Design and Evaluation of Birthmarks for Detecting Theft of Java Programs

Haruaki Tamada   Masahide Nakamura   Akito Monden
Ken-ichi Matsumoto
Graduate School of Information Science,
Nara Institute of Science and Technology,
8916-5 Takayama-cho, Ikoma-shi, Nara, 630–0101 Japan,
email: {harua-t, masa-n, akito-m, matumoto}@is.aist-nara.ac.jp

¡¡¡¡¡¡¡ 200402.tex **ABSTRACT**
To detect theft of Java class files efficiently, we have so far proposed a concept of *Java birthmarks*. Since the birthmarks are unique and native characteristics of every class file, a class file with the same birthmark of another can be easily suspected as a *copy*. However, performance and tolerance of the birthmarks against sophisticated attacks had not been evaluated well. To clarify these issues, this paper conducts two experiments. In the first experiment, we demonstrate that the proposed birthmarks successfully distinguish non-copied files in practical Java application (97.8005%). The second experiment shows that the proposed birthmarks are quite tolerant of attacks with automatic program optimizers/obfuscators (93.3876%). ======= **ABSTRACT**
To detect theft of Java class files efficiently, we have so far proposed a concept of *Java birthmarks*. Since the birthmarks are unique and native characteristics of every class file, a class file with the same birthmark of another can be easily suspected as a *copy*. However, performance and tolerance of the birthmarks against sophisticated attacks had not been evaluated well. To clarify these issues, this paper conducts two experiments. In the first experiment, we demonstrate that the proposed birthmarks successfully distinguish non-copied files in practical Java application (97.8005%). The second experiment shows that the proposed birthmarks are quite tolerant of attacks with automatic program optimizers/obfuscators (93.3876%). ¿¿¿¿¿¿¿ 1.7

KEY WORDS
copyright issues, birthmark, software theft, Java class file

1 Introduction

In today's highly competitive world of computer software, *software theft* is a serious issue that often arises. 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 where a software product "Pocket Mascot" was created based on source code of another product "Minute Mascot", without permission of the author [11].

Software theft can cause severe damage to the software industries. However, since an enormous amount of software have been distributed all over the world, it is quite difficult to *detect* the fact of theft. Moreover, if a part of code was stolen, built into other software product, and distributed without source code, then the detection of the theft generally becomes much more difficult, This requires significant amount of skills and costs.

The goal of our research is to develop an easy-to-use method, which supports the efficient detection of *Java class files* that are quite similar to (or exactly the same as) each other. A Java class file is a small execution unit of a Java program, and a Java program generally consists of many class files. Although a class file is in the binary form (called *bytecode*), it is not very difficult to hack a class file, because of rigorous specification of Java VM, and powerful decompilers (e.g. jad [4]). In this sense, theft of class files is relatively easy to perform, but difficult to detect.

To achieve our goal, we have previously proposed a concept of *Java birthmarks* [10]. Intuitively, a birthmark of a Java class file is a set of unique characteristics that the class file originally possesses. 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 identify doubtful class files. Ideally, the birthmark should tolerate a certain extent of alternation and modification by *software crackers*. Therefore, the birthmark must be characteristics in the code that cannot easily modified. Taking this into account, we have proposed four kinds of birthmarks: constant values in field variables, a sequence of method calls, an inheritance structure and used classes.

In our previous research, however, we did not sufficiently evaluate the birthmarks. Especially, two issues had not been covered yet; *performance with practical applications* and *tolerance against program transformation*.

In this paper, we therefore conducted two experiments to clarify the above issues. In the first experiment, we applied the birthmarks to well-known Java applications (Ant, BCEL, JUnit). These applications are supposed to

be built by open-source communities without committing theft. Hence, we observed how the proposed birthmarks *distinguished* the (non-copied) class files. As a result, the proposed birthmarks identified 97.8005% of all class files. It was also shown that the rest of them were either tiny classes or classes written by "cut and paste".

In the second experiment, we evaluate how the birthmarks can tolerates program transformation by exploiting practical Java optimizers and obfuscators (`ZKM`[13], `Smokescreen`[9], `CodeShield`[2] and `jarg`[7]). 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/after the transformation is as high as 93.3876% on the average.

2 Related Work

Watermarking is a well-known technique to insist on the ownership of the original software for theft. Therefore, it may be used for our objective. Watermarking is basically to embed stealthy information which identifies the program author (in a static [6] or dynamic [3] manner). However, the watermarking is not always feasible, due to the nature of *extra code*. We cannot give proofs for modules into which no watermark is embedded. Strictly speaking, to completely prove software theft, we need to embed the watermarks into *all* the related modules beforehand. This is generally quite difficult when the number of modules is large, or the constraint of program size is strict.

There is also a technique, called *code clone* that could be used for the copy detection of programs (e.g., [1, 5]). The theft is doubted when the code clone is found in different software products. Also, automatic tools for measuring *software similarity* were presented, and use these tools for *plagiarism detection* [8, 12]. However, these code clone and plagiarism detection techniques require the *source code* of target programs. However, the source code is not necessarily available in our problem setting, since software products are often distributed without the source code. In addition, these techniques do not consider program transformation. Hence, those techniques are not complete for detecting software theft.

3 Java Birthmarks

3.1 Definition

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). Then, the relation $\equiv_{cp}$ is called the *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) $p \equiv_{cp} q \Rightarrow q \equiv_{cp} p$, (Transitive) $p \equiv_{cp} q \land q \equiv_{cp} r \Rightarrow p \equiv_{cp} r$.*

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 a $birthmark$ 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. 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 satisfy the following 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. 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*. Obfuscation makes original program harder to read and protects from understanding program. 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 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 birthmarks should be extracted. It is rare that the detail of two programs is completely the same for large programs. However, in the 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 $\equiv_{cp}$.

3.2 Proposed Birthmarks

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 $f(q)$. 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, we have proposed four methods that extract the following four types of birthmarks [10]: **constant values in field variables** (CVFV), **sequence of method calls** (SMC), **inheritance structure** (IS) and **used classes** (UC).

In the following, we present the definition of each birthmark. For more 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 often has *field variables* to store static and/or dynamic attributes. 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 $CVFVbirthmark$ 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)

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 appeared in p in this order (this is not necessarily the execution order), where m_i belongs to a class in C. Then, the sequence $(m_1, m_2, ..., m_n)$ is called $SMCbirthmark$ of p, denoted by $SMC(p)$.

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 structure* except `java.lang.Object`, which is a 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 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 $ISbirthmark$ of p, denoted by $IS(p)$.

The IS birthmark of the program in Fig 1 is:

```
org.apache.tools.ant.Task,
```

Table 1. The result of Experiment 1

		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%

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

3.2.4 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 and argument types of methods, 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 $UCbirthmark$ of p, denoted by $UC(p)$.

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 $f(p) = (p_1, ..., p_n)$ 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, we here introduce *similarity* of birthmark, which is a percentage of elements matched among $f(p)$ and $f(q)$ in the total elements in the birthmark (sequence).

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

4 Experimental Evaluation

To show the effectiveness in the practical settings, this section conducts two experiments. The first experiment evaluates performance of the proposed birthmarks, while the second experiment measures tolerance of the birthmarks against program transformation.

For the experiment, we have implemented a tool called `jbirth`. The main features of `jbirth`are: extraction of the four types of birthmarks directly from Java class files (without source code), pairwise birthmark comparison of Java class files, and plug-in architecture for new birthmarks.

4.1 Experiment 1(Performance)

In this experiment, we validate if the proposed birthmarks can be used as effective birthmarks for practical applications. Usually, all class files in a practical Java product are supposed to be different from each other. If there exist exactly the same class files in one package, it means redundant, thus, inefficient class design. Hence, we evaluate how many class files in a Java package can be *distinguished* from each other by the proposed birthmarks.

Now, let f be a certain birthmarks, and let p, q $(p \not\equiv_{cp} q)$ be class files arbitrarily taken from a product. To evaluate the performance of f, we show how many pairs of p and q are successfully distinguished by f.

As the target applications, we chose the following products: Apache Ant (1.5.4), Jakarta BCEL (5.1), JUnit (3.8.1) and `jbirth`. For each Jar file, we execute `jbirth`to perform pairwise birthmark comparison of class files contained in the Jar file. We used the proposed four birthmarks together. For this, we set the well-known classes (see Definition 4) to be class files contained in contained package of J2SDK SE 1.4.

The result is shown in Table 1. In the table, the *distinction ratio* represents a percentage of pairs of class files

Table 2. The result of Experiment 2

		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%

that are successfully distinguished, in the total pairs compared. The table also includes average, minimum, maximum values of the similarity. As seen in the distinction ratio, the proposed birthmarks were able to distinguish most of class files.

Figure 2 shows the frequency distribution of similarity, 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 number of comparisons. It can be seen in the figure that for most pairs of class files, the similarity is below 20%. This implies that different class files have significantly different birthmarks.

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."

4.2 Experiment 2 (Tolerance against transformation)

In this experiment, we evaluate the tolerance of the proposed birthmarks against program transformation such as obfuscation and optimization. To copy an original class file p, crackers may convert p into an equivalent p' by using certain automatic tools, so that the original birthmark $f(p)$ is altered. Our objective here is to evaluate how much of the original birthmarks are modified by a program transformation using similarity of birthmarks.

For this, we exploited the following practical tools: `ZKM`, `Smokescreen`, `CodeShield`and `jarg`.

Those tools typically implement *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 decompiled source code harder to understand. `ZKM`, `Smokescreen`and `CodeShield`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`provides unique features, *string encryption*, which encrypts string literals in class files, and then add code fragments to decrypt the string at runtime.

We applied each tool to a package `ant.jar` with the strongest obfuscation level, and obtained the obfuscated packages. Then, we executed `jbirth`to measure similarity of birthmarks for all pairs of a class file in `ant.jar` and its obfuscated version.

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

It can be seen in Table 2 that for all the tools, the majority of the original birthmarks were still preserved even after the obfuscation. Thus, the proposed birthmark achieved a relatively strong tolerance against program obfuscation in this experiment.

Note that the frequency distribution in Fig. 3 is significantly different from the one in Figure 2. That is, the similarity between independent (non-copied) class files is lower than the one between automatically converted files. This means that by setting an *appropriate threshold* on the similarity, the proposed birthmarks can provide considerably reliable evidence for the copied class files, even if the copies are obtained by program obfuscation.

We can see, in Figure 3, that the similarity varies slightly, depending on the obfuscation tool applied. It seems that the difference is caused by the obfuscation methods exploited in the tools. More thoughtful examination of the *impact* of specific obfuscation techniques against the proposed birthmarks is left to our future work.

5 Conclusion

In this paper, we presented four types of birthmarks to provide a reasonable evidence of theft of Java class files. The proposed Java birthmarks were thoroughly evaluated by two practical experiments. 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 obfuscation.

Compared to watermarking, the advantage is that the birthmarks are easily used without any extra code. Limitation is that: birthmarks might be a bit weaker evidence than watermarks. Even if we have the same birthmarks $f(p) = f(q)$, we can only suspect that q is *very likely to* be a copy of p. However, watermarking and birthmarking are not exclusive techniques. Hence, 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 many more obfuscation methods. 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] Ira D. Baxter, Andrew Yahin, Leonardo M. De Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM: the International Conference on Software Maintenance*, pages 368–377, 1998.

[2] Codeshield java byte code obfuscator, 1999. http://www.codingart.com/codeshield.html.

[3] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999, POPL'99*, San Antonio, TX, January 1999.

[4] jad - the fast java decompiler. http://kpdus.tripod.com/jad.html.

[5] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Trans. on Software Engineering*, 28(7):654–670, 2002.

[6] Akito Monden, Hajimu Iida, Kenichi Matsumoto, Katsuro Inoue, and Koji Torii. A practical method for watermarking java programs. In *COMPSAC 2000, 24th Computer Software and Applications Conference*, pages 191–197, 2000.

[7] Hidetoshi Ohuchi. jarg - java archiver grinder. http://jarg.sourceforge.net/index.en.

[8] L. Prechelt, G. Malpohl, and M. Philippsen. JPlag: Finding plagiarisms among a set of programs. Technical Report 1, Fakultat fur Informatik, Universitat Karlsruhe, Germany, mar 2000.

[9] Smokescreen java obfuscator, 2000. http://www.leesw.com/.

[10] Haruaki Tamada, Masahide Nakamura, Akito Monden, and Kenichi Matsumoto. Detecting the theft of programs using birthmarks. Information Science Technical Report NAIST-IS-TR2003014 ISSN 0919-9527, Graduate School of Information Science, Nara Institute of Science and Technology, Nov 2003.

[11] Tomohiro Ueno. The protest page to pocketmascot, 2001. http://members.jcom.home.ne.jp/tomohiro-ueno/About_PocketMascot/About_PocketMascot_e.html.

[12] Michael J. Wise. YAP3: Improved detection of similarities in computer program and other texts. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 28, 1996.

[13] Zelix klass master, 1997. http://www.zelix.com/klassmaster/index.html.

```
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());
    }
}
```

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

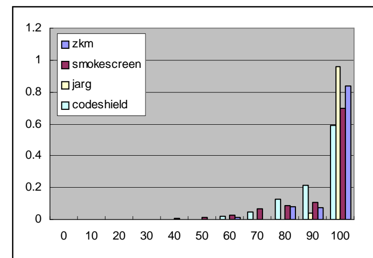

Figure 3. The result of Experiment 2

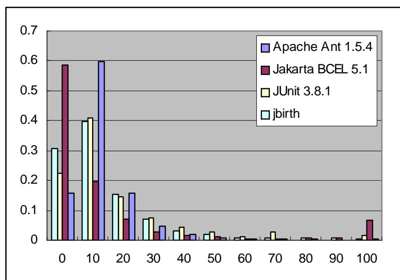

Figure 2. The result of Experiment 1