A Static Java Birthmark Based on Operand Stack Behaviors

Heewan Park  Hyun-il Lim  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
{hwpark,hilim,swchoi}@pllab.kaist.ac.kr  han@cs.kaist.ac.kr

Abstract

A software birthmark means the inherent characteristics of a program that can be used to identify the program. By means of comparing the birthmarks of programs, the fact of the software theft can be detected. In this paper, a static Java birthmark is proposed by utilizing a set of behaviors as the characteristics of the Java applications. A behavior denotes a sequence of bytecodes which share their operands through the operand stack.

We evaluate the proposed birthmark with respect to two properties required for birthmark, i.e., credibility and resilience. The empirical results show that the proposed birthmark is credible and resilient to program transformation. Therefore, the proposed birthmark can be used for identifying the software’s originality.

1 Introduction

Recently an enormous number of software products are developed and distributed all over the world. With the recent advancement of internet environments, the cases of software theft are rapidly increasing. Since software theft can cause severe damage to the developers and software industries, the authors must protect their own intellectual properties from illegal attempts. Therefore, the technology for verifying the software’s originality is demanded increasingly.

A software birthmark means the inherent characteristics of a program that can be used to identify the program. If programs $p$ and $q$ have the same or similar birthmark, $q$ is likely to be a stolen copy of $p$ (and vice versa). For example, comparing the strings analyzed from the program could be a naive birthmarking technique. For this purpose, several properties are needed for the birthmark. The detector should not produce false positives (i.e., it should not say that $p$ and $q$ are in copy relation, if, in fact, they do not originate from the same source), and it should be resilient to semantics-preserving transformations (such as optimization and obfuscation) that an attacker may launch in order to defeat the detector.

Currently, there are a variety of techniques to prevent, discourage, and detect the theft. In this paper we present and evaluate a specific technique — a static Java birthmark based on operand stack behaviors. The program characteristic used as a birthmark of Java program is a set of possible behaviors during the execution of a program. We statically identify behaviors by analyzing Java bytecodes stored in the Java program. A similarity between two programs is calculated by matching the set of behaviors.

We evaluate the proposed birthmark with respect to two properties required for the birthmark, e.g., credibility and resilience, through several real-world Java applications. The empirical results show that the proposed birthmark is credible and resilient enough to identify the software’s originality.

2 Related Work

If the source code of a program is available, plagiarism detection [8, 13, 1] can be applied to detect the suspected copy of the program. It calculates program similarity based on the structural information of each program. However, if the source code of a program is unavailable, it is required to utilize information directly from the program itself.

Tamada et al. [10, 11] suggested the first static birthmarks for Java programs, which consists of four individual birthmarks summarizing the overall structures of Java programs. Myles et al. [7] proposed the $k$-gram birthmark, a
static birthmark based on a set of sequences of \( k \) contiguous Java bytecodes in a program.

Tamada et al. [12] introduced the definition of a dynamic birthmark and proposed two such birthmarks based on the sequence and the frequency of system calls for Windows programs. Myles et al. [6] proposed a whole program path (WPP) birthmark based on the dynamic traces of a program and compared the birthmarks using a graph distance for a maximal common subgraph. Shuler and Dallmeier [9] presented a dynamic birthmark based on API call sequence sets during program execution. Lu et al. [5] presented another dynamic birthmark using the set of \( n \)-grams for instruction sequences extracted from the runtime instances during program execution.

3 A Static Birthmark based on Stack Behaviors

3.1 Software Birthmark

Tamada et al. [10, 11] formally defined a software birthmark using copy relation. The following definition and properties are the restatements of those of Tamada et al. [10, 11] and Myles et al. [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 \( f(P) \neq f(Q) \).

**Property 2 (Resilience)** Let \( p' \) be a program obtained from \( p \) by applying semantics-preserving transformation, such as optimization or obfuscation. Then \( f(P) = f(P') \) holds.

The credibility property is a criterion to exclude the possibility of false positives. In other words, although programs \( P \) and \( Q \) have the same functionality, if they are developed independently, the birthmark should be different. The resilience property specifies that the birthmark of \( P \) must be remained in its original form although program transformation changes the structure of the program, i.e., the software birthmark must be strong enough to endure the semantics-preserving transformation.

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Stack</th>
<th>ABS</th>
</tr>
</thead>
<tbody>
<tr>
<td>invokevirtual</td>
<td></td>
<td></td>
</tr>
<tr>
<td>invokestatic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ireturn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Stack</th>
<th>ABS</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload_0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>icount</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iload_0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ireturn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1. Sample behaviors**

3.2 Behaviors in Java Program

Java bytecode uses the operand stack as a workspace and the bytecodes share the operand each other through the operand stack. Because the specification of Java bytecode is rigorously defined [3], its operand stack status can be determined by static analysis. A behavior and a behavior set are defined as follows:

**Definition 2 (Behavior)** Let \( P \) be a Java program and \( \text{seq} \) be a consecutive subsequence of bytecodes of \( P \). If the operand stack depth just before and after executing \( \text{seq} \) reaches to 0, \( \text{seq} \) is called a behavior of program \( P \).

**Definition 3 (Behavior Set)** Let \( P \) be a Java program. Then the set of all behaviors in program \( P \), \( S(P) \), is called a behavior set of \( P \).

In Definition 2, a behavior represents a minimal sequence of Java bytecodes partitioned via their operand stack status. Each bytecode belongs to only one behavior and it does not appear in more than one behavior. A behavior set is a set of all behaviors found in the sequence of Java bytecodes. Figure 1 shows two sample behaviors and their stack depth after executing a bytecode. \( \text{ABS} \) displays the abstracted bytecode which represents the pure functionality of its operation.

3.3 The Proposed Birthmark

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

**Definition 4 (A Static Birthmark)** Let \( P \) be a Java program, then a behavior set of \( P \), \( S(P) \), is a static birthmark of \( P \).
A similarity between two behaviors can be calculated using longest common subsequence (LCS) and shortest common supersequence (SCS) algorithm [2]. Let \( a \) and \( b \) be behaviors and \( LCS(a, b) \) and \( SCS(a, b) \) be the LCS and SCS of \( a \) and \( b \) respectively. Then the similarity between two behaviors \( a \) and \( b \), \( sim(a, b) \), can be calculated as follows:

\[
sim(a, b) = \frac{|LCS(a, b)|}{|SCS(a, b)|} = \frac{|LCS(a, b)|}{|a| + |b| - |LCS(a, b)|}
\]

where \(|a|\) is the length of a sequence \( a \). For example, the LCS of two behaviors shown in Figure 1 is obtained as \( LCS(a, b) = \{load, load, const, sub, invk, ret\} \). Therefore, the similarity between two behaviors \( a \) and \( b \), \( sim(a, b) \), is as follows:

\[
sim(a, b) = \frac{6}{7 + 10 - 6} = \frac{6}{11}
\]

To calculate a similarity between two birthmarks, the similarities between every pairs of behaviors in each birthmark has to be considered. The similarity between two birthmarks is obtained by maximizing the sum of similarities of matched behaviors contained in two programs. This can be solved by Hungarian algorithm [4] which finds the optimal set of matched pairs in \( O(n^3) \). Therefore, the matched behavior set between programs \( P \) and \( Q \), \( match(P, Q) \), represents the set of the most similar pairs among all behaviors in each birthmark. The similarity of two programs is defined as follows:

**Definition 5 (Similarity of Birthmark)** Let \( P \) and \( Q \) be Java programs and \( S(P) \) and \( S(Q) \) be the birthmarks of \( P \) and \( Q \) respectively. Let \( match(P, Q) \) be the set of matched pairs which maximizes the sum of similarities. Then the similarity of programs \( P \) and \( Q \) is defined as follows:

\[
\text{Similarity}(P, Q) = \frac{\sum_{(a, b) \in \text{match}(P, Q)} \sim(a, b)}{\max(|S(P)|, |S(Q)|)}
\]

\[
\text{Containment}(P, Q) = \frac{\sum_{(a, b) \in \text{match}(P, Q)} \sim(a, b)}{|S(P)|}
\]

where \(|S(P)|\) is the number of behaviors in a behavior set \( S(P) \).

The similarity of two Java programs is calculated by adding all similarities of the matched pairs. To normalize the similarity, the sum is divided by the number of behaviors of programs. Therefore, the resulting similarity ranges from 0 to 1 in proportion to the degree of similarity between two birthmarks. A containment is a measure to investigate whether the part of the original program is contained in the suspicious program or not.

For example, let \( P \) and \( Q \) be programs and \( S(P) = \{a_1, a_2, a_3, a_4\} \) and \( S(Q) = \{b_1, b_2, b_3, b_4\} \) be the behavior set of \( P \) and \( Q \) respectively. Let

<table>
<thead>
<tr>
<th></th>
<th>( b_1 )</th>
<th>( b_2 )</th>
<th>( b_3 )</th>
<th>( b_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_1 )</td>
<td>0.200</td>
<td>0.200</td>
<td>0.333</td>
<td>\textbf{0.100}</td>
</tr>
<tr>
<td>( a_2 )</td>
<td>0.750</td>
<td>\textbf{0.750}</td>
<td>0.666</td>
<td>0.200</td>
</tr>
<tr>
<td>( a_3 )</td>
<td>\textbf{0.300}</td>
<td>0.300</td>
<td>0.400</td>
<td>0.150</td>
</tr>
<tr>
<td>( a_4 )</td>
<td>0.200</td>
<td>0.200</td>
<td>\textbf{1.000}</td>
<td>0.200</td>
</tr>
</tbody>
</table>

be the similarity matrix between \( P \) and \( Q \). Using Hungarian algorithm [2], the matched behavior set is found as \( match(P, Q) = \{(a_1, b_4), (a_2, b_2), (a_3, b_1), (a_4, b_3)\} \). Therefore, the similarity between two programs \( P \) and \( Q \) is obtained as follows:

\[
\text{Similarity}(P, Q) = \frac{0.30 + 0.75 + 1.00 + 0.10}{\max(4, 4)} = \frac{2.15}{4}
\]
4 Experimental Results

The proposed birthmark was implemented in C languages in MS Windows environment. Apache Ant, Jakarta BCEL, and JUnit were used as target applications and Smokescreen was used for program obfuscation.

To show the credibility, we compare the birthmark for every pair in each packages. Figure 2 shows the result of the experiments and Figure 2(a) and 2(b) shows the distribution of similarities and containments respectively. The average similarity was between 17.7–20.6% and most of similarities are located below 40%. From this result, the proposed birthmark distinguished most of programs with low similarity values, showing the credibility of the birthmark. The average containment was between 47.1–56.5% and the distribution shows that a smaller module is partly embedded in a larger class file in many cases.

Since software cracker may use certain transformation or obfuscation to hide the fact of software theft, birthmark must be sufficiently resilient to program transformation. To show the resilience, we compare the programs in each application with the program obfuscated by Smokescreen. Figure 2 shows the result of the experiments and Figure 2(c) and 2(d) shows the distribution of the similarities and containments between the pair of the original program and its obfuscated version. The average similarity was between 82.0–86.7% and the distribution shows that most of birthmarks are remained in similar forms in spite of program obfuscation. Moreover, the average containment reached to 95.2% – 96.5% and the containments higher than 90% amounted to 80%. From the result, the majority of the original birthmarks were still preserved even after the obfuscation was applied. This result implies that the proposed birthmark is robust enough to endure the program obfuscation.

5 Conclusion

A software birthmark means the inherent characteristics of a program that can be used to identify the program. By means of comparing the birthmarks of programs, we can detect the fact of the software theft. In this paper, we proposed a static Java birthmark which uses a set of behaviors as the characteristics of the Java applications. A behavior denotes a sequence of bytecodes which share their operands through the operand stack. The proposed birthmark has been evaluated with respect to credibility and resilience. The empirical result showed that the proposed birthmark was highly credible and resilient to program transformation.

For the future work, we plan to refine the proposed birthmark to improve the credibility by employing the weight scheme to balance the effect of frequently occurred bytecodes.

References