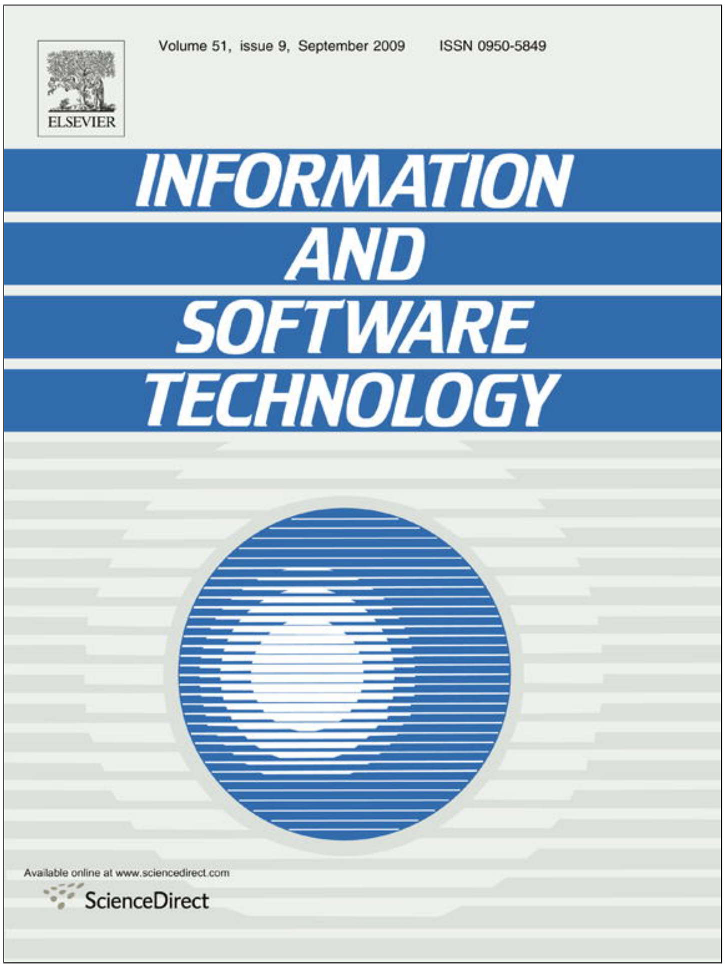


Volume 51, issue 9, September 2009 ISSN 0950-5849

ELSEVIER

INFORMATION AND SOFTWARE TECHNOLOGY

Available online at www.sciencedirect.com

ScienceDirect

Information and Software Technology 51 (2009) 1338–1350

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A method for detecting the theft of Java programs through analysis of the control flow information ☆

Hyun-il Lim *, Heewan Park, Seokwoo Choi, Taisook Han

Division of Computer Science, Korea Advanced Institute of Science and Technology, 335 Gwahangno, Yuseong-gu, Daejeon 305-701, Republic of Korea

ARTICLE INFO

Article history:
Received 26 October 2008
Received in revised form 29 March 2009
Accepted 25 April 2009
Available online 5 May 2009

Keywords:
Software birthmark
Software copyright protection
Software theft detection
Java bytecode analysis

ABSTRACT

A software birthmark refers to the inherent characteristics of a program that can be used to identify the program. In this paper, a method for detecting the theft of Java programs through a static software birthmark is proposed that is based on the control flow information. The control flow information shows the structural characteristics and the possible behaviors during the execution of program. Flow paths (FP) and behaviors in Java programs are formally described here, and a set of behaviors of FPs is used as a software birthmark. The similarity is calculated by matching the pairs of similar behaviors from two birthmarks. Experiments centered on the proposed birthmark with respect to precision and recall. The performance was evaluated by analyzing the *F*-measure curves. The experimental results show that the proposed birthmark is a more effective measure compared to earlier approaches for detecting copied programs, even in cases where such programs are aggressively modified.

1. Introduction

Software is the intellectual property of its developers. It is protected by copyright law and regulations. Software piracy is the act of making unauthorized copies of computer software or reusing the source codes of programs without permission or observing the software license. A recent study of piracy [7] reported that the losses caused by software piracy are increasing every year; lowering the rate of software piracy, according to this study, will have a number of beneficial economic effects. In the case of open source software, the piracy is a different issue. Many programs are distributed with the source code, and subscribers are allowed to modify or redistribute program code under certain types of software licenses. For example, software developed using GNU General Public License (GPL) software should follow the tenets of GPL by opening the source code to the public. However, it has been reported that many software developers and companies do not follow the license policy [1]. If the license policy for individual software is not obeyed, intellectual property rights for software are weakened.

Software is a type of digital media that can be copied and distributed without obtaining the permission of the owner. In order to reduce and cope with software theft, it is necessary to protect against illegal tampering and to identify the originality of software. However, this is not easy, as stolen software can be distributed without the source codes. Moreover, binary executables are not suitable for a direct comparison with other software. The software may be obfuscated or modified to hide the fact of software theft. Hence, in several lawsuits involving GPL violations, including Sony's PlayStation 2 ICO and Skype's VoIP telephone, the evidence has been shown of manual reverse engineering, which is tedious and time-consuming. This environment brings about a pervasive impulse to steal source code of programs. Therefore, in order to deal with software theft, the development of efficient technology that identifies the originality of software is necessary.

A software birthmark is a technique that is used for identifying the originality of software. The technology is often used to denote the inherent characteristics extracted from programs themselves so as to discriminate one from another. By comparing these characteristics, it is possible to observe any similarity between the programs. A high level of similarity relative to an original program implies that a program is a copy.

This paper presents a method for detecting the theft of Java programs by analyzing the control flow information. Given that the control flow information shows the possible control flows when a program is executed, this information represents the behavioral characteristics of the program. It can be also used as a means of identifying the originality of a program. The flow paths (FP) of program are formally described, and the behaviors of the FP are used as the discriminating characteristics of the program. The proposed method can be used to detect the theft of modules or libraries by

☆ This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (R01-2008-000-11856-0).

* Corresponding author. Tel.: +82 42 350 5560.
E-mail addresses: hilim@pllab.kaist.ac.kr (H.-i. Lim), hwpark@pllab.kaist.ac.kr (H. Park), swchoi@pllab.kaist.ac.kr (S. Choi), han@cs.kaist.ac.kr (T. Han).

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2009.04.011

comparing the characteristics extracted from a certain part of an original program with a suspicious program.

The proposed birthmark is evaluated with respect to two properties: credibility and resilience. The credibility property requires the birthmark to differentiate different programs clearly, while the resilience property requires that the birthmark is able to detect the theft of a program even after the stolen programs are modified by semantics-preserving transformations. Experiments are conducted regarding the proposed birthmark with respect to precision and recall and the performance is evaluated by analyzing the *F*-measure curves. Additionally, the methods are compared with three previous approaches, the birthmark of Tamada et al. [43,44], the *k*-gram based birthmark [29,32], and the stack pattern based birthmark [23]. The experimental results show that the proposed method detects copied programs more effectively, even in cases in which the programs are aggressively modified.

This paper is organized as follows. Section 2 reviews existing approaches to software theft detection. Section 3 describes the motivation and key idea of the proposed approach. Section 4 gives the formal definition of a software birthmark and proposes a static Java birthmark based on control flow information. Section 5 presents experimental data and evaluates the proposed birthmark. Section 6 discusses several additional issues regarding the proposed method. Finally, Section 7 concludes the paper.

2. Related work

There have been a lot of researches on detecting or preventing software thefts. If source codes of programs are available, textual based plagiarism detection methods can be applied to detect suspected copies. These methods try to find the plagiarized programs by comparing structures in source codes of the programs, such as text itself, program syntax, or program styles. JPlag [35,36] and YAP3 [48] compared token sequences parsed from the source codes of programs by using greedy string tiling algorithm [47]. Sim [21] compared strings extracted from the parse trees of programs by using a string alignment algorithm. Moss [10] compared the fingerprints of programs that were constructed through winnowing algorithm [40]. Burrows et al. [12] presented an efficient plagiarism detection strategy by separating a feasible set of copied programs from large code repositories using inverted indexes of programs. Because most commercial programs are distributed only in the form of binary executables, these approaches are limited to environments where the source codes are available.

Software watermarks are techniques used to find stolen programs by recognizing previously embedded identifiers. Static watermarks make use of software's features that are available at static time to embed such identifiers. Because static watermarks are stored in the software itself, the recognition of watermarks does not require executing the software. The static watermarks can be encoded in several ways, such as basic block reordering [19] or register reallocation [37,27], and inserted through opaque predicates [8,26,30]. Dynamic watermarks are stored in execution states of programs, rather than in the program codes themselves, so the recognition of watermarks needs to execute the programs. The dynamic watermarks are embedded in such a way that the identifiers can be extracted while programs are running with predefined input sequences. There have been several approaches for dynamic watermarking, such as Easter egg watermark [13,14], data structure watermark [13], execution trace watermark [16,17], and branch-based watermark [31]. Software watermarking methods can be applied only to programs that have been watermarked before releasing.

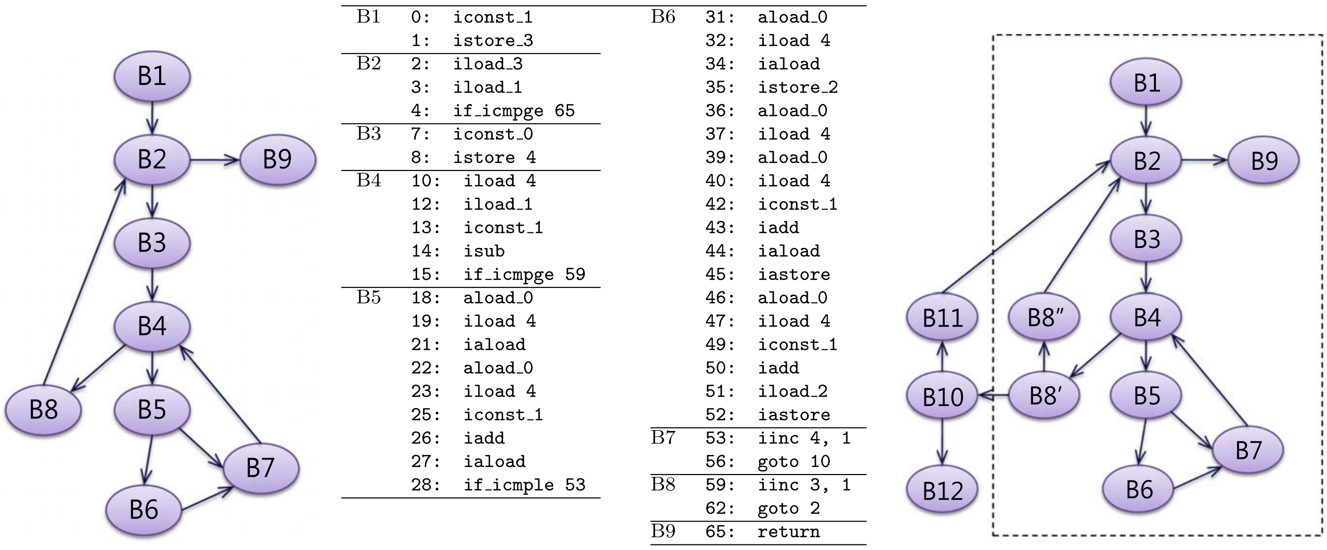
For general approaches for detecting software thefts from binary programs, there have been several researches on software birthmarks. Tamada et al. [43,44] considered four structural characteristics of Java programs to identify the originality of the programs: the constant values in field variables (CVFV), the sequence of method calls (SMC), the inheritance structure (IS), and the used classes (UC). These birthmarks depend on comparisons of the names of classes or methods, so the approach is susceptible to modification of such information. Myles and Collberg [29,32] used the sequences of *k* contiguous opcodes in programs as the discriminating characteristics. Lim et al. [23] used the sequences of contiguous opcodes which were partitioned on the basis of their operand stack depth. Because the two approaches only consider the physical orders of instructions, they are susceptible to program modifications that change control flows of programs. Park et al. presented methods for detecting software thefts by comparing the full traces of instructions [34] or API calls [33] of programs. These approaches can be applied only to small programs, because the number of traces may exponentially increase according to the program structures. Tamada et al. [45] and Schuler et al. [41,42] used the run-time behaviors of API calls during execution stages of programs as the characteristics for identifying the programs. The whole program path birthmark by Myles and Collberg [28,32] extracted the characteristics of programs through graph representations that were compressed from dynamic traces of the programs.

3. Motivation

Dynamic and static approaches exist for software birthmarks. Dynamic approaches abstract the characteristics of a program by recording the actual behaviors during the execution of the program for a given input. These approaches are highly dependent on the given input and the run-time environments. Moreover, dynamic approaches cannot feasibly cover all possible program paths; they can only extract the characteristics from the run-time slice of a program. In particular, when certain suspect modules of a program are believed to have been stolen from an original program, it is difficult to maintain the characteristics of the relevant part of the program. On the other hand, static approaches use the static information of a program and do not require the execution of the program to extract its identifying characteristics. Because static approaches mainly focus on static information that is obtained from program codes, they can readily investigate suspicious parts of a program or cover all components of a program. However, these methods tend to overlook program's features appearing from operational behaviors of a program, which can be a more reasonable measure for examining the originality of software. As a solution that supplements the shortcoming of the static approaches, the control flow information of programs is considered. From the control flow information of programs, the possible execution paths of the programs can be collected at static time.

Many program analysis techniques rely on control flow information that is derived from a control flow graph (CFG) [46]. In the Java language [22], source codes are compiled into a portable binary format which is known as Java bytecodes. These bytecodes are machine codes that can run directly on the Java Virtual Machine [25]. As the design of Java bytecodes focuses on portability issues, which requires a considerable number of high-level features, Java bytecodes inherit most of their structures from the source programs, including the control flows, class inheritance structures, and variable usage patterns. Hence, the control flow graphs generated from the bytecodes can be used with several analysis techniques that are useful for understanding Java programs. Additionally, the information can be used as features for identification among Java programs.

Fig. 1a shows the CFG and the basic blocks of Java bytecodes for a bubble sort program implemented in Java language. In the example, the control structure in the source program is mostly preserved in



```
B1   0:  iconst_1          B6  31:  aload_0
     1:  istore_3              32:  iload 4
B2   2:  iload_3               34:  iaload
     3:  iload_1               35:  istore_2
     4:  if_icmpge 65          36:  aload_0
B3   7:  iconst_0              37:  iload 4
     8:  istore 4              39:  aload_0
B4  10:  iload 4               40:  iload 4
    12:  iload_1               42:  iconst_1
    13:  iconst_1              43:  iadd
    14:  isub                  44:  iaload
    15:  if_icmpge 59          45:  iastore
B5  18:  aload_0               46:  aload_0
    19:  iload 4               47:  iload 4
    21:  iaload                49:  iconst_1
    22:  aload_0               50:  iadd
    23:  iload 4               51:  iload_2
    25:  iconst_1              52:  iastore
    26:  iadd              B7  53:  iinc 4, 1
    27:  iaload                56:  goto 10
    28:  if_icmple 53      B8  59:  iinc 3, 1
                               62:  goto 2
                           B9  65:  return
```

(a) The CFG of Java bytecodes for a bubble sort program.

(b) Obfuscated by Smokescreen.

Fig. 1. The CFG and the basic blocks of a bubble sort program and its obfuscated CFG.

the CFG of the Java bytecodes even after the source code is compiled to Java bytecodes. Fig. 1b shows the CFG of the program transformed by the process of the Smokescreen obfuscator [5]. The obfuscated CFG appears to differ somewhat from the original CFG. However, careful inspection reveals that the core control flows in the original CFG remain intact (as shown in the dotted box). Although some control flow obfuscation modifies the CFG of a program, the sequence of control flows are likely to remain as the semantics of the program must be preserved. Particularly, the sequence of control flows forms execution paths of the program.

Execution paths of a program show feasible sequences of instructions while the program executes. These sequences of instructions can be abstracted from the control flow information of a program and specifically indicate how operations are performed to complete some arranged task. So, the degree of analogy between execution paths of programs can be a viable evidence to determine whether one program is copied from another. In this paper, a method for detecting a theft of Java programs is presented by comparing abridged execution paths analyzed from the control flow information.

4. Software birthmark based on control flow information

4.1. Software birthmark

Software birthmark refers to a method for detecting cases of software theft through comparisons of the inherent characteristics of programs. For criteria that are more definitive in the determination of software theft, the copy relation of programs is extended from [44] as follows:

Definition 1 (*Copy Relation*). For programs P and Q, the two programs are considered to be in copy relation if one of the following conditions are satisfied:

- Program Q is exactly duplicated from program P.
- Program Q is obtained from P by renaming identifiers.
- Program Q is obtained from P by applying semantics-preserving transformations of optimization, obfuscation, or recompilation.
- Program Q is obtained from P by capturing one or more modules in P.
- Program Q is obtained from P by iteratively applying the above modifications.

Copy relation represents a relationship in which one program originates from another. For a more concrete formulation, a *software birthmarking system* [32] that detects programs in copy relation is defined by two functions: *extract*() and *compare*(). For a given program P, the function *extract*(P) generates a set of characteristics (a birthmark) that differs from another program. Correspondingly, for given birthmarks bm_P and bm_Q, the function $compare(bm_P, bm_Q)$ calculates the similarity between the birthmarks. The similarity value ranges between 0 and 1 in proportion to the degree of similarity, and an analysis of the similarity between the two birthmarks can determine if the two programs are in copy relation. For example, two programs are suspected to be in copy relation if the similarity is higher than $1-\epsilon$, for a given threshold value of ϵ. For this purpose, the software birthmarking system must satisfy the following two properties [32]:

Property 1 (*Credibility*). *Let P and Q be independently written programs. The software birthmarking system is then evaluated as credible if the system can discriminate the two programs; that is, $similarity(bm_P, bm_Q) < 1-\epsilon$.*

Property 2 (*Resilience*). *Let program Q be in copy relation with program P. The software birthmarking system is then evaluated as resilient if the system can discern that the two programs are in copy relation; that is, $similarity(bm_P, bm_Q) \geqslant 1-\epsilon$.*

The *credibility* property is a criterion that excludes the possibility of false positives. In other words, although two programs may have functionalities that are identical, the programs will have different birthmarks if they are developed independently. The *resilience* property specifies that the birthmarking system detects programs in copy relation even when the programs have been modified. This implies that the system should reduce the possibility of false negatives by as much as possible.

4.2. Flow path

A *control flow graph* [46,49] is a graph representing a program structure in which nodes and edges describe the basic blocks and the possible control flows, respectively. A *basic block* contains a sequence of instructions with a single entry point and a single exit point. When control enters into a basic block, the control can begin only at the entry point of the basic block and leave the basic block only at its exit point. For basic blocks v_1 and v_2, if an edge exists from v_1 to v_2, this indicates that an execution may occur in which v_2 is executed after v_1.

In order to perform control flow analysis of Java bytecodes, it is necessary to extend existing control flow analysis techniques to adapt Java bytecodes. The procedure for constructing a CFG for a Java program is as follows:

(1) Initially, basic blocks are determined by finding the set of headers that can be the entry point of each basic block. The headers apply in following instructions:
- the first instruction of a method,
- the targets of branch instructions, or
- the immediate following instructions of conditional branches.

(2) After all basic blocks are determined, the CFG is constructed by adding edges between basic blocks, where control flows exist. The Java Virtual Machine provides jump to subroutine (`jsr`, `jsr_w`) and return (`ret`) instructions, which also construct edges in the CFG. After pairing each `jsr` or `jsr_w` with its corresponding `ret`, control flow edges are added from the basic blocks containing `jsr` or `jsr_w` to their jump target basic blocks. Their return edges are then added from the basic blocks containing their corresponding `ret` instructions to the next basic blocks of the `jsr` or `jsr_w` instructions.

(3) In Java, exceptions change the control flow of program. For example, if an exception occurs, the control flow jumps to the handler routine of the exception. In addition, obfuscators may change branch constructs using exception handling routines. In order to manage this situation, the exception edges are added to the CFG of Java program. The exception table maintains the ranges of instructions managed by exception handlers and maintains the addresses of their exception handler routines. From this information, the exception edges are added from the basic blocks in the ranges of the exceptions to the target addresses for their handler routines.

From the CFG of a program, it is possible to determine the *flow paths* (*FP*) of the program. These are the sequences of basic blocks that are possible traces during program execution. The FPs are abstract information that represents the behavioral characteristics of the program. A FP is defined as follows.

Definition 2 (*k-Flow Path*). Let $G = (V, E)$ be the control flow graph of a program. $V = \{v_1, v_2, \ldots, v_n\}$ is the set of basic blocks, and $E = \{(v_i, v_j) | v_j$ is directly reachable from $v_i\}$ is the set of edges in G. Then a k-flow path (k-FP) is defined as follows.

(1) If $k = 1$, each node of the CFG G is the 1-FP. Hence, a set of 1-FP, FP_1, is as follows:

$$FP_1(G) = \{\langle v_1 \rangle, \langle v_2 \rangle, \ldots, \langle v_n \rangle\}. \tag{1}$$

(2) If $k > 1$, for a $\langle v_{i_1}, v_{i_2}, \ldots, v_{i_{k-1}} \rangle \in FP_{k-1}(G)$, and some $v \in N_G(v_{i_{k-1}}), \langle v_{i_1}, v_{i_2}, \ldots, v_{i_{k-1}}, v \rangle$ is a k-FP, where $N_G(v_{i_{k-1}})$ denotes the set of vertices that are directly reachable from the vertex $v_{i_{k-1}}$ through non-exception edges. The set of k-FP, FP_k, is then as follows:

$$FP_k(G) = \{\langle v_{i_1}, v_{i_2}, \ldots, v_{i_{k-1}}, v \rangle | \langle v_{i_1}, v_{i_2}, \ldots, v_{i_{k-1}} \rangle \in FP_{k-1}(G) \quad \text{and} \quad v \in N_G(v_{i_{k-1}})\}. \tag{2}$$

The FPs of a CFG are abstract information constructed from the CFG of a program. In Eq. (1), each basic block constitutes 1-FP of a CFG. The 1-FPs do not include the information of the control flow edges; they only contain the information of each node. In Eq. (2), k-FP is defined by induction in such a way that the k-FPs are constructed from $(k-1)$-FPs by appending basic blocks reachable from the end of the $(k-1)$-FPs. Because throwing exceptions is not in a normal program condition, exception edges are excluded from generated FPs. Since the k-FPs are constructed from the CFG of a program, they represent the possible sequences of basic blocks when the program is executed. The procedure for calculating the k-FPs from a CFG is described in Algorithm 1.

Algorithm 1. Calculating k-FP from control flow graph.

```
INPUT Control flow graph $G = (V, E)$ and $k$ (the length of $FP$)
OUTPUT $FP_k$
for all node $v \in V$ do
    Insert $\langle v \rangle$ to $FP_1$;
end for
for $i = 2$ to $k$ do
    for all $\langle v_1, \ldots, v_{i-1} \rangle \in FP_{i-1}$ do
        for all $v \in N_G(v_{i-1})$ do
            Insert $\langle v_1, \ldots, v_{i-1}, v \rangle$ to $FP_i$;
        end for
    end for
end for
```

The CFG of a program shows the behavioral and structural characteristics of the program, such as branches, loops, and sequences. Hence, the CFG is an effective structure for characterizing individual programs. However, the CFG assumes the form of a graph, which is difficult to manipulate or compare. For this reason, a simple representation of the CFG is necessary to abstract behavioral characteristics. The set of FPs in a CFG is an abridged representation of the CFG. Additionally, for an arbitrary CFG, there exists only one set of FPs. Therefore, this set can be used as characteristics in the identification of program. From the definition of k-FP, the behavior of k-FP is defined as follows.

Definition 3 (*Behavior of k-FP*). Let $G = (V, E)$ be the CFG of a program and $bc(v)$ be the sequence of bytecodes in the basic block $v \in V$. The behavior of a k-FP, denoted by $behavior(k - \text{FP})$, is then defined as follows:

(1) If $k = 1$, let $\langle v \rangle$ be a 1-FP of the CFG G, i.e., $v \in V$. Then $bc(v)$ is the behavior of $\langle v \rangle$.

(2) If $k > 1$, let $\langle v_1, \ldots, v_{k-1}, v_k \rangle$ be a k-FP of the CFG G. Then $concat(behavior(\langle v_1, \ldots, v_{k-1} \rangle), bc(v_k))$ is the behavior of the k-FP $\langle v_1, \ldots, v_{k-1}, v_k \rangle$, where $concat(x, y)$ denotes the concatenation of two sequences x and y.

The *behavior of FP* is the specific sequence of bytecodes executed by the FP. Therefore, the behavior is organized by the bytecodes stored in the basic blocks constructing the FP.

For example, let Fig. 1a be the CFG G of a program P. The sets of k-FP, denoted by $FP_k(G)$, then take the following form:

$$\begin{aligned}
FP_1(G) = \{&\langle B1 \rangle, \langle B2 \rangle, \langle B3 \rangle, \langle B4 \rangle, \langle B5 \rangle, \langle B6 \rangle, \langle B7 \rangle, \langle B8 \rangle, \langle B9 \rangle\},\\
FP_2(G) = \{&\langle B1, B2 \rangle, \langle B2, B3 \rangle, \langle B3, B4 \rangle, \langle B4, B5 \rangle, \langle B5, B6 \rangle,\\
&\langle B6, B7 \rangle, \langle B5, B7 \rangle, \langle B7, B4 \rangle, \langle B4, B8 \rangle, \langle B8, B2 \rangle, \langle B2, B9 \rangle\},\\
FP_3(G) = \{&\langle B1, B2, B3 \rangle, \langle B1, B2, B9 \rangle, \langle B2, B3, B4 \rangle, \langle B3, B4, B5 \rangle,\\
&\langle B3, B4, B8 \rangle, \langle B4, B5, B6 \rangle, \langle B4, B5, B7 \rangle, \langle B5, B6, B7 \rangle,\\
&\langle B6, B7, B4 \rangle, \langle B5, B7, B4 \rangle, \langle B7, B4, B5 \rangle, \langle B7, B4, B8 \rangle,\\
&\langle B4, B8, B2 \rangle, \langle B8, B2, B3 \rangle, \langle B8, B2, B9 \rangle\},\\
&\vdots
\end{aligned}$$

The behaviors of FPs are then constructed by concatenating the specific sequences of bytecodes contained in the corresponding basic blocks.

4.3. The proposed birthmark

In order to detect software theft via a software birthmark, it is important to measure the similarity between two birthmarks. For a complete birthmarking system, it is necessary to provide a function for extracting the birthmarks from programs and a measure for calculating the similarity between the birthmarks.

Definition 4 (*Flow Path Birthmark*). For a Java program P, let $M_1, \ldots, M_n$ be the methods in P. Let G_i be the CFG of the method M_i. The k-FP birthmark of the method $M_i, bm_k^i(P)$, is defined as follows:

$$bm_k^i(P) = \{behavior(a) | a \in FP_k(G_i)\}. \tag{3}$$

The k-FP-based birthmark of the program P, $\mathbf{bm}_k(P)$, is then defined as follows:

$$\mathbf{bm}_k(P) = \bigcup_{i=1}^{n} bm_k^i(P). \tag{4}$$

The *flow path* (*FP*) *birthmark* of a method consists of the behaviors of FPs generated from the method, and the birthmark of a program is the union of FP birthmarks of all methods in the program.

The behaviors in a Java program represent the sequence of bytecodes that is executed along FPs of the program. For two programs, a state of similar CFGs implies the strong possibility that one program has been copied from the other. Because the behaviors originate from the CFG of the program, similar behaviors in two programs indicate that the two CFGs are also similar, which is strong evidence that the two programs are in copy relation.

4.4. Matching two behaviors

The FP birthmark of a program consists of the behaviors of FPs in the CFG of the program. In other words, the birthmark is represented by a set of behaviors of the FPs, and the behaviors are the elements of the birthmark. Hence, it is necessary to examine all behaviors to ascertain the similarity between two programs. A behavior refers to the possible sequence of bytecodes that can be executed during program execution. Thus, if behaviors are similar between programs, it can be inferred that two behaviors perform similar tasks in the programs.

The first step when calculating the similarity between two birthmarks is to determine the matching point among the behaviors in each birthmark. This measure represents the similarity between two behaviors. In order to compare the behaviors, the *semi-global alignment* algorithm [11] was applied. This algorithm is often used in DNA sequence alignment problems. In order to align two sequences, the necessary operations are as follows:

- matches to align pairs of matched elements,
- mismatches to align pairs of mismatched elements in both sequences, and
- gaps to skip one mismatched elements in one of the two sequences.

The procedure for aligning behaviors is accomplished by dynamic programming. Let $a = (x_1, \ldots, x_m)$ be a behavior in the original program and $b = (y_1, \ldots, y_n)$ be a behavior in a suspicious program. Beginning at the top left cell, the matching point up to position $(i, j), c[i, j]$, is calculated as follows:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0, \\ -(i \times gap) & \text{if } i > 0 \quad \text{and} \quad j = 0, \\ c[i-1, j-1] + w(x_i) & \text{if } i,j > 0 \quad \text{and} \quad x_i = y_j, \\ \max \begin{cases} c[i-1, j-1] - \sigma \\ c[i-1, j] - gap \\ c[i, j-1] - gap \end{cases} & \text{if } i,j > 0 \quad \text{and} \quad x_i \neq y_j, \end{cases} \tag{5}$$

where $w(x_i)$ denotes the matching weight of the bytecode x_i, σ denotes the mismatching penalty for one mismatched pair of elements in both sequences, and *gap* denotes the penalty for skipping a mismatched element in one of the sequences.

After calculating the matching point for each cell in dynamic programming, the resulting value of the semi-global alignment is obtained by finding the maximum value among the values in the bottom row, as follows:

$$Semi\text{-}Global(a, b) = \max_j (c[m, j]). \tag{6}$$

By penalizing for mismatches and gaps, this method can discriminate different behaviors with high sensitivity. In addition, it is more appropriate for comparing the original behaviors with modified behaviors that may be augmented by some additional elements, be-

behavior a		behavior b	
Bytecode	w	Bytecode	w
iconst_0	1	aload_0	1
istore	3	iload	3
iload	3	iaload	6
iload_1	2	istore_2	3
iconst_1	1	iload_1	2
isub	4	iconst_1	1
if_icmpge	3	isub	4
		istore_3	2
		iload_3	3
		if_icmpge	3

(a) Sample behaviors and the matching weights of their bytecodes.

		aload_0	iload	iaload	istore_2	iload_1	iconst_1	isub	istore_3	iload_3	if_icmpge
	0	**0**	0	0	0	0	0	0	0	0	0
iconst_0	−1	**−1**	−1	−1	−1	−1	−1	−1	−1	−1	−1
istore	−2	**−2**	−2	−2	−2	−2	−2	−2	−2	−2	−2
iload	−3	−3	**1**	**0**	**−1**	−2	−3	−3	−3	−3	−3
iload_1	−4	−4	0	−1	−2	(a)**1**	(b)0	−1	−2	−3	−4
iconst_1	−5	−5	−1	−2	−3	0	**2**	1	0	−1	−2
isub	−6	−6	−2	−3	−4	−1	1	**6**	**5**	**4**	3
if_icmpge	−7	−7	−3	−4	−5	−2	0	5	4	3	**7**

(b) Semi-global alignment: calculating the matching point between two behaviors.

Fig. 2. Sample behaviors and semi-global alignment algorithm.

cause this method does not consider the penalties caused by the heading and tailing mismatches of one sequence.

The matching point depends on the matching weights of matched bytecodes. The matching point should be determined not by how many bytecodes are matched but by how much similar operations are performed by the behaviors. For example, it is more desirable to assign lower matching weights to general instructions used in several places, such as `aload_0`, `ldc`, and `return`, and higher matching weights to instructions that represent specific operations, such as `iadd`, `isub`, and `if_icmpge`. So, the matching weight is designed with respect to the specificity of bytecodes using a weighting scheme, the inverse document frequency [38].

Let C be the total number of class files observed, and $num(\texttt{opc})$ be the number of class files in which the opcode `opc` appears among the class files. The inverse document frequency of opcode `opc` is then calculated as follows:

$$idf(\texttt{opc}) = \log \frac{C}{num(\texttt{opc})}. \tag{7}$$

The matching weights of individual bytecodes are then calculated by normalizing their inverse document frequencies between 1 and 10. So, the matching weight of each bytecode ranges between 1 and 10 with respect to the specificity of each bytecode.

Fig. 2a shows an example of the two behaviors a and b. This table shows sequences of bytecodes and the matching weights of the bytecodes. Fig. 2b shows the procedure for aligning two behaviors using semi-global alignment. The penalty values are applied with $\sigma = 2$ and $gap = 1$. The traces in the grid provide a method of computing the resulting value for aligning two behaviors by placing the matching point up to the position in each cell. Firstly, the first row and the first column are initialized with zeros and $-(i \times gap)$, respectively. Secondly, the matching point for each cell is calculated by applying the matching condition, such as match, mismatch, and gap, between each pair of instructions of two behaviors. For example, the pair of instructions is matched with `iload_1` in the cell labeled (a). So, the point of the cell is calculated by adding the weight of $w(\texttt{iload\_1}) = 2$ to the upper-left cell. Similarly, consider the cell labeled (b). The pair of instructions is mismatched with `iconst_1` and `iload_1`. The point of the cell is obtained by the maximum of gap-penalized value (from the left cell or the upper cell) and the mismatch-penalized value (from the upper-left cell). So, the value for the cell is obtained from the left cell by subtracting gap penalty 1. After calculating all cells of the alignment, the maximum point among the values in the bottom row is the resulting value for aligning two behaviors a and b. From the calculation, the value for aligning two behaviors is 7.

Definition 5. (*Matching Point of Two Behaviors*). Let a and b be behaviors in two birthmarks $\mathbf{bm}(P)$ and $\mathbf{bm}(Q)$, respectively. Let $Semi\text{-}Global(a,b)$ be the resulting value of semi-global alignment between behaviors a and b. The *matching point* of two behaviors a and b, denoted by $mp(a,b)$, is then defined as follows:

$$mp(a,b) = \max(Semi\text{-}Global(a,b), 0). \tag{8}$$

After finishing the semi-global alignment between two sequences, the resulting value may be negative if the sequences are highly dissimilar. In order to normalize the overall similarity between 0 and 1, the minimum matching points of behaviors are confined to 0.

4.5. Comparing birthmarks

Given that a behavior is the basic element of the FP birthmark, it is necessary to consider relationships between every pair of behaviors in two birthmarks. A relationship refers to the degree of similarity between two behaviors; therefore, it can be represented by the matching point of two behaviors. Suppose birthmarks $\mathbf{bm}(P)$ and $\mathbf{bm}(Q)$ have n and m behaviors, respectively. Then, $n \times m$ matching points should be considered between behaviors in the birthmarks. From all of the pairs of behaviors, it is possible to organize $n \times m$ matching matrix that is composed of the matching points. For example, if $\mathbf{bm}(P) = \{a_1, a_2, \ldots, a_n\}$ and $\mathbf{bm}(Q) = \{b_1, b_2, \ldots, b_m\}$ are birthmarks of two programs P and Q, the *matching matrix* of $\mathbf{bm}(P)$ and $\mathbf{bm}(Q)$, denoted by $Matrix(\mathbf{bm}(P), \mathbf{bm}(Q))$, is then organized as follows:

$$Matrix(\mathbf{bm}(P), \mathbf{bm}(Q)) = \begin{pmatrix} mp(a_1,b_1) & mp(a_1,b_2) & \ldots & mp(a_1,b_m) \\ mp(a_2,b_1) & mp(a_2,b_2) & \ldots & mp(a_2,b_m) \\ \vdots & \vdots & \ddots & \vdots \\ mp(a_n,b_1) & mp(a_n,b_2) & \ldots & mp(a_n,b_m) \end{pmatrix}.$$

From the matching matrix, the overall similarity can be calculated by finding similar pairs of behaviors from each birthmark. This is solved by searching for matched pairs in a manner that maximizes the sum of the matching points. This problem is reduced to a maximum weighted bipartite matching problem [18]. In other words, behaviors and matching points across behaviors correspond to the nodes of a bipartite graph and weight edges, respectively. A greedy algorithm can find the set of pairs in $O(n^3)$ by selecting pairs of behaviors in descending order of the matching point. These results give $\min(n,m)$ pairs of matched behaviors, and this set of pairs is termed the *matching set* between two birthmarks. Intuitively, the matching set represents the set of pairs of the most similar behaviors among all behaviors in two birthmarks.

Definition 6. (*Matching Point of Birthmarks*). Let $\mathbf{bm}(P)$ and $\mathbf{bm}(Q)$ be the birthmarks of two programs P and Q, respectively. The *matching point of birthmarks* $\mathbf{bm}(P)$ and $\mathbf{bm}(Q)$, denoted by $\mathscr{P}(\mathbf{bm}(P), \mathbf{bm}(Q))$, is then defined as follows:

$$\mathscr{P}(\mathbf{bm}(P), \mathbf{bm}(Q)) = \sum_{(a,b) \in \mathsf{M}(\mathbf{bm}(P), \mathbf{bm}(Q))} mp(a,b), \tag{9}$$

where $\mathsf{M}(\mathbf{bm}(P), \mathbf{bm}(Q))$ denotes the *matching set* of $\mathbf{bm}(P)$ and $\mathbf{bm}(Q)$.

The *matching point between two birthmarks* is calculated by accumulating all the matching points of the pairs in the matching set between the birthmarks. If the matching point between two birthmarks is higher than in other pairs, the two programs likely share many sequences of bytecodes in common. This implies that the two programs are likely to be in copy relation, because common sequences of bytecodes are typically used in such cases. From the matching point of birthmarks, the *similarity of birthmarks* is defined as follows:

Definition 7 (*Similarity of Birthmarks*). Let P and Q be programs, and $\mathbf{bm}(P)$ and $\mathbf{bm}(Q)$ be birthmarks of P and Q, respectively. The similarity of birthmarks $\mathbf{bm}(P)$ and $\mathbf{bm}(Q)$ is then defined as follows:

$$Similarity(\mathbf{bm}(P), \mathbf{bm}(Q)) = \frac{\mathscr{P}(\mathbf{bm}(P), \mathbf{bm}(Q))}{\min\left(\sum_{b \in \mathbf{bm}(P)} \mathsf{w}(b), \sum_{b \in \mathbf{bm}(Q)} \mathsf{w}(b)\right)}, \tag{10}$$

where $\mathsf{w}(b)$ denotes the sum of the weights of the behavior b; explicitly, for a behavior $b = \langle x_1, x_2, \ldots, x_n \rangle$, $\mathsf{w}(b) = \sum_{x \in b} w(x) = w(x_1) + w(x_2) + \cdots + w(x_n)$.

The similarity between two birthmarks is calculated based on the sum of the matching points of all the matched pairs. In Eq. (10), the similarity is normalized by dividing the sum by the minimum of sum of the weights of behaviors. Because software theft may augment bogus instructions or control flows, the minimum

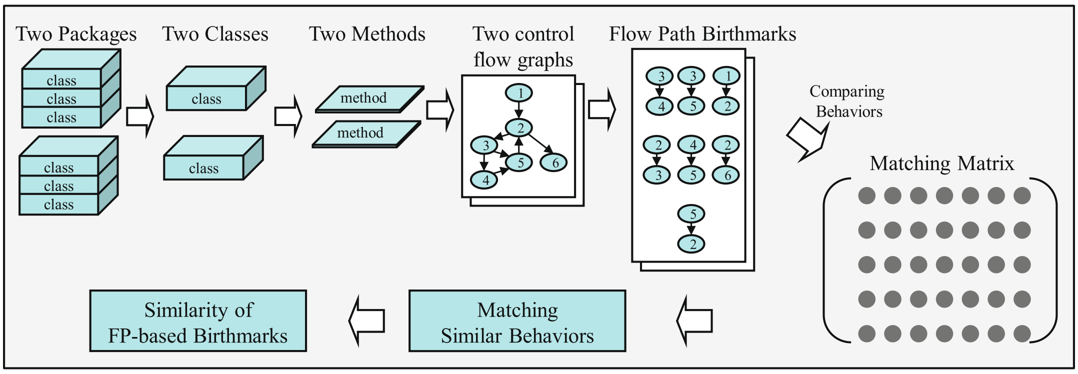


Fig. 3. The procedure for the FP-based birthmarking system.

of the two weight sums are applied so that the similarity can reflect the containment relationship between the programs. If two birthmarks are fully matched, the matching point between two birthmarks becomes equal to the denominator. Hence, the resulting similarity ranges between 0 and 1 in proportion to the degree of similarity. This measure represents the ratio of birthmarks of original program that are contained in that of the target program. If an author believes that his modules were stolen, it is a simple matter to compare the birthmarks of suspicious modules with the modules of the corresponding programs to confirm the theft.

For example, let $\mathbf{bm}(P) = \{a_1, a_2, a_3, a_4\}$ and $\mathbf{bm}(Q) = \{b_1, b_2, b_3, b_4\}$ be birthmarks of programs P and Q, respectively. Let the weight sums of behaviors in two birthmarks be $\sum_{a\in\mathbf{bm}(P)} \mathrm{w}(a) = 100$ and $\sum_{b\in\mathbf{bm}(Q)} \mathrm{w}(b) = 120$, respectively. Let

$$\begin{array}{c|cccc} & b_1 & b_2 & b_3 & b_4 \\ \hline a_1 & \mathbf{16} & 5 & 0 & 6 \\ a_2 & 3 & 0 & \mathbf{15} & 6 \\ a_3 & 2 & \mathbf{10} & 0 & 3 \\ a_4 & 0 & 2 & 3 & \mathbf{9} \end{array}$$

be the matching matrix of $\mathbf{bm}(P)$ and $\mathbf{bm}(Q)$. From the search algorithm for matching behaviors, the matching set is denoted in this case as the boldface points; that is, $\mathrm{M}(\mathbf{bm}(P), \mathbf{bm}(Q)) = \{(a_1, b_1), (a_2, b_3), (a_3, b_2), (a_4, b_4)\}$. Finally, the similarity between the birthmarks is calculated as follows:

$$Similarity(\mathbf{bm}(P), \mathbf{bm}(Q)) = \frac{16 + 15 + 10 + 9}{\min(100, 120)} = 0.50.$$

Hence, the similarity between the programs is 50%.

5. Experimental results

5.1. Preliminaries

In this section, the proposed birthmark is evaluated with respect to two properties required for birthmark: credibility and resilience. Fig. 3 shows a brief procedure for birthmarking two Java programs and calculating the similarity between the birthmarks. In the Java program packages, the FP birthmarks are extracted from original and suspicious methods or classes. Subsequently, comparing the behaviors leads to the matching matrix, and the similarity is calculated by matching similar behaviors from the matrix. The proposed birthmark was implemented in C language on MS Windows XP and then evaluated on a PC system with an Intel Pentium 4 (2.4 GHz) processor with 2 GB RAM.

The two Java program packages shown in Table 1 were used as benchmark programs. Each class file in the packages is regarded as an independent program. The class files are compared with each other to evaluate the detection capability of software birthmarks. The class files in the packages were chosen with respect to following criteria:

Size of the class files: Small programs may not be a target for software theft. In addition, although birthmarks may be similar, it is likely to be considered controversial to conclude that two programs are in copy relation, because small programs may be commonplaces. Therefore, to evaluate birthmarks properly, the class files that contain at least 100 bytecodes were chosen.
Contents of the class files: In a Java program package, the class files use many variables, and some class files are invoked only to initialize the variables before the genuine operation. This kind of class files consist of just one basic block containing a sequence of method invocations and variable initializations without containing control structures, such as loops, conditional, and unconditional branches. Because these class files are often used not for genuine operations, but for only initializations in most program packages, they are excluded from the evaluation.

As shown in Table 1, 177 and 86 class files were chosen out of 407 and 383, respectively. The average numbers of bytecodes for the evaluated class files in Ant and BCEL were 433.4 and 613.8, and the numbers fall between 101 and 7077. The control flow graphs consist of 87–92 edges and 66–90 nodes on average, ranging from 2 to 1271 and 9 to 906, respectively.

5.2. Birthmark evaluation

For the evaluation of a software birthmark, there are two properties that should be satisfied: credibility and resilience. In previous works [23,24,28,29,32,41–45], evaluations were performed in

Table 1
The specifications of benchmark programs.

Program	Size (bytes)	# of class files		Evaluated class files and the CFG		
		Total	Evaluated	# bytecodes	# edges	# nodes
Ant 1.5.4	736,810	407	177	433.4 (101–4162)	87.7 (3–762)	66.9 (9–592)
BCEL 5.2	533,339	383	86	613.8 (101–7077)	92.2 (2–1271)	90.8 (11–906)

independent environments in the evaluation of each property. For example, to evaluate the credibility, many of these studies compared different programs with each other, and measured how low the ranges of the similarity levels were. To evaluate the resilience, they compared the original programs with the modified versions and measured how high the ranges of the similarity levels were. These types of evaluation methods have a number of issues that should be considered.

- Previous evaluation methods only considered the average similarity between different programs or identical programs in separate environments. Thus, these methods did not consider false positives and false negatives, although these may exist in the evaluation of credibility and resilience, respectively. In real-world environments, however, the two properties should be satisfied simultaneously; hence, evaluation must be performed so that false positives and false negatives are taken into account.
- If birthmarking methods are different, the scales of similarity levels may also have different ranges. Therefore, it is unfair to evaluate the performance of birthmarks through a direct comparison of the average similarities of programs. In order to counterbalance the different scales of the similarity levels, it is necessary to normalize the scales according to individual birthmarks.

Precision and recall [9,20,39] are widely used measures for evaluating the quality of results in a binary decision problem, such as information retrieval and statistical classification. *Precision* is a measure used to evaluate the soundness of the results; it indicates the ratio of pairs of programs that are correctly detected to be in copy relation among all pairs of programs that are detected by the birthmark. Higher precision denotes that the pairs of programs detected by the birthmark are more likely to be in copy relations. Higher precision also shows the credibility of the birthmark. To increase precision, a software birthmark must reduce the occurrence of false positives. *Recall* refers to the ratio of pairs of programs that are detected by a software birthmark among pairs of programs that are actually in copy relation. Higher recall denotes that the pairs of programs in copy relation are more likely to be detected by the birthmark and correspondingly shows the resilience of the software birthmark. In order to increase the level of recall, a software birthmark must reduce the occurrence of false negatives, leading to high resilience. The two measures are calculated as follows:

$$Precision = \frac{TP}{TP+FP} = \frac{|\{\text{Pairs in copy relation}\} \cap \{\text{Detected pairs}\}|}{|\{\text{Detected pairs}\}|}, \tag{11}$$

$$Recall = \frac{TP}{TP+FN} = \frac{|\{\text{Pairs in copy relation}\} \cap \{\text{Detected pairs}\}|}{|\{\text{Pairs in copy relation}\}|}. \tag{12}$$

Here, TP, FP, and FN denote the number of true positives, false positives, and false negatives, respectively. To evaluate the credibility and resilience of a software birthmark simultaneously, the harmonic mean of precision and recall, F-measure, is considered as follows:

$$F = 2 \times \frac{Precision \times Recall}{Precision + Recall}. \tag{13}$$

The F-measure is located between 0 and 1 in proportion to the degree of precision and recall; the value remains high only in cases where both of the precision and the recall are high relative to each other.

As mentioned earlier, a birthmarking system detects pairs in copy relation on the basis of the threshold value ϵ. As the threshold value ϵ moves from 0 to 1, F-measure values draw a curve that represents the effectiveness of the birthmark. The area under the F-measure curve (AUC) is used as a metric to define how a software birthmark performs over the entire space of threshold ϵ. Intuitively, a higher AUC implies that the software birthmark can separate two groups: the pairs of programs in copy relation (T group) and the pairs of different programs (F group), by a wider margin, further implying that the birthmark can distinguish the pairs in copy relation with a higher degree of accuracy.

5.3. Comparing FP birthmark with different k values

The FP birthmark is based on k consecutive basic blocks reachable through control flow edges. As the value of k increases, the capability of the birthmark can differ. Therefore, for practical application, it is important to determine a feasible value of k that leads to higher credibility and resilience. In order to measure the property of the birthmark with respect to the value of k, birthmarks with k value ranging from 1 to 5 were evaluated. The penalty values were applied with $\sigma = 2$ and $gap = 1$. BCEL was used as a benchmark program, and four transformation methods were applied to modify the original program. Jikes [4] is a Java compiler that compiles Java source codes into Java class files. Jarg [2] is a Java bytecode optimizer that optimizes Java programs by renaming or eliminating the unnecessary parts of the programs. Smokescreen [5] and ZKM [3] are Java program obfuscators that transform original Java programs into equivalent programs that are more difficult to decompile or analyze. Smokescreen and ZKM perform several transformations, including control flow obfuscation, renaming, and string encryption.

Initially, the source codes of BCEL package were recompiled with the Jikes compiler. Alternatively, an original BCEL package was transformed with using Jarg, Smokescreen, or ZKM with the strongest respective level of modification. The birthmarks of the class files in the original package and their transformed versions were then extracted. Using the previously described criteria, 86 class files in the original package and their transformed versions were considered as independent programs representing original programs and their copied programs, respectively. Subsequently, the birthmarks between all pairs of the original programs and the modified versions were compared. Thus, 7396 comparisons were done for each transformation of class files between the original and the modified version, and 86 pairs out of all comparisons involved the pairs of identical class files (the T group).

Tables 2 and 3 show the results of the comparison of FP birthmarks with k values between 1 and 5. The experiments were performed on BCEL using the four different transformation methods of Jikes, Jarg, Smokescreen, and ZKM. Table 2 shows the information on FPs which were extracted from the original class files when $k = 1, 2, 3, 4$, and 5. The T group and the F group denote the pairs of identical and different class files, respectively. The numbers of FPs show the average numbers of extracted FPs and the ranges according to k. From the results, as the value of k increases, the average number of FPs and the lengths also increase. However, in some class files, the k-FP birthmark is no more extracted if their CFGs have no additional sequence of basic blocks longer than k. Ta-

Table 2
Information on FPs extracted from original BCEL with respect to k values.

	# of comparisons			# of flow paths		
	Total	T group	F group	Total	Avg. (range)	Bytecodes/FP
$k = 1$	7396	86	7310	7809	90.8 (11–906)	6.7 (1–456)
$k = 2$	7396	86	7310	7934	92.2 (2–1271)	14.2 (2–336)
$k = 3$	7396	86	7310	9044	105.1 (0–1427)	20.9 (3–352)
$k = 4$	7396	86	7310	10689	124.2 (0–1604)	27.2 (5–358)
$k = 5$	7396	86	7310	13493	156.8 (0–1908)	33.4 (7–359)

Table 3
Comparison of the time overheads and the *F*-measure analysis with different *k* values.

	Jikes		Jarg		Smokescreen		ZKM	
	Time (s)	AUC	Time (s)	AUC	Time (s)	AUC	Time (s)	AUC
$k = 1$	0.24 (0.0–22.9)	0.66	0.24 (0.0–22.7)	0.67	0.29 (0.0–27.5)	0.63	0.29 (0.0–24.3)	0.66
$k = 2$	0.27 (0.0–45.7)	0.69	0.27 (0.0–47.0)	0.69	0.32 (0.0–47.0)	0.65	0.36 (0.0–49.6)	0.63
$k = 3$	0.39 (0.0–65.2)	0.69	0.38 (0.0–67.3)	0.72	0.47 (0.0–69.8)	0.67	0.57 (0.0–73.0)	0.63
$k = 4$	0.61 (0.0–87.1)	0.71	0.61 (0.0–88.2)	0.73	0.76 (0.0–96.0)	0.67	1.03 (0.0–100.3)	0.61
$k = 5$	1.13 (0.0–124.7)	0.70	1.19 (0.0–126.0)	0.73	1.47 (0.0–138.9)	0.67	2.21 (0.0–149.0)	0.61

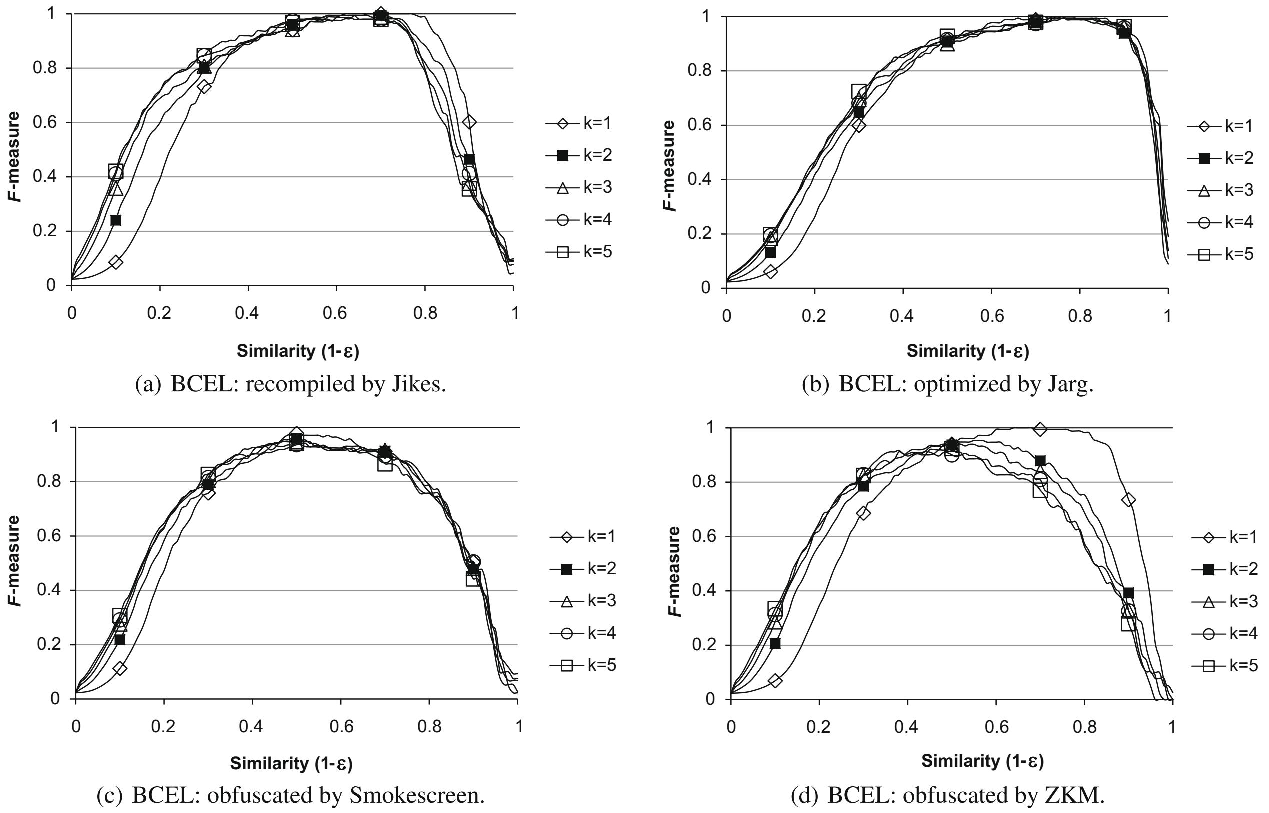
ble 3 shows the results of the comparison of the performances of the birthmarks according to *k* for the four transformations of Jikes, Jarg, Smokescreen, and ZKM. In the table, Time denotes the average times taken to compare a pair of programs and the ranges. Because the numbers and lengths of FPs grow as the value of *k* increases, the comparison times also increase. In evaluating the AUC of *F*-measure curve, the birthmark can be evaluated as effective when AUC is higher than 0.50 and the AUCs of the experiments range between 0.61 and 0.73. On the whole, there are no significant differences in AUCs with respect to the value of *k*.

Fig. 4 shows the *F*-measure curves of the results shown in Table 3. The horizontal axis represents the variation of the threshold value ϵ, which is the basis for determining whether or not two programs are in copy relation. The vertical axis represents the *F*-measure value according to varying ϵ values. With respect to the four transformation methods, the variations caused by the value of *k* are similar without apparent weaknesses. As the value of *k* increases, the overall similarities between the programs decrease slightly. As a result, the *F*-measure curves move slightly leftward, while the AUCs increase to some degree. In considering the AUCs and the distributions of *F*-measure curves, it was determined that the value of $k = 2$ is a viable compromise between the time complexity and the performance of the birthmark.

5.4. Comparing with other static approaches

There are several static approaches for detecting cases of software theft. Good examples are the birthmark of Tamada et al. [43,44], the *k*-gram based birthmark [29,32], and the stack pattern based birthmark [23]. The FP birthmark was evaluated and compared with these three approaches. The FP birthmark was experimented with $k = 2$ as determined in the previous section, and the penalty values for semi-global alignment were applied with $\sigma = 2$ and $gap = 1$. Stigmata 1.1 [6] was used for the birthmark of Tamada et al. and the *k*-gram based birthmark.

A person adept at 'cracking' software may modify or transform an original program to hide the fact of software theft. Therefore, a birthmark must be sufficiently resilient to aggressive program modifications. In order to evaluate four approaches in such environments, combinations of several transformation methods were used. The first modified version (JSJ) was transformed in such a way that the programs were consecutively transformed by Smokescreen and Jarg after recompiling their source codes using the Jikes compiler. The second modified version (JZJ) used the ZKM obfuscator instead of Smokescreen. Ant and BCEL were used as benchmark programs, and the experiment referred to in Section 5.3 was conducted with these benchmark programs and their modified versions JSJ and JZJ.



(a) BCEL: recompiled by Jikes.

(b) BCEL: optimized by Jarg.

(c) BCEL: obfuscated by Smokescreen.

(d) BCEL: obfuscated by ZKM.

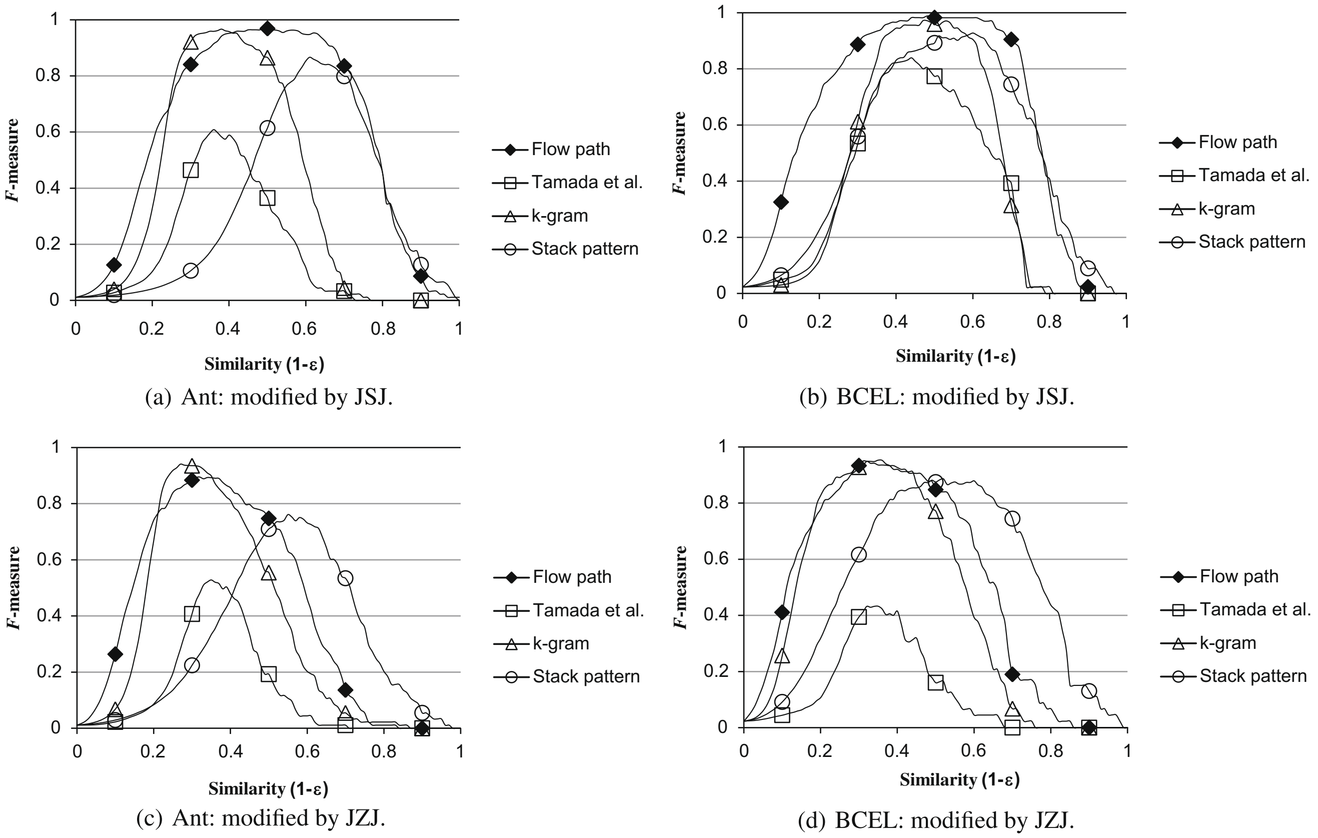
Fig. 4. Comparison of *F*-measure curves of birthmarks with different *k* values.

Table 4
Results of comparisons of FP birthmark with the previous birthmarks.

	Ant			BCEL		
	AUC	Similarity (%)		AUC	Similarity (%)	
		T group	*F* group		*T* group	*F* group
(a) Evaluation with the modification of JSJ						
Flow path	0.57	74.7 (41.8–100)	3.9 (0.0–66.6)	0.61	74.7 (41.1–90.1)	3.0 (0.0–47.1)
Tamada et al.	0.16	39.8 (12.5–72.2)	10.2 (0.0–51.1)	0.33	56.8 (23.8–78.5)	11.0 (0.0–57.0)
k-gram	0.36	55.5 (34.4–76.9)	7.3 (0.0–47.3)	0.39	64.8 (48.8–80.5)	14.3 (0.0–57.0)
Stack pattern	0.32	74.0 (40.0–99.1)	14.5 (0.0–78.2)	0.47	72.5 (41.6–96.8)	8.8 (0.0–64.3)
(b) Evaluation with the modification of JZJ						
Flow path	0.41	51.4 (9.2–86.7)	2.7 (0.0–73.7)	0.49	57.8 (20.0–85.3)	2.4 (0.0–52.8)
Tamada et al.	0.12	36.5 (14.5–70.8)	11.2 (0.0–52.4)	0.12	34.5 (8.9–67.6)	11.5 (0.0–67.7)
k-gram	0.33	46.0 (5.0–83.7)	5.6 (0.0–45.2)	0.42	52.7 (25.6–75.6)	4.6 (0.0–42.1)
Stack pattern	0.30	63.4 (16.8–97.2)	9.9 (0.0–75.4)	0.48	70.4 (35.5–98.6)	7.3 (0.0–67.5)

Table 4 shows the results of the experiments on the four different birthmarks. The heading AUC shows the results of the *F*-measure curve analysis, while Similarity for each birthmark shows the average similarities and their ranges according to the *T* group and the *F* group. In a comparison of the *T* group and the *F* group, to distinguish programs in copy relation more accurately, higher differences can be evaluated as better. Fig. 5 shows comparisons of the *F*-measure curves of the four approaches. With the birthmark of Tamada et al., the differences in the average similarities between the *T* group and the *F* group ranged from 23.0% to 45.8%. The *F*-measure curves were located at the lowest areas among four methods. The AUCs ranged from 0.12 to 0.33. The birthmarks are dependent on comparisons of names of classes or methods in programs. For example, used classes, inheritance structures, and sequence of method calls directly compare sequences of names of classes or methods. In genuine class files that are not processed by obfuscators or optimizers, the birthmarks can be used with high reliability. However, through control flow obfuscation, the order of method calls may be easily modified. Moreover, most obfuscators and optimizers can reorganize class structures and modify class names. These types of aggressive transformations impaired the birthmark of Tamada et al. much more than the others.

With the *k*-gram based birthmark, the differences in the average similarities between the *T* group and the *F* group ranged from 40.4% to 50.5%. The *F*-measure curves were located in areas higher than those found in Tamada et al. The maximum points of the *F*-measure curves were located at high positions comparable to those of the FP birthmark. However, the widths of the curves were narrower. The fact that the maximum point of *F*-measure curve was higher than those of others indicates that the birthmark distinguished the pairs of programs in copy relation more precisely at that point. However, the maximum points can differ according to the individual modification environments. Therefore, the maximum points in the experiments do not always comply with real-



(a) Ant: modified by JSJ.

(b) BCEL: modified by JSJ.

(c) Ant: modified by JZJ.

(d) BCEL: modified by JZJ.

Fig. 5. Comparison of *F*-measure curves of FP birthmark with those of the previous birthmarks.

world environments. A birthmark can be evaluated to be more credible and resilient if its *F*-measure curve has the form of a wide and even distribution that yields a high AUC value. The AUCs ranged from 0.33 to 0.42, and the areas were smaller compared to those of the FP birthmark. The *k*-gram based birthmark uses *k* consecutive opcodes in programs. Hence, it is susceptible to control flow obfuscation and code reordering. However, the *k*-gram based birthmark is considered to be more credible than the birthmark of Tamada et al., as it compares the opcodes itself instead of superficial characteristics such as class names or method names.

With the stack pattern based birthmark, the differences in the average similarities between the *T* group and the *F* group ranged from 53.5% to 63.7%. The *F*-measure curves were located in the more right regions compared to the prior two approaches. This implies that the stack pattern based birthmark was more resilient even after programs were modified by the transformation strategies of JSJ or JZJ. However, the widths and the heights of the *F*-measure curves were lower compared to the high resilience. This result shows that different programs may not be discriminated with credible intervals. This is because similar stack patterns that are composed of frequent stack instructions hinder the birthmark from distinguishing different programs. The AUCs ranged from 0.30 to 0.48, and the performances undulated according to environments where the birthmark was evaluated. For example, in environments with BCEL modified by JZJ, the AUC was almost equal to that of the FP birthmark. However, in the other environments, the AUCs of the stack pattern based birthmark were no more than those of the *k*-gram based birthmark. In our evaluation, this is because the stack pattern based birthmark does not consider control flows of programs. When the control flows of programs are extensively modified by transformations, this method is affected like the *k*-gram based birthmark.

With the FP birthmark, the differences in the average similarities between the *T* group and the *F* group ranged from 48.7% to 71.7%. In comparison with the other approaches, the *T* and *F* groups were located in a considerable distance from each other. The *F*-measure curves were distributed in the wider areas, and the AUCs ranged from 0.41 to 0.61, which were the highest values among four approaches. Regardless of evaluation environments, the FP birthmark showed more uniform performances compared to the other birthmarks. These results are strong evidence that the proposed birthmark is more effective and reliable than other birthmarks in detecting programs in copy relation.

For practical use, it is important to decrease the number of false negatives. This pertains mainly to the pairs in the *T* group, but the similarities are lower than the threshold value. For such cases, their bytecodes were investigated.

- First, it was noted that Smokescreen or ZKM decomposed a basic block into several parts by adding bogus instructions with the help of opaque predicates [8,15] which were always evaluated as either `true` or `false`. When behaviors between two birthmarks are matched, the behaviors in the decomposed parts of the CFG cannot be fully matched to the behaviors in the CFG of the original program. This issue is discussed in Section 6.1.
- Secondly, it was noted that program transformation changed the sequence of bytecodes by means of instruction reordering. For example, Smokescreen transforms the sequence of pairs of `load` and `store` instructions into the sequence of `store` instructions after `load` instructions. As the FP birthmark compares the sequences of bytecodes, the birthmarking method may be ineffective if the bytecode sequence has been reordered.
- Thirdly, it was noted that obfuscation changed a number of patterns of bytecode sequences into explanatory expressions. Semi-global alignment is an effective method of aligning sequences in which heads or tails are modified or augmented. However, if the inner parts of the sequences are modified, semi-global alignment is fragile, because the modified parts may yield penalties for aligning the mismatched parts. For example, the bytecode `ldc` can be changed to the sequence of `getstatic`, `iconst_`*n*, `aaload` or `getstatic`, `bipush`, `aaload`. If the modification occurs in the middle of the bytecode sequences, semi-global alignment can fail to align the two sequences on account of penalty values for aligning the mismatched bytecodes. In order to overcome this defect, it is necessary to refine the alignment algorithm by matching the patterns that can be transformed by obfuscators.
- Finally, it was noted that different compilers generated bytecode sequences differently. For example, the Jikes compiler changed the order of flow directions by inverting the condition of conditional branches.

These problems occurred due to the use of a method that compared the sequence of opcodes directly.

6. Discussion

6.1. Maintaining basic block decomposition

The FP birthmark is based on the sequences of basic blocks. For example, if *k*-FP is considered as a birthmark, the sequences of *k* consecutive basic blocks reachable through control flow edges are used for birthmarking programs. Therefore, if the basic blocks of CFG are not altered, the FP birthmark shows very reliable results. However, if program modification changes the structure of the basic blocks, e.g., by breaking a basic block into several parts or combining several basic blocks into one basic block, the method may be affected detrimentally and output unreliable results.

In cases in which several basic blocks are combined into one basic block, their flow-related basic blocks should be flattened out to compose the basic blocks. Thus, the number of nodes in the CFG and the code size increase. In this case, because the FPs generated from the modified basic blocks overlaps with the counterparts of the original basic blocks, the birthmark of the original program is dominated by that of the modified program, which leads to a high degree of similarity between the birthmarks. However, if one basic block is decomposed into several parts, it is more difficult to detect the software theft. The FPs generated from the decomposed basic blocks may not cover the counterparts of their original basic blocks, because decomposing one basic block reduces the span of the FP related to the decomposed basic block. This mismatch leads to a low degree of similarity, as described in Section 5.4. To alleviate this shortcoming, it is necessary to increase the span of the FPs related to the modified basic blocks. For this purpose, extending the value of *k* into $k + \alpha$ for some lookahead value α can be considered. In other words, the problem can be mitigated by comparing the *k*-FP birthmark of original program with the $(k + \alpha)$-FP birthmark of suspected program.

6.2. Comparing with a dynamic approach

Dynamic software birthmarking is an approach that extracts birthmarks from run-time behaviors as the program is executed. Dynamic approaches are dependent on run-time environments, such as the inputs, user interactions, or run-time traces. So, they cannot characterize the overall aspects of programs; rather, they characterize only a slice of the program execution. When investigating the theft of an entire program, detection may be accomplished easily by summarizing the run-time behaviors extracted from all of the traces. On the other hand, when investigating the theft of libraries or modules, the detection is more complicated because libraries or modules typically are partly contained in an entire program. The concerned parts may be embedded as some parts of the birthmark of entire program. If the suspicious modules are much smaller as compared to their complete program, it may be difficult to extract the birthmark of the concerned parts, because the run-time trace of the program may not be controllable. Moreover, it is possible that the birthmark does not reflect the affected parts at all. Thus, to detect the theft of modules through dynamic approaches, it is essential for the author to grasp the details of the stolen modules.

In experiments on a dynamic birthmark by Schuler et al. [42], several XML parsers were compared with each other. To parse XML documents, XML parsers commonly use the SAX interface, a part of the Java API. Although XML parsers share the SAX interface in common, the degree of similarities between these programs is not high because the portions of the shared modules are much smaller compared to the size of the complete programs. For this reason, to evaluate detection capability of library theft, their experiments involved comparing the birthmarks of the library themselves with the birthmarks of the programs that contain the library. Because library modules generally do not run as a stand-alone program, the main routine requires invoking the library to extract the birthmark from the modules. For this purpose, the author should know precisely which library modules were stolen; moreover, it is important to ensure that the run-time environment of each instance is identical. This circumstance requires additional efforts to detect library theft when using dynamic approaches. Moreover, in cases in which a program is obfuscated, it may not be feasible to manage the run-time environments. Therefore, dynamic approaches have several limitations that prevent them from being widely applied in real-world environments.

Static software birthmarking is performed by investigating the program itself. In cases in which libraries or modules are stolen in other programs, this type of birthmark can freely explore the suspected parts of the programs and compare the birthmarks. For example, in experiments by the authors that compared the XML parsers [24], most of the shared modules in Piccolo and Crimson had higher levels of similarity, indicating that they were in copy relation, although the two programs shared different versions of the relevant libraries.

6.3. Considering FPs with larger value of *k*

The FP birthmark is performed by comparing sequences of reachable basic blocks in the CFGs of programs. With the large value of *k*, the precision of the FP birthmark can be improved, however, the recall of the birthmark deteriorates. For more accurate results of this approach, it is important to maintain the number of FPs and their lengths effectively. As the value of *k* increases, the comparison time increases exponentially, because the number of covered basic blocks conforms to the *k*. When the value of *k* increases to be very large, two problems may arise.

- For simple programs without loops, the number of FPs drops to zero when there is no more basic block to traverse.
- For complex programs (especially with sequences of conditional branches), the number of FPs grows exponentially as the *k* increases.

The first problem may be solved by using full sequences from the beginning to the end of programs although the length is less than *k*. This is similar to the approach of the full trace birthmark [33]. However, the second problem is much more challenging. Because the exponential increase of FPs is mainly caused by serial composition of conditional branches, the explosion problem is unavoidable if the value of *k* is very large. However, restricting the iteration number of loop structures may help to alleviate the problem.

For loop structure of a program, two aspects of the FP birthmark conflict with each other. Loop structure is an important characteristic of programs because programs can be differentiated according to the existence of loop structures. However, the iteration of loops may cause excessive increase of the number of FPs and the lengths as the value of *k* increases. If the value of *k* is reasonably small, it may be more favorable not to restrict the number of loop iteration for maintaining the identity of programs. On the other hand, if the value of *k* is much larger than some acceptable value, the number of loop iteration should be limited to a fixed number to reduce the repetition of duplicated sequences. Although restriction on the number of loop iteration cannot clear up the explosion problem, it may reduce the number of FPs considerably.

In our prior evaluation in Section 5.3, there was no satisfactory improvement on the AUCs although the value of *k* increases. However, the comparison time increased noticeably. Therefore, for practical approaches for detecting software thefts, it is favorable to maintain the value of *k* as small as possible.

7. Conclusion and future work

Software is intellectual property that must be protected against theft. However, the incidences of software theft increase every year. To detect cases of software theft, there have been several investigations of software birthmarks. A software birthmark can be used to identify the origin of software by comparing the inherent characteristics of the program. This paper proposes a method of detecting the theft of Java programs through analysis of the flow paths of the program, specifically of the FP birthmark. Flow paths refer to the sequences of basic blocks that are connected by the edges of the CFG of a program; they represent the possible flows of operations upon execution of the program. They are also known as the behaviors of the program. The semi-global alignment was used to align two behaviors in each program, and the similarity between two programs was calculated by determining the set of the most similar pairs of behaviors in the two programs.

To measure the two properties of credibility and resilience, the proposed birthmark was evaluated with respect to precision and recall, and the *F*-measure curves were then analyzed. The proposed method was also compared with the earlier approaches of Tamada et al., the *k*-gram based birthmark, and the stack pattern based birthmark. Experimental results showed that the proposed birthmark was a more effective measure for detecting programs in copy relation, even in cases where programs had been aggressively modified. In addition, it is decisively shown here that the proposed method can lessen the time and effort required for manual reverse engineering in the identification of software theft. Because the FP birthmark can be applied to the algorithmic structures of a program, this method can also be used to detect the theft of the libraries or modules used in a complete program. The proposed method may be ineffective if the basic blocks have been decomposed or if the instructions in programs are reordered or changed via modifications. In a future work, the authors plan to improve this method by refining the alignment algorithm and by analyzing the control flows of programs.

References

[1] The GPL-violations.org project, <http://gpl-violations.org/>.

[2] Java archive grinder (Jarg), <http://sourceforge.net/projects/jarg/>.

[3] Java obfuscator—zelix klassmaster, <http://www.zelix.com/klassmaster/index.html>.

[4] Jikes Java compiler, <http://jikes.sourceforge.net/>.

[5] Smokescreen Java obfuscator, <http://www.leesw.com/smokescreen/>.

[6] Stigmata—Java birthmark toolkit, <http://stigmata.sourceforge.jp/>.

[7] Business Software Alliance, Fifth Annual BSA and IDC Global Software Piracy Study, 2007.

[8] G. Arboit, A method for watermarking Java programs via opaque predicates, in: The Fifth International Conference on Electronic Commerce Research (ICECR-5), 2002.

[9] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, Addison Wesley, 1999.

[10] K.W. Bowyer, L.O. Hall, Experience using "moss" to detect cheating on programming assignments, in: The 29th Annual Frontiers in Education Conference, vol. 3, 1999.

[11] M. Brudno, S. Malde, A. Poliakov, C.B. Do, O. Couronne, I. Dubchak, S. Batzoglou, Glocal alignment: finding rearrangements during alignment, Bioinformatics 19 (Suppl. 1) (2003) 54–62.

[12] S. Burrows, S.M.M. Tahaghoghi, J. Zobel, Efficient plagiarism detection for large code repositories, Software Pract. Exper. 37 (2) (2007) 151–175.

[13] C. Collberg, C. Thomborson, Software watermarking: models and dynamic embeddings, in: Principles of Programming Languages 1999, POPL'99, San Antonio, TX, 1999.

[14] C. Collberg, C. Thomborson, Watermarking, tamper-proofing, and obfuscation—tools for software protection, IEEE Trans. Software Eng. 28 (11) (2002) 735–746.

[15] C. Collberg, C. Thomborson, D. Low, Manufacturing cheap, resilient, and stealthy opaque constructs, in: Principles of Programming Languages 1998, POPL'98, San Diego, CA, 1998.

[16] C.S. Collberg, E. Carter, S.K. Debray, A. Huntwork, J.D. Kececioglu, C. Linn, M. Stepp, Dynamic path-based software watermarking, in: Conference on Programming Language Design and Implementation, 2004.

[17] C.S. Collberg, C. Thomborson, G.M. Townsend, Dynamic graph-based software fingerprinting, ACM Trans. Program. Lang. Syst. 29 (6) (2007) 35.

[18] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, MIT Press and McGraw-Hill, 2001.

[19] R.L. Davidson, N. Myhrvold, Method and System for Generating and Auditing a Signature for a Computer Program, US Patent 5,664,191.

[20] J. Davis, M. Goadrich, The relationship between precision-recall and roc curves, in: ICML'06: Proceedings of the 23rd International Conference on Machine Learning, ACM, 2006.

[21] D. Gitchell, N. Tran, Sim: a utility for detecting similarity in computer programs, in: Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education, 1999.

[22] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, second ed., Addison Wesley, 2000.

[23] H. Lim, H. Park, S. Choi, T. Han, Detecting theft of Java applications via a static birthmark based on weighted stack patterns, IEICE Trans. Inform. Syst. E91-D (9) (2008) 2323–2332.

[24] H. Lim, H. Park, S. Choi, T. Han, A Static Java Birthmark based on Control Flow Edges, Technical Report CS-TR-2008-292, Department of Computer Science, KAIST.

[25] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, second ed., Addison Wesley, 1999.

[26] A. Monden, H. Iida, K. Matsumoto, K. Inoue, K. Torii, A practical method for watermarking Java programs, in: The 24th Computer Software and Applications Conference (COMPSAC 2000), 2000.

[27] G. Myles, C. Collberg, Software watermarking through register allocation: implementation, analysis, and attack, in: Sixth International Conference on Information Security and Cryptology (ICISC), Lecture Notes in Computer Science, vol. 2971, Springer, Berlin/Heidelberg, 2003.

[28] G. Myles, C. Collberg, Detecting software theft via whole program path birthmarks, in: Information Security, seventh International Conference (ISC 2004), LNCS 3225, 2004.

[29] G. Myles, C. Collberg, *k*-gram based software birthmarks, in: Proceedings of the 2005 ACM Symposium on Applied Computing, 2005.

[30] G. Myles, C. Collberg, Software watermarking via opaque predicates: implementation, analysis, and attacks, Electron. Commer. Res. 6 (2) (2006) 155–171.

[31] G. Myles, H. Jin, Self-validating branch-based software watermarking, in: 7th International Workshop on Information Hiding, Lecture Notes in Computer Science, vol. 3727, Springer, Berlin/Heidelberg, 2005.

[32] G.M. Myles, Software Theft Detection Through Program Identification, Ph.D. Thesis, Department of Computer Science, The University of Arizona, 2006.

[33] H. Park, S. Choi, H. Lim, T. Han, Detecting code theft via a static instruction trace birthmark for Java methods, in: Sixth IEEE International Conference on Industrial Informatics (INDIN 2008), 2008.

[34] H. Park, S. Choi, H. Lim, T. Han, Detecting Java theft based on static API trace birthmark, in: International Workshop on Security (IWSEC), Lecture Notes in Computer Science, vol. 5312, Springer, 2008.

[35] L. Prechelt, G. Malpohl, M. Philippsen, Finding plagiarisms among a set of programs with JPlag, Journal of Universal Computer Science 8 (11) (1999) 1016–1038.

[36] L. Prechelt, G. Malpohl, M. Philippsen, JPlag: Finding Plagiarisms Among a Set of Programs, Technical Report 2000-1, Universität Karlsruhe, Fakultät für Informatik, Germany, 2000, p. 44.

[37] G. Qu, M. Potkonjak, Hiding signatures in graph coloring solutions, in: IH'99: Proceedings of the Third International Workshop on Information Hiding, Springer-Verlag, London, UK, 2000.

[38] S. Robertson, Understanding inverse document frequency: on theoretical arguments for idf, J. Doc. 5 (2004) 503–520.

[39] G. Salton, M.J. Mcgill, Introduction to Modern Information Retrieval, McGraw-Hill Inc., 1986.

[40] S. Schleimer, D. Wilkerson, A. Aiken, Winnowing: local algorithms for document fingerprinting, in: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, 2003.

[41] D. Schuler, V. Dallmeier, Detecting software theft with API call sequence sets, in: Workshop Software Reengineering (WSR 2006), 2006.

[42] D. Schuler, V. Dallmeier, C. Lindig, A dynamic birthmark for Java, in: ASE'07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, 2007.

[43] H. Tamada, M. Nakamura, A. Monden, K. Matsumoto, Design and evaluation of birthmarks for detecting theft of Java programs, in: Proceedings of the IASTED International Conference on Software Engineering (IASTEDSE2004), 2004.

[44] H. Tamada, M. Nakamura, A. Monden, K. Matsumoto, Java birthmark – detecting the software theft, IEICE Trans. Inform. Syst. E88-D (9) (2005) 2148–2158.

[45] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, K. ichi Matsumoto, Dynamic software birthmarks to detect the theft of windows applications, in: Proceeding of the International Symposium on Future Software Technology (ISFST 2004), 2004.

[46] R. Wilhelm, D. Maurer, Compiler Design, Addison Wesley, 1995.

[47] M.J. Wise, String similarity via greedy string tiling and running karp-rabin matching, December 1993.

[48] M.J. Wise, Yap3: improved detection of similarities in computer program and other texts, in: Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education, 1996.

[49] J. Zhao, Analyzing control flow in Java bytecode, in: Proceedings of the 16th Conference of Japan Society for Software Science and Technology, 1999.