PAPER

Analyzing Stack Flows to Compare Java Programs*

Hyun-il LIM[†a)], ***Member and*** **Taisook HAN**[††], ***Nonmember***

SUMMARY This paper presents a method for comparing and detecting clones of Java programs by analyzing program stack flows. A stack flow denotes an operational behavior of a program by describing individual instructions and stack movements for performing specific operations. We analyze stack flows by simulating the operand stack movements during execution of a Java program. Two programs for detection of clones of Java programs are compared by matching similar pairs of stack flows in the programs. Experiments were performed on the proposed method and compared with the earlier approaches of comparing Java programs, the Tamada, *k*-gram, and stack pattern based methods. Their performance was evaluated with real-world Java programs in several categories collected from the Internet. The experimental results show that the proposed method is more effective than earlier methods of comparing and detecting clones of Java programs.

key words: *software clone detection, software copyright protection, Java bytecode analysis*

1. Introduction

With the rapid advances in the Internet and computing environment, the demand for software development is also increasing. Software is the intellectual property of its developers and is protected by copyright law and regulations. In the case of open source software, many programs are distributed with the source code, and subscribers are allowed to modify or redistribute the program code under certain types of software licenses, such as the GNU General Public License (GPL), Open Software License (OSL), and BSD License. However, it has been reported that many software developers and companies do not follow such license policies [1]. Moreover, these cases may lead to legal disputes.

To reduce and cope with software license violations, it is necessary to protect against illegal tampering and to identify the originality of the software. However, this is not easy, because problematic software may be distributed without source code, and binary executables are not suitable for a direct comparison with other software. Moreover, the software may be obfuscated or modified to hide a license violoation. Hence, in several lawsuits involving GPL violations, evidence has been shown of manual reverse-engineering, which is tedious and time-consuming. Therefore, development of efficient technology that identifies the originality of software is necessary.

This paper presents a method to compare and detect cloned Java programs by comparing analyzed stack flows of Java programs. A stack flow can be represented by a minimal sequence of bytecodes that performs some tasks in the common context of the operand stack. Stack flows are analyzed by simulating operand stack movements. They denote the behavior of a Java program by representing individual instructions and stack movements to perform specific operations; thus, a comparison of such stack flows gives a guideline for deciding whether one program is a copied version of the other.

The proposed method is evaluated with respect to two criteria. The first criterion is discrimination between different Java programs, and it requires a program comparison method to differentiate independent programs clearly. The second criterion is detection of cloned programs between modified versions of a single program. Experiments are conducted to test the proposed method in relation to these criteria, and its performance is evaluated by analyzing false-positive rates, false-negative rates, and transformation variations. Additionally, the method is compared with three previous approaches, namely, the Tamada birthmark [2], [3], the *k*-gram based birthmark [4], [5], and the stack pattern birthmark [6]. The experimental results demonstrate that the proposed method is more effective than the previous methods in terms of the above criteria.

In this research, we propose a new approach to compare and detect cloned Java programs. This research makes the following contributions:

1. We analyze the stack flows of a Java program through analysis of operand stack movements.
2. We propose and implement a Java program comparison to detect cloned Java programs.
3. We evaluate and compare the performance of the proposed method with earlier existing approaches in real-world Java applications.
4. Our work contributes to reducing the effort of manual reverse engineering in detecting cloned Java programs.

The remainder of this paper is organized as follows. Section 2 reviews existing approaches to software compari-

Manuscript received March 23, 2011.
Manuscript revised August 11, 2011.

[†]The author is with the Division of Computer Engineering, Kyungnam University, Korea.
[††]The author is with the Department of Computer Science, KAIST, Korea.

[*]This work was partially supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2010-0000258). This research was partially supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (No. 2010-0024658).

a) E-mail: hilimcs@gmail.com

DOI: 10.1587/transinf.E95.D.565

son methods. Section 3 describes the proposed method for detecting cloned Java programs by analyzing the stack flows of Java programs. Section 4 presents experimental data and evaluates the proposed method. Section 5 discusses issues to be resolved regarding the proposed method and future works. Finally, Sect. 6 concludes the paper.

2. Related Work

Baker and Manber [7] proposed a method to deduce similarities in Java bytecodes by applying text similarity measures, such as Siff [8], parameterized pattern matching [9], and diff. Tamada et al. [2], [3] first suggested a practical application of static software birthmarks and presented a method to compare the birthmarks for Java class files. Their technique consists of four individual birthmarks: constant values in field variables, sequence of method calls, inheritance structure, and used classes. These four birthmarks can be used individually, but they are more reliable when combined. Myles et al. [4], [5] proposed the *k*-gram based birthmark, which is based on instruction sequences. Lim et al. [6] proposed the stack pattern based birthmark, which uses sequences of contiguous opcodes partitioned on the basis of the operand stack depth. This method is susceptible to program modifications that change the control flows of the program, because it does not follow the control flow of a program. Lim et al. [10], [11] presented a method of using the control flow information of Java programs to reflect the behavioral characteristics of programs. In this approach, they utilized the relation between basic blocks analyzed from control flow information as a baseline for characterizing the program. Park et al. [12] proposed the object trace birthmark, which is based on object-related instructions in Java programs. Object instruction traces are extracted from the control flow graphs of Java programs, and the traces are compared using the local alignment algorithm [13].

Tamada et al. [14] introduced a dynamic birthmark based on traces of system calls for Windows programs. They proposed the use of sequences and frequencies of API function calls during program execution to compare programs. Schuler et al. [15], [16] presented similar approaches in Java applications. To compare programs, they used sets of API call sequences during program execution. Myles et al. [5], [17] proposed the whole program path (WPP) birthmark. A WPP is obtained from a dynamic trace of a program by using instrumentation, and the trace is compressed into a directed acyclic graph using the SEQUITUR algorithm. Such graph structures are used as birthmarks, and compared by using graph distance for a maximal common subgraph.

Brown et al. [18] presented a fingerprint generator/detector (FiGD) for Java programs. FiGD generates a fingerprint by extracting characteristic *n*-grams from an archived jar file, and detects copied programs by comparing the fingerprint with *n*-grams of suspicious programs.

Sæbjørnsen et al. [19] presented a method for clone detection in binary executables. This method detects binary code clones from the distances between feature vectors, which are summarized from instructions contained in corresponding code regions. Because the feature vectors should be able to present sufficient characteristics of target programs, this method is well suited for programs in a CISC architecture. In comparing two programs, the persistence of instructions of code regions in an original program determines the reliability of this method. However, code regions do not present associations between instructions; they are just constructed by separating a program into several pieces of code with a specific window size. Therefore, this method is susceptible to program modifications, such as the introduction of bogus instructions or code reordering, which may affect the boundary of code regions in a program.

Krinke [20] presented a method for calculating the structural similarity of programs by means of a program dependence graph (PDG), which describes a program as a graph representation attributed with control and data dependence relations. In comparing two programs, this method finds the maximal similar *k*-limited path induced subgraph from PDGs of the two programs. This method relies on the persistence of data and control dependence relations in an original program. Because modifications, such as insertion of bogus instructions or control flows, may affect data or control dependence relations, they may cause several false positives or false negatives in the detection of code clones. Moreover, it is not practical to apply in a large collection of binary code because this method is based on comparisons of graph structures.

Malware detection [21], [22] which detects malicious behaviors from binary programs is also related to binary code analysis. This method is not used for detecting arbitrary similar binary code; rather, it finds predefined malicious behaviors or signatures of binary code. Thus, it is customized to efficiently detect known malicious behaviors from a large collection of binary programs.

3. Analyzing Stack Flows of Java Programs

3.1 Analyzing Stack Statuses

Because the Java bytecodes use the operand stack as a workspace, the context of a Java program can be separated by simulating the operand stack of the JVM. From the specification of the Java bytecode [23], stack statuses of a Java program can be determined by static analysis, as presented in [6]. The stack statuses of each bytecode in a class file, pre-status and post-status, are defined as follows:

Definition 1 (Pre-status and post-status): Let P be a Java program. For a bytecode x contained in P, the operand stack depth just before execution of the bytecode x is called the *pre-status* of x. The operand stack depth just after execution of the bytecode x is called the *post-status* of x.

Because every bytecode is related to operand stack operation, the operand stack depth continuously varies during program execution. The pre-status and post-status of a bytecode represent the operand stack depth in the corresponding context of a program. Therefore, identical bytecodes in Java programs may have different stack statuses according to the context of programs.

3.2 Analyzing Stack Flows

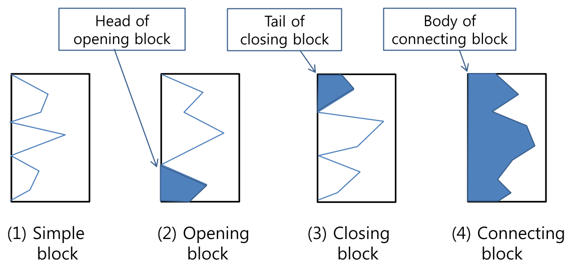
A *stack flow* means a sequence of bytecodes that construct a series of stack movements during the execution of a Java program; thus, a stack flow denotes a specific task in the common context of a Java program.

Definition 2 (Stack flow): For a Java program P, $op(P, i)$ denotes the i-th bytecode in P. $pre(P, i)$ and $post(P, i)$ denote the pre-status and the post-status of the i-th bytecode in P, respectively. The sequence of pairs of bytecodes and their corresponding post-statuses, $a = \langle(op(P, i), post(P, i)), \cdots, (op(P, j), post(P, j))\rangle$, is called a *stack flow* of P iff;

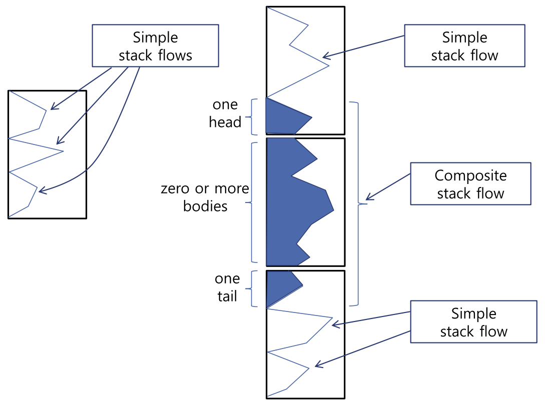
1. $\langle op(P, i), \cdots, op(P, j)\rangle$[†] is a contiguous feasible sequence of bytecodes of program P at runtime.
2. Both $pre(P, i)$ and $post(P, j)$ are 0.
3. The post-statuses of all bytecodes in a other than the last bytecode $op(P, j)$ are higher than 0.

In Definition 2, a stack flow represents a minimal sequence of Java bytecodes decomposed via operand stack statuses. Then, a *stack flow set* is a set of all the stack flows analyzed from a Java program. Stack flows may reflect textual sequences of a program; however, if control flows are forwarded by branch instructions to nonadjacent addresses, stack flows containing the branch instructions also should follow their control flows.

A *control flow graph* [24], [25] is a graph representing a program structure in which nodes and edges describe the basic blocks and the possible control flows, respectively. Every basic block contains a sequence of several bytecodes, and each bytecode has its own pre-status and post-status. Figure 1 (a) illustrates different types of basic blocks according to the movements of bytecode statuses. The rectangles represent basic blocks and the graph line on the basic blocks represent the variations of bytecode statuses during execution of the basic blocks. A basic block is called a *simple block* if both the pre-status of the first bytecode and the post-status of the last bytecode are 0. A basic block is called an *opening block* if the pre-status of at least one bytecode is 0 and the post-status of the last bytecode is not 0. The partial stack flow that initiates but does not yet finish a series of stack movement is called the *head* of the opening block. A basic block is called a *closing block* if the pre-status of the first bytecode is not 0, and the post-status of at least one bytecode is 0. The partial stack flow that corresponds to a closing stack movement is called the *tail* of the closing block. A basic block is called a *connecting block* if no statuses of the bytecodes in the basic block reach 0. The partial stack flow that forms the entire connecting block is called



(a) Various types of basic blocks.



(b) Various types of stack flows.

Fig. 1 Example of basic blocks and stack flows.

the *body* of the connecting block.

Figure 1 (b) illustrates different types of stack flows. A stack flow is called *simple* if it is contained within only one basic block, and it is called *composite* if it spans two or more basic blocks. Thus, a composite stack flow consists of the head of one opening block, zero or more bodies of connecting blocks, and the tail of one closing block, as shown in Fig. 1 (b).

With the stack statuses of bytecodes in a Java program, stack flows can be analyzed by traversing the control flow graph of the program without making repetitive traces. Algorithms 1 and 2 are used to analyze the stack flows of a Java program. For each basic block, simple stack flows are analyzed by sequentially traversing statuses of bytecodes in a basic block. Subsequently, if the basic block is an opening block, then the opening block and its directly reachable blocks are analyzed to find composite stack flows. To complete a composite stack flow, the control flow is traversed with the current partial stack flow until it reaches a closing block. A composite stack flow is constructed by concatenating the current partial stack flow with the tail of a closing block. While traversing the control flow, its current trace is maintained to prevent the flow from entering an infinite loop.

Stack flows in a Java program represent operational procedures which show its behavioral characteristics. Be-

[†]General Java programs do not contain loops in a stack flow, but control flow obfuscation may cause loops in the instructions sequence of a stack flow. If a loop structure is contained in a stack flow, each loop is explored at most once.

```
Input: Control flow graph $G$ and the statuses of bytecodes of an
       input Java program.
Output: A stack flow set of the input program.
begin
    Let $\{B_1, \cdots, B_n\}$ be basic blocks of $G$;
    for $i \leftarrow 1$ to $n$ do /* Analyzing simple stack flow. */
        Call AddSimpleFlow($B_i$);
        if $B_i$ is opening block then /* Beginning of composite stack flow. */
            foreach basic block $B_j$ directly reachable from $B_i$ do
                nextSF ← head ($B_i$);
                nextTrace ← concat (null, $i$);
                Call AddCompositeFlow (nextSF, $B_j$, nextTrace)) for finding composite stack flows;
```

Algorithm 1: Algorithm for analyzing stack flows of a Java program.

```
AddSimpleFlow($B$)
Input: Basic block $B$.
Output: Simple stack flows in the input basic block $B$.
begin
    Let $(bc_1, \cdots, bc_n)$ be a sequence of bytecodes in $B$;
    if $pre(bc_1) = 0$ then /* Beginning of stack flow. */
        flag ← true;
    else
        flag ← false;
    index ← 1 ; /* Current index of stack flow. */
    for $i = 1$ to $n$ do
        if flag = true then
            StackFlow [index] ← $(bc_i, post(bc_i))$;
            index ← index + 1;
        if $post(bc_i) = 0$ then
            if flag = false then /* Beginning of stack flow. */
                flag ← true;
            else /* End of stack flow. */
                Add StackFlow to the stack flow set;
                index ← 1;

AddCompositeFlow(currentSF, $B$, trace)
Input: Current partial stack flow currentSF.
Input: Next traversing basic block $B$.
Input: Current trace of basic blocks trace.
Output: Composite stack flows in current context.
begin
    if $B$ is connecting block then
        foreach Basic block $B_k$ directly reachable from $B$ do
            if $B_k$ makes cycle in trace then /* Skip cycle. */
                continue;
            else /* Forward to the next basic block. */
                nextSF ← concat (currentSF, body ($B$));
                nextTrace ← concat (trace,$k$);
                Call AddCompositeFlow(nextSF, $B_k$, nextTrace);
    else if $B$ is closing block then /* End of composite stack flow. */
        StackFlow ← concat (currentSF, tail ($B$));
        Add StackFlow to the stack flow set;
```

Algorithm 2: Algorithm for analyzing simple and composite stack flows.

cause stack flows may reflect the procedures of source code, similar stack flows in two programs indicate that the two programs are also similar, which is strong evidence that one of the two programs is a copy.

3.3 Lowering Stack Statuses

The stack flows of a Java program are constructed by analyzing the stack statuses of bytecodes. In the process of performing specific tasks during program execution, the movements of statuses are strong enough to endure program modifications, such as obfuscation or optimization. However, program modification may change stack flows by raising statuses by reordering or pushing auxiliary objects before a sequence of original stack flows and popping the objects after finishing the sequence.

For example, a piece of code, which initializes two field variables, can be compiled into bytecode by a Java compiler as follows:

```
0: (iconst_1, 1)
1: (istore_1, 0)
2: (iconst_2, 1)
3: (istore_2, 0)
      . . .
```

This code sequence initializes the first and the second field variables with 1 and 2, respectively. Suppose that we transform the program using some obfuscators, such as Smokescreen. This modification may reorder the program as follows:

```
0: (iconst_2, 1)
1: (iconst_1, 2)
2: (istore_1, 1)
3: (istore_2, 0)
      . . .
```

The two code sequences perform the same operations, but their stack appearances are different.

In summary, a sequence of several stack flows may be transformed to one longer stack flow as shown in Fig. 2. This modification makes the program more difficult to decompile or analyze, and it makes the granularity of stack flows inconsistent. To compare stack flows successfully, it is preferable to make their granularity consistent. To address this situation, we lower statuses in such a way that stack flows can be normalized. Figure 2 shows the effect of raising and lowering stack flows.

A stack flow is lowered if the following conditions are satisfied:

- The bytecode length of the stack flow is longer than

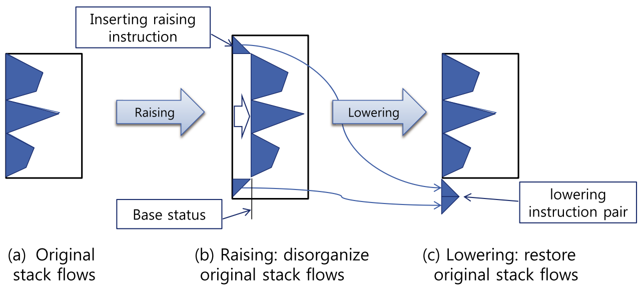

Inserting raising instruction
Raising
Lowering
Base status
lowering instruction pair
(a) Original stack flows
(b) Raising: disorganize original stack flows
(c) Lowering: restore original stack flows

Fig. 2 Raising and lowering stack flows.

some threshold value α.

- Stack movements fluctuate on the basis of a specific base status that can be a base position for lowering.
- The status of the raising instruction is in accordance with that of its counterpart.

If these conditions are satisfied, a raising instruction to push objects and its counterpart are paired as an extra stack flow. Then, the statuses of remaining instructions are lowered by the base status.

Lowering statuses has several advantages.

- Lowering stack statuses can balance the granularity of stack flows. Comparing stack flows is more effective when the number and size of stack flows are at uniform levels. A large stack flow that is raised from several stack flows is not a basic structure to represent a unit operation; rather, the combined structure represents several operations in a larger one.
- The body of a raised stack flow may be a sequence of conditional branches, such as `iflt` or `ifge`. In this case, the lengths of stack flows increase, and the number of stack flows increases exponentially, that is, the number of possible paths across the branch instructions. This complication causes a serious increase in the comparison time. Lowering stack statuses relieves time overhead by dividing a large combined structure into several uniform structures.

3.4 Matching stack flows

It is necessary to examine all stack flows to ascertain the similarity between two programs. The first step, when comparing two Java programs, is to determine a match among stack flows in each program. This matching summarizes the similarity between stack flows of two programs. Stack flows originating from the same source of an identical program may be different because compilation and optimization environments may generate different binary code. Therefore, in comparing stack flows, partial matching should be considered so that the matching can tolerate such subtle distinction.

In comparing stack flows, the semi-global alignment algorithm [26] was applied. To align two instruction sequences, the following operations are necessary:

- *Matches* for matching elements,
- *Mismatches* to align different elements in both sequences, and
- *Gaps* to align one mismatched element in one of the sequences.

The procedure for comparing stack flows is accomplished by dynamic programming as shown in Fig. 3. Let $a = \langle (x_1, s_1), \cdots, (x_m, s_m)\rangle$ be a stack flow in an original program, and let $b = \langle (y_1, t_1), \cdots, (y_n, t_n)\rangle$ be a stack flow in a target program. Beginning at the top left cell, the score up to position (i, j), $c[i, j]$, is calculated as

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

where $w(x_i)$ denotes the matching weight of the bytecode x_i, σ denotes the mismatching penalty for one mismatched pair of elements in both sequences, and gap denotes the penalty for skipping a mismatched element in one of the sequences. The matching weight of a bytecode is applied with $w = 1$, and the penalty values are applied with $\sigma = -1$ and $gap = -1$.

After the score for each cell is calculated, the resulting value of semi-global alignment is obtained by finding the maximum value among values in the bottom row as follows:

$$\textit{Semi-Global}(a, b) = \max_j(c[m, j]).$$

By penalizing for mismatches and gaps, this alignment can distinguish unrelated stack flows with high sensitivity. In addition, it is appropriate for comparing original stack flows with modified ones that may be augmented with some additional elements, because this method does not consider penalties caused by heading and tailing mismatches of one sequence.

Figure 3 shows an example of matching two stack flows. Figure 3 (a) shows two stack flows a and b. For some method f and variable x, the stack flows a and b correspond to expressions $x \times f(x-1)$ and $f(x-2) + f(x-1)$, respectively. This table shows sequences of bytecodes, the statuses of the bytecodes, and the stack appearance after execution of the bytecode. Figure 3 (b) shows the procedure to align two stack flows using semi-global alignment. The traces in the grid provide a method to compute a resulting value to align two stack flows by placing the score at the position of each cell. In Fig. 3 (b), the maximum score among values in the bottom row is the resulting score for aligning two stack flows a and b. From the calculation, the *matching score* of aligning the two stack flows is 3.

3.5 Comparing Java Programs

In comparing two Java programs, the overall similarity can be calculated by finding similar pairs from each stack flow

Stack Flow a				Stack Flow b			
Bytecode	pre-s.	post-s.	Stack	Bytecode	pre-s.	post-s.	Stack
`iload_0`	0	1	[•]	`iload_0`	0	1	[•]
`iload_0`	1	2	[••]	`iconst_2`	1	2	[••]
`iconst_1`	2	3	[•••]	`isub`	2	1	[•]
`isub`	3	2	[••]	`invokestatic`	1	1	[•]
`invokestatic`	2	2	[••]	`iload_0`	1	2	[••]
`imul`	2	1	[•]	`iconst_1`	2	3	[•••]
`ireturn`	1	0	[]	`isub`	3	2	[••]
				`invokestatic`	2	2	[••]
				`iadd`	2	1	[•]
				`ireturn`	1	0	[]

(a) Sample stack flows and the statuses of bytecodes.

		(`iload_0`, 1)	(`iconst_2`, 2)	(`isub`, 1)	(`invokestatic`, 1)	(`iload_0`, 2)	(`iconst_1`, 3)	(`isub`, 2)	(`invokestatic`, 2)	(`iadd`, 1)	(`ireturn`, 0)
	0	0	0	0	0	**0**	0	0	0	0	0
(`iload_0`, 1)	−1	1	0	−1	−1	**−1**	−1	−1	−1	−1	−1
(`iload_0`, 2)	−2	0	−1	−2	−2	**0**	−1	−2	−2	−2	−2
(`iconst_1`, 3)	−3	−1	−1	−2	−3	−1	**1**	0	−1	−2	−3
(`isub`, 2)	−4	−2	−2	−2	−3	−2	0	**2**	1	0	−1
(`invokestatic`, 2)	−5	−3	−3	−3	−3	−3	−1	1	**3**	2	1
(`imul`, 1)	−6	−4	−4	−4	−4	−4	−2	0	2	**2**	1
(`ireturn`, 0)	−7	−5	−5	−5	−5	−5	−3	−1	1	1	**3**

(b) Semi-global alignment: calculating matching score between two stack flows.

Fig. 3 Sample stack flows and matching algorithm.

set. This is accomplished by searching for matched pairs in a manner that maximizes the sum of matching scores. This problem can be considered as an extension of the stable marriage problem (SMP) [27], [28], which is the problem of finding a stable matching between elements in two sets. In this application, the SMP is extended in such a way that preferences to elements in the other set are represented not by ordering numbers but by matching scores to the elements. The preferences (matching scores) are mutually identical between pairs, and the problem can be solved in a greedy manner, which can find a stable matching in $O(n^3)$ by selecting pairs in descending order of their matching scores. These results give $\min(n, m)$ pairs of matched stack flows, and this stable matching is termed the *matching set* between two programs. Intuitively, the matching set represents a set of the most similar pairs of stack flows among all pairs between the two programs. Thus, the set is expected to collect the pairs of stack flows that have an identical origin.

For Java programs P and Q, let $SF(P)$ and $SF(Q)$ be the stack flow sets of P and Q, respectively. The similarity between P and Q is then calculated as

$$Similarity(P,Q) = \frac{\sum_{(a,b)\in \mathsf{M}(SF(P),SF(Q))} score(a,b)}{\min(\sum_{b\in SF(P)} |b|, \sum_{b\in SF(Q)} |b|)},$$

where $\mathsf{M}(SF(P), SF(Q))$ denotes a matching set between two programs, $|b|$ denotes the number of bytecodes in the stack flow b, and $score(a, b)$ denotes the matching score between a and b. The similarity is the sum of the matching scores of all matched pairs. The sum is then divided by the minimum of the total number of bytecodes in the stack flows to normalize the similarity. Because software may be modified by introducing bogus instructions or control flows, the minimum of two values is applied so that the similarity can reflect a containment relationship between the two programs. If two stack flow sets are fully matched, the matching score between two programs becomes equal to this denominator. Hence, the resulting similarity ranges between 0 and 1 in proportion to the degree of similarity. This measure represents the ratio of fractions of an original program that are contained in those of a target program. If an author believes that his modules were reused in others, it is a simple matter to compare the stack flows of the original modules with the modules of suspicious programs to confirm the fact.

4. Experimental Results

4.1 Preliminaries

In this section, the proposed method is evaluated with respect to two criteria: (1) discrimination between independent programs and (2) detection of modified programs. The proposed method was implemented in C on MS Windows XP and then evaluated on a PC system on an Intel Core i7 920 (2.66 GHz) processor with 12 GB RAM.

To evaluate the performance of the proposed method, we established an experimental environment for benchmark programs. For the purpose, we chose several Java programs in various categories from the website

Table 1 Specification of benchmark programs.

(a) Benchmark category and evaluated programs.

Category	Programs
(1) Source Control	jCVS, JSVN, SourceJammer, StatCvs, StatSvn, SuperVersion
(2) Database	Axion, Metanotion BlockFile, db4o, Hypersonic SQL, Berkeley DB, JODB, Mckoi SQL DB, NeoDatis ODB, TinySQL
(3) Code Utility	ObjectWeb ASM, BCEL, BeautyJ, Classycle, Clirr, GroboCodeCoverage, Javassist, Jdepend
(4) Obfuscator	Jarg, JavaGuard, JODE, ProGuard, yGuard
(5) SQL Client	DBBrowser, DBSA, Datastream Pro, GUAM, Henplus, MyJSQLView, SQL Admin, SQLeonardo, SQL-Shell, ViennaSQL,
(6) Parser	HTML Parser, NekoHTML, Apache Betwixt, JOX, NekoPull, Skaringa, Woodstox, XOM, XP Parser, Xstream
(7) Parser Generator	Beaver, Chaperon, Grammatica, JavaCC, Jparsec, Runcc
(8) Etc	ArgParser, BlueJ, C-JDBC, Elvyx, Jedit, Jspider, jTDS, Web-Harvest, WebSPHINX

(b) The status of stack flows according to category.

	# of class files		Avg. #	Total #	# of SFs/class			# of bytecodes/SF		
Category	Total	Eval.	bytecodes	SFs	Avg.	Min.	Max.	Avg.	Min.	Max.
(1) Source Control	3633	367	1069.0	79364	216.2	12	927	4.45	2	1028
(2) Database	4064	392	1187.6	99950	254.9	27	1039	4.17	2	130
(3) Code Utility	2561	190	1106.2	41389	217.8	28	1006	4.84	2	85
(4) Obfuscator	1554	136	1282.5	32669	240.2	27	743	4.68	2	252
(5) SQL Client	1821	161	1020.7	32193	199.9	70	849	4.62	2	71
(6) Parser	1345	122	1146.4	26033	213.3	24	779	4.36	2	393
(7) Parser Generator	814	71	1011.7	13144	185.1	27	720	4.48	2	33
(8) Etc	2656	256	1217.7	61995	242.1	23	940	4.48	2	120
Tot.	18448	1695	1138.8	386737	228.2	12	1039	4.45	2	1028

`http://Java-Source.net`. Each category is determined by the functionality of programs, and each category contains several benchmark programs. Table 1 (a) shows the categories and the programs in these categories that were used as benchmark program sets. The 63 Java benchmark programs are grouped into eight categories. For effective measurement and comparison of the performance of several approaches, it is important to organize the benchmark programs so as to evaluate the capability of code clone detection. A Java program archive consists of many Java class files, which have various roles in a package. Therefore, we tried to concentrate on the main execution modules of each program, which may be the target of code clone. To locate the main modules of Java programs, we investigated the bytecode distribution from 144,198 class files collected from the Internet. Figure 4 shows the numbers of cumulative class files and the cumulative ratio of bytecodes according to bytecode numbers in each class file. The gray region of the distribution graph shows the area of the benchmark set. As this distribution shows, 90% of the class files contain only 500 bytecodes or fewer, and these class files are excluded from evaluation because they are estimated to be too small to be core modules. On the contrary, excessively large files are inadequate to evaluate performance by direct comparison with small files. Therefore, we chose the class files with the range of bytecode numbers between 500 and 5000. This range corresponds to only 10% of the total class files, while the range covers 50% of the total bytecodes in each package.

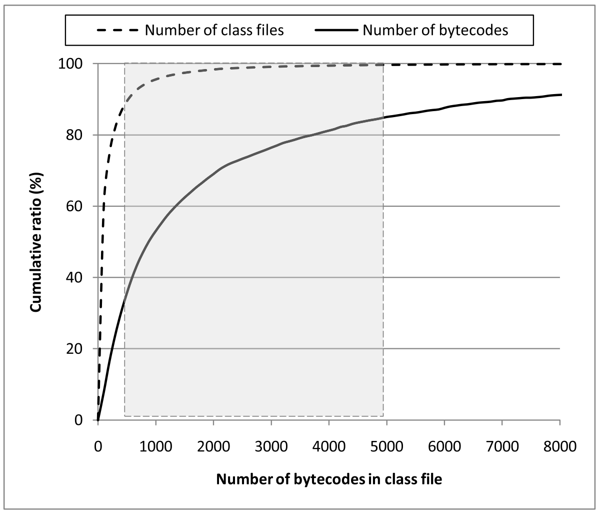


Fig. 4 Distribution graph of Java class files according to the number of bytecodes.

Table 1 (b) shows the statistical characteristics collected from benchmark programs in each category. The table shows the total number of class files and evaluated stack flows from the class files with their statistical characteristics. There were 1695 class files chosen out of 18448 class files

Table 2 Experimental results for evaluating discrimination capability.

	# of	Tamada			k-gram			Stack Pattern			SF-based			
Cat.	Cmp.	Avg.	Max.	Pos. (%)	Avg.	Max.	Pos. (%)	Avg.	Max.	Pos. (%)	Avg.	Max.	Pos. (%)	Time [a]
(1)	48008	0.19	1.00	54 (0.1)	0.27	1.00	113 (0.3)	0.21	1.00	123 (0.3)	0.29	1.00	129 (0.3)	0.43 (7.58)
(2)	63208	0.21	1.00	1 (0.0)	0.28	0.97	1 (0.0)	0.22	1.00	2 (0.0)	0.30	1.00	6 (0.0)	0.53 (8.27)
(3)	14500	0.11	1.00	42 (0.3)	0.27	1.00	52 (0.4)	0.23	1.00	84 (0.6)	0.29	1.00	84 (0.6)	0.43 (7.13)
(4)	6939	0.17	0.75	0 (0.0)	0.24	0.90	2 (0.0)	0.20	0.97	1 (0.0)	0.26	0.97	1 (0.0)	0.54 (5.06)
(5)	10916	0.17	0.56	0 (0.0)	0.31	0.52	0 (0.0)	0.26	0.69	0 (0.0)	0.31	0.67	0 (0.0)	0.38 (5.41)
(6)	6082	0.16	0.71	0 (0.0)	0.24	0.53	0 (0.0)	0.19	0.56	0 (0.0)	0.31	0.93	22 (0.4)	0.42 (5.25)
(7)	1997	0.28	0.67	0 (0.0)	0.24	0.48	0 (0.0)	0.19	0.61	0 (0.0)	0.26	0.81	2 (0.1)	0.30 (2.33)
(8)	26487	0.19	0.67	0 (0.0)	0.26	0.54	0 (0.0)	0.22	0.79	0 (0.0)	0.29	0.82	1 (0.0)	0.51 (6.08)
Tot.	178137	0.19	1.00	97 (0.1)	0.27	1.00	168 (0.1)	0.22	1.00	210 (0.1)	0.29	1.00	245 (0.1)	0.48 (8.27)

[a] Average (and maximum) time measured in seconds.

from 8 categories. The average number of bytecodes for the evaluated class files was 1138.8, and the numbers ranged between 1011.7 and 1282.5 depending on the category. The table also shows the specifications of evaluated stack flows for benchmark programs in each category.

4.2 Discrimination between Independent Programs

In this section, we evaluate the proposed method in terms of discrimination between independent programs. This criterion requires a software comparison method to distinguish independently developed programs. The Java programs described in Table 1 (a) are used as a benchmark program set, and individual class files contained in the programs are used as target modules. For this experiment, target modules are compared with modules in other programs within the same category. In this experiment, we evaluate whether the proposed method can distinguish different modules although programs in same category have similar functionality. To measure discrimination capability, the threshold value ϵ for determining code clones was set as $\epsilon = 0.2$, which was determined to maximize precision and recall in the detection of common modules in Java programs [12]. To evaluate and compare performance of the proposed method, Stigmata 2.0[†] was used for the Tamada birthmark [2], [3] and the k-gram birthmark [4], [5]. We also compared the performance of the stack pattern birthmark [6].

Table 2 summarizes the results of this experiment. This table shows the total number of comparisons between Java class files within a given category and the average and maximum similarity values between pairs. The heading "Pos." indicates the numbers and ratios of pairs detected as clones (positive results) by the four approaches. The heading "Time" indicates the average and maximum times taken to compare a pair of programs for the proposed method. In the results, average similarity values make no difference, but a false-positive rate is much more important for evaluating the credibility of a software comparison method. The proposed method had much more positive results than the other approaches. In our investigation of the positive results, we found several reasons for this. In the Source Control category, the positive results came from comparisons between StatCvs and StatSvn. StatSvn is based on StatCvs, so it inherits most of its modules from StatCvs. StatSvn and StatCvs also share JFreeChart to generate charts as a common library. Thus, the positive results were not false positives, but they were common modules used in the two programs. In the results, the proposed method detected more pairs of common modules than the other approaches. In the Database category, Axion and Mckoi SQL DB share `SimpleCharStream.class` as a common module, so the k-gram birthmark, the stack pattern birthmark, and the proposed method had one positive result. However, the stack pattern birthmark and the proposed method had 1 and 5 false positives in this category, respectively. In the Code Utility category, Clirr and GroboCodeCoverage contained BCEL as a common library, so comparisons between Clirr, GroboCodeCoverage, and BCEL showed many positive results, which were not false results. The difference between the number of positive results was due to version differences of the embedded BCEL library. The proposed method found more pairs of common modules than the Tamada birthmark or the k-gram birthmark. In the Obfuscator category, JavaGuard and yGuard shared `KeywordNameMaker.class` as a common module, so the k-gram birthmark, stack pattern birthmark, and the proposed method each had one positive result. An additional positive result for the k-gram birthmark is considered to be a false positive. In the remaining categories, the proposed method showed 25 false positives, corresponding to 0.05% of all comparisons.

We investigated to find the main reason for the pairs of false positives with the proposed method. The false positive results between different programs are due to program modules, which are composed of only frequent patterns of stack flows in general Java programs. Typical examples are variable initialization, object initialization, and method invocation, which do not contain apparent characteristics, such as control flows of jump and loop, or arithmetic operations.

When comparing the results of the four approaches, the average similarities made no apparent difference. The false-positive rate of the proposed method was slightly higher than other approaches, but the false-positive rate was a neg-

[†] Java birthmark toolkit, http://stigmata.sourceforge.jp/

ligible value. In cases where common modules are contained, however, the proposed method found more pairs of modules than the other approaches.

4.3 Detection of Modified Programs

In this section, the proposed method is evaluated in terms of detection of modified programs. Its performance is compared with the Tamada birthmark [2], [3], the *k*-gram based birthmark [4], [5], and the stack pattern birthmark [6]. The Java programs listed in Table 1 (a) were used as benchmark programs, and three transformation methods were applied to evaluate the detection of modified programs. Jarg[†] is a Java bytecode optimizer which optimizes a program by renaming or eliminating the unnecessary parts of the program. Smokescreen[††] and Zelix KlassMaster (ZKM)[†††] are Java program obfuscators that transform an original Java program into an equivalent program that is more difficult to decompile or analyze. Smokescreen and ZKM perform several transformations, including control flow modification, renaming, and string encryption.

Evaluation of detection capability is performed by comparing original class files in a benchmark set with their own transformed versions. If the similarity values of comparisons are sufficiently high, we can confirm the detection capability of the comparisons. Initially, the original programs in the benchmark set described in Table 1 (a) were transformed using Jarg, Smokescreen, or ZKM with the strongest respective level of modification. The original programs and their modified versions were then compared by each method. The approaches were evaluated by measuring the similarity values and their variations caused by transformation methods.

Table 3 shows the results of comparisons of the four approaches with respect to the transformations of Jarg, Smokescreen, and ZKM. The table shows the total number of comparisons, average similarity, and the minimum similarity values for each method according to the category. The heading "FN" indicates the number of false negatives and ratios while comparing original class files and their own modified versions.

The heading "TV" represents transformation variation, which shows the degree of effect caused by transformation modifications. For a program P and a transformation method $\mathcal{T}$, the transformation effect caused by $\mathcal{T}$, $e^{\mathcal{T}}$, is calculated as

$$e^{\mathcal{T}}(P) = Similarity(P, P) - Similarity(P, P'),$$

where P' denotes a version of P modified by the transformation $\mathcal{T}$. For a benchmark set S, we calculated the transformation variation on S as

$$TV^{\mathcal{T}}(S) = \frac{\sum_{P\in S} (e^{\mathcal{T}}(P))^2}{|S|},$$

where $|S|$ denotes the cardinality of the set S. By raising the transformation effects to the second power, the degree of effect caused by a transformation was reflected in the transformation variation TV. In evaluating detection capability, a higher $TV^{\mathcal{T}}(S)$ indicates that the method is more affected by the transformation $\mathcal{T}$ on the benchmarking set S. Therefore, if a comparison method has a lower value of $TV^{\mathcal{T}}$ than others, then the method can be determined to be more effective in detecting programs modified by $\mathcal{T}$.

With the experiment on optimization by Jarg, the average similarities made no difference in any of the four approaches, but minimum similarities of the Tamada birthmark were located in lower ranges compared to those of the other three approaches. The other three approaches had false-negative rates lower than 1%, and low TV values ranged between 0.04 and 0.11. Jarg does not modify the sequence of bytecodes in Java programs, but renames class files and eliminates the unnecessary parts of programs. Therefore, the three birthmarks other than Tamada birthmark show almost equivalent performance in these experiments. However, the Tamada birthmark showed high false-negative rates and higher TV values than those of the other approaches. The renaming optimization of the Jarg transformation deteriorates the resilience of the Tamada birthmark because it affects the used classes (UC) and the inheritance structures (IS) of the birthmark.

In the experiments using Smokescreen and ZKM, the performance of the methods showed more differences because the two obfuscators modify class files by means of several aggressive transformations, such as reordering control flows, exchanging bytecodes, renaming identifiers, and encrypting strings. The average similarities of the Tamada birthmark and the *k*-gram based birthmark were similar, and the average similarities of the stack pattern birthmark were higher than those of the two birthmarks in experiments with ZKM. The proposed method had the highest similarity values among the four methods. For the false-negative rates, the proposed method showed 16.4% and 1.6% for Smokescreen and ZKM, respectively. However, the Tamada birthmark and *k*-gram based birthmark missed a considerable number of programs transformed by Smokescreen and ZKM. The stack pattern birthmark showed better results than Tamada birthmark or *k*-gram birthmark in the experiments with ZKM, but it did not in the experiments with Smokescreen. The average TV values of the proposed method for Smokescreen and ZKM were 0.16 and 0.10, respectively, and they were the lowest values among the four approaches.

In the case of the Tamada birthmark, the UC, IS, and SMC birthmarks are dependent on the comparison of the names of classes, methods, or identifiers in programs. Therefore, if names are modified by some transformation, the birthmarks are seriously impaired.

The *k*-gram based birthmark compares *k* consecutive

[†]Java archive grinder. http://sourceforge.net/projects/jarg/

[††]Smokescreen Java obfuscator. http://www.leesw.com/smokescreen/

[†††]Zelix KlassMaster. http://www.zelix.com/klassmaster/index.html

Table 3 Experimental results for evaluating detection capability.

(a) Experimental results: detection of code clones modified by Jarg. Because of transformation failure, the following programs were excluded in this evaluation: db4o, Berkeley DB, JODB, NeoDatis ODB, DBBrowser, DBSA, SQLShell, Xstream, and Jparsec.

	# of	Tamada				*k*-gram				Stack Pattern				SF-based			
Cat.	Cmp.	Avg.	Min.	TV	FN (%)	Avg.	Min.	TV	FN (%)	Avg.	Min.	TV	FN (%)	Avg.	Min.	TV	FN (%)
(1)	367	0.86	0.52	0.18	163 (44.4)	0.92	0.91	0.08	0 (0.0)	0.93	0.78	0.08	4 (1.1)	0.96	0.87	0.04	0 (0.0)
(2)	180	0.95	0.67	0.10	19 (10.5)	0.90	0.69	0.11	4 (2.2)	0.93	0.80	0.08	0 (0.0)	0.96	0.82	0.05	0 (0.0)
(3)	190	0.92	0.63	0.12	25 (13.1)	0.92	0.82	0.08	0 (0.0)	0.94	0.76	0.08	2 (1.1)	0.96	0.88	0.05	0 (0.0)
(4)	136	0.95	0.52	0.11	21 (15.4)	0.94	0.84	0.07	0 (0.0)	0.95	0.75	0.07	2 (1.5)	0.97	0.71	0.05	1 (0.7)
(5)	128	0.92	0.63	0.13	25 (19.5)	0.92	0.75	0.09	1 (0.7)	0.93	0.72	0.09	1 (0.8)	0.96	0.88	0.05	0 (0.0)
(6)	114	0.94	0.37	0.14	17 (14.9)	0.92	0.82	0.09	0 (0.0)	0.93	0.83	0.08	0 (0.0)	0.96	0.91	0.04	0 (0.0)
(7)	68	0.95	0.75	0.10	8 (11.7)	0.93	0.81	0.08	0 (0.0)	0.94	0.79	0.07	1 (1.5)	0.97	0.91	0.04	0 (0.0)
(8)	201	0.96	0.59	0.08	13 (6.4)	0.92	0.83	0.08	0 (0.0)	0.94	0.79	0.07	1 (0.5)	0.97	0.89	0.04	0 (0.0)
Tot.	1384	0.92	0.37	0.13	291 (21.0)	0.92	0.69	0.09	5 (0.4)	0.93	0.72	0.08	11 (0.8)	0.96	0.71	0.05	1 (0.1)

(b) Experimental results: detection of code clones modified by Smokescreen. Because of transformation failure, the following programs were excluded in this evaluation: Berkeley DB.

	# of	Tamada				*k*-gram				Stack Pattern				SF-based			
Cat.	Cmp.	Avg.	Min.	TV	FN (%)	Avg.	Min.	TV	FN (%)	Avg.	Min.	TV	FN (%)	Avg.	Min.	TV	FN (%)
(1)	367	0.76	0.24	0.26	288 (78.4)	0.72	0.48	0.29	350 (95.3)	0.71	0.03	0.33	254 (69.2)	0.86	0.34	0.17	62 (16.8)
(2)	278	0.67	0.33	0.35	250 (89.9)	0.69	0.02	0.33	257 (92.4)	0.74	0.00	0.30	157 (56.5)	0.89	0.46	0.15	35 (12.5)
(3)	190	0.56	0.07	0.46	182 (95.7)	0.70	0.18	0.31	183 (96.3)	0.71	0.00	0.35	126 (66.3)	0.88	0.35	0.15	26 (13.6)
(4)	136	0.58	0.10	0.44	130 (95.5)	0.72	0.12	0.29	124 (91.1)	0.76	0.00	0.31	60 (44.1)	0.88	0.34	0.15	18 (13.2)
(5)	161	0.72	0.50	0.29	154 (95.6)	0.67	0.48	0.34	161 (100)	0.62	0.06	0.42	137 (85.1)	0.82	0.44	0.20	60 (37.2)
(6)	122	0.65	0.33	0.37	112 (91.8)	0.68	0.43	0.32	118 (96.7)	0.80	0.31	0.22	57 (46.7)	0.91	0.45	0.12	7 (5.7)
(7)	71	0.69	0.37	0.33	65 (91.5)	0.67	0.13	0.35	69 (97.1)	0.70	0.00	0.37	41 (57.7)	0.87	0.39	0.17	15 (21.1)
(8)	256	0.70	0.36	0.32	234 (91.4)	0.72	0.13	0.29	231 (90.2)	0.72	0.00	0.05	159 (62.1)	0.87	0.44	0.16	37 (14.4)
Tot.	1581	0.68	0.07	0.35	1415 (89.5)	0.70	0.02	0.31	1493 (94.4)	0.71	0.00	0.33	991 (62.7)	0.87	0.34	0.16	260 (16.4)

(c) Experimental results: detection of code clones modified by Zelix KlassMaster. Because of transformation failure, the following programs were excluded in this evaluation: StatCvs, StatSvn, Hypersonic SQL, GroboCodeCoverage, ProGuard, NekoHTML, Woodstox, XOM, and WebSPHINX.

	# of	Tamada				*k*-gram				Stack Pattern				SF-based			
Cat.	Cmp.	Avg.	Min.	TV	FN (%)	Avg.	Min.	TV	FN (%)	Avg.	Min.	TV	FN (%)	Avg.	Min.	TV	FN (%)
(1)	102	0.89	0.33	0.18	14 (13.7)	0.75	0.54	0.27	79 (77.4)	0.91	0.44	0.13	7 (6.9)	0.94	0.79	0.07	1 (0.9)
(2)	324	0.84	0.52	0.19	104 (32.0)	0.80	0.16	0.23	141 (43.5)	0.94	0.04	0.11	11 (3.4)	0.96	0.11	0.08	4 (1.2)
(3)	131	0.50	0.01	0.57	102 (77.8)	0.78	0.28	0.24	80 (61.0)	0.93	0.65	0.10	5 (3.8)	0.94	0.10	0.11	1 (0.7)
(4)	96	0.74	0.01	0.36	45 (46.8)	0.79	0.24	0.23	47 (48.9)	0.94	0.64	0.10	5 (5.2)	0.94	0.21	0.10	2 (2.0)
(5)	161	0.85	0.34	0.19	53 (32.9)	0.71	0.38	0.30	144 (89.4)	0.88	0.51	0.15	20 (12.4)	0.92	0.74	0.09	5 (3.1)
(6)	46	0.78	0.27	0.31	19 (41.3)	0.75	0.57	0.27	34 (73.9)	0.91	0.32	0.14	4 (8.7)	0.93	0.63	0.09	1 (2.1)
(7)	71	0.82	0.60	0.21	27 (38.0)	0.72	0.23	0.31	47 (66.1)	0.92	0.62	0.12	8 (0.11)	0.91	0.11	0.18	3 (4.2)
(8)	229	0.88	0.50	0.16	48 (20.9)	0.76	0.23	0.26	138 (60.2)	0.92	0.66	0.10	11 (4.8)	0.94	0.13	0.09	2 (0.8)
Tot.	1160	0.80	0.01	0.28	412 (35.5)	0.77	0.16	0.26	710 (61.2)	0.92	0.04	0.11	71 (6.1)	0.94	0.10	0.10	19 (1.6)

opcode sequences in programs, and this method mainly depends on opcode sequences in comparing two programs. Therefore, it is susceptible to control flow obfuscation or code reordering. So, the *k*-gram based birthmark showed high false-negative rates and high TV values in aggressive modification environments where the order of opcodes is changed by control flow modification.

The stack pattern birthmark analyzes operand stack behaviors as the proposed method; however, it only considers textually contiguous sequences of opcodes without considering control flows of programs. Therefore, it is also susceptible to control flow obfuscation or code reordering. In the experiments, the stack pattern birthmark showed low grades in the experiments with Smokescreen, which applied control flow modification or code reordering frequently.

The proposed method showed lower false-negative rates and TV for Smokescreen and ZKM than the other three approaches. These results are strong evidence that the method is more effective in detecting code clones modified by Smokescreen or ZKM. Because the proposed method follows the control flows of programs, it can tolerate code reordering or control flow obfuscation. Because the method compares opcodes and stack operations, it may be susceptible to program transformation that changes opcodes or stack operations directly.

5. Discussion and Future Work

For practical use, it is important to decrease the number of false negatives. In the experimental evaluation in Sect. 4, there were several cases of false negatives. For such cases, we investigated the reasons for low similarities.

- First, it was noted that obfuscation changed a number of patterns of bytecode sequences into explanatory expressions. For example, the bytecode `ldc` can be changed to the sequence of `getstatic`, `iconst_n`, `aaload` or `getstatic`, `bipush`, `aaload`. If the modification occurs in the middle of a bytecode sequence, semi-global alignment can fail to align the two sequences on account of penalty values for aligning the mismatched bytecodes. To overcome this defect, it is necessary to refine the alignment algorithm by matching patterns that can be modified or substituted.
- Second, it was noted that obfuscation changed the sequence of bytecodes by means of instruction reordering or stack operation reordering. For example, Smokescreen transforms a sequence of pairs of `load` and `store` instructions into a sequence of `store` instructions after `load` instructions. As the proposed method compares sequences of bytecodes, the comparison method may be ineffective if bytecode sequences have been reordered.
- Third, a Java program package often contains constant initialization modules, which mainly initialize objects or field variables. The control flow of bytecodes in such modules constructs one long sequence of uniform patterns, which may be lowered to regular patterns of stack flows. However, control flow obfuscation can change these long sequences of control flows into irregular patterns of code sequences by control flow flattening. In these cases, the proposed method did not match irregular stack flow patterns in the modified versions.
- Finally, it was noted that the obfuscation inserted several bogus instructions with the help of opaque predicates [29], [30], which were always evaluated as either `true` or `false`. These modifications made some sequences of bytecodes dissimilar.

These problems occur because the method compares sequences of opcodes directly. More customized analysis is required to overcome these problems. Instead of direct matching of bytecodes in stack flows, comparing abstract behaviors of stack flows is likely to improve the detection capability. For a future work, we plan to analyze the abstract behaviors of stack flows, and refine the matching scheme so that it can reflect the meaning of the stack flows. In addition, we plan to organize data flow relation between stack flows and characterize a Java program by using flow graphs of stack flows. By comparing stack flow graphs, code clones can be identified more effectively.

6. Conclusion

Software is an intellectual property that must be protected against license violations; however, the incidence of software license violations increases every year. To cope with this incidence, there have been several methods of software comparison. This paper proposes a method to detect clones of Java programs by analysis of the stack flows of programs. A stack flow denotes a minimal sequence of bytecodes that performs specific tasks in the common context of the operand stack. Stack flows are presented and formally described by simulating operand stack movements. The semi-global algorithm was used to align two stack flows in each program, and the similarity between two programs was calculated by determining a set of the most similar pairs of stack flows in the two programs.

To measure the two criteria of discrimination and detection capabilities, the proposed method was evaluated with respect to false-positive rates, false-negative rates, and transformation variations. The proposed method was also compared with the earlier approaches of the Tamada birthmark, the *k*-gram based birthmark, and the stack pattern birthmark. Experimental results showed that the performance of the proposed method was uniform without apparent weaknesses in relation to three different modification methods, namely, Jarg, Smokescreen, and ZKM. Our results demonstrate that the proposed method is more effective in detecting code clones than earlier approaches. Based on these results, we are convinced that the proposed method can lessen the time and effort required for manual reverse engineering in identification of code clones developed in Java.

References

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

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

[3] H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto, "Java birthmark –detecting the software theft," IEICE Trans. Inf. & Syst., vol.E88-D, no.9, pp.2148–2158, Sept. 2005.

[4] G. Myles and C. Collberg, "k-gram based software birthmarks," Proc. 2005 ACM Symposium on Applied Computing, pp.314–318, 2005.

[5] G.M. Myles, Software Theft Detection Through Program Identification, Ph.D. thesis, Department of Computer Science, The University of Arizona, 2006.

[6] H. Lim, H. Park, S. Choi, and T. Han, "Detecting theft of java applications via a static birthmark based on weighted stack patterns," IEICE Trans. Inf. & Syst., vol.E91-D, no.9, pp.2323–2332, Sept. 2008.

[7] B.S. Baker and U. Manber, "Deducing similarities in java sources from bytecodes," ATEC '98: Proc. the annual conference on USENIX Annual Technical Conference, pp.179–190, Berkeley, CA, USA, 1998.

[8] U. Manber, "Finding similar files in a large file system," the USENIX Winter 1994 Technical Conference, pp.1–10, Jan. 1994.

[9] B.S. Baker, "Parameterized pattern matching: Algorithms and applications," Journal of Computer and System Sciences, vol.52, no.1, pp.28–42, Feb. 1996.

[10] H. Lim, H. Park, S. Choi, and T. Han, "A method for detecting the theft of java programs through analysis of the control flow information," Inf. Softw. Technol., vol.51, no.9, pp.1338–1350, 2009.

[11] H. Lim, H. Park, S. Choi, and T. Han, "A static java birthmark based on control flow edges," Computer Software and Applications Conference, Annual International, vol.1, pp.413–420, 2009.

[12] H. Park, H. Lim, S. Choi, and T. Han, "Detecting common modules in java packages based on static object trace birthmark," The

Computer Journal, vol.54, no.1, pp.108–124, 2011.

[13] T.F. Smith and M.S. Waterman, "Identification of common molecular subsequences," J. Molecular Biology, vol.147, no.1, pp.195–197, March 1981.

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

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

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

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

[18] C. Brown, D. Barrera, and D. Deugo, "Figd: An open source intellectual property violation detector," SEKE, pp.536–541, 2009.

[19] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," Proc. Eighteenth International Symposium on Software Testing and Analysis, ISSTA '09, pp.117–128, New York, NY, USA, 2009.

[20] J. Krinke, "Identifying similar code with program dependence graphs," 8th Working Conference on Reverse Engineering (WCRE'01), pp.301–309, Stuttgart, Germany, Oct. 2001.

[21] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," Proc. 12th USENIX Security Symposium, pp.169–186, 2003.

[22] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in Detection of Intrusions and Malware & Vulnerability Assessment, ed. R. Büschkes and P. Laskov, LNCS, vol.4064, pp.129–143, Berlin, Germany, 2006.

[23] J. Gosling, B. Joy, G. Steele, and G. Bracha, The Java Language Specification, second ed., Addison-Wesley, June 2000.

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

[25] J. Zhao, "Analyzing control flow in java bytecode," Nippon Sofutowea Kagakkai Taikai Ronbunshu, vol.16, pp.313–316, 1999.

[26] M. Brudno, S. Malde, A. Poliakov, C.B. Do, O. Couronne, I. Dubchak, and S. Batzoglou, "Glocal alignment: finding rearrangements during alignment," Bioinformatics, vol.19, Suppl.1, pp.54–62, 2003.

[27] D. Gusfield and R.W. Irving, The Stable Marriage Problem: Structure and Algorithms, The MIT Press, 1989.

[28] D.E. Knuth, Stable Marriage and Its Relation to Other Combinatorial Problems, An Introduction to the Mathematical Analysis of Algorithms, American Mathematical Society, 1997.

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

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

Hyun-il Lim received his B.S., M.S., and Ph.D. degrees in computer science from KAIST, Korea, in 1995, 1997, 2009, respectively. He is currently a lecturer in the Division of Computer Science and Engineering, Kyungnam University. His current research interests include software security, software protection, watermarking, and program analysis.

Taisook Han received his B.S. degree in electronic engineering from Seoul National University, Korea in 1976, M.S. degree in computer science from KAIST, Korea, in 1978, and Ph.D. degree in computer science from University of North Carolina at Chapel Hill, USA, in 1990. He is currently a professor in the Department of Computer Science, KAIST. His research interests include programming language theory, secure software, and analysis of embedded systems.