Open Source Software Detection using Function-level Static Software Birthmark

Dongjin Kim[1], Seong-je Cho[1], Sangchul Han[2]*, Minkyu Park[2], and Ilsun You[3]
[1]Dankook University, Yongin 448-701, Korea
{kdjorang, sjcho}@dankook.ac.kr
[2]Konkuk University, Chungbuk 380-701, Korea
{schan, minkyup}@kku.ac.kr
[3]Korean Bible University, Seoul 138-791, Korea
isyou@bible.ac.kr

Abstract

As open-source software (OSS) is widely used, many IT organizations adopt OSS without obeying some guidelines for open-source license agreements. To reduce risks related to open-source licenses, the organizations should meet the requirements for OSS licenses. Because some OSS components may be given from major upstream suppliers in binary form, it is very hard to verify whether a binary program contains unlicensed OSS components. In this paper, we propose a novel technique for determining whether a binary includes certain OSS components without respecting the OSS licensing terms. Our technique employs function-level static software birthmark to detect code clones in binaries. In our technique, the birthmark is a sequence of the size information of arguments and local variables of functions inside a binary, and the similarity between birthmarks is computed using semi-global sequence alignment or k-gram method. We evaluate the effectiveness of the proposed techniques by performing experiments with some binaries and OSS components.

Keywords: Open-source software, Static analysis, Software birthmark, Sequence alignment

1 Introduction

Recently open source is ubiquitous and the use of Open-Source Software (OSS) components is proliferating. This is because many software developers contribute their time and effort to create high-quality open-source software. The success of OSS movement is evident from the large and growing number of software industry executives, managers, and academics who are trying to use OSS components. According to Gartner's findings, 75% of Global 2000 enterprises included OSS in mission-critical software portfolios in 2010, and going to 99% by 2016 [10, 8].

The main reason for using OSS is that it is free to use and distribute, which results in substantial financial benefits. OSS enables organizations to develop solutions that can meet their requirements best with low cost and effort. However, OSS used in IT organizations is distributed often in a way that it does not obey the licenses imposed by the original developers [8, 9]. This has caused many license violations, exposing the organizations to significant legal risk. Gartner predicted that 50% of Global 2000 organizations would experience technology, cost and security challenges through lack of open-source governance by 2014 [10, 8]. Many organizations have inadvertently violated a copyright law. Such organizations often receive software components from major upstream suppliers in binary form, and thus cannot verify whether they include unlicensed third-party code. Therefore, it is necessary for

Journal of Internet Services and Information Security (JISIS), volume: 4, number: 4 (November 2014), pp. 25-37
*Corresponding author: Dept. of Computer Engineering, 268 Chungwondaero, Chungju-si, Chungcheongbuk-do, 380-701, Tel: +82-(0)43-840-3605

downstream users or IT organizations to examine which third-party software (OSS), if any, is contained in binary files from upstream suppliers [9, 13].

Software birthmark is unique characteristics of a binary, which can be used to identify each binary and detect software theft [11, 7, 14, 17, 16]. It is also effective in detecting software component theft where only partial code is stolen [16, 6]. In this paper, we propose a novel software birthmarking technique that detects code clone in binaries. Our technique detects binary clones by comparing software birthmarks extracted from a target binary and OSS components for evidence that the target code contains specific OSS component(s). Our software birthmark is made up of the sizes of arguments[1] and local variables of functions inside binaries. For a program module or component with many functions, the sequence of the size information of arguments and local variables of the functions can be unique characteristics of the module or component. Note that it is almost impossible to obtain the type or the number of arguments and local variables of functions from released binaries.

The proposed technique tells us whether a specific third-party software component (OSS) is included in a target binary, that is, whether the birthmark of a component is similar to or the same as a partial birthmark of a target binary. Our technique detects binary clones by compare similarity of our birthmarks extracted from a target binary and OSS components. We use the semi-global alignment algorithm [5, 4] and k-grams [11] to compute the similarity between the birthmarks of an OSS component and a target binary. We carry out extensive experiments in order to verify the effectiveness of the proposed technique. Note that license management is beyond the scope of this paper such as examining what license should exist to protect OSS and determining any license violations may have taken place.

This paper is organized as follows. We first review background and related work in Section 2, and describe our software birthmarking technique to detect binary clones in Section 3. We then carry out extensive experiments and evaluate the results in Section 4. We finally conclude with a discussion of our future work in Section 5.

2 Related Work

2.1 OSS Detection

There are some tools for OSS detection and license verification. Black Duck Software's Protex [3] is a well-known commercial tool. It detects the use of OSS and identifies the software origin by source code scanning. FOSSology [2], initiated by HP, is an OSS under GPL v2 license. It detects the use of OSS based on the license information in the comments written in source codes. If the license information is modified or removed from source codes, FOSSology has difficulty in finding OSS. Binary Analysis Tool (BAT)[1] is an OSS under Apache 2 license. It inspects files that are in binary format to find out what software is inside. Since it reads symbol and string table in binary files and compares it with information extracted from source codes of OSSs, if symbol is removed during compilation or string is obfuscated or encoded, BAT cannot find OSS.

As mentioned above most previous OSS detection and license verification tools are based on source codes. They cannot be used if the source code of the target program is not available such as programs distributed via Internet and third-parity libraries. Binary-based analysis tools such as BAT make use of information that can be easily extracted from binaries. It is not difficult to circumvent such tools by removing or modifying the information. In this paper, from this perspective, we propose a software birthmark-based method that can easily extract/compare characteristics of programs and is resilient to modification.

[1] Throughout the paper, we use the terms 'argument' and 'parameter' interchangeably.

2.2 Software Birthmark

Software birthmark can be classified into two categories. Static birthmark [11, 7] is information that can be extracted from binary executables through static analysis. Dynamic birthmark [15, 12] is information collected or extracted during program execution. Static birthmark has relatively better code coverage and smaller extraction overhead than dynamic birthmark. Dynamic birthmark has not good code coverage and is dependent upon execution environments. However, it can collect information that can be obtained during execution such as the value of function arguments, return value, and indirected branch.

Myles and Collberg [11] applies k-gram technique, which is used in document similarity analysis, to the sequence of instructions (mnemonics) extracted from binary executables. This k-gram based static birthmark is vulnerable to obfuscation such as statement reordering or invalid instruction insertion. Moreover, since the size of birthmark is very large, the extraction and comparison overhead grows fast as the size of program becomes larger.

Tamada et al. [15] and Shuler and Dallmeier [14] propose API-based dynamic birthmark. Since the proposed API call sequence (EXESEQ) and frequency (EXEFREQ) are related to the functionality of program, the birthmarks are more resilient to various obfuscation technique. However, the reliability of the similarity measurement may be low if some API call are substituted with similar API call or unintended APIs are called, e.g. user-interaction related APIs. In addition, programs with similar functionality may have similar pattern of API calls. The birthmarks are also ineffective for programs that rarely call APIs such as small programs and algorithm-centric programs like mathemetical computation or encryption programs. Choi et al. [7] propose an API-based static birthmark. This technique has good code coverage and achieves fast analysis, but has the same limitation as API-based dynamic birthmark.

Wang et al. [16] proposes a system call-based dynamic birthmark. Since similar APIs (or library functions) eventually invoke the same system call, the birthmark is more resilient to modification attack exploiting similar APIs than API-based birthmark. However, it is subject to false positive and is applicable only to UNIX-like operating systems. This birthmark is ineffective for programs that rarely invoke system calls as the API-based birthmark is.

Myles and Collberg [12] propose Whole Program Path Birthmark (WPPB) for Java programs. WPPB is a dynamic control flow graph-based birthmark, which improves the graph comparison overhead with graph summarization. However, it is not suitable for similarity measurement between large scale programs. Zhou et al. [17] proposes a Component Dependence Graph (CDG) based static and dynamic birthmark, which can efficiently measure similarity between large scale programs. However, such graph-based birthmarks are not suitable for programs that have a few components or libraries.

The main purpose of our research is to detect a *small* OSS contained in a *large* binary program with *low* runtime overhead. The above mentioned techniques do not satisfy this condition. We need a new technique that can extract birthmark from binary executables, can measure similarity between a small birthmark and a part of large birthmark, and can do these fast.

3 Function-level Static Birthmark-based OSS Detection

To detect Software theft and piracy, existing birthmark-based methods perform 1-to-1 similarity analysis, so they do not take into account the space overhead of birthmark size or the time overhead of extracting and comparing birthmarks. Whereas, OSS detection is a 1-to-N similarity analysis; it needs to investigate a target program in order to search for many OSSs. From this perspective, this paper proposes a novel birthmarking and similarity analysis technique for OSS detection. This technique is based on static analysis and has low time and space overhead. It extracts characteristic of each function from OSS (or a target program) in a binary and build a birthmark. Then it examines whether the birthmark of OSS is similar to a part of the birthmark of the target program.

The strength of the proposed function-level static birthmark is as follows. First, by using function-level characteristics, the size of birthmark is small and the comparison of birthmarks is fast. The characteristics used in previous work, such as instruction, API information, control flow, and call graph, are not suitable for OSS detection. Although API information is good for characteristic in terms of resilience, many OSS libraries, such as tree, searching and encoding/decoding, rarely call APIs. In regard to call graph, all functions included in OSS may not be linked in a graph. In addition, the complexity of graph similarity measurement is very high. As to instruction information, the size of characteristic is too big and the comparison time is too long. In contrast, since the number of functions is far less than the number of instructions, the size of function-level characteristic is very small and the comparison is also fast. Another reason for using function-level characteristic is that a target program usually includes not a small part of codes but the whole library or a set of functions of OSSs.

Second, the proposed function-level birthmark makes use of intrinsic characteristic of source code that survives in binary code, which is the size of arguments and local variables of functions. In case of instruction-based characteristics, the characteristics of source codes may be lost by compilation. For example, a *for/while* repetition statements may be converted to *do-while* statements by compiler for efficient execution. On the other hand, the size of arguments and local variables of functions is not changed by compilation, which implies it is a reliable characteristic.

Finally, in OSS detection, static analysis is better than dynamic analysis in terms of the time overhead of extracting characteristics and reliability. In dynamic analysis, it is not guaranteed that the OSS codes are executed while the target program runs. For OSS codes to be executed, the inputs of the target program should be given so that the functions of OSS are executed. However, it is very difficult to find out such inputs. Moreover, dynamic analysis is usually conducted on virtual machines with debugging tools to exclude unexpected influence of the execution environments, which may result in unacceptable time overhead. Therefore, dynamic analysis is not suitable for OSS detection.

In the following subsections, we describe the function-level birthmark in detail and propose two methods for OSS detection that uses the birthmark. The first method employs semi-global sequence alignment and the second method employs k-gram.

3.1 Arguments and Local Variables Size based Birthmark

We analyze the instructions in the code section (.text) of binary executable files and extract the size of function arguments and local variables in bytes. In Intel CPU envirionments, *ebp* register is used to reference the function arguments and local variables. The size of arguments can be calculated by analyzing the instructions referencing memory at a higher address than *ebp*[2]. For example, *move eax, [ebp+8]* copies the 4 bytes at memory address *ebp+8* into *eax* register. Usually *ebp+8* is the address of the first argument. To compute the size of arguments we find the instruction that adds the largest offset to *ebp* in a function. Then the size of argument is the largest offset minus 4 because a return address is stored at *ebp+4*. Suppose there are N instructions that reference memory at higher address than *ebp* in a function f and the offsets added to *ebp* are $o_1, o_2, ..., o_N$. Then the size of arguments of f is

$$S_{arg}(f) = \begin{cases} \max\{o_1, o_2, ..., o_N\} - 4 & \text{if } N > 0 \\ 0 & \text{otherwise} \end{cases}$$

When calculating the size of local variables we consider only the actual local variables. We exclude the instructions that use stack to pass arguments for function calls (e.g. *push* instructions). The size of local variables can be calculated by analyzing the instructions referencing memory at a lower address than *ebp*. For example, *move eax, [ebp-18]* copies the 4 bytes at memory address *ebp-18* into *eax* register.

[2]Hereafter, *ebp* means the address contained in *ebp* register.

Unlike when calculating arguments size, the largest offset subtracted from *ebp* in the instructions of a function is the size of local variables because there is nothing between *ebp* and the address of local variables. Suppose there are M instructions that reference memory at lower address than *ebp* in a function f and the offsets subtracted from *ebp* are $o_1, o_2, ..., o_M$. Then the size of local variables of f is

$$S_{var}(f) = \begin{cases} \max\{o_1, o_2, ..., o_M\} & \text{if } M > 0 \\ 0 & \text{otherwise} \end{cases}$$

The characteristic of a function f is defined as follows.

$$Char(f) = (S_{arg}(f), S_{var}(f))$$

Suppose there are F functions, $f_1, f_2, ..., f_F$, in an OSS library or a target program P. The birthmark of P is a sequence:

$$Birthmark(P) = (Char(f_1), Char(f_2), ..., Char(f_F))$$

The problem of this birthmark is that $Char(f)$ is not unique. More than one functions (in a program or in different programs) may have the same arguments size and local variables size. However, $Birthmark(P)$ is distinguishing because $Birthmark(P)$ is a long sequence of $Char(f_i)$ in many OSS libraries.

3.2 Semi-global Sequence Alignment based OSS Detection

To detect OSS in a target program we apply pairwise sequence alignment, which is used in bioinformatics, to comparing birthmarks extracted from OSS library and a target program. The sequence alignment algorithms are classified into three categories: global alignment, local alignment, and semi-global alignment. Global alignments are useful when the sequences are similar and of roughly equal size. In global alignment, the alignment is spanned to the entire length. Local alignments are more useful for dissimilar sequences that are suspected to contain similar sequence. By contrast, Semi-global alignments are useful when one sequence is short and the other is very long. Semi-global alignments attempt to find the best possible alignment that includes the entire short sequence [5, 4].

Global alignment algorithm maintains scoring matrix to find optimal alignment. For two sequences a and b with length n and m respectively, scoring matrix $S(n,m)$ is determined by Equation 1 and the algorithm returns an alignment with the maximum alignment score. And match premium, mismatch penalty, and gap penalty can be adjusted depending on the type of sequences and the purpose of alignment.

$$S(i,j) = \max \begin{cases} S(i-1, j-1) + s(a_i, b_i) \\ S(i-1, j) + \sigma \\ S(i, j-1) + \sigma \end{cases} \quad (1)$$

where $s(a_i, b_i) = \rho = +1$ if $a_i = b_j$ (match premium), $s(a_i, b_i) = \mu = -1$ if $a_i \neq b_j$ (mismatch penalty), and $\sigma = -1$ (gap penalty).

In semi-global alignment, the scoring is similar to the global alignment except that there is no penalty for the prefix and suffix of the short sequence. Hence, let m be the length of the long sequence, the scoring matrix is initialized as follows.

$$S(0,0) = S(0,1) = S(0,2), ..., S(0,m) = 0$$

The method proposed in this paper examines whether the birthmark of OSS (a short sequence of $Char(f_i)$) is similar to a part of the birthmark of a target program (a long sequence of $Char(f_j)$). Thus,

we employ semi-global sequence alignment for birthmark comparison. Semi-global sequence alignment can identify the location of similar sequence as well as the existence of such sequence, which implies we can identify the location of OSS library in a target program.

The similarity between OSS library and target program is the value of sequence identity according to the result of semi-global alignment. Given two aligned sequence T_1 and T_2, the sequence identity is the ratio of $SC(T_1,T_2)$ to $\min\{n,m\}$, where $SC(T_1,T_2)$ is the number of identical elements.

$$Similarity(T_1,T_2) = \frac{SC(T_1,T_2)}{\min\{n,m\}} \tag{2}$$

3.3 K-gram based OSS Detection

Semi-global sequence alignment based method is very effective if OSS library is included in a target program without modifying source codes. However, if developer of the target program inserts or removes some functions, the detection results may be unsatisfactory. In this section, we propose a k-gram based OSS detection method. K-gram is a well-known method for similarity measurement. It generates a set of partial sequence with k contiguous elements from each given sequence, then computes the similarity between the generated sets. Since a sequence is converted to a set, insertion or removal of some functions affect the similarity a little, which implies that k-gram is a resilient method. Figure 1 shows an example of k-gram generation from a function arguments and local variables size based birthmark. If k-grams are extracted from a sequence of length n, the number of elements of the k-gram set is $(n-k+1)$.

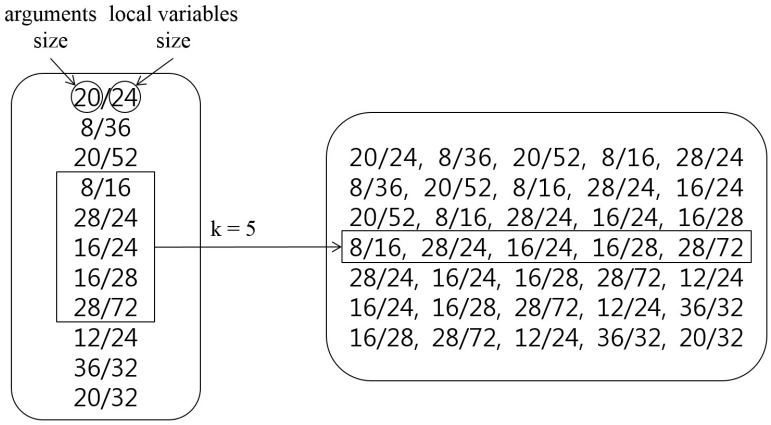

Figure 1: Example of extracting arguments/local variables size based k-gram

Usually k-gram based similarity is calculated using *Jaccard similarity* or *containment*. Suppose A and B are sets of k-grams extracted from two sequences. The similarity is calculated as follow.

$$Jaccard(A,B) = \frac{|A \cap B|}{|A \cup B|} \quad Containment(A,B) = \frac{|A \cap B|}{|A|}$$

Calculating similarity using $Jaccard(A,B)$ or $Containment(A,B)$ is not suitable for OSS detection where the difference between the length of two sequence is very large. Suppose A is the set of k-grams from OSS and B is the set of k-grams from a target program. Since the size of B is far larger than A, $Jarccard(A,B)$ may be very small even if the OSS is included in the target program. As to $Containment(A,B)$, since B is very large it is probable that there exist functions with the same characteristics in both A and B, resulting in false positive.

From this observation, we do not compare the k-grams of OSS with the k-grams of the entire target program. We employ a *sliding window* whose size is equal to the birthmark of OSS, i.e. the number of functions in OSS. Moving the window from the beginning of the birthmark of the target program, we extract a set of k-grams from the *partial birthmark* shown through the window. Let n be the size of OSS's birthmark and m be the size of a target program's birthmark. The number of partial birthmark is $(m-n+1)$. Let P be the k-gram set of OSS and let $Q_i(1 \leq i \leq m-n+1)$ be the k-gram set of the partial birthmark. The similarity between OSS's birthmark (P) and a target program's birthmark (Q) is computed by the following equation.

$$Similatiry(P,Q) = \max_i\{\ PS(P,Q_i)\ \} \quad \text{where} \quad PS(P,Q_i) = \frac{|P \cap Q_i|}{|P|}$$

4 Performance Evaluation

We conducted experiments to compare the proposed methods with existing ones with respect to feature size, detection time, and detection accuracy. The experiments were done on MS Windows 7 64 Bits operating system on a computer with 16GB memory and Intel Core i5-4670 CPU (3.40GHz). Feature extracting module is implemented with IDA pro and IDA Python plug-in and detection module in Python language.

We have selected a program of various sizes as much as possible in order to show the validity of our method. 'Kdtree' and 'AEStable' are OSS libraries that implement tree algorithm and AES encryption algorithm, respectively. 'Kd_test' uses 'Kdtree' and 'AES_test' uses 'AEStable'. The compression program, '7lzma', uses the library 'LzmaDec' and Notepad++ uses xml-related 'Tinyxml' and 'Tinyxmlparser' libraries (see Table 1). All program were built in debug mode in Visual Studio 2008.

Table 1: OSS libraries and executables

Files used	Programming language	File type	File size
Kdtree	C	object file	30 KB
AEStable	C	object file	30 KB
LzmaDec	C	object file	38 KB
Tinyxml	C++	object file	186 KB
Tinyxmlparser	C++	object file	125 KB
Kd_test (uses kdtree)	C	executable	27 KB
AES_test (uses aestable)	C	executable	58 KB
7lzma (uses LzmaDec)	C	executable	510 KB
Notepad++ (uses tinyxml, tinyxmlparser)	C++	executable	3,922 KB

4.1 Feature Size

We measured feature size and the total time to detect OSS code in executables of our method, instruction-based one, and API-based one. Feature size is the length of a sequence of a meaningful unit in each method. In our method, we count the number of functions in an executable, in instruction-based one, the number of all machine instructions, and in API-based one, the number of instructions that call APIs.

Table 2: Comparison of Feature size

Executables	Feature size (length of sequence)		
	Instruction-based	API -based (ref. sequence)	Our method
Kdtree	1,733	68 (3 APIs)	32
AEStable	1,814	69 (9 APIs)	19
LzmaDec	2,591	2 (1 APIs)	18
Tinyxml	4,703	110 (14 APIs)	139
Tinyxmlparser	2,966	64 (9 APIs)	60
Kd_test	2,714	207 (54 APIs)	72
AES_test	6,572	761 (68 APIs)	83
7lzma	96,021	944 (106 APIs)	1,319
Notepad++	772,099	11,866 (450 APIs)	13,297

The smaller number of functions than the number of all instructions implies our method is more efficient for feature comparison and feature DB construction. The number of functions are almost equal to the number of API calls. However, for 'Kdtree', 'AEStable', 'LzmaDec', and 'Tinyxmlparser', the number of APIs is less than 10 and thus API-based one is not appropriate for OSS detection (Table 2).

4.2 Detection Time

We measured the detection time of our methods using two similarity calculation methods, semi-global alignment and k-gram method. The results is shown in Table 3. Our method showed a reasonable overhead except Notepad++ case. The feature size of Notepad++ is the largest among targets and the large overhead is understandable. We also see semi-global scheme is more efficient than k-gram in calculating similarity.

Table 3: Detection Time of the proposed method

Executables		Detection time (*ms*)	
Target executables	OSS libraries	Semi-global	k-gram (k=2)
Kd_test	Kdtree	1.7	8.9
AES_test	AEStable	1.2	6.1
7lzma	LzmaDec	15.8	43.7
Notepad++	Tinyxml	1034.4	6,405.6
	Tinyxmlparser	462.8	1,720.7

4.3 Detection Accuracy

We have experimented on the accuracy of the proposed birthmark based on the size of the function's arguments and local variables. The similarity are identified using two schemes as described above: semi-

global alignment and k-gram.

4.3.1 Using semi-global alignment

Semi-global alignment is a sequence alignment scheme and dependent on 3 factors: match premium (ρ), mismatch penalty (μ), and gap penalty (σ). These factors affect the selection among alignment results and must be adaptively set according to targets and objectives. To find best settings, we have measured ratio of correct identification in detecting OSS libraries used in executables.

- Case 1: Just maximum matches
  (match premium (ρ) = 1, mismatch penalty (μ) = -1, gap penalty (σ) = -1)
  The scheme aligns two sequences to have the maximum matches. If there are alignments with the same number of matches, the scheme selects any of them.
- Case 2: Case 1 + minimum mismatches
  (match premium (ρ) = 1, mismatch penalty (μ) = -1, gap penalty (σ) = 0)
  The scheme aligns two sequences to have the maximum matches. If there are alignments with the same number of matches, the scheme selects one with minimum mismatches without considering gaps.
- Case 3: Case 2 + less gaps
  (match premium (ρ) = 2, mismatch penalty (μ) = -2, gap penalty (σ) = -1)
  The scheme works similar to case 2. However, if there are alignments with the same number of minimum mismatches, the scheme selects the alignment with less gaps.

Table 4: Detection Accuracy when using semi-global alignment scheme

Target executables	OSS libraries	Ratio of Correct identification (%)		
		Case 1 just max. matches	Case 2 Case1 + min. mismatches	Case 3 Case 2 + less gaps
Kd_test	Kdtree	100.0	100.0	100.0
AES_test	AEStable	100.0	100.0	100.0
7lzma	LzmaDec	100.0	100.0	100.0
Notepad++	Tinyxml	77.6	96.4	77.6
	Tinyxmlparser	0	93.3	46.6

In Table 4, The proposed birthmark using semi-global alignment scheme works well for ‘Kd_test’, ‘AES_test’, and ‘7lzma’, which have a smaller file size. But you can see we had some cases where our technique failed to find ‘Tinyxml’ and ‘Tinyxmlparser’ in ‘Notepad++’. The sequence of functions of ‘Tinyxml’ in ‘Notepad++’ remains the same as the original one in the library, but other additional functions exist between the library functions. This fact account for lower detection ratio. In the case of ‘Tinyxmlparser’, the sequence is also changed, so detection ratio further decreases. For ‘Tinyxmlparser’, the results of case 2 settings show ratio of 93.3% and it is because case 2 tries to include as many gaps as possible to maximize the number of matches. Case 2 showed high detection ratio, but actually failed to find the library code. Case 3 showed the detection ratio of 46.6% and is lower than the case 2. Case 3 approximately finds the location of the library code and shows the most correct detection and most resilient to sequence variations.

Table 5: Uniqueness Test Results: c.*n* for case *n*

	Kd_test			AES_test			7lzma			Notepad++		
Libraries	c.1	c.2	c.3	c.1	c.2	c.3	c.1	c.2	c.3	c.1	c.2	c.3
Kdtree	-	-	-	5.5	27.7	5.4	5.5	33.3	5.4	0	86.4	0
AEStable	0	68.4	0	-	-	-	0	26.3	0	0	78.9	10.5
LzmaDec	0	70.2	5.5	5.4	35.1	5.5	-	-	-	0	88.8	0
Tinyxml	not tested			not tested			1.4	52.5	1.4	-	-	-
Tinyxmlparser	3.3	15.0	3.3	1.6	20.0	1.6	1.6	66.6	1.6	-	-	-

To show our birthmark is unique and intrinsic to binary file, we tested whether the scheme can find the library code in executables that does not include the code or not. The length of sequence of 'Tinyxml' is much longer that those of 'Kd_test', 'AES_test' and thus excluded in this experiment. The Table 5 shows the results of this test. Case 1 and 3 showed similar ratio and the largest ratio is 10.5%. Case 2, however, showed high ratio and up to 88.8%. We can see case 2 has high false-positive ratio and is not appropriate for code similarity identification. In sum, case 3 setting is most effective for OSS code detection.

4.3.2 Using k-gram

When k-gram is used to identify OSS, k is a very important factor. When k is low, two different programs share the same features and it is likely to be identified they are similar. The low k has high similarity identification ratio, but is resilient to code transformation like code obfuscation. On the contrary, high k shows higher credibility, but is weak at code transformation.

To find the appropriate value of k for code detection, we calculate the similarities with varying k from 2 to 10 by 2. The results are shown in Table 6. $k = 2$ shows the highest detection ratio and the identified position is close to actual position. Its results are almost identical to those of case 3 when using semi-global alignment.

Table 6: Detection Accuracy according to k when using k-gram

Executables	Libraries	Ratio of Correct identification (%)				
		$k=2$	$k=4$	$k=6$	$k=8$	$k=10$
Kd_test	Kdtree	100.0	100.0	100.0	100.0	100.0
AES_test	AEStable	100.0	100.0	100.0	100.0	100.0
7lzma	LzmaDec	100.0	100.0	100.0	100.0	100.0
Notepad++	Tinyxml	72.7	51.5	37.1	26.1	15.6
	Tinyxmlparser	51.4	16.6	5.6	1.9	0.0

Table 7 shows the uniqueness of the proposed birthmark when using k-gram scheme. When k is two, the scheme works well. In sum, the proposed birthmark using k-gram is effective for detecting OSS codes in target executables.

Table 7: Uniqueness Test Results when using k-gram ($k = 2$)

ratio (%)	Kd_test	AES_test	7lzma	Notepad++
Kdtree	-	7.1	25.0	21.4
AEStable	0	-	6.2	18.0
LzmaDec	7.1	0	-	14.0
Tinyxml	not tested	not tested	15.0	0
Tinyxmlparser	8.5	11.4	14.2	-

5 Conclusion and Future Work

Software released in binary form frequently includes third-party components without the suppliers following the requirements of their software license. Even though such license violations are not intentional, organizations that adapted OSS may be exposed to significant legal risk. To mitigate the risk from such violations, it is necessary to develop a system for code clone detection in binaries. We motivated the need for a system to detect code cloning in binaries including OSS components. We proposed a new technique based on function-level static software birthmark in order to detect cloning in binaries. Our birthmark is the information associated with the size of arguments and local variables of functions in a target binary or OSS components. We could detect binary code clone by measuring the similarity between the birthmarks extracted from a binary and an OSS component. Our technique has shown good performance for detecting code clones especially in C program. However, it has some limitation to detect code clones in C++. The size of our birthmark is small compared to the conventional birthmarks such as instruction-level or API-level ones. As future work, we have a plan to find out more effective and efficient birthmark which can detect precisely code clones in C++ and is resilient to compiler optimization.

6 Acknowledgments

This research was supported by Ministry of Culture, Sports and Tourism(MCST) and from Korea Copyright Commission in 2014 and the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (NIPA-2014-H0301-14-1023) supervised by the NIPA(National IT Industry Promotion Agency).

References

[1] Binary analysis tool. `http://www.binaryanalysis.org/en/home`, last viewed September 2014.

[2] Fossology. `http://www.fossology.org`, last viewed September 2014.

[3] Protex. `http://www.blackducksoftware.com/products/black-duck-suite/protex`, , last viewed September 2014.

[4] Sequence alignment (wikipedia). `http://en.wikipedia.org/wiki/Sequence_alignment`, last viewed September 2014.

[5] M. Brudno, S. Malde, A. Poliakov, C. B. Do, O. Couronne, I. Dubchak, and S. Batzoglou. Glocal alignment: finding rearrangements during alignment. *Journal of Bioinformatics*, 19:i54–i62, 2003.

[6] P. P. Chan, L. C. Hui, and S. Yiu. JSBiRTH: Dynamic JavaScript Birthmark Based on the Run-Time Heap. In *Proceedings of the 35th IEEE Annual Computer Software and Applications Conference (COMPSAC'11), Munich, Germany*, pages 407–412. IEEE, July 2011.

[7] S. Choi, H. Park, H. Lim, and T. Han. A Static Birthmark of Binary Executables Based on API Call Structure. In *Proc. of the 12th Advances in Computer Science Conference: computer and network security (ASIAN'07), Doha, Qatar*, pages 2–16. ACM, December 2007.

[8] Gartner. Predicts 2011: Open-Source Software, the Power Behind the Throne, November 2010.

[9] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra. Finding Software LIcense Violations Through Binary Code Cone Detection. In *Proc. of the 8th Working Conference on Mining Software Repositories (MSR'11), Hawaii, USA*, pages 63–72. ACM, May 2011.

[10] P. Marshall. Webinar: Open source component governance and management using cobit 5, June 2012. `http://www.isaca.org/Education/Online-Learning/Pages/webinars.aspx`.

[11] G. Myles and C. Collberg. k-gram Based Software Birthmarks. In *Proc. of the 2005 ACM Symposium on Applied Computing (SAC'05), New Mexico, USA*, pages 314–318. ACM, March 2005.

[12] G. Myles, C. C. G. Myles, and C. Collberg. Detecting Software Theft via Whole Program Path Birthmarks. In *Proc. of the 7th International Conference (ISC'04), Palo Alto, CA, USA, LNCS*, volume 3225, pages 404–415. Springer-Verlag, September 2004.

[13] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proc. of the 18th international symposium on Software testing and analysis (ISSTA'09), Chicago, IL, USA*, pages 117–128. ACM, July 2009.

[14] D. Schuler and V. Dallmeier. Detecting Software Theft with API Call Sequence Sets. In *Proc. of the 8th Workshop Software Reengineering (WSR'06), Bad Honnef, Germany*, May 2006.

[15] H. Tamada, K. Okamoto, M. Nakamura, and A. Monden. Dynamic Software Birthmarks to Detect the Theft of Windows Applications. In *Proc. of the 8th International Symposium on Future Software Technology 2004 (ISFST'04), Xian, China*, October 2004.

[16] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Detecting Software Theft via System Call Based Birthmarks. In *Proc. of the 25th Annual Computer Security Applications Conference (ACSAC'09), Hawaii, USA*, pages 149–158. IEEE, December 2009.

[17] X. Zhou, X. Sun, G. Sun, and Y. Yang. A Combined Static and Dynamic Software Birthmark Based on Component Dependence Graph. In *Proc. of the 4th International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP'08), Harbin, China*, pages 1416–1421. IEEE, August 2008.

Author Biography

Dongjin Kim received the B.E. and M.E. degree in Computer Science from Dankook University in 2009 and 2011, respectively. He is now a Ph.D candidate student in Dankook University, Korea. His current research interests include computer security, software protection, smartphone security and system software.

Seong-je Cho received the B.E., the M.E. and the Ph.D. in Computer Engineering from Seoul National University in 1989, 1991 and 1996 respectively. He was a visiting scholar at Department of EECS, University of California, Irvine, USA in 2001, and at Department of Electrical and Computer Engineering, University of Cincinnati, USA in 2009 respectively. He is a Professor in Department of Computer Science and Software Science, Dankook University, Korea from 1997. His current research interests include computer security, operating systems, software protection, real-time scheduling, and embedded software.

Sangchul Han received his B.S. degree in Computer Science from Yonsei University in 1998 and his M.E. and Ph.D. degrees in Computer Engineering from Seoul National University in 2000 and 2007, respectively. He is now an associate professor of Dept. of Computer Engineering at Konkuk University. His research interests include real-time scheduling and computer security.

Minkyu Park received the B.E. and M.E. degree in Computer Engineering from Seoul National University in 1991 and 1993, respectively. He received Ph.D. degree in Computer Engineering from Seoul National University in 2005. He is now an Associate Professor in Konkuk University, Korea. His research interests include operating systems, real-time scheduling, embedded software, computer system security, and HCI.

Ilsun You received his MS and PhD degrees in Computer Science from Dankook University, Seoul, Korea in 1997 and 2002, respectively. Also, he obtained his second PhD degree from Kyushu University, Japan in 2012. In 2005, he joined Korean Bible University, South Korea as a full time lecturer, and he is now working as an associate professor. Dr. You has published more than 110 papers as well as edited more than 20 special issues with focus on the topics including internet security, mobility management, cloud computing, pervasive computing, and so forth. Dr. You is IET Fellow and IEEE Senior Member.