March 31, 2006

RT0782
Computer Science 19 pages

Research Report

Code Security in Transformed Java Bytecode

Michiaki Tatsubori, Akihiko Tozawa, Akira Koseki

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Code Security in Transformed Java Bytecode

Michiaki Tatsubori[1], Akihiko Tozawa[1], and Akira Koseki

IBM Tokyo Research Laboratory
{tazbori,atozawa,akoseki}@jp.ibm.com,

Abstract. Program code transformation is a useful technique for achieving language extensions like persistence of objects. However, a meta-level program may often introduce security holes unexpectedly through its code transformation, and result in breaking Java's code security model that base-level application programs assume in order to preserve their application-level security. This paper shows the problem of code security as broken by program transformations and explains the possibility of a security threat caused by it. Based on the security model of program transformations, we formalize a novel *action-based* security model, which describes a mechanism suitable for safely handling program transformations in Java. Also, this paper describes the design and implementation of our secure program transformation framework. The framework is built upon the standard Java virtual machine using a customized class loader. It checks and transforms bytecode of application classes when they are loaded. Experimental results show its overhead is acceptably small.

1 Introduction

Program transformation is a powerful and useful technique for providing a functional extension which is difficult to provide with ordinary object-oriented mechanisms like class inheritance [11]. Such extensions include object serialization [9], state persistency [12], remote objects [14], behavioral reflection [16, 13] and security mechanisms [2], for example. Even though direct bytecode manipulation is difficult to implement, recent research works [7, 10, 3, 6] provide toolkits or libraries for rewriting/generating bytecode easier. These efforts have made load-time bytecode transformations commodities.

However, in a system using program transformation, a *meta-level* program, who transforms application (*base-level*) programs, often breaks the code security that base-level application programs assume in order to preserve their application-level security. For example, a meta-level program may make a private member field of a class accessible to the code generated by the meta-level program on demand for probing the object status. In this case, misused or malicious base-level programs can also access the field that was originally declared private and not accessible. If the original base-level program is designed to preserve their own security depending that the private fields of a class are only

accessible from the class itself, its security is broken due to the transformations made by the meta-level program.

Even though the existing language environments such as *Java2 security* [8] provide a flexible, programmer-definable security mechanism, it is not sufficient to secure this situation. In a transformed program, what actually implements the meta-level intent is code fragments deployed across the entire application (base-level) program code. A meta-level program sometimes needs to allow these code fragments to run with the access rights of the meta-level domain like accessing the private resources. However, existing code security frameworks merely provide the package or class-based protection domains which cannot handle this. Programmers need to do complex coding tricks and error-prone tasks for writing a meta-level program which does not break the security while giving rich functions to base-level programs.

We think that to rigorously capture security requirements introduced by the use of program transformation systems, it is essential to have a new fine-grained security model. In this paper, we introduce a formal model of *action-based* security. Comparing to Java2 security, our model differs in the definition of units of programs which belong to a single domain, and to which a single security policy is applied. In our action-based security model, a security domain can be set to each atomic action, i.e., a minimum unit of the program. Thus, our model gives a sufficient solution to the security of program transformation in which we modify each atomic bytecode instruction of existing application classes.

We also give an algorithm to implement the proposed security model on top of the standard Java virtual machine. Though the idea of action-based security is simple, its implementation requires a careful design. For example, it is obviously not feasible to associate each atomic action with a protection domain object. Instead, we employ a load-time bytecode transformation technique to modify the security property of each atomic action. Though the design of this implementation is ingenious aiming at reducing runtime overhead as much as possible, the correctness of this implementation with respect to the model is guaranteed in terms of soundness and completeness theorems.

The contributions of the work presented in this paper are:

- The proposal of a new security model that can suitably treat a practical use of the program transformations in Java.
- A design of the security mechanism on top of a standard Java virtual machine. We use static checking of bytecode by a customized class loader. This does not imply any additional overhead over regular method execution since all the checks are done at load time.

The rest of this paper is organized as follows. In Section 2, we state the problem we address in this paper. Giving a few examples of program transfor-

mations at application platforms, we point out a security flaw caused by the introduction of program transformation. Section 3 describes our approach of security model extension and formally defines our security model using ABLP logic. We propose a new security mechanism for safely using program transformations in Java. Section 4 explains the design and implementation of our program transformation framework based on the proposed security model. Finally, after discussing related work in Section 5, we conclude the paper in Section 6.

2 Security Threat in Transformed Code

Program transformation is regarded as a powerful tool to add new functionalities to existing programs. We can think of many applications of the transformation, which can give us several benefits including high portability, productivity, reusability, and so on. For a typical example, we use an application platform where applications and libraries provided by some third parties run on top of the platform middleware. In such a platform, the middleware provides extended functionalities using transformation of application programs. Figure 1 illustrates the architecture of the platform. In the application platform, a middleware programmer gives a meta-level program that controls the entire system. Applications are provided typically through the network and given some functionalities, for example, in order to complete the application logic or to monitor the object status. Later we see that while the programmer can enjoy the benefit of the program transformation, the program has to be exposed to potential security vulnerability.

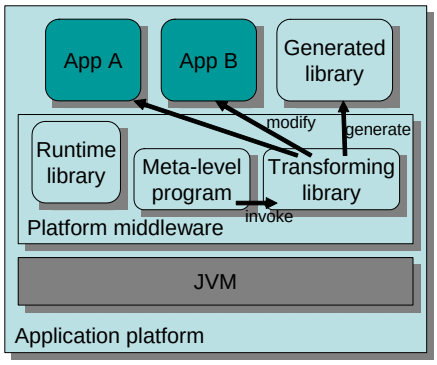

Fig. 1. A multi-application platform transforming application programs.

2.1 Program Transformation

Before showing actual examples of security problems, it is better to refer to the mechanism of program transformation and define our terminology. In this paper, we refer to the program transformation as mechanisms found in *Javassist* [6], which are powerful libraries for manipulating the *structure* of programs by the programs themselves. The modified programs are called the *base-level* programs, while the modifying programs are called *meta-level* programs. In object-oriented languages like Java, the atomic declared units of programs are classes and methods. A meta-level program inserts or modifies classes and methods, so that we obtain new functionalities for the original base-level programs. The common way of achieving these addition/modifications thought to be a compile-time or load-time approach. In this paper, we focus on a load-time mechanism as found in Javassist.

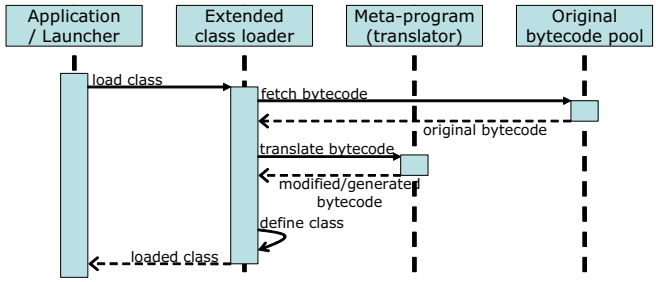

Fig. 2. The behavioral model of extended class loaders transforming bytecode of loaded classes.

Next we show the common program transformation mechanism model in such platforms with extended class loaders as illustrated in Figure 2. The extended class loader of an application platform is invoked from an application or an application launcher of the platform directly or indirectly through JVM as classes required to execute a program. It first fetches original bytecode of the specified class and then modifies the code using translators called by meta-level programs, or it can just generate required bytecode from scratch. The loader invokes its underlying JVM to make the binary representation executable and returns a class object for a class name given as a parameter when it is invoked.

2.2 A Privileged Logging Example

Here, we consider a server platform that records application services behavior. This platform tracks every method call within application services. The platform provides a `ServerRecord` class as follows.

```
package platform;
public class ServerRecord {
    public static void log(..) {
        AccessController.checkPermission(..);
        .. priviledged access to a file system..
    }
}
```

Since the record should not be accessed anonymously, it is protected using an access control mechanism in Java. This access controller for the logging method `log()` allows only the invocation originated from the server platform code without involving application services code.

Below is an example of application code. The middleware programmer wants to track and log any invocation on the method `serve()` in the application code `ServiceApp`.

```
package app2;
public class ServiceApp {
    void serve() {...
         ServerRecored.log();          // inserted code
    }
}
```

We assume we want to utilize this logging mechanism from several places in application services. Unfortunately, it would not help to simply insert `ServerRecored.log()` as illustrated. This is due to the call of `checkPermission(..)` inside the mehtod `log()`. The application class `ServiceApp` may not have a sufficient permission to pass the check.

Program Transformation A possible way of achieving this logging mechanism is to rewrite methods of application services and remove access check code in `log()`. Below is an example of resulting code. A middleware programmer wants to modify the method `serve()` in the base-level class `ServiceApp` and he has to modify the method `log()`.

```
package platform;
public class ServerRecord {
    public static void log(..) {
    //  AccessController.checkPermission(..); <- removed
        .. privileged access to a file system..
    }
}
```

An alternative way is, to give a special permission to the class `ServiceApp` so that it can freely access `log()` using privileged actions provided by the Java security mechanism.

```
package app2;
public class ServiceApp {
    void serve() {...
        AccessController.doPrivileged(          // added
            new PrivilegedAction() {            //
                public Object run() {           //
                    .. ServerRecord.log(..)  .. //
                }});                            //
    }
}
```

Attacks Exploiting Transformed Code After changing the programs for the sake of the middleware, it is now obvious that the `log()` faces a security threat of unsolicited misuses or malicious attacks as follows.

```
package app3;
public class Malicious {
    public void doEvil (..) {
        // able to put false information
        .. ServerRecord.log(....) ..
    }
}
```

In the alternative case, any code in `ServiceApp` would have the special permission. As a result, any malicious code in the class can do anything allowed using the special permission. For example, it can use privilege actions to illegally invoke `log()` from anywhere in the base-level class `ServiceApp`.

```
package app2;
public class ServiceApp {
    public void doEvil (..) {
        AccessController.doPrivileged(
            new PrivilegedAction() {
            public Object run() {
                // able to put false information
                ServerRecord.log(....);
            }
        });
    }
}
```

2.3 Problem Summary

We have seen the typical example of the use of program transformation which causes problems. This logging example reveals problems in conjunction with Java2's security mechanism, where what we would like to do was to relax an access restriction only for the method call code added by the middleware. However without exposing a security hole, this goal cannot be achieved neither by relaxing the access restriction of `ServerRecord.log()`, nor by strengthening the access right of `ServiceApp`.

The point of these problems is that a meta-level program is forced to give base-level programs unnecessary extra permissions that allows more than the intent of the meta-level program. We need an appropriate security model for handling this issue and an application platform framework which allows meta-level programs to avoid this problem.

3 Action-based Security

We have discussed the problem when a meta-level code programmer adds new functionalities to an application platform. The programmer may want to make the resources accessible to the meta-level code fragments, which are scattered across the program.

To solve this problem we introduce primitive APIs for program transformation to which our new semantics to secure resources are newly added. We call it *action-based security*. We give the formal description of the security properties for each primitive API. Then, we introduce our model of action-based security.

3.1 The Semantics of Program Transformation

We here focus on several transformations that are essentially needed for our purpose, i.e., building an application platform. We think it suffices to discuss the following four fundamental functions that affect the base-level code. Other functionalities are possibly used to implement the entire application logic, however, they impose no relevant issues for our interest on the subject.

addCall() Modify or add bytecode instructions in an existing method.
addMethod() Add a public method to an existing class.
replaceMethod() Replace an existing method by new one.
bePublic() Change a private method to be public.

For the simplicity, we do not discuss field modifications, such as transformations adding a new public field or changing a private field to be public. The same functionality can be obtained by adding a public accessory method.

We could actually implement a secured version of Javassist [6] based on the primitive operations above. Javassist is known to provide a powerful transformation functionality. Its APIs for class generation and class member modification can be implemented using `addMethod()`, `replaceMethod()` and `bePublic()`. The APIs for method body modification can be implemented using `addCall()`.

For those functions, we add new semantics regarding security properties of the affected method. Table 3.1 shows each API and defines the security properties of the affected method. Those have been carefully designed to facilitate

meta-level programmers to control the resources without jeopardizing security or complicating the coding work.

API	*security properties of affected method*		
	visibility from		*protection*
	meta level	*base level*	*domain*
`addMethod()`	visible	invisible	meta level
`replaceMethod()`, `addMethod()` (override)	visible	unchanged	meta level
`addCall()`	unchanged	unchanged	base level
`bePublic()`	visible	unchanged	base level

Table 1. How program transformation APIs affect the security properties of methods.

For example, a public method added by `addMethod()` will be executed with the access rights of the meta-level domain, and is accessible from the meta-level code but not from the base-level code, as long as not overriding an existing method. We treat method overriding differently, since it is natural to think adding an overriding method as the replace of an existing inherited method. In this case, the visibility of this method remains unchanged.

The underlying policy of our security design is that the meta-level code is given an *almighty pass* but is never exploited by the base-level code. If needed, the programmer can give the meta-level domain the stronger access rights than the base-level domain and modify the base-level classes for the sake of the meta-level code. Those are all safe.

One important feature of our framework is the support of `addCall()`. This API adds a new bytecode inside an existing method so that only the added code has the access rights of the meta-level domain. In code security, we split a program into each unit which is associated with a domain, and in which a single security policy is applied. In Java2, we can specify protection domains at the granularity of classes. However, this is not enough for the security of program transformation which modifies each atomic action of the program. We require a new *action-based* security in which a security property can be set to each atomic action, e.g., bytecode instruction.

3.2 A Formal Model for Action-based Security

We introduce a new abstract model for the action-based code security. Our model extracts the behaviors of program code and their security requirements in terms of *action*s and *target*s. Intuitively, an action corresponds to each atomic

Java bytecode instruction inside a method, whereas a target is a target of access control, e.g., a method to be invoked itself, or an access to the resource, etc.

Our model is based on ABLP logic [1] which is a modal logic designed to model authentication systems. ABLP logic is suitable to represent our model since it naturally represents an authentication system with concepts of delegation. Its modal operator *says* is defined over *principals*. In this logic, we do not deal only with the absolute truth value of a statement s, but we are also interested in a value of the statement X *says* s relative to a principal X. The code security is also considered as an authentication system where each code fragment and runtime object behaves as a principal.

The idea of Java2 security is briefly as follows.

- Each code belongs to a certain protection domain.
- Each domain is associated with a set of accessible resources.
- Whenever the code need to access the resource, we check domains of the code and its runtime context, i.e., the history of stack frames, to see if this access is allowed or not.

In modeling such a security, we first assume a certain execution environment in which we have a fixed set of resources T, domains D, actions A and frames F. We then introduce a set of axioms among these principals, according to the given environment.

3.3 Security Mechanism for Program Transformation

Based on the new model, we formulate our security framework. All we have to do is to add three fairly simple axioms (1), (2) and (3) in Figure 3. In defining axioms, we assume that program transformation APIs are used in the meta-level domain D. The use of APIs from D also should be secured by defining some actions and targets, but this does not explicitly appear in our model.

If D executes `bePublic`(T) then $D \Rightarrow T$	(1)
If D executes `addCall`($\cdots, A$) then A is privileged and $A \Rightarrow D$	(2)
If D executes either `addMethod`($T, [A_1, ...A_n]$) or `replaceMethod`($T, [A_1, ..., A_n]$) then $D \Rightarrow T$ and $A_i \Rightarrow D$ for $i \in 1..n$	(3)

Fig. 3. Security Model for Program Transformation

The API `bePublic`(T) adds an access to the method T from the meta-level domain. We introduce $D \Rightarrow T$ (Axiom (1)). It is easy to see the correspondence

of this axiom to Table 3.1 in Section 3. This axiom states that the use of the API changes the visibility of the method T from the domain D.

The API `addCall`(T, A) adds an action A to a method T. We can use A with the privilege of D. Thus we introduce $A \Rightarrow D$ (Axiom (2)). This means that we newly set the domain of an action A to D. We always treat A as a privileged call.

Lastly, the API `replaceMethod`$(T, [A_1, ...A_n])$ replaces an existing (public) method T whose content contains a set of actions $A_1, ..., A_n$. Similarly `addMethod`$(T, [A_1, ...A_n])$ adds a (public) method T. We allow an access to the method T from the meta-level domain D. Furthermore, for newly added actions A_i, similarly to `addCall`(), we associate them with the privilege of D. These are modeled by Axiom (3). When either we replace or override an existing method T, we assume that T continues to have an access from the application domain. In any case, the above $D \Rightarrow T$ just adds a new access to T from D, and does not modify existing access to T.

4 Implementation

We design an implementation of the proposed security model by extending a custom class-loader for program transformation. The approach to realize the security is itself based on the bytecode transformation technique, which controls the security property of each atomic bytecode instruction. Our algorithm realizes the secure semantics to the program transformation functions described so far. After giving the algorithm, we discuss its correctness lastly.

4.1 Assigning Protection Domains

First we assign two protection domains to groups of classes. This process is same as the one for normal application platforms. We use the normal protection domain mechanism in Java so the units of protection domain assignment are classes or packages but not methods here.

Assigned protection domains are:

An application-level protection domain Each application running on the transformational platform middleware belongs to this domain. The bytecode of programs in this domain is transformed by a platform middleware, which belongs to another domain.

A middleware-level protection domain Components of transformational platform middleware belong to this domain. They include meta-level program which translates bytecode, and the runtime library provided by the platform. They should be able to put permission checking code (e.g. `AccessControl.checkPermission()` in the Java2 security.) in order to deny undesirable accesses from application code.

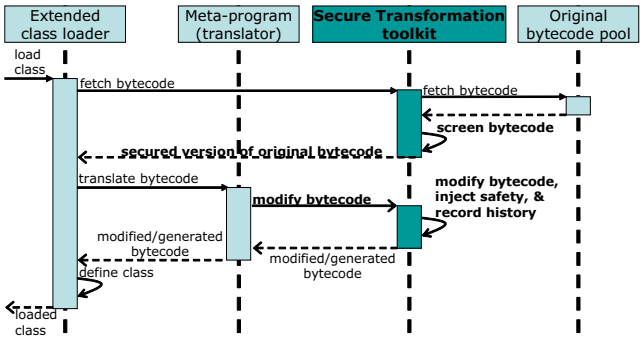

Fig. 4. Secured bytecode transformation

4.2 Secure Transformation Toolkit

To add a security functionality, we extend the customized class loader for the load-time program transformation. This class loader works for classes in the application domain. The *secure* bytecode transformation behaves as shown in Figure 4. Let us compare the new mechanism with the original one (Figure 2) explained in Section 2. We can see that we now have a module called *Secure Transformation Toolkit*.

The toolkit processes the input bytecode through the following two phases.

- In the *bytecode screening* phase, the toolkit checks and modifies any fetched bytecode, according to the original bytecode pool and a *modification history*. This guarantee that the pool does not contain any *illegal* method calls in it.
- For the *security injection* phase, we provide secured versions of transformation APIs, which meta-level applications use in replace with the original APIs. The new APIs inject some code in addition to the original transformation, so that the required security property is guaranteed. Also, it records bytecode modifications by the translators of the platform middleware in the *modification history* to help static checks for fetched bytecode.

Here the *modification history* is a table structure which records a set of pairs of (1) the information of the modified method, and (2) the type of the API used to modify the method. For the API type, it suffices to record either `BEPUBLIC` or `ADDMETHOD` for our particular algorithm.

Bytecode Screening The aim of bytecode screening is to enforce the requirement that “meta-level methods should not be visible from application domain”.

Here meta-level methods mean the methods either added by `addMethod()`, or made public by `bePublic()`.

The screening process is implemented by a procedure (see method `screen()` in Appendix for more details), in which we replace all call instructions (i.e., invoke instructions of JVML) in the input bytecode with the result of `screenCall()`. The `screenCall()` method actually replaces an illegal method access to added or made-public methods with an instruction to throw an exception. Thus when the execution reaches that point, we get a `MetaLevelAccessException` thrown rather than wrongly calling a meta-level method.

It is possible that the target method of the call instruction is not available as its defining class is not loaded, i.e., program transformation is not yet performed on that class. The screening process fetches the target class to see if the target method is available. If it is available and accessible then it is okay, because `addMethod()` and `bePublic()` only affect non-existing or inaccessible methods. Fortunately, we need this fetch just one-level deep rather than recursing it, since we just need a method information and not requiring the screening of the fetched class. Thus the cost of fetch is not so large.

Security Injection The security injection is implemented as extensions to the original translator APIs (Figure 5). These extensions ensure the requirement "the modified programs have the privilege of the meta-level domain".

In APIs `addCall()`, `addMethod()` and `replaceMethod()`, we use `getPrivilegedProxy()` (Lines 2, 15, 24), which replaces a call to a certain method with a call in the proxy class in the meta-level domain, which wraps the actual method call as a `PrivilegedAction.run()`. Thus the actual method call is performed with the access right of the meta-level domain.

The process of `replaceMethod()` is slightly complicated because a replaced method (and an overriding method (cf. Section 3)) is not always called from the meta-level domain. If the replaced method is called from application domain, since its stack frame contains an application code, we need to associate each code with the application domain (which is the default behavior, as the new method actually belongs to the application domain). However, when the code contains a privileged call, it must be treated as a call from the meta-level domain (Line 22–26 of Figure 5). Otherwise, the new method is called from the meta-level domain, in which case we need to associate any call with the meta-level domain (Line 29–32).

To solve this requirement, we introduce two duplicate methods, T_{meta} and T_{base}, for the same input bytecode. The `if` statement in Line 35 is responsible for detecting whether the caller context is meta-level or not to switch which one of methods to call. This detection is actually implemented by exploiting

```
1  void addCall(Context c, CallInstruction A) {
2      super.addCall(c, getPrivilegedProxy(A));
3  }

4  void bePublic(Method T) {
5      add (T, BEPUBLIC) to the modification history
6      super.bePublic(T);
7  }

8  void addMethod(Method T, Instruction[] A) {
9      if (T overrides an existing method)
10         this.replaceMethod(T, A);
11     add (T, ADDMETHOD) to the modification history
12     Instruction[] A′;
13     for (int k = 0; k < A.length; k++)
14         if (A[k] instanceof CallInstruction)
15             A′[k] = getPrivilegedProxy(A[k]);
16         else
17             A′[k] = A[k];
18     super.addMethod(T, A′);
19 }



20 void replaceMethod(Method T, Instruction[] A) {
21     Instruction[] A′;
22     for (int k = 0; k < A.length; k++)
23         if (A[k] is a privileged call)
24             A′[k] = getPrivilegedProxy(A[k]);
25         else
26             A′[k] = A[k];
27     Instruction[] A″;
28     for (int k = 0; k < A.length; k++)
29         if (A[k] instanceof CallInstruction)
30             A″[k] = getPrivilegedProxy(A[k]);
31         else
32             A″[k] = A[k];
33     Method T_meta = meta version of T;
34     Method T_base = base version of T;
35     super.replaceMethod(T, new Instruction[] {
36           if (the call stack has the meta-privilege)
37               invoke T_meta
38           else
39               invoke T_base
40           });
41     super.addMethod(T_base, A′);
42     super.addMethod(T_meta, A″);
43 }
```

Fig. 5. The algorithms for security injection

Java's security manager, and is thus costly because we need to look at the call stack. However, we can use some optimization techniques here. When we use `getPrivilegedProxy()` for the replaced method, since we know that the call is always from meta-level, we directly set the target method to T_{meta}. Similarly, when we know that the context is always application-level, we directly call T_{base}.

The secured translator is also responsible for recording modified methods in the modification history. This is written in Lines 5 and 11.

Limitation Since the implementation relys on the static analysis of bytecode, it cannot detect illegal method calls through the Reflection API in Java. We must deprive applications of their permission to use the Reflection API in order to prohibit application code from reflectively calling originally invisible methods, such as methods newly added by a platform middleware.

4.3 Overhead Measurement

We measured the overhead of program transformations applied to application software available in the SPEC JVM98 benchmarks [1]. The details of the experiments will be given in Appendix. We observed slight overheads of security injection in benchmark applications, due to static bytecode checking in the implemented secure transformation toolkit. Our implementation does not impose any overhead on normal execution of methods unless code accesses meta-level code illegally.

5 Related Work

Security Model One of the earliest works on Java security is that by Wallach and Felten [15], which explains the meaning of code security in Java using ABLP logic [1]. Indeed, their work precedes the notions of permissions and protection domains introduced into Java by Gong [8] in 1999. Wallach and Felten focus on the security mechanism based on `enablePrivilege`() and `disablePrivilege`(), which is slightly different from Gong's mechanism based on `doPrivilege`(). Our formalization of the security model is inspired by the Wallach-Felten's work, but we completely reformulate and simplify their model to do with Gong's mechanism. More importantly, the crucial difference of our model is the granularity of units to which we can set protection domains. In our model, the minimum unit is each atomic action, and thus providing a very fine-grained security suitable in securing program transformation. Also our model handles member accessibility (e.g., `private`, `public`, ..) which was not modeled by Wallach and Felten.

[1] http://www.spec.org/jvm98/

Secure Reflection Caromel et al. investigated the security issues raised by the use of meta-programming systems with Java [5]. They provided a set of rules for combining together the permissions associated with the different protection domains of a typical reflective-component-based application. Also, they proposed an approach for making meta-level code transparent to base-level code in the context of a simple proxy-based runtime meta-object protocol [4], using the standard Java security mechanism. Our work, however, is about program transformation rather than a simple proxy. Simply applying the regular security model does not work for the program transformation. Instead, we need to extend the security model.

6 Summary

In return for its power, the introduction of program transformations often breaks Java's code security model that the base-level application assumes in order to preserve their application-level security. This paper points out a problem of code security as broken by program transformations and explains a possible security threat caused by it. We propose an extended access control model suitable for safely using program transformations at class loading time in Java. We presented the formal model of the proposed extension and also give a design and implementation. We employed static code checking in a customized class loader on a standard Java virtual machine.

References

1. M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.
2. M. Ancona, W. Cazzola, and E. B. Fernandez. Reflective authorization systems: Possibilities, benefits, and drawbacks. In *Secure Internet Programming 1999*, LNCS 1603, pages 35–49. Springer-Verlag, 1999.
3. G. Back. Datascript – a specification and scripting languages for binary data. In D. Batory, C. Consel, and W. Taha, editors, *Proc. of Generative Programming and Component Engineering (GPCE 2002)*, LNCS 2487, pages 66–77, Pittsburgh, PA, USA, October 2002. Springer-Verlag.
4. D. Caromel, F. Huet, and J. Vayssière. A simple security-aware MOP for Java. In A. Yonezawa and S. Matsuoka, editors, *Reflection 2001*, LNCS 2192, pages 118–125, Kyoto Japan, September 2001. Springer-Verlag.
5. D. Caromel and J. Vayssière. Reflections on mops, components, and java security. In L. Knudsen, editor, *ECOOP 2001 - Object Oriented Programming*, LNCS 2072, pages 256–274, Budapest, Hungary, June 2001. Springer-Verlag.
6. S. Chiba. Load-time structural reflection in Java. In *ECOOP 2000 - Object Oriented Programming*, LNCS 1850, pages 313–336, Sophia Antipolis and Cannes, France, June 2000. Springer-Verlag.

7. M. Dahm. Byte code engineering with the javaclass api. Techincal Report B-17-98, Institut für Informatik, Freie Universität Berlin, Berlin, Germany, July 1999.
8. L. Gong. *Inside Java2 Platform Security*. The Java Series. Addison-Wesley Longman, 1999.
9. M.-O. Killijian, J.-C. Ruiz-Garcia, and J.-C. Fabre. Portable serialization of CORBA objects: a reflective approach. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002*, SIGPLAN Notices vol.37, pages 68–82, Seattle, Washington, USA, November 2002. ACM.
10. G. Kniesel, P. Costanza, and M. Austermann". JMangler - a framework for load-time transformation of java class files. In *Proc. of IEEE Workshop on Source Code Analysis and Manipulation*. IEEE, 2001. none.
11. C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.
12. A. Marquez, J. N. Zigman, and S. Blackburn. Fast portable orthogonally persistent java. *Software — Practice and Experience*, 30(4):449–479, April 2000.
13. E. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA 2003*, number 10 in SIGPLAN Notices vol.38, pages 27–46, Anaheim, Carifornia, USA, October 2003. ACM.
14. M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of "legacy" Java software. In L. Knudsen, editor, *ECOOP 2001 - Object Oriented Programming*, LNCS 2072, pages 236–255, Budapest, Hungary, June 2001. Springer-Verlag.
15. D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *Proceedings of 1998 IEEE Symposium on Security and Privacy (S&P'98)*. IEEE, 1998.
16. I. Welch and R. Stroud. From Dalang to Kava — the evolution of a reflective Java extension. In *Proceedings of Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, LNCS 1616, pages 2–21, Saint-Malo, France, July 1999. Springer.

A Appendix: Secure Transformation Alogrithms

Figure 6 describes the algorithm for scrrening bytecode. The screening process is implemented by the method `screen()`, in which we replace all call instructions (i.e., invoke instructions of JVML) in the input bytecode with the result of `screenCall()`. The `screenCall()` method actually replaces an illegal method access to added or made-public methods with an instruction to throw an exception. Thus when the execution reaches that point, we get a `MetaLevelAccessException` thrown rather than wrongly calling a meta-level method.

It is possible that the target method of the call instruction is not available as its defining class is not loaded, i.e., program transformation is not yet performed on that class. Lines 25-29 handles this case by fetching the target class to see if the target method is available.

B Appendix: Experimental Results of Overhead Measurement

We measured the overhead of program transformations applied to application software available in the SPEC JVM98 benchmarks [2]. We prepared 4 meta-level programs of:

[2] http://www.spec.org/jvm98/

```
1  Instruction[] screen(Instruction[] A) {
2      Instruction[] A′;
3      for (int k = 0; k < A.length; k++)
4          if (A[k] instanceof CallInstruction)
5              A′[k] = screenCall(A[k]);
6          else
7              A′[k] = A[k];
8      return A′;
9  }

10 Instruction screenCall(CallInstruction A) {
11     Instruction A′;
12     String C = A.getTargetClass();
13     Method T = A.getMethod();
14     if (the class C is already loaded)
15         if ((T, ADDMETHOD) is in the history)
16             A′ = getThrowingInstruction(A);
17         else if ((T, BEPUBLIC) is in the history)
18             if (original T is invisible)
19                 A′ = getThrowingInstruction(A);
20             else
21                 A′ = A;
22         else
23             A′ = A;
24     else
25         fetch bytecode of C
26         if (T doesn't exist)
27             A′ = getThrowingInstruction(A);
28         else
29             A′ = A;
30     return A′;
31 }
```

Fig. 6. The bytecode screening algorithm. It looks up illegal method calls to replace them with exception-throwing code.

- doing nothing (*XX-empty*)
- adding a simple method to every loaded class (*XX-onemeth*)
- adding a method containing a call to a privileged operation to every loaded class (*XX-privmeth*), and
- adding 100 methods to a single loaded class (*XX-100meth*).

We applied each meta-level program with the normal program transformations with Javassist (*refl-XX*) to the SPEC's applications, and also did with the secured program transformations with an extended Javassist implemented using the mechanism proposed in this paper (*secure-XX*).

We measured the elapsed time from the start to the end of application execution including the start up time of its underlying JVM. The start-up time of JVM includes the loading time for some classes. In addition to 8 types of application execution, we measured normal execution time of application (*normal*).

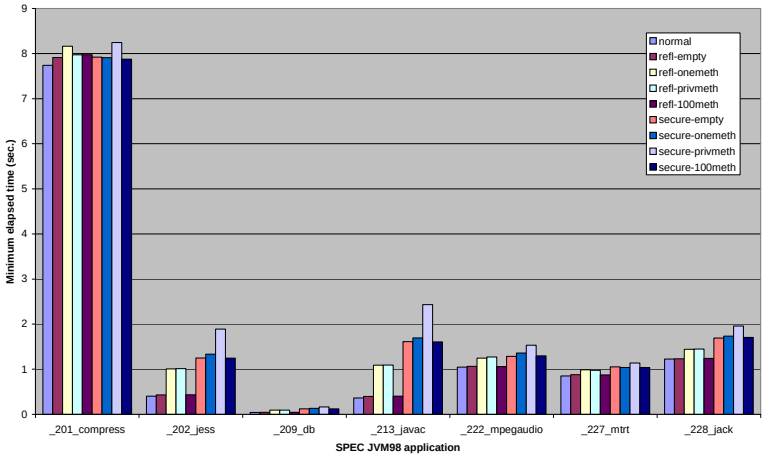

Fig. 7. Elapsed time for overall application execution (interpreter, input size=1)

We focused on the program startup regime of benchmark program execution. During program startup, an execution process includes class loading and initializing. After a while, the program reaches a steady state. We evaluate the startup regime by timing the first run of the SPECjvm98 benchmarks with the size 1 (small) inputs, on a JVM with interpreter-mode. We also the steady state by timing the benchmarks with the size 100 (large) inputs, on a JVM with server-mode. We run the application benchmarks 3 times and took the minimum for each.

Figure 7 and Figure 8 show our experimental results. These experiments were run on the Sun Java 2 Runtime Environment, Standard Edition, version 1.4.2 (HotSpot Server VM) / Linux 2.4.18-14 / IBM IntelliStation M Pro with a single Pentium(R) 4 Processor 2.4GHz and 2GB memory.

The graph showed slight overhead because of static bytecode checking. Our implementation does not impose any overhead on normal execution of methods unless code accesses meta-level code illegally. A benchmark with secured transformation mechanism sometimes ran faster than non-secured one. This might be of the just-in-time (JIT) compiler behavior or in a possible error range. Since the bytecode size of instrumented methods got larger than ones without security injection, they interfere with optimal JIT compilation on the methods. The cost of optimal JIT compilation sometimes does not pay before enough execution repetition of the optimized methods.

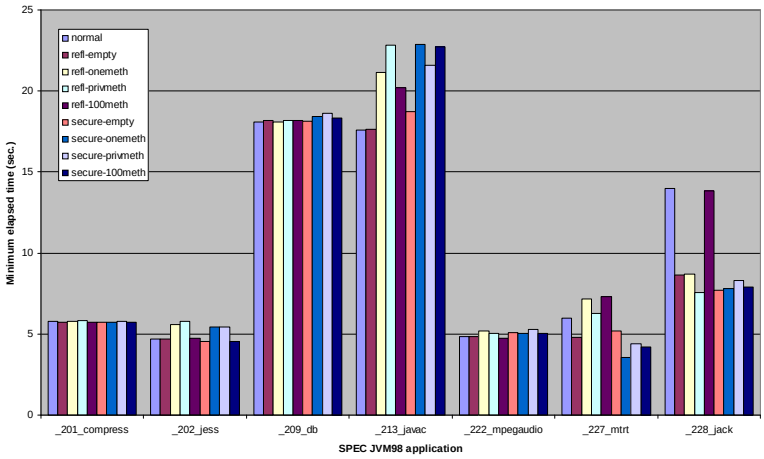

Fig. 8. Elapsed time for overall application execution (JIT, input size=100)