On Instrumenting Obfuscated Java Bytecode with Aspects
Kung Chen and Ju-Bing Chen
Department of Computer Science,
National Chengchi University, Taiwan
{chenk, g9405}@cs.nccu.edu.tw

ABSTRACT
Code obfuscators are widely used tools for protecting commercial Java software. Advanced obfuscation techniques make decompiled Java programs not re-compilable, thus greatly raising the barrier of instrumenting Java bytecode for malicious purpose. However, we have found that the aspect-oriented programming language AspectJ can be abused to overcome advanced code obfuscation and to modify obfuscated Java software effectively using its bytecode instrumentation mechanism. This paper describes such issues and reports our experiment results. We argue that the simplicity and very low cost of such malicious aspects make them worth wider attention from the Java and AspectJ community.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features; K6.5 [Security and Protection]: unauthorized access

General Terms
Security, Languages

Keywords

1. INTRODUCTION
In the Java developer community, it is well-known that, as much of the source-level information is retained in the bytecode, decompilation of Java programs is much easier than with traditional native code [14]. Hence we see many decompilers of Java available freely or commercially. In the hands of a malicious user, a decompiler becomes an essential tool for performing reverse-engineering attacks on Java bytecode. For example, a decompiler can be used to extract proprietary algorithms from compiled Java programs; or it can assist in subverting protection checks of commercial Java packages and then distributing unauthorized copies. As a result, commercial Java software developers take seriously the threat posed by such decompilation-based reverse-engineering attacks.

Currently, the most widely used weapon to protect Java bytecode against decompilation is code obfuscator [6][12]. Almost all Java obfuscators use the techniques of name obfuscation that change variable and type names into meaningless names to make decompiled programs unreadable. Some of them also replace user-defined identifiers with reserved words or illegal identifiers [4] so that the decompiled programs are not re-compilable. There are also more advanced obfuscators that perform control flow obfuscation [11], which make subtle changes to the bytecode that obscure the control flow without changing what the code does at runtime. Usually the control flow related statements, such as selection and looping constructs, are modified so that they no longer have a direct Java source code equivalent. Most decompilers are forced to insert a series of labels and illegal goto statements into the source code they produce, thus leaving the decompiled programs not re-compilable. Furthermore, string encryption is often used together with code obfuscation to hide any critical information that may be revealed by string literals.

Such results have important practical implications for Java software protection. First, these advanced obfuscators make common Java debuggers useless for tracing the decompiled programs to help identify a program’s vulnerabilities, as debuggers need the source code for stepping through a program’s execution. Second, they also make unauthorized code changes a rather difficult task since it is hard to fix the compilation errors in the decompiled programs, not to say modify them for malicious purposes. It is conceivable that determined attackers can still develop some sophisticated bytecode rewriting tools to exploit the information included in the Java class files and perform reverse-engineering attacks. Yet that would require in-depth knowledge of the Java bytecode format and cryptography, and the attack process may take quite some time that makes it not cost-effective to crack the target. Hence, in practice, code obfuscators are widely used for Java software protection.

Nevertheless, we shall illustrate in this paper that malicious users with rudimentary knowledge of Java bytecode and cryptography can exploit the join point model and code instrumentation mechanism of AspectJ [2] to render code obfuscation and string encryption ineffective for some intended protection. Furthermore, such exploits are accomplished through bytecode level instrumentation, which means that we do not need to get the source code of the underlying target to conduct them. This in turn implies that it is fruitless to defend against aspect instrumentation by making the decompiled Java programs not re-compilable using techniques such as control obfuscation, for one can run the instrumented bytecode directly without having to re-compile them.

To test the effectiveness of such an approach, we have conducted an experiment of aspect-based instrumentation on several obfuscated commercial Java software packages using AspectJ. We focus on only stand alone Java applications and select six commercial packages as our targets from the Google directory, including three commercial obfuscators since it is very common...
to demonstrate the capability of an obfuscator is to apply bytecode obfuscation to itself. In this paper we shall describe this approach in detail and report our experiment results.

In short, our attacks are fairly straightforward yet cost-effective. The experiment took us less than four weeks to finish; only one target escapes our attacks. We have only very rudimentary knowledge of Java bytecode and cryptography. We did not use any special tools; all the tools employed are freely available through the Internet. Besides, the techniques we used are all but direct applications of AspectJ’s standard mechanisms. Admittedly, our experiment is not comprehensive and our results may not be conclusive. However, it is due to the simplicity and very low cost for such attacks that we believe the related issues merit wider attention from the Java and aspect-oriented community.

The paper is outlined as follows. Section 2 presents the background of the aspect-based instrumentation and describes related work. Section 3 describes the instrumentation method and techniques we used. Our experiment results are presented in Section 4, followed by a discussion on why it is difficult to defend such malicious aspects and related issues. Section 6 concludes.

2. BACKGROUND AND RELATED WORK
This section outlines the basics of aspect-oriented programming and the relevant features of AspectJ, and describes related work.

2.1 AOP and AspectJ
Aspect-oriented programming (AOP) aims at modularizing concerns such as profiling and security that crosscut the components of a software system [9]. In AOP, a program consists of many functional modules and some aspects that encapsulate the crosscutting concerns. An aspect module provides two kinds of specifications: Pointcut, comprising a set of well-defined points in the execution of a program, designates when and where to crosscut other modules; and advice, which is a piece of code, that will be executed when a pointcut is reached. The complete program behavior is derived by some novel ways of composing functional modules and aspects according to the specifications given within the aspects. This is called weaving in AOP. Weaving results in the behavior of those functional modules impacted by aspects being modified accordingly.

AspectJ [2] is a seamless aspect-oriented extension to the Java programming language. It provides a rich pointcut language through a powerful join point model. Typical join points in AspectJ are method call/execution and field access. For example, the following pointcut, setField, refers to all the points in the class C of package p where a field of String type is set. It will also grab the value to set and bind it to a parameter named sval.

\[
\text{pointcut setField( String sval ) ; set( String p.C.* ) & args( sval ) ;}
\]

There are three kinds of advice in AspectJ: before, after, and around. The before advice and the after advice are executed before and after the intercepted method, respectively. The case for the around advice is more subtle. Inside the around advice, we can choose to return the intercepted method by calling the special built-in method proceed(), or simply bypass its execution. Advice can get access to much program information exposed by its binding pointcuts. First, the parameter values captured by a pointcut can be passed to its advice like method arguments. Besides, much of other context information of an advice’s join point is encapsulated in an object bound to a special variable called thisJoinPoint, and made available in the body of an advice.

The following code fragment sketches two partial aspects in AspectJ which motivate our approach to instrumenting a Java program for tracing and modifying its behavior.

Listing 1. Example Aspects of Tracing and Modifying

\[
1 \text{public aspect Tracing } \\
2 \quad \text{private Tracer tracer = new Tracer();} \\
3 \quad \text{pointcut mCall()::call("com.a.b.Check."*(..));} \\
4 \quad \text{before() : mCall()::/trace calling sequence} \\
5 \quad \text{tracer.addLog( "method call", thisJoinPoint );} \\
6 \quad \text{pointcut showAnyDialogCall() : // Swing UI} \\
7 \quad \text{call(*..*.show*Dialog(..)) &&} \\
8 \quad \text{within( com.a.b );} \\
9 \quad \text{before() : showAnyDialogCall() { } } \\
10 \quad \text{if ( tracer.isShowMessageLog() ) } \\
11 \quad \text{tracer.addLog("show * Dialog call",} \\
12 \quad \text{thisJoinPoint );} \\
13 \quad \} \ldots \\
14 \text{public aspect Modify } \\
15 \quad \text{boolean around() : // anonymous pointcut} \\
16 \quad \text{execution(boolean com.a.b.Check.method())} \\
17 \quad \{ //skip the check and always return true} \\
18 \quad \text{return true; } \\
19 \quad \} \ldots \\
20 \]

The Tracing aspect includes two pairs of pointcut and advice and a tracer object which provides the logging facility and tells the aspect what should be logged. The first pair (line 3–6) simply logs any calls to methods in the class of com.a.b.Check, while the second pair (line 7–12) logs only calls to dialog-related UI methods that originate from the package com.a.b. They are used to generate runtime traces that help locating specific code modules targeted for modification. Note that the within pointcut restricts the set of join points to a certain scope and the thisJoinPoint variable prints the signature of the caller. The around advice (line 15–19) in the Modify aspect is bound to an anonymous pointcut that corresponds to the execution of the target method. It skips the method and simply return true to fool the caller of the method into believing that targeted method has successfully executed, thus modifying the behavior of the target.

In AspectJ, the weave tool, ajc, is integrated with the Java compiler. It compiles AspectJ and Java language files, weaving aspects as designated by the pointcuts to produce class files compliant with any Java VM (1.1 or later) [7]. Since version 1.1, ajc also supports bytecode weaving, i.e., it can insert aspects into Java class files directly without the presence of source code.

2.2 Related Work
In his popular book [8], Kalinovsky discussed many techniques for hacking as well as protecting Java software, yet the potential abuse of AspectJ is not covered. Besides free or commercial Java bytecode obfuscators, some academic researchers also published several papers on this topic. Collberg et al [6] wrote a comprehensive survey of recent work on program obfuscation. Low [11] gave a general introduction on obfuscation techniques for Java, and his master thesis [12] presented some techniques for control flow obfuscation. Few commercial Java obfuscators have
included control obfuscation in their obfuscating transformations; we successfully defeated on one such commercial Java obfuscator. Recently, Chan and Yang [4] proposed an advanced obfuscation algorithm and some techniques that make the decompiled program difficult to understand by reusing the same identifier as often as possible. Among others, they modify the bytecode by replacing user-defined identifiers with reserved words or illegal identifiers so that decompiled programs are not re-compilable while the obfuscated bytecode works properly. However, Cimato et al [5] shows that their obfuscation techniques can be made ineffective by some intelligent modification of identifiers in Java bytecode.

Miller et al [13] showed that it is possible to overcome similar software protection in a commercial product by using dynamic instrumentation tools both for understanding the way the protection works for that program and avoiding the execution of the protection functions. However, they discussed the approach used for attacking a specific piece of software written in C, rather than presenting a general strategy for fulfilling this kind of attacks.

3. METHOD AND TECHNIQUES
This section presents the method and techniques we used to develop the aspect-based instrumentation for malicious purposes.

3.1 Overview of the Method
We focus on stand alone commercial Java software that requires input of some proper data from its user to launch the software. Inside these software packages there must be some code module for decoding and checking the data the user entered and reporting its correctness. Usually the decoding steps are rather complicated and are almost always obfuscated. Yet, for our purpose, we do not need to figure out the details of the decoding process. Instead, our task consists of two steps: locating the methods where the checking result is reported and then attempting to overcome it by faking a success report, as exemplified by the Modify aspect defined in Listing 1.

However, due to code obfuscation, it is usually very difficult to spot exactly where the check result is reported. We still need a Java decompiler to make our work easier. But, as described earlier, leading-edge obfuscators outperform decompilers in many aspects, so decompilers can provide only limited clues in locating the target methods. The missing information, on the other hand, can be collected effectively by properly designed tracing aspects using the powerful pointcut and advice mechanisms of AspectJ.

We developed a smart tracer in AspectJ to trace the program flow and log proper information of the instrumentation target for human inspection. Clearly, we must do the tracing more efficiently than simply tracing the underlying program’s execution sequence all the way from the main method until it stops due to the wrong data entered. Indeed, we take full advantage of the rich join point model of AspectJ to supply control flow based tracing as well as data flow based tracing that provide critical clues for finding the Java method that reports the data decoding result. On the one hand, control flow based tracing supplies the calling sequence of Java methods within a designated tracing scope using the method call join points. On the other hand, data flow based tracing employs field access join points to track where decoding and checking related strings are accessed. Such information often plays a key role in narrowing down the scope of control flow tracing. In fact, these two kinds of runtime traces are complementary to each other, for the control based tracing can also provide clues to restrict the scope of data flow based tracing. Together, they greatly simplify the tracing task.

Compared to the tracing step, the second step of modifying check results with aspects is easier. The major work is to replace the method that reports the check result with a piece of around advice that always returns a forged success result.

3.2 Locating the Methods for Instrumentation
We now present the set of pointcuts and advice we devised to trace an obfuscated program for locating the data checking method for instrumentation.

3.2.1 Control Flow Based Tracing
The main purpose of control flow based tracing is to generate a method call log that will help us find the data checking code. We used both top-down and bottom-up tracing. Top-down tracing follows the execution sequence of the underlying program and is the most common approach when we use a debugger to trace a program. In AspectJ, this can be achieved through a catch-all pointcut and a plain logging advice as follows.

```java
before() : call( * *..*.*(..) ) { 
    tracer.addLog( "method call", thisJoinPoint ); 
    // log arguments using thisJoinPint.getArgs() 
}
```

Obviously, this way of tracing is extremely inefficient. We better acquire as much information as we can to restrict the scope of tracing. If the target program is only partially obfuscated, we may tell from the package names where data checks are done and narrow down the trace scope to a particular package and get a much smaller log for inspection. For example, the mCall() pointcut (line 3) defined in Listing 1 is simplified from a real case we encountered in our studies. For that case, we can limit the trace scope to methods in the package whose name includes “check.”

In contrast, bottom-up tracing starts from where some key methods are invoked and attempts to trace back to our target point. Along the way, it provides important clues for us to narrow down the tracing scope. The idea is as follows. Since the user of a commercial package will be prompted for data input, if we discover where the dialog-related methods are called, it is very likely that the data checking code is nearby. Therefore, we choose a few core UI methods from Java Swing and Java AWT library and define the proper pointcut and advice pairs to log where such methods are called. The showAnyDialogCall() pointcut defined in Listing 1 (line 7–8) is one such example. For another example, we picked the setVisible( boolean ) method which is essential to many UI components in Java, and defined the following pointcut/advice pair to log where it is called.

```java
before() : call( * *..*..setVisible(..) ) { 
    if ( tracer.isSetVisibleLog() ) //should logged? 
        tracer.addLog( "setVisible: ",thisJoinPoint); 
}
```

It turns out that such bottom-up tracing not only reduce the size of the log significantly, but also is very effective in discovering the data checking code.
3.2.2 Data Flow Based Tracing
The data flow based tracing concerns where data-checking related strings are accessed; it is complementary to the control flow based tracing. Obviously, during the data checking process, some strings such as “Please enter the proper data …” or “Wrong data input” will be displayed to the user. If we know which methods access these strings, we have a more focused scope for control flow based tracing. Hence we defined several pointcut and advice pairs to log where such strings are referenced. However, if such string literals are encrypted, then we have to spend more efforts to decrypt them first; we shall discuss this issue later.

Since most string literals are stored as class fields, we can use the field access join points of AspectJ to conduct data flow tracing. The following after advice template illustrates how we log all references to some candidate strings. It can be customized by replacing the candidate string to match by other similar strings and be further restricted to references originate from a particular scope by appending a within pointcut.

```java
after() returning( String s ) :
    get( String ".*");
    // && within( aType ) optional
    if ( contains( s, "DataChecking" ) )
      // or other similar strings
      tracer.addLog( "String " + s,
    thisEnclosingJoinPointStaticPart );
}
```

Note that, for advice on field access, the thisJoinPoint object refers to the access point itself. Since we are interested in which the candidate strings are referenced, we need to use thisEnclosingJoinPointStaticPart to get the enclosing join point. Such information often provides us concrete clues about our target.

Another very effective data flow based tracing is targeting at the specific data that the user entered. It is perfectly fine that we do not know the correct data; all we need is to trace where the invalid data entered are used. The idea is that the program under attack will check if the data entered is correct, and it is very likely that the invalid data will be set into a class field before the checking. Since the code segment that sets the data entered is definitely very close to the method that reports the check result, knowing the point where it is set will surely greatly helps us find the target method. Here is the template advice we used.

```java
before( String s ) ; set( String ".*");
    // && args( s ) // value to set
    // && within( aType ) optional
    if ( s.compareTo( DataWeEntered ) )
      tracer.addLog( "Data set"+s, thisJoinPoint );
}
```

The advice will log in which class and to which field the data entered is set through the thisJoinPoint object. For example, if we get “String DataCheck.udata” printed in the log, we know that the data we entered is set to the field, udata, defined in the class, DataCheck. Once we know the specific field holding the data entered, we can proceed to trace where this field is used to pinpoint the data checking method.

3.2.3 Handling Encrypted Strings
Besides code obfuscation, the other essential barrier we have to overcome is string encryption. As string literals are easy targets for reverse-engineering attacks, advanced obfuscators try very hard to hide them through encryption. At first sight, it may seem very difficult to decipher those encrypted strings for narrowing the scope of tracing. Nevertheless, it turns out that it is not really difficult to get the decrypted strings using aspects. The reason is as follows.

Since the encryption and decryption of string literals are not a standard service of Java runtime, at some point the program under attack must decrypt the string literals by itself before they are displayed to the user. In other words, there will be a method invoked to perform the string decryption. Moreover, it is natural to define class fields of string type for holding the decrypted strings. Both give us very good join points to trap and log the decrypted strings. The following code segment shows a simplified version of string decryption used in one of our experiment target.

```java
public class C {
public C() {
  ds = decryptMethod( es );
}
private String es = ENCRYPTED_STRING_LITERAL;
private String ds;
}
```

Clearly, given such code patterns, we can easily trap the decrypted strings and log them using either a piece of after advice with a method call join point or a before advice with field set join point. All this is done without having to know anything about how the decryption works. Afterwards, we can examine the logged strings and look for those that contain key words related to the data checking. The logged information regarding those strings’ definition and use will provide us critical clues for further tracing.

It is also worth mentioning that the attempt to obfuscate java class files through some special encoding and decoding scheme has the same type of vulnerable points as string encryption. Hence it will also lend itself to such interceptor-based attacks [15].

3.3 Modifying the Target Methods’ Behavior
Once we have located the exact code segment that reports the check result, the following steps are rather straightforward. The main task is to devise an aspect that will intercept the request to data checking result and return a forged success result. Fig. 1 shows the overall picture of how we modify the data checking result using aspect instrumentation. It is conceivable that the check result will be reported in only a few ways. Hence by looking into the specific way a check result is reported, it will be clear how we may overcome the check and return a forged success result using aspects. In particular, we show three typical ways to report the check result and the corresponding modifying aspects as follows.

First, the data check result is returned from a specific method using a Boolean value, date time value, or any other values of a proper type. For such cases, we can write a piece of around advice with method execution join point that skips the underlying method and simply returns a success result. The Modify aspect defined in Listing 1 is such an example.
Figure 1. Modifying target module with aspects

Second, if some wrong data is entered, the data checking method throws an exception to signal the situation. To disable it, we can simply replace the method by a piece of dummy around advice as follows.

```java
void around() : execution( void C.dataCheck() ) {
    // do nothing
}
```

Third, the data checking result is stored in a field and a method is invoked to examine its value and take any abort action if failed. The following is the code skeleton of such checks.

```java
public class C {
    private boolean b;
    // set b’s value according to some internal logic
    public void dataCheck() {
        if ( b ) { // wrong data entered
            // abort the execution,
            // put something difficult to bypass
        }
    }
}
```

A simple way to overcome such check is to intercept the assignment to the field and replace its value by a complementary one. Clearly, we can use the field access join point and around advice to achieve the task as follows.

```java
void around( boolean b ) : set( boolean C.b ) && args( b ) {
    // resume execution but set b’s value to false
    proceed( false );
}
```

Compared to locating the data checking code, modifying it with aspects is apparently easier. This is due mainly to the peculiar nature of around advice. Furthermore, as mentioned earlier, we can weave these malicious aspects into Java programs in bytecode format and run the instrumented programs directly, thus overcoming the barrier of re-compilability established by advanced obfuscators.

4. CASE STUDIES

To test the effectiveness of our ideas of aspect attacks, we have selected six commercial Java packages as our targets. Three of them are Java bytecode obfuscators and the others are development tools. They are all chosen from the Google directory without special preferences other than that they must be obfuscated and the obfuscators better support control obfuscation or string encryption. We downloaded the trial version of the six Java packages from their vendors’ websites and then started to attack them one by one.

Our working environment is MS Windows XP Professional with SP1. The following lists the software tools we used.

- Java 2 Standard Development Kit, Version 1.4.2_04
- Jad Decompiler¹, Version 1.5.8e2 (released August 2001)

They are all free for non-commercial use and are downloadable through the Internet.

While each target package may require different tactics in the attack, the major steps are quite generic and can be summarized as follows.

1. **Step 1**: Perform bottom-up control flow tracing and data flow tracing.

2. **Step 2**: Examine the log and decompile any Java classes that look suspicious of doing the data checking work. Study the code and continue the tracing and decompiling round if necessary. Usually this will continue for a few rounds until we find the right point to attack. If strings are encrypted in the target, we also need to get the decrypted strings using the advice described in Section 3.2.3.

3. **Step 3**: Inspect the identified attack point and choose the appropriate around advice to conduct the attack. We may need to weave in more than one piece of attack advice to overcome the data checks completely.

4. **Step 4**: Run the instrumented package and make sure that most of its functions are runnable without entering the correct data. It is also likely that we chose the improper attack point and need to backtrack to Step 2 for another try.

Among the six targets, only one escapes our attack. It is not a failure of our method per se, but rather a weakness of AspectJ implementation. This particular Java package comes with a jar file greater than 14 MB and it seems that the sheer size of it made AspectJ compiler hung and thus prevented us from weaving in the attack aspects. Table 1 summarizes the basic information of the other five packages and the experiment results of our aspect attacks.

The five attacked targets are listed in Table 1 from left to right by the sequence of attacks. For protection purpose, we omit their package names and refer to them by package A, B, C, D, and E. Package A has the largest main jar file and is the first one we attacked. In the beginning, we spent some time to adjust our tracing aspects and get used to read the decompiled yet obfuscated Java program. Hence it takes us the longest time, about fourteen man days, to finish overcoming its data check successfully. Package B’s data checking code looks intriguing to us initially, yet the string literals are not encrypted and thus reveal its secrets about how it verifies data inputs. We used two pieces of around advice to disable its data checks. Compared to Package A, we spent about half of the time to finish the attack.

Table 1: Summary of the Java packages attacked in the experiment

<table>
<thead>
<tr>
<th>Packages</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Development tool</td>
<td>Development tool</td>
<td>Obfuscator</td>
<td>Obfuscator</td>
<td>Obfuscator</td>
</tr>
<tr>
<td>jar file size</td>
<td>7.52MB</td>
<td>2.18MB</td>
<td>503KB</td>
<td>147KB</td>
<td>2.35MB</td>
</tr>
<tr>
<td>Obfuscated</td>
<td>Partially</td>
<td>Fully</td>
<td>Fully</td>
<td>Fully</td>
<td>Fully</td>
</tr>
<tr>
<td>Control obfuscation</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>String encryption</td>
<td>no</td>
<td>no</td>
<td>field set</td>
<td>method execution</td>
<td>method execution</td>
</tr>
<tr>
<td>Bypassing join point</td>
<td>method execution</td>
<td>method execution</td>
<td>field set</td>
<td>method execution</td>
<td>method execution</td>
</tr>
<tr>
<td>Time spent</td>
<td>14 man/day</td>
<td>7 man/day</td>
<td>1 man/day</td>
<td>1 man/day</td>
<td>1 man/day</td>
</tr>
</tbody>
</table>

Afterwards, we felt quite confident about our approach and defeated the remaining three obfuscators without much effort. Package C uses a specific field to store the data check result. Package D stored the encrypted strings in a poorly camouflaged gif file. Both are easy to defeat. Yet we must admit that the size of their jar files is also significantly smaller than the others. Package E is special because it employs both string encryption and control flow obfuscation to protect itself. The decompiled Java program is illegal and hardly readable because there are many unresolved bytecode mixed with the source code. Even though we were puzzled when looking at the mysterious code, we soon found out the pattern of how it performs string decryption and were able to trap it to log the decrypted strings. Once this is done, it is fairly straightforward to pinpoint where the data check result is reported and to weave in an aspect that returns a forged success result.

5. DISCUSSION

After having conducted the experiment of aspect-based instrumentation attacks on stand alone Java programs, we feel obliged to suggest some ways for defending against such attacks. A naive solution is to perform a preventive weaving for the key classes of the program to release without using the "Xreweavable" option, which is available in AspectJ since version 1.2. This will make those classes non-reweavable and thus save the program from aspect attacks. However, this non-reweavable option is implemented as a flag in a class file, which is quite easy to locate and modify. Hence we must turn to other more robust solutions. This turns out to be more of a challenge, for it may even link back to the nature of JVM in general and the design of Java bytecode format in specific. A general discussion on such issues is beyond the scope of this paper; hence we shall only focus on the specific issues related to AspectJ. We argue that the rich join point model of AspectJ is a double-edged sword that gives software developers as well as malicious crackers the same great instrumentation capability. We do not have any solid solutions, yet we shall summarize our findings and suggestions on some proper restrictions of the instrumentation mechanisms of AspectJ.

5.1 Join Point Elimination

Since the attack targets are the vulnerable join points of a Java program, a direct defense would be protecting those join points from aspect weaving. One idea that occurred to us is to employ the illegal identifiers obfuscation technique of [4] so that all the field and method names associated with those critical join points are replaced by illegal identifiers. This will prohibit crackers from writing aspects that aim to attack those join points, for AspectJ compiler cannot accept those illegal join point specifications in an aspect. However, after second thought, we realize that this idea does not work. A smart cracker can run a second obfuscation on the obfuscated bytecode using another obfuscator to replace those illegal identifiers with randomly generated yet legal strings. Then the cracker will be able to perform the aspect attacks. This technique is also confirmed by Cimato et al [5].

Since we cannot disguise those join points, an alternative defense is to eliminate them from the data checking code. Given the powerful join point model of AspectJ, this means that we somehow have to perform the key steps of data checking in one big step, for example, manually inlining the data verification and result reporting code in the “main” method of the Java program. Furthermore, we should not use any class fields for storing the check results, for they could be trapped by field access join points. These actions will seal the program and leave no room for aspect attacks, but doing so is really a great sacrifice of modularity for security. While this method inlining solution is apparently workable in principle, it should be performed by tools, not by the programmers. Indeed, it is highly desirable that Java compilers provide some kind of security-directed method inlining for protecting key components of a Java program. The annotation mechanism introduced in Java 5 seems to be a good medium for communicating such inlining requirements to the compiler. However, it seems that, in most Java implementations, method inlining is performed at the JVM bytecode level, not at the source code level [17]. According to a recent study [3], common method inlining is not generally available in popular Java compilers. For example, the javac compiler from standard Java Development Kit does not seem to perform any kinds of method inlining. On the other hand, we are glad to learn that security researchers have started to investigate techniques for security-driven compilation
[16], in particular, the joint compiler-hardware approaches. We believe that our results can contribute to extending the scope of this emerging subject of research to include Java and JVM.

5.2 Join Point Encapsulation
In addition to seeking ways to defend against the malicious aspects, we should also reflect on what features of AspectJ enable these aspects to perform malicious instrumentation in the first place, and whether they should be properly adjusted to disable such attacks. In our experiment, we noticed that, in AspectJ, an aspect’s advice can target at both public and private members of a class, be it a method or a field. There is no adequate way to control the accessibility of join points to aspects for behavioral modification. As a result, our malicious aspects can easily tap into any join points of the underlying program and apply their advice without any restrictions.

If we can somehow “hide” the critical join points in a program from external aspects, then we will be able to save the data checking code from aspect attacks. In other words, we need some mechanism to encapsulate join points just like using private modifiers to encapsulate class members. In fact, we are not the first to advocate the notion of join point encapsulation; the term join point encapsulation was first proposed by Larochelle et al [10] for addressing modularity issues. They proposed to use aspects as constructs for expressing join point visibility policies and suggested a new kind of advice called restriction advice that identifies which join points are encapsulated against aspects. If given this mechanism, we can use a piece of restriction advice to prohibit any aspects from accessing the critical join points of a program’s data checking code. In another work, called Open Modules, Aldrich [1] suggests extending a module’s interface to include pointcut declarations so that external aspects can target at only pointcuts declared publicly. If such encapsulation mechanism was available, the data checking code could have been protected by hiding their join points from the module interfaces. Therefore, our results provide strong support from security perspective for such proposals of adding an encapsulation facility for AOP, and AspectJ in particular.

6. CONCLUSIONS
In this paper we have illustrated that binary weaving tools of AOP may become a new breed of productivity tools that facilitate the tampering of software integrity and malicious attacks. In particular, using the powerful join point based instrumentation mechanism of AspectJ, one can overcome the protection barriers established by advanced Java bytecode obfuscators. We have successfully applied this approach to overcome the data checks of several commercial Java packages with very little efforts. We published this result because we believe that the simplicity and very low cost of such instrumentation-based attacks make them worth more attention from the community. Besides, since these malicious aspects also rely on the rich join point model of AspectJ to overcome data checks, our results can provide some food for thought for aspect-oriented language designers when reviewing the join point model.

7. REFERENCES