Detecting Theft of Java Applications via a Static Birthmark Based on Weighted Stack Patterns*

Hyun-il LIM†, Student Member, Heewan PARK†, Seokwoo CHOI†, and Taisook HAN†, Nonmembers

SUMMARY  A software birthmark means the inherent characteristics of a program that can be used to identify the program. A comparison of such birthmarks facilitates the detection of software theft. In this paper, we propose a static Java birthmark based on a set of stack patterns, which reflect the characteristics of Java applications. A stack pattern denotes a sequence of bytecodes that share their operands through the operand stack. A weight scheme is used to balance the influence of each bytecode in a comparison of the birthmarks. We evaluate the proposed birthmark with respect to two properties required for a birthmark: credibility and resilience. The empirical results show that the proposed birthmark is highly credible and resilient to program transformation. We also compare the proposed birthmark with existing birthmarks, such as that of Tamada et al. and the k-gram birthmark. The experimental results show that the proposed birthmark is more stable than the birthmarks in terms of resilience to program transformation. Thus, the proposed birthmark can provide more reliable evidence of software theft when the software is modified by someone other than the author.

key words: software birthmark, software theft detection, software protection, Java bytecode

1. Introduction

Software is an intellectual property of developers and it is protected by copyright law. However, cases of software theft are increasing every year [1]. Because software theft causes many problems to the software industry as well as to companies or authors, the license must be protected from illegal tampering. Hence, it is necessary to develop technology for verifying the originality of software.

A software birthmark, which was first introduced by Grover [2], refers to program’s inherent characteristics that can be used to identify the program. If two programs have the same or similar birthmarks, one is likely to be an illicit copy of the other. For example, comparing the strings in a program can be a naive birthmarking technique. For this purpose, several properties are needed for the birthmark. A birthmark should clearly discriminate different programs; that is, it should not say that a program is copied from another if it isn’t. Furthermore, a birthmark should be resilient to any semantics-preserving transformation (such as optimization or obfuscation), which may be applied to hide the fact of software theft.

Currently, there are various ways of detecting software theft. In this paper, we present and evaluate a specific technique: namely, a static Java birthmark based on weighted stack patterns. One program characteristic that is used as a birthmark of a Java program is a set of possible stack patterns that may form during the execution of the program. A stack pattern denotes a sequence of bytecodes that share their operands through the operand stack. We statically identify the stack patterns by analyzing the Java bytecodes stored in a class file. The similarity between two class files is calculated by matching the set of stack patterns and weight values which balance the effect of each bytecode.

Using several real-world Java applications, we evaluate the proposed birthmark with respect to two properties required for the birthmark: credibility and resilience. We also evaluate and compare the proposed birthmark with existing birthmarks, such as that of Tamada et al. [5], [6] and the k-gram birthmark [7]. The empirical results show that the similarities obtained from the proposed birthmark are credible and resilient enough to identify originality of software. From the comparison, we can also confirm that our birthmark is more stable than the previous birthmarks in terms of resilience to program transformation.

The remainder of this paper is organized as follows: In Sect. 2, we review existing approaches to software identification and birthmarks. In Sect. 3, we describe the formal definition of a software birthmark and propose a static Java birthmark based on weighted stack patterns. In Sect. 4, we describe experimental data and evaluate the proposed birthmark. In Sect. 5, we discuss various birthmark issues. Finally, in Sect. 6, we present our conclusion.

2. Related Works

Software watermarking [8], [9] can be a good choice for detecting software theft. Because software watermarking requires the embedding of a watermark to verify the originality of software, it can be applied only to programs that are already watermarked. A software birthmark, however, relies solely on the inherent characteristics of software instead of a previously embedded identifier. A limitation of the software birthmark is that it cannot be used to prove the authorship of a program; rather, it indicates whether one program is likely to be a copy of another. However, it can also be used in instances where watermarking is not feasible.
Tamada et al. [5], [6] were the first approach to suggest a practical application of static software birthmarks for Java class files. Their technique consists of four individual birthmarks: constant values in field variables, sequence of method calls, inheritance structure, and used classes. These four birthmarks can be used individually but they are more reliable when combined. Because the birthmarks are based on the underlying structures of a program, their usage is inappropriate when a program is partly contained in another program.

Myles et al. [7] proposed the $k$-gram birthmark, which is a static birthmark based on instruction sequences. A $k$-gram means a sequence of $k$ contiguous opcodes in a program and a set of $k$-grams is used as a software birthmark of the program. The $k$-gram birthmark is credible but highly susceptible to program transformation, such as optimization or obfuscation.

Tamada et al. [10] introduced the definition of a dynamic birthmark and proposed two such birthmarks based on traces of system calls for Windows programs. They proposed the use of sequences and the frequencies of API function calls during program execution as software birthmarks. Schuler and Dallmeier [12] presented similar approaches in Java applications. They used sets of API call sequences during program execution. These birthmarks are reasonably robust against program transformation. However, credibility of these birthmarks relies heavily on user interactions, inputs, and system environments. To alleviate this limitation, they restricted inputs and user interactions in their experiment.

Myles et al. [11] proposed another dynamic birthmark: namely, the whole program path (WPP) birthmark. A WPP can be obtained by using instrumentation to get a dynamic trace of a program and the trace is compressed into a directed acyclic graph using the SEQUITUR algorithm. The WPP is used as a birthmark, and two birthmarks are compared by using the graph distance for a maximal common subgraph.

3. A Static Birthmark Based on Weighted Stack Patterns

3.1 Software Birthmark

Tamada et al. [5], [6] formally defined a software birthmark in terms of a copy relation. The following definition and properties are restatements from [5]–[7].

**Definition 1** (Static Birthmark): Let $p$, $q$ be programs. Let $f$ be a method for extracting a set of characteristics from a program. Then $f(p)$ is called a birthmark of $p$ iff:

1. $f(p)$ is obtained only from $p$ itself (without any extra information), and
2. $q$ is a copy of $p$ $\Rightarrow f(p) = f(q)$.

**Property 1** (Credibility): Let $p$ and $q$ be independently written programs which 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 specifies that a birthmark of $p$ must remain in its original form even though a program transformation changes the structure of the program: that is, a software birthmark must be strong enough to endure semantics-preserving transformation.

3.2 Stack Patterns in the Java Bytecode

The characteristics of a birthmark must be strong enough to endure an attack from a cracker who wants to break the birthmark. The Java bytecodes use the operand stack as a workspace, and the bytecodes share the operand with each other through the operand stack. Because the independence of the bytecodes must be retained to preserve the semantics, a good way of designing a birthmark is to use a sequence of bytecodes, which share their operands through the operand stack. Furthermore, because the specification of the Java bytecode is rigorously defined [3], its operand stack behavior can be determined by static analysis. In other words, every bytecode in a program has its own unique stack status during a runtime execution and the status can be calculated at static time. Hence, the stack status of each bytecode in a class file means the stack depth after the bytecode has been executed.

Table 1 shows the classification table of the Java bytecodes in relation to their operand stack behaviors. An act, which represents the behavior of a bytecode, summarizes the stack depth variation after the bytecode has been executed. Generally, for an instruction in the NORMAL category, the instruction has a fixed stack behavior in relation to its own opcode.

For an instruction in the BRANCH category, the stack behavior of the bytecode is also determined by its own opcode. However, because this instruction changes the control flow of the program, the stack status must be forwarded to the target instructions. For this reason, the instructions in the BRANCH category must manage the branch table, which maintains information about the branch target address and the forwarded stack status.

For an instruction in the OBJECT category, because a double or long type field is 64-bits wide, the type of its field variable must be discovered. The size of the variable $v$, $sz(v)$, and the act of an instruction $x$ in the OBJECT category are calculated as follows:

$$sz(v) = \begin{cases} 2 & \text{if type}(v) = \text{long or double}, \\ 1 & \text{otherwise}. \end{cases}$$
Table 1 Classification table of the Java bytecodes in relation to the operand behavior.

<table>
<thead>
<tr>
<th>Category</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORMAL</td>
<td>dastore caste</td>
</tr>
<tr>
<td></td>
<td>caste dcmpl dcmpg</td>
</tr>
<tr>
<td></td>
<td>dadd ddiv dmul drem dstore dret</td>
</tr>
<tr>
<td></td>
<td>dsipush dstore</td>
</tr>
<tr>
<td>BRANCH</td>
<td>aconst_b invokespecial invokevirtual invokestatic invokevirtual</td>
</tr>
<tr>
<td>OBJECT</td>
<td>aaload iload iload</td>
</tr>
<tr>
<td></td>
<td>iload iload iload</td>
</tr>
<tr>
<td></td>
<td>iload iload iload</td>
</tr>
<tr>
<td></td>
<td>iload iload iload</td>
</tr>
<tr>
<td></td>
<td>iload iload iload</td>
</tr>
</tbody>
</table>

From the stack status of each bytecode, the stack patterns can be obtained by tracing the bytecodes sequentially. The stack pattern and the stack pattern set are defined as follows:

**Definition 2 (Stack Pattern):** Let \( p \) be a Java class file and \( bytecode(p) \) be a full sequence of Java bytecodes stored in \( p \). The bytecode sequence \( a = (x_1, x_2, \ldots, x_n) \) is called a stack pattern of \( p \) iff:

1. \( x_1, x_2, \ldots, x_n \) is a contiguous subsequence of \( bytecode(p) \).
2. Before \( x_1 \) is executed, the stack status starts from 0.
3. After \( x_1, \ldots, x_{n-1} \) are executed, the stack status never reaches 0.
4. After \( x_n \) is executed, the stack status reaches 0.

**Definition 3 (Stack Pattern Set):** Let \( p \) be a Java class file. The set of all the stack patterns in \( bytecode(p) \) is then called the stack pattern set of \( p \).

In Definition 2, the stack pattern represents a minimal sequence of Java bytecodes partitioned via their operand stack status. Each bytecode belongs to only one stack pattern and does not appear in more than one stack pattern. The stack pattern set is the set of all the stack patterns found in the sequence of Java bytecodes. There is unique stack pattern set for every Java class file because the stack pattern is a minimal sequence of bytecodes and it does not overlap with other stack patterns.

### 3.3 Weight of Stack Patterns

In a comparison of two objects, it is important to catch their discriminating characteristics of each object. In this paper, we utilize the sequences of instructions in each program as the program discriminators. However, because some frequently occurring instructions are widely used in almost every program, these instructions cannot be a good identifier for a program. \(^1\) Hence, the gravity of each bytecode must

\(^1\)In our experiment, the six most frequent bytecodes (invokevirtual, aaload, invokevirtual, invokespecial, dup, and dastore) amounted to 45% of the total bytecodes.
be balanced in relation to its specificity.

The inverse document frequency (IDF), which is based on the occurrence ratio of each term in the documents, is well-suited for this purpose. The intuition is that any term which frequently appears in many documents cannot be a good identifier of a document and should be given less weight than other terms [13]. Let \( N \) be the number of documents in the collection. If a term \( t_i \) occurs in \( n_i \) documents, the weight, \( idf(t_i) \), is as follows:

\[
idf(t_i) = \log \frac{N}{n_i}.
\]

In a similar way, the weight of opcode, \( w(\text{opcode}) \), is calculated as follows:

\[
w(\text{opcode}) = \log \frac{C}{\text{num}(\text{opcode})},
\]

where \( C \) is the total number of class files and \( \text{num}(\text{opcode}) \) is the number of class files in which the opcode appears. From the weight of each bytecode, we can define the weight of a stack pattern as follows:

**Definition 4 (Weight of Stack Patterns):** Let \( a = (x_1, x_2, \ldots, x_n) \) be a stack pattern in a class file and \( w(x) \) be the weight of bytecode \( x \). The weight of stack pattern \( a \), which is denoted by \( w_a \), is then defined as follows:

\[
w_a = \sum_{x \in a} w(x) = w(x_1) + w(x_2) + \cdots + w(x_n).
\]

The fundamental task of the birthmarking system based on weighted stack patterns is to find similarity values among each pair of weighted stack patterns between two class files. To clarify the similarity between two stack patterns, we need to find the most weighted common subsequence (WCS) between two stack patterns. The WCS, which maximizes the weight of the common subsequence, represents a sequence of bytecodes that are matched to indicate the weighted similarity between two stack patterns. The WCS can be calculated by means of a variant of the longest common subsequence algorithm [14]; that is, the WCS algorithm focuses on the weight rather than the length of the common subsequence between two stack patterns.

**Figure 1** (a) shows an example of two stack patterns. The full sequences of bytecodes represent the stack patterns, and \( w \) shows the matching weight of each bytecode. The Stack shows the stack status of each bytecode after the bytecode has been executed by means of some bullets, which denote the number of elements pushed onto the stack. Figure 1 (b) shows the algorithm for finding a WCS between \( a \) and \( b \); this value is denoted by \( WCS(a, b) \). Suppose \( a = (x_1, \ldots, x_n) \) and \( b = (y_1, \ldots, y_m) \) are two stack patterns to be matched. Beginning at the top left cell, the matching weight up to position \((i,j)\), \( c(i,j) \), is calculated as follows:

\[
c(i,j) = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0, \\ c[i-1, j-1] + w(x_i) & \text{if } i, j>0 \text{ and } x_i=y_j, \\ \max(c[i-1,j], c[i,j-1]) & \text{otherwise.} \end{cases}
\]

The maximum weight up to the position is placed in each cell. The traces in the grid provide a method of computing the WCS between two stack patterns.

In Fig. 1 (b), the weight in the bottom right cell is the weight of \( WCS(a, b) \), which is denoted by \( w_{(a,b)} \). Furthermore, the trace of the boldfaced weights leads to the WCS between two stack patterns. From the calculation, \( WCS(a, b) = (i\text{load}_0, i\text{load}_0, \text{isub}, \text{invokestatic}, \text{return}) \), and the weight of \( WCS(a, b) \) is \( w_{(a,b)} = 21 \).

**Definition 5 (Weight of the WCS):** Let \( a \) and \( b \) be stack patterns in programs \( p \) and \( q \), respectively. Let \( WCS(a, b) \) be the most weighted common subsequence between \( a \) and \( b \). The weight of \( WCS(a, b) \), \( w_{(a,b)} \), is then defined as follows:

\[
w_{(a,b)} = \begin{cases} \sum_{x \in WCS} \text{w}(x) & \text{if } \alpha \times |WCS| > \frac{|a| + |b|}{2}, \\ 0 & \text{otherwise,} \end{cases}
\]

where \( \alpha (\geq 1) \) is a coefficient for matching two stack patterns, and \(|a|\) denotes the length of bytecode sequence \( a \).

As the matching coefficient \( \alpha \) increases, the matching condition is weakened, leading to a positive matching weight of the WCS. Thus, as the matching coefficient \( \alpha \) increases, resilience increases but credibility may decrease.
3.4 The Proposed Birthmark

To detect software theft via a software birthmark, we need to measure the similarity between two birthmarks. So, for a whole birthmarking system, it is necessary to provide a function for extracting the birthmark from a program and a measure for finding the similarity between birthmarks.

**Definition 6 (Stack Pattern Based Birthmark):** Let \( p \) be a Java class file. The stack pattern set of \( p \) is then called the stack pattern based birthmark of \( p \), denoted by \( \mathcal{B}(p) \).

Let \( \mathcal{B}(p) \) and \( \mathcal{B}(q) \) be birthmarks of Java class files \( p \) and \( q \), respectively. Then \( p \) and \( q \) are suspected of having a copy relation if \( \mathcal{B}(p) \simeq \mathcal{B}(q) \); that is, \( \text{Similarity}(\mathcal{B}(p), \mathcal{B}(q)) \geq 1 - \epsilon \), where \( \epsilon \) is a threshold value for identifying software in a copy relation.

To calculate the similarity between two birthmarks, we have to consider the relations between every pair of stack patterns in each birthmark. The relation between two stack patterns represents the degree of similarity through the weight value, so it can be represented by the weight of the WCS between two stack patterns (see Definition 5). For example, if \( \mathcal{B}(p) = \{a_1, a_2, \ldots , a_n\} \) and \( \mathcal{B}(q) = \{b_1, b_2, \ldots , b_m\} \) are birthmarks, then the weight matrix of \( \mathcal{B}(p) \) and \( \mathcal{B}(q) \), denoted by \( W(\mathcal{B}(p), \mathcal{B}(q)) \), is organized as follows:

\[
WM(\mathcal{B}(p), \mathcal{B}(q)) = \begin{pmatrix}
W(a_1, b_1) & W(a_1, b_2) & \cdots & W(a_1, b_m) \\
W(a_2, b_1) & W(a_2, b_2) & \cdots & W(a_2, b_m) \\
\vdots & \vdots & \ddots & \vdots \\
W(a_n, b_1) & W(a_n, b_2) & \cdots & W(a_n, b_m)
\end{pmatrix}
\]

From the weight matrix, the overall similarity can be obtained by matching similar pairs of stack patterns in each birthmark. The matching task is solved by a search problem for finding the matched pairs which maximize the total sum of their weights. This problem can be reduced to the maximum weighted bipartite matching problem [14]. In other words, when each stack pattern and weight of the WCS correspond to a node of a bipartite graph and a weighted edge, respectively, the similarity between two birthmarks is calculated by maximizing the total weight of matched pairs in the weighted bipartite matching problem. The matching is solved by the Hungarian algorithm [15], which finds an optimal set of the matched pairs in \( O(n^3) \). The matched pattern set between two birthmarks represents the set of pairs of the most similar stack patterns among all the pairs of stack patterns between birthmarks.

**Definition 7 (Matched Pattern Set \( M \)):** Let \( \mathcal{B}(p) \) and \( \mathcal{B}(q) \) be birthmarks of Java class files \( p \) and \( q \), respectively. The matched pattern set of \( \mathcal{B}(p) \) and \( \mathcal{B}(q) \), which is denoted by \( M(\mathcal{B}(p), \mathcal{B}(q)) \), is defined as the set of pairs of stack patterns that maximizes the total sum of the weights of the WCS.

**Definition 8 (Weight of Birthmark \( W \)):** Let \( \mathcal{B}(p) \) and \( \mathcal{B}(q) \) be birthmarks of Java class files \( p \) and \( q \), respectively. Let \( M(\mathcal{B}(p), \mathcal{B}(q)) \) be the matched pattern set of \( \mathcal{B}(p) \) and \( \mathcal{B}(q) \). The weight of the birthmark, \( W \), is then defined as follows:

\[
W(\mathcal{B}(p)) = \sum_{a \in \mathcal{B}(p)} w_a ,
\]

\[
W(\mathcal{B}(p), \mathcal{B}(q)) = \sum_{(a,b) \in M(\mathcal{B}(p), \mathcal{B}(q))} w_{(a,b)} .
\]

The weight of the birthmark is calculated by summing the weights of all the stack patterns in the birthmark. The weight between two birthmarks is calculated by summing the weights of all the pairs in the matched pattern set of two birthmarks.

Broder [16] defined the notions of resemblance and containment to measure the similarity of two documents. Software can be used in a larger program as a module or it can be modified to a different structure by inserting additional codes. In these cases, the original program is contained as some modified structures in the other program. A measure based on containment is therefore more reasonable when investigating whether some part of an original program is contained in a suspicious program. The similarity of two birthmarks is defined as follows:

**Definition 9 (Similarity of Birthmarks):** Let \( p \) and \( q \) be Java class files and \( \mathcal{B}(p) \) and \( \mathcal{B}(q) \) be birthmarks of \( p \) and \( q \), respectively. The similarity of birthmarks \( \mathcal{B}(p) \) and \( \mathcal{B}(q) \) is then defined as follows:

\[
\text{Similarity}(\mathcal{B}(p), \mathcal{B}(q)) = \frac{W(\mathcal{B}(p), \mathcal{B}(q))}{W(\mathcal{B}(p))} = \frac{\sum_{(a,b) \in M(\mathcal{B}(p), \mathcal{B}(q))} w_{(a,b)}}{\sum_{a \in \mathcal{B}(p)} w_a} .
\]

The similarity of two birthmarks, as shown in Eq. (3), is calculated by summing all the weights of the matched pairs. To normalize the similarity, we need to divide the sum by the weight of the original program’s birthmark. Thus, the resulting similarity ranges from 0 to 1 in proportion to the degree of similarity between the two birthmarks.

For example, let \( \mathcal{B}(p) = \{a_1, a_2, a_3, a_4\} \) and \( \mathcal{B}(q) = \{b_1, b_2, b_3, b_4\} \) be birthmarks of programs \( p \) and \( q \), respectively. Let the weights of \( \mathcal{B}(p) \) and \( \mathcal{B}(q) \) be \( W(\mathcal{B}(p)) = 75 \) and \( W(\mathcal{B}(q)) = 90 \), respectively. Let

\[
WM(\mathcal{B}(p), \mathcal{B}(q)) = \begin{pmatrix}
6 & 6 & 9 & 3 \\
21 & 18 & 18 & 6 \\
9 & 6 & 12 & 3 \\
6 & 9 & 30 & 9
\end{pmatrix}
\]

be the weight matrix of \( \mathcal{B}(p) \) and \( \mathcal{B}(q) \). Using the search algorithm for matching stack patterns, the matched pattern set can be obtained as \( M(\mathcal{B}(p), \mathcal{B}(q)) = \{(a_1, b_2), (a_2, b_1), (a_3, b_2), (a_4, b_3)\} \). Finally, the similarity between two birthmarks is calculated as follows:

\[
\text{Similarity}(\mathcal{B}(p), \mathcal{B}(q)) = \frac{21 + 6 + 30 + 3}{75} = 0.80 .
\]
4. Experimental Results

4.1 Preliminaries

In this section, the proposed birthmark is evaluated with respect to two properties required for a birthmark: credibility and resilience. Algorithm 2 shows the brief procedure for birthmarking two Java class files and calculating the similarity between them. The proposed birthmark was implemented in C language and evaluated on an Intel Pentium-4 2.4 GHz PC with 2 GB RAM running MS Windows XP. In this experiment, the proposed birthmark was evaluated with the matching coefficient of \( \alpha = 3 \). We calculated the weight of each bytecode by using Apache Ant 1.5.4 [20], Jakarta BCEL 5.2 [21], and JUnit 4.4 [23] as a universal library set.

Four Java programs were used as target applications: namely, Ant, BCEL, Apache Commons [22], and JUnit. Table 2 shows the features of the applications. Each Java class file consists of about 13.4–45.8 stack patterns on average, ranging from 0 to 869. Some Java class files have no stack pattern; that is, the class files perform no stack operation. Each stack pattern consists of about 3.9–5.0 bytecodes on average, ranging from 2 to 2051.

### Table 2 Description of the Java applications.

<table>
<thead>
<tr>
<th>Program</th>
<th>Size (bytes)</th>
<th>Number of class files</th>
<th>Number of bytecodes</th>
<th>Number of stack patterns</th>
<th>Number of bytecodes / class</th>
<th>Number of stack patterns / class</th>
<th>Number of bytecodes / stack pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant 1.5.4</td>
<td>736,810</td>
<td>406</td>
<td>92,623</td>
<td>18,607</td>
<td>228.1 (0–4155)</td>
<td>45.8 (0–820)</td>
<td>4.4 (2–2051)</td>
</tr>
<tr>
<td>BCEL 5.2</td>
<td>533,339</td>
<td>383</td>
<td>68,258</td>
<td>11,843</td>
<td>178.2 (0–7077)</td>
<td>30.9 (0–869)</td>
<td>5.0 (2–1507)</td>
</tr>
<tr>
<td>Commons 3.2</td>
<td>161,477</td>
<td>450</td>
<td>59,401</td>
<td>14,590</td>
<td>130.0 (0–2011)</td>
<td>32.4 (0–455)</td>
<td>3.9 (2–44)</td>
</tr>
<tr>
<td>JUnit 4.4</td>
<td>571,259</td>
<td>154</td>
<td>9,715</td>
<td>2,068</td>
<td>68.1 (0–540)</td>
<td>13.4 (0–117)</td>
<td>4.3 (2–37)</td>
</tr>
</tbody>
</table>

4.2 Experiment 1: Credibility

In this experiment, the proposed birthmark is evaluated in terms of credibility. A Java application package consists of many class files and each class file is supposed to be different from each other. Hence, a birthmark must distinguish the different class files in the same package. The applications described in Table 2 were used as target applications. For each package, the birthmarks of all class files were extracted and compared with all other birthmarks in the same package.

**Algorithm 2** Calculating similarity between two programs.

**INPUT** Programs \( p \) and \( q \)

**OUTPUT** Similarity between two programs \( p \) and \( q \)

\[
B(p) \leftarrow \text{ExtractBirthmark}(p)
\]

\[
B(q) \leftarrow \text{ExtractBirthmark}(q) \text{ (See Sect. 3.2.)}
\]

for all \( i \) such that \( a_i \in B(p) \)

for all \( j \) such that \( b_j \in B(q) \)

\[
\text{WM}(i, j) = w(a_i, b_j) \text{ (See Sect. 3.3.)}
\]

end for

end for

Find the matched pattern set, \( M(B(p), B(q)) \) (See Sect. 3.4.)

Calculate the Similarity \( \text{Similarity}(B(p), B(q)) \)

Figure 2 (a) shows the results of the credibility experiment. To evaluate the time overhead of the birthmark, we measured the execution time taken for birthmarking and comparing all the class files from each Java package. The average execution time for one comparison was about 0.03–0.12 seconds, so its time overhead was tolerable. The average similarities of the different class files ranged between 20.6–22.8%. Figure 2 (b) shows the distribution of the similarities between the birthmarks in each package. The graph shows the overall distribution of all comparisons and can be the evaluation criterion of the effectiveness of a birthmark. The horizontal axis represents the similarity ranges between birthmarks, and the vertical axis represents the percentage of similarities that belong to each similarity range. For the credibility evaluation, a birthmark must clearly distinguish different programs. Thus, a birthmark has more credibility if more comparisons are distributed in lower similarity ranges. The similarities lower than 10% amounted to about 35–55% among 268,174 comparisons of the birthmarks, and the similarities higher than 50% were sparsely distributed. From these results, the proposed birthmark distinguished most of the class files with low similarity values, showing credibility of the birthmark.

In this experiment, however, not all the pairs of the class files were distinguished by the proposed birthmark. We observed the class files to determine the reason why the birthmark could not distinguish all of them. The cases are as follows:

1. when two class files are exactly the same including the class name;
2. when two class files have a set of identical stack patterns except for the operands of some bytecodes; and
3. when a smaller class file is embedded in a larger one.

In case 1, the two class files are located in different directory structures but identical files. In case 2, two class files perform identical routines using partially different data, yielding identical sequences of bytecodes. In case 3, two class files have different structures but a smaller class file is embedded in a larger one. From our observation, we deduce that the undistinguished pairs are not false positives but pairs detected as being in a copy relation by the proposed birthmark.

4.3 Experiment 2: Resilience to Program Transformation

Because a software cracker may use certain transformation or obfuscation to hide the fact of software theft, a birth-
mark must be sufficiently resilient to program transformation. In this experiment, the proposed birthmark is evaluated in terms of its resilience to program transformation. Smoke-screen [26] and Jarg [27] were used for program transformation and Jikes [25] and javac were used for a different compiler. Smoke-screen is a Java obfuscator that can obfuscate the control flows by modifying the bytecodes in the class files. Jarg optimizes Java class files by removing unnecessary attributes for execution. We transformed Ant, BCEL, and Commons by using Smoke-screen and Jarg with the strongest transformation level. Next, we compared the extracted birthmarks of each pair of original class files and the transformed version. The source programs were also compiled with the aid of Jikes and javac, and the class files were birthmarked and compared.

Figure 3 (a) shows the results of the experiment on resilience to program transformation. A detection ratio represents the percentage of detecting pairs of class files that are in a copy relation; that is, the similarity of two birthmarks is 100%. The average similarities reached 90.0–96.7% and the similarities higher than 90% amounted to about 65–85% of 3,717 comparisons. In addition, the detection ratios reached about 38.9–66.5%. Figures 3 (b) to 3 (d) show the distribution of the similarities between the pairs of the original program and a version of the program transformed by Smoke-screen, Jarg, and Jikes. For the resilience evaluation, a birthmark must be able to detect software theft in spite of program transformation. Therefore, if a birthmark is resilient to program transformation, as many comparisons as possible must be distributed in higher similarity ranges. From the graph, most of comparisons are located in the similarity ranges higher than 80%. These results suggest that the majority of birthmarks in the original programs are still preserved even after the program transformation. Moreover, the proposed birthmark seems robust enough to endure program transformations resulting from such means as obfuscation, optimization, or a different compiler.

We observed bytecodes in the pairs of class files with lower similarity than other pairs. Firstly, we found that when obfuscation changes the control flow of a program through branch instructions, such as goto, a stack pattern can be separated into several parts. This problem can be solved if the static order among the stack patterns can be determined by analysis of the branch instructions. Secondly, for birthmarks created by different compilers, we found that the evaluation order of operands was sometimes different in accordance with the compiler’s code generation strategy. This difference made the evaluation order of the stack operation different, yielding a different sequence of bytecodes. Moreover, some operations were generated to similar but different

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of comparisons</th>
<th>Exec. time (s)</th>
<th>Similarity (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ant</td>
<td>82,215</td>
<td>9699.3</td>
<td>20.6</td>
</tr>
<tr>
<td>BCEL</td>
<td>73,153</td>
<td>7311.6</td>
<td>22.5</td>
</tr>
<tr>
<td>Commons</td>
<td>101,025</td>
<td>7239.2</td>
<td>21.7</td>
</tr>
<tr>
<td>JUnit</td>
<td>11,781</td>
<td>357.0</td>
<td>22.8</td>
</tr>
</tbody>
</table>

(a) Results of credibility experiment.

(b) The distribution graph.
bytecodes because some operands were changed differently by the evaluation order.

4.4 Comparison with Existing Birthmarks

The previous experiments show that the proposed birthmark is credible and resilient to program transformation. There are also other static birthmarks, such as that of Tamada et al. [5], [6] and the $k$-gram birthmark [7], which can be applied to Java applications. However, the performance evaluation of these birthmarks is not satisfactory. Myles [18] compared the $k$-gram birthmark with that of Tamada et al. but the experiment was performed on only small Java programs, such as factorial, fibonacci, and wc. For a comprehensive comparison, we need to evaluate the birthmarks in real-world applications. Hence, we evaluated and compared the proposed birthmark with the existing birthmarks in real-world applications.

Stigmata 1.1 [24] was used for the birthmark of Tamada et al. and the $k$-gram birthmark was implemented along with the proposed birthmark. To evaluate credibility and resilience of the birthmarks, we performed the experiment referred to in Sects. 4.2–4.3. Apache Ant 1.5.2 was used as a target application, and Jikes and Smokescreen were used to evaluate the resilience of the birthmarks.

Figures 4(a) and 4(c) show the results of the experiment with respect to credibility. The average similarities between different class files ranged between 11.1% and 20.6% for each birthmark, and the distribution of the similarities make no distinctive difference. Figures 4(b) and 4(d) show the results of the experiment with respect to resilience. From the results, the distribution in Fig. 4(d) takes on a different look according to individual birthmark. In the case of the birthmark of Tamada et al., the average similarities were 75.6% for Jikes and 75.4% for Smokescreen. The detection ratios were 77.7% and 76.4%, respectively, and the similarities between 70–80% amounted to about 50–70%. For the $k$-gram birthmark, the average similarities were 89.2% for Jikes and 91.1% for Smokescreen. The detection ratios were 33.0% and 31.2%, respectively, and the highest peaks were located between 80–90%. For the proposed birthmark, the average similarities were 93.1% and 91.5%, and the similarities between 90–100% reached about 60–70%. Moreover, the detection ratios were 40.3% and 43.8%, respectively, and they are more highly ranked than other birthmarks. In short, all of the birthmarks were similar in terms of credibility. However, the proposed birthmark was more stable than the existing birthmarks in terms of resilience to program transformation. We can confirm therefore that the proposed birthmark is more advantageous for detecting software theft when someone other than the original author modifies the software to hide the fact of software theft.
file. Thus, if an original program is embedded as a module in a larger program, software theft is difficult to detect. For practical application we need to consider the notion of containment. The $k$-gram birthmark is similar to the proposed birthmark in that they compare two birthmarks in terms of the bytecodes of programs. However, the $k$-gram birthmark uses blind $k$ sequences of bytecodes as its birthmark. This process is problematic when a program is modified by transformation because modification or obfuscation can change the $k$-gram itself by reordering the instructions, changing the control flows, inserting additional codes, and so on. On the other hand, the proposed birthmark uses the stack patterns as its birthmark. Because the stack pattern is a sequence of bytecodes that are dependent on each other through their operand stack, it is more reliable even in cases of program modification or transformation.

5. Discussion

Strictly speaking, credibility is contrary to resilience. In other words, as the credibility improves, the resilience is likely to decrease. There are several design issues for this trade-off.

In Java bytecodes, there are many kinds of similar bytecodes designed to optimize the Java Virtual Machine operations. Depending on the strategy of compilers, the same operation might be compiled into similar but different bytecodes; for example, to push an integer onto the operand stack, `bipush` or `iconst_j` may be used. Thus, the compiler can choose a different bytecode for the same operation. Moreover, some bytecodes may be exchanged with similar bytecodes by an optimizer or obfuscator. For this reason, resilience can be improved if the bytecodes in the stack patterns can be abstracted to absolute codes that represent their pure functionality. However, if the bytecode abstraction is applied, the overall similarity between the birthmarks may also increase because abstract bytecodes have a greater chance of being matched. Thus, credibility of the birthmark may deteriorate.†

To calculate the similarity between two birthmarks, we need to obtain the weights of all the matched pairs between two birthmarks. The Hungarian algorithm is used for this problem and it optimally maximizes the total weight of the matched pairs. If different programs are birthmarked and compared, this algorithm tries to maximize the weight by compulsive matching of different stack patterns, thereby wasting most of the execution time. A greedy algorithm can be used to alleviate this shortcoming by finding pairs of stack patterns in the weight order of the matched pairs. The overall similarity may then decrease and the execution time is reduced.††

When the birthmarks are extracted from Java class files, only the opcodes of the bytecodes are considered. However, Java bytecodes have operands which contain much of the information required during the execution of a program, such as the name of invoked methods, the name of used classes, and the type of field variables. This information can also be useful to refine the proposed birthmark.

6. Conclusion

A software birthmark refers to program’s inherent characteristics that can be used to identify the program. By comparing the birthmarks of programs, we can detect the occurrence of software theft. In this paper, we proposed a static Java birthmark that uses a set of stack patterns as the characteristic of Java applications. A stack pattern denotes a sequence of bytecodes that share their operands through the operand stack. In a comparison of the birthmarks, a weight scheme is used to balance the influence of each bytecode. The similarity between two birthmarks is obtained by matching the most similar stack patterns in each birthmark.

We have evaluated the proposed birthmark with respect to two properties required for a birthmark; namely, credibility and resilience. The empirical results show that the proposed birthmark is highly credible and resilient to program transformation. We also compared the proposed birthmark with the birthmark of Tamada et al. and the $k$-gram birthmark. The results of the comparison confirms that all three birthmarks are similar in terms of credibility and that the proposed birthmark is more stable than the existing birthmarks in terms of resilience to program transformation. The proposed birthmark can therefore provide more reliable evidence of software theft than the other birthmarks when software is modified by someone other than the original author. At the same time, the proposed birthmark can be used with other techniques to provide more conclusive evidence of program theft.

There is a possibility that the proposed birthmark may be confused with branch instructions generated by program transformations. For future work, we plan to refine the proposed birthmark to tolerate such shortcomings by analyzing the branch instructions.

††In our experiments on bytecode abstraction, the similarities increased by 0.3%–9.8% on average and the detection ratio also increased by 1.8%–6.9%. Most of the similarities increased but about 0.5% of them decreased because the matching weight of the abstract bytecode does not preserve that of the original Java bytecode. The similarity differences by abstraction ranged between −41.1% and 100%. In some cases, a small Java program is perfectly matched with a large program through bytecode abstraction.

††In our experiments on greedy matching, the overall similarity decreased by 0.2% on average. In most cases, the greedy algorithm found the optimal matching. Suboptimal matching was found in only 3.7%–7.5% of the comparisons. The comparing time was also reduced by 35.5% on average. The greedy algorithm yielded similarity differences that ranged between −25% to 0% and no differences in the detection ratios. Thus, when the execution time is more critical, the greedy algorithm is a good candidate for matching without much loss of resilience.
References

[27] “Java Archive Grinder (jarg)” http://sourceforge.net/projects/jarg/