Bytecode Verification for Enhanced JVM Access Control

Dongxi Liu
School of Information Science and Technology, University of Tokyo
liu@mist.i.u-tokyo.ac.jp

Abstract

This paper presents an approach to addressing the known weaknesses and security issues of JVM stack inspection in a unified framework. We first propose an enhanced JVM access control mechanism. In this mechanism, values are also associated with security levels. When enforcing access control, this mechanism checks not only the permissions of code on stack as the usual stack inspection, but also the security levels of values to make sure they are used legally. We then present a static type system to verify whether a bytecode program satisfies the security property achieved by this enhanced mechanism. This type system performs modular and context-sensitive analysis at the method level by generating and solving constraints, and path-sensitive analysis at the code block level by using a trace-based approach. In addition, this type system does not need any user annotation for verification.

1 Introduction

The extensible programs based on mobile code always consist of software components with different origins and different trust levels. When an extensible program runs, all its components share the same system resources, so it is desirable to enforce access control at the code level rather than according to the rights of users running this program. A well known mechanism for this situation is stack inspection implemented in Java Virtual Machine (JVM) [4].

In stack inspection, each piece of code is associated with some permissions according to the prescribed security policy. During runtime, when some code intends to access sensitive resources, this mechanism scans the current stack frames top-down to guarantee that all running code has the required permissions, unless it meets a stack frame with a special flag. In Java, the method `checkPermission` implements the stack inspection mechanism and the method `doPrivileged` sets the special flag. To enforce access control, the method `checkPermission` must be explicitly called in the API code for accessing resources.

However, this mechanism has been found some inherent weaknesses and security issues. The known weaknesses include: inspecting stack dynamically incurs runtime overhead; bytecode programs cannot be optimized by some methods if they change the calling stack after optimization; it is difficult to guarantee that the implementers always insert the method `checkPermission` in each API code for accessing sensitive resources. On the other hand, the known security issues are caused by two reasons. The first reason is that stack inspection does not check all code that have run, and some untrusted code possibly left the running system in an unexpected state after they terminate. The second is that some classes encapsulating the sensitive resources, such as `FileInputStream` and `Socket`, invoke stack inspection only when creating the instances of the classes, and do not perform security check when really accessing resources by using the methods, such as `read` or `write`. However, the objects of these classes created by trusted code could be carelessly passed or returned to untrusted code, and then used by them illegally beyond the control of stack inspection. Exploiting these security issues, untrusted code can access sensitive resources without required permissions. There are more details and examples about these security issues in [1, 4].

There have been much work, such as the security-passing style in [10] and the type systems in [8, 5, 7], to avoid all or some the weaknesses of stack inspection, but they do not address the known security issues. The work in [1] proposes a method to address the security issue caused by the first reason by dynamically examining the permissions of all code that has run in execution history. However, it cannot solve the security problem caused by the second reason, and still suffers from some weaknesses of stack inspection, such as runtime overhead and manually-inserted code for security check. The purpose of this work is to address these weaknesses and security issues in one framework.

1.1 Our Approach

In stack inspection, only code is associated with permissions, while in our approach, the values like objects or strings are also specified with some security level (i.e., a set of permissions). The security mechanism is enforced not only by checking the permissions of code on stack, but also checking whether a value is used legally. An object is legally used only when its methods are invoked by the objects with higher security level, i.e., owning more permissions, and for a string, it is legally used when its security level is higher than that of the objects whose methods take this string as an argument to specify sensitive resources.

The enforcement of access control in this model is completely guided by security policies, which specify not only the permissions of code,but also the permissions required to access sensitive resources. The stack inspection is invoked when some code tries to access the resources protected by some security policy, rather than by explicitly calling a special method like `checkPermission`. Thus, in order to check which resources are in protection, the administrators just need to look at the current policies, which is more effective than to seek calls of `checkPermission` scattered in code base.

We first present an operational semantics for a secure JVM calculus. This operational semantics enforces security check based on security policies, embodying our idea of enhancing stack inspection by ensuring the legal use of values. If some code fails to pass security check, it will get stuck according to the semantic rules. This mechanism can address the security issues of stack inspection, but it still incurs runtime overhead since runtime security check is used.

To address the shortcoming of dynamic security check, we then propose a type system to verify the enhanced security property of bytecode programs statically. In this type system, the types of values are annotated with security levels, and the type signatures of methods are annotated with permissions required to invoke them. Hence, we can depend on these security levels or permissions to check whether a program is secure. However, it is boring and error prone for users to annotate their programs for verification. In our approach, users do not need to annotate the code, and the security levels on types are inferred automatically by this type system based on security policies.

Several technical difficulties are encountered when designing this type system. First, a method or a code block might require a context-sensitive analysis because they may be called or entered with different security contexts. However, because of recursive methods or loops in code blocks, the type analysis will fail to terminate if we naively analyze a code block or the body of a method at each jumping or calling point. Moreover, it is not a modular approach. Second, due to the mechanism of initializing objects in JVM [3], the security level of an object cannot be determined at the time it is created, and when its security level can be determined, the type of this object probably has been duplicated and moved, and thus appears in several different places. So this type system must be able to recognize the types of the same object and annotate them with the same security level.

We propose several techniques to tackle these difficulties. First, this type system analyzes each method only once and generates constraints for each method after analysis. At each method calling point, the constraints of the invoked method are instantiated with the information specific to this calling context to achieve context-sensitive analysis. A program is secure if all these constraints can be satisfied. Second, this type system analyzes the code blocks in one method by following the control flow among them. In order to prevent infinite analysis loops, this type system builds and maintains a set of predecessor traces for the entry point of each block from which this block has been entered and analyzed, which can then be used to determine whether this block needs to be analyzed again when there is a reentry to it. In addition, this trace-based approach allows the analysis of polymorphic code blocks, such as bytecode subroutines [9]. Third, if the security level of a class type cannot be determined, this type is represented as a singleton type, and at the time its security level can be determined, all the singleton types with the same index value, indicating the replica types of the same object, will be annotated with the same security level. In addition, singleton class types can also help in detecting the use of uninitialized objects. Singleton string types allows more precise permissions. That is, a permission is defined by an access operation together with a resource identifier, not just by an access operation like those in [8, 7].

The remainder of this paper is organized as follows: Section 2 describes the syntax of security policies; Section 3 gives the secure JVM calculus and its operational semantics; Section 4 presents the type system; Section 5 gives the related work and the conclusion.

2 Security Policies

In this work, security policies not only assign permissions to code but also specify the permissions required for accessing sensitive resources. The syntax of security policies is defined in Figure 1. A permission π or $\theta(x)$ is a pair of an access operation and a resource identifier. A resource identifier is the name of some resource, such as a file name or a network address. π is a permission instance, and $\theta(x)$ is a parameterized permission, where x is a string variable and can be instantiated.

The permissions of code are specified by P which maps a class name to a set Π of permission instances. These permissions are assigned generally according to the features of code, such as its origin or signature. For example, if some code has the permissions $\{$ (`FileRead`, “pwd.dat”), (`FileDelete`, “pwd.dat”) $\}$, then it can read and delete

$$
\begin{array}{rcl}
C & \in & \mathit{Classes} \\
\pi & \in & \{(\texttt{FileRead}, str_1),(\texttt{FileDelete}, str_2), ...\} \\
\Sigma, \Pi & ::= & \{\pi, ...\} \\
P & ::= & \{C \mapsto \Pi, ...\} \\
\theta(x) & \in & \{(\texttt{FileRead}, x), (\texttt{FileDelete}, x), ...\} \\
\Theta(x) & ::= & \{\theta(x), ...\} \\
O & ::= & \{(C, i) \mapsto \lambda x.(\Theta(x), \Theta'(x))\}, ...)\} \\
Q & ::= & \{(C, m, i) \mapsto \lambda x.\Theta(x), ...\} \\
x & \in & \mathit{String\ Variables} \\
i & \in & \mathit{Integers}
\end{array}
$$

Figure 1. Syntax of Security Policies

the file "pwd.dat"; if it has an empty permission set, denoted by ϕ, then it has no right to access any resource.

The permissions for sensitive resources are specified either on class constructors by the mapping O or on methods by the mapping Q.

The mapping O maps a pair of a class name and an integer to a pair of parameterized permission sets, with the free variable x bound by λ. The notation $Dom(O)$ is for the domain of O. A mapping $(C, i) \mapsto \lambda x.(\Theta(x), \Theta'(x))$ in O declares that when creating an object from C, the ith argument of its constructor should be a string, say *str*, for identifying the resource encapsulated by this class, and the permission sets Θ(*str*) and Θ'(*str*) obtained by applying $\lambda x.(\Theta(x), \Theta'(x))$ to *str* specify respectively the permissions required for creating this object and the security level of the created object.

For example, to protect from unauthorized file read through the class `FileInputStream`, the mapping entry $(\texttt{FileInputStream}, 1) \mapsto \lambda x.(\{(\texttt{FileRead}, x)\}$, $\{(\texttt{FileRead}, x)\})$ should be declared in O. This protection applies to the constructor of `FileInputStream` taking the file name as its argument. However, the class `FileInputStream` also has a constructor taking a `File` object as its argument. In this case, this constructor is not protected by the security policy, and instead the constructor of the class `File` with a file name parameter should be protected by declaring $(\texttt{File}, 1) \mapsto \lambda x.(\Theta(x), \Theta'(x))$, where both $\Theta(x)$ and $\Theta'(x)$ are the set $\{$(`FileDelete`, x), (`FileRead`, x) $\}$, since this class can be used to delete files and also read files. Note that $\Theta(x)$ and $\Theta'(x)$ can be different.

The mapping Q protects those methods that directly take resource identifiers as arguments, such as `createTempFile` in the class `File`. A mapping $(C, m, i) \mapsto \lambda x.\Theta(x)$ in Q means that the code calling the method m defined in the class C must have permission $\Theta(str)$, where str is the ith argument of this call. To support overloaded class constructors and methods, the domain formats of O and Q can be extended with the type signature of each constructor and method. For simplicity, they are omitted.

$$
\begin{array}{rcl}
Prog & ::= & \{C_1 <: C'_1 \mapsto MS_1, ..., C_n <: C'_n \mapsto MS_n\} \\
MS & ::= & \{md_1 \mapsto BS_1, ..., md_n \mapsto BS_n\} \\
md & ::= & (C, m, \tau) \\
\sigma & ::= & \texttt{int} \mid \texttt{string} \mid C \\
\Delta & ::= & \epsilon \mid \sigma \cdot \Delta \\
\tau & ::= & \Delta \rightarrow \sigma \\
BS & ::= & \{l_1 \mapsto B_1, ..., l_n \mapsto B_n\} \\
B & ::= & \texttt{return} \mid \texttt{goto}\ l \mid in \cdot B \\
in & ::= & \texttt{new}\ C \mid \texttt{iconst}\ n \mid \texttt{ldc}\ str \mid \texttt{dopriv} \\
 & & \mid \texttt{store}\ x \mid \texttt{load}\ x \mid \texttt{dup} \mid \texttt{ifeq}\ l \\
 & & \mid \texttt{invokevirtual}\ md \mid \texttt{invokespecial}\ md \\
v & ::= & n \mid str \mid \{\}_C
\end{array}
$$

Figure 2. Syntax of Secure JVM Calculus

3 A Secure JVM Calculus

In this section, we formalize a secure JVM calculus. Its operational semantics enforces the enhanced stack inspection by runtime check based on security policies.

3.1 Syntax of the Calculus

The syntax of the JVM calculus is defined in Figure 2. A program *Prog* consists of a set of classes. A class is denoted by $C <: C' \mapsto MS$, which means the class C extends the class C' and has a collection *MS* of methods. The inheritance relation $<:$ is transitive. A method is described by a method descriptor *md* associated with a list of code blocks *BS*. Each block B has a label l, and the block labeled with l_1 is the entry block. For a method descriptor *md* of the form $(C, m, \Delta \rightarrow \sigma)$, we use the notation *class*(*md*) for its owner class C,*mname*(*md*) for its name m,*ty*(*md*) for its signature $\Delta \rightarrow \sigma$, and *argty*(*md*) and *resty*(*md*) for Δ and σ, respectively. The notation $blk(md, l)$ denotes the code block l in the method described by md, that is, $blk(md, l) = BS(l)$, where $BS = MS(md)$ and $MS = Prog(class(md))$.

A code block B consists of a sequence of instructions ended with `return` or `goto` l. The instruction `new` *C* allocates an uninitialized object of class *C* on the heap and pushes its reference onto the operand stack; `iconst` *n* and `ldc` *s* push the integer *n* or the string *s* onto the stack, respectively; `dopriv` models the method `doPrivileged` in Java; `load` *x* pushes the value of local variable *x* onto the stack, and `store` *x* puts it back; `dup` duplicates the value at the top of stack, and is used for object initialization [3]; `ifeq` l jumps to the block l if the top value of the stack is zero; `invokevirtual` calls an ordinary method, and `invokespecial` invokes a class constructor to initialize an object. A value v can be an integer n, a string str or an object $\{\}_C$. An object is annotated with its class C, and is an empty record since classes in this calculus do not include fields. Adding fields to classes does not affect the security mechanism of this calculus.

1: $(f, v \cdot s, \Pi, \texttt{return})_{md} \cdot (f', s', \Pi', B')_{md'} \cdot A; h$
$\quad \rightarrow (f', v \cdot s', \Pi', B')_{md'} \cdot A; h$

2: $(f, s, \Pi, \texttt{goto}\ l)_{md} \cdot A; h \rightarrow (f, s, \Pi, \mathit{blk}(md, l))_{md} \cdot A; h$

3: $(f, s, \Pi, \texttt{new}\ C \cdot B)_{md} \cdot A; h \rightarrow (f, o \cdot s, \Pi, B)_{md} \cdot A; h[o \mapsto \{\}^{\phi}_{C}]$
$\quad$ where $o \notin \mathit{Dom}(h)$

4: $(f, s, \Pi, \texttt{iconst}\ n \cdot B)_{md} \cdot A; h \rightarrow (f, n^{\Sigma} \cdot s, \Pi, B)_{md} \cdot A; h$
$\quad$ where $\Sigma = P(\mathit{class}(md))$

5: $(f, s, \Pi, \texttt{ldc}\ \mathit{str} \cdot B)_{md} \cdot A; h \rightarrow (f, \mathit{str}^{\Sigma} \cdot s, \Pi, B)_{md} \cdot A; h$
$\quad$ where $\Sigma = P(\mathit{class}(md))$

6: $(f, s, \Pi, \texttt{dopriv} \cdot B)_{md} \cdot A; h \rightarrow (f, s, \Pi \cup \Sigma, B)_{md} \cdot A; h$
$\quad$ where $\Sigma = P(\mathit{class}(md))$

7: $(f, v \cdot s, \Pi, \texttt{store}\ x \cdot B)_{md} \cdot A; h \rightarrow (f[x \mapsto v], s, \Pi, B)_{md} \cdot A; h$

8: $(f, s, \Pi, \texttt{load}\ x \cdot B)_{md} \cdot A; h \rightarrow (f, f(x) \cdot s, \Pi, B)_{md} \cdot A; h$

9: $(f, v \cdot s, \Pi, \texttt{dup} \cdot B)_{md} \cdot A; h \rightarrow (f, v \cdot v \cdot s, \Pi, B)_{md} \cdot A; h$

10: $(f, 0^{\Sigma} \cdot s, \Pi, \texttt{ifeq}\ l \cdot B)_{md} \cdot A; h \rightarrow (f, s, \Pi, \mathit{blk}(md, l))_{md} \cdot A; h$

11: $(f, n^{\Sigma} \cdot s, \Pi, \texttt{ifeq}\ l \cdot B)_{md} \cdot A; h \rightarrow (f, s, \Pi, B)_{md} \cdot A; h\ (n \neq 0)$

12: $(f, s' \cdot o \cdot s, \Pi, \texttt{invokespecial}\ md' \cdot B)_{md} \cdot A; h$
$\quad \rightarrow (\{0 \mapsto o, |s'|..1 \mapsto s'\}, \epsilon, \Pi \cap P(C), \mathit{blk}(md', l_1))_{md'} \cdot$
$\qquad (f, s, \Pi, B)_{md} \cdot A; h'$
$\quad$ where $h(o) = \{\}^{\phi}_{C}, \mathit{class}(md') = C,$
$\qquad \mathit{mname}(md') = <\mathrm{init}>, |s'| = |\mathit{argty}(md')|,$
$\qquad \mathit{sc1}(C, 1, s', \Pi) = \Sigma$ and $h' = \mathit{anno}(h, o, \Sigma)$

13: $(f, s' \cdot o \cdot s, \Pi, \texttt{invokevirtual}\ md' \cdot B)_{md} \cdot A; h$
$\quad \rightarrow (\{0 \mapsto o, |s'|..1 \mapsto s'\}, \epsilon, \Pi \cap P(C'), \mathit{blk}(md'', l_1))_{md''} \cdot$
$\qquad (f, s, \Pi, B)_{md} \cdot A; h$
$\quad$ where $h(o) = \{\}^{\Sigma}_{C}, C <: \mathit{class}(md'), |s'| = |\mathit{argty}(md')|,$
$\qquad \Sigma \subseteq \Pi, m = \mathit{mname}(md'), C' = \mathit{srcclass}(C, m),$
$\qquad \mathit{sc2}(C', m, 1, s', \Pi) = \phi$ and $md'' = (C', m, \mathit{ty}(md'))$

Figure 3. Operational Semantics with Run-time Security Check

$\mathit{sc1}(C, i, s, \Pi) = \phi,$ if $i > |s|$
$\mathit{sc1}(C, i, s, \Pi) = \mathit{sc1}(C, i+1, s, \Pi)$
$\quad$ if $i < |s|$ and $(C, i) \notin \mathit{Dom}(O)$
$\mathit{sc1}(C, i, s, \Pi) = \Theta'(\mathit{str}) \cup \mathit{sc1}(C, i+1, s, \Pi)$
$\quad$ if $i < |s|, O((C, i)) = \lambda x.(\Theta(x), \Theta'(x)), s[|s| - i + 1] = \mathit{str}^{\Sigma},$
$\qquad \Theta(\mathit{str}) \subseteq \Pi$ and $\Theta(\mathit{str}) \subseteq \Sigma$

$\mathit{sc2}(C, m, i, s, \Pi) = \phi,$ if $i > |s|$
$\mathit{sc2}(C, m, i, s, \Pi) = \mathit{sc2}(C, m, i+1, s, \Pi)$
$\quad$ if $i < |s|$ and $(C, m, i) \notin \mathit{Dom}(Q)$
$\mathit{sc2}(C, m, i, s, \Pi) = \mathit{sc2}(C, m, i+1, s, \Pi)$
$\quad$ if $i < |s|, Q((C, m, i)) = \lambda x.\Theta(x), s[|s| - i + 1] = \mathit{str}^{\Sigma},$
$\qquad \Theta(\mathit{str}) \subseteq \Pi$ and $\Theta(\mathit{str}) \subseteq \Sigma$

Figure 4. Operators *sc1* and *sc2*

The type $\Delta \rightarrow \sigma$ is the type for methods. Its argument type Δ is a stack type, which is a sequence of types. An empty sequence is written as ϵ. There are several notations about stack types: $|\Delta|$ is the length of Δ, and $\Delta\,[i]$ returns the ith entry of the stack type Δ for $1 \le i \le |\Delta|$, which is defined as $(\sigma \cdot \Delta)\,[1] = \sigma$ and $(\sigma \cdot \Delta)\,[2] = \Delta\,[1]$, and so on. These notations are also used on other kinds of sequences, such as sequences of values. The operator $\cdot$ is used to concatenate two sequences.

3.2 The Operational Semantics

The operational semantics describes how programs execute from one state to the next state. The operational semantics for the calculus is defined in Figure 3. An execution state is a configuration $A; h$ consisting of a sequence A of stack frames and a global heap h. A stack frame for executing the method *md* has the form $(f, s, \Pi, B)_{md}$, where f maps local variables (specified by integers) to values, s is the operand stack (a sequence of values), and Π is the valid permissions owned by code B. The heap h maps an object reference to an object, and the notation $h[o \mapsto v]$ means a new mapping h', such that $h'(o) = v$, and $h'(o') = h(o')$ if $o \neq o'$.

The state $(f, v \cdot s, \Pi, \mathit{return})_{md}; h$ is regarded as the final state since there is no instruction to execute and the whole program will return successfully a value v. In the following, we will introduce how this operational semantics annotates values with security levels, enforces stack inspection, and checks whether values are used legally.

The instruction `new` allocates a new uninitialized object on the heap. For this object, its initial security level does not contain any permission, denoted by ϕ. This security level will be changed when initializing this object using the method `invokespecial` if the class of this object is protected by some security policy. The `ldc` instruction annotates the string constant *str* with the security level $P(\mathit{class}(md))$, which is the permissions of the method *md* being executed since the constant string *str* belongs to this method. The instruction `dopriv` amplifies the valid permissions of its subsequent code by incorporating the permissions $P(\mathit{class}(md))$ owned by the current method md.

The instruction `invokespecial` invokes the class constructor of class C to initialize the object o. The values o and s' in the operand stack are the arguments for invoking the constructor. The notation $|s'|..1 \mapsto s'$ means the integer i $(1 \le i \le |s'|)$ is mapped to the value $s'[|s'| - i + 1]$ in the new stack frame. The security check in this rule is done by the operator *sc1* in Figure 4. If the string argument *str* refers to some sensitive resource according to security policies, this operator checks the condition $\Theta(\mathit{str}) \subseteq \Pi$ to make sure the current code has enough permissions to create the object o and the condition $\Theta(\mathit{str}) \subseteq \Sigma$ to make sure *str* is legally used to instantiate this object. The former check enforces the stack inspection mechanism, and the latter prevents trusted code from accessing resources specified arbitrarily by untrusted code. If the security check succeed, *sc1* will return a set of permissions Σ, which is then used by *anno* to update the security level of the object o on the heap. That is, suppose $h(o) = \{\}^{\phi}_{C}$ and $h' = \mathit{anno}(h, o, \Sigma)$. Then $h'(o) = \{\}^{\Sigma}_{C}$, and $h'(o') = h(o')$ if $o \neq o'$.

The instruction `invokevirtual` invokes a method of object o. This instruction first checks whether object o is legally used by comparing its security level with the permissions of current code, i.e., $\Sigma \subseteq \Pi$. And then, based on

$$\begin{array}{rcl}\gamma & ::= & \texttt{int} \mid \texttt{string}(str) \mid C \mid C(o) \\ \delta & ::= & \gamma^Y \\ S, T & ::= & \epsilon \mid \delta \cdot S \\ \tau & ::= & T \xrightarrow[D]{X,Y} \delta \\ X, Y & \in & Variables\ of\ Permission\ Sets\end{array}$$

Figure 5. Syntax of Internal Types

the security policy, it uses the operator *sc2* to check whether current code has enough permissions to invoke this method and the validity of string arguments if they refer to sensitive resources. Note that due to dynamic dispatching, we need to determine the method which is really executed. The notation $srcclass(C, m)$ returns the class C itself if it implements the method m or returns its least super class C' that implements m. Hence, in this instruction, the security policy declared on the method m in the class $srcclass(C, m)$ is really enforced by *sc2*.

4 The Type System for Verification

In this section, we give a static type system to enforce the same security check done before by the operational semantics.

4.1 Internal Types

The syntax of types used internally by the type system is given in Figure 5. The singleton type $\texttt{string}(str)$ denotes a string type for the string str, $C(o)$ a class type for the object o. The type δ is the result of annotating γ with a security level variable. In the method type τ, the annotation X is the security level of the implicit object argument (i.e., `this` in Java), Y indicates the permissions required to invoke this method, and the variable D represents the constraints among the components of this type signature. The value of D is determined by security policies and the conditions for invoking other methods in this method body, which will be described in later sections.

Each method in JVM bytecode programs has explicit type signatures. In order to verify them using this type system, we need to translate their type signatures into the internal form. For the method type $\Delta \rightarrow \sigma$ in Figure 2, it can be translated into the form $T \xrightarrow[D]{X,Y} \delta$ by the following steps:

1) The type `string` in Δ or σ is changed into $\texttt{string}(str)^X$, and the class type C is changed into C^X, where *str* is a fresh string variable, X a fresh security level variable.

2) The annotation X should be ϕ if $\Delta \rightarrow \sigma$ is the type of a class constructor, since when calling a constructor the object to be initialized has no real security level yet; otherwise, X must be a fresh variable. The variables Y and D must also be fresh.

The singleton class type $C(o)$ is not used in the method signature, that is, uninitialized objects cannot be passed as arguments or returned values. It is used only when an object has been allocated on the heap, but has not been initialized. This is because the type of this newly allocated object has to be duplicated to initialize it [3], and after determining its security level in initialization, we must update all replicas of this type by annotating them with this security level. The index value o is to help recognize all such replicas.

4.2 Analysis Traces among Blocks

The body of a method generally consists of a number of code blocks. This type system starts from the first code block and analyzes all reachable blocks following the control flow. But the type system cannot naively begin to analyze a block each time when it meets a `goto` or `ifeq` instruction targeting to this block. The code blocks in a method may contain loops, and by this way the type analysis will fail to terminate. To solve this problem, this type system builds and maintains for the entry point of each block a set of predecessor traces from which this block has been entered and analyzed. A block is analyzed only when the current trace entering this block is new.

An analysis trace tr, defined below, is a sequence of block labels: $l_1 \cdot l_2 ... \cdot l_n$, meaning that the blocks l_1, l_2,..., l_n have been analyzed in sequence by the type system.

$$tr ::= \epsilon \mid tr \cdot l \qquad J ::= \{l_1 \mapsto \{tr, .., tr\}, ..., l_n \mapsto \{tr, .., tr\}\}$$

The mapping J in this definition is used by the type system to record for each block its predecessor traces. At the beginning of analyzing a method, J maps each block label to an empty trace set. When entering a block l from the trace tr, J is updated by the operator $log(J, l, tr)$, which returns a new mapping J', such that $J'(l)$=$J(l) \cup \{tr\}$, and $J'(l')$=$J(l')$ if $l' \neq l$. At the end of analysis, J records for each block all its predecessor traces covered by the type system.

A trace tr is new for a code block l with respect to the mapping J, written as $newtr(tr, J, l)$, if there does not exist $tr' \in J(l)$, which satisfies: 1) $tr' = tr \cdot tr_1$; or 2) $tr = tr' \cdot tr_2$ and $tr' = tr_3 \cdot tr_2$ for some traces $tr_i (1 \leq i \leq 3)$. The first condition says tr has been covered by the previous trace tr', and the second condition says tr falls in a loop trace by repeating the subtrace tr_2. The notation $oldtr(tr, J, l)$ means tr is not new.

Suppose there is a method with the control flow graph given in Figure 6. The type system will finally generate the mapping J in Figure 7. For example, under this J, the analysis trace $l_1 \cdot l_2 \cdot l_4 \cdot l_2 \cdot l_4$, coming from the block l_4, is

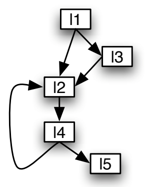

Figure 6. The Control Flow Graph

$$\begin{array}{lcl}
l_1 & \mapsto & \phi \\
l_2 & \mapsto & \{l_1, l_1 \cdot l_3, l_1 \cdot l_2 \cdot l_4, l_1 \cdot l_3 \cdot l_2 \cdot l_4\} \\
l_3 & \mapsto & \{l_1\} \\
l_4 & \mapsto & \{l_1 \cdot l_2, l_1 \cdot l_3 \cdot l_2, l_1 \cdot l_2 \cdot l_4 \cdot l_2, l_1 \cdot l_3 \cdot l_2 \cdot l_4 \cdot l_2\} \\
l_5 & \mapsto & \{l_1 \cdot l_2 \cdot l_4, l_1 \cdot l_3 \cdot l_2 \cdot l_4, l_1 \cdot l_2 \cdot l_4 \cdot l_2 \cdot l_4, \\
 & & \ l_1 \cdot l_3 \cdot l_2 \cdot l_4 \cdot l_2 \cdot l_4\}
\end{array}$$

Figure 7. The Entering Traces of Each Block

not new for the block l_2 because the trace $l_1 \cdot l_2 \cdot l_4 \in J(l_2)$ satisfies the second condition above. The traces $J(l_5)$ for the exit block l_5 gives all traces (with the concatenation of l_5) covered by this type system.

Based on the mapping J obtained at the end of analysis, we can check whether all blocks in a method reachable from its entry block can lead to one of its exit blocks. Suppose a method md contains n blocks and $Tr = J(l_1) \cup ... \cup J(l_n)$. Then, the predicate $wellcode(md, J)$ defined below performs such a check, which will be used later for the well-typedness of method md.

The predicate $wellcode(md, J)$ holds under the condition that the last instruction of the block $blk(md, l_i)$ $(1 \leq i \leq n)$ must be `return` if there exists a trace $tr \in J(l_i)$, which is not covered by any trace in $Tr \setminus J(l_i)$.

4.3 Constraints

This type system verifies program modularly, that is each method is analyzed only once. After analysis of a method, a set of constraints is generated, as the value of the variable D in its type, to keep all conditions of securely executing this method. The syntax of constraints is given in Figure 8. R represents a set of constraints. A constraint can be either an equation between strings or an inclusion between permission sets. Θ is just $\Theta(x)$ without restricting the resource identifiers to be variable x. A set of constraints can be satisfied if all used string and security level variables can be unified to some strings or permission instances such that all inclusions and equations in this set hold.

Before analyzing each method, we need to prepare an initial constraint as the input of the type system. The initial

$$\begin{array}{lcl}
R & ::= & \{I, ..., I\} \\
I & ::= & Z_1 \subseteq Z_2 \mid str_1 = str_2 \\
Z & ::= & X \mid Y \mid \Pi \mid \Theta
\end{array}$$

Figure 8. Syntax of Constraints

constraint contains the conditions derived from the security policy and the conditions of the legal use of strings.

Suppose the constructor of the class C has the type $T \xrightarrow[D]{\phi, Y} \texttt{void}^{Y'}$, where Y' on a void type is specially for the security level of the object being initialized. Then the initial constraint for this constructor is built by the following rules: 1) If $\not\exists i.(C, i) \in Dom(O)$, then its initial constraint is the set $\{Y \subseteq P(C)\}$; or 2) $\forall i.O((C, i)) = \lambda x.(\Theta(x), \Theta'(x))$, and if $T[i] = \texttt{string}(str)^{X'}$, then the set $\{Y \subseteq P(C), \Theta(str) \subseteq X', \Theta(str) \subseteq Y, \Theta'(str) \subseteq Y'\}$ is its initial constraint.

Suppose the method m of the class C has the type $T \xrightarrow[D]{X, Y} \gamma^{Y'}$. Then the initial constraint for this method is built by the following rules: 1)If $\not\exists i.(C, m, i) \in Dom(Q)$, then its initial constraint is the set $\{Y \subseteq P(C)\}$; or 2) $\forall i.Q((C, m, i)) = \lambda x.\Theta(x)$, and if $T[i] = \texttt{string}(str)^{X'}$, then the set $\{Y \subseteq P(C), \Theta(str) \subseteq X', \Theta(str) \subseteq Y\}$ is its initial constraint. Note that a high level object can flow into a low level code since we do not have condition on Y', but cannot be used in the low level code.

4.4 The Type System

The typing rules are given in Figure 9. The judgment has the form

$$F, S, X, tr, J, R \vdash^{l}_{md} B \rightsquigarrow (J', R')$$

which means that the code block B with the label l in the method md is checked under the context described by F, S, Π, tr, J and R, and after type checking, a new J' and R' are generated. In this judgment, F maps local variables to types; S is a stack type for the operand stack; X represents the valid permissions owned by the current code; tr indicates the trace from which the current block l is entered; J records the traces the typing procedure has covered; R is the constraint generated for the method md so far.

For a method *md* with the type $T \xrightarrow[D]{X, Y} \gamma^{Y'}$, we start from analyzing its entry block by deriving the judgment $F_I, \epsilon, Y, \epsilon, J_I, R_I \vdash^{l_1}_{md} blk(md, l_1) \rightsquigarrow (J', R')$, where F_I maps 0 to $class(md)^X$, and i to $T[i]$ for $1 \leq i \leq |T|$, and the initial J_I and R_I are built as discussed in the previous subsections. The constraint R' will be used as the value of the constraint variable D. The method md is well-typed if the above judgment can be derived and the predicate $wellcode(md, J')$ holds because when all possible traces can reach an exit block, the first two typing rules in Figure 9 guarantee each trace has the correct returning type.

The operation `match` is used in the typing rules for the instructions `invlokespecial` and `invlokevirtual`. The `match`(S,T) operation checks whether the argument type S from the caller matches the argument type T declared in the method type, and returns a set of substitutions ρ if S and T are matched, otherwise indicates a type error. A substitution consists of the items X/Y or str/str', meaning that the substitution of X for Y, or str for str'. The notation ρD means the application of substitutions ρ to the terms in the constraint D. By this way, at each calling point, the constraint on securely invoking a method is instantiated with the arguments and security annotations specific to that point. Hence, the analysis in this work is context-sensitive without sacrificing modularity.

The `update` operation is used in the rule for `invlokespecial` to update the security level on the types of uninitialized objects. A type for the same object probably appears in multiple places in the stack type S or in the range of the mapping F because of, for instance, the typing rules for `dup` and `store`. The operator $\texttt{update}(S, o, Y')$ (or $\texttt{update}(F, o, Y')$) changes the security level of all singleton class types in S (or in the range of F) into Y' if their index values are o. And after updating their security level, the `update` operation also changes them into ordinary class types. This approach can also be used to detect whether an object is used without initialization. If the implicit object argument in the instruction `invlokevirtual` has a singleton class type, then this program is trying to use an uninitialized object, which is not allowed.

4.5 The Property of the Type System

After analyzing a byecode program, if all methods are well-typed, then we solve the generated constraints to check whether this program is secure or not. Let $\rightarrow^*$ be the transitive closure of $\rightarrow$, and ϕ specify the empty local variable mapping or the empty heap. The property of the type system is stated as follows.

Suppose a closed program executes from a static method *md*, and *ty*(md) = $\epsilon \longrightarrow \sigma$. If all methods are well-typed and their constraints can be satisfied, then the following transition

$$(\phi, \epsilon, P(class(md)), B)_{md}; \phi \rightarrow^* \\ (f', v \cdot s, P(class(md)), \texttt{return})_{md}; h'$$

holds and v has the type σ.

That is, for a well-typed program, if their constraints can be satisfied, then the program can run without causing security check failure, and return the value with correct type upon termination.

$$\frac{resty(md) = \gamma^Y \quad \gamma' <: \gamma \quad \gamma \text{ is not a string type}}{F, \gamma'^{X'} \cdot S, X, tr, J, R \vdash^{l}_{md} \texttt{return} \leadsto (J, R \cup \{X' \subseteq Y\})}$$

$$\frac{\begin{array}{c} resty(md) = \texttt{string}(str)^Y \\ \nexists str''.str = str'' \in R \text{ and } str'' \neq str' \end{array}}{\begin{array}{l} F, \texttt{string}(str')^{X'} \cdot S, X, tr, J, R \vdash^{l}_{md} \texttt{return} \leadsto \\ \qquad (J, R \cup \{str = str', Y \subseteq X'\}) \end{array}}$$

$$\frac{oldtr(tr \cdot l, J, l')}{F, S, X, tr, J, R \vdash^{l}_{md} \texttt{goto}\ l' \leadsto (J, R)}$$

$$\frac{\begin{array}{l} newtr(tr \cdot l, J, l') \quad J' = log(J, l', tr \cdot l) \\ F, S, X, tr \cdot l, J', R \vdash^{l'}_{md} blk(md, l') \leadsto (J'', R') \end{array}}{F, S, X, tr, J, R \vdash^{l}_{md} \texttt{goto}\ l' \leadsto (J'', R')}$$

$$\frac{F, C(o)^{\phi} \cdot S, X, tr, J, R \vdash^{l}_{md} B \leadsto (J', R') \quad o \text{ is fresh}}{F, S, X, tr, J, R \vdash^{l}_{md} \texttt{new}\ C \cdot B \leadsto (J', R')}$$

$$\frac{F, \texttt{int}^{P(class(md))} \cdot S, X, tr, J, R \vdash^{l}_{md} B \leadsto (J', R')}{F, S, X, tr, J, R \vdash^{l}_{md} \texttt{iconst}\ n \cdot B \leadsto (J', R')}$$

$$\frac{F, \texttt{string}(str)^{P(class(md))} \cdot S, X, tr, J, R \vdash^{l}_{md} B \leadsto (J', R')}{F, S, X, tr, J, R \vdash^{l}_{md} \texttt{ldc}\ str \cdot B \leadsto (J', R')}$$

$$\frac{F, S, X \cup P(class(md)), tr, J, R \vdash^{l}_{md} B \leadsto (J', R')}{F, S, X, tr, J, R \vdash^{l}_{md} \texttt{dopriv} \cdot B \leadsto (J', R')}$$

$$\frac{F[x \mapsto \delta], S, X, tr, J, R \vdash^{l}_{md} B \leadsto (J', R')}{F, \delta \cdot S, X, tr, J, R \vdash^{l}_{md} \texttt{store}\ x \cdot B \leadsto (J', R')}$$

$$\frac{F, F(x) \cdot S, X, tr, J, R \vdash^{l}_{md} B \leadsto (J', R')}{F, \delta \cdot S, X, tr, J, R \vdash^{l}_{md} \texttt{load}\ x \cdot B \leadsto (J', R')}$$

$$\frac{F, \delta \cdot \delta \cdot S, X, tr, J, R \vdash^{l}_{md} B \leadsto (J', R')}{F, \delta \cdot S, X, tr, J, R \vdash^{l}_{md} \texttt{dup} \cdot B \leadsto (J', R')}$$

$$\frac{oldtr(tr \cdot l, J, l') \quad F, S, X, tr, J, R \vdash^{l}_{md} B \leadsto (J', R')}{F, \texttt{int}^{Y} \cdot S, X, tr, J, R \vdash^{l}_{md} \texttt{ifeq}\ l' \cdot B \leadsto (J', R')}$$

$$\frac{\begin{array}{l} newtr(tr \cdot l, J, l') \quad J' = log(J, l', tr \cdot l) \\ F, S, X, tr \cdot l, J', R \vdash^{l'}_{md} blk(md, l') \leadsto (J'', R') \\ F, S, X, tr, J'', R' \vdash^{l}_{md} B \leadsto (J''', R'') \end{array}}{F, \texttt{int}^{Y} \cdot S, X, tr, J, R \vdash^{l}_{md} \texttt{ifeq}\ l' \cdot B \leadsto (J''', R'')}$$

$$\frac{\begin{array}{l} md' = (C, \texttt{<init>},\ T \xrightarrow[D]{\phi, Y} \texttt{void}^{Y'}) \quad |T| = |S'| \\ \rho = match(S', T) \\ R' = R \cup \{Y \subseteq X\} \cup \rho D \quad S'' = \texttt{update}(S, o, Y') \\ F' = \texttt{update}(F, o, Y') \quad F', S'', X, tr, J, R' \vdash^{l}_{md} B \leadsto (J', R'') \end{array}}{\begin{array}{l} F, S' \cdot C(o)^{\phi} \cdot S, X, tr, J, R \vdash^{l}_{md} \texttt{invokespecial}\ md' \cdot B \\ \qquad \leadsto (J', R'') \end{array}}$$

$$\frac{\begin{array}{l} |argty(md')| = |S'| \quad C <: class(md') \\ C' = srcclass(C, mname(md')) \quad MS = Prog(C') \\ (C', mname(md'),\ T \xrightarrow[D]{X', Y} \gamma^{Y'}) \in Dom(MS) \\ \rho = match(S', T) \cup \{X''/X'\} \\ R' = R \cup \{Y \subseteq X, X'' \subseteq X\} \cup \rho D \\ F, \gamma^{Y'} \cdot S, X, tr, J, R' \vdash^{l}_{md} B \leadsto (J', R'') \end{array}}{\begin{array}{l} F, S' \cdot C^{X''} \cdot S, X, tr, J, R \vdash^{l}_{md} \texttt{invokevirtual}\ md' \cdot B \\ \qquad \leadsto (J', R'') \end{array}}$$

Figure 9. Typing Rules of the JVM Calculus

$$
\begin{aligned}
&match(\epsilon, \epsilon) = \phi \\
&match(\texttt{int}^X \cdot S, \texttt{int}^Y \cdot T) = \{X/Y\} \cup match(S, T) \\
&match(\texttt{string}(str)^X \cdot S, \texttt{string}(str')^Y \cdot T) = \\
&\qquad\qquad\qquad \{X/Y, str/str'\} \cup match(S, T) \\
&match(C^X \cdot S, C'^Y \cdot T) = \{X/Y\} \cup match(S, T), \text{if } C <: C'
\end{aligned}
$$

Figure 10. Operator `match`

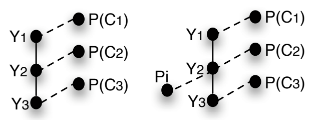

Figure 11. Two Constraints

4.6 An Example

Due to space limitation, only a brief example is explained to convey some intuitiveness of stack inspection enforced by this type system. A prototype implementation of our approach can be found at [6], where some examples are provided.

Suppose in a program the method md_1 calls md_2, which in turn calls md_3. Let $C_i (1 \leq i \leq 3)$ be the class defining md_i, and Y_i be the annotation on md_i's type for the permissions required to invoke it. In Figure 11, the dashed edges represent the inclusion relation between permissions from the initial constraint, and the solid edges are the inclusion relation built according to the definition of R' in the last two rules in Figure 9. The upper node of an edge is required to have more permissions than the lower node. The left constraint is for the case where no class is protected, and the right one is for the case where the method md_2 is protected by requiring the permissions Π (`Pi` in the figure) to access it. Hence, for the first case, the stack inspection succeeds since there is a solution in which every Y_i is empty, while for the second case, the program is secure only when $\Pi \subseteq P(C_1)$ and $\Pi \subseteq P(C_2)$ hold, that is, all code on the calling chain should have the permissions Π.

5 Related Work and Conclusion

As introduced before, there have been some work, such as [8, 5, 7, 1], proposed to address the weaknesses or security issues of stack inspection, but each of them always focuses on one aspect and ignores the other. This paper addresses all known weaknesses and security issues in a unified model. Morevoer, the security policies here are more flexible and permissions are more precise. The work [2] incorporates runtime stack inspection into information flow control, so that a single interface can provide information with different security levels for different callers.

Our first contribution is the approach to enhancing the usual stack inspection by checking whether values are used legally according to security policies. Our second contribution is the type system to verify whether a bytecode program satisfies the security property guaranteed by this enhanced mechanism. It uses different techniques to perform context-sensitive analysis at the method level and pathsenstive analysis at the code block level. This design choice is based on the observations that in JVM methods already have explicit type signatures and code blocks may be polymorphic. In addition, singleton types are used to make the analysis more accurate and detect the use of uninitialized objects. The type analysis in this work is based on the declared security policy, not program annotations from users.

6 Acknowledgments

Thank Prof. Masato Takeichi for his support and Prof. Zhenjiang Hu and other PSD members in Tokyo University for discussions. Also thank anonymous reviewers for their detailed comments.

References

[1] M. Abadi and C. Fournet. Access control based on execution history. In *The 10th Annual Network and Distributed System Security Symposium*, 2003.

[2] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005.

[3] S. N. Freund and J. C. Mitchell. A Type System for Object Initialization in the Java Bytecode Language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196 – 1250, 1999.

[4] L. Gong. *Inside Java 2 Platform Security*. Addison Wesley, 1999.

[5] T. Higuchi and A. Ohori. A Static Type System for JVM Access Control. *ACM Transactions on Programming Languages and Systems*. (accepted).

[6] D. Liu. Enhanced JVM Access Control. http://www.ipl.t.u-tokyo.ac.jp/˜ liu/enJAC.html.

[7] F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. *ACM Trans. Program. Lang. Syst.*, 27(2):344–382, 2005.

[8] C. Skalka and S. Smith. Static Enforcement of Security with Types. In *ACM International Conference on Functional Programming*, 2000.

[9] R. Stata and M. Abadi. A Type System for Java Bytecode Subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1), 1999.

[10] D. S. Wallach, A. Appel, and E. W. Felten. SAFKASI: A Security Mechanism for Language-based Systems. *ACM Transactions on Software Engineering and Methodology*, 9(4), 2000.