Detecting the Theft of Programs Using Birthmarks

Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto

Graduate School of Information Science, Nara Institute of Science and Technology,
8916-5 Takayama-cho, Ikoma-shi, Nara, 630–0101 Japan,
{harua-t, masa-n, akito-m, matumoto}@is.aist-nara.ac.jp,
WWW home page: http://se.aist-nara.ac.jp/

Abstract. To support the efficient detection of theft of Java class files, this paper presents a new method to derive *birthmarks* from given Java class files. The proposed method extracts from a class file a set of unique characteristics, the birthmarks, based on constant values in field variables, the sequence of method calls, the inheritance structure and used classes. By using the birthmarks, we can easily identify the doubtful class files (those which seem to be copies). Two experiments were conducted to evaluate the proposed method. The first experiment showed that the proposed birthmark successfully distinguished non-copied files in practical Java applications (97.50%). In the second experiment, it was shown that the proposed birthmark had quite a good tolerance for program optimization (97.30%).

Key Words: birthmark, software theft, Java, class files, watermarking

1 Introduction

Today, an enormous number of software products are developed and distributed all over the world. The recent advancement of the Internet dramatically improves easy and fast distribution of software. Unfortunately, however, there are many cases of *software theft* reported. Software theft copies the original software, then uses the copy for other purposes without keeping the copyright notice. Typical scenarios include: crack and duplicate a whole product and sell the copies (i.e. software piracy), or steal a part of a product (e.g, modules) and use it as a part of other product. For example, there was an incident with an illegal version of "Office XP" in which the product authentication routine was cracked, and posted to a news group (alt.binaries.warez.ibm) [21].

Software is also stolen because of unawareness of software license. For example, the GPL, which is one of the free software licenses, prescribes a rule that modified versions of the GPL software must use the GPL [9]. People who are unaware of such regulation of the GPL might infringe the copyright (for example, [8]).

If software theft occurs for *a part of* software such as some modules, files, and documents, it is much more difficult to detect the fact. For such cases, we need to identify stolen modules from a number of components. However, if stolen modules are built into a different application and the application is released without source code, it is quite difficult to detect the evidence of theft.

The goal of this paper is to propose an easy-to-use method to provide reasonable evidence for theft of Java programs. Software written in Java is composed of a collection of *class files*. A class file is an atomic execution module containing binary data, which can dynamically invoke other class files. Our concern is to identify theft of the class files. Java programs have been widely used, and a number of class files are distributed in various platforms. A number of class files are distributed in various platforms. Also, rigorous specification of the Java virtual machine enables powerful *decompilers* (e.g. [11]). Thus, it is not difficult to obtain source code from class files. In this sense, theft of class files is relatively easy to do, but difficult to find.

The key idea of the proposed method is to extract a *birthmark* from a given Java class file. Intuitively, a birthmark of a Java class file is a set of unique characteristics that the class file originally possesses. A birthmark is carefully extracted from critical portions of the class file. Ideally it should tolerate a certain extent of *program transformation* such as obfuscation and optimization. Hence, changing the birthmark makes the class file malfunction, or makes it completely different from the original. If a class file q has the same birthmark as another class file p's, q is very likely to be a copy of p. Thus, the birthmark can be used as a simple but powerful signature to distinguish doubtful class files. Since the proposed method does not require source code for extraction of birthmarks, we can show the evidence of theft efficiently. Specifically, we propose four kinds of birthmarks based on the following characteristics: constants in class field variables, sequence of the method calls, structure of inheritance, and used classes.

We evaluate the proposed method by two experiments. The first experiment evaluates the performance of the birthmarks to distinguish non-copied files. The result shows that the proposed birthmarks can identify 97.50% (on average) of class files in practical Java packages: Ant, BCEL and JUnit. In the second experiment, we evaluate how the proposed method tolerates program optimization (transformation). The result shows that the proposed birthmarks can achieve a strong tolerance against program optimization, where the similarity of birthmarks of a class file before/after the optimization is as high as 97.30% on the average.

2 Related Work

Watermarking is a well-known technique to insist on the ownership of the original software for software theft [6, 7, 16, 19, 20, 22]. Therefore, it may be used for our objective. Watermarking is basically to embed stealthy information which identifies the program author. Monden et al. [16, 19, 20] presented methods of watermarking for Java codes which embeds secret messages into numeric operands and opcodes in static dummy methods. Collberg developed a watermarking tool named *sandmark* [22]. Sandmark implements many algorithms of watermarking. One of the algorithms is dynamic watermark. Dynamic watermarking is not like the ordinary watermarking (called static watermark). While static watermarks are stored in program code, dynamic watermarks are embedded into states of program [6, 7].

However, these watermarking techniques are not perfect due to the following two essential problems. The first problem is that: watermarks can be detached if detected, since watermarks are basically *extra* codes added to the original code later. Intelligent crackers may be able to locate embedded information, because the information is not necessary for code execution. In addition, locating embedded information in Java codes is not very difficult since many libraries and tools for engineering Java byte code are available. It is no longer possible to give a proof of theft if the embedded watermarks are completely removed. The second problem is that: we cannot give proofs for modules into which no watermark is embedded. Strictly speaking, to completely prove software theft, we need to embed watermarking into *all related* modules beforehand. This is quite difficult by the add-in nature of the watermarking.

There is also a technique, called *code clone* [1, 3, 4, 18], that could be used for the copy detection of programs. The code clone is basically a set of duplicated code in source programs. The theft is doubted when the code clone is found in different software products. However, the problem of these methods is that they require source code of target programs, which are not necessarily available in our problem setting. The code clone is also quite fragile against program transformation. There are also a techniques for detecting plagiarism in programs [5, 23, 24]. Those techniques also require source code of target programs, and they are not registant to program transformations.

3 Birthmark

To make our discussion clearer, we try to formulate a notion of a birthmark in this section.

Definition 1 (Copy Relation). *Let $Prog$ be a set of all (given) programs. Let $\equiv_{cp}$ denote an (given) equivalent relation over $Prog$ such that: for $p, q \in Prog$, $p \equiv_{cp} q$ holds iff q is a copy of p (vice versa). Then, the relation $\equiv_{cp}$ is called copy relation.*

The criteria whether or not q is a *copy* of p can vary depending on the context. For example, the following criterion are relatively reasonable for general computer programs: (a) q is an exact duplication of p, (b) q is obtained from p by renaming all identifiers in the source code of p, or (c) q is obtained from p by eliminating all the comment lines in the source code of p. To avoid confusion, we suppose that $\equiv_{cp}$ is *originally given* by the user. Since $\equiv_{cp}$ is an equivalent relation, the following proposition holds.

Proposition 1. *For $p, q \in Prog$, the following properties hold. (Reflexive) $p \equiv_{cp} p$, (Symmetric) $p \equiv_{cp} q \Rightarrow q \equiv_{cp} p$ and (Transitive) $p \equiv_{cp} q \wedge q \equiv_{cp} r \Rightarrow p \equiv_{cp} r$.*

All the above properties meet well the intuition of copy. Next, if q is a copy of p, the external behavior of q should be identical to p's.

Proposition 2. *Let $Spec(p)$ be a (external) specification conformed by p. Then, the following property holds: $p \equiv_{cp} q \Rightarrow Spec(p) = Spec(q)$.*

Note that the reverse of this proposition does not necessarily hold, since we can see, in general, different program implementations conforming the same specification. Now we are ready to define a *birthmark* of a program.

Definition 2 (Birthmark). *Let p, q be programs and $\equiv_{cp}$ be a given copy relation. Let $f(p)$ be a set of characteristics extracted from p by a certain method f. Then $f(p)$ is called birthmark of p under $\equiv_{cp}$ iff both of the following conditions are satisfied.*

Condition 1. $f(p)$ is obtained only from p itself (without any extra information).

Condition 2. $p \equiv_{cp} q \Rightarrow f(p) = f(q)$

Condition 1 means that the birthmark is not an extra information and is required for p to run. Hence, extracting a birthmark does not require extra code as watermarking does so. Condition 2 is saying that the same birthmark has to be obtained from copied programs. Also, by the contraposition, if birthmarks $f(p)$ and $f(q)$ are different, then $p \not\equiv_{cp} q$ holds. That is, we can guarantee that q is not a copy of p.

Hopefully, a birthmark should conform the follow properties.

Property 1. For p' obtained from p by any program transformation, $f(p) = f(p')$ holds.

Property 2. For p and q such that $Spec(p) = Spec(q)$, if p and q are written independently, then $f(p) \neq f(q)$.

These two properties strengthen Condition 2 of Definition 2, so that $p \equiv_{cp} q \Leftrightarrow f(p) = f(q)$. First, Property 1 is stating the greatest tolerance to program transformation. We consider that wise crackers may modify birthmarks by converting the original program into an equivalent one. One of such techniques is *obfuscation* [7]. Obfuscation is originally proposed for protecting programs from theft, which makes the original program harder to read. However it can be abused as an attack against birthmarking (as well as watermarking). Property 1 specifies that the same birthmark from p and converted p'. However, since many obfuscation methods have been proposed, it is, in fact, hard to extract such strong birthmark that *perfectly* satisfies Property 1.

On the other hand, Property 2 is saying that: even though the specification of p and q is the same, if implemented separately, different birthmark should be extracted. It is rare that detail of two programs is completely the same for large programs. However, in case that p and q are both tiny programs, extracted birthmarks could become the same, even if p and q, and their specifications are written independently. Those properties should be tuned within allowable range at user's discretion.

The problem is how to develop an effective method f for a set $Prog$ of Java class files and copy relation $\equiv_{cp}$.

4 Proposed Method

4.1 Overview

Here we outline how the proposed method works. First, from a given pair of class files p and q, we extract birthmarks $f(p)$ and $f(q)$ with a method f. Next, we compare $f(p)$ and $f(q)$. Finally, if birthmarks $f(p)$ and $f(q)$ are different, we conclude that q is not a copy of p.

As for the above f, we propose four methods that extract the following four types of birthmarks: **constant values in field variables** (CVFV), **sequence of method calls** (SMC), **inheritance structure** (IS) and **used classes** (UC). For a pair of p and q, if at least one of four birthmarks is different, we conclude that q is not a copy of p. If all of them match, q is very likely to be a copy of p.

Sometimes, a class file p may not contain relevant information enough to extract a certain type of birthmark. In this case, we regard that birthmarks of p and q are the same only when the same type of birthmark is not obtained from q. Similarly, if a type of birthmark is obtained from p but not from q, then we assume that the birthmarks are not the same.

4.2 Proposed Birthmarks

In this subsection, we propose the four types of birthmarks for Java class files. For better comprehension, we use a Java source code in Fig. 1 to show an example for each birthmark. Note that in our problem setting, the source codes of given class files are not necessarily available.

Constant Values in Field Variables (CVFV)

A class of Java often has field variables to store static and/or dynamic attributes of instantiated objects. If the field variables are initialized to be certain constant values upon their declaration, these initial values are essential information to determine the way of object instantiation. Modifying these values is dangerous since the modification may change output of the program. Therefore, the initial values can be used as a good signature that characterizes the class.

Definition 3 (CVFV Birthmark). *Let p be a class file and $v_1, v_2, ..., v_n$ be field variables declared in p. Also, let t_i $(1 \leq i \leq n)$ be the type of v_i and a_i $(1 \leq i \leq n)$ be the initial value assigned to v_i in the declaration. (If a_i is not present, we regard a_i as "null"). Then, the sequence $((t_1, a_1), (t_2, a_2), ..., (t_n, a_n))$ is called CVFV birthmark of p, denoted by $CVFV(p)$.*

The CVFV birthmark of the program in Fig. 1 is:
(`java.lang.String`, ".")
(`java.io.File`, null)

Sequence of Method Calls (SMC)

Usually in Java, general-purpose functions are already implemented as methods of *well-known classes*, such as J2SDK and Jakarta project. So, a class usually calls one or more methods of these well-known classes. We consider that the sequence of method calls can be used as a good birthmark by the following two reasons.

The first reason is that it is difficult for crackers to modify the sequence automatically because of dependencies between the method calls. The second reason is that replacing a method in the sequence with another one takes much effort, since making the alternative requires as much effort as making the well-known class from scratch.

Definition 4 (SMC Birthmark). *Let p be a class file and C be a given set of well-known classes. Let $m_1, m_2, ..., m_n$ be a sequence of methods m_i's invoked in p in this order, where m_i belongs to a class in C. Then, the sequence $(m_1, m_2, ..., m_n)$ is called SMC birthmark of p, denoted by $SMC(p)$.*

The SMC birthmark of the program in Fig. 1 is:

```
org.apache.tools.ant.taskdefs.MatchingTask(),
java.io.File(String),
String[] org.apache.tools.ant.DirectoryScanner#getIncludedFiles(),
java.io.File(java.io.File, String),
void String#endsWith(String),
int String#length(),
String String#substring(int, int),
String String#replace(char, char),
String String#replace(char, char),
Class Class#forName(String),
java.io.ObjectStreamClass java.io.ObjectStreamClass#lookup(Class),
long java.io.ObjectStreamClass#getSerialVersionUID(),
StringBuffer(),
String Class#getName(),
StringBuffer StringBuffer#append(String),
StringBuffer StringBuffer#append(String),
StringBuffer StringBuffer#append(long),
String StringBuffer#toString(),
void java.io.PrintStream#println(String),
String Exception#getMessage(),
org.apache.tools.ant.BuildException(String),
Class[] Class#getInterfaces(),
String Class#getName(),
boolean String#equals(Object).
```

(Package names belonging to `java.lang` are omitted)

Inheritance Structure (IS)

Java is an object oriented programming language. Every class in Java has a hierarchy of *inheritance structure* except `java.lang.Object`, which is a root class of all class. Hence, by traversing the superclasses from a given class p to `java.lang.Object`, we can obtain a sequence of classes. This sequence can be used as a unique characteristics of p. However, the sequence of classes may contain both well-known classes and user-made classes. Since the user-made classes are relatively easily altered, we discard them from the sequence, and use the resultant sequence as a birthmark.

Definition 5 (IS Birthmark). *Let p be a class file and C be a given set of well-known classes. Let $c_1, c_2, ..., c_n$ be a sequence of classes such that $c_1 = p$, $c_i (2 \leq i \leq n)$ is a superclass of c_{i-1}, and c_n is a root of class hierarchy (java.lang.Object). If c_i does not belong to a class in C, we replace c_i with "null." Then, the resultant sequence $(c_2, c_3, ..., c_n)$ is called IS birthmark of p, denoted by IS(p).*

The IS birthmark of the program in Fig. 1 is:
`org.apache.tools.ant.taskdefs.MatchingTask,`
`org.apache.tools.ant.Task,`
`org.apache.tools.ant.ProjectComponent,`
`java.lang.Object.`

Used Classes (UC)

A class (let it say p) generally *uses* other classes to implement new functions by combining existing features of the other classes. These external classes appear in p as a superclass, return types of methods, arguments of methods, implemented interfaces, method calls. Modifying those classes used in p is not easy because of dependencies between the classes. Moreover, if the classes are well-known classes, it is harder for crackers to alter them. Hence, the set of used classes is considered to be a unique birthmark of p.

Definition 6 (UC Birthmark). *Let p be a class file and C be a given set of well-known classes. Let U be a set of classes u's such that u is used in p and $u \in C$. Let $u_1, u_2, ..., u_n$ $(u_i \in U)$ be a sequence obtained by arranging all elements in U in an alphabetical order. Then, the sequence $(u_1, u_2, ..., u_n)$ is called UC birthmark of p, denoted by UC(p).*

The UC birthmark of the program in Fig. 1 is:
`java.io.File,`
`java.io.ObjectStreamClass,`
`java.io.PrintStream,`
`java.lang.Class,`
`java.lang.Exception,`
`java.lang.String,`

```
java.lang.StringBuffer,
java.lang.System,
org.apache.tools.ant.BuildException,
org.apache.tools.ant.taskdefs.MatchingTask,
org.apache.tools.ant.DirectoryScanner.
```

4.3 Similarity of Birthmark

Each of the proposed birthmarks is in the form of a sequence. Suppose that we have a pair of birthmarks $f(p) = (p_1, ..., p_n)$ and $f(q) = (q_1, ..., q_n)$ for class files p and q. Basically, we say that $f(p)$ is the *same* as $f(q)$ (i.e., $f(p) = f(q)$) iff $p_i = q_i$ for all i $(1 \leq i \leq n)$. In other words, even when only a single pair of p_i and q_i is different and other pairs are the same, we have to say $f(p) \neq f(q)$. Thus, the birthmark concludes that q is not a copy of p, although $f(p)$ and $f(q)$ are very *similar* to each other.

Hence, the comparison of birthmarks using equivalence only is somewhat too strict, which may make birthmarks too sensitive against the attacks of program transformation. That is, crackers can easily copy the program with a tiny modification of birthmark. To cope with the problem, we here introduce *similarity* of birthmark.

Definition 7 (Similarity). *Let $f(p) = (p_1, ..., p_n)$ and $f(q) = (q_1, ..., q_n)$ be birthmarks with length n, extracted from class files p and q. Let s be the number of pairs (p_i, q_i)'s such that $p_i = q_i$ $(1 \leq i \leq n)$. Then, similarity between $f(p)$ and $f(q)$ is defined by: $s/n \times 100$.*

The similarity is a percentage of elements matched among $f(p)$ and $f(q)$ in the total elements in the birthmark (sequence).

5 Experimental Evaluation

We have implemented a tool called `jbirth` [15] to support the proposed method. `jbirth` is written in Java (J2SDK 1.4)[14] with Byte Code Engineering Library (BCEL 5.1)[12], comprising about 6000 lines of code. The main features are:

- extraction of the four types of birthmarks directly from Java class files (without source code),
- pairwise birthmark comparison of Java class files,
- Jar file support, and
- plug-in architecture for new birthmarks.

We conduct two experiments here. The first experiment evaluates performance of the proposed birthmarks, while the second experiment measures tolerance of the birthmarks against program optimization.

5.1 Experiment 1 (Performance)

In this experiment, we validate if the proposed birthmarks can be used as *effective* birthmarks for practical applications. Specifically, we evaluate how many class files in an application package can be distinguished from each other by the proposed birthmarks.

Now, let f be one of the four types of birthmarks, and let p, q ($p \not\equiv_{cp} q$) be class files arbitrarily taken from a Java application package. To evaluate the performance of f, we show how many pairs of p and q are successfully distinguished by f (i.e., $f(p) \neq f(q)$, according to Condition 2 in Definition 2).

As the target applications, we chose the following Java packages: `bcel-5.1.jar` in Jakarta BCEL (5.1) [12], `ant.jar` in Apache Ant (1.5.2) [2], and `junit.jar` in JUnit (3.8.1) [17]. For each package (Jar file), we execute `jbirth` to perform pairwise birthmark comparison of class files contained in the Jar file. For this, we set the well-known classes (see Definition 4) to be class files contained in the following general-use packages: `java, javax, org.omg, org.xml, org.w3c, org.ietf, com.sun, sun`.

The result is shown in Table 1. The first column shows the type(s) of birthmarks used to distinguish class files. The second to the last columns contain *distinction ratio* — a percentage of comparisons that are successfully distinguished by the corresponding birthmarks, in total pairwise comparisons. As seen in the table, the proposed birthmarks was able to distinguish most of class files. Note that, the more birthmarks are used together, the more the distinction ratio improves, which is very closed to 100%.

In this experiment, the proposed birthmarks could not achieve 100% of the distinction ratio. We investigated the source code of the class files that could not be distinguished. As a result, we found that these classes are:

(a) very small-inner classes that contains only one or two method calls (e.g., containing `System.exit(0)` only), or
(b) small classes with almost identical routines (which seem to be written by copy and paste, considering from adjunct comment lines).

The case (a) shows that such tiny and trivial classes do not have enough information to *characterize* themselves. For such class files, birthmarking is not appropriate to protect them from theft. However, we consider that it is not a very serious problem even if they are stolen, since such small class files hardly contain intellectual properties. For the case (b), we can say that the proposed birthmarks worked very well, since the birthmarks conclude "The one is very likely to a copy of another."

5.2 Experiment 2 (Tolerance against Transformation)

In this experiment, we evaluate the tolerance of the proposed birthmarks against a program transformation. To copy an original class file p, malicious crackers may convert p into an equivalent p' by using certain automatic tools, so that the original birthmark $f(p)$ is modified (i.e., $f(p) \neq f(p')$). Our objective here is to

Table 1. The result of distinguishing classes by birthmark

	Ant 1.5.2	BCEL 5.1	JUnit 3.8.1
Number of Class files	375	339	90
Number of Comparisons	70,125	57,291	4,005
CVFV	96.3736%	67.5778%	95.6304%
SMC	98.5055%	80.8661%	95.8801%
IS	78.0734%	77.3210%	68.0649%
UC	98.8648%	86.4533%	98.2022%
CVFV, SMC	99.3825%	81.3880%	98.9513%
CVFV, IS	99.0944%	89.7104%	97.8776%
CVFV, UC	99.7262%	87.0206%	99.4007%
SMC, IS	99.2912%	91.8468%	95.8801%
SMC, UC	99.3996%	86.6506%	98.3770%
IS, UC	99.1130%	93.0949%	98.2521%
CVFV, SMC, IS	99.6991%	91.9149%	98.9513%
CVFV, SMC, UC	99.7618%	87.0747%	99.4257%
CVFV, IS, UC	99.7532%	93.2467%	99.4007%
SMC, IS, UC	99.5522%	93.2432%	98.3770%
CVFV, SMC, IS, UC	99.7832%	93.2834%	99.4257%

evaluate how much the original birthmark is modified by a program transformation using similarity of birthmarks.

For this, we chose a tool `jarg` (Java archive grinder) [13]. `jarg` is a kind of code optimizer for Java package, which can delete unreachable code, shorten long-name identifiers, compress bytecode, and so on.

We apply `jarg` to a package `junit.jar`, and obtain an optimized package `junit_s.jar`. Then, we execute `jbirth` to measure similarity of birthmarks for all pairs of a class file p in `junit.jar` and the optimized p' in `junit_s.jar`.

Table 2 summarizes the result. We compared ninety pairs of the original and the optimized class files, by means of the proposed four types of birthmarks. The upper-half of the table shows the average, minimum and maximum values of the similarity between class files before and after the optimization. The lower-half shows a frequency table for the number of class files within a range of similarity.

It can be seen in the table that the quite large part of the original birthmarks are still preserved even after the optimization. None of pairs are below 80% of similarity. Thus, the proposed birthmark achieved a relatively strong tolerance against program transformation in this experiment. Also we can expect that the similarity of birthmarks can be utilized as a useful information, even when the original birthmark is modified by a program transformation. That is, if birthmarks $f(p)$ and $f(q)$ are different, but quite similar, we can suspect that q is a copy of p obtained by the transformation. More systematic and thoughtful discussion about the similarity of birthmarks is left to our future work.

Table 2. The result of tolerance to optimization

Number of class		90
Similarity percentage	**Average**	97.30%
	Minimum	83%
	Maximum	100%
Number of condition fulfilling	$similarity = 100$	65
	$similarity \geq 95$	73
	$similarity \geq 90$	77
	$similarity \geq 85$	88
	$similarity \geq 80$	90

6 Conclusion

In this paper, we proposed a method to provide a reasonable evidence of theft of Java class files, by means four types of birthmarks: CVFV, SMC, IS and UC. We also conducted two experiments to evaluate the proposed birthmarks. The results showed that the proposed birthmarks could successfully distinguish (non-copied) class files in practical Java packages except some tiny classes, and that they achieved relatively good tolerance to program optimization.

Compared to watermarking techniques, advantage of the proposed method is that it is simple and easy to use. Moreover, since a birthmark is a native characteristics of a program, it can be used as solid and un-detachable information without any extra effort.

Limitation is that: even if the same birthmarks $f(p)$ and $f(q)$ are extracted, we can just suspect that q is *very likely to* a copy p. In this sense, birthmarks might be bit weaker evidence than watermarks. Since watermarks and birthmarks are not exclusive methods, combined use of them would cover the limitation of each other.

Finally, we summarize our future work. We plan to evaluate tolerance of the birthmarks against stronger program transformations (e.g., obfuscation). Also, we want to clarify the relevance of the similarity to the copy relation, through more experiments. Investigation of other types of birthmarks is also an interesting issue.

References

1. Antoniol, G., Villano, U., Merlo, E., Penta, M.: Analyzing Cloning Evolution in the Linux Kernel, Information and Software Technology, **44**(13), (2002), 755-765
2. Apache Ant `http://ant.apache.org/`
3. Baker, B.: A Program for Identifying Duplicated Code, In Proc. 24th Symposium on the Interface: Computing Science and Statistics, (1992) 49–57
4. Baxter, I., Yahin, A., Moura, L., Sant'Anna, L., Bier, L.: Clone Detection Using Abstract Syntax Trees, In Proc. IEEE Int'l Conf. on Software Maintenance (ICSM'98), Bethesda, Maryland, (1998) 368–377

5. Chen, X., Li, M., Mckinnon, B., Seker, A.: A Theory of Uncheatable Program Plagiarism Detection and Its Practical Implementation, SID Website at `http://dna.cs.ucsb.edu/SID/` (2002)
6. Collberg, C., Thomborson, C.: Software Watermarking: Models and Dynamic Embeddings, In Proc. 26th ACM SIGPLAN-SIGACT Symposium on Principle Languages (POPL'99), San Antonio, Texas, (1999) 311–324
7. Collberg, C., Thomborson, C.: Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection, IEEE Transactions on Software Engineering **28**(8) (2002) 735–746
8. Epson Pulls Linux Software Following GPL Violations `http://slashdot.org/article.pl?sid=02/09/11/2225212`
9. GNU General Public License `http://www.gnu.org/copyleft/gpl.html`
10. Haruaki, T., Kanzaki, Y., Nakamura, M., Monden, A., Matsumoto, K.: A Method for Extracting Program Fingerprints from Java Class Files, The Institute of Electronics, Information and Communication Engineers Technical Report **29** (2003) 127–133
11. Jad - The Fast Java Decompiler `http://kpdus.tripod.com/jad.html`
12. Jakarta BCEL `http://jakarta.apache.org/bcel/`
13. Jarg - Java Archiver Grinder `http://jarg.sourceforge.net/index.en`
14. Java 2 Platform, Standard Edition (J2SE) `http://java.sun.com/j2se/`
15. jbirth `http://se.aist-nara.ac.jp/jbirth/`
16. jmark: A Lightweight Tool for Watermarking Java Class files `http://se.aist-nara.ac.jp/jmark/`
17. JUnit `http://www.junit.org/`
18. Kamiya, T., Kusumoto, S.,Inoue, K.: CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code, IEEE Trans. on Software Engineering, **28**(7), (2002), 654–670
19. Monden, A., Iida, H., Matsumoto, K., Inoue, K., Torii, K.: Watermarking Java Programs, International Symposium on Future Software Technology'99 (ISFST'99), Nanjing, China, (1999) 119–124
20. Monden, A., Iida, H., Matsumoto, K., Inoue, K., Torii, K.: A Practical Method for Watermarking Java Programs, The 24th Computer Software and Applications Conference, Taipei, Taiwan, (2000) 191–197
21. Pirates Experience Office XP `http://www.wired.com/news/business/0,1367,42402,00.html`
22. Sandmark: A Tool for the Study of Software Protection Algorithms `http://cgi.cs.arizona.edu/~sandmark/sandmark.html`
23. Verco, K., Wise, M.: Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Counting Systems, In Porc. of 1st Ausutralian Conference on Computer Science Education, Sydney, Australia, (1996), 86–95.
24. Wise, M.: YAP3: Improved Detection of Similarities in Computer Program and Other Texts, Proceedings of 27th SCGCSE Technical Symposium, Philadelphia, USA, (1996), 130–134.

```java
import java.io.*;
import org.apache.tools.ant.Project;
import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.DirectoryScanner;
import org.apache.tools.ant.taskdefs.MatchingTask;

public class ViewSerialVersionTask extends MatchingTask{
  private static final String DEFAULT_BASE_DIR = ".";
  private File baseDir;

  public ViewSerialVersionTask(){
  }
  public void setBasedir(File baseDir){
    this.baseDir = baseDir;
  }
  public void execute() throws BuildException{
    if(baseDir == null) baseDir = new File(DEFAULT_BASE_DIR);
    DirectoryScanner scanner = getDirectoryScanner(baseDir);
    String[] list = scanner.getIncludedFiles();
    for(int i = 0; i < list.length; i++){
      log(list[i], Project.MSG_DEBUG);
      printSerialVersionUID(list[i]);
    }
  }
  private void printSerialVersionUID(String target){
    File inFile = new File(baseDir, target);
    if(!target.endsWith(".class")) return;
    try{
      String className = target.substring(0, target.length() - 6);
      className = className.replace('/', '.').replace('\\', '.');
      Class c = Class.forName(className);
      if(checkSerializable(c)){
        ObjectStreamClass osc = ObjectStreamClass.lookup(c);
        long serialVersionUID = osc.getSerialVersionUID();
        System.out.println(c.getName() + ": " + serialVersionUID);
      }
    } catch(Exception e){
      throw new BuildException(e.getMessage());
    }
  }
  private boolean checkSerializable(Class c){
    Class[] interfaces = c.getInterfaces();
    for(int i = 0; i < interfaces.length; i++){
      if("java.io.Serializable".equals(interfaces[i].getName()))
      return true;
    }
    return false;
  }
}
```

Fig. 1. Example of Java source code (`serialver` ported for Apache Ant)