IEICE TRANS. INF. & SYST., VOL.E88–D, NO.9 SEPTEMBER 2005

PAPER

Java Birthmarks ——Detecting the Software Theft——

Haruaki TAMADA[†a)], ***Student Member***, **Masahide NAKAMURA**[†], **Akito MONDEN**[†], ***and*** **Ken-ichi MATSUMOTO**[†], ***Members***

SUMMARY To detect the theft of Java class files efficiently, we propose a concept of *Java birthmarks*, which are unique and native characteristics of every class file. For a pair of class files p and q, if q has the same birthmark as p's, q is suspected as a *copy* of p. Ideally, the birthmarks should satisfy the following properties: (a) *preservation* – the birthmarks should be preserved even if the original class file is tampered with, and (b) *distinction* – independent class files must be distinguished by completely different birthmarks. Taking (a) and (b) into account, we propose four types of birthmarks for Java class files. To show the effectiveness of the proposed birthmarks, we conduct three experiments. In the first experiment, we demonstrate that the proposed birthmarks are sufficiently robust against automatic program transformation (93.3876% of the birthmarks were preserved). The second experiment shows that the proposed birthmarks successfully distinguish non-copied files in a practical Java application (97.8005% of given class files were distinguished). In the third experiment, we exploit different Java compilers to confirm that the proposed Java birthmarks are core characteristics independent of compiler-specific issues.

key words: *copyright protection, software protection, watermark, birthmark*

1. Introduction

Today, an enormous number of software products are developed and distributed all over the world. The recent advancement of the Internet dramatically has improved easy and fast distribution of software. Unfortunately, however, there are many cases of *software theft* reported [1]. Software theft copies the original software, then uses the copy for other purposes without keeping the copyright notice. Typical scenarios include:

- Defeat of a license check, duplicating a whole product, and selling or distributing the copies (called software piracy).
- Stealing a part of a product (*e.g.* modules and code fragments) and reusing it in other products without permission.

In software theft, many incidents have been reported, for example, distribution of WAREZ (pirated software) whose license checks were defeated [2], SCO Group's lawsuit against IBM for ownership violation [3], GPL infringement [4], and copyright infringement of freeware/shareware [5]. Software theft can cause severe damage to the software industries; hence, companies must protect their own intellectual properties from theft.

However, protecting software from such theft is not easy. Since enormous amounts of software products are distributed today, even detecting "suspected copies" is quite difficult, unless the product is well-known to the public. Moreover, each product generally consists of many modules and data files. Suppose that an adversary steals only some modules, builds them into his or her own code, and distributes this "new" code without the source code. Detection of the theft becomes much more difficult because, proving the new code is a copy using manual binary analysis generally requires a significant amount of skill and high costs.

This situation has become worse in the recent trend of *Java applications* [6], [7]. A Java application is composed of a collection of *class files*. A class file is an atomic execution module containing platform-independent binary data (called *bytecode*). Due to its portability, a number of class files are distributed for various platforms. Also, rigorous specification of the Java VM [8] leads to the development of powerful decompilers (e.g. jad [9]). Therefore, software crackers can relatively easily reverse-engineer and steal class files, and then reuse these class files as if they were the original developers themselves. Thus, the theft of Java class files is easy to perform, but difficult to detect.

To cope with this problem, this paper presents an easy-to-use method to provide reasonable evidence for the theft of Java programs. Specifically, we propose *Java birthmarks* to support the efficient detection of class files that are quite similar to (or exactly the same as) each other. Intuitively, a birthmark of a Java class file is the set of unique characteristics that the class file *originally* possessed. If class files p and q have the same birthmark, q is very likely to be a stolen copy of p (and vice versa).

Ideally, the birthmark should tolerate a certain extent of *program transformation* [10] since the clever crackers may alter or modify the original class file to hide the fact of theft. Hence, the birthmark must have characteristics that cannot be modified easily. In other words, changing the birthmark should make the class file malfunction, or make it completely different from the original. On the other hand, for class files that are independently implemented, the birthmark must be able to *distinguish* among them even when

Manuscript received November 5, 2004.
Manuscript revised March 22, 2005.
[†]The authors are with the Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma-shi, 630–0101 Japan.
a) E-mail: harua-t@is.naist.jp
DOI: 10.1093/ietisy/e88–d.9.2148

there is no theft.

Taking these issues into account, we propose four kinds of birthmarks: (1) constant values in field variables (*CVFV birthmark*), (2) a sequence of method calls (*SMC birthmark*), (3) an inheritance structure (*IS birthmark*), and (4) used classes (*UC birthmark*). These birthmarks are essential characteristics of each class file, which can be extracted without a source code. Based on the proposed method, we have implemented a birthmark extraction tool, called `jbirth`[11].

This paper conducts three experiments to show the effectiveness of the proposed birthmarks. In the first experiment, we evaluate the tolerance against the program transformation by exploiting practical Java optimizers and obfuscators (`CodeShield`[12], `jarg`[13], `SmokeScreen`[14] and `ZKM`[15]). Introducing a notion of *similarity* of birthmarks, we demonstrate that the proposed birthmarks cannot be altered easily. The result shows that the similarity of birthmarks of every class file before and after the transformation is as high as 93.3876% on the average.

The second experiment evaluates the distinction property of the birthmarks with practical Java applications (Ant [16], BCEL [17], JUnit [18]). These well-known applications are assumed to be built by open-source communities whose members do not engage in theft. The proposed birthmark distinguished 97.8005% of all class files. The remaining class files were either tiny classes or classes written by "copy and paste".

The third experiment investigates class files obtained by *different Java compilers* (`javac`, `jikes`[19], `kjc`[20]) to see the effect of reverse-engineering on the birthmarks. The result shows that the similarity of birthmarks of every class file generated by a different compiler is 90.1959% on average.

As a result, the proposed birthmarks are shown to be simple but powerful native signatures of Java class files, which can be extensively utilized for the detection of theft of Java applications. This paper was originally published as a conference paper presented in [21]. Changes were made to this version, most significantly, the addition of the new experiment and the qualitative discussion on the applicability and limitation of the proposed birthmark. We believe that these new results clarify the practical applications of the proposed birthmarks.

2. Related Work

Software watermarking (often called *software fingerprinting*) is a well-known technique used to provide a way to prove ownership of stolen software. Therefore, it may be used for our objective. Watermarking is basically used to embed stealthy information in a piece of software, such as a software developer's copyright notation or a unique identifier of software, in a static manner [22]–[24], or in a dynamic manner [25]–[27]. Unfortunately, watermarking is not always feasible because it requires software developers to embed a watermark *before* releasing the software. Thus, proofs cannot be given for already-released software without watermarks. In addition, strictly speaking, to protect all the modules in a software package, we need to embed watermarks into all of these modules. This is generally difficult to meet when the number of modules is large. Our birthmark approach provides a way to detect stolen software without embedding any additional information beforehand.

The most commonly used technique to detect a suspected copy is software similarity computation [28]–[31], which is generally used for *plagiarism detection* in programming classes. A plagiarized program is defined as a program that has been reproduced from another program with only a small number of editions, and with no detailed understanding of the program required [32]. In order to detect plagiarized programs, various methods for similarity computation have been proposed based on attribute counting [33], structure metrics [28], [30], [31], and Kolmogorov complexity [29]. Unfortunately, since these methods require the source code of software to compute the similarity, they are not applicable in our problem setting where software products are usually distributed without a source code. Moreover, these techniques did not consider attacks by practical code transformation tools, such as software optimizers [13] and obfuscators [12], [14], [15]. Our birthmarks are intended to be used for Java class files without their source code, and to be resilient against code transformation attacks. We also define the similarity of birthmarks to utilize our birthmarks.

Another way to detect software theft is to find code clone pairs between two software products [34], [35]. Code clones are exact or nearly exact duplicate lines of code within the source code. In [35], by using a code clone detection tool called *CCFinder*, Kamiya et al. found that `sys/net/zlib.c` of FreeBSD and `drivers/net/zlib.c` of Linux are almost identical. Such code clone techniques are useful for detecting suspected copies; however, they are also susceptible to code transformation attacks.

Finally, we consider authorship analysis methods [36], [37]. In [36], programming style metrics and programming layout metrics are used to identify the author of a source code. In [37], Spafford and Weeber suggest that it might be feasible to analyze code remnants in executable code, such as data structures and algorithms and choice of system and library calls made by the programmer, which are typically the remains of a virus or Trojan horse, and identify its author. Such identifying information are called a "software birthmark" by Grover [38] although we propose its formal definition in Sect. 3.1. These previously-proposed birthmarks are rather out of date and are not feasible in today's Java class files; nonetheless, we took them into account to propose our birthmarks, e.g. the choice of library calls in [37] are a part of our SMC (Sequence of Method Calls) birthmark.

3. Java Birthmarks

3.1 Definition

To make our discussion clearer, this subsection formulates a notion of a birthmark. We start with formulation of the *copy relation* of programs.

Definition 1 (Copy Relation): Let *Prog* be a set of given programs. Let $\equiv_{cp}$ denote an 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). The relation $\equiv_{cp}$ is called a *copy relation*.

The criteria for whether or not q is a *copy* of p can vary depending on the context. For example, each of the following criterion is 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) $p \equiv_{cp} q \Rightarrow q \equiv_{cp} p$, (Transitive) $p \equiv_{cp} q \wedge q \equiv_{cp} r \Rightarrow p \equiv_{cp} r$.

All the above properties meet well the intuition of a 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 to 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 a *birthmark* of p under $\equiv_{cp}$ iff both of the following conditions are satisfied.

Condition 1: $f(p)$ is obtained 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 extra information and is required for p to run. Hence, extracting a birthmark does not require extra code as watermarking does. Condition 2 states that the same birthmark has to be obtained from copied programs. By 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 will satisfy the following properties.

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

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

These properties strengthen Condition 2 of Definition 2. Property 1 specifies the *preservation property* of the birthmark against program transformation [10]. We believe that clever crackers may try to modify birthmarks by transforming the original program into an equivalent one to hide the fact of theft. There are several automated tools used to perform the transformation, involving program *obfuscators* and *optimizers* (e.g., [13], [15]). These tools can be used as a means of attack against the birthmarks. Property 1 specifies that the same birthmark must be obtained from p and converted to p'. However, there exist many ways to transform a program into an equivalent one. Hence, in reality, it is difficult to extract strong enough birthmarks to perfectly satisfy Property 1.

Property 2 specifies the distinction property of the birthmark, stating that: even though the specification of p and q is the same, if implemented separately, different birthmarks should be extracted. In general, the detail of two independent programs is almost never completely the same. However, in the case that p and q are both *tiny* programs, extracted birthmarks could become the same, even if p and q are written independently. Those properties should be tuned within an allowable range at the user's discretion.

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

3.2 Proposed Birthmarks

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 $f(q)$. If $f(p) \neq f(q)$, then $p \not\equiv_{cp} q$, so we conclude that q is not a copy of p.

As for the above f, this subsection presents four methods which 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).

Simply for added 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 code of given class files is not necessarily available.

3.2.1 Constant Values in Field Variables (CVFV)

A class in Java often has *field variables* to store static and/or dynamic attributes. If the field variables are initialized to

```
package jp.ac.aist_nara.se.tama.ant.taskdefs;

import org.apache.tools.ant.Task;
import org.apache.tools.ant.Project;
import org.apache.tools.ant.BuildException;

public class Echo extends Task{
  public String message = "";
  public int logLevel = Project.MSG_DEBUG;

  public void setMessage(String message){
    this.message = message;
  }

  public String getMessage(){
    return message;
  }

  public void setLevel(String level){
    level = level.toLowerCase();
    if(level.equals("debug"))
      logLevel = Project.MSG_DEBUG;   // 4
    else if(level.equals("verbose"))
      logLevel = Project.MSG_VERBOSE; // 3
    else if(level.equals("info"))
      logLevel = Project.MSG_INFO;    // 2
    else if(level.equals("warn"))
      logLevel = Project.MSG_WARN;    // 1
    else if(level.equals("error"))
      logLevel = Project.MSG_ERR;     // 0
    else
      logLevel = Project.MSG_DEBUG;   // 4
  }

  public int getLevel(){
    return logLevel;
  }

  public void execute()
                  throws BuildException{
    log(message, getLevel());
  }
}
```

Fig. 1 Example of Java source code (simple echo task for Apache Ant).

be certain constant values upon their declaration, then these initial values are essential information for determining the way of object instantiation. Modifying these values is dangerous since the modification may change the 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 $v_1, v_2, \ldots, v_n$ 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"). Therefore, the sequence $((t_1, a_1), (t_2, a_2), \ldots, (t_n, a_n))$ is called a *CVFV birthmark* of p, denoted by $CVFV(p)$.

The CVFV birthmark of the program in Fig. 1 is:
(`java.lang.String`, "")
(`int`, 4)

3.2.2 Sequence of Method Calls (SMC)

In Java, general-purpose functions are already implemented as methods of *well-known classes*, such as J2SDK [7] and the Jakarta project [39]. A class often calls one or more methods of these well-known classes. We assume that the sequence of the method calls characterizes the class well due to the following two reasons:

First, modifing the sequence automatically is difficult for crackers because the modification crashes the dependencies between the methods. Second, replacing a method in the sequence with another takes significant effort because creating the new method 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 $m_1, m_2, \ldots, m_n$ be a sequence of method m_i's appearing in p in this order (this is not necessarily the execution order), where m_i belongs to a class in C. Hence, the sequence $(m_1, m_2, \ldots, m_n)$ is called a *SMC birthmark* of p, denoted by $SMC(p)$.

Let C be a set of all classes in J2SDK or the Jakarta Project. In this case, the SMC birthmark of the program in Fig. 1 is:

```
org.apache.tools.ant.Task(),
String String#toLowerCase(),
boolean String#equals(Object),
boolean String#equals(Object),
boolean String#equals(Object),
boolean String#equals(Object),
boolean String#equals(Object),
void org.apache.tools.ant.Task#log(String, int)
```

3.2.3 Inheritance Structure (IS)

Java is an object oriented programming language. Every class in Java has a hierarchy of *inheritance structures* except `java.lang.Object`, which is the root class of all classes. 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 characteristic of p. However, the sequence of classes may contain both well-known classes and user-made classes. Since the user-made classes are relatively easy to alter, 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 $c_1, c_2, \ldots, c_n$ 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 the root of a class (java.lang.Object). If c_i does not belong to a class in C, we replace c_i with "null." Therefore, the resultant sequence $(c_2, c_3, \ldots, c_n)$ is called an *IS birthmark* of p, denoted by $IS(p)$.

Let C be a set of all classes in J2SDK or the Jakarta Project. Then the IS birthmark of the program in Fig. 1 is:

`org.apache.tools.ant.Task`,
`org.apache.tools.ant.ProjectComponent`,
`java.lang.Object`.

3.2.4 Used Classes (UC)

A class (say p) generally *uses* other classes to implement new functions by combining existing features of the other classes. These external classes in p appear as a superclass, a return type, argument types of methods, or 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 difficult for crackers to alter them. Hence, the set of used (well-known) classes is considered to be a unique signature characterizing 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, or u's, such that u is used in p and $u \in C$. Let $u_1, u_2, \ldots, u_n$ ($u_i \in U$) be a sequence obtained by arranging all elements in U in alphabetical order. Then, the sequence $(u_1, u_2, \ldots, u_n)$ is called a *UC birthmark* of p, denoted by $UC(p)$.

Let C be a set of all classes in J2SDK or the Jakarta Project. Then, the UC birthmark of the program in Fig. 1 is:
`java.lang.String`,
`org.apache.tools.ant.Task`,
`org.apache.tools.ant.Project`,
`org.apache.tools.ant.BuildException`.

3.3 Similarity of Birthmark

Each of the proposed birthmarks is in the form of a sequence. Suppose that we have a pair of birthmarks $f(p) = (p_1, \ldots, p_n)$ and $f(q) = (q_1, \ldots, q_n)$ 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.

Thus, the comparison of birthmarks with equivalence only is somewhat overly strict, which may make the birthmarks too sensitive against the attack of program transformation. To cope with this problem, here we introduce the *similarity* of birthmark. The similarity is a percentage of elements matched among $f(p)$ and $f(q)$ in the total elements in the birthmark (sequence).

Definition 7 (Similarity): Let $f(p) = (p_1, \ldots, p_n)$ and $f(q) = (q_1, \ldots, q_n)$ 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: $s/n \times 100$.

4. Evaluation

In this section, we evaluate the proposed birthmarks from the following viewpoints: (a) *distinction property*, (b) *preservation property*, and (c) *dependency on compilers*. As a tool support, we have implemented a software application called `jbirth` [11]. In following experiments, we set the set C of well-known classes (see Definition 4) to be all classes in J2SDK SE 1.4.

4.1 `jbirth`: A Tool for Java Birthmarks

`jbirth` is a tool for extracting and comparing the proposed Java birthmarks [11]. It is written in Java (J2SDK SE 1.4 [7]) with Byte Code Engineering Library (BCEL 5.1 [17]), comprising about 13,000 lines of code. Figure 2 shows a screenshot of `jbirth`. 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 future birthmarks.

As a performance indicator, we have measured the execution time taken for extracting the proposed four kinds of birthmarks from practical Java packages: bcel-5.1.jar, ant.jar, and jbirth.jar. The measurement was performed on a Windows PC with Pentium 4-3.00 GHz and 496 MB RAM. Table 1 shows the result. It can be seen that the time taken for extracting the birthmarks is sufficiently short from the practical viewpoint (0.0180s per a class file in this experiment). Thus, using `jbirth`, one can extract the birthmarks from any Java class files quite easily and efficiently. `jbirth` is freely available at [11].

4.2 Experiment 1: Preservation Performance

Using `jbirth`, we first evaluate the *preservation property* (See Property 1 in Sect. 3.1) of the proposed birthmarks. According to their definitions, the proposed birthmarks are robust enough to tolerate such simple cases (a)-(c) mentioned below Definition 1. However, clever crackers may use certain automatic tools and convert p to an equivalent q in a more sophisticated way. Our concern here is how much of the original birthmarks is altered by the program transformation tools.

In this experiment, we exploited the following tools: `ZKM` [15], `Smokescreen` [14], `CodeShield` [12], and `jarg` [13]. Each of the tools performs a unique program transformation for given Java class files. The first three are known as the program *obfuscator*, which converts the original class file into an equivalent one that is more difficult to analyze. `jarg` is a program *optimizer*, which optimizes the redundant part of the class file.

More specifically, all of the tools implement the *name obfuscation* and *elimination of debug information* for Java class files. The name obfuscation changes meaningful symbol names (i.e., class, field and method names) to meaningless ones, which makes the decompiled source code harder to understand. `ZKM`, `Smokescreen` and `CodeShield`

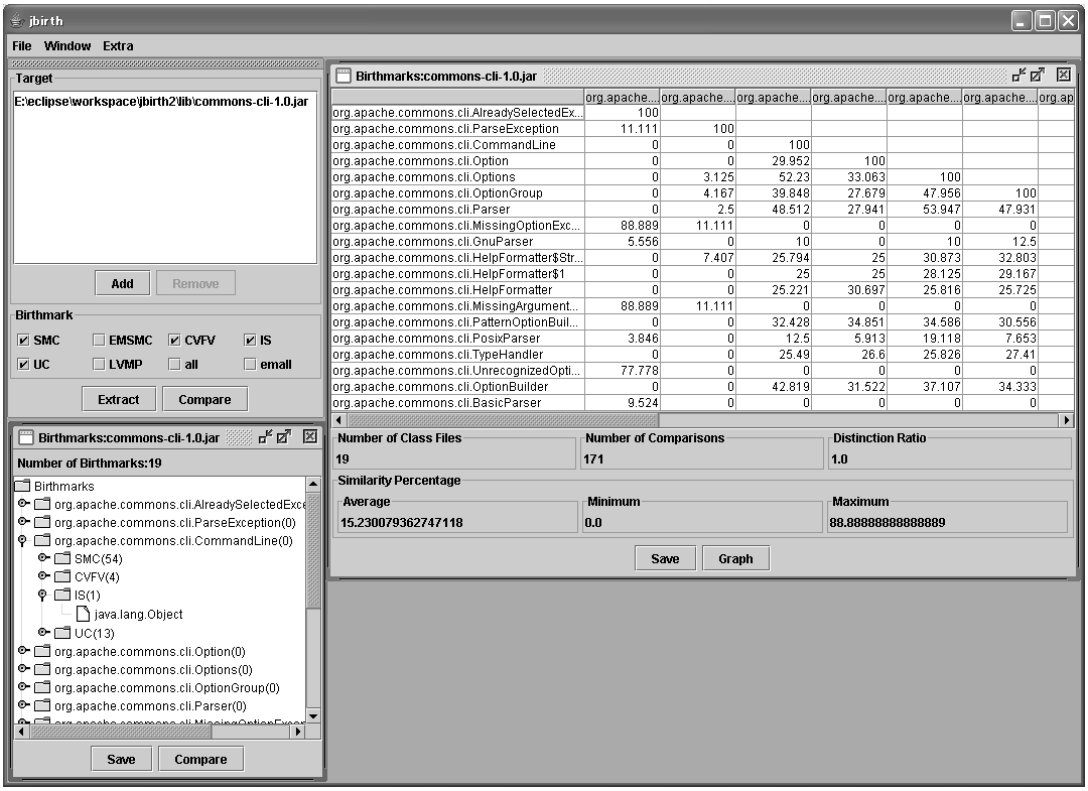


Fig. 2 A screenshot of `jbirth`.

Table 1 Execution performance.

Package	# of classes	Size	Execution time
bcel-5.1.jar	339 classes	515,920 bytes	4.675 s
ant.jar	487 classes	958,858 bytes	8.240 s
jbirth.jar	134 classes	213,594 bytes	2.239 s

Table 2 The result of Experiment 1.

		ZKM	Smokescreen	jarg	CodeShield
Similarity Percentage	**Average**	94.4096%	90.9628%	98.9016%	89.2766%
	Minimum	50%	27%	82%	57%
	Maximum	100%	100%	100%	99%

adopt *flow obfuscation*, which scrambles the control flow without changing the original runtime behavior. `jarg` and `Smokescreen` support optimization of unreachable code and unused fields and methods. `ZKM` has unique features, such as *string encryption*, which encrypt string literals in class files, and then add code fragments to decrypt the string at runtime.

We applied each tool to a package Apache Ant (version 1.5.4, ant.jar) [16] with the strongest obfuscation (or optimization) level, and obtained its transformed package. Next, we executed `jbirth` to measure the similarity of birthmarks for all pairs of a class file in `ant.jar` and the transformed version.

Table 2 summarizes the result. We compared 376 pairs of the original and the transformed class files by means of the proposed four types of birthmarks. Figure 3 depicts the

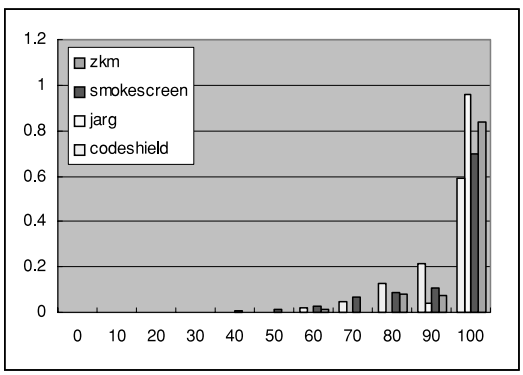


Fig. 3 Birthmark similarity between original and transformed class files.

frequency distribution, where the horizontal axis represents the similarity, and the vertical axis plots the number of pairs of class files with the corresponding similarity, normalized by the total number of comparisons. In the results for all the tools, the majority of the original birthmarks were still preserved even after the sophisticated program transformations. Thus, the proposed birthmark achieved a practically strong robustness against program transformation in this experiment. Also, the similarity varies slightly, depending on the obfuscation tool applied. It appears that the method of program transformation has different effects on the birthmarks.

Although the combined use of the four birthmarks reflects a stronger characteristic of a class file, it is also interesting to evaluate each birthmark *individually*. Figure 4 shows the preservation performance in Experiment 1, with respect to the individual birthmarks. In the figure, X-axis represents the birthmarks, Y-axis depicts the frequency of each birthmark, and Z-axis plots the similarity. From the figure, it can be seen that most birthmarks achieve 100% similarity.

However, we can see that the UC birthmark is a bit fragile against the transformation with `CodeShield`. Through the investigation, we found that `CodeShield` translates the conditional branches specified by `if` (or `for`) into `try` and `catch`, by adding dummy exception classes. For example, the statement `if(EXP1){ s1; } else{ s2; }` is transformed into `try{ throw e; } catch(exp_1 e1){ s1; } catch(exp_2 e2){ s2;}`. These exception classes add extra used-classes to the original bytecode, which decreases the performance of UC birthmark. In general, an exception class can be easily altered with another exception class. Therefore, we should have excluded any exception class from the set of well-known classes C. If we exclude any exception class from C, the performance of UC birthmark can be improved significantly, as shown in Fig. 5.

4.3 Experiment 2: Distinction Performance

Next, we evaluate the *distinction property* (See Property 2 in Sect. 3.1) of the proposed birthmarks. Usually, all class files in each practical Java application are supposed to be *different* from each other. If one package contains some class files that are exactly the same, this represents a redundant and inefficient class design. Hence, we evaluate how many class files in a Java package can be distinguished from each other by the proposed birthmarks.

As target applications, we chose the following products: Apache Ant (1.5.4) [16], Jakarta BCEL (5.1) [17], JUnit (3.8.1) [18], and `jbirth` [11]. For each Jar file, we executed `jbirth` to perform pairwise birthmark comparison of class files contained in the Jar file. Note that the objective of Experiment 2 is to compare *class files* within each single package, but *not* to to see the similarity between *packages*.

The result is shown in Table 3 with using proposed four birthmarks together. In the table, the *distinction ratio* represents a percentage of pairs of class files that are successfully distinguished, in the total pairs compared. The table also includes average, minimum, and maximum values of the similarity. As seen in the distinction ratio, the proposed birthmarks were able to distinguish most of class files. Figure 6 shows the frequency distribution of the similarity, where the horizontal axis represents the similarity, and the vertical axis

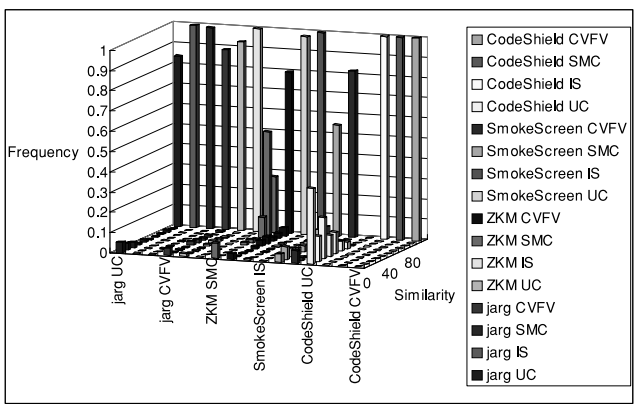


Fig. 4 Similarity w.r.t. individual birthmark.

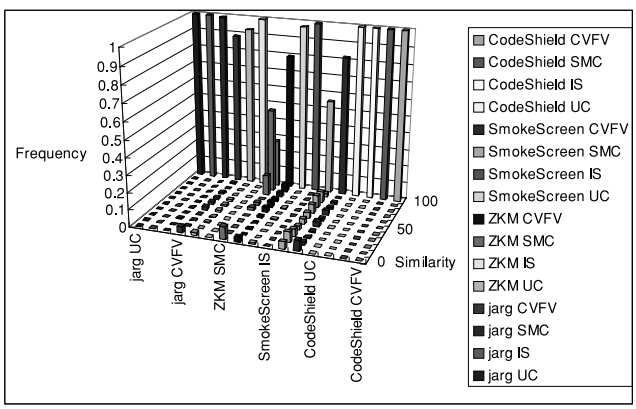


Fig. 5 Similarity w.r.t. individual birthmark (after exception classes are excluded).

Table 3 The result of Experiment 2.

		Ant 1.5.4	BCEL 5.1	JUnit 3.8.1	jbirth
Number of Class Files		376	339	90	63
Number of Comparisons		70,500	57,291	4,005	1891
Distinction Ratio		99.7872%	93.29389%	98.3770%	99.7440%
Similarity Percentage	**Average**	8.4035%	12.1585%	14.4709%	9.3815%
	Minimum	0%	0%	0%	0%
	Maximum	100%	100%	100%	100%

plots the number of pairs of class files normalized by the number of comparisons. For most pairs of class files, the similarity is below 20%. This implies that different class files have significantly different birthmarks.

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 contain 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).

The above (a) indicates that such tiny and trivial classes do not have enough information to characterize themselves. Such class files cannot be protected from theft by the birthmarks. However, we believe that it is not a serious problem even if they are stolen since such small class files seldom contain intellectual properties. As for case (b), the proposed birthmarks worked very well since the birthmarks confirm that one is very likely to be a copy of another.

Figure 7 shows the birthmark-wise evaluation on the similarity. In the figure, X-axis represents the birthmarks, Y-axis depicts the frequency of each birthmark, and Z-axis plots the similarity. It can be seen that the IS birthmark sometimes failed to distinguish completely different class files. As we examined the situation, we found that most of those classes are immediate children (i.e., direct subclasses) of the root class `Object`. Also, there was no surprising to see the CVFV birthmarks failed to distinguish classes that have *no* field variables. Such classes do not have enough information to characterize themselves solely by the IS or CVFV birthmarks.

Thus, using the CVFV (or IS) birthmark only may yield a very *short* birthmark sequence. This is obviously the limitation of the IS and CVFV birthmarks, but is a trade-off against the simplicity of birthmark computation. An idea to improve the distinction performance of the CVFV and IS birthmarks is to discard such short birthmarks from the similarity evaluation, or to use the CVFV and IS with other birthmarks together.

4.4 Experiment 3: Different Compilers

This experiment examines class files produced by different compilers. Other Java compilers in addition to `javac` exist. Java compilers such as `jikes` [19] and `kjc` [20] have been developed and distributed. Indeed, different compilers cause differences in the generated bytecode. Our objective here is to validate whether or not the proposed birthmarks are *core characteristics* of class files that are independent of the compiler-specific bytecodes.

In this experiment, we chose BCEL (5.1) as the target application. We compiled the source files of BCEL with `javac`, `jikes`, and `kjc`, and obtained three sets of class files. Next, using `jbirth` we measured the similarity of birthmarks between any pair of the three sets.

Table 4 shows the result. The BCEL package consists of 339 class files. Hence, for each pair of compilers, we measured the similarity through 339 comparisons, using all four types of the birthmarks. Figure 8 shows the frequency distribution of similarity. The horizontal axis represents the similarity, and the vertical axis plots the number of pairs of class files normalized by the total number of comparisons.

For all the compilers, most class files with the same source have similar birthmarks regardless of the compilers used. For the class files with low similarity, we conducted a manual investigation. As a result, we found differences in the sequence of methods in some class files. Specifically, class files generated by `kjc` and `javac` contain the methods

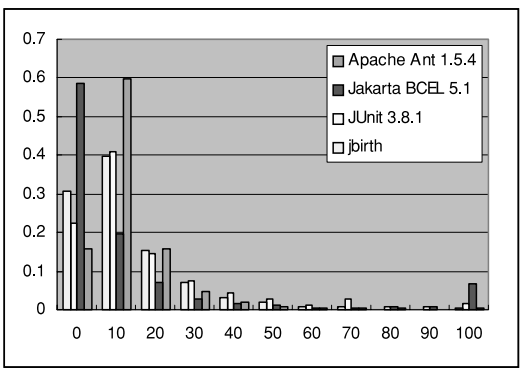


Fig. 6 Birthmark similarity among different class files.

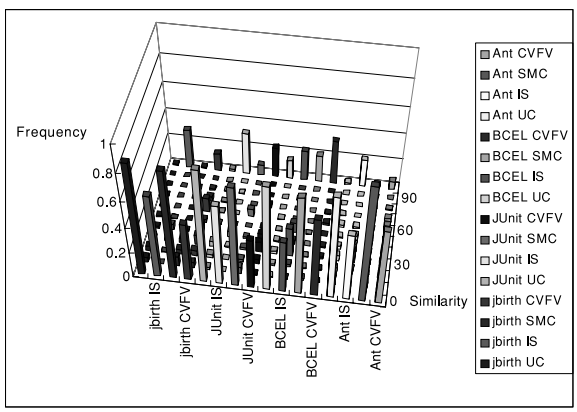


Fig. 7 Similarity w.r.t. individual birthmark.

Table 4 The result of Experiment 3.

Pair of Compilers		javac, jikes	javac, kjc	jikes, kjc
Similarity Percentage	**Average**	89.5182%	93.2469%	87.8225%
	Minimum	45.9451%	33.3333%	33.3333%
	Maximum	100%	100%	100%

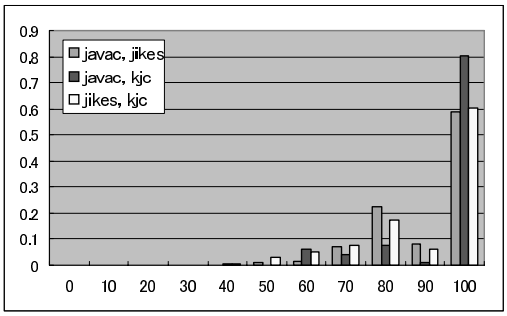


Fig. 8 Birthmark similarity among class files obtained by different compilers.

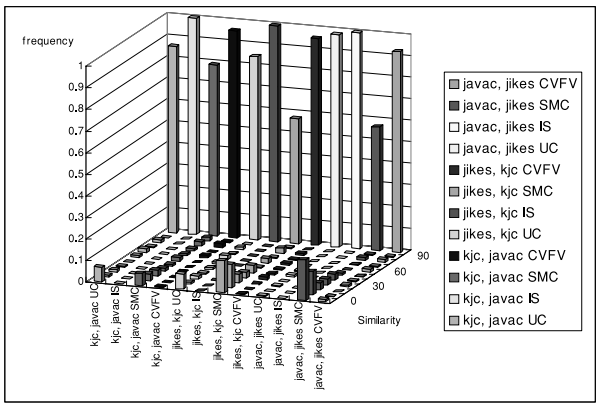


Fig. 9 Similarity w.r.t. individual birthmark.

in the same order as described in their source files. However, `jikes` seems to generate the methods in a different order on rare occasion.

To some extent, the result also indicates the robustness of the proposed birthmarks against an attack of reverse-engineering and re-compilation, although there is no existing *decompiler* that can perfectly reproduce the valid source code from any class file [40].

Figure 9 shows the result with respect to the individual birthmarks. It can be seen that the SMC birthmark is often scrambled by `jikes`. This seems to be due to the code optimization of `jikes`, which re-arranges the (static) order of methods defined in the source code into a different order in the bytecode. For example, suppose that we apply `jikes` to compiling the source code in Fig. 1. `jikes` may generate the bytecode where `setLevel()` comes after `execute()`, which ruins the SMC birthmark (see the example in Definition 4).

Currently, the SMC birthmark does not support this kind of code optimization. However, the problem could be solved as follows. For a class p containing multiple methods, we first derive an SMC birthmark *locally* in each method, and then gather all of the method-wise SMC birthmarks in a *set* $SMCSET(p)$. When comparing the class p with another q, we compare $SMCSET(p)$ and $SMCSET(q)$ as a set, ignoring the order of the elements in each set. The extension of `jbirth` to support such grouped comparison of the SMC birthmarks is left to our future work.

5. Discussion

5.1 Practical Use

Suppose that we obtain the identical birthmark from a pair of class files p and q (i.e., $f(p) = f(q)$). Then, a question arises: "*can we prove that q is a stolen copy of p?*" Theoretically, the answer is NO, since our definition of the birthmark (see Definition 2) does *not* require $f(p) = f(q) \Rightarrow p \equiv_{cp} q$. Even if p and q are independent (i.e., $p \not\equiv_{cp} q$), there is a possibility that $f(p) = f(q)$ holds. Hence, one might think that the birthmarks are too weak to prove the theft.

However, from a practical point of view, the birthmarks provide a powerful clue to proving that q is a copy of p. As seen in Experiment 2, class files whose birthmarks happen to match are usually simple and small classes. In reality, the adversary would steal relatively complex (thus large) classes rather than such simple ones. If the sizes of p and q are both reasonably large, $f(p) = f(q)$ leads to a greater potential that $p \equiv_{cp} q$ holds. Therefore, our birthmarks can be used to efficiently *detect* suspected copies from enormous combinations of class files.

In the case that the birthmarks can be tampered with (as in Experiment 1), introducing the birthmark similarity makes theft detection more reliable than using only the birthmark equivalence. The difference between Figs. 3 and 6 is rather interesting. In the figures, independent class files have quite different birthmarks. Class files obtained from their originals have birthmarks very similar to the originals'. Therefore by setting an *appropriate threshold* of the similarity, the proposed birthmarks can provide considerably reliable evidence for the copied class files.

5.2 Manual Modification

Indeed, the adversary would try to modify the birthmarks by *manual hacking*. For example, to erase the CVFV birthmark, the attacker may tamper with the initial value of a field variable, and add extra statements to adjust the value, such as converting "`int i=5;`" into "`int i=1; i+=4;`" [41]. This sort of transformation can modify the original birthmarks significantly. However, we are optimistic about the vulnerability against such manual modification due to the following reasons.

- The adversary must be highly skilled in Java bytecode to modify a class file manually. Even so, such modification is a delicate and time-consuming task to make to a class file while preserving the original code semantics.
- Erasing the birthmark without decreasing the execution performance is difficult. For example, the transformation mentioned above surely introduce an extra overhead.
- Even if the adversary succeeds in changing a single

birthmark, the similarity may not decrease much as long as the rest of birthmarks are still alive. Hence, manual modification must be done in the *entire code*, which is quite time-consuming.

Thus, we consider that the proposed birthmarks are simple but reasonably robust native signatures of Java class files.

5.3 Other Birthmarks

Apart from the proposed four birthmarks (CVFV, SMC, IS, and UC), we also examined whether or not other program characteristics could be used as birthmarks. The following are characteristics we decided not to use as birthmarks.

Constant Pool Java class files contain a data storage called *constant pool* in which constant values and strings are stored. These constants are necessary for program execution; however, we found that many of them are easily changed via obfuscation tools.

Control flow Control flow is a fundamental aspect of computer programs, which can be described as a directed graph. However, since some obfuscators change a control flow graph, they should not be used as birthmarks.

Data flow Data flow is also characterized in several ways, e.g. live variables in each line, variable span [42], and a sequence of substitutions of local variables. However, data flow is also easily changed by optimizers and obfuscators.

Thus far we have also considered using commonly-used software metrics, e.g. the number of methods in a class, the depth of conditional nesting, and Fan-In/Out. However, such single-value metrics do not satisfy the distinction property well because different programs can have the same value. Regardless a number of ways exist to characterize different aspects of software; therefore, we will seek for other potential birthmarks in our future work.

6. Conclusion

In this paper, we have proposed Java birthmarks to provide reasonable evidence of theft of class files. First, we formulated the birthmark of programs, and then presented four types of birthmarks: CVFV birthmark, SMC birthmark, IS birthmark, and UC birthmark.

The proposed birthmarks were thoroughly evaluated by three practical experiments. The evaluation was conducted from the viewpoints of preservation, distinction, and compiler-specific properties. The result showed that the proposed birthmarks have

- a good preservation property against automatic program transformation,
- a distinction property for most practical class files except tiny classes,
- sufficient native information of class files, which is not tightly coupled with compiler-specific issues.

We have also presented a qualitative discussion on feasibility in a pratical situation and in vulnerability for manual hacking.

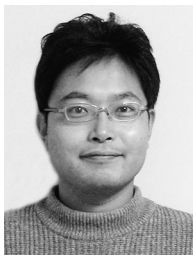
We plan to conduct a deeper security analysis of the proposed birthmarks through more experiments. For this, we will need a reasonable attack model that an adversary would take although quantifying the behaviors of the attackers is also challenging. We are also looking for more *real* case studies, in which we detect the fact of the theft from the same(or similar)-purpose software packages. An investigation of other types of birthmarks is also an interesting problem for future research.

References

[1] BSA, "Global software piracy study," June 2004. http://www.bsa.org/globalstudy/
[2] A. Patrizio, "Pirates experience Office XP (wired news)," March 2001. http://www.wired.com/news/business/0,1367,42402,00.html
[3] E. Raymond and R. Landley, "OSI position paper on the SCO vs. IBM complaint," May 2004. http://www.opensource.org/sco-vs-ibm.html
[4] "Epson pulls linux software following GPL violations (slashdot.org)," Sept. 2002. http://slashdot.org/article.pl?sid=02/09/11/2225212
[5] T. Ueno, "The protest page to pocketmascot," Sept. 2001. http://members.jcom.home.ne.jp/tomohiro-ueno/About_PocketMascot/About_PocketMascot_e.html
[6] B. Joy, G. Steele, J. Gosling, and G. Bracha, The Java Language Specification Second Edition, Addison-Wesley, June 2000.
[7] "Java 2 SDK standard edition," http://java.sun.com/se/
[8] T. Lindholm and F. Yellin, The Java^{TM} Virtual Machine Specification Second Edition, Addison-Wesley, April 1999.
[9] P. Kouznetsov, "jad - the fast java decompiler," Feb. 2004. http://kpdus.tripod.com/jad.html
[10] "Program transformation," http://www.program-transformation.org/
[11] H. Tamada, "jbirth: A tool for extracting birthmarks from java class files," 2003. http://se.aist-nara.ac.jp/jbirth/
[12] "Codeshield java byte code obfuscator," 1999. http://www.codingart.com/codeshield.html
[13] H. Ohuchi, "jarg - java archiver grinder," Jan. 2003. http://jarg.sourceforge.net/index.en
[14] "Smokescreen java obfuscator," 2000. http://www.leesw.com/
[15] "Zelix klass master," 1997. http://www.zelix.com/klassmaster/index.html
[16] "Apache Ant," http://ant.apache.org/
[17] M. Dahm, J. van Zyl, and E. Haase, "Jakarta BCEL," http://jakarta.apache.org/bcel/
[18] E. Gamma, E. Meade, and K. Beck, "JUnit," Feb. 2004. http://www.junit.org/
[19] "jikes," http://www-124.ibm.com/developerworks/oss/jikes/
[20] "Kjc kopi java compiler," http://www.dms.at/kopi/
[21] H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto, "Design and evaluation of birthmarks for detecting theft of java programs," Proc. IASTED International Conference on Software Engineering (IASTED SE 2004), pp.569–575, Innsbruck, Austria, Feb. 2004.
[22] R.L. Davidson and N. Myhrvold, "Method and system for generating and auditing a signature for a computer program," US Patent 5,559,884, Sept. 1996. Filed: June 30, 1994.
[23] A. Monden, "jmark: A lightweight tool for watermarking java class files," 2002. http://se.aist-nara.ac.jp/jmark/
[24] A. Monden, H. Iida, K. Matsumoto, K. Inoue, and K. Torii, "A practical method for watermarking java programs," Proc. COMPSAC 2000, 24th Computer Software and Applications Conference, pp.191–197, 2000

[25] C. Collberg, "Sandmark: A tool for the study of software protection algorithms," 2000. http://www.cs.arizona.edu/sandmark/
[26] C. Collberg and C. Thomborson, "Software watermarking: Models and dynamic embeddings," Proc. Principles of Programming Languages 1999, POPL'99, pp.311–324, San Antonio, TX, Jan. 1999.
[27] C. Thomborson, J. Nagra, R. Somaraju, and C. He, "Tamper-proofing software watermarks," Proc. second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation, pp.27–36, Australian Computer Society, Dunedin, New Zealand, 2004.
[28] A. Aiken, "MOSS: A system for detecting software plagiarism," June 2004. http://www.cs.berkeley.edu/~aiken/moss.html
[29] X. Chen, B. Francia, M. Li, B. Mckinnon, and A. Seker, "SID plagiarism detection," Dec. 2003. http://genome.math.uwaterloo.ca/SID/
[30] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," J. Universal Computer Science, vol.8, no.11, pp.1016–1038, Nov. 2002.
[31] M.J. Wise, "YAP3: Improved detection of similarities in computer program and other texts," Proc. 27 SIGCSE technical symposium on Computer science education, pp.130–134, Philadelphia, Pennsylvania, United States, 1996.
[32] A. Parker and H.O. James, "Computer algorithms for plagiarism detection," IEEE Trans. Educ., vol.32, no.2, pp.94–99, May 1989.
[33] K.J. Ottenstein, "An algorithmic approach to the detection and prevention of plagiarism," SIGCSE Bulletin, vol.8, no.4, pp.30–41, 1976.
[34] I.D. Baxter, A. Yahin, L.M.D. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," ICSM: The International Conference on Software Maintenance, pp.368–377, 1998.
[35] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," IEEE Trans. Softw. Eng., vol.28, no.7, pp.654–670, 2002.
[36] I. Krsul and E.H. Spafford, "Authorship analysis: Identifying the author of a program," Comput. Secur., vol.16, no.3, pp.233–257, 1997.
[37] E.H. Spafford and S.A. Weeber, "Software forensics: Can we track code to its authors?," Comput. Secur., vol.12, no.6, pp.585–595, 1993.
[38] D. Grover, ed., The protection of computer software – its technology and applications Second edition, The British Computer Society Monographs in Informatics Cambridge University Press, May 1992.
[39] "Jakarta project," http://jakarta.apache.org/
[40] "Java decompiler tests," http://www.program-transformation.org/Transform/JavaDecompilerTests
[41] K. Fukushima, T. Tabata, and K. Sakurai, "Program birthmark scheme with tolerance to equivalent conversion of java classfiles," IPSJ SIG Notes 2003-126, pp.81–86, Dec. 2003.
[42] S. Conte, H.E. Dunsmore, and V.Y. Shen, Software Engineering Metrics and Models (Benjamin/Cummings series in software engineering), Addison-Wesley, March 1986.

Haruaki Tamada received the BE and ME in Information and Communication Engineering from Kyoto Sangyo University, Japan in 1999, 2001. He is currently a Ph.D candidate in Graduate School of Information Science, Nara Institute of Science and Technology, Japan. He is interested in software security including; software obfuscation, watermarking, fingerprinting, and plagiarism detection. He is a student member of the IEEE.



Masahide Nakamura received the B.E., M.E., and Ph.D. degrees in Information and Computer Sciences from Osaka University, Japan, in 1994, 1996, 1999, respectively. From 1999 to 2000, he has been a post-doctoral fellow in SITE at University of Ottawa, Canada. He joined Cybermedia Center at Osaka University from 2000 to 2002. He is currently an assistant professor in the Graduate School of Information Science at Nara Institute of Science and Technology, Japan. His research interests include the feature interaction problem in network services, software validation and verification, and software metrics and security. He is a member of the IEEE.

Akito Monden received the BE degree (1994) in electrical engineering from Nagoya University, Japan, and the ME degree (1996) and DE degree (1998) in information science from Nara Institute of Science and Technology, Japan. He was honorary research fellow at the University of Auckland, New Zealand, from June 2003 to March 2004. He is currently Associate Professor at Nara Institute of Science and Technology. His research interests include software security, software measurement, and human-computer interaction. He is a member of the IEEE, ACM, IPSJ, JSSST, and JSiSE.

Ken-ichi Matsumoto received the BE, ME, and PhD degrees in Information and Computer sciences from Osaka University, Japan, in 1985, 1987, 1990, respectively. Dr. Matsumoto is currently a professor in the Graduate School of Information Science at Nara Institute of Science and Technology, Japan. His research interests include software measurement and software user process. He is a senior member of the IEEE, and a member of the ACM and IPSJ.