Design and Evaluation of Dynamic Software Birthmarks Based on API Calls

Haruaki Tamada **Keiji Okamoto** **Masahide Nakamura** **Akito Monden**
Ken-ichi Matsumoto
Graduate School of Information Science,
Nara Institute of Science and Technology,
8916-5, Takayama, Ikoma, Nara 630-0101, Japan,
Email: {harua-t, keiji-o, masa-n, akito-m, matumoto}@is.naist.jp

Abstract

This paper presents a technique of dynamic software birthmarks to support efficient detection of software theft. A dynamic birthmark $f(p, I)$ is a set of unique and native characteristics of a program p, obtained by executing p with a given input I. For a pair of software p and q, if $f(p, I) = f(q, I)$ holds, q is suspected as a copy of p. In this paper, we propose two kinds of dynamic birthmarks, EXESEQ and EXEFREQ for the above f. In general, it is difficult for adversaries to alter API calls in the binary code automatically. Based on the fact, we extensively use runtime information of API calls as a strong signature of the program, specifically, the execution order for EXESEQ and the frequency distribution for EXEFREQ. We evaluated the proposed birthmarks through two experiments. The first experiment evaluates the preservation and distinction properties of the birthmarks with a set of the same-purpose applications. In the second experiment, we examined the impact of using different compilers. The results showed that the birthmarks of an extended-version application was very similar to that of its ancestor application, and that the birthmarks are robust enough to tolerate different compilers.

Key Words: software theft, copyright protection, software birthmark, API calls, dynamic analysis

1 Introduction

In highly competitive world of software industry, the *software theft* is an issue that often arises. Recently, many incidents have been reported, including; SCO Group's lawsuit against IBM [19], GPL infringement [5], piracy of MS-Office [18], and copyright infringement of freeware/shareware [24]. According to a recent report by Business Software Alliance [2], in 2003 more than 2.8 billion dollars worth of computer software products were copied, sold illegally, or stolen. It is therefore crucial for software companies to protect their own intellectual properties.

It is, however, not an easy task to protect software from such theft. Since a number of software products are distributed today, even *detecting suspected copies* is quite difficult, unless the product is well-known to the public. Moreover, an adversary would add a little crafty modification to the stolen code, and would insist of the ownership for the modified code, as if it was completely developed by the adversary. In such a case, detection of the theft becomes much more difficult, since it requires a manual binary code analysis to get a reasonable evidence. The binary code analysis is very expensive and needs extremely high skill.

To facilitate the detection of the theft, a concept of *software birthmark* has been proposed [7, 15, 22]. Intuitively, a (static) birthmark $f(p)$ of a programs p is the set of *native* and *unique* characteristics that p *originally* possesses. $f(p)$ is computed by applying a certain method f to p itself. A set of used classes, inheritance relationships, or types of field variables are examples of birthmarks [22]. For a pair of programs p and q, if $f(p) = f(q)$ holds, q is very likely to be a stolen copy of p (and vice versa). Unfortunately, for the static birthmark, some effective attacking methods to alter the original birthmarks have been proposed[6, 15].

To complement the weakness of the static birthmark, a technique using program's runtime information was proposed, specifically called *dynamic birthmark* [15]. For a given program p and an input I, a dynamic birthmark $f(p, I)$ is derived based on an information obtained by executing p with I. In [15], Myles et al. presented a dynamic birthmark WPP, and evaluated it with a small Java program. The WPP birthmark utilizes the structure of *program execution paths* as a unique signature of the program. However, as mentioned by the authors, the WPP birthmark is fragile to *program optimization*, including loop transformations and inlining functions. Therefore, the WPP birthmark is not good at programs for which various compilers and compiler options are available, such as Windows applications.

In this paper, we propose two new dynamic birthmarks EXESEQ and EXEFREQ, which are well applicable to Windows applications. In general, a complex program is built with the support of various *APIs* (Application Program Interfaces), to make full use of ready-made features provided by the operating system. For a program p that calls some APIs, it is difficult for adversaries to alter the API calls in p with other equivalent ones, especially low-level APIs such as file I/O, semaphore and GUIs. Also, since the APIs are already built in the OS, the compiler does not optimize the APIs themselves. Therefore, we consider that the API calls can be used as a robust signature to characterize the program. In the proposed method, for a given program p and an input I, we execute p with I, and measure the *ordered sequence* and *frequency* of API calls during runtime. Then, we use the ordered sequence and the frequency as birthmarks $EXESEQ(p, I)$ and $EXEFREQ(p, I)$, respectively.

The proposed birthmarks are evaluated through two experiments. The first experiment evaluates the *preservation* and *distinction properties* for a set of same-purpose applications (Windows MP3 tag editors). As a result, the proposed birthmarks were able to distinguish well different applications independently developed. Also, it was shown that an application and its extended version have quite similar birthmarks. In the second experiment, we evaluate how the proposed birthmarks can tolerate the program optimization, by employing *different commercial compilers* varying their compile options. In general, compiling the same program with debug option and release option yields different executable with different instruction sequences. An adversary who could obtain source code might exploit this phenomenon. However, the proposed birthmarks were shown to be well preserved even if different compilers and options were used.

The preliminary idea of the proposed birthmarks was originally published in a workshop paper [23]. Changes were made for this version, most significantly the addition of the quantitative evaluation with the practical MS-Windows applications and commercial compilers. These new results clarify the applicability and limitations of the proposed birthmarks against realistic situation of software theft.

2 Related Work

Software watermarking (often called *software fingerprinting*) is a well-known technique used to provide a way to prove ownership of stolen software [4, 14]. Unfortunately, watermarking is not always feasible for our objective. A watermark is basically an extra information for the program. Therefore, it requires software developers to embed a watermark *before* releasing the software. Thus, proofs cannot be given for already-released software without watermarks.

Software similarity computation is also a technique, which is commonly used for plagiarism detection in programming classes. Various methods for similarity computation have been proposed based on attribute counting [17], structure metrics [1], Kolmogorov complexity [3], and code clones [10]. Unfortunately, since these methods require the *source code* of software, they are not applicable in our problem setting where software products are usually distributed *without* source code.

There is also a technique called *authorship analysis* [11]. It tries to identify the author (as a programmer) from different programs by focusing on the programming style and program layouts peculiar to the programmer. Although the objective is different from ours, the term *(program) birthmark* was originally used by Grover [7] to refer such peculiarity.

3 Definitions

3.1 Copy Relation

We start with formulation what it means for a program q to be a *copy* of another program p.

Definition 1 (Copy Relation) Let $Prog$ be a set of given programs. Let $\equiv_{cp}$ denote an equivalent relation over $Prog$ such that: for $p, q \in Prog$, $p \equiv_{cp} q$ holds iff q is a *copy* of p (vice versa). The relation $\equiv_{cp}$ is called a *copy relation*.

The criteria for whether or not q is a *copy* of p can vary depending on the context. For example, each of the following criterion is relatively reasonable for general computer programs:

(a) q is an exact duplication of p,

(b) q is obtained from p by renaming all identifiers in the source code of p, or

(c) q is obtained from p by eliminating all the comment lines in the source code of p.

To avoid confusion, we suppose that $\equiv_{cp}$ is *originally given* by the user. Since $\equiv_{cp}$ is an equivalent relation, the following proposition holds.

Proposition 1 *For $p, q \in Prog$, the following properties hold. (Reflexive) $p \equiv_{cp} p$, (Symmetric) $p \equiv_{cp} q \Rightarrow q \equiv_{cp} p$, (Transitive) $p \equiv_{cp} q \wedge q \equiv_{cp} r \Rightarrow p \equiv_{cp} r$.*

All the above properties meet well the intuition of copy. Next, if q is a copy of p, the external behavior of q should be identical to p's.

Proposition 2 *Let $Spec(p)$ be a (external) specification conformed by p. Then, the following property holds: $p \equiv_{cp} q \Rightarrow Spec(p) = Spec(q)$.*

Note that the inverse of this proposition does not necessarily hold since we can see, in general, different program implementations conforming the same specification.

3.2 Software Birthmarks

Given $Prog$ and $\equiv_{cp}$, we define the concept of *dynamic birthmark* of a program. The following definition is a restatement of the definition by Myles et al. [15] [1].

Definition 2 (Dynamic Birthmark) Let $p, q \in Prog$ be programs, I be a given input and $\equiv_{cp}$ be a given copy relation. Let $f(p, I)$ be a set of characteristics extracted from p by executing p with I. $f(p, I)$ is called a *dynamic birthmark* of p under $\equiv_{cp}$ iff both of the following conditions are satisfied:

1. $f(p, I)$ is obtained only from p itself by executing p with I, and
2. $p \equiv_{cp} q \Rightarrow f(p, I) = f(q, I)$

Condition 1 means that the birthmark is not extra information and is required for p to run. Hence, extracting a birthmark does not require extra code as watermarking does. Condition 2 states that the same birthmark has to be obtained from copied programs. By contraposition, if birthmarks $f(p, I)$ and $f(q, I)$ are different, then $p \not\equiv_{cp} q$ holds. That is, we can guarantee that q is not a copy of p. Hopefully a birthmark should satisfy the following properties.

Property 1 (Preservation) *For p' obtained from p by any program transformation, $f(p, I) = f(p', I)$ holds.*

Property 2 (Distinction) *For p and q such that $Spec(p) = Spec(q)$, if p and q are written independently, then $f(p, I) \neq f(q, I)$.*

Property 1 specifies the *preservation property* of the birthmark against program transformation. We consider that clever crackers may try to modify birthmarks by transforming the original program into an semantically-equivalent one to hide the fact of theft. Property 1 specifies that the same birthmark must be obtained from p and converted p'.

Property 2 specifies the *distinction property* of the birthmark, stating that: even though the specification of p and q is the same, if implemented separately, different birthmarks should be extracted. In general, the detail of two independent programs is almost never completely the same. However, in the case that p and q are both *tiny* programs, extracted birthmarks could become the same, even if p and q are written independently.

Those properties should be satisfied ideally. However, there exist many ways of transformation and implementation of a program. Therefore, in reality, it is difficult to extract a birthmark that perfectly achieves both Properties 1 and 2. Those properties should be tuned within an allowable range at the user's discretion. Now, the the question is how to develop an effective method f for a set $Prog$ of programs and the copy relation $\equiv_{cp}$.

4 New Dynamic Birthmarks Based on API Calls

4.1 Key Idea

As mentioned before, applications running on an operating system (OS) can use various build-in features of the OS by calling *APIs*. The typical API function calls include the file input/output, synchronized objects such as semaphore, mutex and critical section and graphic user interface (GUI). For API calls involved in a program p (in a binary form), we focus on the following properties.

- It is hardly possible to replace the API calls with other instructions without changing the behavior of p.
- In general, a compiler does not optimize the APIs themselves.

As for the first property, this paper assumes a relatively complex program (application) operating on the recent sophisticated operating systems such as MS Windows, where every access to system resources is strictly managed via APIs. In such OS, any operation to the file system, for instance, must be done via file I/O APIs. The operations to GUIs (widgets) must be also performed by API calls. Hence, it is almost impossible to alter these API calls with other user-made instructions.

These properties motivated us to use the history (i.e., execution log) of API calls for robust dynamic birthmarks. In the following, we propose two new dynamic birthmarks: EXESEQ and EXEFREQ.

4.2 EXESEQ: Sequence of API Function Calls Birthmark

For a given program p and an input I, executing p with I yields an ordered sequence of API calls. It is difficult

[1] The definition of static birthmark is originally given in our previous research [22]. The definitions of the static and dynamic birthmarks are almost the same, except the static birthmark is independent of the input I.

for an adversary to hack p and scramble the order of API calls completely without changing the original behavior of p. Also, it is a rare case that two different programs p and q have the same order of API calls, even if p and q are using the same set of APIs. Hence, we use the ordered sequence of API calls as a dynamic birthmarks. In practice, it is sufficient to pay our attention to only a certain set W of APIs, which is defined as follows.

Definition 3 (EXESEQ) Let p be a given program, I be a given input, and W be a given set of API functions. Let $w_1, w_2, ..., w_n$ be a sequence of API calls executed by p with I in this order. If $w_i (1 \leq i \leq n)$ does not belong to W, we eliminate it from the sequence. Then, the resultant sequence $(w_1, w_2, ..., w_m)$ is called an *EXESEQ birthmark* of p with I, denoted by *EXESEQ(p, I)*.

4.3 EXEFREQ: Frequency of API Function Calls Birthmark

The frequency distribution in the number of executed API calls also provides a unique and robust characteristic of a program. Even if an adversary succeeds in changing the order of some APIs in *EXESEQ(p, I)*, the total number of calls for each of the APIs could still remain the same. Hence, we use the frequency distribution as the second birthmark EXEFREQ.

Definition 4 (EXEFREQ) Let p be a given program, I be a given input, and $(w_1, w_2, ..., w_n)$ be *EXESEQ(p, I)*. Let K be a set of all *API names* appearing in *EXESEQ(p, I)*. Then, let $k_1, k_2, ..., k_m$ be a sequence of API names obtained by arranging all elements in K in a certain (fixed) order. For k_i $(1 \leq i \leq m)$, let a_i be the number of appearances of k_i in *EXESEQ(p, I)*. Then, the sequence $((k_1, a_1), (k_2, a_2), ..., (k_m, a_m))$ is called an *EXEFREQ birthmark* of p, denoted by *EXEFREQ(p, I)*.

4.4 Similarity of Birthmarks

Each of the proposed birthmarks is in a form of a sequence. Suppose that we have a pair of birthmarks $f(p, I) = (p_1, p_2, ..., p_n)$ and $f(q, I) = (q_1, q_2, ..., q_n)$ for two programs p and q. Basically, we say that $f(p, I)$ is *equal to* $f(q, I)$, denoted by $f(p, I) = f(q, I)$, iff $p_i = q_i$ for all i. In other word, even if only a single pair of p_i and q_i is different and other pairs are the same, $f(p, I) \neq f(q, I)$. Then, we have to conclude that p is not a copy of q, although the both birthmarks are quite *similar* to each other. Thus, comparing the birthmarks by the equality only makes the method too sensitive.

To cope with the problem, we introduce the *similarity* of the proposed birthmarks. Intuitively, the similarity between a pair of EXESEQ birthmarks is defined as the ratio of *matched sequence* commonly contained in both birthmarks. On the other hand, the similarity between a pair of EXEFREQ birthmarks is defined as an *vector angle*, by regarding the birthmarks as vectors.

Definition 5 (Similarity of EXESEQ) Let p and q be given programs, I be a given input. Suppose that $EXESEQ(p, I) = \rho_p = (p_1, p_2, ..., p_m)$ and $EXESEQ(q, I) = \rho_q = (q_1, q_2, ..., q_n)$ are derived from p and q, respectively. Then, a sequence $\rho = (r_1, r_2, ..., r_k)$ is called a *match* among ρ_p and ρ_q, iff both ρ_p and ρ_q commonly contain ρ as a sub-sequence, i.e., there exist i and j such that $r_1 = p_i = q_j$, $r_2 = p_{i+1} = q_{j+1}$, ..., $r_k = p_{i+k} = q_{i+k}$. A match ρ is called the *longest match* iff there is no other match longer than ρ. Let $\rho = (r_1, r_2, ..., r_k)$ be the longest match among $EXESEQ(p, I)$ and $EXESEQ(q, I)$. Then, the similarity of the two birthmarks is defined by $2k/(m + n)$, which is the ratio of the longest match in the total of $EXESEQ(p, I)$ and $EXESEQ(q, I)$. Since $1 \leq k \leq m$, $1 \leq k \leq n$, the similarity varies within the range of 0.0 to 1.0.

Definition 6 (Similarity of EXEFREQ) Let p and q be given programs, I be a given input. Suppose that $EXEFREQ(p, I) = ((k_1, p_1), (k_2, p_2), ..., (k_n, p_n))$ and $EXEFREQ(q, I) = ((k_1, q_1), (k_2, q_2), ..., (k_n, q_n))$ are derived from p and q, respectively. Then, let $\vec{v_p} = [p_1, p_2, ..., p_n]$ and $\vec{v_q} = [q_1, q_2, ..., q_n]$ be two vectors obtained from $EXEFREQ(p, I)$ and $EXEFREQ(q, I)$. Then, the similarity of the two birthmarks is defined by:

$$\frac{p_1 q_1 + p_2 q_2 + ... + p_n q_n}{\sqrt{p_1^2 + p_2^2 + ... + p_n^2}\sqrt{q_1^2 + q_2^2 + ... + q_n^2}}$$

which is the *cosine angle* of $\vec{v_p}$ and $\vec{v_q}$. The range of the similarity is from 0.0 to 1.0, since all p_i and q_i are positive integers.

5 Experimental Evaluation

5.1 Preliminaries

For the experiment, we have implemented a software suite — *K2 Birthmark Toolkit* (`K2BTk`), which extracts and evaluates the proposed two dynamic birthmarks for any MS Windows applications (executables). `K2BTk` mainly consists of two modules: *WinAPI capture* (`wapicapture`) and *birthmark evaluator* (`bmeval`).

`wapicapture` is a module to *log* every observable API call during runtime for a Windows application. When the OS loads an application p on the memory, `wapicapture` rewrites the *import section* (i.e., a table for function pointers) of p, so that every call of an API c is redirected to a wrapper function. The wrapper function outputs a log for c to a file, and then invokes the original API c. During the execution of p, `wapicapture` acts as a *parasite* to monitor and log the API calls. To achieve this, `wapicapture` employs a technique of *Windows Hook* [20]. `bmeval` extracts EXESEQ and EXEFREQ from the log obtained by `wapicapture`, then evaluates the similarity among the derived birthmarks.

Using `K2BTk`, we conduct two experiments to show the effectiveness of the proposed birthmarks. As a set of APIs to be used as the birthmark computation (i.e, the set W, see Definition 3), we have selected 604 APIs from `winbase.h` of Windows SDK, which is a header file of Windows' standard APIs. Indeed, `winbase.h` defines total 613 API functions. However, we omit the following 8 APIs since for any application, they are called so many times that the execution of `bmeval` does not terminate in a reasonable time: `EnterCriticalSection`, `LeaveCriticalSection`, `TLSGetValue`, `RestoreLastError`, `LocalLock`, `LocalUnlock`, `HeapAlloc`, and `HeapFree`. Moreover, `WriteProfileStringA` is also omitted, since it is used in `wapicapture`.

5.2 Experiment 1: Preservation and Distinction Performance

In this experiment, we evaluate the preservation and distinction properties (see Section 3.2) of the proposed birthmarks. We employ a set of the *same-purpose* applications, and evaluate the similarity among the applications by the proposed birthmarks.

We take the following five applications: Super Tag Editor 2.00b7 (`STE`) [13], Super Tag Editor extended (`STE-ext`) [8], Tsuyutagu 2.02 (`Tsuyutagu`) [21], TeaTime 2.525 (`Teatime`) [12], and Mp3Tag 2.2.6.0 (`MP3Tag`) [9]. All of them are an MP3 tag editor, which allows the user to edit information in MP3 music files.

Among the five applications, `STE-ext` is an extended version of `STE`, where the author of `STE-ext` *legally* took over the original `STE` project and added various features. Thus, it is expected that `STE` and `STE-ext` are quite *similar* to each other. [2] Other three applications are developed independently, so the birthmarks should distinguish them.

For just comparison, we first see the similarity with respect to the *difference* among executable files. For this, we use a tool `WDIFF` [16], which can create a *patch* among two

[2]More practically, we should have taken an example from actual incidents of theft, but we could not obtain any products concerning the incidents. However, we can *simulate* the scenario of theft, if we ignore the fact that the author of `STE` legally agrees with the one of `STE-ext` for the reuse of `STE`.

Table 1. Difference on executable files

Product name	File size	Patch size	%
STE 2.00b7	565,248	—	—
STE-Ext	966,656	706,189	73.055%
Tsuyutagu	956,928	903,345	94.400%
TeaTime	1,006,080	954,839	94.907%
Mp3tag	1,810,432	1,626,527	89.841%

Table 2. The result of Experiment 1

	EXESEQ			**EXEFREQ**
	# of calls	**len. of match**	**similarity**	**similarity**
STE 2.00b7	13,286	—	—	—
STE-Ext	13,447	12,781	0.9562	0.9997
Tsuyutagu	10,834	129	0.0107	0.3855
TeaTime	1,647	70	0.0094	0.2891
Mp3Tag	191,783	189	0.0018	0.181

applications as a binary code difference (like UNIX's `diff` for the source code difference). The result is shown in Table 1. The first column shows the file size of each product. The second column represents the patch size created by `WDIFF`, which reflects the degree of the binary-level difference from `STE`. The *% (percentage)* column represents the percentage of patch size over file size. From this result, we cannot say anything but "`STE` is quite different from the other four applications".

Next, we evaluate the similarities by the proposed dynamic birthmarks. As the input to each application, we operate a simple scenario; launch the application and then immediately quit. The birthmarks are computed from API calls captured during the scenario.

Table 2 summarizes the result. First, we see the result of EXESEQ. The column *# of calls* represents a number of API calls found in the EXESEQ birthmark, which is equal to the length of EXESEQ. The column *len. of match* shows the length of the longest match among EXESEQ birthmarks of `STE` and each of other four applications. An fragment of a match computed by `bmeval` is shown in Figure 1. As seen in the column *similarity* in Table 2, `STE-ext` has a quite similar birthmark to that of `STE`, compared to the other three products.

Next, we examine the result of EXEFREQ. Figure 2 shows frequency distribution diagrams obtained from the EXEFREQ birthmarks, where the X-axis represents the APIs ordered by a sequence number, Y-axis depicts the frequency (i.e., the number of calls) for each API, and Z-axis enumerates the five applications. From the figure, it can be seen that the graph shapes of `STE` and `STE-ext` are quite similar, which achieves 0.9997 of the similarity. On the other hand, other three applications are not similar to `STE` at all.

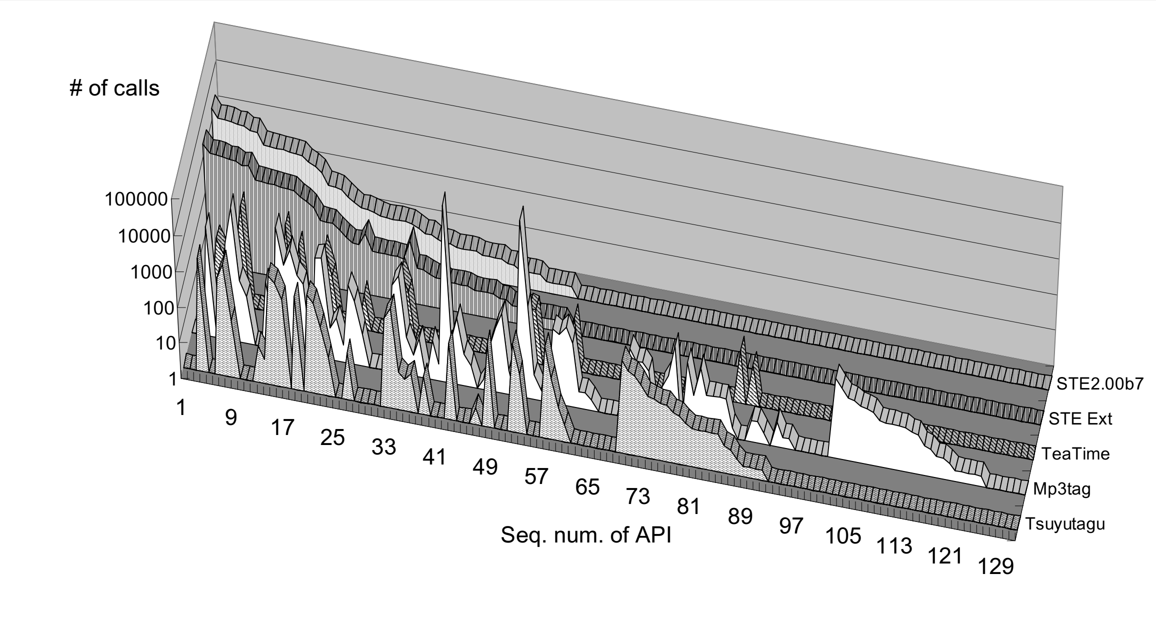

Figure 2. Frequency distribution of EXEFREQ

Thus, in this experiment, it was shown that the proposed birthmarks achieved a high preservation property for related programs (`STE` and `STE-ext`), and a good distinction property for independently developed applications. This implies that by using the proposed birthmarks, we can identify the copy-prone products from a set of the same-purpose applications quite efficiently in a quite practical setting.

5.3 Experiment 2: Impact on Different Compilers and Compile Options

In general, for a source code s, if we use different compiler options to compile s, different binary codes are generated. For example, a binary code b_d compiled from s with the *debug option* contains debug codes and information (like symbol table). On the other hand, a binary code b_r compiled from s with the *release option* is optimized for fast and efficient execution. Hence, executing b_d and b_r may yield different API calls although both are generated from the same source s, which might be used as an attack to alter and scramble the proposed birthmarks. Also, the know-how of compilers and optimization techniques (e.g., loop transformation, inlining functions) widely vary among the vendors and the release versions of the compilers. Therefore, different compilers surely generate different binary codes from the same source code. If an adversary succeeds in obtaining the source code s of a product b, he would compile s with the different compiler or different option to eliminate the birthmark of b.

Table 3. Size of STE executables

Compiler	Option	File size
Intel C++ 8.1.024	Debug	2,281,472
	Release	913,408
Visual C++ 6.0	Debug	1,699,960
	Release	581,632
Visual C++ 7.0	Debug	2,113,536
	Release	577,536

Our interest here is to evaluate the resilience of the proposed birthmarks against the program transformation with the different compilers. For the experiment, we chose the following commercial compilers: Intel C++ 8.1.024_ev05 (evaluation version), Microsoft Visual C++ 6.0 Professional Edition SP6, and Microsoft Visual C++ 7.0. As the target product, we selected Super Tag Editor 2.01 (`STE`).

Using each compiler, we compiled the source code of `STE` with the debug and release options, where the debug option is to use the default option of the compiler whereas the release option is to use the strongest optimization level. Table 3 shows the size of the resultant `STE` executables, where we can see the significant difference.

Now, we evaluate the similarity among these `STE` executables by the proposed birthmarks. In the birthmark computation, we excluded the following four from the set W of APIs to be monitored: `lsBadReadPtr`, `lsBadWritePtr`, `HeapValidate`,

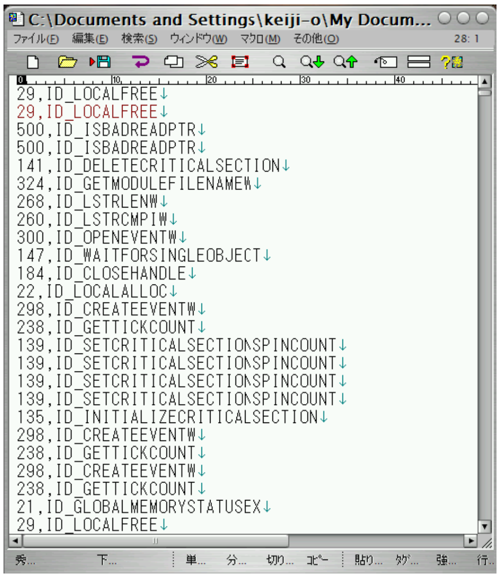

Figure 1. A match among EXESEQ(STE, I) and EXESEQ(STE-Ext, I)

and `WaitForMultipleObjectsEx`. These four APIs are so sensitive to the current status of the OS (memory usage, time, running processes) that they vary even when executing the same executable in different timing. Hence, these four APIs are omitted to *normalize* the birthmark similarity between the same executable. The scenario of the execution is the same as the one used in Experiment 1.

Tables 4 and 5 summarize the similarity of EXESEQ and EXEFREQ birthmarks, respectively, for every pair of the `STE` executables. We can see in Table 4 that every pair of the executables achieves a high similarity over 0.9. Especially, the release builds are quite similar to each other, where the similarity is around 0.97 or more. The subtle difference on the EXESEQ birthmarks among the release builds is completely eliminated by EXEFREQ birthmarks, as shown in 5. This implies that small changes in the order of API calls may give an impact on the EXESEQ birthmark, but it can be recovered by the EXEFREQ birthmark. For the EXEFREQ birthmarks, the similarity between every pair of executables is more than 0.99.

In summary, it was shown in this experiment that the proposed birthmark is quite robust against the attack with different compilers and compiler options. This implies that the proposed birthmarks have quite strong tolerance against the automatic program optimization.

6 Conclusion

In this paper, we have proposed dynamic birthmarks to provide reasonable evidence of theft. First, we formulated the dynamic birthmark of programs, and then presented two types of birthmarks: EXESEQ birthmark and EXEFREQ birthmark.

The proposed birthmarks were thoroughly evaluated by two experiments. The evaluation was conducted from the viewpoints of birthmark properties and compiler-specific properties. The result showed that proposed birthmarks have a good performance in practical use, and sufficient native information of applications which cannot be erased by different compilers.

We are currently conducting a deeper security analysis of the proposed birthmarks through more experiments. Investigation of other types of dynamic birthmarks is also an interesting problem for our future work.

References

[1] Alex Aiken. MOSS: A system for detecting software plagiarism, Jun 2004. http://www.cs.berkeley.edu/~aiken/moss.html.

[2] Business Software Alliance. Global software piracy study, Jun 2004. http://www.bsa.org/globalstudy/.

[3] Xin Chen, Brent Francia, Ming Li, Brian Mckinnon, and Amit Seker. SID plagiarism detection, Dec 2003. http://genome.math.uwaterloo.ca/SID/.

[4] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Proc. of Principles of Programming Languages 1999, POPL'99*, pages 311–324, Jan 1999. San Antonio, TX.

[5] Epson pulls linux software following gpl violations (slashdot.org), Sep 2002. http://slashdot.org/article.pl?sid=02/09/11/2225212.

[6] Kazuhide Fukushima, Toshihiro Tabata, and Kouichi Sakurai. Program birthmark scheme with tolerance to equivalent conversion of java classfiles. In *IPSJ SIG Notes*, volume 2003, pages 81–86, Dec 2003. (in Japanese).

[7] Derrick Grover, editor. *The protection of computer software – its technology and applications Second edition*. The British Computer Society Monographs in Informatics Cambridge University Press, May 1992.

[8] haseta. Super tag editor extended, Feb 2004. http://hp.vector.co.jp/authors/VA012911/mp3DB/ste.html.

[9] Florian Heidenreich. Mp3tag - the universal tag editor and more, Nov 2004. http://www.mp3tag.de/en/index.html.

Table 4. Similarities of EXESEQ in Experiment 2

		Intel C++ 8.1.024		Visual C++ 6.0		Visual C++ 7.0	
		Debug	Release	Debug	Release	Debug	Release
Intel C++ 8.1.024	Debug	1.0000	0.9492	0.9243	0.9492	0.9988	0.9492
	Release	—	1.0000	0.9393	0.9698	0.9492	0.9996
Visual C++ 6.0	Debug	—	—	1.0000	0.9393	0.9243	0.9393
	Release	—	—	—	1.0000	0.9492	0.9699
Visual C++ 7.0	Debug	—	—	—	—	1.0000	0.9492
	Release	—	—	—	—	—	1.0000

Table 5. Similarities of EXEFREQ in Experiment 2

		Intel C++ 8.1.024		Visual C++ 6.0		Visual C++ 7.0	
		Debug	Release	Debug	Release	Debug	Release
Intel C++ 8.1.024	Debug	1.0000	0.9963	0.9966	0.9963	1.0000	0.9963
	Release	—	1.0000	0.9912	1.0000	0.9963	1.0000
Visual C++ 6.0	Debug	—	—	1.0000	0.9912	0.9966	0.9912
	Release	—	—	—	1.0000	0.9963	1.0000
Visual C++ 7.0	Debug	—	—	—	—	1.0000	0.9963
	Release	—	—	—	—	—	1.0000

[10] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Trans. on Software Engineering*, 28(7):654–670, 2002.

[11] Ivan Krsul and Eugene H. Spafford. Authorship analysis: identifying the author of a program. *Computers and Security*, 16(3):233–257, 1997.

[12] Toru Kuroda. Teatime, Jul 2002. http://hp.vector.co.jp/authors/VA011396/.

[13] Mercury. Super tag editor, Dec 2001. http://www5.wisnet.ne.jp/~mercury/supertag/index.html.

[14] Akito Monden, Hajimu Iida, Kenichi Matsumoto, Katsuro Inoue, and Koji Torii. A practical method for watermarking java programs. In *Proc. of COMPSAC 2000, 24th Computer Software and Applications Conference*, pages 191–197, 2000.

[15] Ginger Myles and Christian Collberg. Detecting software theft via whole program path birthmarks. In *Proc. Information Security 7th International Conference, ISC 2004*, volume 3225, pages 404–415. Springer-Verlag GmbH, Sep 2004. Palo Alto, CA, USA.

[16] Tetsuhiro Nakagawa. Wdiff, May 1998. http://www.vector.co.jp/soft/win95/util/se057654.html.

[17] Karl J. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *SIGCSE Bulletin*, 8(4):30–41, 1976.

[18] Andy Patrizio. Pirates experience Office XP (wired news), Mar 2001. http://www.wired.com/news/business/0,1367,42402,00.html.

[19] Eric Raymond and Rob Landley. OSI position paper on the sco-vs.-ibm complaint, May 2004. http://www.opensource.org/sco-vs-ibm.html.

[20] Jeffrey Richter. *Programming Applications for Microsoft Windows (Microsoft Programming Series)*. Microsoft Press, 1999.

[21] P's soft. Mpeg id3 tag editor tsuyutagu, Mar 2004. http://www.lares.dti.ne.jp/~mk3/Akmsoft/tuyutag.htm.

[22] Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken ichi Matsumoto. Design and evaluation of birthmarks for detecting theft of java programs. In *Proc. of IASTED International Conference on Software Engineering (IASTED SE 2004)*, pages 569–575, Feb 2004. Innsbruck, Austria.

[23] Haruaki Tamada, Keiji Okamoto, Masahide Nakamura, Akito Monden, and Ken ichi Matsumoto. Dynamic software birthmarks to detect the theft of windows applications. In *International Symposium on Future Software Technology 2004 (ISFST 2004)*, Oct 2004. Xi'an, China.

[24] Tomohiro Ueno. The protest page to pocketmascot, Sep 2001. http://members.jcom.home.ne.jp/tomohiro-ueno/About_PocketMascot/About_PocketMascot_e.html.