A Static Birthmark of Binary Executables Based on API Call Structure*

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

Division of Computer Science and
Advanced Information Technology Research Center(AITrc).
Korea Advanced Institute of Science and Technology
{swchoi,hwpark,hilim}@pllab.kaist.ac.kr, han@cs.kaist.ac.kr

Abstract. A software birthmark is a unique characteristic of a program that can be used as a software theft detection. In this paper we suggest and empirically evaluate a static birthmark of binary executables based on API call structure. The program properties employed in this birthmark are functions and standard API calls when the functions are executed. The API calls from a function includes the API calls explicitly found from the function and its descendants within limited depth in the call graph. To statically identify functions, call graphs and API calls, we utilizes IDAPro disassembler and its plug-ins. We define the similarity between two functions as the proportion of the number of all API calls to the number of the common API calls. The similarity between two programs is obtained by the maximum weight bipartite matching between two programs using the function similarity matrix. To show the credibility of the proposed techniques, we compare the same applications with different versions and the various types of applications which include text editors, picture viewers, multimedia players, P2P applications and ftp clients. To show the resilience, we compare binary executables compiled from various compilers. The empirical result shows that the similarities obtained using our birthmark sufficiently indicates the functional and structural similarities among programs.

Keyword: software piracy, software birthmark, binary analysis.

1 Introduction

Recently a large amount of software is developed in the form of open source projects. Most open source projects contain software licenses. A widely used software license for open source software is the GNU Public License(GPL). The GPL allows developers to use software freely, but requires new projects using the original work to be licensed under the GPL. There are also more permissive software licenses like the MIT license and the BSD licenses which allow the original source code to be combined in commercial software. The permissive licenses, however, require the copyright notice of the original software to be included.

* This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center(AITrc).

I. Cervesato (Ed.): ASIAN 2007, LNCS 4846, pp. 2–16, 2007.

There have been reported that many companies use open source software for commercial purpose without permission. To detect code theft when source code is available, we can utilize well-known plagiarism detection tools like MOSS, JPlag and YAP [1,2,3]. Suppose that source code under the GPL is contained in commercial software, which is distributed in compiled binaries without indicating the copyright notice of the original software. In this case, we need to prove whether the open source code is used or not in the binary executables. Software birthmarking is one of the techniques to solve such software theft problems.

A software birthmark is unique characteristics of a program that can be used to identify the program. If program p and q have the same or very similar birthmark, q is very likely to be a stolen copy of p (and vice versa). Comparing the strings analyzed from binary executables can be a easy birthmarking technique. In this case, a set of strings is a birthmark. Sometimes comparing the structures of binaries can be a good birthmark technique. For example, SabreSecurity BinDiff effectively found similarities between two MacOS emulators named CherryOS and PearPC[4,5]. Tamada et al. suggested a dynamic software birthmark for Windows applications using Win32 API function call sequences [6,7]. Dynamic birthmarks extract program properties from a program execution trace when a sequence of input is given, while static birthmarks extract properties only from the program itself.

In this paper we propose a new static birthmarking technique that can help to identify ownership and similarity of binary executables. Program properties used as our birthmark are summaries extracted from each binary function in a program. The summary of each function is a set of possible standard API calls when the function is executed. We statically identify API function calls by analyzing disassembled code which is generated by IDAPro disassembler[8]. A similarity between two functions is calculated by comparing API call sets of two functions. A similarity between two programs is obtained by matching problem.

We evaluate the proposed birthmark by comparing various categories of Windows applications. To show the credibility, the same applications with different versions are compared. To show the resilience, we compare binary executables compiled from various compilers. The empirical result shows that the similarities obtained using our birthmark sufficiently indicate the functional and structural similarities among programs.

2 Related Work

There are three major threats against the intellectual property contained in software. *Software piracy* is the illegal use, duplication or reselling of legally protected software. *Software tampering* is the illegal modification of software to gain control over restricted code or digital media protected by the software. *Malicious reverse engineering* is the extracting of a piece of a program in order to reuse it in one's own. To deal with these threats, several techniques have been explored, for example, software watermarking to deal with piracy, code obfuscation to deter reverse engineering, and software tamper-proofing [9].

Software watermarking is a well-known technique used to provide a way to prove ownership of stolen software. Software watermarking systems embed watermarks in software and recognize the watermarks. Software watermark can be either static or dynamic [10,11]. Unfortunately, watermarking is not always feasible because it requires software developers to embed a watermark before releasing the software.

Software birthmarking is a technique that identifies the inherent characteristics occurring in a program by chance. Unlike software watermarks, software birthmarks do not embed additional code or identifier. Instead a birthmark relies on an inherent characteristic of the application to show that one program is a copy of another. The result of comparing two programs with software birthmarking is similarities between two programs. With the similarities, we are able to say that one program is a copy of another, totally or in part.

Tamada et al. [12,13] suggested the first practical application of static software birthmarks to identify the theft of programs. This technique is specific to Java class files which is a combination of four individual birthmarks: constant values in field variables(CVFV), sequence of method calls(SMC), inheritance structure(IS), and used classes(UC). These four birthmarks could be used individually but the combination makes this technique more reliable. Their experiment with several sample programs shows that the proposed birthmarks identify a class within a program with high precision, but can easily be confused by several obfuscation techniques.

Tamada et al. [6,7] introduced dynamic birthmarks and proposed two birthmarks based on the trace of system calls for Windows programs. The dynamic birthmarks are the sequence and frequency of API function calls during execution of software. They claim that these birthmarks are reasonably robust against program transformations. The credibility of this birthmark highly relies on user interactions, inputs and system environments. To avoid this weakness, they highly restricted inputs and user interactions in the experiments.

Myles et al. proposed a k-gram based static birthmark [14]. They adopted k-gram, which have been previously used to detect similarity between documents, as their birthmark for Java applications. The k-gram birthmark is the set of unique opcode sequence of length k. For each method in a module they compute the set of unique k-grams by sliding a window of length k over the static instruction sequence. k-gram based birthmark is precise, but highly susceptible to program transformations. They evaluated this birthmarking techniques with several tiny Java programs.

Myles et al. [15,16] proposed the concept of another dynamic birthmark known as Whole Program Path(WPP) birthmark. A WPP is a directed acyclic graph(DAG) representation of a context-free grammar that generates a program's acyclic path [17]. To get WPP, dynamic trace of a program is obtained by instrumentation, and the trace is compressed into a DAG using SEQUITUR algorithm. They used WPP as their birthmarks and computed similarity between two birthmarks using a graph distance for maximal common subgraph [18]. They experimented WPP birthmarking technique with a few tiny Java programs. The

result shows that the credibility and the resilience of the WPP birthmark is between the static Java birthmark of Tamada et al. and k-gram birthmark.

Shuler and Dallmeier[19] presented a dynamic birthmarking technique based on the API call sequence sets during program execution. The result of this birthmark, like Tamada's, is also highly dependent on user interactions, inputs and environments. They improved their reliability by limiting windows length to 5, like k-gram. They evaluated this birthmark on image processing programs and XML processors. This birthmarking technique can be applied to more realistic applications compared to Myles's birthmarking technique.

3 Static API Call Structure Birthmark

3.1 Software Birthmarks

Tamada et al.[13] and Myles et al.[14] formally defined a birthmark of a software using copy relations. The followings are the definition of birthmark by Myles et al.

Definition 1 (Birthmark). *Let p, q be programs. Let f be a method for extracting a set of characteristics from a program. Then $f(p)$ is called a birthmark of p iff:*
1. $f(p)$is obtained only from p itself (without any extra information), and
2. q is a copy of $p \Rightarrow f(p) = f(q)$.

Condition 1 explains the main difference between watermarking and birthmarking. A birthmark extracts characteristics only from the program itself, while a watermark extracts extra copyright information which is previously embedded by authors or program distributors. Condition 2 means that if p and q are in copy relation, the birthmark of p and the birthmark of q is the same. In this definition, the meaning of *copy* implies not only the exact duplication but also the semantics preserving transformation. But when we say the semantics are preserved, we do not mean that the two implementations with the same specification have the same birthmark. The terms are introduced to require the birthmark to be resilient to the obfuscation transformation to avoid theft detection. Thus a good birthmark value should not change after a slight semantics preserving modification of the program.

The following properties are restatements of those of Tamada et al. [13] and Myles[15]. These properties suggest two evaluation criteria which a birthmark should meet.

Property 1 (Distinction). Let p and q be programs with the same functionality. If p and q are implemented independently, then $f(p) \neq f(q)$.

Property 2 (Preservation). For p' obtained from p by any program transformation, $f(p) = f(p')$ holds.

Property 1 explains the distinction property. The distinction property complements Condition 2 of the birthmark definition. It is a criteria related to the possibility of false positives. It means that a good birthmark should catch copy relations well, while it should not falsely say two independently implemented programs with the same functionality are copy.

Property 2 is concerned with the resilience of a birthmark. If a copied code is transformed by compilers, optimizers or obfuscators, the appearance of the transformed code, which is in binary executables in this work, is different from the original code. If a birthmark is resilient to program transformations, it should only catch the inherent properties of the programs.

3.2 Proposed Birthmark

Microsoft Windows applications normally use the Windows API which offers essential libraries for developing Windows applications. The Windows API consists of functional categories which are system managements, diagnostics, graphics, multimedia, networking, security, system services, and user interfaces. Since Windows applications exploit Windows OS capabilities via the Windows API calls, the API calls are hard to be replaced. The API calls reflect the functionalities of a program, that is, the inherent characteristics of the program. If we can analyze the API call patterns correctly, we can use the patterns as a good birthmark of a program.

Previous birthmark research on Windows binaries [6,7] utilized dynamic API call sequences by hooking the executable file. The dynamic API call sequence only shows API call patterns for a given execution trace. The resulting birthmarks are dependent on inputs, user interactions and system environments. Furthermore it cannot cover whole program path. If given inputs do not lead the execution to the theft codes, the dynamic birthmarking cannot give us a meaningful answer. We here suggest a static birthmarking using Windows API calls. The proposed birthmarking technique analyzes whole part of given programs. Therefore it can catch the containment of the theft code. To compare two binaries we use assembly codes generated by IDAPro disassembler. We can also get function information, branch instructions and external calls from IDAPro.

Our birthmark is defined using API call set. We can also consider multisets instead of API call set since we have call graphs. Multisets reflects API call structure more precisely, while they are vulnerable to program transformation like inlinng or wrapping of functions. Inlining occurs when compiler optimizes, and wrapping often occurs when compiled with debug option. For this reason, we currently considers only sets of API calls.

Definition 2 (API Call Set). *API call set of a function is a collection of possible standard API calls that the function can invoke when the function is executed.*

According to the definition of API call set, the API call set of the main function covers all API calls that the program can reach. We simplify the API call set using call depth.

Definition 3 (k-depth API Call Set). *Let k be a integer (with $k \geq 0$). The k-depth API call set of a function is a collection of all possible standard API calls gathered from functions having call depths within k.*

A *k-depth API call set* of a function is a subset of *API call set* of the function. By limiting call depths, though we lose a little precision, we can calculate API call sets in reasonable time. Our experiment showed that the call depth as small as 2 or 3 sufficiently estimates the properties of functions.

Our static API call structure birthmark for a program is defined as follows.

Definition 4 (Static API Call Structure Birthmark). *Given API call sets for each function of a program, a static API call structure birthmark of the program is the collection of all API call sets.*

3.3 Calculating Similarity

To calculate similarity by the proposed birthmark, we first calculate all similarities between functions. A Similarity between two functions is defined as follows.

Definition 5 (Function Similarity). *Let S_A and S_B be sets of standard API call sets of function A and function B. Similarity between two functions is defined as*

$$sim_f(A, B) = \frac{2\ |S_A \cap S_B|}{|S_A| + |S_B|}$$

where $|S_A|$ is the cardinality of S_A, $|S_B|$ is the cardinality of S_B, and $|S_A \cap S_B|$ is the cardinality of common API calls of S_A and S_B.

The similarity function measures the fraction of common calls over all standard 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 become 0. Figure 1 illustrates matching between functions. According to Definition 5, the similarity between `foo` and `joo` is 2/3, and the similarity between `goo` and `koo` is 3/4.

We want to match functions between two programs such that the grand total of the similarities has a maximum value. We compute similarities between all possible pairs of functions between two programs. After the similarity calculation, we can get a $|A|$ by $|B|$ similarity matrix where A and B are the set of functions in each program. With the similarity matrix, we compute the program similarity. We define a program similarity as follows.

Definition 6 (Program Similarity). *Let P_1, P_2 be programs, $|P_1|$ and $|P_2|$ be numbers of functions in P_1 and P_2. We define the program similarity between P_1 and P_2 as*

$$sim_p(P_1, P_2) = \frac{2\ \sum_{(A,B) \in match(P_1,P_2)} sim_f(A, B)}{|P_1| + |P_2|}$$

where $match(P_1, P_2)$ is a set of matched functions between P_1 and P_2.

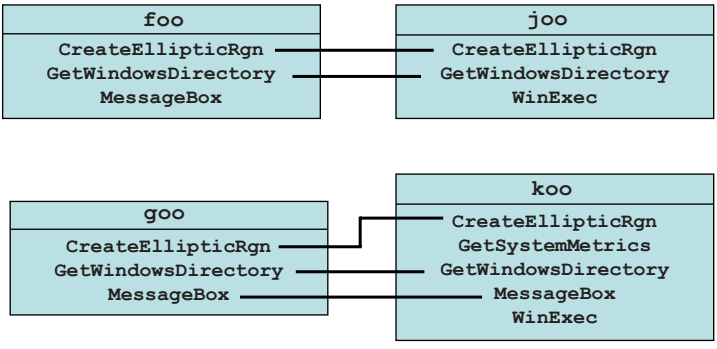

Fig. 1. Matching functions to compute similarity

The program similarity we defined is the maximum value among all possible function matching configurations. The problem maximizing the sum of similarities between the functions from program P_1 and the functions from program P_2 is isomorphic to the weighted $X-Y$ bipartite matching problem. Each function corresponds to each node. The functions from P_1 belong to the partition X of the bipartite graph, and the functions from P_2 belong to Y. Matching from a function from P_1 with a function from B corresponds to inserting an edge from a node in X to a node in Y. Similarities between two functions correspond to weights of edges. To find a maximum matching, we use the Hungarian algorithm[20] which solve the problem in polynomial time. The time complexity of the Hungarian algorithm is $O(n^3)$. Since the algorithm by default performs minimization, we use the difference matrix of which each element has a difference value instead of a similarity. A difference value is $1-similarity$. The program similarity obtained by the Hungarian algorithm is the maximal similarity between two programs.

4 Implementation

Figure 2 shows the structure of the static API Call birthmark system. This system operates as follows.

Step1: Generating idb file

IDAPro generates the IDA database file(.idb) by disassembling and analyzing the binary executable of sample program. We use IDAPro 5.1 for front-end.

Step2: Generating database file

Database file(.db) is generated from idb file using IDA2SQLite3 plug-in. This plug-in stores initial analysis result from IDAPro in sqlite3 database format for future use. Stored item includes program information such as function name, start address of that function, call graphs, assembly codes, etc.

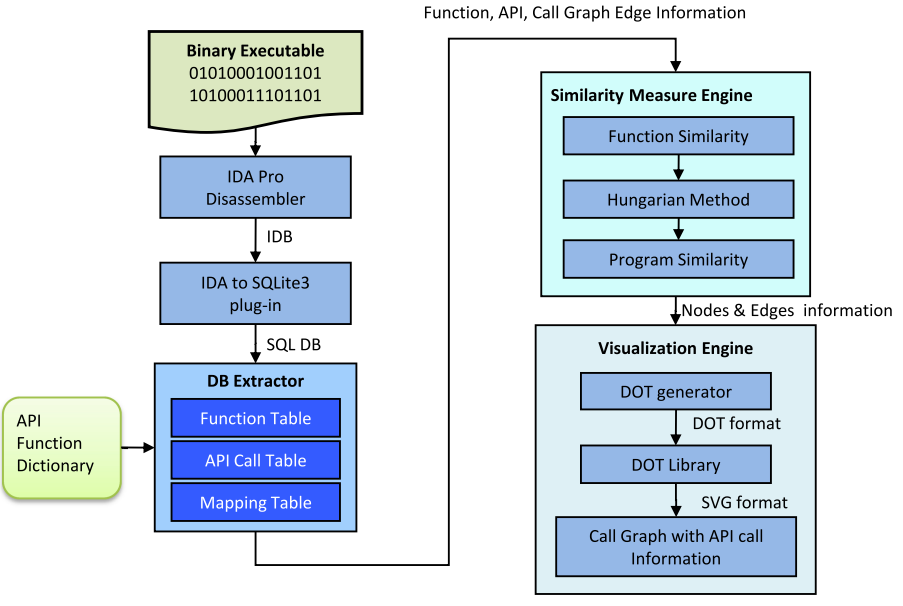

Fig. 2. The architecture of static API call structure birthmark system

Step3: Extracting function, API call, and Mapping table

Function table, API call table and function mapping table are obtained from the sqlite3 database file. Function table contains information about the function name, start address of the function, and library flag, etc. API call table contains API call instructions used in each functions. Mapping table contains the call relation between functions in the program. This program is developed with python 2.5 and pysqlite 2.3.3.

Step4: Calculating program similarity

This routine calculates program similarities using the information of function table, API call table, and function table. From the information of tables, function call graph is generated and set of API calls which can be used in each function is collected. As function call depth increases, each function collects API names of functions that are reachable from the function in the call graph. In this way, as function call depth increases, the number of APIs included in function is also increased. After forwarding APIs by predefined call depth k, API differences between every function in each program are calculated and API difference matrix is constructed. The maximum similarity value is calculated from this matrix using the Hungarian method. Similarity calculation program is implemented in C++. To check function matching result, call graphs with matching information is generated in DOT format. DOT file is translated into SVG(scalable vector graphic) format. Resulting SVG file can be displayed using SVG Viewers.

5 Evaluation

To evaluate the effectiveness of our static API Call birthmark, we conduct two experiments here. The first experiment evaluates credibility of our proposed birthmarks. The second experiment measures resilience of the birthmark against different compilers. To evaluate credibility, we chose some programs in various categories like text editors, FTP clients, Terminals, etc. Sample programs are listed in Table 1.

Table 1. Sample programs

Category	Program 1	Versions	Program 2	Versions
Text Editors	UltraEdit	7.0 / 7.2	Edit Plus	2.0 / 2.1
FTP clients	FileZilla	2.2.14 / 2.2.26	CuteFTP32	3.5.4 / 4.0.19
Terminals	Putty	0.56 / 0.58	SecureCRT	5.5.0 / 5.5.1
P2P clients	Dongkeyhote	2.40 / 2.54	Emule	0.45b / 0.47c
Graphic Tools	ACDSee	4.01 / 4.02	xnView	1.21 / 1.25a
MP3 Players	Winamp	5.23 / 5.35	Foobar2000	0.9.1 / 0.9.4
Video Players	GOM Player	2.0.0 / 2.1.6	Adrenalin	2.1 / 2.2
CD Burners	CDRWin	3.8 / 3.9	DVDCopy	2.2.6 / 2.5.1
Download Managers	Flashget	1.6.5 / 1.7.2	NetTransport	2.3.0 / 2.4.1
Disk Image Emulators	Daemon	4.3.0 / 4.9.0	CD Space	5.0

To evaluate resilience, we chose open source hex editor *frhed*[21] and *Microsoft Visual C++ 6.0, .NET 2003 and .NET 2005* compilers.

To verify the effectiveness of our static API call structure birthmark, we examined API call distributions of sample programs. Figure 3 shows that more than 100 functions have at least one API calls. z-axis represents each program. This result shows that the API call structure reflects the unique characteristics of programs.

Table 2 shows that the similarities of programs are changed by call depth. As call depth increases, the similarity decreases in most cases while the accuracy increases. We should limit call depth to compute API call set in reasonable time. Given the call depth k and the number of nodes n, the time complexity to find k-depth API call set for all functions is $O(k\,n^2)$. Our experiment showed that the call depth as small as 2 or 3 will sufficiently estimates the properties of functions. Hereafter our experiments are evaluated using 3-depth API call sets.

5.1 Credibility

Different Versions of Same Programs. To evaluate credibility of our birthmark, we compared different versions of same programs. Figure 5 shows that similarities between the same programs with a little different versions are over 0.7. In general a minor upgraded version of software shares almost all code from the previous version. This can be restated that the new version copied most

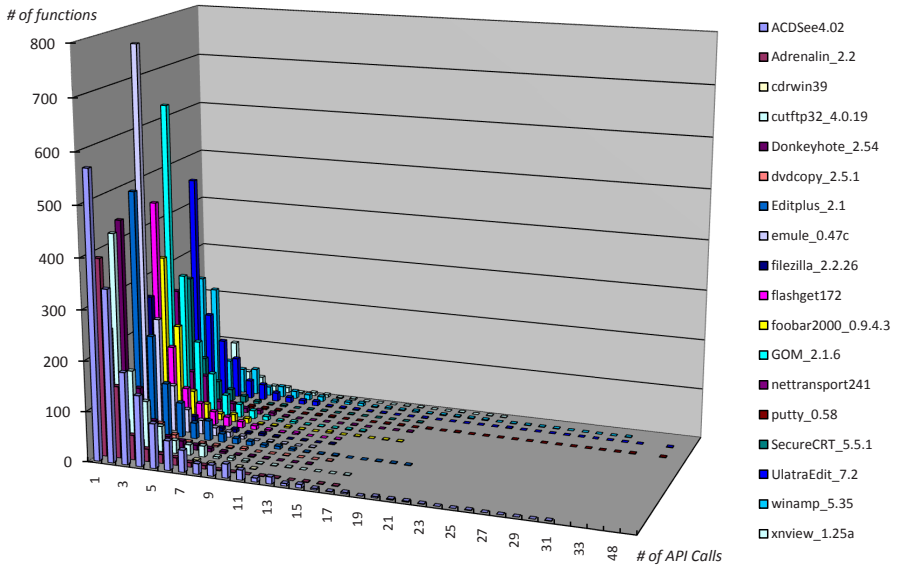

Fig. 3. API call distributions of sample programs

Table 2. Similarities with various call depth

	0	1	2	3	4	5
cutftp32_3.5.4 / cutftp32_4.0.19	0.9180	0.8993	0.8959	0.8948	0.8960	0.8969
filezilla_2.2.14 /filezilla_2.2.26	0.9058	0.8755	0.8522	0.8400	0.8340	0.8323
UltraEdit_7.0 / UltraEdit_7.2	0.9704	0.9151	0.8537	0.8464	0.8381	0.8278
filezilla_2.2.26 / cutftp32_4.0.19	0.4875	0.3865	0.3278	0.3186	0.3115	0.3091
filezilla_2.2.26 / UltraEdit_7.2	0.3574	0.3079	0.2773	0.2710	0.2702	0.2676

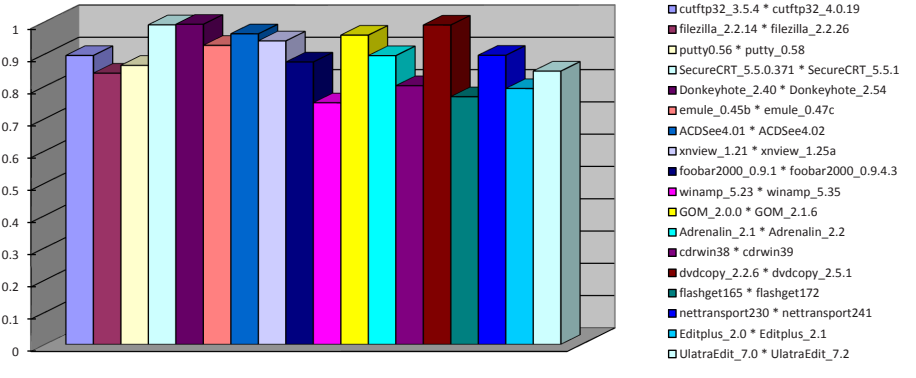

Fig. 4. Similarities between same programs with different versions

part from the previous version. Then the similarity between the old version and the new version is considered to be as large as the proportion of the common code over the whole code. The result shows that our birthmark is sufficiently reflecting the program functionalities.

Similar Category Programs. We compared programs in same categories to prove the distinction property. Even if two programs are in similar category, the similarity is not always high enough. The number of functions is different and each program uses different API calls. Suppose that there are different programs with the same functionalities. They may use almost same APIs because they have the same functionalities. The distinction property says that the similarities should be different when they are implemented independently.

Figure 5 supports that our birthmarking technique suffices distinction property. For example, the multimedia players Gom and Adrenalin have very similar functionalities, but the similarity is very low. And the text editors EditPlus and UltraEdit have almost the same functionalities, similar file sizes and similar numbers of functions extracted from the binaries. The similarity is near 0.5.

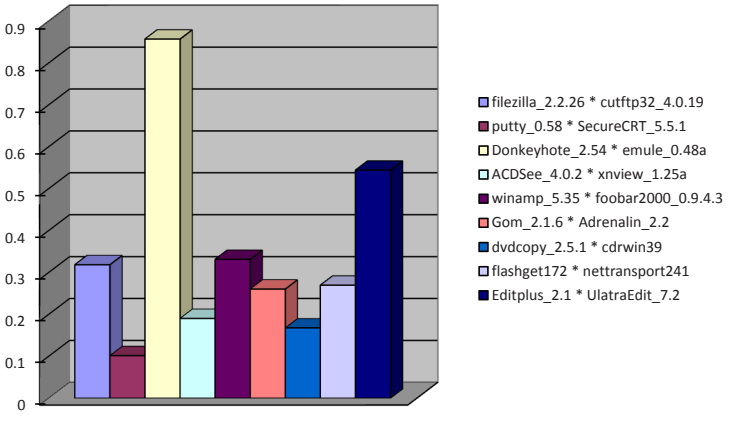

Fig. 5. Similarities between programs with the same category

It is remarkable that the similarity between two P2P programs Dongkeyhote and Emule is very high. In fact, the Dongkeyhote is a clone of Emule. It borrowed the Emule's source code and only modified GUIs.

Different Category Programs. We compared programs with different categories to show that the similarities between totally different programs are sufficiently small.

Figure 6 shows that different category programs have similarities lower than 0.4. Since the programs belong to different category have considerably lower similarities. We can observe that totally different programs have a small size of

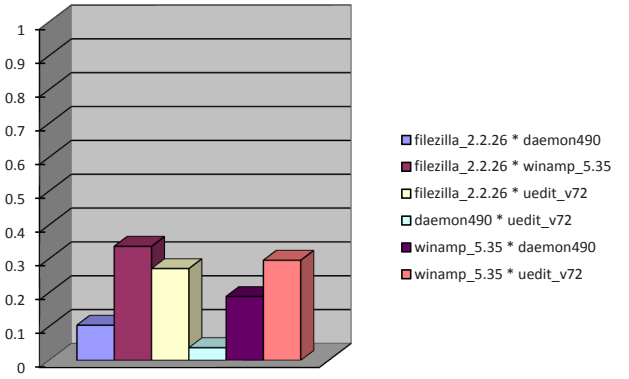

Fig. 6. Similarities between different category programs

similarity. The reason is that Windows applications should use common features like GUI, file management and networks.

5.2 Resilience

To evaluate resilience of our birthmark, we compiled an open source free hex editor, fr-hed with Microsoft Visual C++ 6.0 .NET 2003 and .NET 2005 compilers. Table 3 explains that even if compiler changes or compile option changes, used API calls are almost the same. The number of functions excluding library is different but the number of functions with API calls is nearly equal.

Table 3. Binaries compiled with various versions of compilers

Compilers and options		File size (bytes)	Number of Functions excluding Library	Number of Functions with API Calls
VC++ 6.0	Debug	409,668	441	218
	Release	317,952	479	221
VC++ .NET 2003	Debug	446,464	432	215
	Release	331,776	440	212
VC++ .NET 2005	Debug	716,800	534	218
	Release	377,344	453	215

Table 4 shows the similarity results of the resilience experiment using various compilers. Similarity is always over 0.95 in each combination. So, we concluded that our birthmark is very resilient to different compilers.

This is not enough to conclude that our birthmark is resilient to the program transformation. As far as we know, there is one available commercial C/C++ obfuscator named CloakWare security suite[22]. The CloakWare security suite

Table 4. Similarities between fr-hed binaries generated by various compilers

		VC++ 6.0		VC++ .NET 2003		VC++ .NET 2005	
		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++ .NET 2003	Debug	-	-	1.0000	0.9900	0.9793	0.9924
	Release	-	-	-	1.0000	0.9857	0.9977
VC++ .NET 2005	Debug	-	-	-	-	1.0000	0.9881
	Release	-	-	-	-	-	1.0000

applies data transformations and control transformations to the original code. The control transformation used by this tool is control-flow flattening[23] which makes static analysis of the code almost impossible. But our birthmark is resilient to the control-flow flattening, because control-flow flattening cannot remove the API calls.

5.3 Limitations

Our birthmark relies on the API call set. The birthmark of applications which rarely use the standard API calls like encoders, decoders, scientific application, etc may be very inaccurate. Birthmarks of these applications should catch the algorithmic structure of the program. If WPP birthmark [15] could be applied to binary programs, it will be a good option.

The weak link of our birthmark system is the analysis phase of the binary executables. We rely on IDAPro about function identification. Since IDAPro cannot generate precise call graphs if the binary contains function pointers and virtual calls. It is very hard to resolve virtual calls in binaries. If an unresolved indirect call like virtual calls exists in a function, the function can point to all possible functions. Then the API call set of a function may contain almost all API calls. Virtual call resolution method for binary executables suggested by Balakrishnan et al.[24] may help to improve the accuracy of our birthmark.

6 Conclusion

In this paper we proposed a novel static birthmarking technique that can help to identify ownership and similarity of binary executables. We defined the static API call birthmark of a program as the collection of the k-depth API call set. The program similarity we defined is the maximum value among all possible function matching configurations. The problem maximizing the sum of similarities between the functions from program A and the functions from program B is isomorphic to the weighted X-Y bipartite matching problem. Thus the similarity between two programs was able to be obtained by applying the Hungarian algorithm.

We evaluated the proposed birthmark by comparing various categories of Windows applications. To show the credibility, the same applications with different versions are compared. To show the resilience, we compare binary executables compiled from various compilers. The empirical result shows that the similarities obtained using our birthmark sufficiently indicates the functional and structural similarities among programs.

In the future, we are planning to extend our method by applying indirect call resolution. The proposed method could be also applied to Java class files.

References

1. Schleimer, S., Wilkerson, D., Aiken, A.: Winnowing: local algorithms for document fingerprinting. In: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pp. 76–85. ACM Press, New York (2003)
2. Wise, M.: YAP3: improved detection of similarities in computer program and other texts. In: Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education, pp. 130–134 (1996)
3. Prechelt, L., Malpohl, G., Philippsen, M.: Finding plagiarisms among a set of programs with JPlag. Journal of Universal Computer Science 8(11), 1016–1038 (2002)
4. SABRE BinDiff, `http://www.sabre-security.com/products/bindiff.html`
5. Using BinDiff for Code theft detection, `http://www.sabre-security.com/products/CodeTheft.pdf`
6. Tamada, H., Okamoto, K., Nakamura, M., Monden, A., Matsumoto, K.: Dynamic Software Birthmarks to Detect the Theft of Windows Applications. International Symposium on Future Software Technology 20(22) (2004)
7. Okamoto, K., Tamada, H., Nakamura, M., Monden, A., Matsumoto, K.: Dynamic Software Birthmarks Based on API Calls. IEICE Transactions on Information and Systems 89(8), 1751–1763 (2006)
8. The IDA Pro Disassembler and Debugger, `http://www.datarescue.com/idabase`
9. Collberg, C., Thomborson, C.: Watermarking, tamper-proofing, and obfuscation-tools for software protection. Software Engineering, IEEE Transactions on 28(8), 735–746 (2002)
10. Collberg, C., Thomborson, C.: Software watermarking: models and dynamic embeddings. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 311–324. ACM Press, New York (1999)
11. Collberg, C., Myles, G., Huntwork, A.: Sandmark-A tool for software protection research. Security & Privacy Magazine, IEEE 1(4), 40–49 (2003)
12. Tamada, H., Nakamura, M., Monden, A., Matsumoto, K.: Design and evaluation of birthmarks for detecting theft of java programs. In: Proc. IASTED International Conference on Software Engineering (IASTED SE 2004), pp. 569–575 (2004)
13. Tamada, H., Nakamura, M., Monden, A., Matsumoto, K.: Java Birthmarks–Detecting the Software Theft–. IEICE Transactions on Information and Systems 88(9), 2148–2158 (2005)
14. Myles, G., Collberg, C.: K-gram based software birthmarks. In: Proceedings of the 2005 ACM symposium on Applied computing, pp. 314–318. ACM Press, New York (2005)
15. Myles, G., Collberg, C.: Detecting software theft via whole program path birthmarks. Information Security Conference, 404–415 (2004)

16. Myles, G.M.: Software Theft Detection Through Program Identification. PhD thesis, Department of Computer Science, The University of Arizona (2006)
17. Larus, J.: Whole program paths. In: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation, pp. 259–269. ACM Press, New York (1999)
18. Bunke, H., Shearer, K.: A graph distance metric based on the maximal common subgraph. Pattern Recognition Letters 19(3-4), 255–259 (1998)
19. Schuler, D., Dallmeier, V., Lindig, C.: A Dynamic Birthmark for Java. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering
20. Kuhn, H.: The Hungarian method for the assignment problem. Naval Research Logistics 52(1), 7–21 (2005)
21. Kibria, R.: frhed - free hex editor, `http://www.codeproject.com/tools/frhed.asp`
22. Cloakware security suite, `http://www.cloakware.com/products_services/security_suite`
23. Wang, C.: A Security Architecture for Survivability Mechanisms. PhD thesis, University of Virginia
24. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. Static Analysis Symp. (2006)