The Journal of Systems and Software 82 (2009) 862–873

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A static API birthmark for Windows binary executables

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

Division of Computer Science, Korea Advanced Institute of Science and Technology, 335 Gwahangno, Yuseong-gu, Daejeon 305-701, Republic of Korea

ARTICLE INFO

Article history:
Received 21 January 2008
Received in revised form 8 August 2008
Accepted 27 November 2008
Available online 10 December 2008

Keywords:
Software birthmark
Software theft detection
Software security
Static analysis
Reverse engineering

ABSTRACT

A software birthmark is the inherent characteristics of a program extracted from the program itself. By comparing birthmarks, we can detect whether a program is a copy of another program or not. We propose a static API birthmark for Windows executables that utilizes sets of API calls identified by a disassembler statically. By comparing 49 Windows executables, we show that our birthmark can distinguish similar programs and detect copies. By comparing binaries generated by various compilers, we also demonstrate that our birthmark is resilient. We compare our birthmark with a previous Windows dynamic birthmark to show that it is more appropriate for GUI applications.

1. Introduction

Recently, a large amount of software has been developed in the form of open source projects. Open source software allows users to use, change, and distribute the software under certain types of licenses. The most widely used open source software license is the GNU Public License (GPL). [1] The GPL allows developers to use software freely, but requires new projects using the original work to be licensed under the GPL. There are more permissive licenses such as the MIT license and the BSD licenses which allow combining of the original codes in commercial software. These permissive licenses, however, require the copyright notice of the original software to be included. Many companies violate software licenses by incorporating open source software in their commercial products. Such code theft causes substantial damage to open source software companies.

If source code is available, source code plagiarism detectors can be used to detect code theft. For example, YAP,[2] JPlag,[3] and MOSS[4] have been used to detect copies in university programming assignments (Wise et al., 1996; Prechelt et al., 2002; Schleimer et al., 2003; Gitchell and Tran, 1999). They compare programs using symbols, program styles, and program structures.

In many cases, source code is not available, because the original source code is compiled into binary executables. Software birthmarking is a technique to handle such code theft problems (Tamada et al., 2004a). A software birthmark is a collection of inherent characteristics of a program that can be used to identify the program. A software birthmarking system should be able to extract birthmarks from programs and to compute the similarity between two programs. Fig. 1 illustrates an example of code theft detection using a software birthmarking system. To compare program p and program q, $birthmark(p)$ and $birthmark(q)$ are extracted from executables. A similarity score is computed by comparing those two birthmarks. By the similarity score, we can determine whether q is a stolen copy of p.

Earlier birthmarks were focused on Java programs (Tamada et al., 2005; Myles and Collberg, 2004, 2005; Schuler et al., 2007). Extracting birthmarks from Windows programs is much harder than extracting birthmarks from Java programs, since Windows native codes do not have useful information such as method names, class names or types which can be found in Java bytecodes. There is one dynamic API birthmark for Windows programs (Tamada et al., 2004b). However, the dynamic birthmark of Tamada et al. cannot compare interactive programs. Since dynamic birthmarks require the same input, a dynamic birthmarking technique can only be applied to small modules or batch programs. Interactive programs such as GUI or network programs should be compared with static birthmarks.

In this paper, we propose a static birthmark for Windows programs. Our birthmark is comprised of possible Windows API calls per each function. We utilize the IDA Pro disassembler[5] to identify functions and API calls from binary executable programs. The birthmarks are used to calculate function similarities.

* Corresponding author. Tel.: +82 42 869 5560; fax: +82 42 869 5573.
E-mail addresses: swchoi@pllab.kaist.ac.kr (S. Choi), hwpark@pllab.kaist.ac.kr (H. Park), hilim@pllab.kaist.ac.kr (H.-i. Lim), han@cs.kaist.ac.kr (T. Han).

[1] http://www.gnu.org/licenses/gpl.html.
[2] http://luggage.bcs.uwa.edu.au/~michaelw//YAP.html.
[3] http://www.jplag.de./
[4] http://theory.stanford.edu/~aiken/moss./

[5] http://www.datarescue.com/idabase.

0164-1212/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2008.11.848

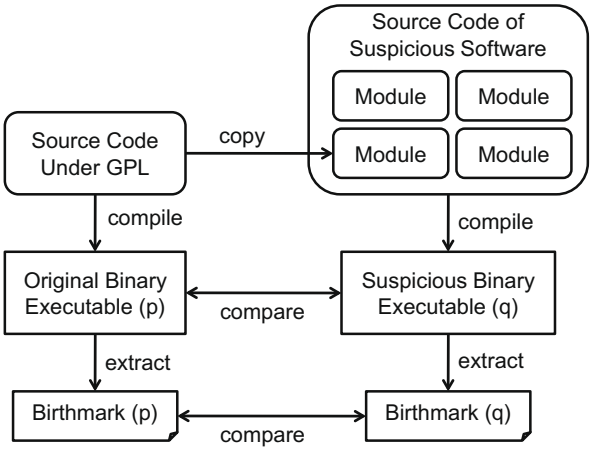


Fig. 1. Detecting binary code theft via birthmarks.

To compare two programs, we model the problem of calculating the similarity between two programs as finding a maximum weighted bipartite matching. In the bipartite graphs, programs correspond to partite sets and functions correspond to vertices. The weights assigned in edges represent the similarities between two functions from each program. We use the shortest augmenting path algorithm of Jonker and Volgenant to find a maximum weight bipartite matching (Jonker and Volgenant, 1987).

The contribution of our work is as follows:

- We suggest a static API birthmark for Windows programs based on API calls and program structure.
- By applying our birthmarking system to real-world Windows programs, we show that our birthmark can help to detect code theft.
- By comparing our birthmark with the previous Windows dynamic birthmark of Tamada et al. (2004b), we demonstrate that our birthmark is more appropriate for Windows applications.

The remainder of this paper is organized as follows. In Section 2, we describe our birthmark in detail. In Section 3, we evaluate our birthmark by comparing Windows applications with experimental results. In Section 4, we compare our birthmark with the previous Windows dynamic birthmark of Tamada et al. through experiments. In Section 5, we describe possible attacks on our birthmark. In Section 6, we review related work. We conclude in Section 7 with a summary and directions for future work.

2. Static API birthmark

2.1. Definition of static API birthmark

Operating systems provide system services such as memory management, accessing graphics, and networking via system calls or OS API calls. Since most application programs frequently use system calls, which are difficult replace, system calls are considered important characteristics to identify a program. For example, system calls have been used to detect malicious codes (Bergeron et al., 1999, 2001; Sung et al., 2004; Sekar et al., 2001; Hofmeyr, 1998; Warrender et al., 1999; Rajagopalan et al., 2005), and dynamic Windows API birthmarks of Tamada et al. (Tamada et al., 2004b; Okamoto et al., 2006) utilized dynamic API call sequences to detect code theft.

We suggest a static API birthmark for Windows application programs based on the Windows API call sets per each function. The Windows API calls are important characteristics of a Windows application program, since all Windows programs except console programs and kernel mode programs use the Windows API. We selected 1946 Windows API functions from the MSDN website[6] to extract birthmarks. The proposed birthmarking technique can analyze whole sections of given programs, while the dynamic birthmarking technique of Tamada et al. can analyze parts of the given programs.

The structures of function birthmarks can be sets, sequences, or graphs. Sequences of method calls have been used for the Java birthmarks of Tamada et al. (2005). The sequences are different from the real execution order of the methods, because they depend on the order of the program code. We can estimate API call sequences using flow graphs, but the number of sequences grows exponentially proportional to the number of branch instructions. Furthermore, the machine code contains indirect branch instructions, which are very hard to resolve. n-gram based methods can reduce the number of sequences such that a similarity calculation becomes possible. An n-gram is a sub-sequence of n items from a given sequence. Myles and Collberg (2005) used Java bytecode sequences of length k in their k-gram birthmarks. Kwon (2008) used n-gram sequences of Windows API calls as a birthmark and compared the sequence birthmark to our set birthmark. If a sequence is summarized to a set, order information is lost such that comparison of two sets becomes less precise. However, the result from the n-gram Windows API birthmark is very close to that of our set birthmark. This shows that sets are precise enough to be used as a birthmark. Birthmarks can take a graph form; for example, control flow graphs annotated with API calls can be a birthmark. Graph representations reflect the structure of original programs. But comparing two graphs is more complex than comparing two sets or sequences. In general, the worst case time complexity to compare two graphs is NP. For these reasons, we utilize sets of API calls as our birthmark.

The birthmark of a function is computed using a call graph of a program. The birthmark is defined as a set of API calls of the function and its descendant functions, which correspond to the descendant nodes of the function in the call graph.

Definition 1. Function birthmark Let k be an integer (with $k \geqslant 0$). Given a program and its call graph, where each node represents a function and each edge represents a function call, the birthmark $B_f(f,k)$ of function f is defined as a set of all API calls in f and its descendant functions having call depths within k, which is computed starting from f.

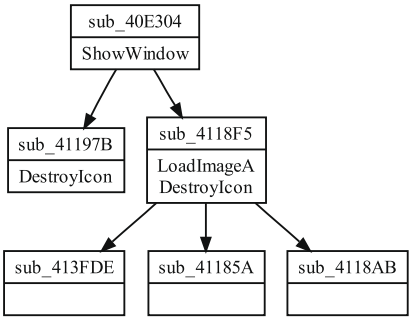
Note that the birthmark not only contains API calls of a function itself but also API calls of the descendant functions in the call graph. In other words, the characteristics of a function are the combination of the function itself and its descendants. This approach is taken because we wish to approximate the runtime behaviors of the function.

In Fig. 2, $B_f(\texttt{sub\_40E304}, 0)$ is {`ShowWindow`}. As k increases, $B_f(f,k)$ includes API calls in the descendants of f from the call graph. $B_f(\texttt{sub\_40E304}, 1)$ is {`ShowWindow`, `DestroyIcon`, `LoadImageA`}. If k becomes equal to or larger than the maximum call depth from the corresponding function, the k-depth API call set contains all possible API calls when the function is executed.

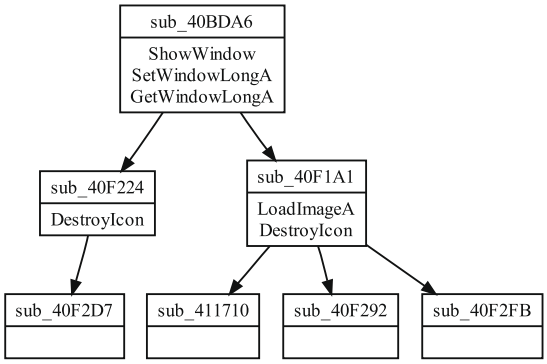
Given the call depth k and the number of functions n, the worst case time complexity for finding all function birthmarks is $O(kn^2)$.

The accuracy of a function birthmark depends on the precision of the call graph. To construct a static call graph, a pointer alias analysis is carried out to identify all possible call targets (points-to set) conservatively. Although a precise pointer alias analysis is a non-decidable problem, type information makes it possible to estimate points-to set within reasonable time for typed programming languages such as Java and C/C++ (Dean et al., 1995, 1996).

[6] http://msdn2.microsoft.com/en-us/library/aa383749(VS.85).aspx.



(a) A subgraph of the call graph of `Winamp 5.35`, whose root is the function `sub_40E304`



(b) A subgraph of the call graph of `Winamp 5.23`, whose root is the function `sub_40BDA6`

Fig. 2. Call graphs of matched functions from two versions of `Winamp`.

However, because binary executables do not have type information and contain indirect calls and arithmetic instructions on addresses, pointer alias analysis for binary executables is unsound, which means the analysis result may contain unresolved indirect calls (Balakrishnan, 2007).

If a call graph is approximated conservatively (over-approximation), an unresolved call can invoke every function in the program. The function birthmark containing the unresolved call then becomes a set of all API calls in the program. As a result, all function birthmarks that have unresolved calls become the same so that unrelated functions are considered as copies just because the functions have unresolved calls. In other words, unresolved calls with over-approximation can cause false-positives. Keeping such a large birthmark also requires heavy computational cost.

If unresolved calls are ignored (under-approximation), the call graph does not contain possible edges that point to functions. Hence, a function birthmark with $k \geqslant 1$ becomes a subset of the function birthmark that is computed with the precise call graph assuming that a precise pointer alias analysis has been performed. The difference between an under-approximated birthmark, which ignores unresolved calls, and the precise birthmark depends on the characteristics of the program such as call structures and unresolved calls. Because of under-approximation, originally related functions with unresolved calls can be considered as unrelated programs. In other words, unresolved calls with under-approximation can cause false-negatives.

We conclude that the cost of over-approximation of a call graph surpasses the cost of under-approximation because over-approximation causes heavy computational cost, and under-approximation is precise when k is 0. Thus, we omit unresolved calls in computing function birthmarks.

The program birthmark is defined using function birthmarks.

Definition 2. Program birthmark Let p be a program, P be a set of functions in p, and k be an integer (with $k \geqslant 0$). The program birthmark $B_p(p,k)$ is defined as

$$B_p(p,k) = \{(label(f), B_f(f,k)) | f \in P\}$$

where $label(f)$ represents the function name of f.

By function labels, function birthmarks that have the same set of API calls are dealt with separately.

2.2. Calculating similarity

We model the problem of calculating the similarity between two programs as finding a maximum weighted bipartite matching. A maximum weighted bipartite matching is a subset of edges, where each vertex belongs to exactly one edge and the sum of the values of the edges in the matching have a maximal value.

Fig. 3 shows an example of comparing two programs p_1 and p_2 using maximum weighted bipartite matching. p_1 has functions $f_1, \cdots, f_m$ and p_2 has functions $g_1, \cdots, g_n$, where m, n are the number of functions. In this bipartite graph, programs correspond to partite sets, functions correspond to vertices, and the weights assigned in edges represent the similarities between two functions from each program. If the set of edges given in Fig. 3 maximize the sum of weights, the maximum weighted bipartite matching is $\{(f_1,g_2), (f_2,g_3), \cdots, (f_m,g_n)\}$.

To find a maximum weighted bipartite matching, we need a complete bipartite graph $K_{m,n}$, where every vertex of p_1 is connected to every vertex of the p_2. Hence, we first compute all possible function similarities. A similarity between two functions is defined as follows.

Definition 3. [Function similarity] Given functions f_1, f_2 and their birthmarks $F_1 = B_f(f_1,k)$, $F_2 = B_f(f_2,k)$ with an integer (with $k \geqslant 0$), the function similarity $sim_f(f_1,f_2)$ is defined as

$$sim_f(f_1,f_2) = \frac{2|F_1 \cap F_2|}{|F_1| + |F_2|}$$

where $|F_1|$ and $|F_2|$ are the number of API calls in F_1 and F_2 respectively, and $|F_1 \cap F_2|$ is the number of common API calls in F_1 and F_2.

In Fig. 2, given $k = 1$, function birthmarks of `sub_40E304` of `Winamp 5.35` and `sub_40BDA6` of `Winamp 5.23` are

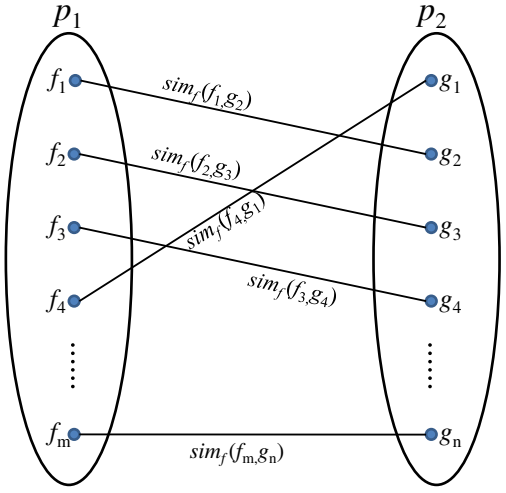


Fig. 3. An example of program comparison modeled as bipartite matching.

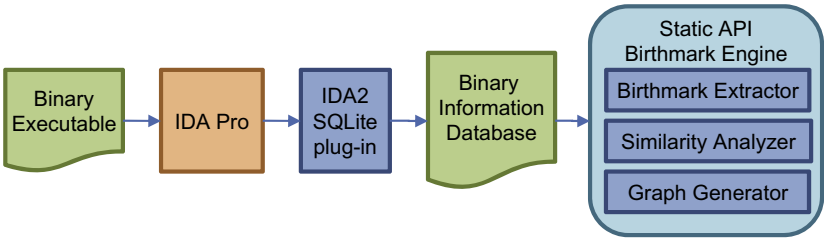


Fig. 4. Overview of static API birthmark system.

$$B_f(\texttt{sub\_40E304}, 1) = \{\texttt{ShowWindow}, \texttt{DestroyIcon}, \texttt{LoadImageA}\}$$
$$B_f(\texttt{sub\_40BDA6}, 1) = \{\texttt{ShowWindow}, \texttt{SetWindowLongA}, \texttt{GetWindowLongA}, \texttt{DestroyIcon}, \texttt{LoadImageA}\}$$

and the intersection of the two birthmarks is

$$B_f(\texttt{sub\_40E304}, 1) \cap B_f(\texttt{sub\_40BDA6}, 1) = \{\texttt{ShowWindow}, \texttt{DestroyIcon}, \texttt{LoadImageA}\}.$$

The function similarity $sim_f(\texttt{sub\_40E304}, \texttt{sub\_40BDA6})$ with $k = 1$ is then computed according to Definition 3 as

$$sim_f(\texttt{sub\_40E304}, \texttt{sub\_40BDA6}) = \frac{2|B_f(\texttt{sub\_40E304}, 1) \cap B_f(\texttt{sub\_40BDA6}, 1)|}{|B_f(\texttt{sub\_40E304}, 1)| + |B_f(\texttt{sub\_40BDA6}, 1)|} = \frac{2 \cdot 3}{3+5} = 6/8 = 0.75.$$

This similarity measure is known as the Dice coefficient or Sorensen index, and is used for information retrieval. The function similarity measures the fraction of common calls over all API calls. If two functions are identical, the similarity between the functions becomes 1. If the two functions have no API calls in common, the similarity value becomes 0.

There exist $m \times n$ function similarities between two programs p_1 with functions $f_1, \ldots, f_m$ and p_2 with functions $g_1, \ldots, g_n$. After the function similarity calculation, we can obtain a complete bipartite graph $K_{m,n} = \{P_1, P_2; E\}$ where $P_1 = \{f_1, \ldots, f_m\}$, $P_2 = \{g_1, \ldots, g_n\}$, $E = \{(f_i, g_j) | f_i \in P_1 \wedge g_j \in P_2\}$ and a weight $sim_f(f_i, g_j)$ is assigned to each edge in E. Because weighted bipartite matching algorithms require that the lengths of two partite sets be the same, we insert dummy nodes and zero weight edges that are incident to the dummy nodes to obtain $K_{n,n}$ assuming that $n \geqslant m$. Inserting dummy nodes and zero weight edges does not affect matching results. The maximum weighted bipartite matching between two programs p_1 and p_2 is then defined using $K_{n,n}$ as

$$match(p_1, p_2) = \{(u_1, v_1), \ldots, (u_n, v_n)\}$$

where $u_i \neq u_j$ if $i \neq j$, $v_j \neq v_j$ if $i \neq j$, and $\sum_{i=1}^{n} sim_f(u_i, v_i)$ has a maximum value among all possible matchings.

Using the maximum weight bipartite matching, we define the program similarity as follows.

Definition 4. Program similarity Given programs p_1, p_2, the program similarity $sim_p(p_1, p_2)$ is defined as

$$sim_p(p_1, p_2) = \frac{2 \times (\sum)_{(f_1, f_2) \in match(p_1, p_2)} sim_f(f_1, f_2)}{m + n}$$

where m, n are the numbers of functions in p_1 and p_2, respectively, and $match(p_1, p_2)$ is a maximum weighted bipartite matching between p_1 and p_2.

We use the shortest augmenting path algorithm for dense graphs of Jonker and Volgenant (DJV) to find a maximum weight bipartite matching (Jonker and Volgenant, 1987). The worst case time complexity of the DJV algorithm is $O(n^3)$, where n is the number of functions. We utilized a C++ implementation[7] written by the authors. In terms of theory, there are better algorithms to solve the maximum weight bipartite matching with the time complexity of $O(\sqrt{n}m \log(nC))$, where n is the number of nodes, m is the number of edges, and C is the range size of integer scaled weight (Gabow and Tarjan, 1989; Goldberg and Kennedy, 1995). However, we observed that the DJV implementation is faster than the best time complexity algorithms. This is because the DJV implementation use less memory, and the constant C in the time complexity is close to the number of functions n.

2.3. Judgment of copy

The primary aim of a birthmark is to detect copies of a program. The program similarity $sim_p(p_1, p_2)$ is between 0 and 1. With the value, we determine whether two programs are classified as copies. Two programs p_1 and p_2 are classified according to the program similarity as:

$$sim_p(p_1, p_2) = \begin{cases} \geqslant 1 - \epsilon & p_1, p_2 \text{ are classified as copies} \\ \leqslant \epsilon & p_1, p_2 \text{ are classified as independent} \\ \text{otherwise} & \text{inconclusive} \end{cases}$$

We determined the threshold as $\epsilon = 0.35$. From the threshold, a similarity range of [0.0, 0.35] is classified as independent, (0.35, 0.65) as inconclusive, and [0.65, 1] as copies.

Schuler et al. also employed such a classification scheme with a threshold of 0.2, such that the similarity range [0, 0.2] is classified as independent, (0, 2, 0.8) as inconclusive, and [0.8, 1] as copies. Myles (2006) classified programs with a slightly different scheme, where the similarity range $[0, 1 - 2\epsilon)$ is classified as independent, $[1 - 2\epsilon, 1 - \epsilon)$ as inconclusive, and $[1 - \epsilon, 1]$ as copies with $\epsilon = 0.2$, such that the similarity range [0, 0.4) is classified as independent, [0.4, 0.8] as inconclusive, and (0.8, 1] as copies. Schuler et al. and Myles do not explain how their thresholds are determined. Our threshold of $\epsilon = 0.35$ is larger than that of Schuler et al's. The threshold is determined by considering the characteristics of Windows applications. The characteristics include the sets of API calls that are very frequently found in Windows applications (e.g. API calls related to threads and locks), and the ratio of unresolved function calls. However, the size of our threshold does not indicate that our birthmark is inaccurate, because our threshold is determined by larger programs than in the case of Schuler et al.

2.4. Static API birthmarking system

To evaluate the proposed birthmark, we implemented our birthmark system on Microsoft Windows XP. Fig. 4 presents an overview of the static API birthmark system. The Windows binary executable is first disassembled using a commercial disassembler IDA Pro. IDA Pro also identifies functions, and discovers library

[7] http://www.magiclogic.com/assignment.html.

Table 1
Benchmark programs.

Category	Name	Ver.	Size (kB)	Num. Fn.	Ver.	Size (kB)	Num. Fn.
Text	UltraEdit	7.0	1112	2625	7.2	1172	2709
Editor	EditPlus	2.0	1128	2874	2.1	1179	3037
FTP	FileZilla	2.2.14	1592	2190	2.2.26	1696	2303
Client	CuteFTP	3.5.4	1408	3143	4.0.19	1632	3514
Terminal	Putty	0.56	372	823	0.58	412	889
	SecureCRT	5.5.0	1890	4089	5.5.1	1890	4858
P2P	Donkeyhote	2.40	5304	6511	2.54	5296	6502
Client	eMule	0.45b	4276	8924	0.47c	4884	9555
Image	ACDSee	4.01	2512	4248	4.02	2536	4300
Viewer	XnView	1.21	1348	1894	1.25a	1464	2036
Audio	Winamp	5.23	1050	1447	5.35	1111	1730
Player	foobar2000	0.9.1	865	3843	0.9.4	944	4326
Video	GOM Player	2.0.0	2614	5758	2.1.6	2494	5959
Player	Adrenalin	2.1	2888	3022	2.2	3248	2676
CD	CDRWin	3.8	948	1972	3.9	1116	2523
Burner	DVDCopy	2.2.6	2180	1278	2.5.1	2192	1285
Download	Flashget	1.6.5	1448	3822	1.7.2	1336	3237
Manager	NetTransport	2.3.0	1224	2120	2.4.1	1360	2474
Disk	Daemon Tools	4.3.0	130	179	4.9.0	162	198
Emulator	CDSpace	4.1	1868	1239	5.0	1784	850

function calls using FLIRT algorithm (Guilfanov, 1997). IDA Pro supports plug-ins to access internal databases. We developed IDA2SQLite plug-in to store the initial analysis results by IDA Pro in SQLite[8] database such that the binary information database can be used by various program analyzers. The static API birthmark engine extracts birthmarks, calculates similarities, and generates graphs. The similarities are given as text files with function match information. The graph generator provides call graphs and control flow graphs for a more detailed comparison. The graphs are represented by DOT language (Koutsofios and North, 1993) and translated into SVG (Ferraiolo et al., 2003) format by GraphViz package (Ellson et al., 2001). The resulting SVG file can be displayed using SVG Viewers.

3. Evaluation

We evaluated our birthmark according to the following two properties:

Property 1. Credibility Let p and q be independently implemented programs with the same functionality. A birthmark B_p is credible if $sim_p(p,q) \leqslant \epsilon$.

Property 2. Resilience Let p be a program and p' be a program obtained from p by applying program transformation T to p. A birthmark B_p is resilient to T if $sim_p(p,p') \geqslant 1-\epsilon$.

The properties are restatements of Tamada et al. (2005). To be credible, the birthmark should indicate high similarity for the same or very close programs and low similarity for independently written programs. To be resilient, the birthmark should have consistent similarities after program transformations.

3.1. Benchmark programs

To evaluate the effectiveness of our static API birthmark, we performed two experiments. The first experiment evaluates the credibility of the proposed birthmark. The second experiment measures the resilience of the birthmark against different compilers.

To evaluate the credibility, we chose twenty programs from various categories such as text editors, FTP clients, Terminals, etc. as listed in Table 1. Two programs were selected per category, and two versions were selected per program.

Table 2 shows the number of functions according to the number of API calls per function for each sample program. Column 1 is the sample program name with its version. The columns from 2 to 10 indicate the numbers of functions by the number of API calls. The column names 0, 1, 2, and 3 are given according to the numbers of API calls in each function. For example, the entries in column 3 are the numbers of functions with 2 API calls for each program. The column named as $\geqslant 4$ is the number of functions that have six or more API calls. The column named as $\geqslant 1$ is the number of functions that have at least one API call. Because our birthmark incorporates API function calls from descendant functions, each function can include API calls used by its descendants. Therefore, as k grows, more functions can obtain API birthmarks. There are opportunities for the functions with no API calls to include API calls from their descendants as k grows. The column name $\geqslant 1(k=3)$ represents the number of functions of which the function birthmark is not an empty set with $k=3$. The average ratio of functions with non-empty birthmarks to the total number of functions over sample programs is about 0.41. Our program similarities are calculated based on functions of which the function birthmark is not an empty set.

Fig. 5 shows the program similarities according to the call depths k. As the call depth increases, the similarity tends to decrease. For call depth larger than 3, in spite of call depth increment, the similarity decrement is negligible. Hereafter, our experiments are evaluated with $k=3$.

To evaluate resilience, we chose open source hex editor *frhed*,[9] and used *Microsoft Visual C++ 6.0*, *Visual C++ 7.1*, and *Visual C++ 8.0* compilers. In this paper, we do not use code obfuscators because there is no available C/C++ code obfuscator that can affect complied binary executables besides the commercial obfuscator *CloakWare Security Suite*[10].

3.2. Credibility

3.2.1. Similarity of different versions

To evaluate the credibility of our birthmark, we compared the same programs with their different versions. Fig. 6 shows that

[8] http://www.sqlite.org./

[9] http://www.kibria.de/frhed.html.

[10] http://www.cloakware.com/products_services/security_suite

Table 2
Number of functions according to the number of API calls per function for each sample program.

Sample program	Number of functions							Total number of functions
	0	1	2	3	$\geqslant 4$	$\geqslant 1$	$\geqslant 1(k=3)$	
ACDSee 4.02	2731	569	342	183	475	1569	2197	4300
Adrenalin 2.2	2011	391	143	49	82	665	915	2676
CDRWin 3.9	2147	244	64	27	41	376	1455	2523
CuteFTP 4.0.19	2699	425	156	98	136	815	1554	3514
Donkeyhote 2.54	5836	444	112	49	61	666	1522	6502
DVD Copy 2.5.1	1078	79	62	34	32	207	386	1285
Editplus 2.1	2043	487	200	106	201	994	1444	3037
eMule 0.48a	6777	447	105	55	66	673	1625	7450
FileZilla 2.2.26	1850	261	85	47	60	453	681	2303
FlashGet 1.72	2455	445	152	69	116	782	1694	3237
foobar2000 0.9.4.3	3677	326	187	54	82	649	1095	4326
Gom Player 2.1.6	4703	629	284	149	194	1256	3110	5959
NET Transport 2.41	2030	242	74	68	60	444	627	2474
PuTTY 0.58	735	80	22	18	34	154	401	888
SecureCRT 5.5.1	4429	254	85	38	52	429	1232	4858
UltraEdit 7.2	1780	435	170	117	207	929	1444	2708
Winamp 5.35	992	238	217	65	218	738	1442	1730
XnView 1.25a	1595	150	86	96	109	441	765	2036

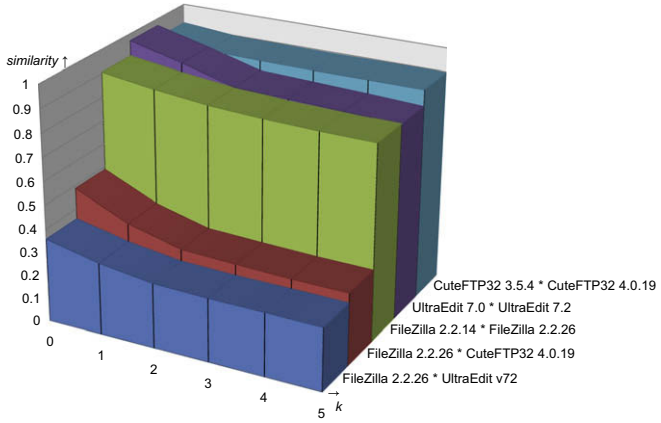


Fig. 5. Program similarities according to call depth k.

the similarities between different versions are greater than 0.75. In general, a minor upgraded version of software shares most of the code from the previous version. In other words, the new version inherited most of characteristics from the previous version. The result shows that our birthmark sufficiently reflects the common program functionalities.

3.2.2. Similarity in the same category

We compared programs in the same categories to prove the credibility. Even if two programs are in the same category, the similarity is not always high, since the programs have different numbers of functions and API calls. Suppose that there are different programs with the same functionalities. They may share many common API function calls, because they have the same functionalities. However, if the two programs are implemented independently, it is highly probable that the structures of functions will be different from each other, and therefore, the function birthmark will be different. Fig. 7 shows that EditPlus and UltraEdit are classified into inconclusive, Donkeyhote and eMule are derivative, and the others are independent. Our birthmark failed to distinguish EditPlus and UltraEdit well. When we investigated the assembly codes of EditPlus and UltraEdit, we found that they have numerous functions that have one or two common API calls in both programs. In such a case, the birthmark is confused by the functions that have a few API calls.

It is remarkable that the similarity between two P2P programs, Donkeyhote and eMule, is more than 0.85. In fact, Donkeyhote is an extended version of eMule which is under the GPL. Donkeyhote modified interfaces and added a few features such as filtering copyrighted files.

3.2.3. Similarity among different categories

We compared programs in different categories to show whether the similarities between totally different programs are sufficiently small. Fig. 8 shows that similarities between programs in different categories are lower than the threshold $\epsilon = 0.35$. The programs belonging to different categories have considerably lower similarities. The similarities near the threshold are due to the common features in Windows programs such as GUI, file management, network, etc.

3.3. Resilience

To evaluate the resilience of our birthmark, we compiled an open source free hex editor, *frhed* with *Microsoft Visual C++ 6.0*, *.NET 2003*, and *.NET 2005* compilers. Table 3 shows the features of *frhed* binaries generated by the three compilers with two build options, debug and release. The binary size and the number of functions without API calls vary according to the compilers and build options but the number of functions with API calls is nearly equal.

Table 4 shows the similarities of the resilience experiment using various compilers and options. All the similarities are higher than 0.95. We conclude that our birthmark is resilient to different compilers. Although we did not test our birthmark with binary code obfuscators, our birthmark is resilient to known code obfuscation techniques. In general, code obfuscators utilize well known code obfuscation techniques such as data transformation, control transformation, and anti-debug (Collberg et al., 1998, 2000, 2006). Data and control transformations change the control flow and hide secret data, but they will not affect similarities, because they do not remove the API calls. Our birthmark appears to be resilient to obfuscation techniques that are applied within function boundaries, as we only examine the existence of API calls for each function. While they are not frequently used because of execution cost, obfuscation methods that alter function structure such as splitting or merging of functions can confuse our birthmarking system.

Obfuscation techniques for binary executables can also confuse our birthmarking system. An obfuscation technique of Linn and

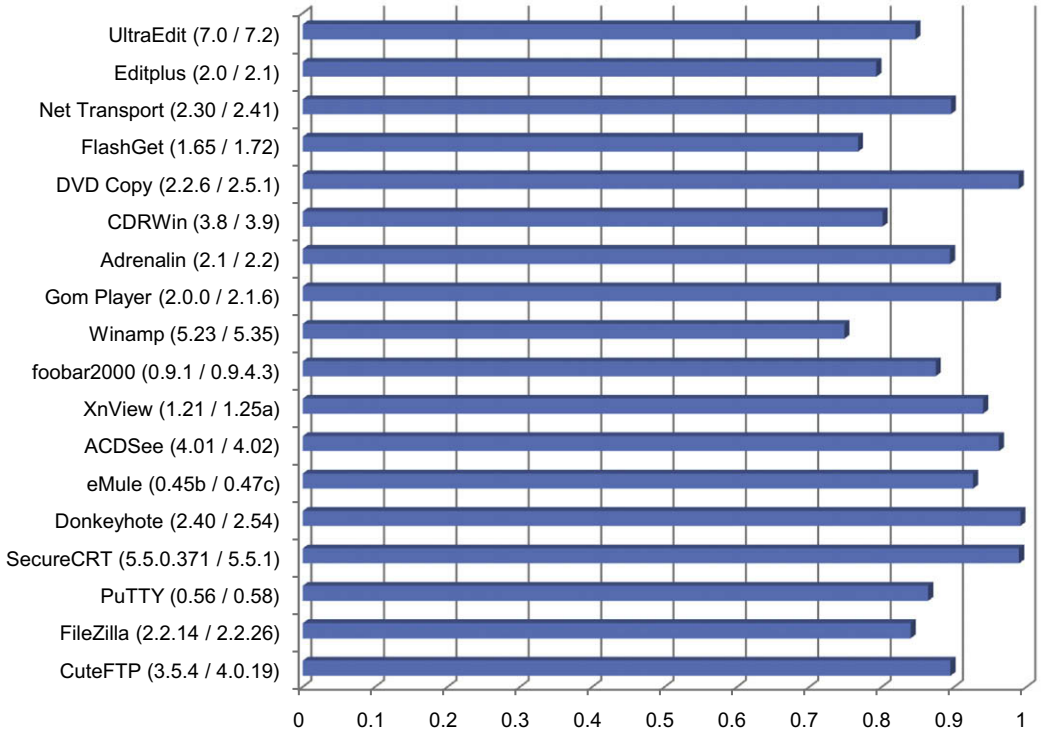


Fig. 6. Similarities between the same programs with different versions.

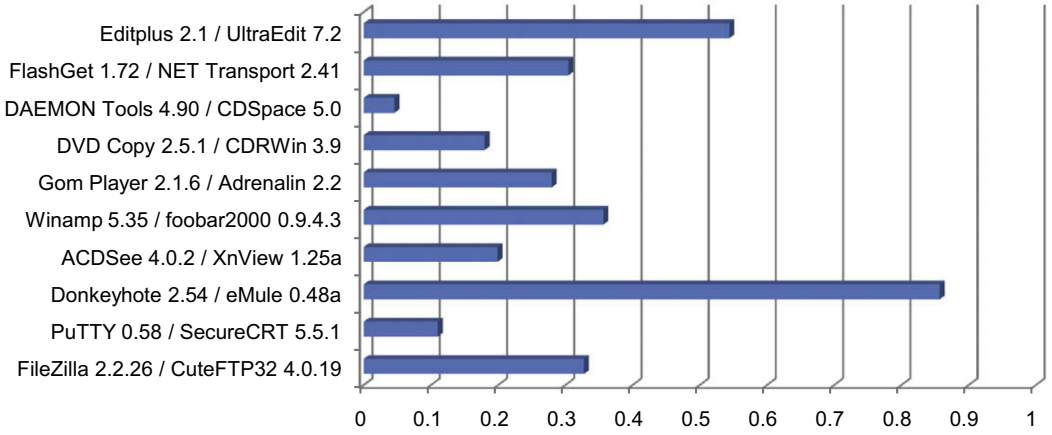


Fig. 7. Similarities between programs in the same category.

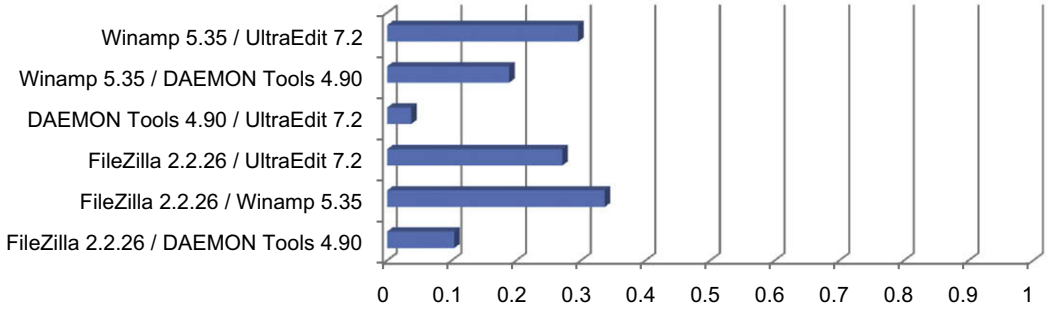


Fig. 8. Similarities between programs in different categories.

Table 3
frhed binaries compiled with various versions of compilers.

Compiler	Build option	Size of binary (B)	Number of functions without API call	Number of functions with API call
VC 6.0	Debug	409,668	441	218
	Release	317,952	479	221
VC 7.1	Debug	446,464	432	215
	Release	331,776	440	212
VC 8.0	Debug	716,800	534	218
	Release	377,344	453	215

Table 4
Similarities between *frhed* binaries generated by various compilers.

		VC++ 6.0		VC++ 7.1		VC++ 8.0	
		Debug	Release	Debug	Release	Debug	Release
VC 6.0	Debug	1.0000	0.9823	0.9809	0.9767	0.9694	0.9797
	Release	–	1.0000	0.9751	0.9755	0.9590	0.9780
VC 7.1	Debug	–	–	1.0000	0.9900	0.9793	0.9924
	Release	–	–	–	1.0000	0.9857	0.9977
VC 8.0	Debug	–	–	–	–	1.0000	0.9881
	Release	–	–	–	–	–	1.0000

Table 5
Comparison between Windows static birthmark and Windows dynamic birthmark.

Sample	Dynamic birthmark	Static birthmark
CXImage-UI-1 vs. CImage-UI-1 without input	SimEXESEQ = 0.748 SimEXEFREQ = 1.000	SimStatic = 0.051
CXImage-UI-1 vs. CImage-UI-1 with input(lena.bmp)	SimEXESEQ = 0.595 SimEXEFREQ = 0.990	
CXImage-UI-1 vs. CXImage-UI-2 without input	SimEXESEQ = 0.310 SimEXEFREQ = 0.991	SimStatic = 0.712
CXImage-UI-1 vs. CXImage-UI-2 with input(lena.bmp)	SimEXESEQ = 0.072 SimEXEFREQ = 0.780	

Debray (2003) prevents static disassembly of binary executables. Because the technique confuses IDAPro disassembler, we cannot even commence to analyze programs. However, the obfuscated binary code can be correctly disassembled by combining two disassembly algorithms (Kruegel et al., 2004). Function call obfuscation techniques that are widely used to hide calls in malwares can also confuse our birthmarking system. Call obfuscation can be solved by using abstract stack graphs (Lakhotia et al., 2005). In general, the binary code obfuscation techniques by automatic tools can be deobfuscated. Manually obfuscated code is very hard to automatically analyze. We discuss the effects of manual code obfuscation in Section 5.

4. Comparison with Windows dynamic birthmarks

In this section, we compare our birthmark with the Windows dynamic birthmarks based on API calls (Tamada et al., 2004b; Okamoto et al., 2006). The dynamic birthmarking system collects API calls at runtime. The birthmarks are the sequence and frequency of API calls. The weakness of dynamic birthmarks is that birthmarks are affected by input. For batch programs, it is relatively easy to determine the input (Myles and Collberg, 2004; Schuler et al., 2007). For interactive programs with a GUI, it is very difficult to determine the input, because programs are executed by GUI events, which generate Windows API calls. To avoid these problems, Okamoto et al. (2006) selected a scenario wherein an application is closed immediately after launch. The birthmarks extracted with this scenario only reflect initialization characteristics of the program and do not consider essential functionalities. Another weakness of dynamic birthmarks is that the birthmarks are affected by factors related to the given environment, such as resource usage and thread scheduling.

To compare our birthmark with the Windows dynamic birthmarks, we implemented a similar birthmarking system using the Microsoft Detour Windows hooking library (Brubacher and Hunt, 1999), because the dynamic birthmarking system is not publicly available. For comparison with our static Windows birthmarks, we also implemented sample image viewers that only read an image file and display the image. We wrote two sets of image viewers. One set has image viewers that have the same GUI but use different image processing libraries, that is, *CXImage*[11] and *CImage*.[12] The other set has image viewers that use the same library *CXImage* but have different GUIs. We used '`lena.bmp`' image as input.

Table 5 shows the results comparing the dynamic Windows API birthmarks with our static birthmark. In the sample column, CXImage and CImage represent image processing library names, and UI-1 and UI-2 represent the types of GUI. UI-1 is composed of a textbox that obtains an image file name and a panel to display images. UI-2 has menus, buttons, and a panel to display images. When the file open menu is selected or a file open button is clicked, a file dialog is opened and a user can select an image file to open. In the dynamic birthmark and the static birthmark columns, the resulting similarities are listed. *SimEXESEQ* is the similarity between two API call sequence birthmarks. *SimEXEFREQ* is the similarity between two API call frequency birthmarks. *SimStatic* is the similarity between our static API birthmarks.

The results of a comparison of CXImage-UI-1 and CImage-UI-1 without an input image reveal that the scenario of the Windows dynamic birthmark is inappropriate. Since different graphics libraries are used, the similarities should be low, but the result is $SimEXESEQ = 0.748$. Our static birthmark gives a more credible result with $SimStatic = 0.051$.

The results of a comparison of CXImage-UI-1 and CXImage-UI-2 with an input image reveal that the Windows dynamic birthmark is vulnerable to user interaction and resource usage. We can expect the similarity between two programs to be sufficiently high to reflect the common library part. But the result, $SimEXESEQ = 0.072$, shows that the two programs are totally different. We observed that the API calls generated by the execution of the file dialog control overwhelm the API calls generated by the execution of the graphics library. Without separating the interesting features of the programs, the dynamic birthmark can be dominated by the execution of the GUI modules. Our static birthmark gives more a credible result with $SimStatic = 0.712$.

We can conclude that the dynamic birthmark is highly susceptible to user interaction and frequently misses essential program

[11] http://www.xdp.it/cximage.htm.
[12] http://cimg.sourceforge.net./

paths, depending on the input scenario. Our static birthmark is more appropriate for interactive Windows applications.

5. Attacks on static API birthmark

Code thieves try to avoid theft detection. If they know a theft detection technique, they will apply specialized code obfuscation techniques to spoof the birthmark. Thus, it is necessary to deobfuscate the codes before applying our birthmarking technique. Theoretically perfect obfuscation is impossible (Barak et al., 2001). Hence, the underlying obfuscation techniques can be identified manually from an obfuscated code. If the underlying obfuscation techniques are known, deobfuscation is NP-easy (Appel, 2002). The NP-easy time complexity means that we can build deobfuscators that work faster than most sound program analyses. In this section, we describe three attacks and their effects on our birthmark, and discuss methods to defend our birthmark against these attacks.

5.1. Control obfuscation by pointer aliasing

Attackers can obfuscate branch instructions such as jumps or calls by control obfuscation techniques that utilize pointer aliasing. Control obfuscation techniques utilize indirect branch instructions, where the branch targets are computed at execution time (Wang, 2000). Udupa et al. (2005) showed that the intra-procedural control flow flattening is broken by applying well known static and dynamic analyses. They also showed that 79% of inter-procedural edges are resolved by their method. Attackers can obfuscate function calls such that the birthmark computation using descendant functions is affected. In this case we can apply a method to detect obfuscated calls (Lakhotia et al., 2005), wherein a Value Set Analysis (Balakrishnan and Reps, 2004) and Abstract Stack Graph (Lakhotia and Kumar, 2004) are employed. However, without deobfuscation, control obfuscation of function calls makes it impossible to identify call targets such that computing $B_f(f,k)$ with $k>0$ is impossible and only $B_f(f,0)$ can be computed. We observed that the average program similarity difference between program similarities computed with $k=0$ and $k=5$ is 6.5% in our benchmark programs. Because the program similarities with $k=0$ becomes higher than the program similarities with $k=3$, the possibility of false positives increases. We should then modify the threshold to a larger value.

5.2. Modification of call structure by function inlining and outlining

Because our birthmark relies on the structure of the call graph, modification of the call structure will change function birthmarks. Attackers can modify the call structure by introducing additional functions (e.g. outlining). For example, a function f can be transformed into $f_1 \rightarrow f_2 \rightarrow \cdots \rightarrow f_{k+1}$, where a right arrow denotes a function call such that f_1 calls f_2, f_2 calls f_3, …, and f_k calls f_{k+1}. If the code of f is copied into f_{k+1} and f_1, …, f_k only call f_2, …, f_{k+1} respectively, then $B_f(f_1,k)$ may not contain any API calls that were in $B_f(f,k)$. To be resilient to attacks of function insertion type, we should increase the call depth k in the function birthmarks to some extent such that the function birthmarks contain all API calls that exist in the original function. The call depth k in the function birthmarks is determined according to the call structures. Program similarities decrease as k increases until k becomes the maximum call depth of the program. Appropriate k is the maximum call depth of the programs. However, even though the original function and the root of the transformed functions are matched using the function birthmarks, other inserted functions may remain unmatched such that similarities between the original and the transformed programs decrease. For example, every function of a program can be outlined such that each function is divided into two functions such that the original program has n functions, the outlined program has $2n$ functions. In this case, the program similarity between the original program and the outlined program becomes $2n/(n+2n) = 2/3 = 66.7\%$. Further research is needed to identify such outlined functions.

Attackers can also modify call structure by merging functions (e.g. inlining). The same method as employed in function insertion attacks can be used to cope with function merge attacks. If we consider the transformed function as the original function, the inlined functions, which exist inside the transformed function, can be considered as outlined functions after program transformation.

5.3. Addition of redundant API calls by opaque predicate

An opaque predicate is a Boolean expression that is always evaluated as either true or false. By opaque predicates, attackers can insert API calls that will not be executed. These calls will be added to the function birthmark such that the similarity computation will fail. In this case, we should use containment instead of similarity because new API calls can be added to the birthmark and existing API calls cannot be removed from the birthmark. Containment between functions f_1 and f_2 is defined as

$$containment_f(f_1,f_2) = \frac{|F_1 \cap F_2|}{max(|F_1|,|F_2|)}$$

where F_1 and F_2 are function birthmarks of f_1 and f_2 respectively. We can also apply deobfuscation techniques to resolve opaque predicates (Dalla Preda et al., 2006, 2007).

6. Related work

There are similar research areas for program identification, for example, software forensics, authorship analysis, plagiarism detections, malware detections, software watermarking, and software birthmarking. They use software characteristics to identify programs.

Software birthmarks are related to software forensics in the sense that the program characteristics utilized by authorship analysis and software forensics can be used for software birthmarks. Spafford and Weeber (1993) proposed that software forensic methods for executable codes and source files to identify malicious code. The characteristics to identify authors in executable code are data structures and algorithms, used compiler and system information, choice of system and library calls, and bugs. Krsul and Spafford (1997) tried to identify authors by various programming style metrics that depend on the source code layout, such as indentation, comments, and identifier naming.

Plagiarism detection methods can be applied to detect a suspected copy for source code (Prechelt et al., 2002; Wise, 1996; Schleimer et al., 2003; Gitchell and Tran, 1999). Plagiarism detection tools calculate program similarity based on text itself, program styles, and program structures. Text-based and program layout-based comparison can be easily defeated by simple program modification. For example, program similarity is greatly reduced by stripping comments and simple renaming of identifiers. For this reason, many source code similarity checking tools utilize the structural information of programs. These detectors have been successfully applied to many computer science courses.

Malware detectors compare malware signatures with binary files by matching byte streams or strings. They are defeated by code obfuscation such as inserting *nop* instruction (Christodorescu and Jha, 2004). Dalla Preda et al. (2007) and Christodorescu and Jha (2003) suggested methods to identify malware using model checking and abstract interpretation of disassembled code. Flow graphs

of binary executables can be used to identify programs (Sabre-Security, 2007,). Bonfante et al. (2007) used control flow graphs to identify malwares. Sabre-Security (2007) showed a copy detection example between two MacOS emulators named CherryOS and PearPC using BinDiff,[13] which compares two binaries by control flows and function calls.

Software watermarking is used to prove ownership of stolen software (Collberg and Thomborson, 1999, 2003). A software watermarking system can embed watermarks to code and recognize the embedded watermarks from the code. A software watermark is different from a software birthmark in that the watermark adds extra code to the original code before releasing the software whereas a birthmark does not embed additional code. The other difference is that a watermark can provide program owner information, while a birthmark only gives similarities between two programs.

Tamada et al. (2004a, 2005) suggested static Java birthmarks. The birthmarks are constant values in field variables, sequence of method calls, inheritance structure, and used classes. These birthmarks are selected because Java code obfuscators seldom change these program properties. The result of these birthmarks is the average similarities from the four birthmarks. They evaluated their birthmark using four sample programs: BCEL, ANT, JUnit and JBirth. The experimental results show that their birthmarks can distinguish different programs with a high ratio and they are resilient to code obfuscation and compilation. However, their birthmarks are vulnerable to manual modification attacks and cannot be applied to small sized classes that contain few or no field variables, method calls, inheritances, or used classes.

Myles and Collberg (2005) proposed a static birthmark for Java using *k*-gram, which has been used to calculate document similarity. The *k*-gram birthmark is a set of unique Java bytecode sequences of length *k*. For each method in modules, they compute the set of unique *k*-grams by sliding a window of length *k* over the static instruction sequence. They evaluated this birthmark with several tiny Java modules: `factorial`, `fibonacci`, `decode`, `fft`, and `wc`. The result shows that the *k*-gram birthmark is precise, but highly susceptible to program transformations.

Myles and Collberg (2004) proposed a dynamic birthmark for Java called whole program path (WPP) birthmark. WPP is a directed acyclic graph (DAG) representation of context-free grammar that generates a program's acyclic path (Larus, 1999). To obtain WPP, the dynamic trace of a program is obtained by code instrumentation and the trace is compressed into a DAG using the SEQUITUR algorithm. They used WPP as their birthmark and computed similarity using a graph distance metric between two WPPs (Bunke and Shearer, 1998). They evaluated WPP birthmarks with a few tiny Java programs. The results show that the credibility and the resilience of the WPP birthmark fall between those of the static Java birthmark of Tamada et al. and the *k*-gram birthmark.

Schuler et al. (2007) suggested a dynamic Java birthmark called Java API birthmark. The birthmark is the union of *k*-long call sequences observed by API objects during the execution of a program with a given input. To collect the API call sequence, program instrumentation is performed on each API call statement such that each API call is redirected to proxy methods that record the API call sequence and back to the original API call. They evaluated this birthmark with image processing programs and XML processors. The experimental results show that the birthmark is highly credible and resilient to code obfuscators. They also compared their birthmark with the WPP birthmark and showed that their birthmark is more scalable and resilient than the latter.

Tamada et al. (2004b) introduced dynamic birthmarks for Windows applications. The birthmarks are the sequence (EXESEQ) and frequency (EXEFREQ) of API function calls at run-time. They evaluated this dynamic birthmarking technique with a few MP3 tag editors. The birthmarks may change according to user interaction and system environment. To obtain credible birthmarks, the birthmarks were extracted with no input such that they immediately closed the programs after launching the programs. Hence, the birthmarks do not reflect the essential features of the programs.

This paper is an extended version of our previous work (Choi et al., 2007). Changes were made to provide more detailed explanation of our birthmark and to present a comparison with the previous Windows dynamic birthmark of Tamada et al.

7. Summary and future work

In this paper we propose a static birthmarking technique to identify ownership and similarity of binary executables. We define a function birthmark as a set of API calls of functions within *k* depths. The program similarity is defined as the maximum value among all possible function matchings. We model the problem of calculating the similarity between two programs as finding a maximum weighted bipartite matching. The similarity between two programs is obtained by the shortest augmenting path algorithm of the Jonker and Volgenant algorithm.

We used a set instead of a sequence or other structures. A set is more abstract than a sequence or a graph. It is difficult to extract API call sequences without execution. Regarding sequences, it is difficult to precisely extract sequences from assembly code. Birthmarks represented as graphs are complex to compare.

We evaluate the proposed birthmark by comparing various categories of Windows applications. To show the credibility, same programs with different versions, programs in the same categories, and programs from different categories are compared. To show the resilience, we compare binary executables compiled with various compilers. The empirical results show that the similarities obtained using our birthmark can provide sufficient indication for the functional and structural similarities among programs.

We compare our birthmark with the Windows dynamic birthmarks. We implemented a dynamic birthmark system and experimented with the same benchmark programs. The result shows that the dynamic birthmark is susceptible to user interaction and misses essential program paths depending on the input scenario. We conclude that current dynamic birthmark system is not suitable for interactive programs but for batch programs. Our static birthmarking system is more appropriate for Windows applications.

Our birthmark relies on the API function calls. The birthmarks of programs that rarely use API calls such as encoders, decoders, and scientific applications may be very inaccurate. Birthmarks of these applications should detect the algorithmic structure of the program. To detect such algorithm theft, manual reverse engineering using debuggers is required.

We explain three possible attacks on our static API birthmarks. The first attack is control flow obfuscation. Attackers can modify control flows and hide branch targets (Lakhotia and Singh, 2003). We can apply deobfuscation techniques suggested by Lakhotia and Kumar (2004), Balakrishnan and Reps (2004), Lakhotia et al. (2005), and Udupa et al. (2005). The second attack is call structure modification. Attackers can add or remove functions to disturb birthmark computation. To provide resilience to this attack, it is necessary to increase the call depth *k* in the function birthmarks to some extent such that the function birthmarks contain all API calls that exist in the original function. We also need to remove or merge functions to restore transformed functions to their original functions. Further research is needed to find such functions. The third attack is addition of redundant API calls by opaque predicate. Because attackers cannot remove API calls, we can change

[13] http://www.sabre-security.com/products/bindiff.html.

our comparison method from similarity to containment. We can also apply deobfuscation techniques by abstract interpretation to resolve opaque predicates (Dalla Preda et al., 2006, 2007).

For future work, we plan to improve our birthmark by applying the proposed deobfuscation methods. We also plan to extend our birthmark with control flow information and non-API features such as instructions.

Acknowledgements

This work was partially supported by the Korea Science and Engineering Foundation(KOSEF) grant funded by the Korea government(MEST) (No. R01-2008-000-11856-0). Also, this work was partially supported by the MKE(Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) Support program supervised by the IITA(Institute of Information Technology Advancement)" (IITA-2008-C1090-0801-0020).

References

Appel, A., 2002. Deobfuscation is in NP. Preprint available from <http://www.cs.princeton.edu/appel/papers/deobfus.pdf>.

Balakrishnan, G., 2007. WYSINWYX: What You See Is Not What You eXecute. Ph.D. Thesis, University of Wisconsin-Madison.

Balakrishnan, G., Reps, T., 2004. Analyzing memory accesses in x86 executables. Compounds Construction, 5–23.

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K., 2001. On the (im) possibility of obfuscating programs. Lecture Notes in Computer Science 2139, 19–23.

Bergeron, J., Debbabi, M., Desharnais, J., Erhioui, M., Lavoie, Y., Tawbi, N., 2001. Static detection of malicious code in executable programs. International Journal of Requirements Engineering.

Bergeron, J., Debbabi, M., Erhioui, M., Ktari, B., 1999. Static analysis of binary code to isolate malicious behaviors. In: (WET ICE'99) Proceedings of the IEEE Eighth International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, pp. 184–189.

Bonfante, G., Kaczmarek, M., Marion, J., 2007. Control Flow Graphs as Malware Signatures. WTCV, May.

Brubacher, D., Hunt, G., 1999. Detours: binary interception of Win32 functions. In: Proceedings of the Third USENIX Windows NT Symposium, pp. 135–143.

Bunke, H., Shearer, K., 1998. A graph distance metric based on the maximal common subgraph. Pattern Recognition Letters 19 (3–4), 255–259.

Choi, S., Park, H., Lim, H., Han, T., 2007. A static birthmark of binary executables based on API call structure. In: Proceedings of the 12th Annual Asian Computing Science Conference.

Christodorescu, M., Jha, S., 2003. Static analysis of executables to detect malicious patterns. In: Proceedings of the 12th USENIX Security Symposium, pp. 169–186.

Christodorescu, M., Jha, S., 2004. Testing malware detectors. ACM SIGSOFT Software Engineering Notes 29 (4), 34–44.

Collberg, C., Myles, G., Huntwork, A., 2003. Sandmark-A tool for software protection research. Security and Privacy Magazine, IEEE 1 (4), 40–49.

Collberg, C., Thomborson, C., 1999. Software watermarking: models and dynamic embeddings. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 311–324.

Collberg, C., Thomborson, C., Low, D., 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 184–196.

Dalla Preda, M., Christodorescu, M., Jha, S., Debray, S., 2007. A semantics-based approach to malware detection. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 377–388.

Dalla Preda, M., Madou, M., De Bosschere, K., Giacobazzi, R., 2006. Opaque predicates detection by abstract interpretation. Algebraic Methodology and Software Technology, 81–95.

Dean, J., Grove, D., Chambers, C., 1995. Optimization of object-oriented programs using static class hierarchy analysis. In: Proceedings of the Ninth European Conference on Object-Oriented Programming, pp. 77–101.

Ellson, J., Gansner, E., Koutsofios, L., North, S., Woodhull, G., 2001. Graphviz-open source graph drawing tools. Graph Drawing, 483–485.

Ferraiolo, J., Jun, F., Jackson, D., 2003. Scalable Vector Graphics (SVG) 1.1 Specification. W3C Recommendation 14.

Gabow, H., Tarjan, R., 1989. Faster scaling algorithms for network problems. SIAM Journal on Computing 18 (5), 1013–1036.

Gitchell, D., Tran, N., 1999. Sim: a utility for detecting similarity in computer programs. In: Technical Symposium on Computer Science Education, pp. 266–270.

Goldberg, A., Kennedy, R., 1995. An efficient cost scaling algorithm for the assignment problem. Mathematical Programming 71 (2), 153–177.

Guilfanov, I., 1997. FLIRT Fast Library Identification and Recognition Technology. URL: <http://www.datarescue.com/idaflirt.htm>.

Hofmeyr, S., 1998. Intrusion detection using sequences of system calls. Journal of Computer Security 6 (3), 151–180.

Jonker, R., Volgenant, A., 1987. A shortest augmenting path algorithm for dense and sparse linear assignment problems. Computing 38 (4), 325–340.

Koutsofios, E., North, S., 1993. Drawing graphs with dot. AT&T Bell Laboratories, Murray Hill, NJ.

Krsul, I., Spafford, E., 1997. Authorship analysis: identifying the author of a program. Computers and Security 16 (3), 233–257.

Kruegel, C., Robertson, W., Valeur, F., Vigna, G., 2004. Static disassembly of obfuscated binaries. In: Proceedings of the 13th USENIX Security Symposium (Security04).

Kwon, J., 2008. N-gram Based Static Birthmark for Windows Binary Applications. Master's Thesis, KAIST.

Lakhotia, A., Kumar, E., 2004. Abstracting stack to detect obfuscated calls in binaries. In: Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation, pp. 17–26.

Lakhotia, A., Kumar, E., Venable, M., 2005. A method for detecting obfuscated calls in malicious binaries. IEEE Transactions on Software Engineering 31 (11), 955–968.

Lakhotia, A., Singh, P., 2003. Challenges in getting formal with viruses. Virus Bulletin 9 (1), 14–18.

Larus, J., 1999. Whole program paths. In: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, pp. 259–269.

Linn, C., Debray, S., 2003. Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, pp. 290–299.

Madou, M., Van Put, L., De Bosschere, K., 2006. Loco: an interactive code (de)obfuscation tool. In: Proceedings of ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM'06). ACM Press, Charleston, South Carolina, pp. 140–144.

Myles, G., Collberg, C., 2004. Detecting software theft via whole program path birthmarks. Information Security Conference, 404–415.

Myles, G., Collberg, C., 2005. K-gram based software birthmarks. In: Proceedings of the 2005 ACM Symposium on Applied computing, pp. 314–318.

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

Okamoto, K., Tamada, H., Nakamura, M., Monden, A., Matsumoto, K., 2006. Dynamic software birthmarks based on API calls. IEICE Transactions on Information and Systems 89 (8), 1751–1763.

Pande, H., Ryder, B., 1996. Data-flow-based virtual function resolution. In: Proceedings of the Third International Symposium, Sas' 96, Static Analysis, September 24–26, Aachen, Germany.

Prechelt, L., Malpohl, G., Philippsen, M., 2002. Finding plagiarisms among a set of programs with JPlag. Journal of Universal Computer Science 8 (11), 1016–1038.

Rajagopalan, M., Hiltunen, M., Jim, T., Schlichting, R., 2005. Authenticated system calls. In: Proceedings of the International Conference on Dependable Systems and Networks, DSN 2005, pp. 358–367.

Sabre-Security, 2007. Using BinDiff for Code Theft Detection. <http://www.sabre-security.com/products/CodeTheft.pdf>.

Schleimer, S., Wilkerson, D., Aiken, A., 2003. Winnowing: local algorithms for document fingerprinting. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 76–85.

Schuler, D., Dallmeier, V., Lindig, C., 2007. A dynamic birthmark for Java. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering.

Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P., 2001. A fast automation-based method for detecting anomalous program behaviors. In: IEEE Symposium on Security and Privacy, pp. 144–155.

Spafford, E., Weeber, S., 1993. Software forensics: can we track code to its authors? Computers and Security 12 (6), 585–595.

Sung, A., Xu, J., Chavez, P., Mukkamala, S., 2004. Static analyzer of vicious executables (SAVE). In: 20th Annual, Computer Security Applications Conference, pp. 326–334.

Tamada, H., Nakamura, M., Monden, A., Matsumoto, K., 2004a. Design and evaluation of birthmarks for detecting theft of Java programs. In: Proceedings of the IASTED International Conference on Software Engineering (IASTED SE 2004), pp. 569–575.

Tamada, H., Nakamura, M., Monden, A., Matsumoto, K., 2005. Java birthmarks – detecting the software theft. IEICE Transactions on Information and Systems 88 (9), 2148–2158.

Tamada, H., Okamoto, K., Nakamura, M., Monden, A., Matsumoto, K., 2004b. Dynamic software birthmarks to detect the theft of windows applications. In: International Symposium on Future Software Technology, vol. 20(22).

Udupa, S., Debray, S., Madou, M., 2005. Deobfuscation: reverse engineering obfuscated code. In: Proceedings of the 12th Working Conference on Reverse Engineering, pp. 45–54.

Wang, C., 2000. A Security Architecture for Survivability Mechanisms. Ph.D. Thesis, University of Virginia.

Warrender, C., Forrest, S., Pearlmutter, B., 1999. Detecting intrusions using system calls: alternative data models. In: IEEE Symposium on Security and Privacy, p. 145.

Wise, M., 1996. YAP3: improved detection of similarities in computer program and other texts. In: Proceedings of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, pp. 130–134.

Seokwoo Choi received his B.S. and M.S. degrees in computer science from Korea Advanced Institute of Science and Technology, Korea, in 1998 and 2000, respectively. He is currently a Ph.D. degree student in the Division of Computer Science, Korea Advanced Institute of Science and Technology. His current research interests include program security, clone detection, and reverse engineering.

Heewan Park received his B.S. degree in Computer Engineering from Dongguk University, Korea, in 1997, M.S. degree in Computer Science from Korea Advanced Institute of Science and Technology, Korea, in 1999. From 2004 to 2007, he was a senior engineer at Samsung Electronics, Co., Ltd. He is currently a Ph.D. degree student in the Division of Computer Science, Korea Advanced Institute of Science and Technology. His current research interests include reverse engineering, software obfuscation, software watermarking and software birthmarking.

Hyun-il Lim received his B.S. and M.S. degrees in computer science from Korea Advanced Institute of Science and Technology, Korea, in 1995 and 1997, respectively. He is currently a Ph.D. degree student in the Division of Computer Science, Korea Advanced Institute of Science and Technology. His current research interests include program analysis, software security, and reverse engineering.

Taisook Han received his B.S. degree in electrical engineering from Seoul National University, Korea in 1976, M.S. degree in computer science from Korea Advanced Institute of Science and Technology, 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 Division of Computer Science, Korea Advanced Institute of Science and Technology. His current research interests include programming language theory, software safety, and verification of embedded systems.