The Computer Journal Advance Access published November 5, 2009

© The Author 2009. Published by Oxford University Press on behalf of The British Computer Society. All rights reserved.
For Permissions, please email: journals.permissions@oxfordjournals.org
doi:10.1093/comjnl/bxp095

Detecting Common Modules in Java Packages Based on Static Object Trace Birthmark

HEEWAN PARK*, HYUN-IL LIM, SEOKWOO CHOI AND TAISOOK HAN

Department of Computer Science, KAIST, Gwahangno 335, Yuseong-gu, Daejeon 305-701, Republic of Korea
**Corresponding author: hwpark@pllab.kaist.ac.kr*

A software birthmark means inherent characteristics that can be used to identify a program. In this paper, we propose a birthmark technique based on object traces of Java programs. Java is an object-oriented programming language that provides various predefined class libraries that help programmers to produce software easily. In order to utilize Java class libraries, we have to use Java object instructions. The Java object instructions are hard to replace or remove, and so a set of sequences of object instructions is a proper candidate to represent inherent characteristics of a program. We propose a new birthmark using the sequences of object instructions. We evaluate the proposed birthmark with open source programs and compare it with previous static birthmarks. Experiments show that the detection capability of our birthmark is much higher than that of other static birthmarks despite obfuscations by Smokescreen and ZKM.

Keywords: software birthmark; software theft detection; GPL violation; Java bytecode analysis

Received 25 April 2009; revised 29 July 2009
Handling editor: Manfred Broy

1. INTRODUCTION

Recently, many software products have been developed as open source projects [1]. Such open source projects started from the concept of information sharing and have been growing with the support of many developers. Adopting an open source has the advantage of saving time and cost for developing software. For this reason, not only many programmers but also many software companies actively utilize open sources.

However, illegal use of open sources may violate the license because open sources are released under a corresponding license. The sorts of open source licenses that are registered in the Open Source Initiative number over 70 [2]. Among them, the most restricted license is the GNU General Public License (GPL) because most of source codes that are developed by utilizing open sources under the GPL have to be open to the public. The ratio of GPL projects is over 70% in the statistical data of SourceForge [3]. There were many GPL violations. For example, Skype violated the GPL in the VoIP phone [4], and CISCO also violated the GPL in the iPhone [5]. Besides these, many similar cases [6] exist.

Generally, it is hard to recover original source codes from distributed commercial software because software is distributed in a binary format. To analyze the software binary executables for reverse engineering, even an expert has to spend much time and effort. However, Java [7] applications are easy to analyze because Java applications are usually distributed as Java bytecodes that include much information to achieve high portability. Moreover, various decompilers [8–10] were developed owing to the strict specification of the Java virtual machine (JVM). Thus, in most cases we can recover source codes from Java class files through decompilers even though the original source code is not available. As a result, Java applications are one of the most frequent targets of software theft. Hence, it is essential to devise techniques to detect code theft, especially for Java. Figure 1 shows a typical procedure of code theft.

Research on code theft has been conducted mostly on source code plagiarism detection techniques [11, 12]. If we can get the source code of a suspicious program, plagiarism detection may be the most effective technique. However, we cannot always get the source code because an application program is generally distributed in compiled binary format. It is possible to recover a source code by decompiling the executable binaries. However, we can easily imagine that if someone steals a source code,

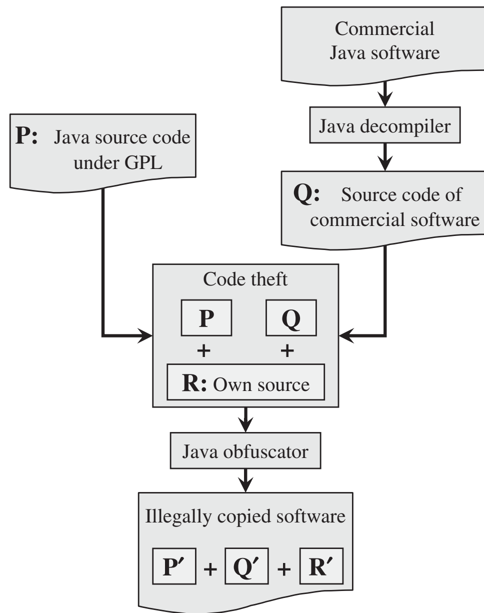

FIGURE 1. Procedure of code theft for Java.

he will distribute binary files transformed by an obfuscator to hide the theft. In this case, complete decompiling is impossible. After all, when we cannot get a source code, we have to use copy detection techniques that are applicable to the Java binary files directly.

There is another research, code clone detection [13, 14], for software theft detection. Code clones are duplicated lines of codes. If two different programs have code clones, we can suspect those programs as copied programs. However, such code clone techniques need source codes of two compared programs, like code plagiarism detection. Moreover, they are vulnerable to code transformation attacks.

To identify copied software, software watermarking [15, 16] is one of the most credible methods that proves copyright. A software watermark is similar to a multimedia watermark: it embeds information about the copyright of the software as a fingerprint. However, it is not applicable to software that has already been distributed without embedded watermarks. The developers must insert watermarks before releasing the software. In addition, watermarks can deteriorate the efficiency of programs and increase the size of programs.

When a program has already been distributed without a watermark, a birthmark can be considered as an alternative method. The software birthmark was introduced by Grover [17]. He defined a software birthmark as inherent characteristics that can be used to recognize or identify a program. If the same or similar birthmarks are extracted from two programs, we consider that the two programs may be in copy relation. In order to be effective, a birthmark has to satisfy two properties: credibility and resilience. Credibility means that the birthmark has no false positives; even though two programs have similar functions developed independently, birthmarks are able to decide that the programs are not in copy relation. Resilience means that the birthmark must be sustained even though a program is transformed by semantics-preserving transformations such as obfuscation [45] and optimization. A birthmark has to be resilient since code thefts may apply transformations to pretend the copied codes look different.

In this paper, we propose a new birthmark based on object traces of Java programs. An object trace means the sequence of object instructions; an object instruction means the instruction that creates an object, accesses field variables or invokes member functions. Java is an object-oriented programming language, and it provides various predefined Java class libraries that can help programmers to develop software easily. When a program utilizes Java class libraries, the compiled code must contain Java object instructions. Because Java object instructions are hard to replace or remove, an object trace birthmark (OTB), which is the sequence of object instructions, is a good candidate to represent inherent characteristics of the program. We extract an OTB by analyzing Java bytecodes statically and generating a control flow graph (CFG) of each method; we extract all possible sequences of object instructions from the entry node of a CFG to exit nodes. Then we calculate similarities among the sequences using a sequence alignment algorithm.

In order to evaluate the OTB, we perform experiments on open source programs that have common modules. We evaluate the capability of birthmarks to detect common modules among benchmark programs with respect to precision and recall. We compare the proposed birthmark with two existing birthmarks: the birthmark of Tamada *et al.* [18, 19] and the *k*-gram birthmark [20, 21]. From the experimental results, we confirm that the proposed birthmark is more resilient than existing static birthmarks, especially when the benchmark programs are transformed aggressively by various obfuscators.

In this research, we apply a new approach to designing a static birthmark that represents inherited characteristics of a program by observing the usual usage of Java libraries. Our contributions are summarized as follows.

(i) We proposed and implemented a novel static birthmark based on Java object traces. Java is an object-oriented programming language. Regarding objects, a Java program is executed by creating objects, accessing object variables and invoking object methods. Because these object-related instructions are considered the core properties of Java programs, we selected object instructions as our birthmark.

(ii) To evaluate the birthmark, we applied the well-known measures of precision and recall that are frequently used in information retrieval. Precision and recall imply the credibility and the resilience of the birthmark, respectively. Precision indicates the ratio of pairs that are

correctly detected as common modules out of all pairs that are indicated by the birthmark. Recall indicates the ratio of pairs that are correctly detected as common modules out of all the actual common module pairs.

(iii) We evaluated the proposed birthmark in real-world Java applications. To measure the common module detecting capability of the birthmark, we chose open source Java application pairs that share common modules. Thus, we are convinced that our birthmark can be used as a practical tool for detecting software theft or license violation.

(iv) We compared the proposed birthmark with existing static birthmarks. From the experimental results, we confirmed that the proposed birthmark is more resilient than existing static birthmarks, especially when the benchmark programs are transformed aggressively by various obfuscators.

The remainder of this paper is organized as follows. In Section 2, we review existing approaches of software birthmarks. In Section 3, we describe the definition of a software birthmark and propose the OTB. In Section 4, we describe the implementation of our birthmark. In Section 5, we evaluate the proposed birthmark. In Section 6, we discuss various birthmark issues and in Section 7 we conclude with a summary and suggestions for future work.

2. RELATED WORK

Software birthmarks are classified into static birthmarks and dynamic birthmarks according to the method of extracting the birthmarks from programs. For static birthmarks, characteristics of programs are gathered by static analyses of programs as a whole. Consequently, a static birthmark represents the overall behavior of a program, but it is vulnerable to program transformations. For dynamic birthmarks, information is collected while programs are running by monitoring the dynamic behaviors of programs. Dynamic birthmarks are more reliable and precise than static birthmarks. However, since the behaviors of programs vary depending on inputs and environments, birthmarks may look different even with the same program. Moreover, only parts of the programs are covered with a certain input so that copying parts of the program may not be detected.

Tamada *et al.* [18, 19] suggested static software birthmarks for Java class files. Their birthmarks are composed of four characteristics: *constant values in field variables*, *sequence of method calls, inheritance structures* and *used classes*. Constant values in field variables and sequence of method calls are easily attacked by code obfuscators. Inheritance structures and used classes utilize class relations. Therefore, the two birthmarks are resilient against existing Java obfuscators. On the other hand, they are not able to show a distinction between small classes. Furthermore, the four birthmarks cannot distinguish two programs that have the same structure but use different algorithms.

Myles and coworker [20, 21] suggested static k-gram birthmarks. A k-gram birthmark of a program is a set of k-length sequences of op-codes of the program. The sequences are extracted from the bytecodes while sliding a k-length window from the beginning of the program to the end. The k-gram birthmarks are vulnerable to attacks using program transformations such as code obfuscation and optimization because the order of instruction sequences is easily altered by program transformations. For example, the statement '`if (x>1) A; else B;`' can be easily transformed into '`if (x<=1) B; else A;`'. By doing the simple transformation, the sequence is altered from '`(cond);A;B;`' to '`(!cond);B;A;`'. Because the k-gram birthmarks do not consider control flows and extract physically contiguous instructions, they are not resilient to program transformations.

Park *et al.* [22] suggested a static Java trace birthmark: this is composed of all the possible execution traces that begin from the entry point to the exit points of a method. This approach considers control flows so that the Java trace birthmark may reflect actual execution sequences, while the k-gram birthmarks do not coincide with the runtime sequences. Therefore, the Java trace birthmark is resilient to code transformations such as the reordering of basic blocks and the insertion of unreachable codes by obfuscators. However, it is applicable only to small programs because the number of execution traces increases exponentially as the number of branch instructions increases.

Park *et al.* [23] also suggested a Java API trace birthmark that summarizes traces of API calls. By excluding instructions that are not API calls, they can reduce the number of traces so that the Java API trace birthmark can handle larger programs. However, since the number of API traces still increases exponentially in proportion to branches, the size of programs is not scalable depending on their control structures.

Myles and Collberg [24] suggested a whole program path (WPP) birthmark. To extract a WPP birthmark, instructions are gathered during the execution of a program. A graph is constructed by analyzing the gathered instruction trace with the SEQUITUR algorithm [25]. The graphs of the two programs are compared to get a maximum common subgraph that is used to compute a similarity between the two programs. The weakness of this approach is that huge amounts of instructions are gathered during the runtime. Thus, the WPP birthmarks are applicable only to tiny programs.

Schuler *et al.* [26] suggested a dynamic Java API birthmark. It extracts sequences of API calls per each class by executing a program. This approach is more efficient than WPP birthmarks because the size of the collected API sequences is much smaller than that of collected instruction sequences. This approach is resilient against obfuscations because most Java API calls cannot be replaced or be removed. However, this approach has a limitation in that the birthmark of a method with no API calls cannot be distinguished.

Dynamic birthmarks are more resilient to program transformations than static birthmarks because the execution path of a program is mostly preserved. The disadvantages of dynamic birthmarks are as follows. First, the amount of information that is collected during the execution time is huge. Second, dynamic birthmarks can fluctuate because the collected information varies according to inputs and runtime environments. Third, dynamic birthmarks can reflect the executed parts of the whole program, while static birthmarks can analyze the whole program.

3. SOFTWARE BIRTHMARK BASED ON STATIC OBJECT TRACES

3.1. Definition of software birthmark

Tamada *et al.* [18, 19] and Myles and coworker [20, 21] formally defined a software birthmark. The following definition of a software birthmark is a restatement of the definition given by them.

Definition 3.1 (Software Birthmark). *Let p and q be programs. Let f be a method for extracting a set of characteristics from a program. Then $f(p)$ is called a birthmark of p if and only if:*

(i) *$f(p)$ is obtained only from p itself (without any extra information);*
(ii) *q is a copy of $p \Rightarrow f(p) = f(q)$.*

Condition (i) explains the difference between birthmarks and watermarks. A birthmark extracts characteristics only from the program itself, while a watermark extracts extra information that was previously embedded. Condition (ii) means that if p and q are in copy relation, the birthmark of p and the birthmark of q are the same. In this definition, the meaning of copy implies not only the exact reproduction but also the semantics-preserving transformation.

The following properties are restatements of those of Tamada and Myles. These properties suggest two evaluation criteria that a birthmark should meet.

Property 1 (Credibility). *Let p and q be independently written programs that accomplish the same task. Then we say f is a credible measure if $f(p) \neq f(q)$.*

Property 2 (Resilience). *Let p' be a program obtained from p by applying semantics-preserving transformation T. Then we say f is resilient to T if $f(p) = f(p')$.*

The credibility property is a criterion that excludes the possibility of false positives. In other words, although two programs have the same functionality, independently developed programs should have different birthmarks. The resilience property is a criterion that excludes the possibility of false negatives. That is, a software birthmark has to be strong enough to endure semantics-preserving transformations.

3.2. Definition of static object trace birthmark

There are 204 instructions for the JVM according to the Java Virtual Machine Specification [27]. These instructions are categorized as follows:

(i) load and store instructions (`iload, istore`, ...);
(ii) arithmetic instructions (`iadd, isub`, ...);
(iii) type conversion instructions (`i2l, f2i`, ...);
(iv) object creation and manipulation (`new, getfield, putfield`, ...);
(v) operand stack management instructions (`pop, dup`, ...);
(vi) control transfer instructions (`ifeq, iflt`, ...);
(vii) method invocation and return instructions (`invokevirtual, ireturn`, ...);
(viii) throwing exceptions instruction (`athrow`);
(ix) subroutine instructions (`jsr, jsr_w, ret`);
(x) synchronization instructions (`monitorenter, monitorexit`).

Java is an object-oriented programming language. Regarding objects, a Java program is executed by creating objects, accessing object variables and invoking object methods. These object-related instructions are considered the core properties of Java programs. We select object-related instructions as our birthmark because compilers or code obfuscators cannot replace object-related instructions easily.

Table 1 shows JVM instructions that operate on objects. We devise an OTB using 11 JVM instructions of Table 1. Object-related instructions will be extracted as a birthmark from the CFG of a method.

Before the formal definitions of the OTB, we show an intuitive example.

TABLE 1. Object-related JVM instructions.

Sort of object instruction	JVM instruction
Creation	`new`
Member variable access	`putfield, getfield` `putstatic, getstiatic`

Method invocation	`invokevirtual`
	`invokestatic`
	`invokeinterface`
	`invokespecial`

Type checking	`checkcast, instanceof`

Figure 2 shows an example of a Java source program that initializes a server for client–server communications. The constructor `MyServer` inputs a port number and initializes a server with the port number if the port number is greater than 0; otherwise, it initializes a server with the default port number. Figure 3 shows disassembled Java bytecodes that are compiled from the program of Fig. 2.

Figure 4a is a CFG that is constructed from the Java bytecodes of Fig. 3. The numbers in the nodes represent the addresses of the corresponding bytecodes. To build an object flow graph (OFG), instructions that are not object-related are eliminated. Figure 4b is the OFG from Fig. 4a.

```
import java.net.*;
import java.io.*;
public class MyServer {
    int Port = 9999;
    ServerSocket Sock;
    public MyServer(int port)
    {
        try {
            if ( port > 0 )
            {
                Port = port;
                Sock = new ServerSocket(port);
            }
            else
                Sock = new ServerSocket();
        } catch (IOException e) {}
    }
}
```

FIGURE 2. Sample Java source code.

```
public MyServer(int);
0:  aload_0
1:  invokespecial java/lang/Object.init()V
4:  aload_0
5:  sipush  9999
8:  putfield      Port:I
11: iload_1
12: ifle    35
15: aload_0
16: iload_1
17: putfield      Port:I
20: aload_0
21: new           java/net/ServerSocket
24: dup
25: iload_1
26: invokespecial java/net/ServerSocket.init(I)V
29: putfield      Sock:Ljava/net/ServerSocket
32: goto    46
35: aload_0
36: new           java/net/ServerSocket
39: dup
40: invokespecial java/net/ServerSocket.init()V
43: putfield      Sock:Ljava/net/ServerSocket
46: goto    50
49: astore_2
50: return
Exception table:
from   to  target type
 11    46    49   Class java/io/IOException
```

FIGURE 3. Sample Java bytecode.

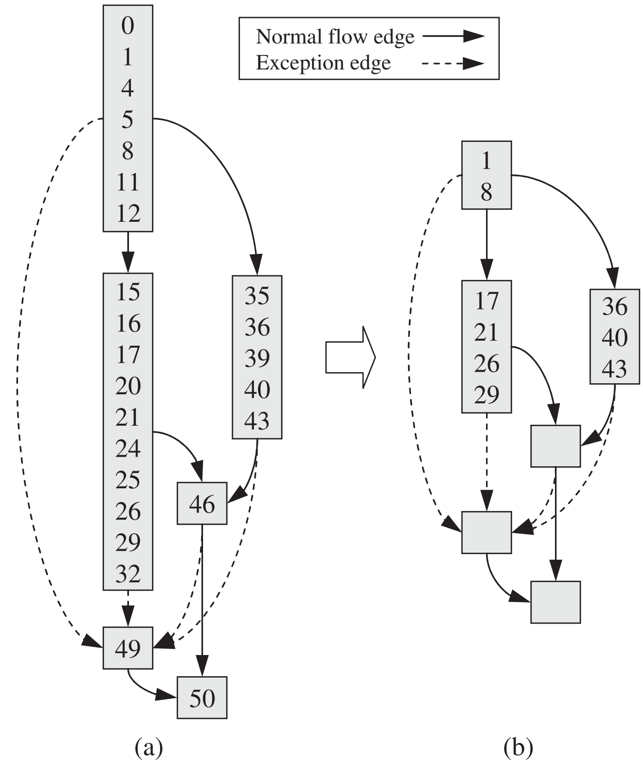

FIGURE 4. (a) Control flow graph and (b) object flow graph for `MyServer.class`.

Tables 2 and 3 are OTBs with $k = 2$ and 3, respectively. Duplicated object traces are removed.

In the case of a user-defined method call, we use only the type of the method; we exclude the method name because a programmer can rename it at any time. For example, a user-defined method call, '`m=max(x,y);`', is compiled to JVM instruction '`invokevirtual max(II)I`'. We replace it by '`invokevirtual (II)I`', because the name '`max`' can be altered easily by obfuscating transformations. For a user-defined class '`mypackage/myclass`', the JVM instruction '`new mypackage/myclass`' is replaced with '`new`', because the name of a user-defined class can be altered, too. In detail, we extract the operands of object trace instructions as a birthmark when they are instances of a primitive type or pre-defined classes in the Java standard packages. The 202 packages are predefined from '`java.applet`' to '`org.xml.sax.helpers`' in the Java Platform Standard Edition 6 API Specification [28].

Henceforth, we shall give formal definitions of the OTB.

DEFINITION 3.2 (CFG OF A JAVA METHOD). *The CFG of a Java method M is $G = (V, E)$, where V is a set of JVM instruction sequences that represent basic blocks in M and $E \subseteq V \times V$ is a set of edges that represent possible control flows between basic blocks in M.*

TABLE 2. Object traces for `MyServer.class` with $k = 2$.

Address	Object instruction
1	`invokespecial java/lang/Object.init()V`
8	`putfield I`
8	`putfield I`
17	`putfield I`
8, 17	`putfield I`
36, 21	`new java/net/ServerSocket`
21	`new java/net/ServerSocket`
26	`invokespecial java/net/ServerSocket.init(I)V`
26	`invokespecial java/net/ServerSocket.init(I)V`
29	`putfield java/net/ServerSocket`
36	`new java/net/ServerSocket`
40	`invokespecial java/net/ServerSocket.init()V`
40	`invokespecial java/net/ServerSocket.init()V`
43	`putfield java/net/ServerSocket`

TABLE 3. Object traces for `MyServer.class` with $k = 3$.

Address	Object instruction
1	`invokespecial java/lang/Object.init()V`
8	`putfield I`
17	`putfield I`
1	`invokespecial java/lang/Object.init()V`
8	`putfield I`
36	`new java/net/ServerSocket`
8	`putfield I`
17	`putfield I`
21	`new java/net/ServerSocket`
8	`putfield I`
36	`new java/net/ServerSocket`
40	`invokespecial java/net/ServerSocket.init()V`
17	`putfield I`
21	`new java/net/ServerSocket`
26	`invokespecial java/net/ServerSocket.init(I)V`
21	`new java/net/ServerSocket`
26	`invokespecial java/net/ServerSocket.init(I)V`
29	`putfield java/net/ServerSocket`
36	`new java/net/ServerSocket`
40	`invokespecial java/net/ServerSocket.init()V`
43	`putfield java/net/ServerSocket`

DEFINITION 3.3 (OFG of a Java Method). *Given a CFG $G = (V, E)$ of a Java method M, $G' = (V', E')$ is an OFG of M if G' satisfies the following two conditions:*

(i)

$$V' = \{v' | (v \in V) \wedge v = \langle jvm_1, jvm_2, \ldots, jvm_n \rangle \wedge v' = \langle obj_1, obj_2, \ldots, obj_k \rangle, \text{ where } v' \text{ is a sequence of object instructions extracted from } v \text{ considering the order of instruction in } v\}; \quad (1)$$

(ii)

$$E' = \{(v'_1, v'_2) | (v_1, v_2) \in E, \text{ where } v'_1 \text{ and } v'_2 \text{ are object instruction sequences extracted from } v_1 \text{ and } v_2, \text{ respectively}\}. \quad (2)$$

DEFINITION 3.4 (STATIC OBJECT TRACE WITH LENGTH k). *Given an OFG $G' = (V', E')$, a static object trace (SOT) with length k is defined as follows.*

(i) *If $k = 1$, then each object instruction of the OFG is a* (SOT) *with length* 1. *Thus, the set of SOTs with length 1, SOT^1, is as follows:*

$$SOT^1(G') = \{\langle obj_i \rangle | \forall v \in V' \wedge v = \langle obj_1, obj_2, \ldots, obj_n \rangle, 1 \leq i \leq n\}. \quad (3)$$

(ii) *If $k > 1$, then for a $\langle obj_1, obj_2, \ldots, obj_{k-1} \rangle \in SOT^{k-1}(G')$ and an $obj \in next_{G'}(obj_{k-1})$, the set of SOT^k is defined as follows:*

$$SOT^k(G') = \{\langle obj_1, obj_2, \ldots, obj_{k-1}, obj \rangle | \langle obj_1, obj_2, \ldots, obj_{k-1} \rangle \in SOT^{k-1}(G') \wedge obj \in next_{G'}(obj_{k-1})\}, \quad (4)$$

where

$$next_{G'}(obj_{k-1}) = \begin{cases} f(obj_{k-1}) & \text{if } obj_{k-1} \text{ is the last instruction of a basic block,} \\ g(obj_{k-1}) & \text{otherwise,} \end{cases}$$

where $f(obj_{k-1})$ is the set of the first instructions of the basic blocks that are directly reachable from the basic block containing the instruction obj_{k-1} and $g(obj_{k-1})$ is the set of instructions that directly follow obj_{k-1} in the same basic block.

DEFINITION 3.5 (OTB WITH LENGTH k). *For a Java program P, let $M_1, \ldots, M_n$ be the methods in P. Let G'_i be the OFG of the method M_i. The OTB with length k of the program P is defined as follows:*

$$OTB^k(P) = \{SOT^k(G'_i) | 1 \leq i \leq n\}. \quad (5)$$

3.3. Similarity of static OTB

To compare two object traces extracted from two methods, we apply sequence alignment algorithms [29–31] that are frequently used in bioinformatics to compare genomic sequences. Because sequence alignment algorithms are based on the policy of matching scores and mismatching penalties of sequences, they can compensate for the distortion of birthmarks due to partial modifications by obfuscators or attackers.

Well-known sequence alignment algorithms are global alignment [29], local alignment [30] and semi-global alignment algorithms [31]. In general, the semi-global alignment algorithm is used to compare short sequences with long sequences because it ignores the penalties caused by heading and tailing mismatches. The global alignment algorithm calculates similarities based on whole sequences, and the local alignment algorithm focuses on partial sequences. The length of traces in an OTB is always fixed at k, and hence it is reasonable to apply the global alignment or local alignment algorithm rather than the semi-global algorithm.

We evaluated local alignment and global alignment algorithms for our OTB and found that these two alignment algorithms have trade-offs. The global alignment algorithm is more credible than the local alignment algorithm. Since the global alignment algorithm focuses on the similarity of whole sequences, matching scores of two sequences are accumulated from the beginning to the end of the sequences even though the matching scores decrease below 0.

The local alignment algorithm is more resilient than the global alignment algorithm because the former focuses on the similarity of subsequence of traces. In the local alignment algorithm, matching scores do not decrease below 0 despite the mismatching penalties of the prefix; new subsequence matching begins from the matching score 0. Therefore, even though some parts of a trace are altered by obfuscators or attackers, the similarity between two traces is measured as much by the amount of the common subsequences.

When a malicious developer copies and includes another's modules illegally, he will try to hide the copied modules by applying various obfuscating transformations. To detect these kinds of attacks, birthmarks need to be resilient to obfuscation. As a result, we choose the local alignment algorithm to support resilience more than credibility for our OTB.

Given two sequences $T_1[1 \ldots i]$ and $T_2[1 \ldots j]$, the local alignment matrix $L[i, j]$ maximizing the alignment score of T_1 and T_2 can be summarized as follows.

(i) If $i = 0$ or $j = 0$, then

$$L[i, j] = 0. \quad (6)$$

(ii) If $i > 0$ and $j > 0$, then

$$L[i, j] = \max \begin{cases} L[i-1, j-1] + 1 & \text{if } T_1[i] = T_2[j], \\ L[i-1, j-1] - 1 & \text{if } T_1[i] \neq T_2[j], \\ L[i-1, j] - 1, \\ L[i, j-1] - 1, \\ 0. \end{cases} \quad (7)$$

After calculating the local alignment matrix L, the resulting score of the local alignment is obtained by finding the maximum value in the matrix, as follows:

$$LocalAlignmentScore(T_1, T_2) = \max_{i,j}(L[i, j]). \quad (8)$$

Definition 3.6 (Trace Similarity). *Given two object traces T_1 and T_2, the similarity between T_1 and T_2, $T_{\text{sim}}(T_1, T_2)$, is defined as*

$$T_{\text{sim}}(T_1, T_2) = \frac{LocalAlignmentScore(T_1, T_2)}{k}. \quad (9)$$

The object traces are k-length object instruction sequences. If T_1 and T_2 are exactly the same, the local alignment score is calculated as k. As a result, the trace similarity has a value in the range between 0 and 1.

Tables 4 and 5 show examples of object traces. In these examples, we assume that the length of the object traces is 4.

Table 6 shows the local alignment matrix for the object traces T_1 and T_2. The maximum value of the matrix is 3. As a result, the trace similarity between T_1 and T_2 is computed by Definition 3.6 as follows.

$$T_{\text{sim}}(T_1, T_2) = \frac{LocalAlignmentScore(T_1, T_2)}{4} = \frac{3}{4}.$$

TABLE 6. Local alignment matrix for the object traces T_1 and T_2.

			$T_2[1]$	$T_2[2]$	$T_2[3]$	$T_2[4]$
	0	0	0	0	0
$T_1[1]$	0	0	**1**	0	0
$T_1[2]$	0	0	0	**2**	1
$T_1[3]$	0	0	0	1	**3**
$T_1[4]$	0	0	0	0	2

Definition 3.7 (Similarity Matching Set). *Given two sets $S_1 = \{x_1, \ldots, x_m\}$ and $S_2 = \{y_1, \ldots, y_n\}$, let each pair of elements (x_i, y_j) have a similarity value. The similarity matching set between two sets is defined as the set of injective mappings between S_1 and S_2 that maximizes the total sum of the similarity values, and is denoted by $MatchingSet(S_1, S_2)$.*

To calculate a similarity between two sets, the similarity between every pair of elements in each set has to be considered. The similarity between two sets is obtained by maximizing the sum of similarities of matched elements contained in the two sets. The matching problem can be solved by an optimal algorithm, such as Hungarian [32].

When birthmarks of two different programs are compared, the optimal algorithm does excessive matching to maximize the overall matching value even though the two compared birthmarks are not similar pairs. This decreases the credibility that distinguishes the two different programs.

Therefore, we utilized a greedy algorithm instead of the optimal algorithm. The greedy algorithm can prevent the excessive matching through sequential matching from the high similarity pairs to low ones. Moreover, the greedy algorithm is faster and requires a lower amount of memory than the optimal algorithm.

TABLE 4. Example of object trace T_1 ($k = 4$).

	Object trace T_1
$T_1[1]$	`putfield I`
$T_1[2]$	`new java/net/ServerSocket`
$T_1[3]$	`invokespecial java/net/ServerSocket.init()V`
$T_1[4]$	`putfield java/net/ServerSocket`

TABLE 5. Example of object trace T_2 ($k = 4$).

	Object trace T_2
$T_2[1]$	`invokespecial java/lang/Object.init()V`
$T_2[2]$	`putfield I`
$T_2[3]$	`new java/net/ServerSocket`
$T_2[4]$	`invokespecial java/net/ServerSocket.init()V`

DEFINITION 3.8 (METHOD SIMILARITY). *Given sets of traces $TS_1[1 \ldots m]$ of a method M and $TS_2[1 \ldots n]$ of a method N, and the trace similarity matching set $MatchingSet(TS_1, TS_2)$ between two sets of traces, the method similarity between M and N, $M_{\text{sim}}(M, N)$, is defined as*

$$M_{\text{sim}}(M, N) = \frac{\sum_{(TS_1[i], TS_2[j]) \in MatchingSet(TS_1, TS_2)} T_{\text{sim}}(TS_1[i], TS_2[j])}{\min(m, n)}. \tag{10}$$

The similarity of the two Java methods is calculated by adding all the values of similarity of matched pairs. To normalize the similarity, the sum is divided by the minimum number of traces of methods; the resulting similarity ranges from 0 to 1 in proportion to the degree of similarity between the two methods.

For example, Table 7 shows the similarity matrix between TS_1 and TS_2. A greedy algorithm finds the trace similarity matching set $MatchingSet(TS_1, TS_2) = \{(TS_1[1], TS_2[2]), (TS_1[2], TS_2[4]), (TS_1[3], TS_2[1])\}$. Then, the similarity between the two methods M and N is obtained as follows:

$$M_{\text{sim}}(M, N) = \frac{1.00 + 0.92 + 0.67}{\min(3, 4)} = 0.86.$$

DEFINITION 3.9 (CLASS SIMILARITY). *Given sets of methods $M[1 \ldots c]$ in class C and $N[1 \ldots d]$ in class D, and the method similarity matching set $MatchingSet(M, N)$ between two sets of methods, the class similarity between C and D is defined as follows:*

$$C_{\text{sim}}(C, D) = \frac{\sum_{(M[i], N[j]) \in MatchingSet(M, N)} M_{\text{sim}}(M[i], N[j])}{\min(c, d)}. \tag{11}$$

To calculate a similarity between two classes, the similarity between every pair of methods in each class has to be considered. Since the matched method set represents the set of the most similar pairs among all the methods in each class, the similarity between the two classes is obtained by maximizing the sum of similarities of matched methods contained in two classes.

The main purpose of a birthmark is to detect a copy of a program. The class similarity of the proposed birthmark lies in the range from 0 to 1, and this value is used to determine whether the two classes are copied classes or not. The two classes C and D are classified according to the class similarity as follows:

$$C_{\text{sim}}(C, D) \begin{cases} \geq 1 - \varepsilon & C \text{ and } D \text{ are classified as copied classes,} \\ < 1 - \varepsilon & C \text{ and } D \text{ are classified as independent classes.} \end{cases} \tag{12}$$

The threshold value ε is a criterion to classify two classes as copied classes or independent classes. If ε becomes small, two classes are compared precisely. Therefore, the credibility of birthmark increases. Otherwise, if ε becomes large, the two classes are compared loosely. Thus, the resilience of birthmark increases. Therefore, it is necessary to choose a feasible ε value considering the credibility and resilience trade-off.

TABLE 7. Similarity matrix between TS_1 and TS_2.

	$TS_2[1]$	$TS_2[2]$	$TS_2[3]$	$TS_2[4]$
$TS_1[1]$	0.33	**1.00**	0.83	0.58
$TS_1[2]$	0.25	0.42	0.50	**0.92**
$TS_1[3]$	**0.67**	0.75	0.17	0.50

4. IMPLEMENTATION

Figure 5 shows a procedure for the proposed static object trace birthmarking system. The birthmarking system computes the class similarity between two Java classes. To calculate the similarity between two classes, we need to calculate the similarity of each method in each class. In order to calculate the similarity of each method, two OFGs are generated from the two methods. The nodes of OFGs contain Java object instructions. With OFGs, SOTs are generated. Finally, the SOT sets are compared to ascertain the similarity between the two methods using the local alignment algorithm. Because the matrix is computed using dynamic programming, the time complexity of the local alignment algorithm is $O(m \cdot n)$, where m and n are the sizes of the two object traces. After we have object trace similarities, we can get method similarities and class similarities by Definitions 3.8 and 3.9, respectively.

The proposed birthmark was implemented in C/C^{++} languages with a GNU g++ compiler in a Linux environment. We utilized a `javap` disassembler [33] as a front-end to get the bytecodes from each class file. The `javap` is a well-known Java disassembler published by Sun Microsystems. From the disassembled bytecodes, we generated the CFG, OFG and OTB in our birthmarking system. Our birthmark was evaluated on a Linux system with an Intel® Xeon™ CPU 3.2 GHz processor and 4 GB RAM.

5. EVALUATION

5.1. Evaluation measure for software birthmark

To evaluate the efficiency of birthmarks, evaluation measures need to be suggested. In previous works, the average values of credibility and resilience were used to measure the efficiency of birthmarks. Credibility means that two programs developed independently must have different birthmarks, even though they have the same functionality. Resilience means that software

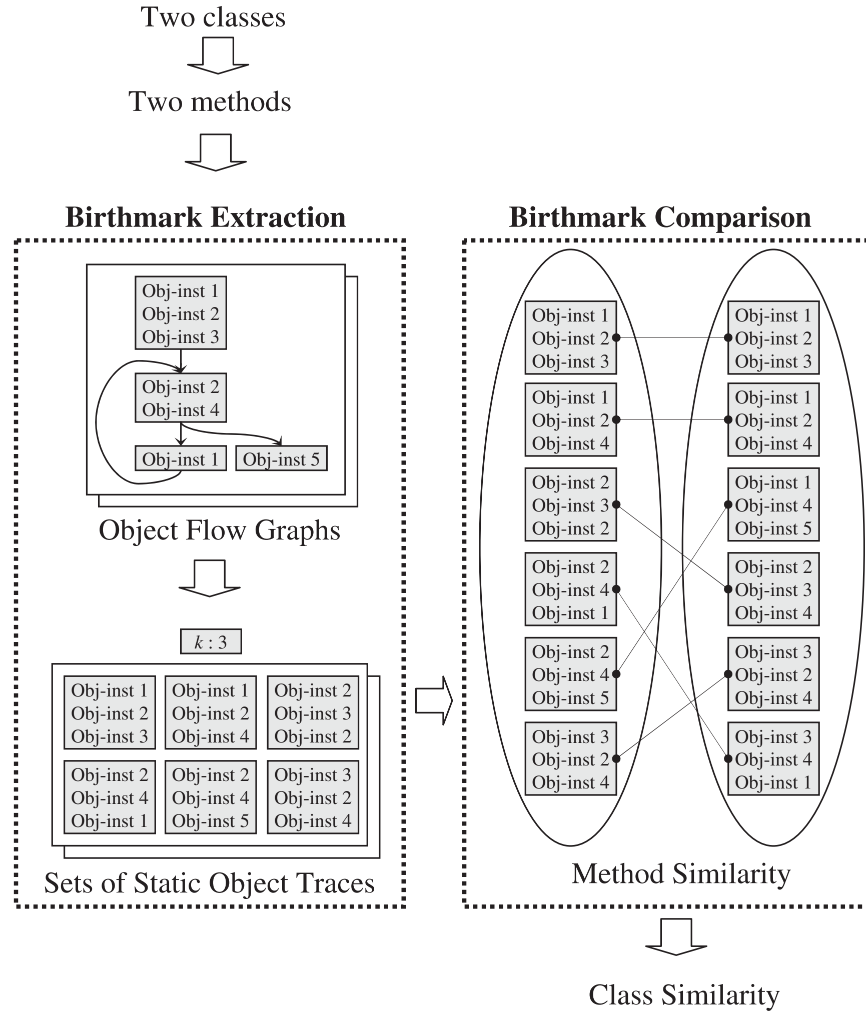

FIGURE 5. Procedure for static object trace birthmarking system.

birthmarks have to be strong enough to endure semantics-preserving transformations. For example, previous works calculate the average similarity between two different programs to evaluate the credibility, and then calculate the average similarity between an original program and the transformed program to evaluate the resilience.

However, the previous evaluation measures have a weak point in that they cannot detect common modules when the ratio of common modules to all modules is not high enough. If the portion of common modules between two programs is large, the similarity of two programs becomes high. However, if the portion of common modules is low, it causes low similarity of two programs, although the two programs obviously have some common modules. Therefore, in order to detect common modules with birthmarks, the similarities between pairs of modules are more valuable than the average similarities between two programs. Thus, a birthmark needs the capability to detect common modules even though two programs have one common module, and an evaluation measure for birthmarks reveals the capability of birthmarks to detect sharing modules.

In this paper, we have applied well-known measures for birthmark evaluation: precision and recall. Precision and recall [34, 35] are frequently used in information retrieval and statistical classification. Precision and recall imply the credibility and resilience of birthmarks, respectively. Specifically, precision indicates the ratio of pairs that are correctly detected as common modules out of all the pairs that are chosen as common modules. Recall is the ratio of pairs that are correctly detected as common modules out of all the common module pairs. The two measures are defined as follows:

$$Precision = \frac{TP}{TP + FP} = \frac{|\{\text{Common Module Pairs}\} \cap \{\text{Detected Pairs}\}|}{|\{\text{Detected Pairs}\}|}, \quad (13)$$

$$Recall = \frac{TP}{TP + FN} = \frac{|\{\text{Common Module Pairs}\} \cap \{\text{Detected Pairs}\}|}{|\{\text{Common Module Pairs}\}|}, \quad (14)$$

where TP is the number of true positives, FP is the number of false positives and FN is the number of false negatives.

To increase precision, birthmarks must reduce the number of false positives, which means that birthmarks can detect common modules more precisely. To increase recall, birthmarks must reduce the number of false negatives, which means that birthmarks can detect common modules more completely. Often there is a trade-off between precision and recall: where it is possible to increase one at the cost of reducing the other. For example, a birthmark can be designed to enhance the recall by detecting more modules as copied, which may result in the increase of false positives.

The harmonic mean of precision and recall, F-measure, is a well-known measure in the area of information retrieval. The F-measure is used to evaluate the precision and recall together and is defined as follows:

$$F\text{-}measure = 2 \times \frac{Precision \times Recall}{Precision + Recall}. \quad (15)$$

Using the F-measure for birthmark evaluation considers precision and recall at the same time because the F-measure is high only when the precision and recall are high together.

5.2. Evaluation setup

To be a good code theft detector, a birthmark must be based on characteristics that distinguish a program from the others with a unique signature. Our birthmark is defined as a set of SOTs with length k ($SOT^k s$) of a Java program. We investigated what value of k is suitable to identify programs with reasonable uniqueness and complexity. We examined all the $SOT^k s$ of class files included in the *Eclipse* 3.0 SDK package [36]. The package has a total of 19 502 class files in 144 jar files. We used the *Eclipse* 3.0 SDK since the package had a variety of jar files that represent many programming styles of Java programs.

Table 8 and Fig. 6 show the variations of the most frequent and top 10 frequent $SOT^k s$ according to the length k. As the k increases, the ratio tends to decrease. For lengths larger than 3, despite a sequence length increment, the ratios of the most frequent $SOT^k s$ were small enough to consider that the $SOT^k s$ were sufficiently different. Hence, we conducted our experiments with $k = 3$.

To evaluate the capability of birthmarks to detect common modules, we selected benchmark programs that share common classes. As shown in Table 9, five pairs of open source Java packages were used as benchmark programs. Each pair of packages has common classes. The pairs of (JavaCUP, VtdXML), (JIMI, Jitac) and (JLayer, tMP3) have the exactly same versions of

TABLE 8. Number of unique $SOT^k s$, total number of $SOT^k s$, ratio of the most frequent SOT^k, and ratio of top 10 frequent $SOT^k s$ for $1 \leq k \leq 6$.

k	Number of unique $SOT^k s$	Total number of $SOT^k s$	Ratio of the most frequent SOT^k (%)	Ratio of top 10 frequent $SOT^k s$ (%)
1	7 858	646 428	6.26	31.01
2	86 000	817 748	1.30	7.16
3	285 500	931 490	0.29	1.91
4	564 625	1 073 683	0.11	0.86
5	880 295	1 292 501	0.09	0.47
6	1 297 870	1 678 016	0.07	0.23

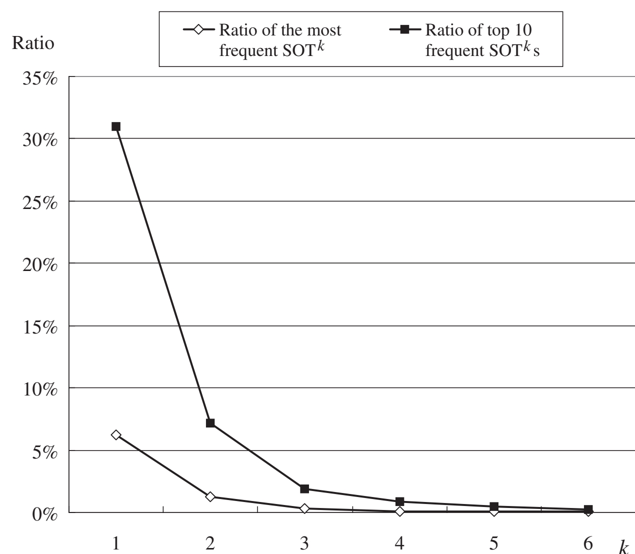

FIGURE 6. Variations of the most frequent SOT^k and top 10 frequent $SOT^k s$ for $1 \leq k \leq 6$.

common classes. However, the pairs of (Piccolo, Crimson) and (jEdit, Jext) have slightly different versions of common classes.

The purpose of the experiments is to measure the capability of detecting known common classes. However, we do not evaluate the whole classes of benchmark programs. We evaluate classes that have at least 20 object instructions, because small classes hardly represent inherent characteristics and may not be targets for theft.

To get the precision and recall values, we need the oracle who can tell the truth. In detail, the oracle has to judge whether the detections of a birthmark are true positives or false positives. Thus, we defined the pairs that had the exact same names as common classes in two compared packages. Of course, the classes that had different names but equal contents could be common classes. To be a good birthmark, it must detect all those classes. However, our evaluation has a weakness that those classes are classified as false positives when a birthmark detects the classes as common classes.

TABLE 9. Benchmark programs.

	Number of total classes	Size of total classes (bytes)	Number of evaluated classes	Size of evaluated classes (bytes)	Size ratio of evaluated classes/ total classes	Number of common classes
JavaCUP[a] 0.10k	41	143 016	19	121 639	0.85	18
VtdXML[b] 2.4	151	491 831	47	376 523	0.77	
JIMI[c] 1.0	324	758 830	137	541 125	0.71	137
Jitac[d] 0.2.0	333	800 222	144	580 374	0.73	
JLayer[e] 0.1.1	65	220 289	24	138 068	0.63	24
tMP3[f] 0.02	89	304 018	35	203 764	0.67	
Piccolo[g] 1.04	87	323 018	29	259 694	0.80	12
Crimson[h] 1.1.3	145	355 230	46	262 304	0.74	
jEdit[i] 4.2	805	2 729 899	355	2 202 417	0.81	19
Jext[j] 5.0	466	1 077 461	200	846 088	0.79	

[a] JavaCUP—LALR Parser Generator for Java, http://www.cs.princeton.edu/∼appel/modern/java/CUP/.
[b] VtdXML—Virtual Token Descriptor-based XML Processing Suite, http://vtd-xml.sourceforge.net/.
[c] JIMI—The Java Image Management Interface, http://java.sun.com/products/jimi/.
[d] Jitac—Image To ASCII Converter, http://www.roqe.org/jitac/.
[e] JLayer—MP3 library for Java Platform, http://www.javazoom.net/javalayer/javalayer.html.
[f] tMP3—Java MP3 Player, http://sourceforge.net/projects/tmp3/.
[g] Piccolo—XML Parser for Java, http://piccolo.sourceforge.net/.
[h] Crimson—Java XML Parser, http://xml.apache.org/crimson/.
[i] jEdit—Programmer's Text Editor, http://www.jedit.org/.
[j] Jext—Free Source Code Editor, http://www.jext.org/.

We compared our birthmark with the existing static Java birthmarks. For the Tamada birthmark and the k-gram birthmark, we used the Java birthmark toolkit, *Stigmata* 1.1 [37]. In the experiment for the k-gram birthmark, we set the value of k to 3, which was used by Myles [21] who suggested the k-gram birthmark.

We have to choose a reasonable threshold value ε for each birthmark. Table 10 compares the results of the three birthmarks with ε values from 0.1 to 0.4. From the table, as the value of ε increases, the average of recall also increases, but the average of precision decreases. In this paper, we choose the threshold value as 0.2 because the F-measure is maximized in all three birthmarks when $\varepsilon = 0.2$. According to this threshold, a similarity range of [0.8, 1.0] is classified as copied classes, and [0, 0.8) is classified as independent classes.

5.3. Common module detection

Table 11 shows the experimental results of common module detections with non-transformed programs. Among the experiments of five pairs, the recall values of pairs (Piccolo, Crimson) and (jEdit, Jext) were calculated relatively lower than the others because those pairs have different versions of common modules. In particular, the pair (jEdit, Jext) has some modules whose differences are big. For example, `TokenMarker.class` of Jext has 12 methods, but 10 methods are replaced with different ones in jEdit. Therefore, all of the birthmarks cannot detect it as a common module. As a result, a large number of false negatives in the pair (jEdit, Jext) is not due to the low detecting capability of birthmarks, but to the difference of common modules in that pair.

TABLE 10. Variations of precision, recall and F-measure of three birthmarks for $0.1 \leq \varepsilon \leq 0.4$.

	Tamada birthmark			k-gram birthmark ($k = 3$)			Object trace birthmark ($k = 3$)		
	Average of precision	Average of recall	Average of F-measure	Average of precision	Average of recall	Average of F-measure	Average of precision	Average of recall	Average of F-measure
$\varepsilon = 0.1$	0.96	0.87	0.90	0.94	0.84	0.88	0.93	0.88	0.90
$\varepsilon = \mathbf{0.2}$	0.92	0.91	**0.92**	0.90	0.91	**0.91**	0.88	0.94	**0.91**
$\varepsilon = 0.3$	0.87	0.92	0.89	0.89	0.93	0.90	0.79	0.95	0.86
$\varepsilon = 0.4$	0.63	0.95	0.75	0.81	0.95	0.87	0.70	0.95	0.80

TABLE 11. Common module detections with non-transformed programs.

	Tamada birthmark					k-gram birthmark ($k = 3$)					Object trace birthmark ($k = 3$)					
	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall	Time[a] (s)
JavaCUP, VtdXML	18	0	0	1.00	1.00	18	0	0	1.00	1.00	18	2	0	0.90	1.00	28
JIMI, Jitac	137	4	0	0.97	1.00	137	4	0	0.97	1.00	137	10	0	0.93	1.00	236
JLayer, tMP3	24	2	0	0.92	1.00	24	0	0	1.00	1.00	24	0	0	1.00	1.00	27
Piccolo, Crimson	11	3	1	0.79	0.92	11	2	1	0.85	0.92	12	3	0	0.80	1.00	35
jEdit, Jext	12	1	7	0.92	0.63	12	5	7	0.71	0.63	13	4	6	0.76	0.68	1,326
Average				0.92	0.91				0.90	0.91				0.88	0.94	

[a]Because the *Stigmata* has no function for checking runtime of each birthmark, we cannot record the exact runtime for the Tamada birthmark and the k-gram birthmark. According to the result of manual checking, the runtime of the two birthmarks did not exceed 10 s in all experiment cases.

The number of false positives for the pair (JIMI, Jitac) is slightly larger in our birthmark than in the others. We investigated the false positive cases and found that the detected classes have similar control structures and behaviors. For example, `BMPEncoder.class` of JIMI and `TGAEncoder.class` of Jitac are different classes but our birthmark classified them as common classes. The two classes have the same control flows and the same object instructions, except the names of user methods and slot numbers of local variables. Thus, our birthmark cannot classify them as different modules.

In this experiment, three birthmarks showed high precisions and recalls. When a vicious programmer copies another's module to develop his software, he will try to hide the fact of his theft by applying various transformation techniques. This requires a birthmark to be resilient to program transformations. In order to evaluate the resilience of birthmarks, we utilize a Java optimizer such as jarg [38], and obfuscators such as CodeShield [39], Smokescreen [40] and ZKM [41].

Table 12 shows the experimental results of the evaluation of the resilience to the jarg optimizer. One of each pair of benchmark programs was optimized with jarg. Jarg changes long names of identifiers into shorter ones, optimizes redundant parts of the class files and compresses local variable slots. The similarities of common modules of each pair are reduced by these optimizations. Thus, the numbers of false negatives of the Tamada birthmark and the k-gram birthmark increase, and this causes the decrement of the recall values. However, our OTB is not damaged by jarg because the object traces, which consist of object instructions, are not the target of optimization by jarg.

Table 13 shows the experimental results with the CodeShield obfuscator. CodeShield exercises name obfuscation like the jarg, obfuscates control flows and adds dummy exception tables to classes. Due to these obfuscations, the numbers of false negatives of the Tamada birthmark and the k-gram birthmark increase. However, our OTB is not affected by these attacks because object traces are resilient to obfuscating names and adding dummy exception tables. As a result, the recall values of the Tamada birthmark and the k-gram birthmark decrease slightly, compared to the results of Table 11.

Table 14 shows the experimental results of the Smokescreen obfuscator. Smokescreen has four major obfuscating

TABLE 12. Common module detections after optimization by jarg.

	Tamada birthmark					k-gram birthmark ($k = 3$)					Object trace birthmark ($k = 3$)					
	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall	Time (s)
JavaCUP, jarg(VtdXML)	17	0	1	1.00	0.94	18	0	0	1.00	1.00	18	2	0	0.90	1.00	28
JIMI, jarg(Jitac)	132	4	5	0.97	0.96	135	2	2	0.99	0.99	137	10	0	0.93	1.00	235
JLayer, jarg(tMP3)	24	0	0	1.00	1.00	23	0	1	1.00	0.96	24	0	0	1.00	1.00	26
Piccolo, jarg(Crimson)	10	3	2	0.77	0.83	9	0	3	1.00	0.75	12	3	0	0.80	1.00	34
jEdit, jarg(Jext)	8	1	11	0.89	0.42	12	5	7	0.71	0.63	13	4	6	0.76	0.68	1,320
Average				0.93	0.83				0.94	0.87				0.88	0.94	

TABLE 13. Common module detections after obfuscation by CodeShield.

	Tamada birthmark					k-gram birthmark ($k = 3$)					Object trace birthmark ($k = 3$)					
	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall	Time (s)
JavaCUP, CodeShield(VtdXML)	17	0	1	1.00	0.94	17	0	1	1.00	0.94	18	2	0	0.90	1.00	29
JIMI, CodeShield(Jitac)	136	4	1	0.97	0.99	134	2	3	0.99	0.98	137	10	0	0.93	1.00	235
JLayer, CodeShield(tMP3)	24	0	0	1.00	1.00	24	0	0	1.00	1.00	24	0	0	1.00	1.00	25
Piccolo, CodeShield(Crimson)	7	4	5	0.64	0.58	6	0	6	1.00	0.50	12	3	0	0.80	1.00	35
jEdit, CodeShield(Jext)	12	1	7	0.92	0.63	12	4	7	0.75	0.63	13	4	6	0.76	0.68	1,319
Average				0.91	0.83				0.95	0.81				0.88	0.94	

TABLE 14. Common module detections after obfuscation by Smokescreen.

	Tamada birthmark					k-gram birthmark ($k = 3$)					Object trace birthmark ($k = 3$)					
	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall	Time (s)
JavaCUP, Smokescreen(VtdXML)	1	0	17	1.00	0.06	1	0	17	1.00	0.06	18	0	0	1.00	1.00	30
JIMI, Smokescreen(Jitac)	18	1	119	0.95	0.13	26	0	111	1.00	0.19	135	10	2	0.93	0.99	243
JLayer, Smokescreen(tMP3)	1	0	23	1.00	0.04	2	0	22	1.00	0.08	24	0	0	1.00	1.00	29
Piccolo, Smokescreen(Crimson)	0	0	12	0.00	0.00	0	0	12	0.00	0.00	10	1	2	0.91	0.83	37
jEdit, Smokescreen(Jext)	1	0	18	1.00	0.05	1	0	18	1.00	0.05	12	3	7	0.80	0.63	1,400
Average				0.79	0.06				0.80	0.08				0.93	0.89	

techniques. First, all the strings in a class file are encrypted and stored in one field member of the class. They are decrypted during the runtime through reconstruction with substring API function calls. Second, control flows of a method are complicated by adding opaque predicates. An opaque predicate [42, 43] is either always true or always false, but its value is hard to guess statically. For example, '`if (x*x >= 0)`' is always true but is not easy to guess with a static analysis. Third, sequences of stack instructions are reordered. For example, '`iconst_0; istore_1; iconst_1; istore_2;`' can be transformed to '`iconst_0; iconst_1; istore_2; istore_1;`' by reordering. Fourth, a string decryption method is added and the order of existing methods is changed.

With these strong obfuscations of Smokescreen, the Tamada birthmark and the k-gram birthmark are severely damaged and the numbers of false negatives increase seriously. In the Tamada birthmark, the number of false negatives increases largely due to the damage of the sequence of method calls caused by the addition of string decryption routines and reordering of the sequence of methods. As a result, the recall values are largely dropped.

The k-gram birthmark shows a similar result to the Tamada birthmark. The k-gram birthmark is damaged because the instructions related to opaque predicates and function calls for string reconstruction are inserted into the original bytecodes. Also, the reordering of stack instructions affects the k-gram birthmark negatively.

Our OTB is degraded so little that the number of false negatives increases slightly in the cases of (JIMI, Smokescreen(Jitac)), (Piccolo, Smokescreen(Crimson)) and (jEdit, Smokescreen(Jext)). This degradation is caused by the string encryption attack of Smokescreen. The local alignment algorithm helps to reduce the effect of added object instructions of string encryption routines by Smokescreen. Since the local alignment algorithm gives partial scores between the original and modified traces, it gives proper similarity despite the insertions of object instructions by Smokescreen. In the cases of (JavaCUP, Smokescreen(VtdXML)), (Piccolo, Smokescreen(Crimson)) and (jEdit, Smokescreen(Jext)), the values of the average precision increase a little unexpectedly even after obfuscation. The reason for this effect is that the number of false positives decreases through obfuscation.

TABLE 15. Common module detections after obfuscation by ZKM.

	Tamada birthmark					k-gram birthmark ($k = 3$)					Object trace birthmark ($k = 3$)					
	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall	Time (s)
JavaCUP, ZKM(VtdXML)	17	0	1	1.00	0.94	11	0	7	1.00	0.61	18	0	0	1.00	1.00	31
JIMI, ZKM(Jitac)	130	4	7	0.97	0.95	101	1	36	0.99	0.74	136	7	1	0.95	0.99	251
JLayer, ZKM(tMP3)	22	0	2	1.00	0.92	14	0	10	1.00	0.58	24	0	0	1.00	1.00	26
Piccolo, ZKM(Crimson)	6	4	6	0.60	0.50	1	0	11	1.00	0.08	12	1	0	0.92	1.00	38
jEdit, ZKM(Jext)	12	1	7	0.92	0.63	9	4	10	0.69	0.47	12	3	7	0.80	0.63	1,507
Average				0.90	0.79				0.94	0.50				0.93	0.92	

Smokescreen reduced the similarity of classes not only in true positives, but also in false positives. As a result, the similarity of classes in false positives decreases below the detection threshold. However, this is not a good effect for a birthmark. If a birthmark is resilient enough, the result of experiments with non-transformed programs should be preserved despite program transformations.

Table 15 shows the experimental results with the ZKM obfuscator. ZKM encrypts strings and complicates control flows by adding opaque predicates. The string encryption method of ZKM uses a technique in which it replaces original strings with encrypted strings and adds decryption method calls. ZKM does not degrade the Tamada birthmark as much as Smokescreen does because it does not use Java standard API calls but user method calls to decrypt strings.

The k-gram birthmark was damaged more than the Tamada birthmark because ZKM added extra JVM instructions to obfuscate control flows and to encrypt strings. For the OTB, the number of false negatives increases because ZKM added string encryption function calls and field variables for opaque predicates in the cases of (JIMI, ZKM(Jitac)) and (jEdit, ZKM(Jext)). However, the number of false positives decreased as in the case of Smokescreen. Therefore, average precision, on the contrary, increased.

Figure 7 shows the evaluation result of three static birthmarks for common module detection. The Tamada birthmark was very vulnerable to the Smokescreen obfuscator. The k-gram birthmark was also vulnerable to the Smokescreen obfuscator and ZKM obfuscator. The OTB showed superior capability for common module detection in all five experiments. It is remarkable that the F-measures increased a little after Smokescreen and ZKM. The reason is, as mentioned above, that the decrease of the number of false positives is larger than the increase of the number of false negatives. However, a good birthmark should get the same results as in the experiment with non-transformed programs, regardless of program obfuscations.

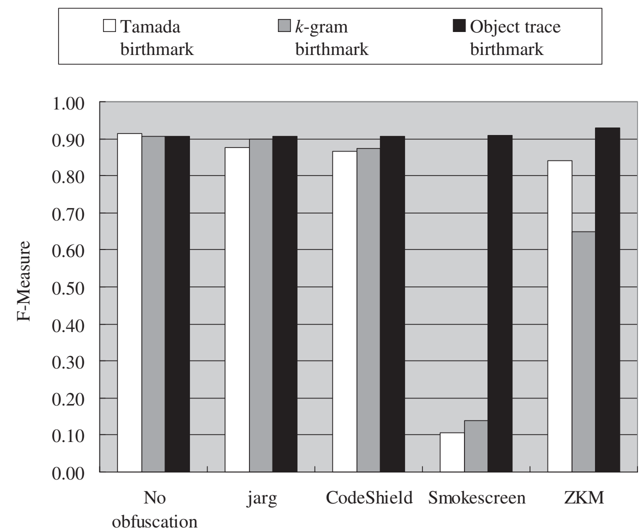

FIGURE 7. Comparison of common module detection of three birthmarks.

6. DISCUSSION

6.1. Attacks on OTB

If a birthmark is opened and a thief wants to hide the code theft against the birthmark, he will try to apply various obfuscations specialized to the birthmark. For example, if he wants to attack OTB, he can add new object instructions using opaque predicates. However, this attack is not a threat to the OTB because original object traces are still alive. In other words, even though some instructions are added by obfuscators to the original program, the similarity between the original and obfuscated programs remains high.

In order to modify the birthmark, the thief can change the order of independent object instructions of a program. However, it is not trivial, because it is very subtle and time-consuming work to reorder object instructions while preserving original

behaviors of the program. Even if the attacker succeeds in removing the birthmark of a tiny part of the program, the portion of the removed part will not be high because only the birthmark of that part is modified. Moreover, we expect that the proposed birthmarking system can get a partial score from the modified part of the program because it uses a local alignment algorithm to compare extracted birthmarks.

6.2. Scalability

It is important that a birthmark have scalability for it to be used as a detector in a large database. Our birthmark is not fast enough, as we showed in the runtime through the experiment in Section 5. Therefore, it is not suitable for detecting code theft in a large database. The runtime performance of the OTB depends on the complexity of the CFG. In detail, if the CFG contains many nodes and edges, a lot of traces are extracted and it causes the trace-comparing time to increase.

In order to have scalability, we can use a method that reduces the quantity of traces and even improves their quality. In general, traces of low frequency are more valuable than traces of high frequency. If we extract only the traces of low frequency, the number of traces will be reduced, and then the trace-comparing time will be reduced. However, as the number of traces is reduced, the credibility of the birthmark may decrease. Nevertheless the credibility will be tolerable because the low-frequency traces are essential traces that properly represent the inherent characteristics of a program.

6.3. Effects on the choice of parameters

We have many choices of parameters for our birthmark. First, if we select only API calls from all object instructions, our birthmark will become more resilient; but if we select more JVM instructions as a birthmark, our birthmark will become more credible. Second, the length of the trace, k, can be changed. If the k value increases, the birthmark will become more credible. Otherwise, the birthmark will become more resilient. Third, we can increase the detection threshold, ε, to make a more resilient birthmark. Furthermore, the birthmark will become more credible when the ε decreases.

7. SUMMARY AND FUTURE WORK

In this paper, we proposed a new birthmark based on traces of Java object instructions. Because Java is an object-oriented programming language, object instructions are hard to replace or remove. Based on this observation, we defined a set of sequences of Java object instructions as inherent characteristics of the program. To extract an OTB, we generated an OFG by static analysis of the Java bytecodes. In order to calculate the similarity between two object traces, we applied a sequence alignment algorithm. Based on this trace similarity, we obtained the similarities of methods and classes. Finally, to detect common modules between two programs, we used class similarity. We evaluated the proposed birthmark with respect to the capability of detecting common modules for open source programs. For the evaluation, we adopted precision and recall as evaluation measures. The experimental results showed that the detection capability of our birthmark is much higher than that of other static birthmarks against obfuscation attacks by Smokescreen and ZKM.

The future work with regard to our birthmark is as follows. First, our birthmark is hard to apply to classes that have a few object instructions. To cope with this problem, we plan to supplement our birthmark with other essential elements such as array-related instructions, which are hard to modify. Second, we plan to compare the OFGs of two programs directly. The OFGs have valuable information such as loops and branches. These characteristics can be extracted and compared as birthmarks. As a previous work related to graph comparison, there is research that generates an object process graph and applies it to protocol recovery [44]. In order to be used as a birthmark, however, graph comparison methods have to consider the graph modifications that are caused by the insertion of dummy nodes or the splitting of existing nodes. As a result, a new graph comparison method has to be suggested that is more appropriate for birthmarks than existing methods.

FUNDING

This work was partially supported by the Korea Science and Engineering Foundation(KOSEF) grant funded by the Korea government(MEST) (No. R01-2008-000-11856-0). Also, this work was partially supported by the MKE(The Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) Support program supervised by the NIPA(National IT industry Promotion Agency) (NIPA-2009-C1090-0902-0020).

REFERENCES

[1] Coar, K. (2006) *The open source definition*. http://www.opensource.org/docs/osd/ (accessed October 27, 2009).

[2] Open Source Initiative (2006) *Open source licenses*. http://www.opensource.org/licenses/ (accessed October 27, 2009).

[3] SourceForge.net: Open Source Software. http://www.sourceforge.net (accessed October 27, 2009).

[4] Kueng, J. (2007) *Skype alleged to have violated the GPL*. http://www.linuxdevices.com/news/NS7183268189.html (accessed October 27, 2009).

[5] Gohring, N. (2007) Cisco's iPhone violates GPL. http://www.infoworld.com/article/07/01/17/HNciscoiphonegpl_1.html (accessed October 27, 2009).

[6] Welte, H. *The gpl-violations.org project*. http://www.gpl-violations.org/ (accessed October 27, 2009).

[7] Gosling, J., Joy, B., Steele, G. and Bracha, G. (2000) *The Java Language Specification* (2nd edn). Addison-Wesley, Reading, MA, USA.

[8] Mocha—the Java Decompiler. http://www.brouhaha.com/~eric/software/mocha/ (accessed October 27, 2009).

[9] JODE—Open source Java decompiler and obfuscator. http://jode.sourceforge.net/ (accessed October 27, 2009).

[10] SourceAgain—Java Decompiler. Ahpah Software, Inc. http://www.ahpah.com/products.html (accessed October 27, 2009).

[11] Wise, M. (1996) YAP3: improved detection of similarities in computer program and other texts. *ACM SIGCSE Bull.*, **28**, 130–134.

[12] Prechelt, L., Malpohl, G. and Philippsen, M. (2002) Finding plagiarisms among a set of programs with jPlag. *J. Univ. Comput. Sci.*, **8**, 1016–1038.

[13] Koschke, R. (2007) Survey of Research on Software Clones. *Proc. Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software*, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Dagstuhl, Germany.

[14] Roy, C.K. and Cordy, J.R. (2007) A Survey on Software Clone Detection Research. *TR 2007-541 School of Computing,* Queen's University, Kingston Ontario, Canada.

[15] Collberg, C. and Thomborson, C. (2002) Watermarking, tamper proofing, and obfuscation—tools for software protection. *IEEE Trans. Softw. Eng.*, **28**, 735–746.

[16] Collberg, C., Thomborson, C. and Townsend, G. (2007) Dynamic graph-based software fingerprinting. *ACM Trans. Program. Lang. Syst.*, **29**, 35.

[17] Grover, D. (1989) Program Identification. *The Protection of Computer Software—Its Technology and Applications*, pp. 122–154. Cambridge University Press, New York, NY, USA.

[18] Tamada, H., Nakamura, M., Monden, A. and Matsumoto, K. (2004) Design and Evaluation of Birthmarks for Detecting Theft of Java Programs. *Proc. IASTED Int. Conf. Software Engineering*, Innsbruck, Austria, February 17–19, pp. 569–575. ACTA Press, Calgary, Canada.

[19] Tamada, H., Nakamura, M., Monden, A. and Matsumoto. K. (2005) Java birthmarks-detecting the software theft-*IEICE Trans. Inf. Syst.*, **E88-D**, 2148–2158.

[20] Myles, G. and Collberg, C. (2005) k-gram Based Software Birthmarks. *ACM Symp. Applied Computing*, Santa Fe, NM, USA, March 13–17, pp. 314–318. ACM Press, New York, NY, USA.

[21] Myles, G. (2006) Software theft detection through program identification. PhD Thesis, Department of Computer Science, The University of Arizona, Tucson, AZ, USA.

[22] Park, H., Choi, S., Lim, H. and Han, T. (2008) Detecting Code Theft via a Static Instruction Trace Birthmark for Java Methods. *The 6th IEEE Int. Conf. Industrial Informatics* (*INDIN*), Daejeon, Korea, July 13–16, pp. 551–556. IEEE Computer Society.

[23] Park, H., Choi, S., Lim, H. and Han, T. (2008) Detecting Java Theft Based on Static API Trace Birthmark. *The 3rd Int. Workshop on Security* (*IWSEC*), Lecture Notes in Computer Science 5312, Kagawa, Japan, November 25–27, pp. 121–135. Springer, Berlin, Heidelberg, Germany.

[24] Myles, G. and Collberg, C. (2004) Detecting Software Theft via Whole Program Path Birthmarks. *Information Security Conf.* (*ISC*), Lecture Notes in Computer Science 3225, Palo Alto, CA, USA, September 27–29, pp. 404–415. Springer, Berlin, Heidelberg, Germany.

[25] Nevill-Manning, C.G. and Witten, I.H. (1997) Compression and explanation using hierarchical grammars. *Comput. J.*, **40**, 103–116.

[26] Schuler, D., Dallmeier, V. and Lindig, C. (2007) A Dynamic Birthmark for Java. *22nd IEEE/ACM Int. Conf. Automated Software Engineering*, November 5–9, Atlanta, GA, USA, pp. 274–283. ACM Press, New York, NY, USA.

[27] Lindholm, T. and Yellin, F. (1999) *The Java Virtual Machine Specification* (2nd edn). Addison-Wesley, Reading, MA, USA.

[28] Java Platform Standard Edition 6 API Specification. http://java.sun.com/javase/6/docs/api/ (accessed October 27, 2009).

[29] Needleman, S. and Wunsch, C. (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, **48**, 443–453.

[30] Smith, T. and Waterman, M. (1981) Identification of common molecular subsequences. *J. Mol. Biol.*, **147**, 195–197.

[31] Brudno, M., Malde, S., Poliakov, A., Do, C.B., Couronne, O., Dubchak, I. and Batzoglou. S. (2003) Glocal alignment: finding rearrangements during alignment. *Bioinformatics*, **19**, 54–62.

[32] Kuhn, H. (1955) The Hungarian method for the assignment problem. *Nav. Res. Logist. Q.*, **2**, 83–97.

[33] javap–The Java Class File Disassembler. http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/javap.html (accessed October 27, 2009).

[34] Salton, G. and Mcgill, M.J. (1986) *Introduction to Modern Information Retrieval*. McGraw-Hill, Columbus, OH, USA.

[35] Baeza-Yates, R. and Ribeiro-Neto, B. (1999) *Modern Information Retrieval*. Addison-Wesley, Reading, MA, USA.

[36] Eclipse. http://www.eclipse.org/ (accessed October 27, 2009).

[37] Stigmata–Java birthmark toolkit. http://stigmata.sourceforge.jp/ (accessed October 27, 2009).

[38] Java Archive Grinder. http://sourceforge.net/projects/jarg/ (accessed October 27, 2009).

[39] CodeShield Java Bytecode Obfuscator. http://www.codingart.com/codeshield.html (accessed November 18, 2008).

[40] Smokescreen Java Obfuscator. http://www.leesw.com/smokescreen/ (accessed October 27, 2009).

[41] Java Obfuscator—Zelix KlassMaster. http://www.zelix.com/klassmaster/ (accessed October 27, 2009).

[42] Collberg, C., Thomborson, C. and Low. D. (1998) Manufacturing Cheap, Resilient and Stealthy Opaque Constructs. *Proc. 25th ACM SIGPLAN-SIGACT Symp. Principles of programming Languages* (*POPL*), San Diego, CA, USA, January 19–21, pp. 184–196. ACM Press, New York, NY, USA.

[43] Arboit, G. (2002) A Method for Watermarking Java Programs via Opaque Predicates. *Proc. Int. Conf. Electronic Commerce Research* (*ICECR-5*), Montreal, Canada, October 23–27.

[44] Quante, J. and Koschke, R. (2007) Dynamic Protocol Recovery. *Proc. IEEE Working Conf. Reverse Engineering* (*WCRE*), Vancouver, Canada, October 28–31, pp. 219–228. IEEE Computer Society.

[45] Damiani, M. L., Silvestri, C. and Bertino E. (2008) Semantics-aware Obfuscation for Location Privacy. *Journal of Computing Science and Engineering*, 2(2), 137–160.