第42卷 第8期 电 子 与 信 息 学 报 Vol. 42No. 8
2020年8月 Journal of Electronics & Information Technology Aug. 2020

一种基于程序执行时间量化分析的软件水印方法

张颖君[①] 陈 恺[②] 鲍旭华*[③]

[①](中国科学院软件研究所可信计算与信息保障实验室 北京 100190)

[②](中国科学院信息工程研究所信息安全国家重点实验室 北京 100093)

[③](网神信息技术(北京)股份有限公司 北京 100015)

摘 要：当前，应用软件面临的重要问题是不法分子通过软件剽窃、重打包等技术，将恶意负载或广告加载到合法应用软件中，并形成新软件进行发布，给用户和应用软件作者的合法权益带来威胁。为了实现对应用软件剽窃、重打包等安全风险的测评，该文提出一种基于程序执行时间量化分析的软件水印方法(SW_PET)。通过生成多种相互抵消功能的操作组，实现对水印信息的时间化编码，并植入应用软件中；在检测过程中，需要提取相应的水印信息，对照之前的时间编码对应的原始水印，比较不同操作的执行时间，判断水印相似度，进而判别原始水印的存在性，完成应用软件合法性的判断。该方法也可以与其它类型的水印信息相结合，增强水印的鲁棒性。最后，通过搭建仿真模拟器，实现对不同应用软件水印信息的比较和判断，验证该方法的有效性。

关键词：软件水印；程序执行；水印编码

中图分类号：TP309 **文献标识码**：A **文章编号**：1009-5896(2020)08-1811-09

DOI: 10.11999/JEIT190850

A Software Watermarking Method Based on Program Execution Time

ZHANG Yingjun[①] CHEN Kai[②] BAO Xuhua[③]

[①](*Trusted Computing and Information Assureance Laboratory*, *Institute of Software*, *Chinese Academy of Sciences*, *Beijing* 100190, *China*)

[②](*State Key Laboratory of Information Security*, *Institute of Information Engineering*, *Chinese Academy of Sciences*, *Beijing* 100093, *China*)

[③](*Legendsec Information Technology*(*Beijing*) *Inc*, *Beijing* 100015, *China*)

Abstract: Currently, a main problem in software is repackaging or plagiarization, which means attackers can add malicious payloads or advertisements into legitimate APPs through piggybacking, it greatly threatens the users and original developers. In this paper, a novel Software Watermarking method based on Program Execution Time (SW_PET) is proposed. By generating a variety of effect-canceling operations, the watermark information can be encoded into the form of program execution time, and can be embedded into Android APPs. In the detection process, the watermark information is extracted and compared with the original watermark to check whether the APP is repackaged. This method can be combined with other types of watermarks (e.g., picture-based watermarks) in order to enhance the robustness. Finally, the effectiveness of the proposed approach is verified, and the overhead introduced by the watermark is measured, which is demonstrated to be minimal.

Key words: Software watermarking; Program execution; Watermark encoding

收稿日期：2019-11-01；改回日期：2020-03-20；网络出版：2020-07-08

*通信作者： 鲍旭华 xuhua.bao@outlook.com

基金项目：国家重点研发计划项目(2016QY04W0805)，国家自然科学基金(U1836211)，大数据协同安全国家工程实验室开放课题

Foundation Items: The National Key Research and Development Program of China(2016QY04W0805), The National Natural Science Foundation of China (U1836211), The Open Project of National Engineering Laboratory of Big Data Collaborative Security

1 引言

随着互联网的不断发展，各种应用程序也呈爆发式增长，应用软件已逐渐成为人们日常生活中不可获缺的工具。仅在移动智能终端方面，2018年全球应用软件下载量超过了1940亿次[1]。丰富多彩的应用程序(APP)在给用户带来便利的同时，也吸引了众多的不怀好意的人甚至不法分子的注意。应用软件面临多种威胁，潜在危险软件很多都是通过重打包技术生成的，即添加恶意负载或广告给合法的应用软件并形成新的重打包软件，从而危害原始作者的利益。重打包[2]通常是在原始代码中加入新的代码，并重新编译形成一个新的应用软件。据统计[3]，安卓市场中5%～13%的应用软件为重打包软件；在移动应用软件TOP 100下载软件中，苹果ISO的92个，安卓前100个全部被进行重打包[4]。由于重打包软件是一种相对隐藏的攻击方式，普通用户很难区分合法软件和重打包的软件。因此，迫切需要提出一种对重打包软件进行检测的方法，保护用户和应用软件作者的合法权益。

目前，针对软件重打包检测的方法主要通过检测软件间的相似性进行判断。在相似性检测中，主要包括基于哈希的方法[5]进行重打包快速检测、基于质心的快速检测方法[6]、基于图的快速检测方法[7]等。但这些方法多是针对源码存在且不混淆的情况进行分析。但当恶意开发人员使用代码混淆或者保护工具(如Proguard[8], DexGuard[9]等)将代码进行混淆或者变换(如对控制流和数据流进行转换)，则会增加逆向分析的难度，使现有技术难于快速检测出重打包软件。

数字水印是永久镶嵌在其他数据中具有可鉴别性的数字信号或模式[10]，在应用软件中加入数字水印可以有效解决这个问题。通过检验水印的存在与否能够鉴别出非法复制和盗用的相关软件[11]，对重打包软件进行有效判定。现有的水印方法主要分为静态水印和动态水印。静态水印[12]主要是将水印植入到可执行程序的代码或数据中，其提取过程不需要运行程序，通过静态分析完成识别或提取，主要分为代码替换法[13]、静态图法[14]和抽象解释法[15]等。由于静态水印提取水印不需要执行代码，破坏静态水印相对较为容易。动态水印是将水印植入到程序的执行过程或运行状态中，即根据程序某个时刻运行的状态进行信息的编码，主要包括基于线程[16]、基于图[17]、基于路径[18,19]、基于权限[20,21]等水印技术。但是由于现有的水印技术大多生成过程较为复杂，且抗混淆能力较差，因此，本文提出一种基于程序执行时间量化分析的软件水印方法SW-PET。考虑到水印若以代码形式存在，则更易与其他程序代码结合，不易被水印检测方法发现，易于隐藏；同时，代码的执行时间具有一定区分度，因此，本文考虑将代码执行时间作为水印判断的依据加入原始程序。通过将给定水印信息进行时序化编码，并与程序代码相结合，当需要对水印进行提取与验证时，通过观察程序特定语句的执行时间来进行判断。值得注意的是，本文所述方法为通用方法，在PC平台和移动平台均可使用。在无源码的情况下，也可直接在逆向后的代码(如二进制代码或者字节码)中加入水印代码。论文的主要贡献包括：

(1) 首先，论文提出了一种水印时间化编码方法SW_PET。该方法中嵌入程序中的水印不再是具体信息，而是涉及基本操作的简单操作组代码。这种水印生成方法相对于明文的静态水印或者加密后的无意义数据来说，具有更好的隐蔽性，且添加过程简单易操作。

(2) 其次，该方法具有较高的鲁棒性。目前的水印以程序部分代码的执行时间作为水印，不易在程序中改变该执行时间。即使被偶然加入的混淆代码改动部分水印内容，仍然有其他部分代码水印可供检测。

(3) 最后，该水印生成方法可以与其它类型水印编码方法相结合，例如与基于图像ASCII码的方法结合，该方法的水印不易破坏。多方法结合后的水印具有更高鲁棒性等优点。

2 主要流程

基于程序执行时间量化分析的水印方法(SW_PET)可以用于应用软件的合法性检测，主要是利用程序执行过程中操作组时间量化分析进行水印代码的生成，该方法可以与其他水印方法相结合，完成对程序的版权保护。主要流程如图1所示：首先，需要定义相互抵消功能的操作组(Effect-Canceling Operations, ECO)，用于水印编码。在编码过程中，为了增加水印的多样性，可以生成多种相互抵消功能的操作组。由于软件代码存在多种功能相互抵消的操作，用其对相应水印信息进行编码后，可以使得水印多样，不易被攻击者猜解。这类编码植入目标程序的某个特定的函数或者部分，因此需要一定的输入才能触发(仅编码植入者知晓)。在水印验证阶段，需要将待检测的软件通过特定的输入选取函数后，对照之前的编码方式插入相关的时间函数，并对水印相关代码的执行时间进行计算；计算结果通过水印相似度比较得到最终判断结果。该水印方法可以与其他类型水印信息相结合，增强水印的鲁棒性。

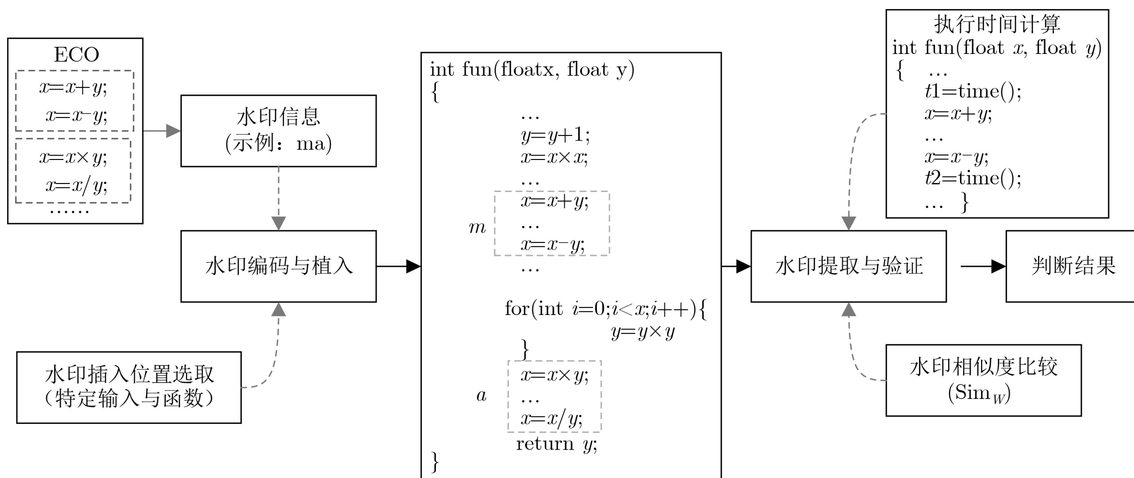

图 1 基于程序执行时间量化分析的水印方法流程图

2.1 准备工作

准备工作主要是对程序执行时间量化分析，也是水印编码与验证的基础。主要包括对指令运行时间的估算和针对多组操作运行时间差异计算两部分。

在指令运行时间估算中，主要对备选指令进行分析，获取其运行时间。具体来说，我们需要对指令的运行时间进行估算，包括源代码以及对应的汇编代码的时间。主要操作类型如表1所示，其中，C表示常数。

通过对指令运行时间的估算，可以为后续的水印编码等步骤提供支撑。具体来说，可以设计一组相互抵消功能的操作组，如$x = x + y$; $x = x - y$。这部分操作可以放入原始程序中，但不会影响原始程序本身的执行功能。观察这个操作组运行时间并且记录。若这部分执行时间相对于其他代码执行时间相对较为固定，则可以作为水印进行嵌入。在相同执行环境下，其运行时间较为稳定。在不同执行环境下(如不同的CPU)，则执行时间会发生变化。针对这类情况，我们不使用时间的绝对值进行水印的比较。相对地，我们使用这部分代码对于其他代码的相对量进行水印的判断。即使在不同的执行环境下，这部分时间相对于其他代码段的执行时间较为稳定。我们可以将其作为一个基本的水印构造单元用于水印的编码。

具体水印编码构造过程中(图1)，为了增加水印的多样性，可以生成多种相互抵消功能的操作组(ECO)。之后随机选取ECO，将其所需时间记录。当多个操作组组合后，其运行时间会有差异。利用该差异可以作为信息的区分。考虑到水印中的信息有所不同，这些ECO可以提供这些不同种类的信息，即为水印嵌入过程中的水印编码提供服务。本文以水印“mark”为例进行说明：假设要植入的信息为mark，则包括4个字母。我们对每个字母进行编码，则上述操作应该提供至少4个不同时间的操作序列。例如字母m可以随机从表1中选取几个ECO操作，如$x = x + y$; $x = x - y$。对于其他几个字母，可以使用表2中的编码方法。

例如在Intel Core i5-8265U机器上搭建模拟器，对上述例子进行测试，加法操作所需要的时间为2.15×10^{-6} s(参见3.1节)；减法操作所需的时间为2.15×10^{-6} s，则利用这类信息可以简单有效地提供不同编码方式，从而植入水印。

表 1　主要操作类型表

操作类型	操作	示例
算术操作	加法	$x = x + y$, $x = x +$ C，等
	减法	$x = x - y$, $x = x -$ C，等
	乘法	$x = x \times y$, $x = x \times$ C，等
	除法	$x = x / y$, $x = x /$ C，等
逻辑操作	逻辑与	$x = x \&\& y$, $x = x \&\&$ C，等
	逻辑或	$x = x \| y$, $x = x \|$ C，等
	逻辑非	$x = !x$
比特位操作	比特and操作	$x = x \& y$, $x = x \&$ C，等
	比特or操作	$x = x \mid y$, $x = x \mid$ C，等
移位操作	左移位	$x = x << y$, $x = x <<$ C，等
	右移位	$x = x >> y$, $x = x >>$ C，等

表 2　mark的时间编码序列

水印字母	操作代码
m	$x = x + y$; $x = x - y$
a	$x = x \times y$; $x = x / y$
r	$x = x + y$; $x = x - y$; $x = x \times y$; $x = x / y$
k	$y = x$; $x = (x >> 2)$; $y = x \& 0 \times 3$; $x = (x << 2) \mid y$

2.2 水印编码与植入

将水印信息进行时间化编码，主要分为3类编码方式：与程序无关编码，程序变量相关编码和程序代码相关编码。第1种相对简单，可以直接植入程序，但通过细致的数据流分析可能发现这部分代码与程序无关，可能相对较为容易去除；第2种利用原始程序程序变量进行编码，相对复杂，但与原始程序结合度高，不易被数据流分析发现是额外添加的代码，因此也较难去除；最后一种是利用原始程序中部分代码语句作为水印一部分，可以减少构建ECO的代码量，更难被发现。

(1) 与程序无关编码：具体实现中，与程序无关的编码主要是根据给定信息(如“mark”)，根据其字母序进行编码。每个字母可以对应一组需要不同时间的操作组，只要这类操作组数量超过所需要的字母数量即可。例如字母“m”可以对应“$x = x + 3; x = x - 3$”，字母“a”对应“$x = x \times 5; x = x / 5$”。因此上述信息“mark”可以对应操作组序列。这类方法通过细致的数据流分析，可以发现与原始程序没有关联，存在被去除的可能性。

(2) 程序变量相关编码：第2类与程序相关的编码主要是将水印编码信息作为代码编入程序，其中水印编码用到原始程序变量等。但需要注意，该类操作不能影响原始程序的操作。相对较为简单的方法是利用原始程序中的变量，这样不易被静态分析方法发现额外变量的使用。图2是程序添加后水印的代码(水印代码由灰色底颜色标识)。

```
int fun(float x, float y)
{
        …
        y=y+1;
        x=x×x;
        …
        x=x+y;
        …
        x=x−y;
        …
        for (int i=0;i<x; i++){
                y=y×y;
        }
        x=x×y;
        …
        x=x/y;
        return y;
}
```

图 2 程序相关编码示例

可以看到，添加的水印代码(如$x = x + y$)利用到了原始程序的输入(x和y)，因此与源程序结合相对紧密。攻击者难以通过数据流依赖关系的分析并区分水印代码与原始程序代码，增加了攻击者区分水印代码与原始代码的难度。其次，添加的水印代码操作过程中，不应产生额外的副作用。对于代码本身，使用$x = x + y$与$x = x - y$，这两个操作保证了操作过程不影响x和y本身的值。其次，还需要考虑添加入的水印代码与原始程序代码之间的位置关系也不能影响原始程序的执行。例如原始代码中有“$y = y + 1;$”，不能将该类代码加入水印代码中间(如$x = x + y$与$x = x - y$)，否则会改变原始x的值。为了实现这一过程，我们有如下具体计算方法：

S1: 在水印代码中，确认好原始程序不变量，即不应该修改的原始程序变量。上例中，水印代码的原始程序不变量为$<x, y>$。

S2：确认依赖变量。若依赖变量的值发生改变，则水印代码中的原始程序不变量会发生改变。在上述例子中，对于第1条水印代码语句W_1，x依赖于y及其自身(即x)，则其依赖变量(或称为引用变量)use(W_1)={x, y}；式中x, y即为W_1引用的变量。对于第2条水印代码语句则有use(W_2)={x, y}。其中use(W_2)表示第2条水印语句使用的变量(即x和y)。

S3：假设夹在水印代码中的原始程序代码为P，则P中定义的变量不应被后续的水印代码使用。例如水印代码为“$x = x + y; x = x - y$”，则不应该在其中夹入定义y的代码(如$y = y + 1$)，否则水印代码最后一条语句$x = x - y$将会改变x的值，使之与水印代码前的状态不同。此外，也不应该在水印代码中间夹入使用x的原始程序代码。如假设原始程序中有语句定义某新变量$u = x + 1$，则该语句不应夹入两条水印代码之间，否则u的值会和植入水印代码前不同。因此，简单来说，P中使用的变量use(P)不应与水印代码中改变的变量有所交集(def(W_1))；P中改变的变量def(P)不应与水印代码中第2条语句使用的变量(use(W_2))有所交集。

使用如上方法即可安全地把水印语句植入原始程序，即不影响原始程序的功能。

(3) 利用原始程序进行编码：我们可以进一步利用原始程序中部分代码作为水印的一部分，即利用原始程序进行相关编码。其基本思想是结合程序原本的运行时间与我们提供的相互抵消功能操作组ECO。利用这种方式，不需要构建很多ECO操作，部分ECO的过程可以使用原始程序，使得水印代码与原始程序结合性更高。具体方法如下：对于给定程序，给定一个输入，其程序中每个步骤的运行时间基本确定(需要去除网络或者外部输入操作语句，参见本小节最后部分)。例如对于如下程序，给定输入x，其中函数for循环和函数调用

sub_fun()的运行时间基本确定(不含有依赖于外部输入或者网络等操作)。如需测算该时间，我们可以运行该程序多次(如100次)取其平均值，以确保其他环境因素对运行时间的影响最小化。这样可以在程序中设计观测点分别获取4个时刻的时间$t1$, $t2$, $t3$, $t4$(如图3)，即可对输入x在程序执行的路径中不同部分进行测算时间。例如，对于循环操作的时间为$\Delta t1 = t2 - t1$；对于sub_fun()操作的时间为$\Delta t2 = t4 - t3$。该类方法与第1类方法相比，与程序的结合更加紧密，也更难以被发现是水印代码。另一方面，可以利用原有程序的部分执行代码构建ECO，因此也可以减少需要构建ECO的代码量。

以上程序本文选择了2组观测点，分别观察了两段代码的执行时间。根据植入信息量的大小，可以在程序中设计不同数量的观测点，从而匹配所需的信息。例如若需要植入“mark”信息，则需要4组观测点。在发布程序的过程中，无需将带有time()信息的观测点发布出去(因其会暴露出水印代码的位置)。

仅从程序中获取代码的执行时间，有可能无法满足植入任意信息的需求，即从程序中获取的时间序列与植入信息并不相同。如上述$\Delta t1$不等于水印信息m对应的代码序列所需时间。因此需要将原始程序进行微调，加入相关的ECO。具体来说，对于给定的一个程序P和输入x，获取的原本时间序列为ori=$<\Delta t_1, \Delta t_2, \cdots, \Delta t_n>$，所需植入的信息为des=$<m_1, m_2, \cdots, m_n>$，则计算差值为Δs=des-ori=$<m_1, m_2, \cdots, m_n>-<\Delta t_1, \Delta t_2, \cdots, \Delta t_n>=<m_1-\Delta t_1, m_2-\Delta t_2, \cdots, m_n-\Delta t_n>$。因此仅需从相关的ECO集合中找到合适时间对应的代码指令即可。例如上述代码中第1个水印信息匹配的ECO序列对应的时间为$m_1-\Delta t_1$。很容易找到相应的水印代码序列，并使用前述水印植入算法将其植入程序完成水印的植入。同理，对于上述操作，仍然不应该影响原始程序的执行。

```
int fun(int x, int y)
{
        ...
        y=y+1;
        x=x×x;
        ...
    t1=time();
        for (int i= 0; i< x; i++){
                y=y×y
        }
    t2=time()
        ...
    t3=time()
        sub_fun(x, y);
    t4=time()
        ...
        return y;
}
```

图 3 使用原始程序中代码作为水印示例

值得注意的是，若插入的水印代码的ECO间有执行时间不确定的语句，例如输入输出语句或者网络操作的函数等，则会影响水印的准确性。因此，需要对水印代码之间的语句进行分析，避免该类型语句存在于水印代码ECO之间。具体过滤过程可以通过构建输入输出语句等的白名单进行过滤。对于某类程序设计语言(例如C/C++, Python, Java等)，本文将其执行时间不确定的语句标记在字典D中，则可通过字典中语句与程序语句比对的方法确认当前水印代码ECO间是否包括执行时间不确定的语句。例如对于Python程序，input()与print()即为典型输入输出语句，其执行时间不确定(即input$\in D$, print$\in D$)，则在相互抵消功能的操作组ECO之间不应含有该类语句。另一方面，考虑到程序调用语句可能包括D中的语句，因此需要对程序调用函数进行分析，确认其不能含有D中语句。本文使用集合Ca表示程序调用语句且被调用的函数含有D(为了简单，我们仅做1层调用分析，不再分析被调用的函数再调用其他的函数)。典型地，本文选取程序中的执行路径Pa=$\{s_1, s_2, \cdots, s_n\}$，即在Pa中包括原始程序语句$s_1, s_2, \cdots, s_n$。对于相互抵消功能的操作组ECO=$<W_1, W_2>$以及任意$s_i \in$ Pa $\wedge W_1 <_c s_i <_c W_2$，需要满足$s_i \notin D \cup$ Ca。其中$<_c$表示语句顺序，$W_1 <_c s_i <_c W_2$表示程序语句s_i是W_1和W_2之间的语句。通过本文方法，可以选择合适的语句进行水印代码的加入。注意在多个不同的ECO中间，可以有D或者Ca中的语句存在，因其不影响ECO的执行时间。

2.3 水印提取与验证

根据水印拥有者提供的信息，对照之前的时间编码方式提取出水印信息。获取原始的水印后，比较不同操作的执行时间，进而判断水印相似度。具体来说，在原始水印的位置插入提取时间代码time()，从而能够获取每个执行部分代码的时间。该部分操作是源代码无关的：如果嵌入水印过程是在源代码上进行嵌入，则提取过程可以在源代码相应位置加入time()操作提取信息；如果嵌入过程是在原始程序编译后的二进制代码上进行，则提取过程也可以在后续的代码上进行。对于触发的代码，可以与水印嵌入时选择一样的输入。例如对于某个PDF应用软件进行水印保护，则水印嵌入时选定的PDF文件即可作为水印提取与验证时的输入文件。

如图3所示，图中time()即为加入的代码。通过与前期设计过程中的代码段时间对照表，可以获取其具体水印字符，即可提取水印。

考虑到执行时间在多次操作中可能会存在不同，因此，本文定义水印相似度Sim。当相似度达到一定程度，我们即可认为水印被提取。类似于传统的水印技术，即使不完全一致，也可认为水印被成功提取。我们先定义字符相似度$\mathrm{Sim_C}(l_1, l_2)=|t_{l1} - t_{l2}|$，其中$l_1$和$l_2$分别表示两个字符(字母)，$t_{l1}$和$t_{l2}$分别表示其执行的时间。考虑到一些随机因素的影响，可以容忍一些微小时间的变化。即：若两个字母对应的执行时间t_{l1}和t_{l2}接近(即$\mathrm{Sim}<=\theta$)，则我们认为两个字符相同。之后可以定义水印相似度Sim_W。假定提取中的程序水印为W_s，预期水印为W_o，则

$$\mathrm{Sim}_W = \frac{\sum_{i=1}^{|W_o|} \mathrm{Jud}(\mathrm{Sim_C}(W_{oi} - W_{si}))}{|W_O|}$$

其中$\mathrm{Jud}(x) = \begin{cases} 1, x \ge \theta \\ 0, x < \theta \end{cases}$，该函数$x$是否大于某个参数$\theta$：若大于则为1；否则为0。$\mathrm{Sim}_W$可以判断$W_s$和$W_o$两者相同的字符个数，若两者完全相同，则$\mathrm{Sim}_W$= 1；若不同，则$0 <= \mathrm{Sim}_W < 1$。利用该方法可以判断水印是否被修改。在实际使用过程中，可以定义参数λ，若$\mathrm{Sim}_W >= \lambda$，则认为水印被提取；若$\mathrm{Sim}_W < \lambda$，则认为水印失效。

2.4 其它类型的水印信息植入

考虑到可以使用鲁棒性更强的算法配合本文方法以增加鲁棒性。本文这里以ASCII图片水印[22]为例，可以植入该类型的水印代码，该嵌入方法利用视觉效应可以容忍代码上产生的变化导致运行时间变化。对于其他种类水印的选择，可以选择不同编码方式的水印，例如ASCII图片水印将原始水印的图片形式变为二进制形式进行水印植入，具有高鲁棒性等优点。具体地，当字符的水印被转换为多个ASCII代码后，增加了冗余量。该类冗余量可以增加水印代码的鲁棒性。之后，将对应ASCII代码逐一通过本文方法转换为ECO，即可实现水印的植入。在提取过程中，通过本文方法，提取出对应的ASCII代码，即可再通过图片水印方法转换为原始水印。

3 实验

3.1 指令操作时间

根据表1，本文评估了多种类型的程序指令操作时间，在Deepin Linux操作系统、Intel Core i5-8265U的CPU, 8 GB内存环境下搭建仿真模拟器，对相关软件进行了测试，每条指令执行了100次，从而测算其时间。目前指令为Python指令，可以进行时间分析。由于本文方法具有通用性，对其他平台指令也可用类似的方法处理。实验可知，大部分指令执行时间相差并不大，仅移位指令和除法指令与其他指令具有较大的差异。本文又对移位指令选取较小的常数(数字2)，测试结果表明其执行时间并无太大差异。对于多个操作的简单组合(如x |= x>>2, x &= x<<2等)，虽然指令执行时间也与其他指令相差不大，但可以看出其时间仍然有所差异。因此对于指令水印的编码，可使用多类指令的不同序列叠加以及使用不同的常数，这样可以使得程序执行时间具有时间特征。可以看出水印程序指令序列与原始程序中的其他指令相似，不易被检查出。

本文对mark按照表2进行编码，通过类似的方法，可以看到其执行时间(仍然每个编码对应的代码序列执行100次)，其结果如表3所示。可以看出不同的字母已被编码。本文多次尝试，在不同的环境下，其执行时间仍然遵从类似的分布，因此可以用其作为水印代码。

3.2 对原始程序的影响

本文添加入原始程序的水印代码指令序列不能过多地影响原始程序的执行时间，因此需要测算其对原始程序执行时间的影响。本文使用了多个典型的程序，并在其中植入了水印信息(即上例中的水印“mark”)，可以发现其实际运行时间几乎不变。具体来说，本文选用了流行开源软件库github中影响力较大的示例代码集合“geekcomputers/Python”(在github中搜索“python example”后显示的第1条返回结果，具体网址是：https://github.com/geekcomputers/Python)。其中包括典型的python代码可供使用。本文随机挑选了其中的5个程序(CounterMillionCharacter.py, osinfo.py, pan.py, read_excel_file.py, nodepad/notepad.py)。考虑到避免人工输入的影响，本文选择的程序不包括需要交互式方式给定输入的程序，也不选择需网络连接的程序(网络连接时间不确定，易对比较结果造成影响)。值得注意的是，虽然我们这里不选

表 3 mark的时间编码序列

水印字母	操作代码	时间(μs)
m	$x = x + y; x = x - y$	2.14
a	$x = x \times y; x = x / y$	4.04
r	$x = x + y; x = x - y; x = x \times y; x = x / y$	7.14
k	$y = x; x = (x>>2); y = x \,\&\, 0x3; x = (x<<2)\|y$	6.19

择交互式程序与网络连接类的程序，但实际加入水印时，对这两类程序仍然可以植入水印。这里仅为了实验比较准确性不去选择这两类程序，结果显示添加水印后的执行时间与原始程序相比几乎无变化。

事实上，一方面水印代码的大小相对原始程序显得非常小，因此水印代码执行时间仅占程序的很小一部分。例如表3中，水印代码在μs量级，而从上述实验中可以看出，程序执行时间多在10^{-2} s量级。在大型程序中，水印代码更不明显。另一方面，植入的水印在程序运行过程中可能并不被实际使用，即添加的代码没有被执行，因此执行时间无变化。综上，水印代码的执行时间与原始代码相比，几乎可以忽略。

4　安全性评估与讨论

4.1　安全性评估

本文对当前较为常用水印方法进行安全性比较，选取表4中性能指标进行具体比较。其中，安全性是指在能否抵抗代码混淆等攻击手段；隐蔽性是指评估水印是否能被识别和定位的方法；水印容量是指水印信号携带的信息量，也成为水印的有效载荷；复杂度主要是针对水印植入和提取的复杂程度进行评估。具体如表4所示。

表4中，选择静态水印、动态水印中的经典算法与本文的方法SW-PET进行比较。例如，静态水印方法DMI为代码替换法，当发生代码混淆等操作时，水印容易被破坏；而且，水印隐蔽性不高，通过多份植入水印的程序进行分析，可以对水印位置进行定位；水印容量较高，可以植入任意尺寸的水印代码；该方法添加过程较为简单。GTW是静态图法，当发生基本块划分等混淆时，水印容易被破坏；可以将水印与原程序的流图(flow graph)相合并，增加水印的隐蔽性；可以植入任意大小的水印；水印复杂度一般。CT是动态图结构水印，通过将水印分成几个子图插入到代码图中，该方法拆分混淆后，水印容易被破坏；水印不容易被发现，具有一定的隐蔽性；可以植入任意大小的水印代码；复杂度一般。AppInk使用置换图，置换图采用特殊的图结构对水印对象的置换映射进行编码，因此具有较强的安全性；通过置换图变换，具有隐蔽性；可以植入任意大小的水印；水印对程序有一定的性能影响。本文SW_PET方法可以与其它类型水印编码方法相结合，具有较强的安全性；同时，水印相关代码隐蔽性强；水印大小没有限制；且复杂度低，容易实现。综上所述，本文提出的方法具有一定优势，实现方法简单，可以快速实现重打包软件检测。

表 4　典型水印算法安全性比较

水印方法	安全性	隐蔽性	水印容量	复杂度
DMI[13]	+	+	+++	+
GTW[14]	++	++	+++	++
CT[21]	++	+++	+++	++
AppInk[2]	+++	+++	+++	++
SW_PET	+++	+++	+++	+

4.2　安全性分析与讨论

对以下几种攻击的鲁棒性评估。

(1) 删除攻击：考虑到本文的水印代码隐蔽在程序中，很难被发现，也因此很难被删除。而且水印代码相对于原始程序，代码量非常少，攻击者很难找到。攻击者也无法采用随机删除的方法，因其会破坏原有程序执行的功能。即使攻击者偶然去除了部分水印代码，考虑到水印相似度Sim_W的计算过程，也可以比较出大部分的水印。同时，如果防御者使用了更为健壮的基于图片ASCII的水印，则删除攻击更显得难以有效了。

(2) 变换攻击：攻击者可以对程序进行无差异化的混淆(obfuscation)。主流的混淆方法包括变量改名、函数改名、类改名、添加无效语句等。其中变量改名、函数改名、类改名对本文的水印代码没有影响。下面来分析添加无效语句的影响。考虑到水印代码只是占用原始程序代码中非常小的一部分，攻击者相对较难碰巧改变水印代码部分。即使攻击者偶然添加部分语句到原始代码中，改变了部分时间，防御方仍然可以通过水印相似度Sim_W进行计算以及基于图片ASCII的水印等方法进行水印的鲁棒性增强。

(3) 追加攻击：追加攻击即攻击者往载体软件中加入其它水印，版权所有者在提取水印验证版权时，由于攻击者添加了新的水印，前后两者的差异性将引起版权纠纷，使原有的水印达不到版权证明的作用[23]。考虑到本文方法水印的隐蔽性，攻击者很难直接修改并覆盖水印的代码(参考去除攻击，否则攻击者可以直接删除本文水印)。因此攻击者绝大多数情况下会直接在附带水印的程序中添加其自己的水印。若该类水印是静态文本水印(如加入特殊的字符)等，则无法改变程序的执行时间，因此该类水印不会改变执行时间，也无法改变该水印。若该类水印和本文水印类似，考虑到本文使用time()进行某些程序语句执行时间的计算，若攻击者水印不在本文水印之间，则也难以修改本文水印。若碰巧加入了部分水印之间，则可以通过水印相似度进行水印的比较，仍然可以发现大部分本文水印。

(4) 其他攻击讨论：攻击者可以对原始程序进行修改，例如在移动应用程序上加入一个单独的Activity。这种方法不改变原始程序代码，但可以通过在系统中注册一个回调函数(或者通知事件)，从而使得该Activity得到调用。亦或者攻击者可以使用不修改程序代码的方法重打包程序，例如修改广告的注册号(ID)为攻击者自身的ID，使得正常程序的收益将会被分配到攻击者手中。值得注意的是，程序水印无法保证程序不被修改，仅能确认修改后的程序是否是来自于原始作者。通过检测水印的存在性(仅原作者或者仲裁人知晓)，判别被修改的软件是否存在原始作者预设置隐藏的行为，从而判别该程序是否来自于原始作者或存在来自于原始作者的部分程序代码。若需要对程序被修改这一攻击进行检测，则需要对程序做完整性保护。

此外，对于源代码程序，水印的植入与提取相对较为容易。对于可执行程序来说，将程序进行反编译后，水印的植入相对较为容易，可以直接在特定位置植入即可。本文中的源代码ECO可替换为可执行程序的ECO。但水印的提取存在的问题主要是攻击者可能随意修改程序，导致植入特定代码的位置标记发生改变，即与原先程序水印代码植入位置不一致。本文可通过如下方法解决：对于水印植入过程中的输入，可以记录其执行路径。对于植入后水印需要验证的程序，也可以使用同一个输入得到路径，通过匹配这两部分路径，即可得到相对比较准确的水印代码植入点，在该类位置加入水印时间提取函数，即可提取水印。利用某一输入寻找路径方式，可以仅比较同一路径。因此，可以简化比较过程，使得比较过程更为准确。此外，本文所述方法具备通用性，可以应用于Android移动终端和Windows, Linux等个人电脑应用软件中。

5 结束语

本文通过一种基于程序执行时间量化分析的软件水印方法，实现对应用程序软件剽窃、重打包等安全风险进行测评。该方法将生成的多组操作对应的代码执行时间作为水印信息，通过将给定水印信息进行时间化编码，并与程序代码结合完成水印的植入。进行水印验证时，按照之前的水印编码方式提取水印，并通过观察程序特定语句的执行时间来判定程序的合法性。该方法能够与其它类型的水印编码方法相结合，具有较好的隐蔽性和鲁棒性。后续，还将进一步优化该水印生成与验证过程，提高水印的鲁棒性。

参 考 文 献

[1] 林迪. 2018年中国App下载量排名全球第一：占全球50%[EB/OL]. https://www.sohu.com/a/289551518_162522, 2019.

[2] ZHOU Wu, ZHANG Xinwen, and JIANG Xuxian. AppInk: Watermarking android apps for repackaging deterrence[C]. The 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. Hangzhou, China, 2013: 1–12. doi: 10.1145/2484313.2484315.

[3] ZHOU Wu, ZHOU Yajin, JIANG Xuxian, *et al*. Detecting repackaged smartphone applications in third-party android marketplaces[C]. The 2nd ACM Conference on Data and Application Security and Privacy. San Antonio, United States, 2012: 317–326. doi: 10.1145/2133601.2133640.

[4] Arxan Technologies. State of security in the App Economy: Mobile apps under attack[EB/OL]. http://www.arxan.com/assets/1/7/state-of-security-appeconomy.pdf, 2012.

[5] CHEN Kai, ZHANG Yingjun, and LIU Peng. Leveraging information asymmetry to transform android apps into self-defending code against repackaging attacks[J]. *IEEE Transactions on Mobile Computing*, 2018, 17(8): 1879–1893. doi: 10.1109/TMC.2017.2782249.

[6] CHEN Kai, LIU Peng, and ZHANG Yingjun. Achieving accuracy and scalability simultaneously in detecting application clones on android markets[C]. The 36th International Conference on Software Engineering. Hyderabad, India, 2014: 175–186. doi: 10.1145/2568225.2568286.

[7] CRUSSELL J, GIBLER C, and CHEN Hao. AnDarwin: Scalable detection of semantically similar android applications[C]. The 18th European Symposium on Research in Computer Security on Computer Security. Egham, UK, 2013: 182–199. doi: 10.1007/978-3-642-40203-6_11.

[8] Guardsquare. Proguar[EB/OL]. http://proguard.sourceforge.net/, 2013.

[9] Guardsquare. A specialized optimizer and obfuscator for android[EB/OL]. http://www.saikoa.com/dexguard, 2013.

[10] 陈明奇, 钮心忻, 杨义先. 数字水印的攻击方法[J]. 电子与信息学报, 2001, 23(7): 705–711.

CHEN Mingqi, NIU Xinyi, and YANG Yixian. The attack methods of digital watermarking[J]. *Journal of Electronics & Information Technology*, 2001, 23(7): 705–711.

[11] 毛琼, 陈明奇, 夏光升, 等. 安全数字水印体系的研究[J]. 电子与信息学报, 2001, 23(9): 833–840.

MAO Qiong, CHEN Mingqi, XIA Guangsheng, *et al*. The research of secure digital watermarking architecture[J]. *Journal of Electronics & Information Technology*, 2001, 23(9): 833–840.

[12] HAMILTON J and DANICIC S. A survey of static software watermarking[C]. 2011 World Congress on Internet Security. London, UK, 2011: 100–107. doi: 10.1109/

worldcis17046.2011.5749891.

[13] MONDEN A, IIDA H, MATSUMOTO K, *et al.* A practical method for watermarking java programs[C]. The 24th Annual International Computer Software and Applications Conference. Taipei, China, 2000: 191–197. doi: 10.1109/CMPSAC.2000.884716.

[14] VENKATESAN R, VAZIRANI V, and SINHA S. A graph theoretic approach to software watermarking[C]. The 4th International Workshop on Information Hiding. Pittsburgh, USA, 2001: 157–168. doi: 10.1007/3-540-45496-9_12.

[15] COUSOT P and COUSOT R. An abstract interpretation-based framework for software watermarking[C]. The 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Venice, Italy, 2004: 173–185. doi: 10.1145/964001.964016.

[16] NAGRA J and THOMBORSON C. Threading software watermarks[C]. The 6th International Workshop on Information Hiding. Toronto, Canada, 2004: 208–223. doi: 10.1007/978-3-540-30114-1_15.

[17] COLLBERG C, HUNTWORK A, CARTER E, *et al.* More on graph theoretic software watermarks: Implementation, analysis, and attacks[J]. *Information and Software Technology*, 2009, 51(1): 56–67. doi: 10.1016/j.infsof.2008.09.016.

[18] COLLBERG C, CARTER E, DEBRAY S, *et al.* Dynamic path-based software watermarking[J]. *ACM Sigplan Notices*, 2004, 39(6): 107–118. doi: 10.1145/996893.996856.

[19] COLLBERG C, CARTER E, DEBRAY S, *et al.* Dynamic path-based software watermarking[C]. The 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation, Washington, USA, 2004: 107–118. doi: 10.1145/996841.996856.

[20] ZENG Lingling, REN Wei, LEI Min, *et al.* DroidMark: A lightweight android text and space watermark scheme based on semantics of XML and DEX[C]. The 5th International Conference on Emerging Internetworking. Wuhan, China, 2017: 756–766. doi: 10.1007/978-3-319-59463-7_75.

[21] COLLBERG C and THOMBORSON C. Software watermarking: Models and dynamic embedding[C]. The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. San Antonio, USA, 1999: 311–324. doi: 10.1145/292540.292569.

[22] ZHANG Yingjun and CHEN Kai. AppMark: A picture-based watermark for android apps[C]. The 8th International Conference on Software Security and Reliability (SERE). San Francisco, USA, 2014: 58–67. doi: 10.1109/SERE.2014.19.

[23] 王叶茂, 车生兵. 软件水印及其研究现状概述[J]. 计算机应用与软件, 2015, 32(4): 6–10. doi: 10.3969/j.issn.1000-386x.2015.04.002.

WANG Yemao and CHE Shengbing. Summary on software watermarking and its research progress[J]. *Computer Applications and Software*, 2015, 32(4): 6–10. doi: 10.3969/j.issn.1000-386x.2015.04.002.

张颖君：女，1982年生，副研究员，主要研究方向为安全测评、系统安全.

陈　恺：男，1982年生，研究员，主要研究方向为系统安全、人工智能安全.

鲍旭华：男，1977年生，高级工程师，主要研究方向为信息安全.

责任编辑：陈　倩