Detecting Code Theft via a Static Instruction Trace Birthmark for Java Methods

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
Email: {hwpark, swchoi, hilim, han}@pllab.kaist.ac.kr

Abstract—A software birthmark is an inherent program characteristic that can identify a program. In this paper, we propose a static instruction trace birthmark to detect code theft of Java methods. Because the static instruction traces can reflect the algorithmic structure of a program, our birthmark can be used to detect algorithm theft which existing static birthmarks cannot handle. Because the static instruction traces are extracted by static analyses, they can be applied to library programs which previous dynamic birthmarks could not. We evaluate the proposed birthmark with respect to two criteria: credibility and resilience. Experimental result shows that our birthmark is more resilient than and at least as credible as the existing Java birthmarks.

I. INTRODUCTION

For many IT companies, source codes are core part of their intellectual property. Code theft detection is one of the methods to protect codes. Recently many programs are under development in the form of open source projects. Open source programs allow users to modify or distribute program codes under certain types of software licenses. The most popular software license is the GNU Public License (GPL), under which, program codes are freely used while the software using the original codes should also be under the GPL. However, many software developers use codes under the GPL secretly in their company projects. Sometimes illegal usages of the open source code are detected by the code owners and the whole projects fail. IT companies should inspect their codes developed by the employees to avoid these occasions. Moreover, many cases are reported that software engineers, who are scouted by competitors, secretly take the core codes from the former companies. In these cases, code theft detection can be used to prove the ownership of the code in lawsuits.

There are plagiarism detectors such as YAP, JPlag, and MOSS that are commonly used to discover code theft [1][2][3]. The plagiarism detectors target to the source codes of the programs. Because most commercial programs distribute only binary executables, binary code analysis is required. Software birthmarking techniques can be used to detect the code thefts of the binary executables. A software birthmark is a combination of unique characteristics from a program. For example, the strings extracted from an executable or binary checksums are candidates for 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 determine that one program is a

copy of the other. A software watermark is similar to software birthmark in the sense that they can be used to detect code theft. The difference is that a software birthmark extracts characteristics from the software itself while a software watermark extracts pre-embedded fingerprints from the software. As software watermark provides certain evidence of code theft by extracted copyright information, while a software birthmark suggests the possibility of code theft by the similarity between programs. In some cases, however, a software birthmark is better than a software watermark 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 instruction trace birthmark for Java methods. The static instruction trace birthmark is the set of all possible run-time traces of Java methods. The set of traces covers all the edges of the control flow graph of a module. Unlike earlier birthmarks, the static instruction trace birthmark does not simply extract properties such as instruction sequences, class hierarchy, or method calls which appear on the surface, but analyzes the control flow of methods and generates possible run-time traces. Since the run-time traces can reflect the algorithmic structure of a program, our birthmark can be used to detect algorithm theft which existing birthmarks cannot handle.

We evaluate the proposed birthmark with respect to credibility and resilience. The credibility of a birthmark is its ability to distinguish different programs. The resilience of a birthmark is its ability to resist program transformations. Experimental result shows that our birthmark is more resilient than and at least as credible as the existing Java birthmarks.

II. THE STATIC INSTRUCTION TRACE BIRTHMARK

A software birthmark of a program is the inherent characteristics that can identify the program. A software birthmarking system is the system that provides two functions: birthmark *extraction* and *comparison*. In this section, we define the

static instruction trace birthmark and suggest birthmark extraction and comparison methods.

A. The Definition of the Static Instruction Trace Birthmark

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

Definition 1: (Static Instruction Trace) A *static instruction trace* of a method is a sequence of instructions starting from the entry point of the method and ending at one of the exit points of the method.

A static instruction trace may match a dynamic trace of the method. The static analysis performed to get static instruction traces can generate abstracted traces that contain all possible dynamic traces. In Definition 1, however, we define the static instruction trace as a concrete trace rather than 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 2: (Edge Covering Static Instruction Trace Set) An *edge covering static instruction trace set* of a method is a set of static instruction traces where the union of all the static instruction traces covers all the edges of the control flow graph of the method.

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

Definition 3: (Static Instruction Trace Birthmark) The *static instruction trace birthmark* is the minimal edge covering static instruction trace set.

B. Static Instruction Trace Generation

We generate control flow graphs from the Java bytecodes. One peculiarity when extracting birthmarks from the Java bytecode is that the Java bytecode contains subroutine call instructions, such as `jsr` and `jsr_w`, and exception handling. To generate the control flow graph for `jsr` and `jsr_w` instructions, `ret` instructions should be linked to the next instruction of the `jsr` instruction. For handling exceptions, all instructions contained in the exception range should be linked to the exception target. Because the number of edges generated by the exceptions is huge, we add only one edge from the end of the each block to the exception handler.

After the control flow graph is generated, we generate edge covering traces. In order to get all possible traces, we first

Fig. 1. An example to show the tree generation method for branches

Fig. 2. An example of the tree generation method for loops

construct an edge covering tree, and by traversing the tree, we get the edge covering traces.

Figure 1 shows that the duplication of nodes occurs for branch instructions. Figure 2 shows how to handle loops. Since a loop can generate infinite instruction traces during execution, we limit the iteration of the loop to one cycle unlike other program analyses. One iteration is sufficient to generate an edge covering trace. Figure 3 shows an example of the tree generation of subroutine instructions `jsr` and `ret`. The `jsr` instruction stores the return address that is next to the executed `jsr` instruction. When a `ret` is executed, the control flow goes to the address stores by `jsr` instructions. We use a subroutine stack for `jsr` and `ret` instructions to build the correct tree.

Once the trace tree is fully constructed, a traversal of the whole tree generates an edge covering static instruction trace set.

C. Similarity Calculation with Semi-global Alignment

To compare two traces extracted from two programs, we utilize sequence alignment algorithms that are frequently used for 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 similarity using the whole sequences [4]. The local alignment algorithm finds the maximum subsequence matching [5]. The semi-global alignment algorithm basically uses the global alignment algorithms, but it compensates the score penalties caused by heading and tailing mismatches [6].

Fig. 3. An example of the tree generation method for jsr instruction

Fig. 4. An example of the tree generation method for exceptions

We evaluated three alignment algorithms above for our trace birthmark of Java methods. In most cases, the similarities calculated by the local alignment algorithm were higher than the similarities by the semi-global alignment algorithm, meaning that the local alignment algorithm is not as credible as the semi-global alignment algorithm. The global alignment algorithm has the disadvantage that the similarity is sensitive to the length of the traces such that if one sequence is contained in another sequence and the difference in the length is great, the similarity is relatively low. We chose the semi-global alignment because it can handle the containment problem and it is more credible than the local alignment algorithm.

Definition 4: (Global Alignment) Let the original trace be T_1 , the copy trace be T_2 , $\sigma(i, j)$ be the match score between $T_1[i]$ and $T_2[j]$, and gap be the penalty score according to 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 penalty are

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

$$gap = -1.$$

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

Definition 5: (Semi-global Alignment) Given a global alignment $G(i, j)$, a semi-global alignment $SG(i, j)$ is defined as

$$SG(T_1, T_2) = \max\{G(i, 1), G(i, 2), \dots, G(i, j), 0\}$$

where $G(0, 0) = G(0, 1) = \dots, G(0, j) = 0$.

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

Definition 6: (Trace Similarity) Given traces T_1 and T_2 , the trace similarity between T_1 and T_2 is defined as

$$sim_t(T_1, T_2) = \max\left(\frac{SG(T_1, T_2)}{|T_1|}, \frac{SG(T_2, T_1)}{|T_2|}\right).$$

In Definition 6, the trace similarity considers both cases where T_1 is a copy and T_2 is original, and vice versa.

Definition 7: (Module Similarity) Let the set of traces generated from the module P be $p[1..m]$ and the set of traces generated from the module Q be $q[1..n]$ where m, n is the numbers of traces from P and Q . The m by n SG matrix is composed of $SG(p[i], q[j])$ which is computed by applying the semi-global assignment algorithm to all combinations of traces from P and Q . The module similarity between P and Q is defined by solving the matching problem between $p[1..m]$ and $q[1..n]$ using a greedy method as

$$sim_m(P, Q) = \max\left(\frac{\sum_{i=1}^{\min(m,n)} val[i]}{\sum_{i=1}^m |p[i]|}, \frac{\sum_{j=1}^{\min(m,n)} val[j]}{\sum_{j=1}^n |q[j]|}\right)$$

where $val[1..\min(m, n)]$ is an array that contains the values that are selected by the greedy maximum matching method.

We can use the Hungarian algorithm [7] for the matching algorithm instead of the greedy algorithm to maximize the similarity. However, we selected the greedy algorithm. Even though the difference in the similarities between the greedy

TABLE I
AN EXAMPLE OF AN $SG(P, Q)$ MATRIX COMPUTED BY THE SEMI-GLOBAL ALIGNMENT

	$q[1]$ ($ q[1] = 3$)	$q[2]$ ($ q[2] = 7$)	$q[3]$ ($ q[3] = 5$)
$p[1]$ ($ p[1] = 5$)	1	4	3
$p[2]$ ($ p[2] = 7$)	3	6	0

algorithm and the Hungarian algorithm is negligible and the complexity of the two algorithms is the same as $O(n^3)$, the Hungarian algorithm requires a huge amount of memory.

Table I is an example of an $SG(P, Q)$ matrix computed by the semi-global alignment algorithm. The $sim_t(p[1], q[1])$ is computed by Definition 6 as

$$sim_t(p[1], q[1]) = \max(1/5, 1/3) = 1/3.$$

The $sim_m(P, Q)$ is computed using the greedy method. The maximum value from Table I is 6. We choose the score 6 by the semi-global alignment of $p[2]$ and $q[2]$. The score is recorded in $val[1]$ and the row $p[2]$ and the column $q[2]$ are deleted because we already selected one match between $p[2]$ and $q[2]$. We again choose the maximum value 3 among the remnants and delete row $p[1]$ and column $q[3]$. Since there is no remaining element in the matrix, we compute the module similarity $sim_m(P, Q)$ as

$$\begin{aligned} sim_m(P, Q) &= \max((6 + 3)/12, (6 + 3)/15)) \\ &= \max(3/4, 3/5) = 3/4. \end{aligned}$$

III. EVALUATION

A. Implementation

Figure 5 shows the architecture of our static instruction trace birthmarking system. The static instruction trace birthmarking system gets two functions in the form of Java bytecode. Two control flow graphs are generated from the two functions. The nodes of the control flow graphs contain the Java bytecode instructions and operands. All branch instructions including subroutine call instructions are not contained in the control flow graphs. To cope with code obfuscation, we discard symbol names and references. The edges are constructed according to the method explained in Section II-B. With the control flow graphs, static instruction trace trees are built. By traversal of the trees, static instruction traces are generated. The instruction traces contain opcodes and operands without symbol names and references. Finally, the instruction trace sets are compared to assess the similarity between two methods using the semi-global alignment algorithm and the greedy maximum matching algorithm. Because the matrix is computed using dynamic programming, the time complexity of the semi-global alignment

Fig. 5. Static Instruction Trace Birthmarking System

TABLE II
BENCHMARK PROGRAMS

	Number of Bytecodes	Number of Nodes	Number of Edges	Number of Traces	Average Trace Size
IterFib	26	5	5	2	21.5
IterFact	16	5	5	2	12.0
RecurFib	20	6	6	3	9.7
RecurFact	13	5	4	2	7.5
Bubble	44	9	11	4	22.8
Heap	142	22	31	289	122.9
Insertion	39	8	10	5	26.6
Merge	103	16	20	14	67.9
Quick	73	11	13	17	65.9
Selection	46	9	11	4	32.8
Shell	54	11	14	6	31.5
Hanoi	51	4	4	2	26.0
we	91	8	10	6	86.3
Knapsack	81	12	15	6	54.2
BinSearch	39	11	10	4	18.3
PrimeNum	33	11	14	9	24.8

algorithm is $O(m \cdot n)$ where m, n are the sizes of the two traces. The time complexity of the greedy maximum matching algorithm is $O(m \cdot n \cdot \min(m, n))$.

B. Experimental Result

To prove the effectiveness of our birthmark, we compare the static instruction trace birthmark with the k -gram birthmark of G. Myles [8]. The credibility of the birthmarks is measured by comparing the benchmark programs in Table II. The resilience of the birthmarks is measured by comparing the original codes with the transformed programs by the Java code obfuscator *Smokescreen* [9], compressing utility *Jarg* [10], and Java compilers *javac* and *jikes* [11].

Tables III and IV show the similarities among benchmark programs to evaluate the credibility of the static instruction trace birthmark. The average of the similarities of the 3-gram birthmark between different programs is 0.052. The average of the similarities of the static instruction trace birthmark is

TABLE III
SIMILARITIES OF k -GRAM BIRTHMARK WITH $k = 3$ TO EVALUATE THE CREDIBILITY

	IterFib	IterFact	RecurFib	RecurFact	Bubble	Heap	Insertion	Merge	Quick	Selection	Shell	Hanoi	wc	Knapsack	BinSearch	PrimeNum
IterFib	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.042	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.042
IterFact	-	1.000	0.000	0.000	0.071	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
RecurFib	-	-	1.000	0.636	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
RecurFact	-	-	-	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Bubble	-	-	-	-	1.000	0.306	0.273	0.139	0.222	0.278	0.222	0.000	0.056	0.111	0.028	0.065
Heap	-	-	-	-	-	1.000	0.303	0.068	0.158	0.256	0.178	0.000	0.014	0.016	0.028	0.032
Insertion	-	-	-	-	-	-	1.000	0.212	0.121	0.273	0.061	0.000	0.000	0.121	0.000	0.032
Merge	-	-	-	-	-	-	-	1.000	0.246	0.205	0.067	0.000	0.000	0.129	0.000	0.097
Quick	-	-	-	-	-	-	-	-	1.000	0.256	0.111	0.000	0.000	0.070	0.028	0.000
Selection	-	-	-	-	-	-	-	-	-	1.000	0.077	0.000	0.000	0.026	0.028	0.032
Shell	-	-	-	-	-	-	-	-	-	-	1.000	0.000	0.022	0.044	0.000	0.097
Hanoi	-	-	-	-	-	-	-	-	-	-	-	1.000	0.118	0.000	0.000	0.000
wc	-	-	-	-	-	-	-	-	-	-	-	-	1.000	0.032	0.028	0.032
Knapsack	-	-	-	-	-	-	-	-	-	-	-	-	-	1.000	0.000	0.097
BinSearch	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1.000	0.000
PrimeNum	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1.000

TABLE IV
SIMILARITIES OF THE STATIC INSTRUCTION TRACE BIRTHMARK TO EVALUATE THE CREDIBILITY

	IterFib	IterFact	RecurFib	RecurFact	Bubble	Heap	Insertion	Merge	Quick	Selection	Shell	Hanoi	wc	Knapsack	BinSearch	PrimeNum
IterFib	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.001	0.015	0.000	0.000	0.000	0.000	0.000	0.018
IterFact	-	1.000	0.000	0.000	0.055	0.000	0.023	0.001	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000
RecurFib	-	-	1.000	0.600	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
RecurFact	-	-	-	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Bubble	-	-	-	-	1.000	0.209	0.376	0.132	0.187	0.220	0.165	0.011	0.000	0.264	0.011	0.077
Heap	-	-	-	-	-	1.000	0.143	0.093	0.273	0.298	0.048	0.000	0.000	0.000	0.000	0.014
Insertion	-	-	-	-	-	-	1.000	0.188	0.293	0.391	0.263	0.008	0.000	0.203	0.008	0.030
Merge	-	-	-	-	-	-	-	1.000	0.019	0.115	0.037	0.002	0.000	0.034	0.096	0.005
Quick	-	-	-	-	-	-	-	-	1.000	0.382	0.228	0.002	0.000	0.029	0.000	0.003
Selection	-	-	-	-	-	-	-	-	-	1.000	0.153	0.000	0.000	0.008	0.000	0.076
Shell	-	-	-	-	-	-	-	-	-	-	1.000	0.000	0.000	0.037	0.011	0.048
Hanoi	-	-	-	-	-	-	-	-	-	-	-	1.000	0.058	0.000	0.000	0.000
wc	-	-	-	-	-	-	-	-	-	-	-	-	1.000	0.000	0.000	0.000
Knapsack	-	-	-	-	-	-	-	-	-	-	-	-	-	1.000	0.027	0.018
BinSearch	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1.000	0.000
PrimeNum	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1.000

0.050. We conclude that both birthmarks are credible since the averages are very low and the difference in the averages between the 3-gram and the static instruction trace birthmarks is negligible.

Table V shows the similarities to evaluate the resilience. The benchmark programs are transformed with *Smokescreen*, *Jarg*, and *Jikes*, and the resulting bytecodes are compared to the original bytecodes. The differences of average similarities between the 3-gram and the static instruction trace birthmarks prove that our birthmark is more resilient than the 3-gram birthmark. Our birthmark shows much higher resilience than the 3-gram birthmark, especially with *Jikes*. We inspected the transformed Java bytecodes to find the reasons for this. We observed that branches which have the same semantics are compiled into different bytecode instructions. For example, `ifgt`, which means *greater than*, generated by the *javac* compiler, is compiled into `iflge`, and the order of two following statements is exchanged by the *Jikes* compiler. The k -gram birthmark is vulnerable to the program transformations which replace instructions with semantically equivalent instructions. In addition, the k -gram method cannot generate the correct k -grams in the case when the true branch and the false branch are

TABLE V
SIMILARITIES OF k -GRAM BIRTHMARK AND THE STATIC INSTRUCTION TRACE BIRTHMARK TO EVALUATE RESILIENCE

	Smokescreen		Jarg		Jikes	
	3-gram	Trace	3-gram	Trace	3-gram	Trace
IterFib	0.292	0.581	1.000	1.000	0.667	1.000
IterFact	0.571	0.792	1.000	1.000	0.429	1.000
RecurFib	0.889	1.000	1.000	1.000	0.833	0.897
RecurFact	0.818	1.000	0.727	0.867	1.000	1.000
Bubble	0.944	0.967	1.000	1.000	0.611	1.000
Heap	0.797	0.897	0.944	1.000	0.797	1.000
Insertion	0.939	0.970	1.000	1.000	0.606	1.000
Merge	0.763	0.916	0.960	1.000	0.829	1.000
Quick	0.842	0.905	0.946	1.000	0.875	1.000
Selection	0.718	0.809	1.000	1.000	0.718	1.000
Shell	0.911	0.974	1.000	1.000	0.600	1.000
Hanoi	0.941	1.000	0.912	1.000	0.879	0.880
wc	0.887	0.931	0.870	0.900	0.916	0.861
Knapsack	0.807	0.874	1.000	1.000	0.710	0.969
BinSearch	0.889	1.000	0.889	0.877	1.000	1.000
PrimeNum	0.613	0.776	0.897	0.834	0.516	1.000
Average	0.789	0.899	0.947	0.967	0.749	0.975
Difference	0.110		0.021		0.226	

swapped and equivalent instructions such as `ifgt` and `ifle` are used, because the k -gram does not follow the control flow edge, but instead follows the physically adjacent instructions. The static instruction trace birthmark can handle this type of problem because it compares two branches simultaneously by generating all traces. This makes the static instruction trace birthmark highly resilient.

IV. 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 program behaviors. 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 input 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 [12]. This birthmark is defined as the combination of the 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 method because it only compares externally observable features.

G. Myles et al. suggested the k -gram birthmark for Java [8]. The k -gram birthmark is the k length sequence of bytecode instructions. The birthmark is highly credible, but frail to program transformations. They also suggested a Whole Program Path(WPP) birthmark for Java [13]. The WPP birthmark records run-time instruction traces and constructs a dynamic control flow graph. Because the WPP birthmark are generated using a sequence of executed instructions, the birthmark is resilient to some obfuscation transformations such as inserting opaque predicates [14] or garbage instructions. However, as a dynamic birthmark, the WPP birthmark can be changed depending on input and environments.

D. Schuler et al. suggested a dynamic API birthmark for Java [15]. The dynamic API birthmark is a set of API call traces per object. The collection of API calls per object makes this birthmark more credible than the earlier dynamic API birthmark for Windows [16]. The limitation of this birthmark is that it can handle only applications that call API frequently.

S. Choi et al. suggested a static API birthmark for Windows [17]. Their static API birthmark is a set of possible API calls which is extracted statically by analyzing disassembled code. This birthmark can be applied only to Windows API applications.

V. CONCLUSION AND FUTURE WORK

In this paper, we propose a static instruction trace birthmark for Java methods. Because the static instruction traces reflect the algorithmic structure of a program, our birthmark can be

used to detect algorithm theft which existing static birthmarks cannot handle. Because the static instruction traces are extracted by static analysis, they can be applied to library programs which earlier dynamic birthmarks cannot. We adopted the semi-global alignment algorithm that is widely used for comparison of DNA sequences to compare two instruction traces. We evaluated the proposed birthmark with respect to credibility and resilience using several benchmark programs. The experimental results show that the resilience is much higher than the k -gram birthmark and the credibility of our birthmark is as high as the k -gram birthmark, which is known as the most credible static birthmark to date. For future work, we plan to extend our birthmark to Java classes and libraries. We also plan to improve our comparison method by assigning weights to instructions according to the instruction frequency.

ACKNOWLEDGMENT

This research was supported by the Ministry of Knowledge Economy, Korea, under the Information Technology Research Center support program supervised by the Institute of Information Technology Advancement(IITA-2008-C1090-0801-0020)

REFERENCES

- [1] M. Wise, "YAP3: improved detection of similarities in computer program and other texts," *Proceedings of the 27th SIGCSE technical symposium on Computer science education*, pp. 130–134, 1996.
- [2] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 1016–1038, 2002.
- [3] S. Schleimer, D. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pp. 76–85, 2003.
- [4] S. Needleman and C. Wunsch, "A general method applicable to search for similarities in amino acid sequence of 2 proteins," *J. Mol. Biol.*, vol. 48, p. 443, 1970.
- [5] F. Temple and S. Michael, "Identification of Common Molecular Subsequences," *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.
- [6] M. Brudno, S. Malde, A. Poliakov, C. Do, O. Couronne, I. Dubchak, and S. Batzoglou, "Glocal alignment: finding rearrangements during alignment," *Bioinformatics*, vol. 19, no. 90001, pp. 54–62, 2003.
- [7] H. Kuhn, "The Hungarian method for the assignment problem," *Naval Research Logistics*, vol. 52, no. 1, pp. 7–21, 2005.
- [8] G. Myles and C. Collberg, "K-gram based software birthmarks," *Proceedings of the 2005 ACM symposium on Applied computing*, pp. 314–318, 2005.
- [9] "Smokescreen," <http://www.leesw.com/smokescreen/>.
- [10] "Java Archive Grinder(jarg)," <http://sourceforge.net/projects/jarg/>.
- [11] "Jikes Java Compiler," <http://jikes.sourceforge.net/>.
- [12] H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto, "Java Birthmarks-Detecting the Software Theft," *IEICE Transactions on Information and Systems*, pp. 2148–2158, 2005.
- [13] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," *Information Security Conference*, pp. 404–415, 2004.
- [14] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 184–196, 1998.
- [15] D. Schuler, V. Dallmeier, and C. Lindig, "A Dynamic Birthmark for Java," *In Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007.
- [16] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. Matsumoto, "Dynamic Software Birthmarks to Detect the Theft of Windows Applications," *International Symposium on Future Software Technology*, vol. 20, no. 22, 2004.
- [17] S. Choi, H. Park, H. Lim, and T. Han, "A Static Birthmark of Binary Executables Based on API Call Structure," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 4846, p. 2, 2007.