A Static Java Birthmark Based on Control Flow Edges

Hyun-il Lim Heewan Park Seokwoo Choi Taisook Han
Division of Computer Science
Korea Advanced Institute of Science and Technology
Gwahangno 335, Yuseong-gu, Daejeon 305-701, Republic of Korea
Email: {hilim, hwpark, swchoi, han}@pllab.kaist.ac.kr

***Abstract*—A software birthmark is an inherent characteristic of a program that can be used to identify that program. By comparing the birthmarks of two programs, it is possible to infer if one program is a copy of another. In this paper, we propose a static birthmark based on the control flow edges in Java programs. Control flow edges can represent possible behaviors in program execution. Thus, a set of the control flow edges of a program can be used as a birthmark for that program. The similarity between two programs can then be calculated by finding pairs of similar behaviors of the control flow edges in the two birthmarks. The proposed birthmark is evaluated and compared with previous approaches in terms of credibility and resilience. Experimental results show that the proposed birthmark is more reliable than previous methods for detecting programs that are suspected to be copied.**

***Keywords*-software birthmark; software security; software copyright; program analysis;**

I. Introduction

Software under copyright is the intellectual property of software companies or its authors, and the license for such software must be protected. However, it has been reported that the violation of software licenses is pervasive [1], [2]. Because software theft causes many problems for the software industry and for software engineers, it is necessary to deter and detect software theft.

A software birthmark is an inherent characteristic of a program by which it may be identified. By comparing sets of birthmarks extracted from programs, the similarity between programs themselves may be inferred. A high level of similarity can be evidence of software theft: that is, if the similarity is sufficiently high, we can be convinced that one program is a copy of another.

In this paper, we present a static Java birthmark based on control flow edges. Because control flow information shows the possible control flows when a program is executed, it can be used as the identifying characteristic of a program. We use a set of control flow edges as a software birthmark representing possible sequences of instructions of a program. The proposed birthmark can also be used to detect the theft of the core modules of a program, because the set of control flow edges can reflect the algorithmic structures of the program.

The proposed birthmark method is evaluated here with respect to credibility and resilience. The credibility property requires a birthmark to clearly discriminate different programs although they accomplish the same tasks. The resilience property requires a birthmark to be preserved in its original form even after semantics-preserving transformations, such as obfuscation and optimization, are applied to the program. The results of the credibility and resilience experiments show that the proposed method is more reliable for detecting copied programs than existing approaches.

The remainder of this paper is organized as follows. Section II reviews existing approaches to software birthmarks and describes the motivation of our approach. Section III describes the formal definition of a software birthmark and proposes a static Java birthmark based on control flow edges. Section IV describes the experimental data and evaluates the proposed birthmark. Section V discusses additional issues for the proposed birthmark. Section VI concludes the paper.

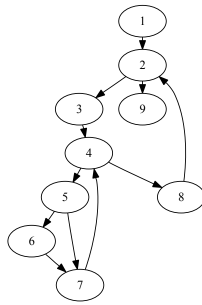
II. Related Work and Motivation

There are two approaches for software birthmarks: static and dynamic. In static birthmark schemes, static information is used as the defining characteristic of a program. Tamada et al. [3], [4] considered four structural characteristics of a Java program for use as the identifying characteristics. Myles and Collberg [5], [6] presented the k-gram based birthmark, which was a set of k-long sequences of opcodes in each program. Lim et al. [7] presented a method for identifying the instruction patterns of a Java program by analyzing the operand stack variations. Static birthmarks cannot reflect dynamic characteristics because they rely only on static characteristics that are visible superficially. Thus, such approaches are susceptible to program transformations. Park et al. [8] presented the API trace birthmark for Java programs that proceeds by constructing sets of static API traces, but the usage of this approach is limited to programs which use plentiful API calls.

Dynamic birthmarks, by contrast, collect the actual behaviors of program execution for given inputs. Tamada et al. [9] and Schuler et al. [10], [11] used the runtime behaviors of API calls in a program as birthmarks. Myles and Collberg [12], [6] used a graph representation compressed from a dynamic trace of a program as a birthmark. Such dynamic birthmarks are highly dependent on the given inputs

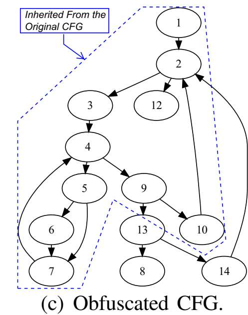
```java
public static void BubbleSort(int list[],
                              int size)
{
  int temp;
  for(int i = 1; i < size; i++)
  {
    for(int j = 0; j < size-1; j++ )
    {
      if(list[j] > list[j+1])
      {
        temp = list[j];
        list[j] = list[j+1];
        list[j+1] = temp;
      }
    }
  }
}
```

(a) A simple Java program (bubble sort).



Block	Offset	Instruction
v_1	0:	iconst_1
	1:	istore_3
v_2	2:	iload_3
	3:	iload_1
	4:	if_icmpge 65
v_3	7:	iconst_0
	8:	istore 4
v_4	10:	iload 4
	12:	iload_1
	13:	iconst_1
	14:	isub
	15:	if_icmpge 59
v_5	18:	aload_0
	19:	iload 4
	21:	iaload
	22:	aload_0
	23:	iload 4
	25:	iconst_1
	26:	iadd
	27:	iaload
	28:	if_icmple 53
v_6	31:	aload_0
	32:	iload 4
	34:	iaload
	35:	istore_2
	36:	aload_0
	37:	iload 4
	39:	aload_0
	40:	iload 4
	42:	iconst_1
	43:	iadd
	44:	iaload
	45:	iastore
	46:	aload_0
	47:	iload 4
	49:	iconst_1
	50:	iadd
	51:	iload_2
	52:	iastore
v_7	53:	iinc 4, 1
	56:	goto 10
v_8	59:	iinc 3, 1
	62:	goto 2
v_9	65:	return

(b) The CFG of Fig. 1(a).



(c) Obfuscated CFG.

Figure 1. A simple Java program and its control flow graph (bubble sort).

and the execution environments, so it is difficult to maintain runtime environments consistently. Moreover, dynamic approaches cannot cover all possible program paths. As a result, they are not suitable for characterizing the behaviors of all components, especially for interactive programs that require continuous user interactions. To compensate for the shortcomings of each approach, the control flow information of a program can be considered. Through the control flow information of a program, all the possible execution paths can be obtained at static time.

In Java [13], a source program is compiled into a portable binary format, called Java bytecodes. The bytecodes are machine codes which run directly on the Java Virtual Machine [14]. However, because the Java bytecodes are focused on portability, requiring plenty of high-level features, they inherit most of the structures from source programs, such as control-flows, class inheritance structures, and variable usage patterns. Hence, the reverse-engineering of Java programs prevails. On the other hand, because the Java bytecodes inherit most of the structural characteristics of source programs, their control flow graphs (CFG) can be used by several analysis techniques that are useful for the understanding of Java programs. Thus, the control flow information can be also used for identifying various features in Java programs.

Practically, open source software is frequently plagiarized in developing other software without regard to license policy [1]. Moreover, software theft is facilitated by modifying or obfuscating the software. However, such modification does not change the core control flow of programs to preserve the semantics of the programs. This notion may be applied to confirm the originality of programs.

Fig. 1(a) shows a simple Java program for the bubble sort algorithm. Fig. 1(b) shows the CFG and the basic blocks of the Java bytecodes. As shown in the example, the control structure in the source program is preserved in the CFG of the Java class file. Fig. 1(c) shows the CFG of the program transformed by the Smokescreen obfuscator [15]. The obfuscated CFG seems to be somewhat different from the original CFG. However, careful inspection reveals that the core control flows in the original CFG remain intact (see the dotted box). Hence, the control flow information of a program can be a good candidate to serve as a birthmark that identifies the originality of a program.

III. Birthmark Based on Control Flow Edges

A. Software Birthmark

Tamada et al. [3], [4] formally defined a *software birthmark* in terms of copy relation [4], which represents an originality relation between programs. The following definition and properties are restatements from [3], [4], [5].

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 $\mathcal{T}$. Then, we say f is *resilient* to $\mathcal{T}$ if $f(p) = f(p')$.

The *credibility* property requires a birthmark to discriminate different programs distinctively. Although two programs have the same functionality, the programs should have different birthmarks if they are developed independently. The *resilience* property specifies that the birthmark of a program must remain in its original form even after the program is altered by transformations. So, this property requires a software birthmark to be strong enough to endure semantics-preserving transformations.

B. Control Flow Edge

A *control flow graph* [16], [17] is a graph representing a program structure where nodes and edges describe basic blocks and possible control flows, respectively.

Definition 2 (Control Flow Graph and Edge): A control flow graph (CFG) of a Java method M is defined as (V, E), where V is a set of nodes which represent basic blocks in M, and $E \subseteq V \times V$ is a set of edges which represent possible control flows between basic blocks in M. For an edge $(v_i, v_j) \in E$, (v_i, v_j) is then called a control flow edge of the CFG.

In order to perform a control flow analysis on Java bytecodes, existing control flow analysis techniques should be extended to adapt to Java bytecodes. The procedure for constructing the CFG for a Java program is as follows. First, basic blocks are determined by finding the set of leaders that can be the entry point of each basic block. After all basic blocks are determined, the CFG is constructed by adding edges between the basic blocks, where control flows can reach.

In Java, exceptions change the control flow of a program. For example, if an exception occurs, the control jumps to the handler routine of the exception. In addition, obfuscators may change branch constructs with exception handling routines. To manage this situation, the exception edges in a Java program are added to the CFG. The exception table maintains the ranges of instructions managed by exception handlers and the addresses of the exception handler routines. So, exception edges are added from the basic blocks in the range of an exception to the target address of its handler routine.

From the CFG of a program, the control flow edges of the program can be identified. For a CFG $G = (V, E)$, an edge $(v_i, v_j) \in E$ is a control flow edge in the CFG. The control flow edges are abstract information of the program constructed from the CFG. The behavior of a control flow edge can be defined as follows:

Definition 3 (Behavior of Control Flow Edge): For a Java method M, let (v_i, v_j) be a control flow edge in M, and $bc(v_i)$ and $bc(v_j)$ be sequences of bytecodes in the basic blocks v_i and v_j, respectively. Then, the behavior of the control flow edge (v_i, v_j), denoted by $behavior(v_i, v_j)$, is defined as follows:

$$behavior(v_i, v_j) = concat(bc(v_i), bc(v_j)),$$

where $concat(x, y)$ denotes the concatenation of two sequences x and y.
The behavior of a control flow edge is the specific sequence of bytecodes executed by the control flow edge. Therefore, a behavior is constructed from the bytecodes stored in two basic blocks of the edge.

For example, for a program P, let Fig. 1(b) be the CFG of P. Then, the set of control flow edges, denoted by $Edge(P)$, is as follows:

$$\begin{aligned} Edge(P) \quad = \quad & \{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5), \\ & (v_5, v_6), (v_6, v_7), (v_5, v_7), (v_7, v_4), \\ & (v_4, v_8), (v_8, v_2), (v_2, v_9)\}. \end{aligned}$$

The behaviors of the control flow edges are the sequences of bytecodes contained in the corresponding basic blocks of each edge.

The CFG of a program can be used to estimate the dynamic characteristics of the program, that is, possible behaviors during the program execution, such as branches, loops, and sequences. Hence, a CFG is an effective structure for characterizing an individual program. However, a CFG assumes the form of a graph, which is difficult to manipulate or compare. For this reason, a simple representation for a CFG is necessary to abstract behaviors of a program. If every node of the CFG of a program is uniquely distinguished from every other, the CFG can be fully reconstructed from the set of control flow edges. Therefore, the set of control flow edges of a program is another representation of the CFG of the program. Thus, the set can be used as characteristics for identifying the program.

C. The Proposed Birthmark

For a complete 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 4 (CFE-Based Birthmark): For a Java program P, let $M_1, \cdots, M_n$ be methods in P. Let $G(V_i, E_i)$ be the CFG of the method M_i. The set of behaviors of all the control flow edges in M_i,

$$E_{method}(M_i) = \{behavior(v_j, v_k) | (v_j, v_k) \in E_i\},$$

is called the CFE-based birthmark of method M_i. The union of the CFE-based birthmarks of all methods in P,

$$E(P) = \bigcup_{i=1}^{n} E_{method}(M_i),$$

is called the CFE-based birthmark of program P.

For two programs P and Q, let $E(P)$ and $E(Q)$ be birthmarks of P and Q, respectively. P is then suspected to be copied from Q (and vice versa), if $E(P) \simeq E(Q)$, that is, $\mathbf{sim}(E(P), E(Q)) \geq 1 - \epsilon$, where $\mathbf{sim}$ is the similarity measure between two birthmarks, and ϵ is a threshold value for identifying software theft.

D. Matching Two Behaviors

To measure a similarity between CFE-based birthmarks, the behaviors in two birthmarks should be compared. The behavior of a control flow edge represents the possible sequence of bytecodes executed along the edge. Thus, similar behaviors imply that the behaviors perform similar tasks in their programs. The similarity between two behaviors can be intuitively calculated from the longest common subsequence (LCS) [18], and the length of the LCS is used as a *matching point* between two behaviors.

Fig. 2(a) shows an example of two behaviors a and b, which are represented by sequences of bytecodes of

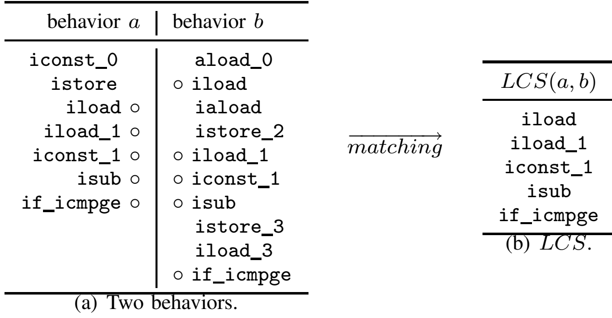


Figure 2. Sample behaviors and LCS.

the control flow edges. The $\circ$ denotes the matched bytecodes by LCS matching. Fig. 2(b) shows the LCS between two behaviors a and b. So, $LCS(a,b)$ is (`iload`, `iload_1`, `iconst_1`, `isub`, `if_icmpge`), and its length is $|LCS(a,b)| = 5$.

Definition 5 (Matching Point of Two Behaviors): Let a and b be behaviors in two birthmarks $E(P)$ and $E(Q)$, respectively. Let $LCS(a,b)$ be the longest common subsequence between a and b. The *matching point* of two behaviors a and b is then defined as follows:

$$mp(a,b) = \begin{cases} |LCS| & \text{if } \alpha \times |LCS| > \frac{|a|+|b|}{2}, \\ 0 & \text{otherwise,} \end{cases}$$

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

Although two behaviors perform different tasks, they may contain a few identical bytecodes. The matching coefficient α is necessary to clearly distinguish one behavior from the other. As the matching coefficient α increases, the matching condition is weakened, leading to a positive matching point of behaviors. Therefore, as the matching coefficient α increases, the two behaviors are more probable to be matched with a positive matching point. The coefficient can be used to achieve a reasonable trade-off between credibility and resilience. We evaluated the matching coefficient with several values and decided that $\alpha = 3$ was a reasonable trade-off between credibility and resilience for comparing behaviors [7].

E. Comparing Birthmarks

To compare two birthmarks, every element of one birthmark is matched with the parallel elements in the other. Each of the matching points between behaviors in the two birthmarks shows an absolute degree of similarity between the behaviors, and the points constitute a *matching matrix* between the two birthmarks. For example, if $E(P) = \{a_1, a_2, \cdots, a_n\}$ and $E(Q) = \{b_1, b_2, \cdots, b_m\}$ are birthmarks, then the matching matrix of $E(P)$ and $E(Q)$ is organized as follows:

$$M(E(P),E(Q)) = \begin{pmatrix} mp(a_1,b_1) & \dots & mp(a_1,b_m) \\ \vdots & \ddots & \vdots \\ mp(a_n,b_1) & \dots & mp(a_n,b_m) \end{pmatrix}.$$

From the matching matrix, the overall similarity can be obtained by finding similar pairs of behaviors from each birthmark. This is solved by a search problem for finding the matched pairs that maximize the sum of their matching points. A greedy algorithm can find the set of $\max(n,m)$ matched pairs in $O(n^3)$ by selecting pairs of behaviors in the order of the matching point. Such a set of pairs is called the *matching set* between two birthmarks. This set represents the pairs of feasibly most similar behaviors among all the behaviors in the two birthmarks. From the matching set between birthmarks, the matching point of birthmarks is defined as follows.

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

$$\mathcal{P}(E(P),E(Q)) = \sum_{(a,b)\in \hat{M}(E(P),E(Q))} mp(a,b).$$

The *matching point between two birthmarks* is calculated by accumulating all the matching points of the pairs in the matching set of two birthmarks. If the matching point between two birthmarks is higher than the typical matching point between birthmarks, we can be convinced that the two programs share many sequences of bytecodes in common. Therefore, we can infer that one of the two programs is likely to have been copied from the other, because common sequences of bytecodes can be suspected to have been used by stealth.

Definition 7 (Similarity of Birthmark): Let P and Q be programs, and $E(P)$ and $E(Q)$ be birthmarks of P and Q, respectively. Then, the similarity of birthmarks $E(P)$ and $E(Q)$ is defined as follows:

$$\mathbf{sim}(E(P),E(Q)) = \frac{\mathcal{P}(E(P),E(Q))}{\min(\sum_{a\in E(P)}|a|, \sum_{b\in E(Q)}|b|)}.$$

The similarity of two birthmarks is calculated by normalizing the matching point of the birthmarks. So, the point is divided by the minimum of the sums of the behavior lengths of the two birthmarks. If two birthmarks are fully matched, the matching point between the two birthmarks becomes equal to the denominator. Hence, the resulting similarity ranges from 0 to 1 in proportion to the degree of similarity. This measure represents how many behaviors of the original program are contained in those of the suspected program. Thus, to detect whether some modules or parts of a program

Table I
THE SPECIFICATIONS OF BENCHMARK PROGRAMS.

Program	Version	Size (bytes)	# of total class files	# of included class files	# of behaviors Total	# of behaviors Average
Crimson	1.1.3	355,230	145	67	5,679	84.8
Piccolo	1.04	323,018	87	43	4,187	97.4
XP	0.5	150,562	88	28	3,566	127.4

are copied, the birthmark extracted from specified core modules may be more appropriate than from the complete program.

For example, let $E(P) = \{a_1, a_2, a_3, a_4\}$ and $E(Q) = \{b_1, b_2, b_3, b_4\}$ be birthmarks of programs P and Q, respectively. Let the sums of the behavior lengths in the two birthmarks be $\sum_{a \in E(P)} |a| = 60$ and $\sum_{b \in E(Q)} |b| = 80$, respectively. Let

$$M(E(P), E(Q)) = \begin{pmatrix} \mathbf{16} & 5 & 0 & 6 \\ 3 & 0 & \mathbf{15} & 6 \\ 2 & \mathbf{10} & 0 & 3 \\ 0 & 2 & 3 & \mathbf{9} \end{pmatrix}$$

be the matching matrix of $E(P)$ and $E(Q)$. From a search algorithm for matching behaviors, the matching set can be found as

$$\hat{M}(E(P), E(Q)) = \{(a_1, b_1), (a_2, b_3), (a_3, b_2), (a_4, b_4)\}.$$

Finally, the similarity between two birthmarks is calculated as follows:

$$\mathbf{sim}(E(P), E(Q)) = \frac{16 + 10 + 15 + 9}{\min(60, 80)} = 0.833.$$

The similarity between two birthmarks in this calculation is 83.3%. Such a high similarity can provide strong evidence in detecting a copied program.

IV. EXPERIMENTAL RESULTS

A. Preliminaries

In this section, the proposed birthmark is evaluated with respect to two properties required for a birthmark: credibility and resilience. Fig. 3 shows a brief procedure for birthmarking two Java programs 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 $\alpha = 3.0$.

We used three XML processors as benchmark programs, as shown in Table I. Small programs are less likely be the targets for software theft. In addition, even though the birthmarks of two small programs may be similar, it can remain ambiguous whether one is copied from the other or if the similarities are coincidental. Therefore, to clearly evaluate the proposed birthmark, we chose class files that contain more than 50 bytecodes. The numbers of total control flow edges ranged between 3,566 and 5,679, and the average numbers of control flow edges in a class file ranged between 84 and 127. For a comparative evaluation of the performance of the proposed birthmark, two previous approaches, the k-gram based [5], [6] and API Trace [8] birthmarks, were also evaluated.

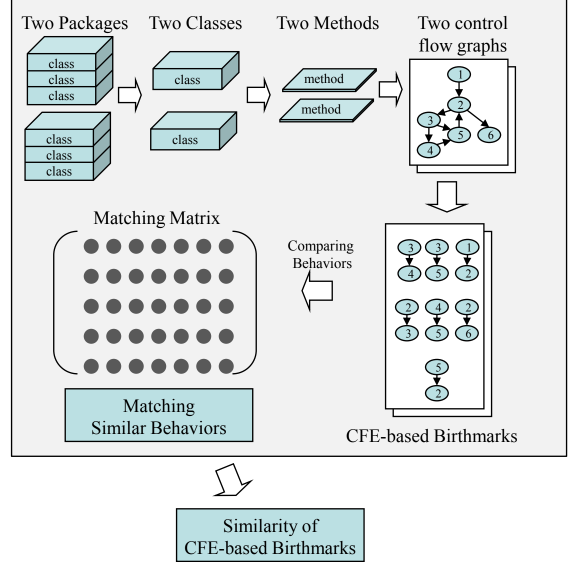


Figure 3. The procedure for the CFE-based birthmarking system.

B. Experiment 1: Credibility

In this experiment, the proposed birthmark is evaluated in terms of credibility. The credibility property means the following: even if two programs accomplish the same tasks, the birthmark must clearly discriminate the programs that are developed independently. For the XML processors shown in Table I, we extracted birthmarks according to the proposed and benchmark methods for a relative evaluation of how effectively the three birthmarks can be used to discriminate different programs. Because different programs are being compared with each other in this experiment, the more birthmark comparisons are ranged within low similarity values, the more credible a measure the birthmark can be evaluated as being. In addition, the birthmark should restrict the occurrences of false positives.

Table II shows the experimental results for the benchmark programs according to each birthmark. This table shows the number of pairs of class files according to similarity ranges.

Table II
THE COMPARISON OF RESULTS: DISTRIBUTIONS OF CREDIBILITY EXPERIMENT.

sim	Crimson vs. Piccolo			Crimson vs. XP			Piccolo vs. XP		
range	k-gram	API	CFE	k-gram	API	CFE	k-gram	API	CFE
0–9	2569	2788	1446	1790	1844	1179	1142	1182	767
10–19	275	35	696	84	10	405	60	9	257
20–29	16	17	350	2	9	174	2	8	106
30–39	2	5	210	0	5	76	0	3	43
40–49	2	4	93	0	1	27	0	1	18
50–59	1	9	48	0	5	10	0	0	10
60–69	1	3	13	0	0	4	0	0	2
70–79	5	2	7	0	0	1	0	0	1
80–89	3	2	7	0	0	0	0	0	0
90–99	3	0	8	0	0	0	0	0	0
= 100	4	16	3	0	2	0	0	1	0
Avg.	4.9	1.5	14.1	3.0	0.6	9.8	3.0	0.5	9.7
Min.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Max.	100	100	100	22.1	100	71.2	23.8	100	71.4
# FPs	0	5	0	0	2	0	0	1	0

It also shows the average, the minimum and the maximum similarities, as well as the numbers of false positives in the experiments. For the results of the three birthmarks, most of the different programs were distinguished with similarities lower than 60%. The k-gram based birthmark distinguished different programs most credibly. The proposed birthmark was also credible enough to distinguish different programs. With the API trace birthmark, several cases of false positives were found in programs where the numbers of API calls were insufficient. This was because short sequences of API calls were matched accidentally.

In the comparison of Crimson and Piccolo, however, several pairs were located in higher similarity ranges than others. We investigated the programs to find the reason why they had such high similarities. From the observation, we noticed that both programs included different versions of common modules: that is, `xml.parsers` from the Apache Software Foundation and `xml.sax` from Megginson Technologies Ltd. Thus, we were convinced that the proposed birthmark did not produce false positive results, but rather that it had accurately detected copied programs. So, in evaluating the credibility of a software birthmark, false positives should be carefully distinguished from true positives caused by the sharing of common modules.

C. Experiment 2: Resilience to Program Transformation

Because software may be modified or transformed to hide the fact of software theft, a birthmark must be sufficiently resilient to program transformations. In this experiment, the proposed birthmark is evaluated and compared with the previous approaches in terms of resilience to program transformation. Smokescreen and Jarg [19] were used to accomplish the program transformations. Smokescreen obfuscates an original Java program to make the program difficult to analyze, and Jarg optimizes an original program by renaming or removing the unnecessary parts in the program. We transformed three XML processor packages using Smokescreen and Jarg with their strongest transformation levels. Then, we extracted birthmarks from the original and the transformed programs. Next, we compared the birthmarks between all the pairs of the original programs and their transformed versions. For the resilience evaluation, a birthmark must be able to detect a software theft in spite of program transformations. Therefore, the more comparisons of birthmarks with high similarity values are located, the more resilient to program transformation the birthmark can be evaluated as being. Moreover, birthmarks should restrict the occurrences of false negatives.

Table III shows the experimental results according to each birthmark. This table shows the distributions of similarities for all the pairs of the original programs and their transformed versions, the average, the minimum and the maximum similarities, as well as the numbers of false negatives. The k-gram based birthmark was susceptible to obfuscation by Smokescreen, because the transformation directly changed the sequence of opcodes. The average similarities ranged between 51.8% and 62.7%. The API trace birthmark relies only on the static API traces of a program. As a result, the method is limited to application in programs that use API functions intensively. In the experiments on programs that rarely used API calls, the API trace birthmark produced false negative results. In addition, false positive results have to be carefully identified because there may be accidental matches of API calls. With the proposed birthmark, except for one pair of class files, the similarities were higher than 80%, and the distributions of similarities were the most even among the three approaches. From the results, the proposed birthmark was the most reliable measure for detecting copied programs in such modification environments. These results imply that control flow information can be an effective characteristic for identifying the originality of software.

For the pairs of class files with lower similarity in the results, we observed their bytecodes. Firstly, we noticed

Table III
THE COMPARISON OF RESULTS: DISTRIBUTIONS OF RESILIENCE EXPERIMENT.

sim range	Smokescreen									Jarg								
	Crimson			Piccolo			XP			Crimson			Piccolo			XP		
	k-gram	API	CFE	k-gram	API	CFE	k-gram	API	CFE	k-gram	API	CFE	k-gram	API	CFE	k-gram	API	CFE
0–9	0	7	0	0	1	0	0	6	0	0	6	0	0	1	0	0	6	0
10–19	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
20–29	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30–39	2	0	0	4	1	0	1	0	0	0	0	0	0	0	0	0	0	0
40–49	7	1	0	1	1	0	2	0	0	0	0	0	0	0	0	0	0	0
50–59	26	0	0	11	0	0	8	0	0	0	0	0	0	0	0	0	0	0
60–69	23	2	0	16	0	0	11	0	0	0	2	0	0	0	0	0	0	0
70–79	6	1	1	11	1	0	5	0	0	4	0	0	3	0	0	0	0	0
80–89	0	1	5	0	1	0	1	0	6	31	1	2	17	0	1	3	0	0
90–99	0	2	58	0	1	42	0	0	21	21	1	54	21	1	40	20	0	17
= 100	0	53	3	0	36	1	0	22	1	11	57	11	2	41	2	5	22	11
Avg.	57.8	87.1	95.3	61.9	92.0	96.5	62.7	78.5	93.3	90.0	89.7	97.3	89.5	97.6	96.2	94.6	78.5	98.8
Min.	21.9	0.0	78.5	30.0	0.0	90.9	39.4	0.0	83.5	73.4	0.0	83.4	77.7	0.0	89.3	86.7	0.0	93.3
Max.	76.3	100	100	75.9	100	100	86.2	100	100	100	100	100	100	100	100	100	100	100
# FNs	0	7	0	0	1	0	0	6	0	0	6	0	0	1	0	0	6	0

that the transformation in the evaluation decomposed a basic block into several parts by adding opaque predicates [20], [21] which were always evaluated as either `true` or `false`. So, in matching behaviors between two birthmarks, some behaviors in the decomposed parts cannot be fully matched with the behaviors of the original program's birthmark. Secondly, we noticed that the transformation changed the sequence of bytecodes by instruction reordering. For example, Smokescreen transforms the sequence of pairs of `load` and `store` instructions into the sequence of `store` instructions after `load` instructions. Because the proposed birthmark compares the order of bytecode sequences, the birthmark may be confused by code sequence reordering.

V. DISCUSSION

In the experiments presented in Section IV-B, different programs were compared with each other, and the distributions of resulting similarities were presented to evaluate the relative credibility of three birthmarks. In comparing the three approaches, the distributions of the CFE-based birthmark were located in somewhat higher ranges than those of the other approaches. This is mainly due to accidental matches of behaviors that appear frequently in conventional Java programs. Such frequent behaviors may not be discriminating characteristics of individual programs, but merely common characteristics. For more credible discrimination, such frequent behaviors need to be excluded from program birthmarks.

The quality of a software birthmark depends on the value of threshold ϵ, which provides a basis for deciding whether one program is copied from another. An inappropriate value of ϵ may cause a number of false positives or false negatives in detecting software thefts. The matching coefficient α is also related to the decision of proper threshold value. For more reliable results with our software birthmark, it is necessary to determine the threshold ϵ and the matching coefficient α through experiments using large repositories of real-world applications.

VI. CONCLUSION AND FUTURE WORK

In this paper, we proposed a static Java birthmark based on control flow edges. A control flow edge means the edge in the CFG of a program and represents the possible flow of instructions of program execution, namely, its behavior. Behaviors were matched by using the LCS algorithm, and the similarity between birthmarks was calculated by finding the set of the most similar pairs of behaviors in two birthmarks.

We evaluated and compared the proposed birthmark with two previous approaches in terms of credibility and resilience. The experimental results showed that the proposed birthmark was credible and resilient enough to detect cases of software theft. Of the three approaches compared, the proposed birthmark was the most reliable measure to detect copied programs. Because the CFE-based birthmark can reflect the algorithmic structure of a program, the birthmark can also be used to detect stolen core modules. For future work, we plan to refine the birthmark by adjusting the matching algorithm to improve resilience. We also plan to conduct experiments in large code repositories for tuning the parameters, such as the threshold value ϵ and the matching coefficient α. It is expected that this will provide a more reliable method for detecting copied programs in real-world environments.

ACKNOWLEDGMENT

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

REFERENCES

[1] "The gpl-violations.org project," http://gpl-violations.org/.

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

[3] H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto, "Design and evaluation of birthmarks for detecting theft of java programs," in *Proceedings of IASTED International Conference on Software Engineering (IASTEDSE2004)*, February 2004, pp. 569–575.

[4] H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto, "Java birthmark –detecting the software theft," *IEICE Transactions on Information and Systems*, vol. E88-D, no. 9, pp. 2148–2158, 2005.

[5] G. Myles and C. Collberg, "k-gram based software birthmarks," in *Proceedings of the 2005 ACM Symposium on Applied Computing*, 2005, pp. 314–318.

[6] G. M. Myles, "Software theft detection through program identification," Ph.D. dissertation, Department of Computer Science, The University of Arizona, 2006.

[7] H. Lim, H. Park, S. Choi, and T. Han, "Detecting theft of java applications via a static birthmark based on weighted stack patterns," *IEICE Transactions on Information and Systems*, vol. E91-D, no. 9, pp. 2323–2332, September 2008.

[8] H. Park, S. Choi, H. Lim, and T. Han, "Detecting java theft based on static api trace birthmark," in *International Workshop on Security (IWSEC)*, ser. Lecture Notes in Computer Science, vol. 5312. Springer, 2008, pp. 121–135.

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

[10] D. Schuler and V. Dallmeier, "Detecting software theft with api call sequence sets," in *Workshop Software Reengineering (WSR 2006)*, 2006.

[11] D. Schuler, V. Dallmeier, and C. Lindig, "A dynamic birthmark for java," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 274–283.

[12] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," in *Information Security, 7th International Conference (ISC 2004), LNCS 3225*, 2004, pp. 404–415.

[13] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 2nd ed. Addison-Wesley, June 2000.

[14] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed. Addison-Wesley, April 1999.

[15] "Smokescreen java obfuscator," http://www.leesw.com/smokescreen/.

[16] R. Wilhelm and D. Maurer, *Compiler Design*. Addison-Wesley, 1995.

[17] J. Zhao, "Analyzing control flow in java bytecode," in *Proc. 16th Conference of Japan Society for Software Science and Technology*, 1999, pp. 313–316.

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

[19] "Java archive grinder (jarg)," http://sourceforge.net/projects/jarg/.

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

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