mathsf{isSolved}(P_1 \sqsubseteq P_2 \cdot \rho) = P_1 \subseteq P_2$
$\mathsf{isSolved}(P_1 \cdot \rho \sqsubseteq P_2) = P_1 \subseteq P_2$
$\mathsf{isSolved}(P_1 \cdot \rho_1 \sqsubseteq P_2 \cdot \rho_2) = P_1 \subseteq P_2$

Figure 8. Algorithm Solve

Assume that `SomeClass` is owned by principal *somebody* and $\mathcal{A}$ is set up such that $\texttt{FWrite} \in \mathcal{A}(\textit{somebody})$.

For this class, $\mathcal{WC}$ performs the following computation. It first creates method types $\mathcal{M} = \{\texttt{updateFoo} = \texttt{str} \cdot \texttt{str} \xrightarrow{\{\}\cdot\rho} \texttt{void}\}$ and the inclusion constraints $PC_0 = \{\rho \sqsubseteq \mathcal{A}(\textit{somebody})\}$. It then obtains inclusion constraints $PC = \{\texttt{FWrite} \sqsubseteq \rho, \emptyset \sqsubseteq \rho, \rho \sqsubseteq \mathcal{A}(\textit{somebody})\}$ by invoking the function $\mathcal{WM}$. Next, it calls Solve for *PC*. Solve first obtains the substitution $[\texttt{FWrite}\cdot\rho'/\rho]$ from $\texttt{FWrite} \sqsubseteq \rho$ and transforms *PC* to $\{\texttt{FWrite} \sqsubseteq \texttt{FWrite}\cdot\rho', \emptyset \sqsubseteq \texttt{FWrite}\cdot\rho', \texttt{FWrite}\cdot\rho' \sqsubseteq \texttt{FWrite}\}$. It then returns the substitution $\varphi = [\texttt{FWrite}\cdot\rho'/\rho, \emptyset/\rho']$. Finally, $\mathcal{WC}$ applies φ to $\mathcal{M}$ and returns method types $\{\texttt{updateFoo} = \texttt{str}\cdot\texttt{str} \xrightarrow{\{\texttt{FWrite}\}} \texttt{void}\}$.

The constraints $\texttt{FWrite} \sqsubseteq \rho$ and $\emptyset \sqsubseteq \rho$ in *PC* represent the necessary conditions for invoking the method `writeFile` and `readFooFile` respectively. The substitution φ satisfies these constraints and $\rho \sqsubseteq \mathcal{A}(\textit{somebody})$. If $\texttt{FWrite} \notin \mathcal{A}(\textit{somebody})$, *PC* has no solution, and Solve reports failure.

5.3 Correctness of type inference

For the type inference algorithm just defined to serve as a static verification system for code level access control, it must be sound with respect to the type system of JVM_{sec}, which we shall establish in this section.

Another customary criteria of correctness of a type inference algorithm is its completeness. For our type system, this has two aspects. One is on typings of methods of the form $\Delta \to \tau$ and the other is on accuracy of inferred privilege sets Π. The first aspect is sensitive to the language constructs. Since JVM_{sec} only contains explicitly typed methods and does not contain those mechanisms such as subroutines and object initialization which have subtle interaction with type inference, the first aspect does not involve much significant issues. For this reason, in this paper, we omit its discussion and only consider the second aspect which has significant impact on the usefulness of our method. The interested reader is referred to [5] which deals with type inference with polymorphic subroutines.

The soundness of the type inference algorithm is established by the combination of soundness results of the components of the type inference algorithm.

We first verify that the unification algorithm correctly computes a unifier.

LEMMA 2. *Let E be a set of equations under a bound environment $\mathcal{K}$. If $\mathsf{Unify}(E, \mathcal{K}) = \mathcal{S}$ then S respects $\mathcal{K}$ and S is a unifier for E.*

This is verified by simple inspection of each transformation rule. Using this property, we prove the soundness of $\mathcal{WB}$, which is the main lemma for establishing the soundness of the type inference algorithm.

LEMMA 3. *If $\mathcal{WB}(\mathcal{C}, \mathcal{L}, (\Pi, \Delta, p \rhd \tau), B, \mathcal{K}, PC) = (\mathcal{S}, \mathcal{K}', PC')$ then for all $\mathcal{S}_0$, φ_0 such that φ_0 ground for PC', φ_0 satisfies PC, $\mathcal{S}_0$ ground for $\mathcal{K}'$ and $\mathcal{S}_0$ respects $\mathcal{K}$, the following is derivable: $\varphi_0(\mathcal{S}_0(\mathcal{S}(\mathcal{C}))), \varphi_0(\mathcal{S}_0(\mathcal{S}(\mathcal{L}))) \vdash \varphi_0(\Pi), \mathcal{S}_0(\mathcal{S}(\Delta)), p \rhd B : \mathcal{S}_0(\mathcal{S}(\tau))$.*

PROOF. This is proved by induction on B. The proof proceeds by cases in terms of the first instruction of B. Here we only show some of them.

Case $B = \texttt{return}$. By the definition of $\mathcal{WB}$, $\mathcal{K}' = \mathcal{K} \cup \{t = *\}$, $\mathcal{S} = \mathsf{Unify}(\mathcal{K}_1, \{(\Delta, t\cdot\delta), (t, \tau)\})$, and $PC' = PC$. Let $\mathcal{S}_0$, φ_0 be such that φ_0 ground for *PC*, φ_0 satisfies PC', $\mathcal{S}_0$ ground for $\mathcal{K}'$ and $\mathcal{S}_0$ respects $\mathcal{K}$. By Lemma 2, $\mathcal{S}_1(\Delta) = \mathcal{S}_1(t\cdot\delta)$ and $\mathcal{S}_1(\tau) = \mathcal{S}_1(t)$. Then by the typing rule, we have $\varphi_0(\mathcal{S}_0 \circ \mathcal{S}_1(\mathcal{C}))$, $\varphi_0(\mathcal{S}_0 \circ \mathcal{S}_1(\mathcal{L})) \vdash \varphi_0(\Pi)$, $\mathcal{S}_0 \circ \mathcal{S}_1(\Delta)$, $p \rhd B : \mathcal{S}_0 \circ \mathcal{S}_1(\tau)$.

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 $\mathcal{WB}$, $\Pi' = \{\pi\} \cup \Pi$ and $(\mathcal{S}, \mathcal{K}', PC') = \mathcal{WB}(\mathcal{C}, \mathcal{L}, (\Pi', \Delta, p \rhd \tau), B, \mathcal{K}, PC)$. Let be $\varphi_0, \mathcal{S}_0$ such that φ_0 ground for PC', φ_0 satisfies *PC*, $\mathcal{S}_0$ ground for $\mathcal{K}'$, and $\mathcal{S}_0$ respects $\mathcal{K}$. By the induction hypothesis, $\varphi_0(\mathcal{S}_0 \circ \mathcal{S}(\mathcal{C}))$, $\varphi_0(\mathcal{S}_0 \circ \mathcal{S}(\mathcal{L})) \vdash \varphi_0(\Pi')$, $\mathcal{S}_0 \circ \mathcal{S}(\Delta)$, $p \rhd B : \mathcal{S}_0 \circ \mathcal{S}(\tau)$. By the definition of the type system, $\varphi_0(\mathcal{S}_0 \circ \mathcal{S}(\mathcal{C}))$, $\varphi_0(\mathcal{S}_0 \circ \mathcal{S}(\mathcal{L})) \vdash \varphi_0(\Pi)$, $\mathcal{S}_0 \circ \mathcal{S}(\Delta)$, $p \rhd B : \mathcal{S}_0 \circ \mathcal{S}(\tau)$.

Case $B = \texttt{invoke}(c,m)\cdot B_1$. By the definition of $\mathcal{WB}$, $\Delta' \to \tau' = \Theta.c.m$, $\mathcal{K}_1 = \mathcal{K} \cup \{t_1 = \Delta'(1), \cdots, t_n = \Delta'(n)\}$, $\mathcal{S}_1 = \mathsf{Unify}(\mathcal{K}_1, \{(t_1 \cdot \ldots \cdot t_n \cdot t_0 \cdot \delta, \Delta)\})$, $\{c_1, \cdots, c_n\} = \texttt{lookupAll(c,m)}$, $\Delta' \xrightarrow{\Pi_i} \tau' = \mathcal{C}.c_i.m$, $PC_1 = PC \cup \{\Pi_i \sqsubseteq \Pi\}$, $(\mathcal{S}_2, \mathcal{K}_2, PC_2) = \mathcal{WB}(\mathcal{S}_1(\mathcal{C}), \mathcal{S}_1(\mathcal{L}), \mathcal{S}_1(\Delta, \tau'\cdot\delta, p \rhd \tau, B_1, \mathcal{K}_1, PC_1)$, and $\mathcal{S} = \mathcal{S}_2 \circ \mathcal{S}_1, \mathcal{K}' = \mathcal{K}_3, PC' = PC_2$. Let be $\varphi_0, \mathcal{S}_0$ such that φ_0 ground for PC_2, φ_0 satisfies PC_1, $\mathcal{S}_0$ ground for $\mathcal{K}_2$, and $\mathcal{S}_0$ respects $\mathcal{K}_1$. By induction hypothesis, $\varphi_0(\mathcal{S}_0 \circ \mathcal{S}(\mathcal{C}))$, $\varphi_0(\mathcal{S}_0 \circ \mathcal{S}(\mathcal{L})) \vdash \varphi_0(\Pi), \mathcal{S}_0 \circ \mathcal{S}(\tau'\cdot\delta), p \rhd B : \mathcal{S}_0 \circ \mathcal{S}(\tau)$. By Lemma 2, $\mathcal{S}_1(t_1\cdot, \ldots, \cdot\delta) = \mathcal{S}_1(\Delta)$. Then by the typing rule, $\varphi_0(\mathcal{S}_0 \circ \mathcal{S}(\mathcal{C}))$, $\varphi_0(\mathcal{S}_0 \circ \mathcal{S}(\mathcal{L})) \vdash \varphi_0(\Pi)$, $\mathcal{S}_0 \circ \mathcal{S}(\Delta)$, $p \rhd B : \mathcal{S}_0 \circ \mathcal{S}(\tau)$. Apparently, $\mathcal{S}_0$ respects $\mathcal{K}$ and φ_0 satisfies *PC*. □

We next show the soundness of the inference algorithm $\mathcal{WM}$ for method.

LEMMA 4. *If $\mathcal{WM}(\mathcal{C}, (\Delta \xrightarrow{\Pi} \tau), M^p, PC) = PC'$ then $\mathcal{C} \models M^p : \Delta \xrightarrow{\varphi_0(\Pi)} \tau$ for all φ_0 that is ground for PC and satisfies PC.*

Since $\mathcal{WM}$ simply calls $\mathcal{WB}$ for each block in the method, this follows from Lemma 3 with the following additional property of $\mathcal{WB}$: if $\mathcal{WB}(\cdots, \mathcal{K}, \cdots) = (\mathcal{S}, \mathcal{K}')$ then $\mathcal{S}$ respects $\mathcal{K}$ (under $\mathcal{K}'$).

The following lemma shows that Solve computes the minimal solution of a given constraint set.

LEMMA 5. *1.* Solve *terminates on all inputs.*

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

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

PROOF. The first property follows from the facts that Solve 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 Solve. □

By combining Lemma 4 and Lemma 5, we can show the following.

THEOREM 2 (SOUNDNESS OF $\mathcal{WC}$). *If* $\mathcal{WC}(\mathcal{C}_0, C^p, c) = \mathcal{M}$ *then* $\mathcal{C}_0, \Theta \vdash C^p : \mathcal{M}$.

The following minimality result with respect to privilege set follows from the definition of the algorithm $\mathcal{WB}$ and Lemma 5.

THEOREM 3. *If* $\mathcal{WC}(\mathcal{C}_0, C^p, c) = \mathcal{M}$ *and* $\mathcal{C}_0, \Theta \vdash C^p : \mathcal{M}'$ *such that* $\mathcal{M}'$ *is equal to* $\mathcal{M}$ *except for privilege set annotations, then each privilege set* Π *in* $\mathcal{M}$ *is included in the corresponding* Π' *in* $\mathcal{M}'$.

6 Extensions and discussions

The calculus we have considered so far can be extended in several ways to include practically useful features. This section discusses some of them.

6.1 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 Java access control architecture, a permission consists of a target and an action to be performed on the target. This allows finer access control.

One way to incorporate this feature is to refine a privilege π to be a term of the form $F(v)$ where F denotes an operation name as before and v represents the target object. Integration of those privilege terms in our type system requires several refinements. Firstly, since v denotes a possible runtime value which the static type system can only approximate, we need to introduce a type attribute denoting a set of possible values. Secondly, in order to propagate this attribute information across method invocation boundary, some mechanism for abstraction over those sets of possible values is necessary. A complete access control system including these features is beyond the scope of the current paper. In the following, we describe the necessary refinement to the type system to incorporate these features.

The syntax of the refined set of privileges is given below

$$\begin{array}{rcl} \pi & ::= & F(v) \\ v & ::= & \{s_1, \ldots, s_n\} \mid \alpha \mid v \cup v \mid \bot \end{array}$$

s is an address of an object represented by a string such as an URL, and $\{s_1, \ldots, s_n\}$ denotes a set of (possible) target objects identified by the address $s_1, \ldots, s_n$. α is a variable ranging over sets of target objects, and $v_1 \cup v_2$ denotes the union of v_1 and v_2. $\bot$ denotes the set of all possible objects. v is ordered by set inclusion with $\bot$ the largest element. Koved et. al. [9] have used a similar mechanism in their data flow analysis.

The type system can be extended to incorporate these refined notion of privileges. We assume that the language contains string values of type `str` with a set of operations such as `sconst(s)`. The set 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. For example, $\mathtt{str}(\{s\})$ represents the singleton set and $\mathtt{str}(\bot)$ represents the set of all strings. For this refined string types, the subsumption relation is extended to include the relation generated by the rule:

$$\frac{v_1 \subseteq v_2}{\mathtt{str}(v_1) <: \mathtt{str}(v_2)}$$

The type system is refined to keep track of possible runtime values of stack entries. For example, the typing rule for `sconst` instruction is given as follows.

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

The possible set of target objects ($\{s\}$ in the above example) will be promoted through the subsumption relation above when control flow merges.

The another necessary refinement is to consider a method as polymorphic with respect to the object set variables α appearing in its typing, and to give a polymorphic type in the style of ML's let-polymorphism. Since method is not a first-class object, this treatment is compatible with our type system. We can adopt the technique of introducing let-polymorphism in the JVM we have developed [5] for JVM subroutines. For example, a method which receives a file name and opens the file is given the following polymorphic type.

$$\forall \alpha.(\mathtt{str}(\alpha) \stackrel{\{\mathtt{FOpen}(\alpha)\}}{\rightarrow} \mathit{Void})$$

When this method is invoked with a parameter s, `FOpen(s)` privilege is generated through ML-style type instantiation.

With these refinements, the type system of JVM_{sec} can be extended to incorporate possible target objects of privileged accesses. Some more efforts are needed to extend the type inference algorithm.

6.2 Adding other JVM features

To develop a static verification system for the JVM bytecode language based on our method, we must extend our type system to include various other instructions of JVM. Since the type system is based on the logical presentation of the JVM bytecode language [5], 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 [5] supports polymorphic subroutines, whose treatment is orthogonal to typing mechanism for access control presented here. So, our type system should extend smoothly to JVM subroutines without any additional machinery.

In Java, `checkPermission` can be used to protect object fields. This feature is 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 define a typing rule for field manipulation as follows.

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

The rule for `putfiled` can similarly be defined.

6.3 Implementation issues

In order to develop a practical access control system based on our static method, we have to consider a number of implementation issues. We briefly discuss some of them below.

Compatibility with existing programs. As we have explained earlier, the current practice in Java access control is to dynamically invoking static methods `checkPermission` and `doPrivileged` supplied as a JDK security package. A static access control system should work for existing programs using these methods. One approach 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. A security verification is static type-checking similar to bytecode verification, so it is desirable to unite these two verification systems. Since our security verification system is based on a type theory [5] for bytecode verification, it is not hard to develop a static system which checks type consistency and security violation simultaneously. However, since the JVM bytecode verifier is closely related to a complicated feature such a dynamic class loading, development of such a integrated system requires us to modify a major part of JVM runtime system. A simpler strategy is to check all the class files in a program independently of JVM before executing the program. Adopting this approach, we plan to design a verifier as a stand alone system which reads an access policy file, a target class file, and infers types of methods in the class, and reports a security violation if it is detected.

Specifying privilege requirements. We also need to declare the privilege set for each native method. One possible approach is to directly write it in a class file which 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 credibility of the file.

7 Conclusions

We have developed a static access control system for the JVM 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 verify whether a code conforms to a given access policy or nor directly without relying on explicit insertion of `checkPermision`. This approach can therefore be used as a security verification system for foreign and possibly malicious code. For this type system, we have develops a type inference algorithm, which achieves automatic verification for code-level access control.

Acknowledgments

The authors would like to thank Yasuharu Oda, who have helped in implementing the type inference algorithm and testing our methods through examples. These results have been useful for better understanding our framework. The authors also thank the anonymous referees for helpful comments.

8 References

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

[2] A. Banerjee and D.A. Naumann. Representation independence, confinement and access control. In *Proc. ACM POPL Symposium*, pages 166–177, 2002.

[3] C. Fournet and A. Gordon. Stack inspection: Theory and variants. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 307–318, 2002.

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

[5] T Higuchi and A Ohori. Java bytecode as a typed term calculus. In *Proceedings of the conference on Principles and practice of declarative programming*, 2002.

[6] A Ohori. The logical abstract machine: a Curry-Howard isomorphism for machine code. In *Proceedings of International Symposium on Functional and Logic Programming*, 1999.

[7] Gunter Karjoth. An operational semantics of Java 2 access control. In *In Proc. IEEE Computer Security Foundations Workshop (CSFW'00)*, pages 224–232, 2000.

[8] L.Gong. *Inside Java^{TM} 2 Platform Security*. Addison-Wesley, 1999.

[9] L. Koved, M. Pistoia, and A. Kershenbaum. Access Rights Analysis for Java. In *Proc. ACM OOPSLA Conference*, pages 359–372, 2002.

[10] Ú. Erlingsson and F. Shneider. IRM enforcement of Java stack inspection. In *Proc. IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[11] X. Leroy. *Polymorphic typing of an algorithmic language*. PhD thesis, University of Paris VII, 1992.

[12] T. Lindholm and F. Yellin. *The Java virtual machine specification*. Addison Wesley, second edition edition, 1999.

[13] Christian Skalka and Scott Smith. Static enforcement of security with types. In *Proc. International Conference on Functional Programming(ICFP'00)*, pages 34–45, 2000.

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

[15] Dan S.Wallach, Andrew W.Appel, and Edrard W.Felten. Safkasi:a security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9:341–378, 2000.

[16] D.S. Wallach and E.W. Felten. Understanding Java stack inspection. In *Proc. IEEE Symposium on Security and Privacy*, pages 52–63, 1998.