Detecting Java Theft Based on Static API Trace Birthmark[⋆]

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

Division of Computer Science
KAIST
Gwahangno 335, Yuseong-gu, Daejeon 305-701, Republic of Korea
{hwpark,swchoi,hilim,han}@pllab.kaist.ac.kr

Abstract. Software birthmark is the inherent program characteristics that can identify a program. In this paper, we propose a static API trace birthmark to detect Java theft. Because the API traces can reflect the behavior of a program, our birthmark is more resilient than the existing static birthmarks. Because the API traces are extracted by static analysis, they can be applied to library programs which earlier dynamic birthmarks cannot handle properly. We evaluate the proposed birthmark in terms of credibility and resilience. Experimental results show that our birthmark can detect common library modules of two packages while other birthmarks fail to detect.

1 Introduction

Recently many programs are under development in the form of open source projects. Open source programs are allowed to modify or distribute program codes under the certain types of software licenses. The most popular software license is the GNU Public License (GPL). Under the GPL, program codes are freely used, while the software using the original codes should also be under the GPL. Open source softwares such as MySQL have different licenses to support software firms. We call the license types as multi-licenses. Under such licenses the GPL is used for open source projects, and commercial licenses are applied to the commercial softwares of which the source codes are not distributed under the GPL if the license fees are paid.

Several reports such as the IBM and SCO battle, where the SCO claims that IBM illegally used open source codes in proprietary programs, confirm that the violations of software licenses are common. These software thefts cause much damage to open source developers and companies. There are plagiarism detectors such as YAP, JPlag, and MOSS which are commonly used to discover code theft [1][2][3]. The plagiarism detectors target to the source codes of programs. Because most commercial programs distribute binary executables only, binary code analysis techniques are necessary. Software birthmarking techniques can be used to detect code theft of binary executables. A software birthmark is a combination of unique characteristics from a program. For

[⋆] This work was supported by the Korea Science and Engineering Foundation(KOSEF) grant funded by the Korea government(MEST) (No. R01-2008-000-11856-0).

© Springer-Verlag Berlin Heidelberg 2008

example, the strings extracted from a binary or binary checksums are candidates of a software birthmark.

To detect code theft, two birthmarks are compared to get the similarity between two binary programs. If the similarity is sufficiently high, we can conclude that one program is a copy of the other. Software watermark is similar to software birthmark in that they both can be used to detect code theft. The difference is that for a software birthmark we extract inherent characteristics from the software itself while for a software watermark we extract pre-embedded fingerprints from the software. Software watermarks provide a certain evidence of code theft by extracted copyright information, while software birthmarks provide the possibility of code theft by similarity between programs. However, in some cases, software birthmarks are better than software watermarks due to the highly restricted computing power and memory size.

Software birthmarks can be classified into static and dynamic. Static birthmarks are extracted from a program itself without execution. Dynamic birthmarks are extracted from observable program behaviors during the execution of a program. Static birthmarks can cover the whole program paths, while dynamic birthmarks depend on the run-time trace of a program. Dynamic birthmarks are known to be more resilient to program transformations such as code obfuscation than static birthmarks.

In this research, we propose a static API trace birthmark for Java programs. The static API trace birthmark is the set of all possible run-time API traces of each Java method. Unlike existing birthmarks, the static API trace birthmark does not simply extract adjacent API sequences but analyzes the control flow of methods and generates the possible run-time API traces. Because the API traces can reflect the behavior of a program, our birthmark is more resilient than other static birthmarks.

We evaluate the proposed birthmark in terms of two criteria: credibility and resilience. The credibility of birthmark is the ability to distinguish different programs. The resilience of birthmark is the ability to resist against program transformations. Experimental results show that our birthmark can detect common library modules of two packages while the other birthmarks fail to detect.

2 Related Work

Software birthmarks can be classified into static and dynamic. A static birthmark is extracted from a program itself by static analysis. A dynamic birthmark is extracted from observable run-time behaviors of a program. The advantage of static birthmarks is that they can cover the whole program paths, while dynamic birthmarks can contain only the run-time trace of a program. One serious problem with dynamic birthmarks is that they vary depending on the inputs and the run-time environments. Dynamic birthmarks are known to be more resilient to program transformations such as code obfuscation than static birthmarks.

H. Tamada et al. suggested a static birthmark for Java [4]. Their birthmark is defined with the combination of four features: constant values assigned to fields, the sequence of method calls following the order of instructions in the class files, class inheritance hierarchy, and used class information. Their birthmark is resilient to code obfuscations but it cannot compare algorithms in the methods because it only compares externally observable features.

G. Myles et al. suggested the k-gram birthmark for Java [5]. The k-gram birthmark is the k length sequence of bytecode instructions. It is highly credible, but frail to program transformations.

G. Myles et al. suggested the Whole Program Path(WPP)-birthmark for Java [6]. The WPP birthmark records run-time instruction traces and constructs a dynamic control flow graph. Because the WPP birthmark records executed instructions only, it is resilient to some obfuscation transformations such as insertion of opaque predicates [7] that inserts garbage instructions. However, because the WPP birthmark is dynamic, it can be changed depending on inputs and environments.

D. Schuler et al. suggested a dynamic API birthmark for Java [8]. Their dynamic API birthmark is a set of API call traces per object. The collection of API calls per object makes this birthmark credible than the earlier dynamic API birthmark for Windows [9]. The limitation of Schuler's birthmark is that it can handle only applications that call API frequently.

S. Choi et al. suggested a static API birthmark for Windows [10]. Their static API birthmark is a set of possible API calls which are extracted by statically analyzing disassembled codes. It can be applied only to the Windows API applications.

3 A Static API Trace Birthmark

A software birthmark of a program means the inherent characteristics that can identify the program. A software birthmarking system is the system that provides two functions for the birthmark: extraction and comparison. In this section, we propose a static API trace birthmark and suggest the birthmark extraction method and the comparison method.

3.1 The Definition of the Static API Trace Birthmark

Instead of comparing two control flow graphs directly, we compare two sets of API traces because the traces reflect the run-time behaviors of programs compared to the control flow graph comparison. The static API trace birthmark considers API traces of a method as the essential characteristics of the method. Definition 1 explains an API flow graph. Definition 2 explains a static API trace.

Definition 1. *(API flow graph) Given a control flow graph $G = (V, E, v_s, V_e)$, where v_s is a start node and V_e is a set of exit nodes, $G' = (V', E', v_s, V_e)$ is an API flow graph if G' satisfies following two conditions:*

Condition 1

$$V' = V - \{v \in V | v \text{ does not have API call}\} \cup \{v_s\} \cup V_e,$$

Condition 2

$$E' = \{(v_f, v_t) | \exists \text{a path } v_f v_1 \cdots v_n v_t \wedge v_f \in V' \wedge v_t \in V' \wedge v_1, \cdots, v_n \notin V'\}.$$

Definition 2. *(Static API Trace) Given an API flow graph $G' = (V', E', v_s, V_e)$, a static API trace of a method is a sequence of API calls from v_s to V_e.*

A static API trace may match with a dynamic trace of the method. The static analysis to get static API traces may possibly generate abstract traces which contain all dynamic traces. In previous definition, however, we define the static API trace as a concrete trace rather than as an abstract trace. Although concrete traces cannot cover all possible dynamic traces, a comparison of two sets of concrete traces gives us sufficient accuracy because we are not comparing concrete traces with dynamic traces.

Definition 3. *(Edge Covering Static API Trace Set) An edge covering static API trace set of a method is a set of static API traces where the union of the static API traces covers all the edges of the API flow graph of the method.*

To incorporate all possible execution traces, our birthmark requires edge covering.

Definition 4. *(Static API Trace Birthmark) The static API trace birthmark is the minimum size edge covering static API trace set.*

3.2 Static API Trace Generation

We generate the control flow graphs from Java bytecodes. One peculiarity when extracting birthmarks from Java bytecodes is that Java bytecodes contain subroutine call instructions such as `jsr` and `jsr_w` and exception handling.

To generate a control flow graph for the `jsr` and `jsr_w` instructions, the `ret` instructions should be linked to the next instruction of the `jsr` instruction. Figure 1 shows an example of the tree generation of subroutine instructions `jsr` and `ret`. The `jsr` instruction stores the return address which is next to the executed `jsr` instruction. When

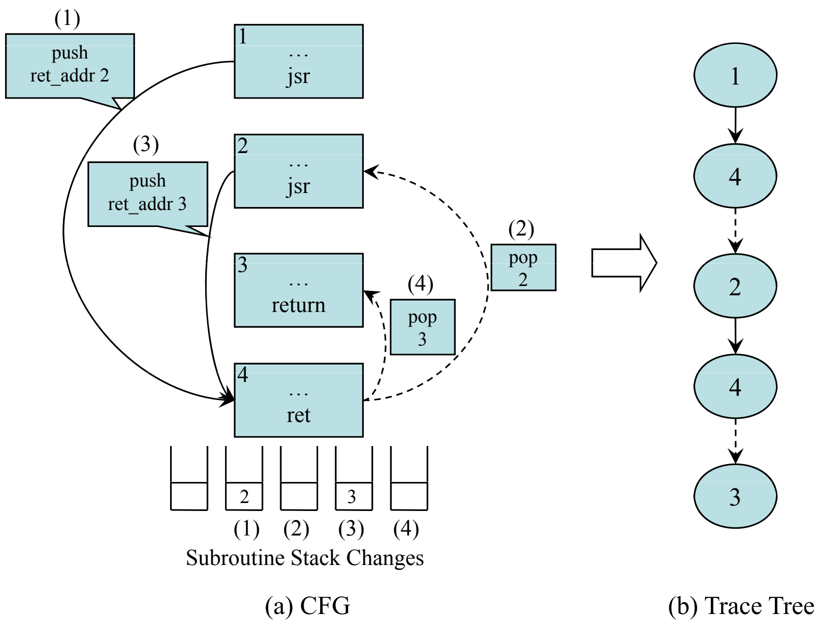

Fig. 1. An example to show the tree generation for `jsr` instruction

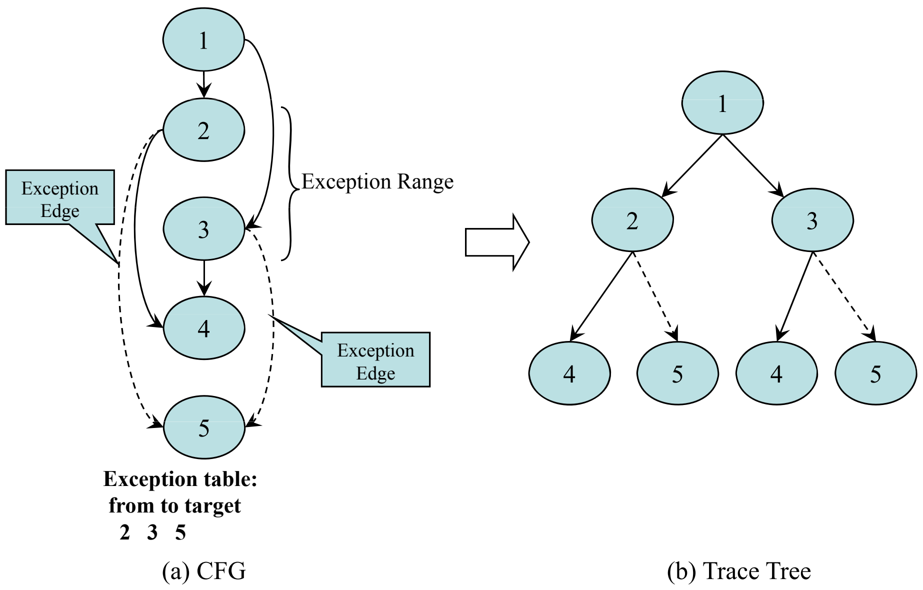

Fig. 2. An example to show the tree generation for exceptions

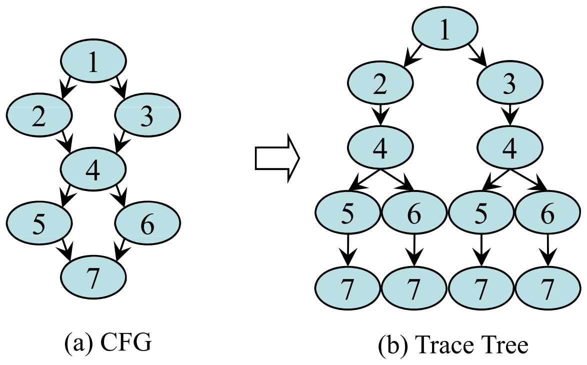

Fig. 3. An example to show the tree generation for branches

a `ret` is executed, the control flow goes to the address stored by the `jsr` instruction. We use a subroutine stack for `jsr` and `ret` instructions to build the correct tree.

For the exception handling, all instructions contained in the exception range should be linked to the exception handler. Because the number of edges generated by exceptions is huge, we add only one edge from the end of each block in the exception range to the corresponding exception handler. Figure 2 explains how a control flow graph is constructed for a exception table in Java bytecodes. Block 2 and block 3 are in the exception range. Two exception edges, which are represented as dotted lines, are linked from the blocks in the exception range to the exception handler block 5.

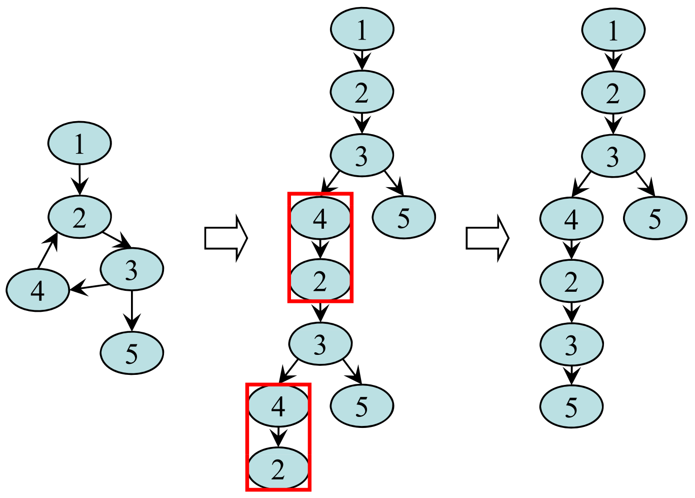

Fig. 4. An example to show the tree generation for loops

After generating the control flow graph, we extract the API flow graph. In order to make the API flow graph, all nodes which have no API calls must be removed, and we add new edges from the sources of incoming edges of the removed nodes to the targets of outgoing edges. After construction of the API flow graph, we can generate API traces. To get all possible API traces, we first construct an edge covering tree. By traversing the edge covering tree we get the edge covering traces. Figure 3 shows how the branches and their descendant nodes are duplicated. Figure 4 shows how to handle loops of API flow graphs. Because a loop can generate infinite API traces during execution, we limit the number of iterations of the loop to one cycle. One iteration cycle is sufficient to generate an edge covering API trace.

Once the API trace tree is fully constructed, traversing the whole tree generates an edge covering static API trace set.

3.3 Similarity Calculation via Semi-global Alignment

To compare two traces extracted from two programs, we utilize sequence alignment algorithms which are frequently used in Bioinformatics to compare DNA sequences. Well known sequence alignment algorithms are global alignment, local alignment, and semi-global alignment algorithms. The global alignment algorithm calculates the similarity using the whole sequences [11]. The local alignment algorithm is used to find maximum subsequence matching [12]. The semi-global alignment algorithm is based on the global alignment algorithm but compensates the score penalty caused by heading and tailing mismatches [13].

We evaluated above three alignment algorithms for our trace birthmark of Java methods. The local alignment algorithm has weakness that the similarity is sensitive to the subsequence of whole traces, which means that the result of local-alignment algorithm is not credible. The global alignment algorithm has weakness that the similarity is sensitive to the length of the traces; if one sequence is contained in the other sequence and the difference of the lengths is high, the similarity is relatively low. We chose the semi-global alignment algorithm because it can handle the containment problem and it is more credible than the local alignment algorithm.

Definition 5. *(Global Alignment) Let the original trace be T_1, copy trace be T_2, $\sigma(i,j)$ be the match score between $T_1[i]$(i'th element of T_1) and $T_2[j]$(j'th element of T_2), and gap be the penalty score incurred by the insertion of gaps. The optimal global alignment $G(i,j)$ maximizing the alignment score of $T_1[1..i]$ and $T_2[1..j]$ is defined as*

$$G(i,j) = \max \begin{cases} G(i-1,j-1) + \sigma(i,j) \\ G(i-1,j) + gap \\ G(i,j-1) + gap. \end{cases}$$

The match, mismatch and gap penalties are

$$\sigma(i,j) = \begin{cases} 1 & \textit{if } T_1[i] = T_2[j] \\ -1 & \textit{if } T_1[i] \neq T_2[j] \end{cases}$$
$$gap = -1.$$

The credibility of birthmark increases in proportional to the mismatch and gap penalty. To increase the resilience of birthmark, we can decrease the penalty.

Definition 6. *(Semi-global Alignment) Given a global alignment $G(i,j)$, a semi-global alignment $SG(i,j)$ is defined as*

$$SG(i,j) = \max\{G(i,1), G(i,2), \ldots, G(i,j), 0\}$$
$$\textit{where } G(0,0) = G(0,1), \ldots, G(0,j) = 0.$$

In contrast to the global alignment, the semi-global alignment does not cut scores by heading and tailing penalties.

Definition 7. *(Trace Similarity) Given traces $T_1[1..i]$ and $T_2[1..j]$, the trace similarity between T_1 and T_2 is defined as*

$$T_{sim}(T_1,T_2) = \max(\frac{SG(i,j)}{|T_1|}, \frac{SG(j,i)}{|T_2|}).$$

In Definition 7, the trace similarity considers both cases where T_1 is a copy and T_2 is original, and vice versa.

Definition 8. *(Method Similarity) Given sets of traces $TS_1[1\ldots m]$ in method M and $TS_2[1\ldots n]$ in method N, the method similarity between M and N is defined as*

$$M_{sim}(M,N) = \max(\frac{sum_{row}(M,N)}{sum_{trace}(M)}, \frac{sum_{col}(M,N)}{sum_{trace}(N)})$$

where

$$
\begin{aligned}
sum_{row}(M,N) &= \sum_{i=1}^{m} row_{max}(i),\\
row_{max}(i) &= \max(SG(i,0),\ldots,SG(i,n)),\\
sum_{col}(M,N) &= \sum_{j=1}^{n} col_{max}(j),\\
col_{max}(j) &= \max(SG(0,j),\ldots,SG(m,j)),\\
sum_{trace}(M) &= \sum_{i=1}^{m} |TS_1[i]|,\\
sum_{trace}(N) &= \sum_{j=1}^{n} |TS_2[j]|.
\end{aligned}
$$

Table 1. An example SG matrix computed by the semi-global alignment algorithm

	$TS_2[1]$ $(\lvert TS_2[1]\rvert = 8)$	$TS_2[2]$ $(\lvert TS_2[2]\rvert = 7)$	$TS_2[3]$ $(\lvert TS_2[3]\rvert = 5)$
$TS_1[1]$ $(\lvert TS_1[1]\rvert = 6)$	2	3	4
$TS_1[2]$ $(\lvert TS_1[2]\rvert = 7)$	5	6	1

Table 1 is an example of SG matrix computed by the semi-global alignment algorithm. The $T_{sim}(TS_1[1], TS_2[1])$ is computed by Definition 7 as

$$T_{sim}(TS_1[1], TS_2[1]) = \max(\frac{2}{6}, \frac{2}{8}) = \frac{1}{3}.$$

The $M_{sim}(M,N)$ is computed by Definition 8 as

$$
\begin{aligned}
M_{sim}(M,N) &= \max(\frac{4+6}{6+7}, \frac{5+6+4}{8+7+5})\\
&= \max(\frac{10}{13}, \frac{15}{20}) = \frac{10}{13}.
\end{aligned}
$$

Definition 9. *(Class Similarity) Given two classes C, D and two method sets $M[1\ldots m]$ in C, $N[1\ldots n]$ in D, the class similarity between C and D is defined as*

$$C_{sim}(C,D) = \max(\frac{\sum_{i=1}^{m} map(M[i])}{\sum_{i=1}^{m} sum_{trace}(M[i])}, \frac{\sum_{j=1}^{n} map(N[j])}{\sum_{j=1}^{n} sum_{trace}(N[j])})$$

where

$$map(M[i]) = \max(sum_{row}(M[i], N[1]), \ldots, sum_{row}(M[i], N[n])),$$
$$map(N[j]) = \max(sum_{col}(M[1], N[j]), \ldots, sum_{col}(M[m], N[j])).$$

Definition 10. *(Package Similarity) Given two packages P and Q, two class sets $C[1 \ldots m]$ in P and $D[1 \ldots n]$ in Q, and method sets $M_i[1 \ldots c_i]$ in $C[i]$ and $N_j[1 \ldots d_j]$ in $D[j]$, the package similarity between P and Q is defined as*

$$P_{sim}(P, Q) = \max\left(\frac{\sum_i \sum_k map(M_i[k])}{\sum_i \sum_k sum_{trace}(M_i[k])}, \frac{\sum_j \sum_k map(N_j[k])}{\sum_j \sum_k sum_{trace}(N_j[k])}\right).$$

4 Implementation

Figure 5 shows the architecture of the proposed static API trace birthmarking system. The static API trace birthmarking system computes the package similarity between two packages. To calculate the similarity between two Java packages, we need to calculate similarities of each class in packages. In order to calculate similarity of each class,

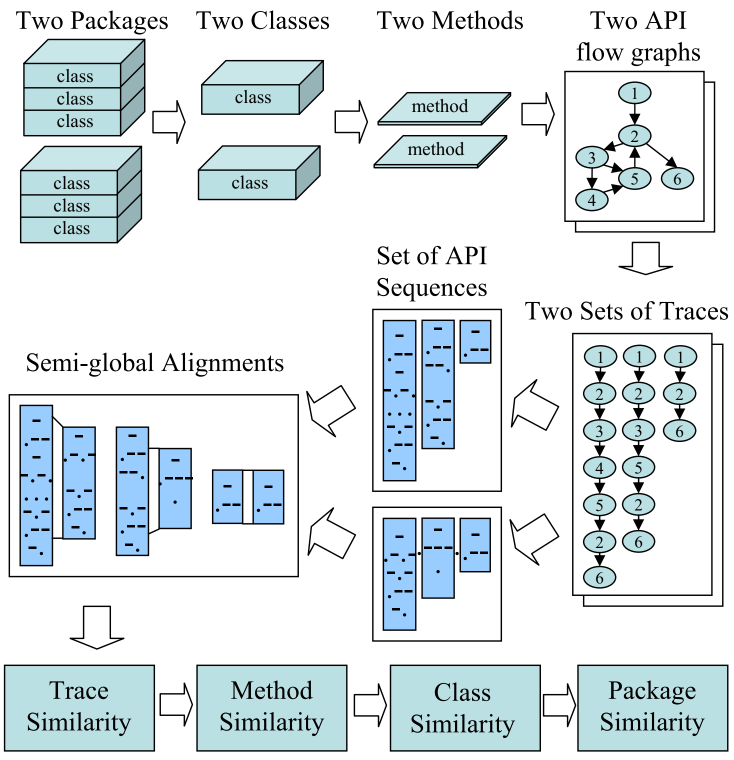

Fig. 5. Static API Trace Birthmarking System

we have to calculate the similarities of each method in classes. To calculate the similarity between two methods, two API flow graphs are generated from two methods. Nodes of the API flow graphs contain the Java API calls. All branch instructions except subroutine calls are removed from the API flow graphs. The edges are constructed by the method explained in Section 3.2. With the API flow graphs, static API trace trees are built. By traversing the trees, static API traces are generated. The API traces contain only standard Java API calls. Finally, the API trace sets are compared to get the similarity between two methods using the semi-global alignment algorithm. Because the matrix is computed using dynamic programming, the time complexity of the semi-global alignment algorithm is $O(m \cdot n)$ where m, n are the sizes of the two API traces. If we have API trace similarities, we can get method similarities, class similarities, and package similarities by Definitions 8, 9, and 10, respectively.

5 Evaluation

We evaluate the proposed birthmark by two criteria of credibility and resilience. To show the credibility and the resilience of our birthmark, we selected the benchmark programs in Table 2. These benchmark programs are well known open source XML processors which are used to evaluate the dynamic Java birthmark [8]. We also compare our birthmark with the existing static Java birthmarks: the Tamada birthmark [4] and the k-gram birthmark [5]. For the experiments, we used our birthmarking system for the static API trace birthmark and a Java birthmark toolkit *Stigmata*[1] for the Tamada birthmark and the k-gram birthmark.

As shown in Table 2, we do not include classes with API call sequences shorter than 3. We consider that such small methods hardly represent inherent characteristics of classes.

The credibility of birthmarks is measured by the similarities among the different programs. The resilience of birthmarks is measured by the similarities between the original programs and the transformed programs. For the code transformation, we used a Java obfuscator *Smokescreen*[2], and Java compilers *javac* and *jikes*[3].

Table 2. Benchmark programs

	Version	Size of Original Classes (bytes)	Number of Excluding Classes	Number of Including Classes	Size of Including Package (bytes)	Size Ratio of Including Package / Original Package
Aelfred	7.0	60,608	5	2	55,036	0.91
Crimson	1.1.3	355,230	88	57	274,735	0.77
Piccolo	1.04	323,108	49	38	227,729	0.71
XP	0.5	150,562	77	11	83,683	0.56

[1] Stigmata: Java birthmark toolkit. Available at http://stigmata.sourceforge.jp/

[2] Smokescreen Java Obfuscator. Available at http://www.leesw.com/smokescreen/

[3] Jikes Java Compiler. Available at http://jikes.sourceforge.net/

5.1 Resilience

Tables 3 and 4 show the similarities of three birthmarks between original programs and obfuscated programs. The benchmark programs are transformed with *Smokescreen* and *Jikes*, and the resulting packages are compared to the original packages. The average similarities among three birthmarks confirm that our birthmark is more resilient than the others. We inspected the transformed classes to find the reasons.

The first reason is the change of instruction order. We observed that the order of bytecodes is changed by the *SmokeScreen* obfuscator. If the original bytecodes have three load and store instruction pairs, the Smokescreen changes these pairs to three load and three store instructions. Because k-gram birthmark is vulnerable to this kind of modification, the average similarity of k-gram birthmark is lower than the others.

The second reason is the change of branch instruction and branch target. We observed that branch statements are compiled into different but semantically equivalent bytecode instructions by the *Jikes* compiler. For example, `ifgt`, which means greater than, generated by the *javac* compiler is compiled into `ifle` and the order of the following statements is exchanged by the *Jikes* compiler. The sequence of method call of the Tamada birthmark is vulnerable to the program transformations which replace instructions with the semantically equivalent instructions. In addition, Tamada birthmark cannot generate correct API call sequences in cases when the true branch and the false branch are swapped and equivalent instructions such as `ifgt` and `ifle` are used, because the Tamada birthmark follows the physically adjacent API calls rather than the control flow edges. Our static API trace birthmark can handle this type of problem because it compares two branches simultaneously by generating all API traces. Hence, our static API trace birthmark is highly resilient.

Table 3. Similarity of 3 birthmarks to evaluate the resilience to Smokescreen obfuscator

	Tamada	k-gram	API Trace
Aelfred	0.758	0.671	0.980
Crimson	0.689	0.563	0.996
Piccolo	0.733	0.614	0.998
XP	0.720	0.589	1.000
Average	0.725	0.609	0.994

Table 4. Similarity of 3 birthmarks to evaluate the resilience to Jikes compiler

	Tamada	k-gram	API Trace
Aelfred	0.857	0.892	0.975
Crimson	0.839	0.871	0.998
Piccolo	0.832	0.891	0.998
XP	0.920	0.894	0.984
Average	0.862	0.887	0.989

5.2 Credibility

Tables 5, 6, and 7 show the similarities among benchmark programs to evaluate the credibility of our static API trace birthmark.

To be a credible birthmark, the similarity between different programs should be near 0. The similarity averages of the Tamada, k-gram, and our API Trace birthmarks are 0.392, 0.230, and 0.111, respectively. The averages show that our API trace birthmark is more credible than other static Java birthmarks. However, we observed that the similarities between the *Crimson* and the *Piccolo* of three birthmarks are the biggest value among the similarities in each table. We inspected the source codes of the *Crimson* and the *Piccolo*. We found that both programs include `xml.parsers` from the Apache software foundation and `xml.sax` from Megginson Technologies. The high similarities between two programs are due to the common modules.

5.3 Module Theft

Suppose that a malicious developer include other developer's module illicitly in his program. In most cases, he tries to hide the stolen code by code obfuscation. Good birthmarks must detect code theft in this case. We evaluated module theft by comparing obfuscated versions of the *Piccolo* and the *Crimson*.

Table 8 shows the number of matched classes between the obfuscated Crimson and the Piccolo. The last row represents the number of perfectly matched classes between

Table 5. Similarity of Tamada birthmark to evaluate the credibility

	Aelfred	Crimson	Piccolo	XP
Aelfred	1.000	0.281	0.445	0.363
Crimson	-	1.000	**0.575**	0.306
Piccolo	-	-	1.000	0.380
XP	-	-	-	1.000

Table 6. Similarity of k-gram birthmark to evaluate the credibility

	Aelfred	Crimson	Piccolo	XP
Aelfred	1.000	0.261	0.224	0.184
Crimson	-	1.000	**0.456**	0.131
Piccolo	-	-	1.000	0.126
XP	-	-	-	1.000

Table 7. Similarity of our static API Trace birthmark to evaluate the credibility

	Aelfred	Crimson	Piccolo	XP
Aelfred	1.000	0.018	0.013	0.110
Crimson	-	1.000	**0.341**	0.115
Piccolo	-	-	1.000	0.068
XP	-	-	-	1.000

Table 8. The numbers of matched classes between the Piccolo and the obfuscated Crimson

Similarity interval	Number of classes included in each similarity interval					
	Tamada		k-gram		our API Trace	
	Original	Obfuscated	Original	Obfuscated	Original	Obfuscated
$0.0 \leq C_{sim} < 0.1$	0	0	4	10	7	10
$0.1 \leq C_{sim} < 0.2$	0	0	12	12	4	3
$0.2 \leq C_{sim} < 0.3$	11	12	6	1	0	1
$0.3 \leq C_{sim} < 0.4$	4	9	0	1	3	2
$0.4 \leq C_{sim} < 0.5$	7	3	1	3	1	1
$0.5 \leq C_{sim} < 0.6$	0	2	0	7	5	3
$0.6 \leq C_{sim} < 0.7$	2	3	1	2	2	2
$0.7 \leq C_{sim} < 0.8$	2	9	3	1	2	2
$0.8 \leq C_{sim} < 0.9$	2	0	2	1	2	2
$0.9 \leq C_{sim} < 1.0$	1	0	3	0	1	1
C_{sim}**=1.0**	**9**	**0**	**6**	**0**	**11**	**11**

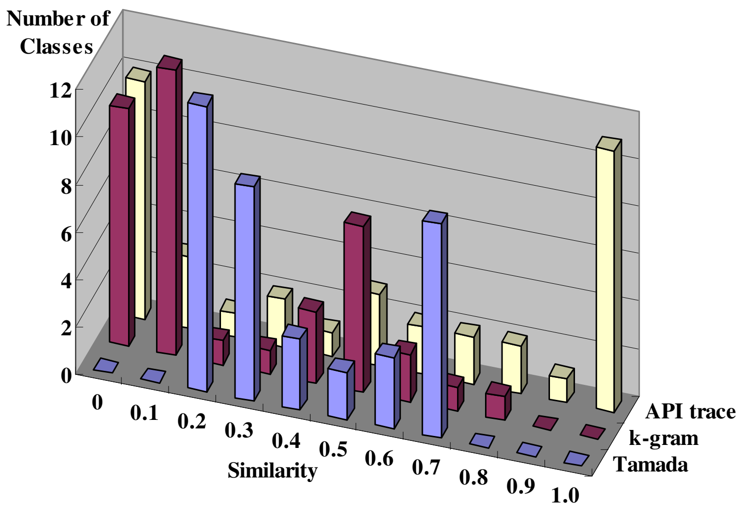

Fig. 6. Number of matched classes after code obfuscation

two programs. The perfect match means that the similarity between two classes is 1.0. Before code obfuscation, the Tamada and the k-gram birthmarks detected 9 and 6 perfectly matched classes, respectively. The Tamada and the k-gram birthmarks detected nothing perfectly matched after obfuscation, while our API trace birthmark detected 11 perfectly matched classes before and after code obfuscation.

We found 11 matched classes out of 16 common classes because two programs included different versions of classes. The *Crimson* includes the `xml.parsers` 1.1 and `xml.sax` 1.1. The *Piccolo* includes the `xml.parsers` 1.2 and `xml.sax` 1.1.1.1. However, all the 16 classes can be detected if we consider that classes are matched when similarities are greater than 0.7.

Figure 6 shows the number of matched classes after code obfuscation. It confirms that our API trace is suitable to detect module theft.

6 Conclusion and Future Work

In this paper, we propose a static API trace birthmark for Java. Because the API traces reflect the behavior of a program, our birthmark is more resilient than the existing static birthmarks. Because the API traces are extracted by static analysis, our method can be applied to library programs that dynamic birthmarks cannot handle properly. We adopted the semi-global algorithm that is widely used for comparing DNA sequences to compare two API traces. We evaluated the proposed birthmark in respect to credibility and resilience for the benchmark programs. The experimental result shows that the resilience of our static API trace birthmark is much higher than the other birthmarks, and our birthmark can detect common library modules of two packages which other birthmarks fail to detect. For future work, we plan to compare our API trace birthmark with the dynamic API birthmark [8]. We also plan to extend the comparison method by weighting API functions.

References

1. Wise, M.: YAP3: improved detection of similarities in computer program and other texts. In: Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education, pp. 130–134 (1996)
2. Prechelt, L., Malpohl, G., Philippsen, M.: Finding plagiarisms among a set of programs with JPlag. Journal of Universal Computer Science 8(11), 1016–1038 (2002)
3. Schleimer, S., Wilkerson, D., Aiken, A.: Winnowing: local algorithms for document fingerprinting. In: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pp. 76–85 (2003)
4. Tamada, H., Nakamura, M., Monden, A., Matsumoto, K.: Java Birthmarks-Detecting the Software Theft. IEICE Transactions on Information and Systems, 2148–2158 (2005)
5. Myles, G., Collberg, C.: K-gram based software birthmarks. In: Proceedings of the 2005 ACM symposium on Applied computing, pp. 314–318 (2005)
6. Myles, G., Collberg, C.: Detecting software theft via whole program path birthmarks. In: Information Security Conference, pp. 404–415 (2004)
7. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 184–196 (1998)
8. Schuler, D., Dallmeier, V., Lindig, C.: A Dynamic Birthmark for Java. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, pp. 274–283 (2007)
9. Tamada, H., Okamoto, K., Nakamura, M., Monden, A., Matsumoto, K.: Dynamic Software Birthmarks to Detect the Theft of Windows Applications. In: International Symposium on Future Software Technology (ISFST 2004) (2004)
10. Choi, S., Park, H., Lim, H., Han, T.: A Static Birthmark of Binary Executables Based on API Call Structure. In: Cervesato, I. (ed.) ASIAN 2007. LNCS, vol. 4846, pp. 2–16. Springer, Heidelberg (2007)

11. Needleman, S., Wunsch, C.: A general method applicable to search for similarities in amino acid sequence of 2 proteins. Journal of Molecular Biology 48, 443–453 (1970)
12. Temple, F., Michael, S.: Identification of Common Molecular Subsequences. Journal of Molecular Biology 147, 195–197 (1981)
13. Brudno, M., Malde, S., Poliakov, A., Do, C., Couronne, O., Dubchak, I., Batzoglou, S.: Glocal alignment: finding rearrangements during alignment. Bioinformatics 19, 54–62 (2003)