A Chaos-Based Robust Software Watermarking

Fenlin Liu, Bin Lu, and Xiangyang Luo

Information Engineering Institute, Information Engineering Univercity,
Zhengzhou Henan Province, 450002, China
liufenlin@sina.vip.com, stoneclever@gmail.com, xiangyangluo@126.com

Abstract. In this paper we propose a robust software watermarking based on chaos against several limitations of existing software watermarking. The algorithm combines the anti-reverse engineering technique, chaotic system and the idea of Easter Egg software watermarks. The global protection for the program is provided by dispersing watermark over the whole code of the program with chaotic dispersion coding; the resistance against reverse engineering is improved by using the anti-reverse engineering technique. In the paper, we implement the scheme in the Intel i386 architecture and the Windows operating system, and analyze the robustness and the performance degradation of watermarked program. Analysis indicates that the algorithm resists various types of semantics-preserving transformation attacks and is good tolerance for reverse engineering attacks.

1 Introduction

Software piracy has received an increasing amount of interest from the research community [1, 2, 3]. Nowadays, software developers are mainly responsible for the copyright protection with encryption, license number, key file, dongle etc. [1, 4]. These techniques are vulnerable suffered from crack attacks and hard to carry out pirate tracing. Moreover, software developers have to spend much time, resources and efforts for copyright protection. If there is a reliable system of software protection as the cryptosystem, the software based on the system can be protected to a certain extent. And software developers could devote most of their resources and efforts to developing the software without spending resource and efforts on intellectual property protection. Software watermarking is just an aspiring attempt in the aspect [5].

There are several published techniques for software watermarking. However, no single watermarking algorithm has emerged that is effective against all existing and known attacks. Davidson et al. [6] involved statically encoding the watermark in the ordering of basic blocks that constitute program. It is easily subverted by permuting the order of the blocks. A comparable spread spectrum technique was introduced by Stern et al. [7] for embedding a watermark by modifying the frequencies of instructions in the program. This scheme is robust to various types of signal processing. However, the data-rate is low and the scheme is easily subverted by inserting redundant instructions, code optimization, etc. With the pointer aliasing effects, Collberg et al. [8] first proposed

K. Chen et al. (Eds.): ISPEC 2006, LNCS 3903, pp. 355–366, 2006.

dynamic software watermarking, which embeds the watermark in the topology of a data structure that is built on the heap at runtime given some secret input sequence to the program. This scheme is vulnerable to any attack that is able to modify the pointer topology of the program's fundamental data types. Cousot et al. [9] embed watermark in the local variable, and the watermark could be detected even if only part of the watermarked program is present. This scheme can be attacked by obfuscating the program such that the local variables representing the watermark cannot be located or such that the abstract interpreter cannot determine what values are assigned to those local variables. Nagara et al. [5] proposed thread-based watermarking with the premise that multithreaded programs are inherently more difficult to analyze and the difficulty of analysis increases with the number of threads that are "live" concurrently. But the scheme need introducing a number of threads, and degradation of the performance could not be ignorable. In general, there are such limitations as follows: (A) the assumed threat-model is almost based on automated attacks (i.e. code optimization, obfuscation, reconstructed data and so on), but hardly on manual attacks (such as reverse engineering attacks). (B) Watermark is just embedded in a certain module of the program so that not all modules can be protected, and it can't resist cropping attacks. (C) Watermark is embedded in the source code; because of its recompiling, the efficiency of embedding is rather low, especially fingerprint. (D) In the embedding procedure, programmers have to take on all the work, especially the complex watermark constructing and embedding, such that the watermark is not always feasible.

This paper designs a new scheme that integrates chaotic system, anti-reverse engineering technology, and the idea of Easter Egg software watermarks-*Chaos-based Robust Software Watermarking* (CRSW) which holds the facility and feasibility of Easter Egg software watermarks, meanwhile resists various types of semantics-preserving code transformation attacks. When chaotic system is involved, dispersing watermark over the whole code provides global protection for the program. Furthermore, by involving anti-reverse engineering techniques, the resistance against anti-reverse engineering attacks is improved. In addition, CRSW embeds watermark into the executable code directly. The watermarked program need not recompile, and the efficiency is improved. The analysis of the proposed algorithm shows that CRSW resists various types of semantics transformation, is good tolerance against anti-reverse engineering technology attacks, and has modest performance degradation.

2 The Structure of CRSW

Easter Egg watermarks, a kind of dynamic software watermarks, is one of the most widely used watermarking [1, 8]. This watermarking, in essence, directly embeds a watermarking detector (or extractor) into the program. When the special input sequence is received, detector (or extractor) is activated, and then the watermark which is extracted from the watermarked program is displayed in a way of visualization. Thus, the semantics of detecting procedure (or extracting procedure) is included in the semantics of the watermarked program, so that Easter Egg water-

marks resist various semantics-preserving transformation attacks [10]. The main problem with Easter Egg watermarks is that once the right input sequence has been discovered, standard debugging techniques will allow the adversary to locate the watermark in the executable code and then remove or disable it [8]. And then, the watermark is just embedded in one piece of the program typically, hence, cropping a particularly valuable module from the program for illegal reuse is likely to be a successful attack [10].

In this paper, basing on the idea of Easter Egg software watermarks, we attempt to propose a more robust and feasible software watermark—CRSW. The watermark is consisted with 4 essential parts: *watermark* W,*Input Monitoring Module* C_m, *Watermark Decoding Module* C_d, and *Anti-reverse Engineering Module* C_a.Unlike the other watermarking, CRSW not only embeds watermark W into the program, but also embeds C_m, C_d, C_a in the form of executable code into the program. In this section, we expatiate the structure and the interrelations of CRSW embedding code which includes C_m, C_d, C_a(see Fig. 1).

Formally, let P be the considered program, $\{\alpha_1, \alpha_2, \cdots\}$be the set of acceptable input of the program,$P' = T(P, W, C_m, C_d, C_a)$ is the watermarked program (T is the watermarking transformation), the extracting procedure is $\widetilde{W} = D(\Gamma(P'))$, where D is the extracting transformation, Γ is the code transformation. If the watermarked program has been attacked, Γ represents the attacking transformation. Otherwise, Γ is identical transformation.If $D(\Gamma(P')) \equiv_{cp} W$ holds,T resist Γ, where $equiv_{cp}$ is user-defined equal relationship.

Input Monitoring Module realizes the mapping, $\Psi : \{\alpha_1, \alpha_2, \cdots\} \longmapsto \{0, 1\}$. If $\Psi(\alpha_i) = 1$ holds, C_d is activated. $\alpha \in \Sigma = \{\alpha_i \mid \Psi(\alpha_i) = 1\}$ is defined as *activation key.*

To describe the *Watermark Decoding Module* clearly, we briefly describe the watermark embedding procedure. Firstly, preprocess W: $W' = E(W, G)$, where G is digital chaotic system; then embed W' into the code of program with *chaotic dispersion coding* and get the code $I_{ew} = \Omega(W', I, G)$, where I is the code of program (we will discuss *chaotic dispersion coding* in the next section). *Watermark Decoding Module* extracts the watermark $\widetilde{W}$ from watermarked program and performs it in the form of visual action. The module consists of *Watermark Output Module*(C_{do}) and *Chaotic system Module* (C_{dc}). In the extracting pro-

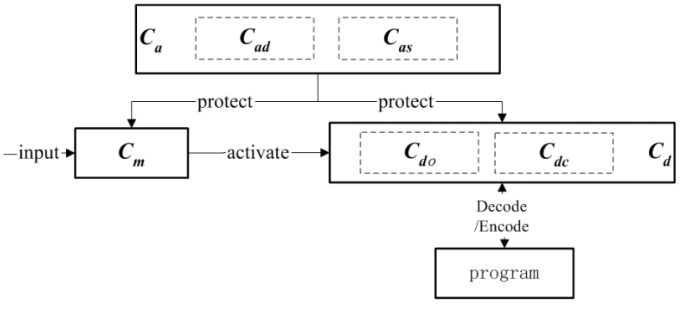

Fig. 1. The structure and interrelations of CRSW embedding code

cedure, firstly, the module extracts $\widetilde{W'}$ with *reverse chaotic dispersion coding*($\widetilde{W'} = \Omega^{-1}(I_{ew}, G)$); then gets the watermark $\widetilde{W}$ by $\widetilde{W} = E^{-1}(\widetilde{W'}, G)$; at last C_{do} transform $\widetilde{W}$ into the visual action, $V_{\widetilde{W}}$, and display $V_{\widetilde{W}}$ to users.

Anti-reverse Engineering Module, which consists of *Anti-static Analyzing Module*(C_{as}) and *Anti-dynamic Debugging Module*(C_{ad}), offers the protection from reverse engineering attacks for C_m, C_d. C_{as} applies the anti-static analyzing techniques, and C_{ad} applies the anti-dynamic debugging techniques.

3 Embedding and Extraction of the CRSW

In this section, we discuss how to embed W, C_m, C_d and C_a into P, the construction of C_m, C_d and C_a will be described in the next section. We present *chaotic substitution* and *chaotic dispersion coding* before describing the embedding and extraction.

3.1 Chaotic Substitution ∂

Chaotic substitution is replacing i with c (c,i are two 8-bit binary integers), the result is that the value of i equates c, and we get s. With c and s, the original value of i is recovered by *reverse chaotic substitution*. Let G is digital chaotic system, without loss of generality, let the state space of G be $[a, b)$. Thus *Chaotic substitution* can be expressed by

$$s = \partial(i, c, G) = \lfloor 2^8 \times \frac{G(x, m) - a}{b - a} \rfloor \oplus i, x = \frac{c(b - a)}{2^8} + a, m = \lfloor \frac{c}{\lambda} \rfloor + 1 \quad (1)$$

where $\oplus$ is XOR. $G(x, m)$ is the state of G which has been iterated m times with the initial value x. λ is the parameter which can adjust iterated times. *Reverse chaotic substitution* is given by:

$$i = \partial^{-1}(s, c, G) = \partial(s, c, G) \quad (2)$$

Generally, the set $A = \{\alpha_1, \alpha_2, \cdots, \alpha_k\}$ replaces $B = \{b_1, b_2, \cdots, b_k\}$ with *chaotic substitution*, and get the result:

$$R = \{r_j\} = \partial(B, A, G) = \{\partial(b_j, a_j, G)\}, j = 1, 2, \cdots, k \quad (3)$$

The reverse procedure is given by

$$B = \partial^{-1}(R, A, G) = \partial(R, A, G) \quad (4)$$

3.2 Chaotic Dispersion Coding ξ

Let $X = \{x_1, x_2, \cdots, x_n\}$ be a chaotic sequence, without loss of generality, supposing $x_j \in [a, b), j = 1, 2, \cdots, n$. Chaotic dispersion coding is that dispersing W over I (the code of program), which is given by

$$[I', S'] = \xi(W, I, X) \quad (5)$$

where I' is the resulting code, S' is the save code.

Let the length of W be n bytes and the length of I be l bytes. Thus, $W = \{w_1, w_2, \cdots, w_n\}$, $I = \{i_1, i_2, \cdots, i_l\}$, $S' = \{s'_1, s'_2, \cdots, s'_n\}$. The steps of ξ are as follows:

1) Initialization: $L \leftarrow l$, $N \leftarrow n$, $m \leftarrow \lfloor L/N \rfloor$, $j \leftarrow i$, $d \leftarrow 0$, let $I' = \{i'_1, i'_2, \cdots, i'_l\} = I$
2) Let $r = \lfloor m \times \frac{x_j - a}{b - a} \rfloor$, $d = d + r$, $s'_j = \partial(i_d, w_j, G)$, $i'_d = w_j$
3) Algorithm is done, if $j = n$ is satisfied. Otherwise go to 4)
4) Let $L = L - r$, $N = N - 1$, $m = \lfloor L/N \rfloor$, $j = j + 1$, go to 2).

When algorithm is done, $S' = \{s'_1, s'_2, \cdots, s'_n\}$, $I' = \{i'_1, i'_2, \cdots, i'_l\}$. The *reverse chaotic dispersion coding*, recovering W and I with S' and I', can be expressed as: $[I, W] = \xi^{-1}(S', I', X)$.

3.3 Embedding

In CBSW, all of W, C_m, C_d, C_a are embedded into the executable code directly, the procedure is described below(Fig. 2 shows the change of executable code after embedding):

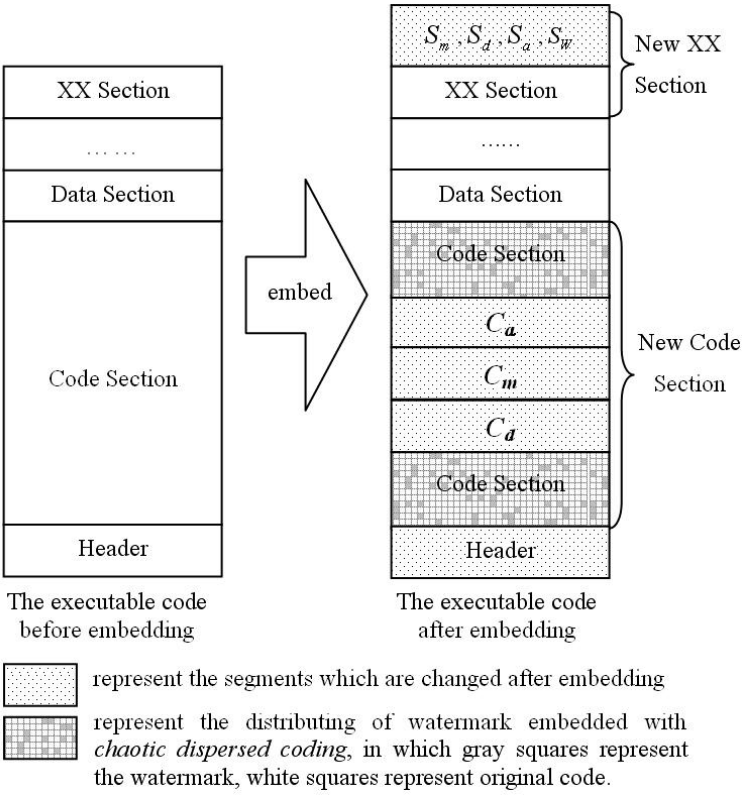

Fig. 2. The drawing of embedding watermark

1) Give Key $< K_1, K_2 >$, where K_1 is *activation key*, K_2 is the key of producing chaotic sequence. Supposing that length of watermark is n bytes, W can be expressed as $\{w_1, w_2, \cdots, w_n\}$.
2) Construct *Watermark Decoding Module* C_d, and *Anti-reverse Engineering Module* C_a; Construct *Input Monitoring Module* C_m with K_1(the details of constructions are discussed in the next section)
3) Produce the chaotic sequence $X = \{x_1, x_2, \cdots\}$
4) Apply *chaotic substitution* to embed C_m, C_d, C_a into the code of P. Let the code blocks which are replaced with C_m, C_d, C_a be I_m, I_d, I_a respectively. We can get $S_m = \partial(I_m, C_m, G)$, $S_d = \partial(I_d, C_d, G)$, $S_a = \partial(I_a, C_a, G)$, where G is the digital chaotic system.
5) Get the subsequence $X^{(1)}$ (the length is n) from X and preprocess W: $W' = E(W, X^{(1)}) = W \oplus X^{(1)} = \{w'_1, w'_2, \cdots, w'_n\} = \{w_1 \oplus x_1^{(1)}, w_2 \oplus x_2^{(1)}, \cdots, w_n \oplus x_n^{(1)}\}$, where $\oplus$ is XOR.
6) Get the subsequence $X^{(2)}$ (the length is n) from X; Embed W' to I (I is the code which is the whole code exclusive the code that is replaced with C_m, C_d, C_a) with *chaotic dispersion coding* and get the result $[I', S_W] = \xi(W', I, X^{(2)})$ (Fig. 2 shows the distribution of W' in the watermarked program).
7) Save S_m, S_d, S_a and S_W to the end of the executable code, and adjust the header of the executable code.

3.4 Extraction

Because the watermark extractor is embedded into the program, the extraction of the watermark is included in the execution of the watermarked program. We describe the execution of the watermarked program to illustrate the extraction.

1) The watermarked program runs.
2) The code of *Anti-reverse Engineering Module* runs.
3) The code of *Input Monitoring Module* runs, which monitor the input of the program.
4) Produce the chaotic sequence Y
5) Get the subsequence Y^2 (the length is n, $Y^{(2)}$ is the same as $X^{(2)}$ in the embedding algorithm) from Y, recover the code which is replaced with W', the procedure can be expressed by $[I', \widetilde{W'}] = \xi^{-1}(S_W, I', Y^{(2)})$.
6) Recover the code which is replaced with C_m, C_d, C_a, the procedure can be expressed by $I_m = \partial^{-1}(S_m, C_m, G)$, $I_d = \partial^{-1}(S_d, C_d, G)$, $I_a = \partial^{-1}(S_a, C_a, G)$.
7) The watermarked program keeps on running.
8) If the input matches with K_1 (*activation key*). Get the subsequence $Y^{(1)}$ (the length is n, $Y^{(1)}$ is the same as $X^{(1)}$ in the embedding algorithm) form Y, and put $\widetilde{W'}$ into $\widetilde{W}$ with inverse preprocess, which is $\widetilde{W} = E^{-1}(\widetilde{W'}, Y^{(1)}) = E(\widetilde{W'}, Y^{(1)})$.
9) Transform $\widetilde{W}$ into $V_{\widetilde{W}}$ (visual action) and perform $V_{\widetilde{W}}$.

4 The Analysis of CRSW

This section is intended to discuss the robustness of CRSW and the performance degradation. Let the lengths of W, C_m, C_a, and C_d be n bytes, l_m bytes, l_a bytes and l_d bytes respectively.

Firstly, we analyze the robustness. Let $R[\![P]\!]$ be the semantics of P, $\omega \in \Gamma_b = \{\varphi | R[\![\varphi(P)]\!]\} = R[\![P]\!]\}$ is semantics-preserving transformation. In CRSW, because of visual output $V_{\widetilde{W}}$, $R[\![V_{\widetilde{W}}]\!] \subseteq R[\![P']\!]$ is hold. Then the following relation holds according to the definition of semantics-preserving transformation:

$$R[\![V_{\widetilde{W}}]\!] \subseteq R[\![P']\!] = R[\![\omega(P)]\!] \tag{6}$$

Equation (6) indicates that the semantics-preserving transformations can not destroy the semantics of $V_{\widetilde{W}}$, and CRSW can resist various types of semantics-preserving transformation attacks except the attacks which can distinguish $R[\![V_{\widetilde{W}}]\!]$ and $R[\![P]\!]$.

In the *Anti-reverse Engineering Module*, anti-static analyzing techniques and anti-dynamic debugging techniques are introduced to thwart reverse engineering attacks. The performance of resistance against reverse engineering depends on anti-reverse engineer techniques applied in CRSW. As we can apply the more effective anti-reverse engineering techniques to the module that is dynamic and scalable, the resistance against reverse engineering will be enhanced. Moreover, watermark is embedded into the code of program by *chaotic dispersion coding*. Therefore, practicing the combination of the instructions and data, it improves the performance of anti-static analyzing.

Because of the application of the chaotic dispersion coding, the watermark will cause the program to fail if the adversary wants to reuse any part of code solely. Since W' is distributed uniformly over the code which is the whole code exclusive the code that is replaced with C_m, C_d and C_a, there is a byte of watermark per $\frac{l-l_m-l_d-l_a}{n}$ bytes code averagely. Thus:

$$l_v = \frac{l - l_m - l_d - l_a}{n} \tag{7}$$

where l_v is the average length of the reused code. If $l_v \leq l_T$ are ensured, n, the length of watermark, must satisfy the inequation $n \geq \lfloor \frac{l-l_m-l_d-l_a}{l_T} \rfloor$.

It is difficult to locate the watermark because the position of W' is generated by chaotic sequence. In addition, $s = \partial(i, c, G)$ (*chaotic substitution*) can be considered that i is encrypted with G and c. If c is tampered, i could not be decoded correctly when $i = \partial^{-1}(s, c, G)$. In CRSW, if W' is tampered, it is impossible to recover the code which is replaced with W' correctly in the extracting procedure; if C_m, C_a and C_d is tampered, it is also impossible to get back the code which is replaced with C_m, C_a and C_d correctly, which could cause the program to fail. As for the given G, c is assumed the secret key, thus the key space should be 2^{l_c} (l_c is the length of c); the key space is $2^{8(n+l_m+l_d+l_a)}$ in CRSW.

We analyze the performance degradation of watermarked program below. From the point of space, embedding watermark increases the size of the program.

In the embedding procedure, the size of the program increases $n + l_m + l_d + l_a$ bytes because of *chaotic substitution* which is applied to our algorithm. From the point of runtime, embedding watermark brings the increasing runtime of the program. The reason is that before the execution of the watermarked program, the original code should be recovered from S_m, S_d, S_a and S_W, of which the recovering time not only depends on the iterative efficiency of digital chaotic system, but also the contents of W', C_m, C_a and C_d. Let t be the time of iterating once, and T_1, the time of recovering code from S_m, S_d and S_a, satisfies the following inequation:

$$(l_m + l_d + l_a)t \leq T_1 \leq \frac{1}{\lambda}(l_m + l_d + l_a)t \times 2^8 \tag{8}$$

With recovering code from S_W, chaotic sequence of n bytes should be generated for ξ^{-1} at first. Thus T_2, the time of recovering code from S_W, satisfies:

$$nt + nt \leq T_2 \leq nt + \frac{2^8}{\lambda}nt \tag{9}$$

$T_1 + T_2$, the time of recovering all code, satisfies

$$2nt + (l_m + l_d + l_a)t \leq T_1 + T_2 \leq nt + \frac{2^8}{\lambda}(n + l_m + l_d + l_a)t \tag{10}$$

If W', C_m, C_a and C_d is bit-balance (Bits 0 and 1 occur at the same frequency), the average time of the procedure is

$$\overline{T} = nt + \frac{2^7 + 0.5}{\lambda}(n + l_m + l_d + l_a)t \tag{11}$$

In general, C_m, C_a and C_d are fixed, that is to say, $l_m + l_d + l_a$ is constant, and t is also a constant for a given digital chaotic system, the equation (11) can be rewrite as follow:

$$\overline{T} = nt(1 + \frac{2^7 + 0.5}{\lambda}) + \frac{2^7 + 0.5}{\lambda}(l_m + l_d + l_a)t = \beta_1 n + \beta_2 \tag{12}$$

where β_1, β_2 are constants. Equation (7) shows that the larger n is, the smaller l_a is, and the more intensive the protection is. Equation (12) shows that $\overline{T}$ is linearly increased in a manner that involves n. The users who are intent to apply CRSW should exhibit a trade-off between intensity of protection and the performance degradation.

5 Implementation of CRSW

The algorithm's implementation is in the Intel i386 architecture and the Windows operating system. This section is to expatiate on the implementation of *Input Monitoring Module*, *Anti-reverse Engineering Module*, and *Watermark Decoding Module*. There are several problems that arise when implementing these modules, and the corresponding solutions are given at the end of this section.

5.1 Input Monitoring Module C_m

The purpose of C_m is to monitor the input of the program. When implementing C_m, we put *activation key* K_1 (or $\mu(K_1)$, μ is a one-way function) into this module. When the input α is a match for K_1 (or $\mu(K_1)$), *Watermark Decoding Module* is activated.

5.2 Anti-reverse Engineering Module C_a

In theory, a sufficiently determined attacker can thoroughly analyze any software by reverse engineering. It is impossible to thwart completely reverse engineering attacks. The goal, then, is to design watermarking techniques that are "expensive enough" to break-in time, effort, or resources—that for most attackers, breaking them isn't worth the trouble. There are two kinds of techniques-static analyzing and dynamic debugging—in the reverse engineering techniques. Therefore, C_a consists of *Anti-Static Analyzing Module* and *Anti-Dynamic Debugging Module*.

Decompile is the foundation of static analyzing techniques, we can disable static analyzing by disturbing decompiler which is developed based on the hypothesis that data and instructions are separated. However, data and instructions in the Von Neumann architecture are indistinguishable. Thus, we can mix data and instructions in order to disturb decompiler by adding special data and instructions (we call them disturbing data) between instructions. In Fig. 3, (a) gives source code by assembly language, lines 1,5,6 are original instructions, but lines 2,3,4 are the disturbing data. (b) shows the instructions from decompiler. We can see that there are errors from line 4 to the end. There are a number of disturbing data in [4]. In this paper, we insert several disturbing data into C_m, C_a and C_d. If we can apply code encryption, compression etc. to *Anti-Static Analyzing Module*, the performance will be further improved.

Dynamic debugging relies on debugging tools highly, so the general principle of anti-virus can be introduced to detect whether program is being debugged or not by the characters of debug tools. If debugged, the program will jump to wrong control flow in order to prevent from debugging .We already have achieved the algorithm based on the characters of SoftICE, Windbg, and Ollydbg. Experiments demonstrate that it is available to resist these debugging tools. The characters of other debugging tools can be introduced to the improved implementation.

```
1        xor     eax,eax
2        jnz     Next
3        jz      Next
4        db      0e8h
5   Next: xor    ebx, 1
6        add     ebx, 2
```

(a)

```
.text:004010D7  xor eax, eax
.text:004010D9  jnz short near ptr loc_4010DD+1
.text:004010DB  jz short near ptr loc_4010DD+1
.text:004010DD  call  near ptr 83420465h
.text:004010E2  retn
```

(b)

Fig. 3. Example of disturbing data

There are several registers for debugging in the processor of the i386 architecture. Several debugging tools design feasible functions, such as BPM[1], hardware breakpoint, by involving the debug registers [4]. In the paper, we modify the value of debugging register and invalidate these functions. In addition, time sensitive code and breakpoint detection are introduced to the implantation.

We have involved several kinds of anti-reverse engineering techniques. It is worthy mentioning that this module is scalable that more efficient anti-reverse engineering techniques can be introduced. Thus, they can enhance the resistance against reverse engineering attacks.

5.3 Watermark Decoding Module C_d

Watermark Decoding Module includes *Watermark Output Module* and *Chaotic System Module*. Watermark Output Module transforms the watermark extracted from watermarked program into visual output; *Chaotic System Module* implements digital chaotic system, which is only applied to the watermark extracting procedure. When chaotic systems are discretely realized in finite precision, some serious problems will arise, such as dynamical degradation, short cycle length and non-ideal distribution. And then, we must compensate for the dynamical degradation in the presence of chaos system. We apply 1D piecewise linear chaotic maps, and the scheme of compensation for degradation in [11] to our implantation.

5.4 Problems and Solutions

Because of directly embedding watermark into the executable code, when implementing C_m, C_a and C_d, two problems arise as follows: (1) After every module is embedded into various executable code, the code and data are loaded onto different addresses of memory, and the code can't access data in memory correctly. Therefore, it must do self-location (locate the memory address by the code itself). (2) Since it is unnecessary to recompile after embedding, modules can't automatically find the address of Windows API by compiler and loader, but get the address by themselves.

The self-location of code and data can be implemented by call/pop/sub instructions. Fig. 4 gives the specific codes. EBX, a register, is used to save the difference of the loading address and the designing address. The loading address is the sum of EBX and the designing address, which is self-location.

The procedure in getting the addresses of Windows APIs is as follows:

1) Get the loading base address of kernel32.dll. There is exception handling in Windows—structured exception handling (SHE). All exception handling functions are in a linked list, and the last element of the linked list is the default exception handling function which is in the module of kernel32.dll. We can gain the address of the default exception handling function through traversing the linked list, from which we can get the loading base address of kernel32.dll.

[1] BPM, an instruction of SoftICE, can set a breakpoint on memory access or execution.

```
        CALL    label
LAB:    POP     EBX
        SUB     EBX, OFFEST LAB
```

Fig. 4. The code of self-location

2) Get the addresses of LoadLibrary and GetProcAddress, which are Windows APIs, from the export table of kernel32.dll by the loading base address of kernel32.dll.
3) Get the address of the arbitrary Windows API with LoadLibrary and GetProcAddress.

6 Conclusion

A chaos-based robust software watermarking algorithm is proposed in this paper, in which the anti-reverse engineering technique and chaotic system are combined with the idea of the Easter Egg software watermarks. In CBSW, *Anti-reverse Engineering Module* is open and scalable, and more efficient anti-reverse engineering techniques can be applied. The program can be protected by embedding the watermark into the entire codes with *chaotic dispersion coding*. It is difficult for the adversary to tamper the message (includes W, C_m, C_d and C_a) embedded in the program with *chaotic substitution*. The analysis of the CRSW shows that the scheme can thwart various types of semantics-preserving transformation attacks, such as dead code wiping, code optimization, code obfuscation, and variable reconstruction. Furthermore, it improves resistance against reverse engineering attacks to a certain extent.

Acknowledgement

The work is supported partially by the National Natural Science Foundation of China (Grant No.60374004), partially by the Henan Science Fund for Distinguished Young Scholar(Grant No.0412000200), partially by HAIPURT(Grant No. 2001KYCX008), and the Science-Technology Project of Henan Province of China.

References

1. C. Collberg, C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. IEEE Trans. Software Engineering. Vol.28, No.8, pages: 735-746
2. Zhang Lihe, Yang YiXian, Niu Xinxin, Niu Shaozhang. A Survey on Software Watermarking. Journal of Software. Vol.14, No.2, pages: 268-277, in Chinese.
3. Business Software Alliance. Eighth annual BSA global software piracy study: Trends in software piracy1994-2002, June 2003.

4. Kan X. Encryption and Decryption: Software Protection Technique and Complete Resolvent. Beijing: Electronic Engineering Publishing Company, 2001, in Chinese.
5. Jasvir Nagra and Clark Thomborson. Threading software watermarks. In 6th Workshop on Information Hiding, 2004, pages: 208-223.
6. Robert L. Davidson and Nathan Myhrvold. Method and system for generating and auditing a signature for a computer program. US Patent 5,559,884, September 1996. Assignee: Microsoft Corporation.
7. Julien P. Stern, Gael Hachez, Franois Koeune, and Jean-Jacques Quisquater. Robust object watermarking: Application to code. In 3rd International Information Hiding Workshop, 1999, pages: 368-378.
8. C. Collberg and C. Thomborson, Software watermarking: Models and dynamic embeddings. Proceedings of POPL'99 of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1999, pages: 311-324.
9. Patric Cousot and Radhia Cousot. An abstract interpretation-based framework for software watermarking. In ACM Principles of Programming Languages(POPL'04), Venice, Italy, 2004, pages: 173-185.
10. Christian Collberg, Andrew Huntwork, Edward Carter, and Gregg Townsend. Graph theoretic software watermarks: Implementation, analysis, and attacks. In 6th Workshop on Information Hiding, 2004, pages:192-207.
11. Liu Bin, Zhang Yongqiang, and Liu Fenlin. A New Scheme on Perturbing Digital Chaotic Systems. Computer Science, Vol.32, No.4, 2005, pages: 71-74, in Chinese