doi:10.3969/j.issn.1000-436x.2013.02.016

基于内存操作的动态软件水印算法

许金超[1,2]，曾国荪[1,2]
(1. 同济大学 计算机科学及技术系，上海 201804；2. 教育部嵌入式系统与服务计算教育部重点实验室，上海 201804)

摘 要：提出了一种基于内存操作的动态软件水印算法，算法通过控制本地代码程序中的内存分配释放和使用等行为隐藏软件水印。定义了动态簇、簇间内存关系、内存关系矩阵等概念，详细描述了软件水印的嵌入和提取过程。分析了算法的平台依赖性、可信性、数据率、开销，并通过实验对该算法的隐蔽性和抗攻击能力进行了探讨。
关键词：数字版权；软件水印；内存操作；水印算法
中图分类号：TP391 **文献标识码**：B **文章编号**：1000-436X(2013)02-0128-10

Dynamic software watermarking algorithm based on memory operation

XU Jin-chao[1,2], ZENG Guo-sun[1,2]
(1. Department of Computer Science and Technology, Tongji University, Shanghai 201804, China;
2. Key Laboratory of Embedded System and Service Computing, Ministry of Education, Shanghai 201804, China)

Abstract: Memory operation based watermarking, a novel approach to software watermarking based on the dynamic memory behaviors of native code programs was introduced. The definition of dynamic cluster, memory relationship between clusters and matrix of memory relationship was given, and a detailed description of the software watermark embedding and extraction procedure was given. The platform independence, credibility, data rate and cost of the algorithm were analyzed, its stealthy and resistance were evaluated by experiments.
Key words: digital copyright; software watermarking; memory operation; watermarking algorithm

1 引言

随着互联网的快速发展，数字产品的分发变得简单快捷，传播范围更加宽广。数字产品的版权保护受到了严峻的挑战，其中，软件产品的状况尤为不容乐观，软件盗版已经成为软件开发者最大的威胁。最近的报告[1]表明，2011 年世界被盗版的软件占总软件数量的 42%，价值高达 630 亿美元，而中国的盗版率更是高达 77%。盗版的猖獗严重打击了软件开发者的积极性，为软件开发者提供有效的软件保护方法成为亟待解决的问题。软件水印是软件保护方法中一种重要的技术，它将标志版权的秘密信息嵌入到要保护的软件中达到保护的目的，这些秘密信息不影响软件使用，不易被察觉，并且难以清除，在盗版发生时可以提取出来证明该文件的版权所有。

软件水印通常分为静态软件水印和动态软件水印 2 种[2]。静态软件水印隐藏在程序文件自身中，

收稿日期：2012-02-15；**修回日期**：2012-08-17
基金项目：国家高技术研究发展计划（“863”计划）基金资助项目(2009AA012201)；国家自然基金资助项目(61272107，61103068)；NSFC-微软亚洲研究院联合基金资助项目(60970155)；教育部博士点基金资助项目(20090072110035)；上海市优秀学科带头人计划基金资助项目(10XD1404400)；教育部网络时代的科技论文快速共享专项研究课题基金资助项目(20110740001)
Foundation Items: The National High Technology Research and Development Program of China(863 Program) (2009AA012201); The National Natural Science Foundation of China (61272107，61103068); The Joint of NSFC and Microsoft Asia Research(60970155); The Ph. D Programs Foundation of Ministry of Education(20090072110035); The Program of Shanghai Subject Chief Scientist (10XD1404400); The Special Fund for Fast Sharing of Science Paper in Net Era by CSTD(20110740001)

如保存在数据段或代码段，而动态软件水印则在程序运行时实时生成，信息隐藏在运行过程中的内部状态中。动态软件水印又可以划分为数据结构水印(data structure watermark)、执行过程水印(execution trace watermark)和复活节彩蛋水印(easter egg watermark) 3 类[2,3]。数据结构水印算法以 CT 算法[2]为代表，它也是已知的第一个动态软件水印算法。CT 算法中，软件水印隐藏在程序运行时动态建立的图拓扑结构中。同一类型的算法还有文献[4]~文献[6]等。执行过程水印算法以动态路径算法[7]为代表，它给出了 Java 程序和本地代码程序 2 种不同载体下的实现方式，软件水印隐藏在程序执行过程中选择的路径分支中。文献[8]是另外一种典型的执行过程水印算法，它通过在 Java 程序中选择部分方法，在这些方法的第一个基本块中使用一定数量的内存，从而改变程序的运行过程中的等分时间片内内存的使用数量来隐藏软件水印。执行过程水印的实现算法还有文献[9]~文献[11]等。文献[12]提出的混沌软件水印算法属于复活节彩蛋水印算法，它利用混沌序列把软件水印信息均匀散列在程序的代码段中，当输入正确的密钥时，软件水印信息通过对话框的形式显示给用户。

已有的动态软件水印算法中仍存在许多不足的地方。首先是算法实现时较少结合载体程序的性质，用于隐藏软件水印的代码和程序中原有代码存在明显不同，从而造成了软件水印和载体程序之间产生了一条分界线，使得软件水印容易被发现并发动针对性的攻击。如 CT 算法中，隐藏软件水印的数据结构的生成过程和程序的正常功能缺少联系，很容易遭到逆向攻击者怀疑，且存在大量重复性的代码，无法抵抗模式匹配攻击。其次，在目前已实现的动态软件水印算法中，保护对象都是以 Java 程序为主，适用于本地代码程序的算法所占比例仍然不高。本地代码程序常见的以 C 或 C++语言等开发，它们在执行效率上存在的优势使之在软件市场中依然占据重要份额，因此用于本地代码程序的软件水印算法仍有很大的需求。最后，已有的许多软件水印算法只能容忍小范围的软件变换，然而目前有许多技术可以将一个程序大范围改变的面目全非，例如代码变形技巧能够将同一段代码变换成数种不同的形式。实用的软件水印算法需要能抵抗形形色色的攻击。

为了解决已有软件水印算法的这些不足，需要面向使用 C 或 C++等语言编程的开发者，提出一种高度隐蔽并且有效抵抗各种攻击的软件水印算法。程序执行过程中，内存操作是很常见的行为，对内存分配释放的方式和使用的内存数量进行修改对程序的大小影响不大，而且不容易引起怀疑。文献[8]正是修改了程序类方法中的内存使用数量达到嵌入软件水印的目的，尽管总是在类方法的第一个基本块中使用内存这一行为也易受瞩目，但仍然提供了一种可借鉴的实现思想。然而在本地代码程序中，程序中的函数通常不像 Java 程序中的类方法那么普遍，也不存在虚拟机可供方便的统计内存的使用信息。除此之外，Java 程序中不需关心内存释放、数组越界等内存问题，而本地代码中的内存操作造成的内存泄露、内存溢出、野指针等内存相关问题至今都是软件开发中需要面对的考验，并且这些问题依然会在很长一段时间内存在。对于拥有程序源代码的开发者则可以通过对程序的了解比攻击者更清楚的优势利用内存操作隐藏软件水印，而对程序细节不了解的攻击者任意修改内存会造成多种内存问题。基于这些考虑，本文提出一种基于内存操作的动态软件水印算法。该方法通过调整内存的分配释放时机和分配的内存数量以及程序执行时使用的内存数量来嵌入软件水印，并充分利用开发者相对于攻击者的优势，通过内存分配和使用之间的制约关系增加攻击者修改内存的难度来保证软件水印的安全。

2　基本概念

令 P 表示一个可执行程序，I 表示 P 的一个合法输入，这里 I 可以为空，此时表示程序不需要外在输入，如不带参数的命令行程序。

定义 1　动态簇。动态簇是 P 在给定输入 I 下执行经过的部分代码的集合。

根据不同的划分准则，可以把程序执行过程中经过的所有代码划分为若干个动态簇。下文中动态簇简称簇。

定义 2　簇间内存分配关系。如果 P 运行过程中簇 a 中分配的所有内存中有 $n$byte 的内存在簇 b 中得到释放，则称簇 b 释放了簇 a 分配的 $n$byte 的内存，记作 $R(a, b)= n$。

定义 3　簇间内存使用关系。如果程序 P 运行过程中簇 a 分配的内存在簇 b 执行结束时（若簇 b 执行结束前这些内存已经释放则是这些内存释放

时）有 $n$byte 的内存较簇 b 执行开始时不同，则称簇 b 使用了簇 a 分配的 $n$byte 的内存，记作 $U(a,b)=n$，若 $a=b$，则令 $U(a,b)=0$。

定义 4 内存关系矩阵。设 P 的执行过程按顺序划分为 n 个簇 c_1、c_2、…、c_n，则矩阵 $\boldsymbol{PR}$、$\boldsymbol{PU}$ 分别是按如下定义的 n 阶方阵。

$$\boldsymbol{PR}=(r_{ij})_{n\times n}: r_{ij}=\boldsymbol{R}(c_i,c_j)$$

$$\boldsymbol{PU}=(u_{ij})_{n\times n}: u_{ij}=\boldsymbol{U}(c_i,c_j)$$

其中，$\boldsymbol{PR}$ 称为内存分配矩阵，$\boldsymbol{PU}$ 称为内存使用矩阵，显然，当 $j<i$ 时，$\boldsymbol{PR}$ 中元素满足 $r_{ij}=0$；当 $j\leqslant i$ 时，$\boldsymbol{PU}$ 中元素 $u_{ij}=0$，即 $\boldsymbol{PR}$、$\boldsymbol{PU}$ 是三角矩阵。为了充分利用矩阵空间，便于隐藏数据，对于三角矩阵 $\boldsymbol{PR}$ 和 $\boldsymbol{PU}$，可以合为一个 $n\times n$ 矩阵 $\boldsymbol{M}$，称为内存关系矩阵。其中，$\boldsymbol{M}=\boldsymbol{PR}+\boldsymbol{PU}^{\mathrm{T}}$ 即 $\boldsymbol{M}$ 中元素满足

$$m_{ij}=\begin{cases}u_{ji}, j<i<n\\ r_{ij}, i\leqslant j\leqslant n\end{cases}$$

3 基于内存操作的软件水印算法

3.1 算法概述

本文算法中软件水印的载体为本地代码程序，在嵌入时需要拥有程序的源代码，在此基础上，通过修改源代码操控内存行为从而在源代码编译来的程序中嵌入软件水印。

程序中涉及到的内存有用户态内存和内核态内存之分，如果同时考虑所有这些内存需要拥有内核权限，这通常不太现实。为了简化操作，本文所指的内存指的能在编程中使用指定函数或关键字分配的内存，如 C 语言中的 malloc、calloc 等函数。

算法在实现过程使用了 IDA Pro 软件[13]，IDA Pro 是一款优秀的反编译工具和调试器，支持 IDC 脚本，提供 SDK 实现功能扩展，并且附带 IDAPython 插件[14]支持 Python 脚本。算法中大部分的实现工作通过编写 Python 脚本完成，涉及到源代码修改时需要人工参与来保证修改的正确性。

嵌入过程分为 4 个主要步骤。首先，对程序进行分簇；其次，监控程序执行，生成内存关系矩阵；然后，将软件水印嵌入到内存关系矩阵中；最后，根据嵌入软件水印后的内存关系矩阵修改程序源代码。提取过程相对简单，它可分为 3 个主要步骤：首先，对已嵌入软件水印的程序进行分簇；然后，监控含软件水印程序的运行得到内存关系矩阵；最后，从内存关系矩阵中提取软件水印。

3.2 簇间内存关系统计

在嵌入软件水印和提取软件水印的过程中，都需要统计程序中的簇间内存关系得到内存关系矩阵。这一准备工作包括簇的划分和生成内存分配矩阵、内存使用矩阵 2 部分。下文中仍用 P 表示可执行程序，I 表示程序 P 运行时的秘密输入。

3.2.1 簇的划分

本节给出按数量划分、按时间划分和按输入分簇 3 种簇划分方式，用以适应不同的软件水印载体程序，划分中都使用密钥 ω 将 P 在输入 I 下运行时产生的执行踪迹划分为 n 个簇。

按数量分簇方式中，首先在输入 I 下运行程序 P，记录程序执行过程中的指令总量，然后以密钥 ω 作为随机种子，将程序运行过程中执行的指令划分为 n 段。每段作为一簇。这种划分方式下的分簇过程实现简单，而且划分时输入 I 可以为空，适用于控制台程序等缺乏输入参数的小规模单线程程序。

按时间分簇方式首先要选择合适的输入 I 使得 P 的运行能够维持在一个相当长的时间内，如用延长 2 次鼠标点击的间隔来控制程序运行总时间。在 I 下执行程序 P 并记录该输入下程序的运行时间，然后利用密钥 ω 作为随机种子对运行时间进行划分。当每个划分的时间片到达时，挂起程序中断 P 的运行，将在分时间片内参与执行的代码作为一簇，这同样将 P 的运行踪迹划分为 n 簇。此分簇方式需要较长的程序执行时间，不适用于无法控制程序运行时间的程序。

按输入分簇同样需要选择秘密输入 I 使程序 P 能够执行相当长的一段时间，同时秘密输入自身也需要由足够多的连续子输入构成，利用密钥 ω 将这些子输入划分为 n 个连续不相交集合，在每个子输入集合下执行的代码作为一簇，同样把程序划分为 n 簇。这种分簇方式适合操作比较复杂的程序。

3.2.2 内存分配矩阵和内存使用矩阵的生成

分簇确定后即可监测簇间的内存关系，进而得到内存分配矩阵和内存使用矩阵。这一过程可以根据对源代码的理解结合调试器通过人工分析的方式来完成，然而对较大的程序这种方法工作量过大，需要给出这一过程的自动化实现方法。

根据簇间内存分配和使用关系的定义，实现中要跟踪程序执行过程中分配的每一内存块从分配到释放整个过程中在每一簇间的变换情况，因此执行过程中每一内存块需要记录表 1 所示的内容。

表 1　内存块信息记录包含的内容

内容名称	内容说明
ID	标识内存块的唯一编号
Address	内存块起始地址
Size	内存块的大小
AllocateClusterID	程序分配该内存块时的簇号
ReleaseClusterID	程序分配该内存块时的簇号
SouceLine	该内存块分配操作在程序源代码中的位置
MemDif_Cluster_1	在第 1 簇执行前后该内存块中发生变化的内存
MemDif_Cluster_2	在第 2 簇执行前后该内存块中发生变化的内存
…	…
MemDif_Cluster_n	在第 *n* 簇执行前后该内存块中发生变化的内存

程序运行过程中指定函数分配的所有内存块信息记录组织成内存块记录表，使用数据库存放。收集这些内存块信息的伪代码实现如图 1 内存块信息收集算法所示。为了防止载体程序在调试态下产生的内存差异，通过在载体程序入口点设置断点的方式由系统启动 IDA，然后在 IDA 中载入脚本运行内存块信息收集算法。内存块信息收集算法中数字标出的 4 个部分是算法中最核心的内容，下面给出进一步说明。

```
input: 载体程序 P、输入 I、分簇信息 C
output：内存块记录表 memTable
begin
    memTable ← ∅
    //1.拦截相关内存函数，创建并启动监控线程
    HookMemfunc()
    CreateMonitorThread(P, I, C)
    //如果有调试事件发生
    while null<>(event ← GetDebuggerEvent())
        //通知监控线程已进入调试事件处理
        do NotifyMonitorThreadEnter()
        //如果是由监控的内存函数触发
          if IsHOOKEVENT(event) then
            if IsAllocateMemory(event) then
          //2.调用内存分配函数时记录分配的内存块信息
            FillMemBlockInfor(memTable)
          else
          //3.当内存块释放时更新该内存块信息记录
            UpdateMemBlockInfo(memTable)
        //如果是由簇断点触发
            elseif IsCLUSTERBREAKEVENT(event) then
        //4.在执行下一簇代码前更新所有未释放的内存块记录
            UpdateAllMemBlockInfor(memTable)
    //通知监控线程已完成调试事件处理
        NotifyMonitorThreadLeave()
    //继续执行程序
        ContinueDebugContent()
    return(memTable)
End
```

图 1　内存块信息收集算法

1) 拦截相关内存函数，创建并启动监控线程

拦截相关内存函数就是在载体程序中相关的内存函数入口处及出口处设置断点，从而接管内存函数获得创建或释放的内存块信息。IDA 提供了 API 可以简化了这一过程的实现。此后创建监控线程并传入载体程序信息，监控线程负责控制载体程序的输入，并根据选择的分簇信息计算载体程序每一簇结束的地址，并择机在载体程序中加入簇断点。

2) 内存分配函数调用时记录分配的内存块信息

当载体程序中发生内存分配操作的时候在内存块记录表里生成一条新内存块记录，在记录中填入该内存块的 ID、Address、Size、AllocateClusterID、SourceLine 等信息。最后分配一个大小和该内存块大小相同的影子内存暂时保存该内存块内容。

3) 内存块释放时更新该内存块信息记录

当监测到内存释放操作时，查找内存块记录表，找出待释放的内存块所在的记录，填入 ReleaseClusterID，比较待释放的内存块对应的影子内存和待释放内存块当前数据的差异并将变化的内存数量值填入该内存块记录中 ReleaseClusterID 对应的 MemDif_Cluster 字段中，然后释放相应的影子内存。

4) 簇中断时更新所有未释放的内存块记录

当发生簇中断时意味着当前簇结束马上进入下一簇的执行，设当前簇簇号为 k。此时在内存块记录表中找到当前每一个未释放的内存块，比较该内存块和对应的影子内存的差异，计算变化的内存数量，然后将此数值填入该内存块记录中的 MemDif_Cluster_k 字段中，最后将该内存块内容保存到影子内存中。

当程序执行完成时，相应的内存块记录表也已经完善。 统计表中的内存分配释放记录，将结果按定义填入内存分配矩阵。统计表中的内存块在各个簇间的变化记录，将结果按定义填入内存使用矩阵。

为了进一步降低误差，可以重复多次生成内存分配矩阵和内存使用矩阵，然后对得到的矩阵中的对应元素求平均值。

3.3　软件水印的嵌入

软件水印首先使用量化技术嵌入在内存分配矩阵和内存使用矩阵合并而来的内存关系矩阵中，内存关系矩阵每个元素嵌入一位软件水印数据，然后修改载体程序源代码使得源代码编译后的本地代码程序生成的内存关系矩阵和量化后的内存关

系矩阵完全一致。

3.3.1 软件水印的嵌入过程

设嵌入到程序 P 中的软件水印 W 是宽高皆为 n 的二值图像，设其对应 $n\times n$ 的二值矩阵为 $\boldsymbol{NW}=\{w_{ij}\}_{n\times n}$，其中，$w_{ij}$=0 或 1。则在 P 中嵌入 W 的步骤如下。

1) 利用密钥 ω 对 $\boldsymbol{NW}$ 进行置乱变换，设置乱后的矩阵为 $\boldsymbol{PW}=\{p_{ij}\}_{n\times n}$。

2) 在输入 I 下运行程序 P 并利用密钥 ω 将其划分为 n 个簇。

3) 监测簇间内存关系得到内存分配矩阵 $\boldsymbol{PR}$、内存使用矩阵 $\boldsymbol{PU}$，计算内存关系矩阵 $\boldsymbol{M}=\boldsymbol{PR}+\boldsymbol{PU}^{\mathrm{T}}$，显然 $\boldsymbol{M}$ 中元素 m_{ij} 和 $\boldsymbol{PM}$ 中元素 p_{ij} 存在一一对应关系。

4) 根据 p_{ij} 的值是 0 还是 1，使用 2 个不同的量化器 Q_0、Q_1 对 m_{ij} 进行量化。2 个量化器分别为

$$Q_0(x)=\left\lceil\frac{x}{\Delta}\right\rceil\Delta$$

$$Q_1(x)=\left(\left\lceil\frac{x-\frac{\Delta}{2}}{\Delta}\right\rceil+\frac{1}{2}\right)\Delta$$

对 Q_0、Q_1 有 $Q(x)\geqslant x$ 且 $Q(Q(x))=Q(x)$。即量化之后的数据不低于没量化以前，并且已量化的数据再次量化将保持不变。为了便于内存对齐，这里取量化步长 $\Delta=4y$，其中，$y\in N^{+}$。

对 $\boldsymbol{M}$ 中所有元素进行量化后得到的矩阵记为 $\boldsymbol{M}'=\{m'_{ij}\}_{n\times n}$。

5) 由于分配的内存数量需要高于使用的内存数量，同时也不能过高造成内存的浪费。在上一步骤的量化过程中，当 $\boldsymbol{M}$ 中来自于内存使用矩阵 $\boldsymbol{PU}$ 的元素 m_{ij} 发生改变时，即 $m_{ij}\neq m'_{ij}$ 且 $j<i$，此时需要分配较多的内存来满足内存使用量增加带来的内存需求，即要对 $\boldsymbol{M}'$中来自 $\boldsymbol{PR}$ 的元素进行修改。若 m_{ij}=0,则令 $m'_{ji}=Qp_{ji}(m_{ji}+m'_{ij}-m_{ij})$。若 $m_{ij}\neq 0$，则至少存在一个簇 j 分配并在簇 i 使用的内存块，设在内存监测记录中找到该内存块将在簇 k 释放，则令 $m'_{jk}=\max\{m'_{jk},Qp_{jk}(m_{jk}+m'_{ij}-m_{ij})\}$，暂存该内存块编号留待下一步骤使用。完成所有这些元素修改后的矩阵用 $\boldsymbol{M}''=\{m''_{ij}\}_{n\times n}$ 表示。

6) 最后修改源程序使其编译后得到的本地代码程序生成的内存关系矩阵和 $\boldsymbol{M}''$一致，这需要找出所有 $\boldsymbol{M}''$和 $\boldsymbol{M}$ 不一致的元素并在源代码中相应位置进行修改。

① 首先修改源代码改动内存分配数量。对任一满足 $m_{ij}\neq m''_{ij}$ 且 $i\leqslant j$ 的元素，显然该元素涉及到簇 i 和簇 j 间的内存分配关系。查找内存块记录表，若存在簇 i 分配簇 j 释放的内存块记录，并且步骤 5)存储的内存块正是其中一个，则根据该内存块记录中的 SourceLine 定位内存分配在源代码中位置，调整分配该内存块的函数申请的内存数量，否则任选一个内存块记录表中簇 i 分配簇 j 释放的内存块对应的分配函数进行调整，使其分配的内存数量增加 $m''_{ij}-m_{ij}$；若簇 i 中不存在对应于簇 j 释放的内存的分配关系，则在簇 i 生成内存分配函数分配 $m''_{ij}-m_{ij}$ 大小的内存并在簇 j 释放，选择合适位置插入生成的代码并保证任何输入下程序执行时分配的内存一定可以释放。

② 然后修改源代码使之满足 $\boldsymbol{M}''$和 $\boldsymbol{M}$ 中 $i>j$ 且 $m_{ij}\neq m''_{ij}$ 的元素的变化，这说明簇 j 中分配并在簇 i 中使用的内存大小需要改为 m''_{ij}，该内存在簇 j 中分配，所以簇 j 分配的内存块中，优先选择上面改动内存分配数量时添加的内存分配函数和改动的内存分配函数分配的内存块，根据该内存块的地址和调整后的大小计算结束地址，然后在簇 i 中插入代码实现在该内存块结束地址处从后往前改动该内存块的内容，共改动 $m''_{ij}-m_{ij}$ 大小的内存。重复这个过程完成所有来自于内存使用矩阵的不一致元素的处理。这一步骤的中的内存修改，在改动数量较小的情况下可以针对内存中的每个字节写入一个不同的值，改动数量较大时也可以利用 memset 之类的库函数一次更改一块内存。

3.3.2 软件水印嵌入过程中的问题和处理方式

嵌入软件水印的过程需要对载体程序的源代码进行修改，修改的内容包括以下 3 种：改变内存分配的数量、成对增加内存分配和释放操作、对分配的内存内容进行修改。修改编译后的载体程序大小和执行时间已经有所变化，需要解决这些变化对软件水印和载体程序可能带来的影响，保证修改后的载体程序中已经正确地包含了软件水印，并且不会影响载体程序的正确运行。

首先，源代码修改导致编译后的载体程序发生变化，由于软件水印实际上隐藏在载体程序所生成的内存关系中，因此需要保持软件水印嵌入前后分簇的位置一致，从而保证软件水印嵌入前后载体程序生成的内存关系矩阵的对应。若算法中使用按时

间分簇方式或按输入分簇方式，只需在选择恰当的秘密输入保证程序的执行时间足够长，此时源代码修改编译后在本地代码程序中增加的代码内容产生的执行时间远远小于整个程序的执行时间，因此程序嵌入软件水印前后簇划分的位置不变或微变。此时无需对已嵌入软件水印的程序做任何处理即可保证程序的修改对簇间内存关系不产生影响。若算法中使用按数量分簇方式，当编译后的载体程序自身参与执行的代码较多且增加的代码在其中所占比例极小时，此时程序修改后分簇中每个簇的位置较程序修改前只相差少数指令，而影响内存关系的代码落入这部分指令中的概率不大，同时适当调整增加的代码的位置，使其不对簇间内存关系产生影响。对于自身体积较小的载体程序来说需要在编译后的载体程序中各个簇之间按比例插入冗余指令保证程序修改前后每个簇仍然划分在正确的位置。

其次，算法利用修改源代码的方式使得软件水印嵌入在编译后的载体程序中，程序编译过程中产生的符号调试文件在编译后的程序和源代码之间架起了桥梁，从而根据载体程序的内存关系矩阵需要改动的数据可以正确地定位在源代码中实现修改的位置，同时源代码修改编译后的程序也可以正确的生成需要的内存关系矩阵，保证了源代码修改编译后的载体程序正确的嵌入了软件水印。

最后，修改不会影响载体程序的正确性，改变内存分配数量只是调整了内存分配函数中的一个参数，成对增加内存分配操作是通过开发者人工参与的方式进行保证所有分配的内存都会得到正确的释放，而对分配的内容进行修改过程修改的内存内容是在额外分配的内存上进行的，不会影响程序中原有的内存内容，因此软件水印嵌入过程中的源代码修改在编译后得到的程序能够正确运行，不会出现内存问题。

3.4　软件水印的提取

提取算法已知分簇总数 n 和密钥 ω，利用输入 I 从含软件水印的程序 P^W 中提取出软件水印，步骤如下。

1) 输入 I 下运行程序 P^W，并利用密钥 ω 将其分为 n 个簇。

2) 监控程序运行并记录内存关系，得到内存分配矩阵 $\boldsymbol{PR}^W$、内存使用矩阵 $\boldsymbol{PU}^W$。

3) 计算 $\boldsymbol{DW}=\boldsymbol{PR}^W+(\boldsymbol{PU}^W)^{\mathrm{T}}$ 将 $\boldsymbol{PR}^W$、$\boldsymbol{PU}^W$ 合并为 n 阶方阵 $\boldsymbol{DW}=\{d_{ij}\}_{n\times n}$，对 $\boldsymbol{DW}$ 中所有元素利用下述计算式寻找最近的量化点。

$$w_{ij}=\arg\min_{k\in\{0,1\}}|Q_k(d_{ij})-d_{ij}|$$

由 w_{ij} 组成的矩阵记作 $\boldsymbol{W}'=\{w_{ij}\}_{n\times n}$。

4)利用密钥 ω 对 $\boldsymbol{W}'$ 进行逆置乱变换得到的矩阵设为 $\boldsymbol{W}$，以 $\boldsymbol{W}$ 中每个元素作为像素得到的二值图像就是要提取的软件水印。

4　算法评估

评价一个软件水印算法可以从平台依赖性、可信性、数据率、隐蔽性、抗攻击能力等方面进行。平台依赖性衡量算法跨平台使用的能力，可信性表明检测到的软件水印能否有效证明用户版权。数据率衡量程序能嵌入的软件水印容量。隐蔽性反映嵌入软件水印前后载体程序的属性差异。开销衡量算法应用时需要花费的代价。抗攻击能力评价软件水印算法在各种攻击下能够正确提取软件水印的能力。

对软件水印算法的评价结合实验分析的方式进行，实验基于 3 个 Windows 下的开源 C 语言程序：Filemon、BraveFable 和 VLC。 Filemon 是文件系统监视程序，源程序共 2 个 c 文件（不包括驱动程序），程序使用的内存大部分是静态分配的，源文件中的动态内存分配函数和释放函数较少；BraveFable 是小型的 ARPG 游戏，有 42 个 c 文件，程序中频繁的进行内存分配和释放，但是大部分分配函数和释放函数在代码中的位置相距不远；VLC 是多媒体播放器，共 94 个 c 文件，其中的内存分配释放操作较多但不很集中。3 个程序编译后的程序文件（包括可执行文件和动态链接库）大小和在选定的秘密输入下运行时间如表 2 所示。实验中分簇方式选择按时间划分方式。

表 2　实验用到的 3 个程序的文件大小和运行时间 (实验环境：Pentium(R) 4 CPU 2.40GHz 1G RAM WinXp)

测试程序	文件大小/byte	运行时间/s
FileMon	98 304	614
BraveFable	1 900 875	1 809
VLC	4 393 984	3 265

4.1　平台依赖性

软件水印的隐藏是通过修改 C 或 C++等高级语言编写的源代码实现的，而源代码最终通过编译器编译成本地代码程序，软件水印嵌入和提取过程不需要考虑具体的 CPU 和其他硬件，因此算法实现不依赖于特定的硬件环境。

C 或 C++等语言编写的程序中内存操作最终仍然是使用操作系统提供的内存管理接口实现的，同时很多高级语言编写的程序在实现内存操作时常常也不使用编程语言提供的标准库函数或关键字，而是直接调用了操作系统提供的编程接口。在不同的操作系统中，内存管理使用的编程接口并不尽相同。此时算法在收集内存块信息时需要相应的根据操作系统平台的不同拦截不同的内存函数。除此之外，修改源代码时也要和源代码原用的内存操作方式保持一致。

在软件水印算法实现过程中使用的 IDA Pro 软件在 Windows 和 Linux 下都有对应的版本，而且这些版本提供的功能差别不大，同时 Python 本身具有跨平台运行的能力，从而本文算法能够以较小的代价在不同的操作系统间进行移植。

4.2 可信性和数据率

本文给出的软件水印算法中的软件水印是以二值图像的形式给出的，而图像具有一定的视觉冗余，即使软件水印被部分篡改仍可以正确识别。同时由于软件水印图像置乱后遍布在载体程序的内存关系中，很难被全部破坏。下文中抗攻击分析也说明破坏软件水印使它无法识别是非常困难的，因此算法有着较高的可信性。

从算法的具体实现过程可知，一个程序能嵌入的软件水印数据量取决于分簇的数量，n 簇可以嵌入 n^2bit 数据，平均每簇可以嵌入 $n$bit 软件水印数据，显然分簇数量越多越有效率。然而程序中已有的内存分配释放函数总数固定的情况下，分簇过多必然导致需要在程序中插入更多内存相关代码，这会导致文件大小增加。分簇需要在对程序中的原有内存关系的个数与分布进行分析的基础上确定最优分簇个数。

4.3 隐蔽性和开销

内存申请、释放和读写是程序中常见的操作，适度的添加这些内存操作并不会引起用户怀疑，并且算法嵌入过程中增加分配的内存大部分会得到使用，不存在明显的异常特征。只有当较小的程序嵌入较大的软件水印时，增加的代码过多才容易被觉察。而软件保护的对象通常具有较高的应用价值，相对比较复杂，程序体积一般较大，因此本软件水印算法有着较好的隐蔽性。

图 2 给出了嵌入不同大小的软件水印后对载体程序大小的影响，图 3 给出了嵌入不同大小的软件水印后对程序运行时间的影响。嵌入不同大小的软件水印前后程序大小和时间增加部分所占的百分比，由下式给出

$$\frac{M(P^W)-M(P)}{M(P)}\times 100\%$$

其中，P 和 P^W 分别表示嵌入软件水印前后的程序，M 在图 2 和图 3 中分别表示程序大小的度量和在给定输入下程序的运行时间的度量。

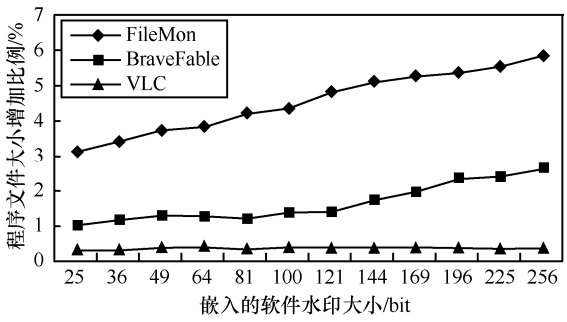


图 2 软件水印对程序大小的影响

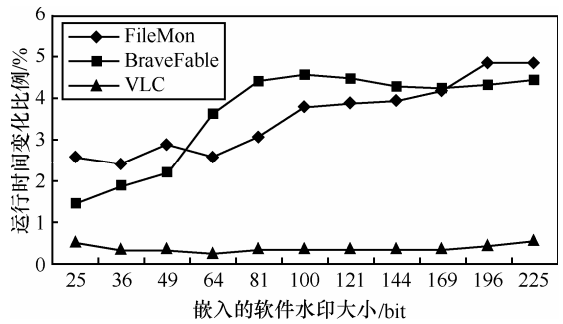


图 3 软件水印对程序运行时间的影响

由图 2 可以看出，程序能嵌入的最佳软件水印大小取决于载体程序自身的性质。当嵌入的软件水印较大时，会增大程序文件的大小。其中，FileMon 程序文件大小的增长较快是由于需要添加的内存分配释放代码较多造成的，而对于 BraveFable 程序，在嵌入的软件水印较小时程序文件大小的增长不显著，当嵌入的软件水印较大时，程序大小增长明显加快。VLC 程序由于自身的内存操作足够多，较少需要增加太多的额外代码，因而文件大小变化最小。

图 3 可以看出，程序的运行时间随嵌入的数据变化而变化。由于实验中使用的输入和软件水印算法中用到的秘密输入相同，因此嵌入的数据增多时，增加的内存操作代码较多并参与执行，因此会造成时间上的延迟。然而相对于全部运行时间，这点延迟对程序运行的影响不大。实验中的 BravelFable 程序中存在大量的循环，加入的内存读写代码有一部

分落在循环中，这导致了图 3 中 BraveFable 的时间变化比例偏高。由于这些代码只在特定输入下才会全部运行，对于采用其他输入的情况下，插入的代码执行的机率不大，载体程序的运行时间几乎没有影响。

算法的实现过程中由于对于每个分配的内存块，都要分配一块同样大小的影子内存块备份其内容，对内存的消耗偏多。同时在软件水印嵌入时需要开发者参与修改源代码，因此整个算法实现过程中的开销比其他可以自动完成的算法稍高。然而相对于开发者开发程序的长周期来比较，修改程序嵌入软件水印的时间代价较小，在开发者可以接受的范围内。

4.4　抗攻击能力

攻击者往往使用一些攻击手段使得软件水印难以识别，软件水印算法必须能提供一定的抗攻击能力。由于攻击者所掌握的是已嵌入软件水印的本地代码程序，在软件大小日益膨胀的今天，攻击者很难对程序完整的进行反汇编加以分析，因此对程序的全部实现细节了解程度不如开发者。此时攻击者可以利用现成的工具进行常规攻击，或者根据自己对程序的理解修改反汇编后的程序然后编译得到攻击后的程序。下面结合实验给出了不同攻击方式下的评价结果，实验中的软件水印选择 40×40 的二值图像，软件水印载体选择 VLC 程序，分簇采用按时间划分方式。

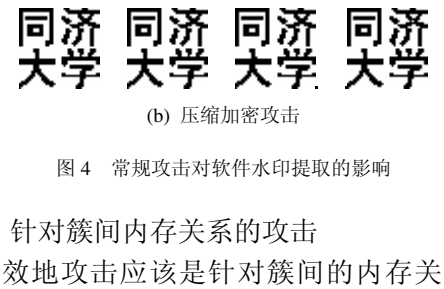
1) 常规攻击

针对本地代码的常规攻击有代码优化、代码变形、压缩加密等方式。代码优化一般通过反编译—编译的方式利用编译器对代码进行优化。代码变形利用等价代码替换的方式改变代码特征。压缩加密通常事先对原始程序压缩或加密，程序执行后使用一小段解压缩和解密程序还原程序然后转到真正的程序入口执行。这些攻击前后的程序对于相同的输入输出保持不变，即这些攻击为语义保持变换攻击。

对于代码优化，由于本文给出的软件水印算法中，插入的代码全部执行，而且这些代码都是必须的，不存在明显冗余之处，代码优化过程不会导致含软件水印程序大小剧烈变化，因此对分簇的影响不大，不会危及软件水印的安全性。对于代码变形攻击，攻击过程中改变的只是代码的形状，程序执行中的实际功能未发生变化，同样不会对簇间内存关系造成影响。而压缩或解密壳一般会在程序的开始部分完成解压缩或解密操作。当第一个簇足够大时或秘密输入控制的程序执行时间足够长时，这些攻击的影响会限制在前几个簇内，而不会对其他部分造成过多影响。

图 4 给出了这几种常规攻击下的实验结果。图 4(a)左侧为原始软件水印图像，然后从左到右依次是对含软件水印的程序用 IDA Pro 反编译后分别使用 TASM、MASM 和 NASM 编译后提取的软件水印图像。图 4(b)左侧为原始图像，然后依次是对含软件水印的程序分别用 UPX 2.02、Aspack 2.12、ASProtect 1.35[16]等压缩加密壳保护后提取的软件水印图像，这些壳或多或少带有代码变形功能。可以看出常规攻击不影响软件水印的正确识别。混合使用上述常规攻击，在加壳操作不超过 3 次的情况下，统计发现攻击前后软件水印图像被修改的像素数依然不超过总像素数的 1%。

(a) 代码优化攻击



(b) 压缩加密攻击

图 4　常规攻击对软件水印提取的影响

2) 针对簇间内存关系的攻击

有效地攻击应该是针对簇间的内存关系进行攻击，最直观的方式是插入冗余代码使得分簇不正确，对于按空间分簇方式，需要插入大量的冗余代码，这会使得程序体积剧烈膨胀。对于按时间分簇方式，可以插入延时函数或循环代码，由于算法选择的秘密输入控制下程序运行时间较长，插入的代码也需要达到较长的延时才能分簇产生影响，由于攻击者不知道算法选择的秘密输入，所以插入的代码必然严重影响用户使用感受，容易引起怀疑而加以去除。而在按输入分簇的情况下攻击对分簇基本没有影响。在无法有效影响分簇的情况下，攻击方式有成对添加内存分配和释放代码、更改分配内存数量、随机修改内存等。

① 添加内存分配释放代码

在插入分配内存的代码后，紧接着插入内存释放代码和在距离内存分配代码最远距离处（不会产生内存问题的前提下）插入内存释放代码 2 种情况

下的实验结果如图 5(a)和图 5(b)所示。图中最左图为原图，从左到右依次是 40%、70%、100%的簇中随机添加内存分配函数和对应的释放函数后的实验结果。如图 5(a)所示，当添加的分配和释放代码相距较近时，分簇的时候有非常大的概率将它们划分在同一簇或相邻簇内，此时对应的是内存关系矩阵中的少量元素被修改，提取的软件水印图像基本不发生变化，因此对软件水印的正确识别影响轻微。图 5(b)中采用的攻击方式对程序的危害较大。然而对攻击者来说，在对程序的细节不清楚的情况下，若添加的内存分配和释放代码如果相隔太远，在不同的输入下分配的内存能否得到释放很难明确判断，容易导致内存泄漏或其他内存问题。因此图 5(b)中的攻击方式不容易在实际情况下应用。

(a) 添加近距离内存分配释放代码

(b) 添加远距离内存分配释放代码

(c) 更改内存分配数量

图 5 不同强度下针对内存关系的攻击对提取的影响

② 更改分配内存数量

改变每次分配的内存数量，是较容易实现的攻击方式，对应的是内存分配矩阵发生改变。图 5(c)给出了对代码中的每一处内存分配数量增加一个随机值时的实验结果，图 5(c)最左边是软件水印原图，然后从左到右依次是 40%、70%、100%的内存分配函数中分配的内存数量改变后提取的软件水印图像。可以看出算法中使用的量化技术对这种攻击有着良好的抵抗能力。

③ 随机修改内存

在不清楚一块内存是否还会使用之前就写入是危险的事情。为了不对程序的正确性造成损害，可以采用 2 种方式写入。第一种方式是在修改内存数据后紧接着还原内存数据，此时改动发生在同一个簇内，显然对整个软件水印没有影响。第二种方式是先为要写入的内存分配一段内存，接着在分配的额外内存中进行写入。这种方式实际上是添加内存分配释放代码和随机修改内存两种攻击的结合，其缺陷和前面分析类似，若代码过于集中，它对整个软件水印的影响轻微，若代码过于分散，由于不完全清楚程序在不同输入下的执行细节，则难以保证内存一定能得到释放，并且在输入不确定的情况下分配的内存在修改的时候是否已释放并重分配不能明确判断，存在内存溢出的风险。

4.5 和同类算法的比较

从上述分析可以看出，本文提出的算法能够充分利用程序中已有的代码，不会增加大量代码，对载体程序的影响较小，并提供了较强的抗攻击能力。

和动态路径算法和混沌软件水印算法相比，在隐蔽性方面，动态路径算法大量重复性的调用同一个函数，因此可以通过简单的静态分析就可以定位并加以攻击。混沌水印算法更是在载体程序中加入了输入监控模块、水印解码模块等大量特征鲜明的代码，显著增大了载体程序文件大小，延长了程序执行时间，而本文给出的算法实现中增加或修改的代码和原始软件中的代码之间不存在明显区别，在不知道原始程序的情况下，很难定位代码的改动位置，具有更好的隐蔽性。在抗攻击能力方面，动态路径算法由于内置了修改检测功能从而具有较强的抗攻击能力，然而算法中用到的函数可以被替换从而绕开防护机制，文献[15]就提出了一种低成本的攻击方式对嵌入的软件水印进行修改替换。混沌水印算法中通过反逆向工程模块来保证软件水印免受逆向工程攻击，然而不存在任何一种方法完美抵抗逆向工程。使用逆向工程方法只需要在载体程序代码恢复时转储程序，即可完全绕过混沌水印算法中所施加的所有保护。而本文给出的算法由于良好的隐蔽性，逆向工程攻击难以发现需要攻击的内容。本文算法不仅可以抵抗软件传输过程中会遇到的各种常规变换，即使使用的内存操作被修改时仍然能够较好的识别软件水印。

比较可以看出，其他面向本地代码的软件水印算法多是从代码不可篡改的角度入手来保证软件水印的安全，这种做法一方面难度较大，另一方面导致软件水印的隐蔽性变的更差，更容易引起攻击者怀疑。而本文给出的算法不使用额外的代码防篡改措施，只通过使用载体程序自身存在的约束来保证软件水印的安全，同时保证了软件水印的隐蔽性而不容易受到攻击者的注意，进一步增强了软件水印的安全。

5　结束语

本文提出了一种新的基于内存操作的软件水印算法，给出软件水印的嵌入算法和提取算法的详细过程。实验表明，本文提出的利用程序中的内存操作嵌入软件水印的算法除了隐蔽性好，对各种攻击有着较好的抵抗能力之外，而且实现简单，对原程序的影响较小。目前给出的算法仍需要改进，如内存关系信息难以用矩阵全部表示，而且矩阵本身的表示效率不够高，对这些问题的进一步的研究将是以后工作的一个方向。

参考文献：

[1] http://portal.bsa.org/globalpiracy2011/index.html[EB/OL]. 2012.

[2] COLLBERG C, THOMBORSON C. Software watermarking: models and dynamic embedding[A]. Proceedings of Symposium on Principles of Programming Languages[C]. New York, USA, 1999. 311-324.

[3] 张立和，杨义先，钮心忻等. 软件水印综述[J].软件学报，2003, 14(2): 268-277.
ZHANG L H, YANG Y X, NIU X X, *et al*. A survey on software watermarking[J]. Journal of Software, 2003, 14(2): 268-277.

[4] COLLBERG C, THOMBORSON C, TOWNSEND G. Dynamic graph-based software fingerprinting[J]. ACM Transactions on Programming Languages and Systems, 2007, 29(6): 35-67.

[5] KAMELA I, ALBLUWIB Q. A robust software watermarking for copyright protection[J]. Computers & Security, 2009,28(6): 395-409.

[6] CHEN X J, FANG D Y, SHEN J B. A dynamic graph watermark scheme of tamper resistance[A]. Proceedings of 5th International Conference on Information Assurance and Security[C]. Xi'an, China, 2009. 3-6.

[7] COLLBERG C, CARTER E. Dynamic path-based software watermarking[A]. Proceedings of the ACM Conference on Programming Language Design and Implementation[C]. Washington, CD, USA, 2004. 107-118.

[8] LARKIN A J, BALADO F, HURLEY N J. Dither modulation watermarking of dynamic memory traces[A]. Proceedings of Information Hiding[C]. Berlin, Germany, 2005. 372-386.

[9] GUPTA G, PIEPRZYK J. Source code watermarking based on function dependency oriented sequencing[A]. Proceedings of the International Conference on Intelligent Information Hiding and Multimedia Signal Processing[C]. Harbin, China, 2008.965-968.

[10] NAGRA J, THOMBORSON C. Threading software watermarks[A]. Proceedings of 6th International Workshop on Information Hiding[C]. Toronto, Canada, 2004, 3200: 208-233.

[11] MYLES G, JIN H. Self-validating branch-based software watermarking[A]. Proceedings of Information Hiding[C]. 2005. 342-356.

[12] 芦斌,罗向阳,刘粉林.一种基于混沌的软件水印算法框架及实现[J]. 软件学报, 2007, 18(2):351-360.
LU B, LUO X Y, LIU F L. A chaos-based rramework and implementation for software watermarking algorithm[J]. Journal of Software, 2007, 18(2):351-360.

[13] The IDA Pro Book[M]. San Francisco: No Starch Press, 2008.

[14] Gray Hat Python[M]. San Francisco: No Starch Press, 2009.

[15] SHI Y Q, JEON B. A low-cost attack on branch-based software watermarking schemes[A]. Proceedings of the 5th International Workshop on Digital Watermarking[C]. Jeju Island, Korea, 2006. 282-293

[16] http://www.pediy.com/tools/packers.htm[EB/OL]. 2012.

作者简介：

许金超（1982-），男，山东临沂人，同济大学博士生，主要研究方向为软件保护、信息安全。

曾国荪（1964-），男，江西吉安人，同济大学教授，主要研究方向为可信软件、信息安全。