A Practical Method for Watermarking Java Programs

Akito Monden
Graduate School of Information Science, Nara Institute of Science and Technology
akito-m@is.aist-nara.ac.jp

Hajimu Iida
Information Technology Center, Nara Institute of Science and Technology
iida@is.aist-nara.ac.jp

Ken-ichi Matsumoto
Graduate School of Information Science, Nara Institute of Science and Technology
matumoto@is.aist-nara.ac.jp

Katsuro Inoue
Graduate School of Engineering and Science, Osaka University
inoue@ics.es.osaka-u.ac.jp

Koji Torii
Nara Institute of Science and Technology
torii@is.aist-nara.ac.jp

Abstract

Java programs distributed through Internet are now suffering from program theft. It is because Java programs can be easily decomposed into reusable class files and even decompiled into source code by program users. In this paper we propose a practical method that discourages program theft by embedding Java programs with a digital watermark. Embedding a program developer's copyright notation as a watermark in Java class files will ensure the legal ownership of class files. Our embedding method is indiscernible by program users, yet enables us to identify an illegal program that contains stolen class files. The result of the experiment to evaluate our method showed most of the watermarks (20 out of 23) embedded in class files survived two kinds of attacks that attempt to erase watermarks: an obfuscactor attack, and a decompile-recompile attack.

1. Introduction

Java technology is widely regarded as revolutionary by means of its portability: an idea that the same program should run on many different kinds of computers and electric devices. Java enables us to let computers and devices communicate with one another much more easily than ever before.

However, this revolutionary property of Java brought us a security problem against *program theft*[18]. Java applets put on Internet sites and Java applications sold to users are now suffering from program theft. It is because the Java program can be easily decomposed into reusable components (called *class files*), and analyzed with class viewers[6][22]. As shown in figure 1, class viewers expose the internals of a class file, displaying class structure (fields and methods), thus, program users may know how to use that class file without asking to the original programmer. To make matters worse, program users can obtain source codes of a class file by using *Java decompilers*[15], such as SourceAgain[2], Jad[14], Mocha[24], etc. In this situation, the Java program developer's intellectual property will be infringed if a program user steals anyone else's class file and builds it into his/her own program without the original programmer's permission. We call this copyright infringement a program theft, which is one of the reasons why many companies hesitate to use Java in the real software development. Although we have copyright law to prohibit the program theft, it is still very important for us to protect our Java program by ourselves [3].

This paper proposes a technique that discourages program theft by embedding Java programs with a digital watermark. Embedding a program developer's copyright notation as a watermark into Java class files will ensure the legal ownership of each class file. Below we describe major features of our watermarking method:

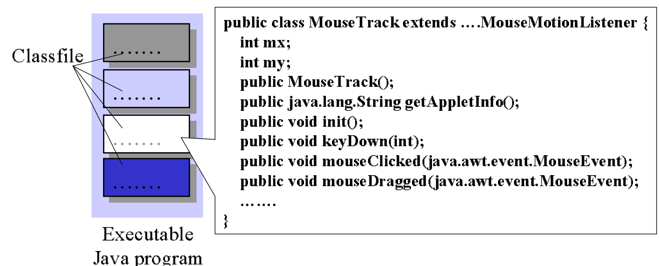

Figure 1. Class information obtained by class viewer

1. Watermarks in Java program do not reduce the execution efficiency.
2. Watermarks in Java program is exposed only when original program developer uses the watermark-decoding tool.
3. Program users can hardly know the location of watermark, thus, erasing and/or tampering with the watermark is very hard for them.
4. Even in case only a part of a program was stolen and was built into other program, the watermark is easily decoded wherever it may exist in that program.
5. Most of the watermarks embedded in class files survive two kinds of attacks that attempt to erase watermarks: an obfuscactor attack, and a decompile-recompile attack.

The remainder of the paper first describes cases where watermark is needed (Section 2). Next, describes related works concerning the program theft, and also introduces some present watermarking techniques for computer programs (Section 3). Then we propose a watermarking method for Java programs including the watermark encoding procedure and the decoding procedure (Section 4). Afterwards, we describe an experiment to evaluate our method (Section 5); and in the end, conclusions and future topics will be shown (Section 6).

2. Cases where watermark is needed

2.1 Proving the fact of program theft

Even in case a suspicious program was found, it is often not easy to say who is the true developer of that program. A program thief who stole anyone else's class file may insist, "I have originally developed this program." In this case, we need a *proof* to explode the thief's claim.

For that situation, a signature previously embedded in the program as a watermark authenticates original program developer's ownership and protects the proprietary interests. The watermark in the program has an effect of proving the fact of program theft. Decoding the watermark from a suspicious program will clarify, who is the original developer of that program.

2.2 Finding Stolen Programs

It is not easy for Java program developers to find an illegal program that contains stolen class files. For example, we may not be able to find out whether it is an illegal program or not by executing the program, because illegal programs often do not resemble original programs in their specifications if the stolen piece is small. This difficulty in finding the illegal program is a crucial problem for Java program developers.

However, we claim that watermark is effective to find the illegal program especially in Internet situation. Illegal programs may be easily detected by employing a watermark-decoding agent (robot) that investigates through Internet. This agent goes through Internet and finds any Java program, then checks the watermark in it. If that program turned out to be an illegal program that contains a watermark of an original program developer, the agent will inform it to the developer. By using such an agent, program developers can automatically find the illegal program on Internet, thus they can protect their programs from program thieves.

3. Related works

3.1 Current solutions for program theft

There are two current solutions for program theft: (1) using NET (native executable translation) compilers, and (2) using obfuscators. Both of these solutions make Java programs difficult to be analyzed, and discourages program thieves in using someone else's class file.

In the first solution, we use NET compilers, such as Symantec Visual Café, Microsoft Visual J++, Asymetrix SuperCede, etc. While common Java compilers compile Java source code into class file, these NET compilers compile source code into machine-specific binary code, which is not reusable. This means, when we use NET compilers, there will be no class file to be stolen. However, this will render the Java program nonportable. An executable code created by NET compiler runs only on the specific computer environment, such as Microsoft Windows. Therefore, this solution is not always useful for software developers.

In the second solution, we use obfuscators – tools that scramble the variable and method names so that decompiled source code is unintelligible gibberish[20]. There are many obfuscators available such as Source-Guard[1], Jshrink[10], DashO[21], etc. To some extent, it is effective to protect Java programs from program thieves by using those obfuscators. However, obfuscated or not, the resultant decompiled code is still source code, which helps thieves to understand the usage of that program.

Although above two solutions have an effect of discouraging the program theft, they are not enough strong for the protection of Java programs.

3.2 Digital Certificate for Java Applet

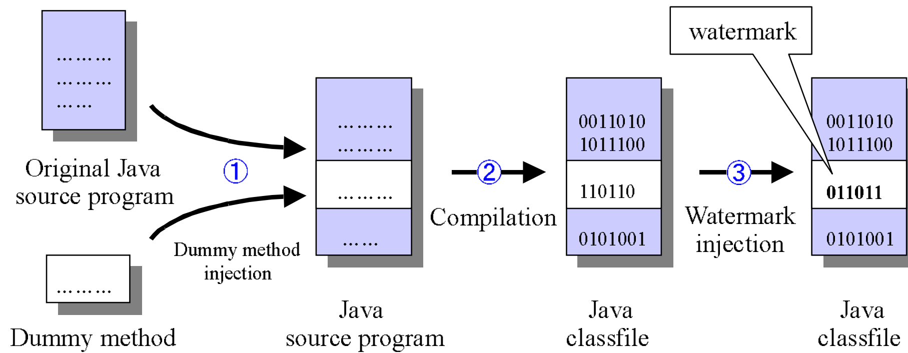

Figure 2. Overview of watermark encoding procedure

Java security model provides us an authentication mechanism known as a concept of *signed applet*[16][23]. A sign (called *digital certificate*) related with an applet indicates, who is the true developer of that applet, just like a watermark does. A user who has downloaded a signed applet can know whether it is worthy reliable or not by checking the sign. The sign also guarantees that the applet is an original one, and has never been tampered with by anyone else.

However, this signed applet does not make any sense for the purpose of protecting class files contained in the applet. It is because the sign is not embedded in the class files, i.e., separated from the class files. The sign is just an encrypted file that can be correctly decrypted only when the applet is an original one. In order to protect a class file, the sign should be embedded in the class file, and never be separated from the class file.

3.3 Other Watermarking methods for computer programs

There were many studies done for watermarking images, sounds, texts, etc [4][7]. On the other hand, a few methods for watermarking computer programs have been proposed [5][8][11][13].

Hirose et al. proposed a method for embedding C source programs with user identification number[11]. In their method, one single change in the target program represents one bit information. This change in the program includes addition of dummy variables, rearrangement of variable declaration statements, changes in coding style (such as ‘n=n+1’ to ‘n++’), and so on. In the decoding phase, the original program and the marked program are compared so that the changes in the program will be identified. However, watermark in the program is undecodable if only a part of the program was stolen. Moreover, if program thieves applied the same method to the already watermarked program, original watermark will be easily erased.

Kitagawa also proposed a method for watermarking Java programs[13]. In this method, new variables are appended to a program; and, watermark codewords (values) are set to those variables. In decoding phase, program developers need to replace a specific class file in the program with a special class file (called *detection class file*), and execute the program. This detection class file outputs the values of the variable in which watermark was embedded. However, this method is not useful from the purpose of defending class files from program thieves. Because, when we want to check whether a target program contains a stolen program or not, we must find out which class file is to be replaced with the detection class file. If we couldn’t find the class file that should be replaced, or simply there is no class file that should be replaced, we can not decode the watermark even if a watermark exists in the target program. It is natural that program developers want to easily check whether a target program contains a stolen program or not. From this viewpoint, watermarks should be automatically decodable from a target program by using a decoding tool. Also, we insist that watermarks should survive various program translations. Program thieves may disassemble and re-assemble the program, decompile and re-compile the program, optimize the program, or use other program translation tools such as obfuscator, scrambler, etc[15]. Watermarks should be reliably decodable even after these program transformations were applied.

Collberg and Thomborson proposed a software watermarking technique in which a dynamic watermark is stored in the execution state of a program[5]. Dynamic watermarks are stored in a program’s execution state, rather than in the program code itself. Their method is easy to tamperproof against various program transformations. However, they watermark *complete applications*, not individual modules (such as class files). Hence, cropping a particularly valuable class file from a Java application for illegal reuse is likely to be a successful attack against this method.

Davidson and Myhrvold proposed a method for generating and auditing a signature for executable program modules[8]. A signature is a means that uniquely identifies an authorized copy of the executable module delivered to each user. The signature (identification number) of each authorized copy is encoded within the order of instructions of the executable module. However, this method is not suitable for watermarking a sentence, such as a copyright notification, into each Java class file because encodable codeword in this method is quite short. This method is also highly susceptible to additive attack: inserting a new watermark overrides original mark so that it can no longer be extracted.

4. Watermarking method

4.1 Encoding procedure

Our watermark encoding procedure consists of the following three phases (see Figure 2):

(Phase 1) Dummy method injection

In the first phase of watermarking, a dummy method (of a class), which will never be executed, is appended to a target Java source program. This dummy method is a *space* for watermark codeword. This dummy method should have enough size for watermark injection.

Next thing we should do is to append a dummy method invocation to the source program. Below we show an example of the invocation statement:

if (Condition) Dummy_Method();

'*Condition*' is an expression that will never become true. So, actually, dummy method is never invoked. If this expression (formula) is complex enough, it is difficult for program users to become aware of the dummy method[12]. Since large programs originally contain many methods that are rarely executed, it is not easy for program users to locate the dummy method.

Although this phase can be automated, the developer of a class file should manually inject a dummy method and its dummy invocation statement by his/her hand. Since program thieves might decompile and read the class file, the developer should carefully write a dummy method that does not seem to be a dummy. This is why we inject a dummy method into the source program, not in the compiled class file. However, if the developer does not care about the possibility of decompilation, or he/she just prefers easy watermarking, he/she may use an automatic dummy method injection rather than a manual injection.

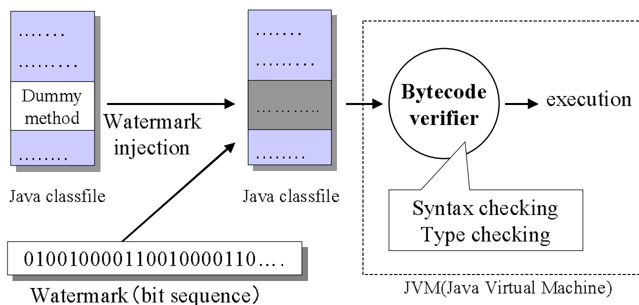

Figure 3. Consideration in the watermark injection phase

(Phase 2) Compilation

In the second phase, the Java source program, in which dummy method was injected, is compiled with a Java compiler. We use common Java compiler for this compilation.

(Phase 3) Watermark injection

In this phase, we need to take account of the *bytecode verifier*[17]. As shown in figure 3, when we execute a Java applet, bytecode verifier checks the syntactical rightness and type consistency of the class file. If we simply overwrite a bit sequence into the dummy method, we will not be able to execute the program because it will not pass the check of bytecode verifier. Therefore, watermark injection should keep the syntactical correctness and type consistency.

In order to keep the syntactical correctness and type consistency, we use following two approaches:

(i) Overwriting numerical operands

One simple way to keep syntactical correctness is to limit the place to overwrite. A numerical operand of an opcode that pushes a value to the stack, and of an opcode

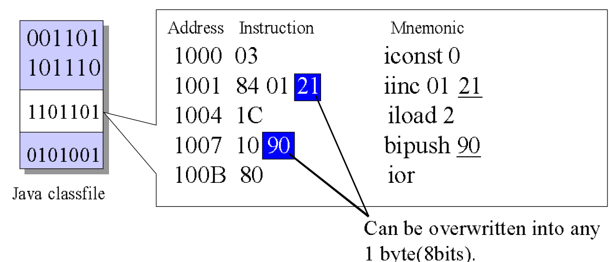

Figure 4. Overwriting numerical operands

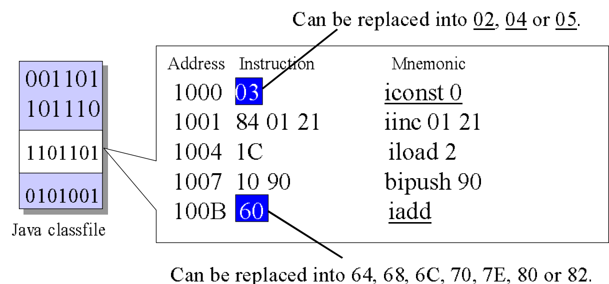

Figure 5. Replacing opcodes

Instruction		assigned bits
60 iadd	…	000
64 isub	…	001
68 imul	…	010
6C idiv	…	011
70 irem	…	100
7E iand	…	101
80 ior	…	110
82 ixor	…	111

Rule 1

Instruction		assigned bits
9B iflt	…	00
9C ifge	…	01
9D ifgt	…	10
9E ifle	…	11

Rule 2

Instruction		assigned bits
C6 inull	…	0
C7 inonnull	…	1

Rule 3

Figure 6. Example of bit assignment rules for opcodes

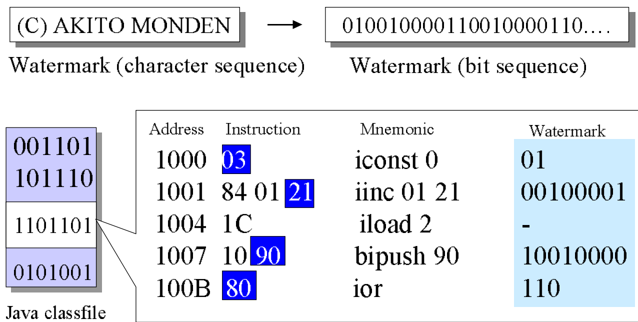

Figure 7. Example of encoded watermark

that increases a value on a stack, can be overwritten without syntactical incorrectness and type inconsistency. For example, an operand '*xx*' of the opcode '*iinc xx*' and '*bipush xx*' can be overwritten into any single byte. Figure 4 shows an example of numerical operands that can be overwritten. Please note that overwriting the numerical operand does not change the specification of a class file because operands to be overwritten are in the dummy method, which will never be executed.

However, most of other operands that indicate a position or an index of class tables or local variables of a method cannot be overwritten because of violating syntactical correctness. For example, an operand such as '*getfield xx*' and '*putfield xx*' cannot be overwritten. This limitation in operands drastically reduces the place that can be overwritten.

(ii) Replacing opcodes

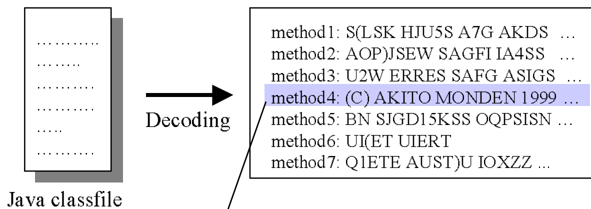

Figure 8. Example of decoded watermark

In order to increase the place for watermark injection, we replace some of the opcodes, such as *iadd*, *ifnull*, and *iflt*, into other kind of opcode. For example, an opcode replacement from *iadd* to *isub* does not violate syntactical correctness and type consistency. Moreover, the opcode *iadd* can be replaced to anything among *isub*, *imul*, *idiv*, *irem*, *iand*, *ior*, and *ixor*. This indicates that above eight opcodes *iadd*, *isub*, ..., and *ixor* can be replaced mutually. By using this ability of mutual replacement, we can encode 3 bits information into these opcodes. For example, we may assign 000_2 to *add*, 001_2 to *isub*, 010_2 to *imul*, ..., and 111_2 to *ixor*..Whichever the above opcode appeared in the dummy method, we will replace them into one of the above eight opcodes according to the bits we want to encode. Figure 5 shows an example of opecodes that can be replaced mutually. Such a bit assignment and an opcode replacement can be also applied in other opcodes. Figure 6 shows an example of bit assignment rules for opcodes.

In case we want to encode a sentence, such as '(C) AKITO MONDEN', first we need to translate the sentence into a bit sequence, then we encode the bit sequence into the program. Figure 7 shows an example of watermark encoding.

4.2 Decoding Procedure

In the watermark-decoding phase, there is an assumption that we know the relation between bytecodes and their assigned bits, and also the relation between bit sequences and alphabets. The decoding algorithm is very simple. We simply do the exactly opposite procedure of watermark injection procedure, from top of every method, i.e., we replace operands and opcodes in each method into bit sequence followed by bit assignment rules, then replace bit sequence into character sequence. After that, watermark will appear from the dummy method (Figure 8). This decoding procedure can be automated, so that even in case only a part of a program was stolen and was built into other program, the watermark is easily decoded wherever it may exist in the program.

5. Experiment

In this section, we are to evaluate the strength of watermarks against program translation attacks that attempt to erase the watermark. We assume that program thieves use two kinds of attacks, (1) obfuscator attack, and (2) decompile-recompile attack (Figure 9). In the experiment, we try to decode the watermark after those two attacks were applied. Both the watermark encoding tool and the decoding tool we have developed are open in our web site [19].

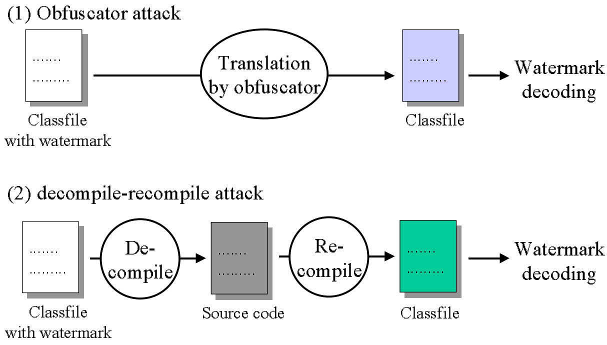

Figure 9. Two kinds of attacks in the experiment

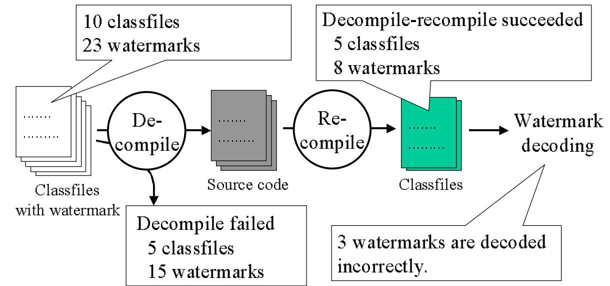

Figure 10. Result of decompile-recompile attack

5.1 Experiment Procedure

(Step 1) Preparing the Java source code

We randomly chose 10 source files from sample Java applets in JDK 1.2. The number of methods that have enough size for the watermark injection was 23 in sum total. Here, we assumed these 23 methods are the dummy methods.

(Step 2) Encoding the watermark

We injected a character sequence '(C) AKITO MONDEN' into each 23 dummy methods. Then, we compiled all source files and got 10 class files in which watermarks are injected.

(Step 3) Attacking the class files

Two kinds of attacks as follows were applied to each 10 class files:

(i) Obfuscator attack

We applied the 4thpass's SourceGuard version 2.0[1], that is widely used as one of the strong obfuscators, to each class file.

(ii) Decompile-recompile attack

We applied Hanpeter van Vliet's Mocha[24], the first and most widely known decompiler, to each class file, and got the source codes of them. As compared among mocha and other decompilers in the Dave Dyer's article[9], Mocha shows pretty good performance in decompilation.

Then, we applied javac, a common compiler in JDK, to each source code, and got the class files again. In this compilation, we used optimization option.

(step 4) Decoding the watermark

Watermark-decoding procedure was applied to each classfile after the attacks.

5.2 Result of experiment

(i) Result of obfuscator attack

After applying the obfuscator, we tried to decode watermarks. Consequently, all watermarks were decoded correctly. Generally, obfuscators translate symbols such as variable name and method name, but they do not affect operands and opcodes in the method.

(ii) Result of decompile-recompile attack

Figure 10 shows the result of decompile-recompile attack. We have 10 class files and 23 watermarks in them at first. When we tried to decompile them, 5 classfiles were failed in decompiling (decompiler crushed), i.e., we got only 5 source files. Next, we recompiled them. The syntax errors occurred in recompilation were fixed manually. We finally obtained 5 classfiles with 8 watermarks in them. Then, we applied decoding procedure to these 5 classfiles. As a result, 3 watermarks are decoded incorrectly. So, the number of watermarks erased by decompile-recompile attack was 3 out of 23.

The result showed that the decompile-recompile attack does not always succeed; and, even if it was succeeded, more than half of the watermarks (5 out of 8) was not erased. By injecting more than two watermarks into each class file, we will be able to protect our class files from decompile-recompile attack.

In addition, class files that are succeeded in decompiling were of small size. This indicates that the decompiler is not practical for the large class file.

6. Conclusions and future topics

We have described the problem of program theft in current Java environment, and have proposed a watermarking method applicable to Java programs. Our method is practical in protecting Java class files and superior to conventional methods because:

(1) A copyright notification message can be encoded into *each class file* as a watermark to protect each of class file. On the other hand, Kitagawa's method[13] and Collberg and Thomborson's method[5] watermark *complete applications*.

(2) Program thieves can not erase the watermark by additive attacks that insert new bogus watermarks into an already watermarked program, while Davidson and Nyhrvold's method[5] is not.

(3) Most of watermarks (20 out of 23) encoded into class files survived two kinds of attacks that attempt to erase watermarks: an obfuscactor attack, and a decompile-recompile attack, while conventional method of Hirose et al.[11] is highly susceptible to these kind of attacks.
(4) Anyone can use our watermarking tool. Both encoding and decoding a watermark is easily done by using tools that we have released in our web site[19]. In addition, the encoded watermark is decodable , from the suspicious program wherever the stolen class file was built in.

However, none of the watermarking methods, including our methods, are immune to all types of attacks. Our method is resilient against additive attack, obfuscator attack and decompile-recompile attack, but, there may be more strong transformation attacks. In the future, in order to make watermarks more tamper-resistant, we are to apply *error correcting code* to our watermarking method.

Acknowledgments

The authors wish to acknowledge helpful suggestions from Dr. Yuuji Ichisugi of Electrotechnical Laboratory.

References

[1] 4thpass LLC, *SourceGuard*, <http://www.4thpass.com>.
[2] Ahpah Software, *SourceAgain*, <http://www.ahpah.com/sourceagain/>.
[3] Behrens, B. C. and Levary, R. R., "Practical legal aspects of software reverse engineering," *Communications of the ACM*, vol. 41, no. 2, Feb. 1998, pp. 27-29.
[4] Berghel, H., "Watermarking cyberspace," *Communications of the ACM*, vol. 40, no. 11, 1997, pp. 19-24.
[5] Collberg, C. and Thomborson, C., "Software watermarking: Model and dynamic embeddings," *The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, San Antonio, Texas, Jan. 1999.
[6] Comware Australia Pty. Ltd., *ClassNavigator*, <http://www.comware.com.au/classnavigator/classnav.htm>.
[7] Craver, S., Memon, N., Yeo, B. and Yeung, M. M., "Resolving rightful ownerships with invisible watermarking techniques: Limitations, attacks, and implications," *IEEE Journal on Selected Areas in Communications*,,vol. 16, no. 4, 1998, pp. 573-586.
[8] Davidson, R. L. and Myhrvold N., "Method and system for generating and auditing a signature for a computer program," *US Patent*,, no. 5559884, Assignee: Microsoft Corporation, Sep. 1996,
[9] Dyer, D., "Java decompilers compared," in article of *JavaWorld*, July 1997, <http://www.javaworld.com/javaworld/jw-07-1997/jw-07-decompilers.html>.
[10] Eastridge Technology, *Jshrink*,, <http://www.e-t.com/ jshrink.html>.
[11] Hirose, N., Okamoto, E., Mambo, M., "A proposal for software protection," in *Proc. 1998 Symposium on Cryptography and Information Security*, SCIS'98-9.2.C, Jan. 1998. (in Japanese)
[12] Ichisugi, Y., "Watermark for software and its insertion, attacking, evaluation and implementation methods," *Summer Symposium on Programming*,, IPSJ, July 1997, pp. 57-64. (in Japanese)
[13] Kitagawa, T. "Digitalwatermarking method for Java programs," *Master's Thesis, Department of Information Processing, Graduate School of Information Science, Nara Institute of Science and Technology*,, NAIST-IS-MT9751041, Feb. 1999. (in Japanese)
[14] Kouznetsov, P., *Jad – the fast Java Decompier*, <http://meurrens.ml.org/ip-Links/Java/codeEngineering/jad15. Html>.
[15] Leininger, K.E., *The Java developer's tool kit*, McGraw-Hill, 1997
[16] McGraw, G. and Felten E., *Java security: hostile applets, holes, and antidotes*,,John Wiley & Sons, 1997.
[17] Meyer, J. and Downing, T., *Java virtual machine*, O'Reilly & Associates, 1997.
[18] Monden, A., Hajimu, I., Matsumoto, K., Katsuro, I. and Torii, K., "Watermarking Java programs," in *Proc. 4th International Symposium on Future Software Technology (ISFST'99)*,,Software Engineers Association, Nanjing, China, Oct. 1999, pp. 119-124.
[19] Monden, A., *Java watermarking tools*,, <http://tori.aist-nara.ac.jp/jmark/>.
[20] Nolan, G., "Decompile once, run anywhere: protecting your Java source," *Web Techniques Magazine*, vol. 2, Issue 9, Sep. 1997.
[21] preEmptive Solutions, *DashO*, <http://www.preemptive.com/>.
[22] Raud, R., *ClassViewer*, <http://raud.net/robert/classinfo/ClassViewer.html>.
[23] Sun Microsystems, Inc., "Security and signed applet," <http://www.javasoft.com/products/jdk/1.1/docs/guide/security/>.
[24] Vliet, H., *Mocha – the Java Decompiler*, <http://www.brouhaha.com/~eric/computers/mocha.html>.