工程硕士学位论文

抗逆向分析 PE 文件保护系统研究与实现

田淞煜

哈尔滨理工大学

2021 年 4 月

国内图书分类号：TP301.6

工程硕士学位论文

抗逆向分析 PE 文件保护系统研究与实现

硕士研究生： 田淞煜

导　　师： 李鹏

申请学位级别： 工程硕士

学科、专业： 计算机技术

所在单位： 计算机科学与技术学院

答辩日期： 2021 年 4 月

授予学位单位： 哈尔滨理工大学

Classified Index：TP301.6

Dissertation for the Master Degree in Engineering

Research and implementation of PE file protection system for antistress analysis

Candidate： Tian Songyu
Supervisor： Li Peng
Academic Degree Applied for： Master of Engineering
Specialty： Computer Technology
Date of Oral Examination： April, 2021
University： Harbin University of Science and Technology

哈尔滨理工大学硕士学位论文原创性声明

本人郑重声明：此处所提交的硕士学位论文《抗逆向分析 PE 文件保护系统研究与实现》，是本人在导师指导下，在哈尔滨理工大学攻读硕士学位期间独立进行研究工作所取得的成果。据本人所知，论文中除已注明部分外不包含他人已发表或撰写过的研究成果。对本文研究工作做出贡献的个人和集体，均已在文中以明确方式注明。本声明的法律结果将完全由本人承担。

作者签名：　　日期：　　年　　月　　日

哈尔滨理工大学硕士学位论文使用授权书

《抗逆向分析 PE 文件保护系统研究与实现》系本人在哈尔滨理工大学攻读硕士学位期间在导师指导下完成的硕士学位论文。本论文的研究成果归哈尔滨理工大学所有，本论文的研究内容不得以其它单位的名义发表。本人完全了解哈尔滨理工大学关于保存、使用学位论文的规定，同意学校保留并向有关部门提交论文和电子版本，允许论文被查阅和借阅。本人授权哈尔滨理工大学可以采用影印、缩印或其他复制手段保存论文，可以公布论文的全部或部分内容。

本学位论文属于

保密□，在　　年解密后适用授权书。

不保密□。

(请在以上相应方框内打 √)

作者签名：　　日期：　　年　　月　　日

导师签名：　　日期：　　年　　月　　日

抗逆向分析 PE 文件保护系统研究与实现

摘 要

为应对逆向分析给 Windows 软件带来的安全威胁，本文提出并实现一个针对 Windows 可执行程序的安全保护系统，该系统对可执行程序逆向中的静态分析和动态分析过程进行保护，有效提高逆向分析者的分析难度。采取的主要保护措施有：一、加壳，在已有虚拟机加壳技术的基础上进行改进，提出多样化 Handler 的保护机制；二、代码混淆、通过对软件进行混淆处理，改变程序的运行时的函数块的结构并增加指令的复杂度，以此来增加逆向分析难度。本文以增强 PE 文件的抗逆向能力为目的，对 Windows 平台下的 PE 文件的二进制混淆技术和虚拟机加壳技术进行研究，主要工作包括：

首先，分析 Windows 下 PE 文件的加壳技术和二进制保护技术的研究现状，对 PE 文件抗逆向分析的常见方法进行总结，简单阐述 PE 文件加壳原理和市面流行壳的类型分析。

其次，对基于虚拟机的多样化 Handler 加壳保护方式进行详细阐述，对建立的数据结构、调用约定和框架都从源码的层面上进行详细解释，通过对比普通 Handler 实现和多样化 Handler 实现的优缺点，最终证明多样化 Handler 具有更强的抗逆向性能，进而证明所提出的加壳方法的有效性。

再次，提出建立索引的方式对函数间基本块进行交换的静态保护算法，对算法的基本思想进行简要阐述，通过对常用的反调试技术进行分析，发现常用反调试技术的局限性，在其基础上进行优化改进，在最后设计并实现一个二进制混淆器。

最后，对以上提出的加壳方式进行实验验证，通过与流行加壳软件对比压缩率、运行时间额外开销、静态和动态指令执行率等参数，间接证明提出的保护方式是有效可性的，最终实验结果得出提出的保护方式在隐蔽性、运行时间开销和占用空间等方面具有一定优势。

关键词　虚拟软件加壳，代码混淆，PE 文件保护，逆向工程，软件保护

Research and implementation of PE file protection system for antistress analysis

Abstract

In order to deal with the security threats brought by reverse analysis to Windows software, this paper proposes and implements a security protection system for Windows executable programs. The system protects the static analysis and dynamic analysis process in the reverse analysis of executable programs and effectively improves reverse analysis. The difficulty of the analysis. The main protection measures adopted are: 1. Packing, improving on the basis of existing virtual machine packing technology, and proposing a protection mechanism of diversified Handler; 2. Code obfuscation, changing the operation of the program by obfuscating the software The structure of the function block at the time and increase the complexity of the instruction, so as to increase the difficulty of reverse analysis. This paper aims at enhancing the anti-reverse ability of PE files, and researches the binary obfuscation technology and virtual machine packing technology of PE files under the Windows platform. The main work includes:

First, analyze the research status of PE file packer technology and binary protection technology under Windows, summarize the common methods of PE file anti-reverse analysis, briefly explain the PE file packer principle and the type analysis of popular shells in the market.

Secondly, the virtual machine-based diversified Handler shelling protection method is elaborated, and the established data structure, calling convention and framework are explained in detail from the source code level, and the advantages of common Handler implementation and diversified Handler implementation are compared. The shortcomings finally prove that the diversified Handler has stronger anti-reverse performance, and then prove the effectiveness of the proposed packing method.

Thirdly, a static protection algorithm for exchanging basic blocks between functions is proposed by establishing an index, and the basic idea of the algorithm is

briefly explained. Through the analysis of the commonly used anti-debugging techniques, the limitations of the commonly used anti-debugging techniques are found. Optimize and improve on the above, and finally design and implement a binary obfuscator.

Finally, the experimental verification of the above-mentioned packing method is carried out. By comparing the compression rate, runtime overhead, static and dynamic instruction execution rate and other parameters with popular packing software, it indirectly proves that the proposed protection method is effective and feasible. The experimental results show that the proposed protection method has certain advantages in terms of concealment, running time overhead and space occupation.

Keywords Virtual software packing; code obfuscation; PE file protection; reverse engineering; software protection

目 录

摘 要 I
Abstract II

第 1 章 绪论 1
1.1 课题研究的目的和意义 1
1.2 国内外研究现状 3
1.2.1 国内研究现状 3
1.2.2 国外研究现状 4
1.3 本课题主要研究内容 5
第 2 章 抗逆向分析的 PE 文件保护技术 7
2.1 PE 文件格式 7
2.1.1 PE 文件总体结构 7
2.1.2 DOS MZ Header 8
2.1.3 PE 文件头 9
2.1.4 相对虚拟地址 9
2.1.5 区块表 10
2.1.6 导入表 10
2.2 PE 文件对抗逆向分析的常见方法 13
2.2.1 压缩壳 15
2.2.2 保护壳 16
2.2.3 工具脱壳 17
2.2.4 手工脱壳 18
2.2.5 壳的类型分析 18
2.3 本章小节 19
第 3 章 多样化 Handler 虚拟机加壳的研究与实现 20
3.1 虚拟机保护方法基本原理 20
3.1.1 相关反汇编工具 21
3.1.2 X86 指令 22
3.2 调用约定和框架 23

3.2.1 虚拟环境 23
3.2.2 调度器 24
3.3 多样化 Handler 设计 25
3.3.1 辅助和普通 Handler 的实现 25
3.3.2 多样化 Hanlder 实现 27
3.4 本章小结 30
第 4 章 静态分析混淆技术研究与实现 32
4.1 常用的反调试技术 32
4.1.1 花指令 33
4.1.2 文件完整性检验 35
4.1.3 调试器检测 35
4.2 建立索引进行函数间基本块交换的混淆算法 35
4.2.1 基本思想 36
4.2.2 算法描述 38
4.3 二进制混淆器设计与实现 39
4.3.1 总体设计 39
4.3.2 二进制分析器设计与实现 40
4.3.3 混淆器的设计与实现 40
4.3.4 二进制文件重建 43
4.4 本章小结 43
第 5 章 系统实现与测试 44
5.1 总体设计方案 44
5.1.1 编程语言的选择 44
5.1.2 软件的系统构架 44
5.2 功能验证 45
5.2.1 虚拟机加壳功能验证 45
5.2.2 二进制混淆器功能验证 46
5.3 性能评估 47
5.3.1 虚拟机加壳性能评估 50
5.3.2 二进制混淆器功能评估 51
5.4 本章小结 56
结论 57

参考文献 58
攻读硕士学位期间发表的学术论文与研究成果 61
致谢 62

第 1 章　绪论

1.1　课题研究的目的和意义

自第一台计算机出现以来，计算机软件安全成为计算机工程师不得花费心力思考的问题。在第一个蠕虫病毒出现之后，计算机的软件信息安全受到严重威胁。普通用户的计算机受到各种病毒和木马的威胁，其中包括大部分是通过网络传输到个人计算机中，通过修改程序的二进制代码，改变程序的执行流程，有的甚至被植入 Shellcode 代码，从而执行破坏或者窃取信息的代码，这是用户面临的软件程序的安全问题。

在当今互联网世界中，软件开发者的版权保护受到越来越多的关注，在二进制的层面上保护软件已经是很多软件生产公司最关注的问题之一，就算是 Microsoft 公司生产的 Windows 操作系统系列产品都会被技术水平较高的黑客破解[1]，可见软件保护在当今互联网时代的难度之高，但是软件保护的意义不能小觑，很多商业软件依然给软件提供了各种注册方式，试图阻碍破解者获取软件正确序列号和逆向分析软件，其中包括通过网络验证、非对称加密 RSA 算法、机器注册码等方式。

类比于自然界植物用壳保护种子，计算机软件壳也是用来保护软件中的内容不被破解者非法修改。加壳软件在被保护软件中加入一段代码，并把原程序中代码执行区域中的代码加密、压缩，在程序正常运行时，被添加的代码会优先运行，为程序提供运行环境，包括整理堆栈、初始化 CPU 寄存器值等指令，最后对代码执行区域中的代码进行解密，转换为原代码后，最后将程序的执行权限交付给原程序。

软件盗版和篡改是世界面临的众所周知的威胁。已经进行了很多尝试来保护软件免受逆向工程和篡改。似乎软件开发人员和破解者之间正在进行一场战争，双方都希望随着时间的流逝相互竞争。审查了一些丰富的软件保护技术[2]，包括多块哈希方案，基于硬件的解决方案，校验和，混淆，防护，软件老化，加密技术和水印。所有这些技术都发挥了自己的作用，以保护软件免受恶意攻击。

软件保护在游戏行业也显得颇为重要，国内游戏软件两巨头腾讯和网易投入大量的人力资源对游戏软件加入反外挂机制，由于大部分网游都是运行

在客户端，破解者可以通过分析二进制代码，使用 ODDbg、IDA 和 WinDbg 等反汇编工具对游戏进行动态或静态的反汇编[3]，技术较为程序的破解者通过分析汇编代码，是可以对整个游戏的流程进行细致分析，从而找出游戏中各种人物属性、技能、天赋在内存中的存放位置，最后可以使用 C 语言，对游戏进行远程内存注入，实现内存区域的控制，最后实现一个任意用户可用的游戏性外挂，这对当今依靠会员充值等方式盈利的公司是一个很大的打击，破解者甚至在分析游戏汇编代码后，可以构造网络游戏的网络包，直接对服务器发出伪造好的数据包，数据包中可能存放着购入金钱代码，直接对游戏进行充值，更有不法份子对这种修改软件进行售卖[4]，对游戏公司产生非常大的利益损失。

程序二进制混淆是软件开发人员和密码学专家长期以来一直感兴趣的话题。此处动机非常简单：找到一种方法，使我们可以为人们提供可以运行的程序，而无需他们弄清楚了程序是如何工作的，最后一部分必定会涉及很多内容[5]。原则上，它包括从所使用的特定秘密算法的性质到可能会使用硬编码加密程序中的秘密信息等方面。

大多数程序都非常易于阅读，攻击者可能只是查看此程序即可恢复序列号密码[3]。程序混淆的想法是，如果能够以某种方式阻止人们这样做，同时让他们拥有并在自己的计算机上运行的程序，那许多需要保护的软件都可以使用这种方式保护起来[6]。

在现实世界的软件系统中，“混淆”通常是指一些临时技术的集合，这些技术将一些设计巧妙的程序变成许多 GOTO 代码组合，有时重要的常量会被切碎并分布在代码周围[7]。该代码的某些部分甚至可以被加密，尽管是暂时的，因为解密密钥必须随程序一起提供才能真正运行[8]。

只要有足够时间、精力和工具[9]，人人都可以摆脱最常见的软件混淆技术[10]。现有软件混淆的质量很差，是密码学家面临的问题。是否存在这样一个程序混淆加密器，用这样的混淆器来加密受保护的软件，同时要证明可保护所有需要保护的信息。如果能够做到这一点，它将拥有惊人的应用场景。股票交易者可以混淆其专有的交易算法[11]，然后将其发送到云端或者最终客户。产生的程序仍然可以正常工作[12]，但是客户永远不会学到“工作原理“的任何信息。在程序外看，程序就是一个黑匣子，而程序的内部将是各种混合机制和原有代码的融合[13]。

因此，软件安全问题是普通用户和公司软件都不容忽视的问题，软件保

护者需要开发出既能保证软件正常运行同时又避免被逆向者分析的软件保护程序，本文将针对软件保护程序进行相关技术的研究。

1.2 国内外研究现状

在软件混淆加密技术领域，全球有诸多小组致力于此研究，也均在保证加密性能和运行性能下给出了出色的解决方案。

1.2.1 国内研究现状

浙江大学的高勇等人提出使用加密狗虚拟技术实现共享式工作平台，加密狗通常是 USB 记忆棒，改 USB 记忆棒连接到计算机上的端口以验证许可证[14]。使用 USB 加密狗运行的软件首先在启动后定期发送一个请求到 I/O 端口进行身份验证。一旦无法检索到与其的验证码，该程序就会自动终止，或者只能访问有限的功能[15]。加密狗尤其可以防止未授权的复制。原理本质上非常简单：没有加密狗，无法访问软件。现代硬件加密狗使用公私钥和对称机密过程，加密密钥不包含在应用程序的某个位置[16]，而是安全地存储在 Flash-ROM 中，在其中它们无法被读出并仅用于加密和解密。除此之外，还有具有网络支持的加密狗，可以将其连接到网络中的任何计算机或服务器。许可证服务器应用程序正在此计算机上运行，并在网络中提供许可证。现在，受保护的应用程序将检查本地连接的加密狗或网络中的许可证服务器[17]，此外，还可以通过基于硬件，如 CPU、主板和硬件驱动器等生成特殊的许可证密钥，讲软件许可证绑定到特定计算机，并将其存储在加密狗中。加密狗可以大大减少软件盗版，并且在数字版权管理中尤为有效，因为生成加密狗的非法副本非常困难。

在研究软件保护的初期，研究人员开发了一些相对有用的专业软件加密保护程序。但是，随着破解技术的发展，即使使用强大的加密算法，如 Twofish，TEA，Blowfish 以及 CRC 循环冗余校验和反调试技术的组合，坚固外壳 AS-Protect 也可以通过使用免费的 OllyDbg 删除动态跟踪外壳后的反汇编代码。使用堆栈平衡原理在程序执行入口之前找到外壳，然后结合 LoadPE 工具的强大功能来导入表，导入地址表和重定位表。当前，VMProtect 和驱动程序保护技术是保护软件的两种最重要的方法[18]。

国立台湾科技大学自动化与控制研究所使用 AES 算法对图像进行了加

密试验，杨成雄教授提出了一种基于四维混沌系统的图像加密算法[19]，以生成密钥并提高高级加密标准。通过使用现场可编程门阵列 FPGA 的流水线和并行计算功能来优化加密算法。首先，混沌系统用作加密算法的密钥生成器。接下来，在改进的高级加密标准中[20]，使用 Spin-Sort 和 Cubic S-Box 修改了 ShiftRows 和 SubByres，并减少了加密次数。我们将加密算法和有线图像传输系统实现到基于 ARM 的 SoC-FPGA。HPS 软件在 Linux 上运行，用于控制 FPGA 加密算法和图像传输。

1.2.2 国外研究现状

比利时密码学家 Vincent Rijmen 和 Joan Daemen 开发了 Rijndael 分组密码[21] 的子集，它们在 AES 选择过程中向 NIST 提交了提案[22]。Rijndael 是具有不同密钥和块大小的密码家族。对于 AES，NIST 选择了 Rijndael 系列的三个成员[23]，每个成员的块大小为 128 位，但是具有三个不同的密钥长度：128、192 和 256 位。AES 已经被美国政府采用，现已在全球范围内使用[24]。它取代了 1977 年发布的数据加密标准 DES。AES 描述的算法是对称密钥算法，意味着同一密钥用于加密和解密数据。

麻省理工大学教授 N. Sasirekha 和 M. Hemalatha 提出了一种基于带 Hadamard 索引表准组加密和数字理论变换的有效安全代码方法[25]，以进行软件保护[26]，此方法使用一种称为准组加密的新颖有效的加密技术对索引表进行加密。加密后，它与原始数据的相似性最小。准确有效地产生了复杂数字的密钥，这是逆向分析者难以识别原始数据。但是，准组加密在扩散纯文本的统计信息方面效率不高。因此，此方法使用链式 Hadamard 变换和数字理论变换将扩散与准群变换一起引入。实验结果基于时间成本和空间成本评估了所提出的加密方法的性能，并且观察到所提出的方法提供了重要的结果。

剑桥大学 Barak 等人提出了一种“不可区分性混淆程序”[27]，简单可概括为如下内容：有两个程序 *C1* 和 *C2* 我们将它们描述为大小相似的电路，他们计算的功能相同，更具体地说，我们可以说它们具有完全相同的输入和输出行为，尽管它们在内部实现的方式可能非常不同，不可区分混淆的定义指出，应该对两个电路 *C1*，*C2* 进行混淆，以使没有有效的算法能够分辨 *Obf*(*C1*) 与 *Obf*(*C2*) 之间的差异。尽管这个想法是在多年前提出的，但实际上没有人知道如何构建这样的东西，称为那些“未解决的问题”之一。直到前年，情况仍然如此，直到 IBM Research 的一组作者基于多线程图的密码学新领域，提出

了一种“候选构造”来构造这种混淆器[28]。这个概念的另一个有趣的变种被称为萃取混淆 *Obf*(*EO*)，这不仅意味着你不能分辨之间 *OBF*(*C1*) 和 *OBF*(*C2*), 但是，如果你能区分这两种情况，那就可以找到一个输出值，*C1* 和 *C2* 都将在该输入值上产生不同的输出。此外，其他工作表明 *IO* 和 *EO* 本质可以为常规程序提供必要好的混淆算法。

加利福尼亚大学 Paul A. Cronce 和 Joseph M. Fontana 等人提出一种将源代码转换为字节码来保护软件[28]。他们的这种方式提供用于编程语言的语言规范，用于实现该语言的库，用于将该语言编译为字节码的编译器以及使用该库执行字节码的解释器[29]; 向软件发布者提供语言规范和说明，以及用于指导软件发布者如何从要保护的应用程序中选择代码部分以及如何准备所选代码部分和应用程序以在服务器上进行处理的说明，包括指示发布者在从应用程序中获取所选代码部分的相应位置创建数据结构；向服务器提供编译器、库、解释器和服务器应用程序，用于从发布者接收要保护的软件应用程序和准备好的代码选择部分。使用编译器将代码的选定部分编译为字节代码，将编译器生成的字节代码嵌入到应用程序中[30]，并用解释程序调用替换数据结构，以调用代表已删除代码部分的适当字节代码模块的解释[31]，从而使应用程序正确运行[32]，并在解释程序上运行经过混淆的代码部分和在应用程序中嵌入库和解释器以支持编译后的字节码的运行时解释，从而使所选段变得模糊[33]。

1.3 本课题主要研究内容

目前国内外的软件加密技术主要是使用硬件加密狗、高强度算法和使用字节码虚拟机等保护方式，以上方式都有各自的局限性，比如硬件保护方式虽然保护强度高，但是保护成本过高，每个程序的发型都需要适配一个 USB 硬件，不适合小型软件的发布，并且软件的运行过程由于需要插入 USB 设备，造成软件运行过于繁琐。高强度算法和字节码虚拟机等保护方式由于在运行过程中需要解码，所以存在软件运行效率急剧下降的问题。本文针对以上提出的问题，进行优化改进，保证了软件安全强度的前提下，尽可能提高软件的运行效率。

本文主要研究内容为：采用在反动态调试和反静态调试两个方面来防止未授权用户进行注册或者调试，反动态调试方面是对软件采用虚拟机加壳的

方式，对软件进行打包，将大部分汇编代码转换为字节码，在运行时通过解释器翻译给操作系统，如果破解者不了解其中字节码和汇编代码的对应关系，将很难实现未授权注册行为; 反静态调试是在已有的函数间基本块进行交换的静态保护算法基础上提出建立索引的方式对函数间基本块进行交换的静态保护算法，这种方式与剑桥大学研究内容的目标基本相同，都是对静态二进制代码进行混淆，加大 IDA 等静态反汇编工具的分析难度，建立索引的方式会在软件内部新增加一个节区，用来记录所有参与过函数块之间交换的块，在程序运行时，解释器会根据这个索引表进行软件的混淆机制恢复过程。

本文设计的虚拟机是一种解释执行系统与加利福尼亚大学提出的将源代码转换为字节码的原理相同，本文会提出一种基于虚拟机加壳的多样化 Handler 动态保护方法，在原有虚拟机保护的基础上，在二进制层面加大逆向者的分析难度，从而让软件得到更有效的保护。与解释执行语言中的虚拟机不同，本文设计的虚拟机是存在于每个可执行程序中的，所有被加密的软件都经过虚拟机加壳器的处理，其中的可执行硬编码都已经变成伪指令，即使破解者进入到虚拟机中跟踪虚拟机的解释器算法，由于每一套虚拟机的指令的不同，破解者将很难理解经过处理后的指令，破解者如果想正确逆向这样的软件，就必须对虚拟机的引擎进行深入的研究，通过不断的反复对比试验和猜测，最后形成伪指令和原始指令的对应关系，就像在破解一套摩尔斯电码但是不知道电码的字典一样，这样可以大大增加破解的难度和破译成本，也正是由于这种破解难度，虚拟机保护已经成为软件保护者的趋势。

第 2 章　抗逆向分析的 PE 文件保护技术

2.1　PE 文件格式

本课题的任务内容是对 PE 文件进行修改并保护[34]，然后使之可以正常运行，所以对于 PE 文件结构的内容不容忽视，由于 PE 文件的修改都是基本本章节内容，所以后文将会对本节内容进行多次引用，本章将会对本课题涉及到的关于 PE 文件结构进行详细的研究，并分析出 PE 文件可以经本课题使用的空间。

2.1.1　PE 文件总体结构

本课题的目的是保护 Windows 的可执行程序，通过改变可执行文件的数据结构，防止被破解者分析篡改，所以一定要对 Windows 下的可执行程序有详细的了解，本节内容将会对 PE 文件进行详细分析，明确每个数据结构的作用，同时对有价值的信息进行保护[35]。

PE（Portable Executable) 是一种 32 位和 64 位 Windodws 下的可执行文件，包括后缀名为 DLL 和 EXE 等的可执行文件。PE 格式是一种数据结构，其中封装了 Windows OS 加载程序管理包装的可执行代码所需的信息。这包括用于链接的动态库引用[36]，API 导出和导入表，资源管理数据和线程本地存储 TLS 数据。在 NT 操作系统上，PE 格式用于 EXE，DLL，SYS 设备驱动程序和其他文件类型。所述可扩展固件接口 EFI 规范规定 PE 是在 EFI 环境标准的可执行格式[37]。与 PE 类似的格式是 ELF, 它在 Linux 和大多数其他版本的 Unix 中使用。

PE 文件的一个非常方便的方面是磁盘上的数据结构与内存中使用的数据结构相同。可以通过调用 LoadLibrary 函数将可执行文件加载到内存中，遇到的问题主要如何将 PE 文件的某些范围数据映射到内存地址空间中。因此，像 IMAGE_NT_HEADERS 结构体这样的数据结构在磁盘和内存中是相同的。透彻了解 PE 文件结构中的内容，可以全面认识 PE 文件在执行时在内存中的布局[38]。重要的是要注意，PE 文件不只是作为单个内存映射文件映射到内存中。取而代之的是，Windows 加载程序查看 PE 文件并决定要映射到文件

的哪些部分。这种映射是一致的，因为当映射到内存时，文件中的较高偏移量对应于较高的内存地址。磁盘文件中项目的偏移量可能与加载到内存后的偏移量有所不同。但是，PE 文件中的信息都会在系统内存中出现[39]，以使您能够从磁盘偏移量转换为内存偏移量，如图2-1所示。

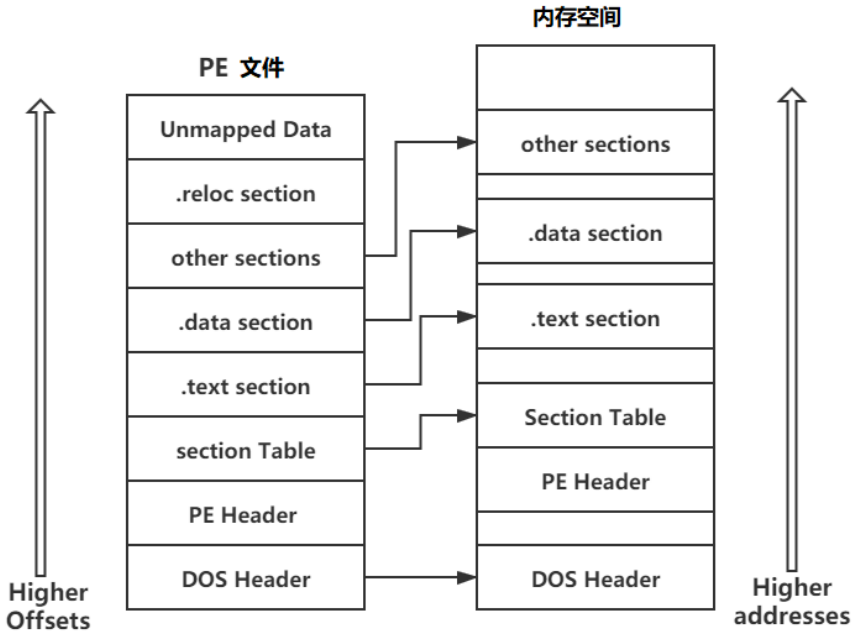

图 2-1　PE 文件运行后再内存中的节区排布情况

Figure 2-1　After the PE file is run, then the section layout in memory

2.1.2　DOS MZ Header

每个 PE 文件都以一个小型 MS-DOS 可执行文件开头[40]。在 Windows 的早期，即大量的用户运行它之前，就需要此存根可执行文件[41]。在没有 Windows 的计算机上执行时，该程序至少可以打印出一条消息，指出要运行该可执行文件是 Windows 平台上的程序。

PE 文件的第一个字节以传统的 MS-DOS 标头开始[42]，称为 IMAGE_DOS_HEADER。任何重要的仅有两个值是 e_magic 和 e_lfanew 字段包含 PE 标头的文件偏移。e_magic 字段（一个 WORD）需要设置为值 0x5A4D。

这个值有一个 #define, 名为 IMAGE_DOS_SIGNATURE。在 ASCII 表示中，0x5A4D 是 MZ,MZ 是 Mark Zbikowski（MS-DOS 的原始体系结构之一）的缩写。

2.1.3 PE 文件头

PE Header 在 PE 文件中的数据结构名为 IMAGE_NT_HEADERS，IMAGE_NT_HEADERS 结构是存储 PE 文件详细信息的主要位置。它的偏移量由文件开头 IMAGE_DOS_HEADER 中的 e_lfanew 字段给出。IMAGE_NT_HEADER 结构实际上有两个版本，一个用于 32 位可执行文件，另一个用于 64 位版本，两个版本的文件结构差异较小[43]。微软认可的唯一正确的区分两种格式的方法是通过 IMAGE_OPTIONAL_HEADER 中的 Magic 字段的值。

IMAGE_NT_HEADER 由三个字段组成，如图2-2。

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;                          //PE文件标识，"PE\0\0"
    IMAGE_FILE_HEADER FileHeader;             //PE标准头
    IMAGE_OPTIONAL_HEADER32 OptionalHeader; //PE扩展头
}IMAGE_NT_HEADER32, *PIMAGE_NT_HEADER32;
```

图 2-2 IMAGE_NT_HEADER 结构定义

Figure 2-2 IMAGE_NT_HEADER structure definition

2.1.4 相对虚拟地址

在可执行文件中，有很多地方需要指定内存地址。例如，引用全局变量时需要一个绝对地址。PE 文件可能在过程地址空间中的任何位置加载。虽然它们确实具有首选的加载地址，首选加载地址可能已经被使用了，这时的绝对地址就会发生改变[44]。因此，使用某种方式指定可执行文件加载位置的地址非常重要。

为了避免在 PE 文件中使用硬编码的内存地址，使用了 RVA（Relative Virtual Addresses）。RVA 只是相对于 PE 文件加载位置的内存偏移量。例如，考虑一个在地址 0x400000 加载的 EXE 文件，其代码段在地址 0x401000[45]。代码部分的 RVA 为：

(Target address)0x401000 - (Load address)0x400000 = (RVA)0x1000

要将 RVA 转换为实际地址，只需完成以下过程即可：将 RVA 添加到实际加载地址以找到实际的内存地址。实际的内存地址在 PE 术语中称为虚拟地址。定位 VA 的另一种方法是，使用首选加载地址的 RVA 通过上述方法计算得出。

2.1.5 区块表

在 PE 文件头 IMAGE_NT_HEADERS 结构之后的是区块表。IMAGE_SECTION_HEADERs 结构的数组。IMAGE_SECTION_HEADER 提供有关其关联节的信息，包括位置，长度和特征。IMAGE_SECTION_HEADER 结构的数目由 IMAGE_NT_HEADERS.FileHeader.NumberOfSections 字段给出。

2.1.6 导入表

在 PE 文件中，存在名为导入表的数据结构数组，导入表给出了所有需要导入的 DLL 的名称，并指向一个函数指针数组。函数指针数组称为导入地址表（IAT）。每个导入的 API 在 IAT 中都有其自己的保留位置，其中，导入函数的地址由 Windows 加载程序编写。最后一点特别重要，加载模块后，IAT 包含调用导入的 API 时调用的地址[46]。

IAT 的优点在于，PE 文件中只有一个地方存储了导入的 API 地址。无论分散通过多少个源文件调用给定的 API，所有调用都将通过 IAT 中的同一函数指针进行[47]。

下面将会通过一个实例来说明导入表的作用，有两种情况需要考虑：直接跳转和间接跳转，在直接跳转中，以执行 CALL DWORD PTR [0x00405030] 为例，对导入的 API 调用如图2-3所示。

CUP 的 EIP 寄存器将会被设定为 0x00405030, 对导入的 API 使用间接跳转调用时，如图2-4所示。

执行的指令如下:

CALL 0x0040100C

•••

0x0040100C:

JMP DWORD PTR [0x00405030]

在这种情况下，CALL 将控制权转移到一个中继地址，中继地址中存储

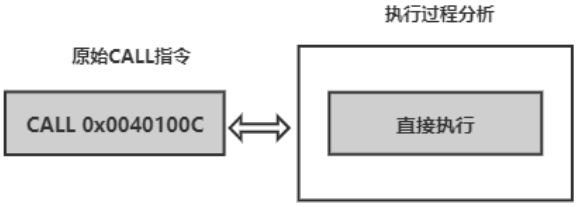

图 2-3　直接调用过程

Figure 2-3　Direct call procedure

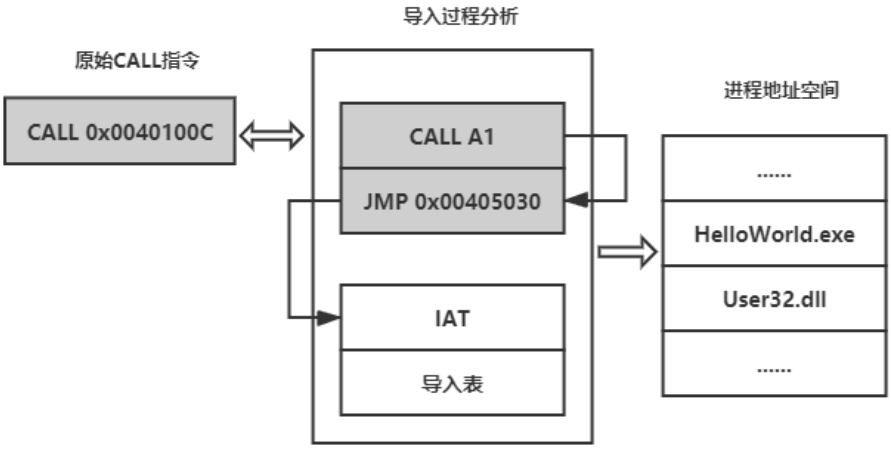

图 2-4　使用导入表间接调用过程

Figure 2-4　The procedure is called indirectly using the import table

着 API 需要跳转的绝对地址 0x405030。0x405030 属于 IAT 中的一项数据。简而言之，效率较低的导入 API 调用使用了五个字节的附加代码，由于额外的 JMP，执行时间更长。

这里使用了一个较低效率的方法来实现 CALL 指令，这是因为编译器无法区分导入的 API 调用和同一模块内的普通函数，对于编译器来说，在执行 CALL 指令时，将会发出如下的指令：CALL XXXXXXXX，其中 XXXXXXXX 是实际的代码地址，稍后将由链接器填写, 这个 CALL 指令不是通过函数指针进行的，而是一个实际的代码地址，为了让程序正常执行，链接器需要有一段代码来代替 XXXXXXXX，最简单的方式是将调用指向 JMP 存根[48]。

JMP 的存根就来自导入函数的导入库，如果检查这个导入库，并且检查如导入的 API 名称相关的代码，会发现它是一个 JMP 存根。这意味着默认情况在没有任何干预的情况下，导入的 API 调用将使用效率较低的形式[49]。

PE 文件由许多标题和节组成，这些标题和节告诉动态链接程序如何将文件映射到内存中。可执行映像由几个不同的区域组成，每个区域需要不同的内存保护。因此，每个部分的开头必须与页面边界对齐。例如，通常.text 节保存程序代码被映射为 execute / readonly，而.data 部分被映射为不执行/读写。但是，为避免浪费空间，不同部分在磁盘上未按页面对齐。动态链接器的部分工作是根据标题中的说明，将每个部分分别映射到内存，并为生成的区域分配正确的权限。

值得注意的一部分是导入地址表 IAT，当应用程序在另一个模块中调用函数时，它用作查找表。它可以采用按序导入和按名称导入的形式。由于编译后的程序无法知道其依赖的库的存储位置，因此，每当进行 API 调用时，都需要进行间接跳转。当动态链接器加载模块并将它们连接在一起时，它会将实际地址写入 IAT 插槽，以便它们指向相应库函数的存储位置。尽管这会增加模块内调用的开销，从而导致性能下降，但它提供了一个关键的好处：需要写时复制的内存页数装入程序更改的内容最小化，从而节省了内存和磁盘 I / O 时间。如果编译器提前知道调用将是模块间的，则可以生成更多优化的代码，这些代码只会导致间接调用操作码。

在导入表中，PE 文件通常不包含与位置无关的代码。相反，它们被编译为首选的基地址，并且编译器/链接器发出的所有地址都提前固定。如果 PE 文件无法在其首选地址加载，操作系统将重新修复它。这涉及重新计算每个绝对地址，并修改代码以使用新值。加载程序通过比较首选加载地址和实际

加载地址并计算增量值来完成此操作。然后将其添加到首选地址，以提供存储位置的新地址。基地搬迁存储在列表中，并根据需要添加到现有存储位置。现在，生成的代码是该过程专用的，不再可共享，因此在这种情况下，DLL 的许多节省内存的好处都丧失了。这也大大降低了模块的加载速度。因此，应尽可能避免重新基准化，Microsoft 提供的 DLL 具有预先计算的基址，以免重叠。因此，在无基准的情况下，PE 具有代码效率非常高的优点。这与使用完全与位置无关的代码和全局偏移表的 ELF 形成对比，后者在执行时间之间进行权衡以降低内存使用量。

在 Windows NT 操作系统上，PE 当前支持 x86，IA-32，x86-64（AMD64 / Intel 64），IA-64，ARM 和 ARM64 指令集体系结构。在 Windows 2000 之前，Windows NT 因此也包括 PE 支持 MIPS，Alpha 和 PowerPC ISA。由于 PE 在 Windows CE 上使用，因此它继续支持 MIPS，ARM 和 SuperH ISA 的多个变体。

以上已经对 PE 文件结构需要修改的地方进行了分析，本文将按照上述内容，对 PE 文件进行修改，实现反调试加密和虚拟机加壳功能。

2.2 PE 文件对抗逆向分析的常见方法

目前市面上的大部分需要都需要软件保护措施，除去普通的方式例如序列号验证、网络注册、花指令等方式，最为普遍的就是软件加壳，软件加壳的根本目的是防止破解者对软件进行逆向分析，从而阻止他们的非法修改和逆向编译，从而保护了上述序列号验证和网络注册等方式，也可以说软件加壳的保护层面更高，如果保护得当，可以从根本上杜绝软件被非法修改。软件保护界也存在一句名言——没有不能脱的壳，所以说本课题研究的抗逆向目标也是在某种程度上加大破解的破解难度，延长他们的破解时间。软件加壳简而言之，将壳与程序通过某种方式混合在一起或进行代码变形，从而让破解者不能正确分辨软件逆向分析无用的壳代码和软件功能流程代码。软件加壳后的程序如图2-5所示。

软件加壳按照方法和目的可以分为两大类，第一类是压缩壳，第二类是保护壳。保护壳是本课题的主要研究内容，这种壳的目的是加大破解者的逆向分析难度，代码会将功能代码和外壳代码混淆，可使用密钥对称加密或者非密钥加密两种方式，密钥对称加密则是在软件购买时，商家为用户提供一

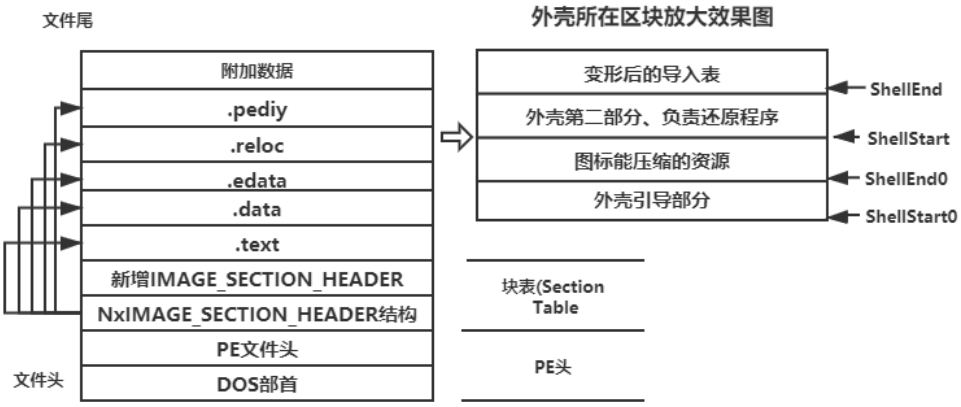

图 2-5　加壳后的程序的结构

Figure 2-5　The structure of the program after shell

个私钥，用户只能通过这个私钥打开这个程序，其中的原理与普通的加密算法相仿，但是由于这种方式容易让破解者对私钥和加密代码进行分析，反而减小了破解难度，所以这种方式逐渐被淘汰，非密钥加密则是使用密码学中难以逆向分析的算法对软件进行保护。第二类的压缩壳的目的是为了减小软件的体积，但是由于软件使用了压缩算法，解密和程序运行时同时进行的，会增加程序的运行时间，压缩算法也是不断在优化当中，在时间和空间上进行取舍，这种方式基本没有保护作用，因为压缩壳并不妨碍软件破解者进行逆向分析，利用 ESP 栈平衡原理，很容易得到软件 OEP，修复 IAT 表后会直接得到解压后的软件，然后再进行破解序列号的逆向分析，所以压缩壳对软件的保护作用形成虚设。

软件加壳按照保护分类可以分为两种方式，一种是本文所采用的虚拟机加壳保护，第二种是采用可执行压缩的方式。所谓可执行压缩是任何手段压缩的可执行文件，并用解压缩代码的压缩数据组合成单个可执行文件。执行此压缩的可执行文件时，解压缩代码会在执行之前从压缩的代码重新创建原始代码。在大多数情况下，压缩的保护方式是透明的，因此可以与原始文件完全相同的方式使用压缩的可执行文件。可执行压缩器通常被称为“运行时打包程序”,“软件打包程序”,“软件保护程序”，甚至是“多态打包程序”和“混淆工具”。压缩的可执行文件可以被视为自解压存档，其中压缩的可执行文件与相关的解压缩代码一起打包在可执行文件中。某些压缩的可执行文件

可以解压缩以重建原始程序文件，而无需直接执行。可以用于执行此操作的两个程序是 CUP386 和 UNP。大多数压缩的可执行文件在内存中对原始代码进行解压缩，并且大多数需要更多的内存才能运行因为它们需要存储解压缩器代码，压缩的数据和解压缩的代码。此外，某些压缩的可执行文件还具有其他要求，例如那些在执行前将解压缩的可执行文件写入文件系统的要求。

2.2.1 压缩壳

软件程序的压缩壳软件与我们通常意义上的压缩软件如 RAR 不同，RAR 软件压缩之后的文件格式不能被 Windows 内核程序加载器识别，压缩之后的文件结构已经不是 EXE 程序，所以不能成功运行。

压缩壳的产生时间较早，早在 DOS 时代，受到当时的硬件设备的限制，计算机的内存和硬盘需要严格计划使用，也受到移动存储介质的限制，软件开发者不得不想出办法来减小应用软件的占用空间，于是程序压缩技术产生了。对可执行文件进行压缩后，其内部的功能代码结构大部分发生改变，其中的汇编代码和硬编码地址已经和初始状态不能直接进行反汇编，加大了破解难度，起到了一定的反破解作用。压缩同一款产品，压缩比例越高，程序的功能代码变化越大，破解的难度也会随之提高。

压缩算法通常是使用市面上的压缩引擎，由于解压和程序的运行是同时进行的，解压会增加程序的运行时间，所以解压效率是衡量一款解压算法的重要标志，压缩的速度可以慢下来，但是解压速度需要不断的提高，如果解压速度太慢会影响用户体验。有代表性的压缩壳有 ASPack、UPX 和 PECompact 等。

ASPack 是 Windows 下的应用程序压缩软件，市面上很多软件都使用这款压缩软件，但是由于使用用户太多，导致研究的人也多，其中的保护作用已经越来越来，现在用这款软件的目的大部分是为了减小应用软件的体积，压缩壳通常可以将可执行文件和其他依赖文件打包在一起压缩，从而减少了文件的数量，ASPack 压缩后的程序是后缀为 EXE 的可执行文件，压缩后的程序可直接运行，并且程序的运行并不依赖 ASPack 这款软件，这也是与普通压缩软件差别最大的地方。

UPX(Ultimate Packer for eXecutables) 是一款针对 Windows 的对可执行文件进行加密的文件压缩器，压缩率高达 50-70%，大大减小了可执行文件的磁盘占用空间，同时也减小了移动介质和网络之间的传输时间。通过 UPX 压

缩过的程序同 ASPack 软件一样，不会产生功能损失会和压缩之前一样运行。其运行界面如图2-6所示。

图 2-6　UPX 加壳工具

Figure 2-6　UPX shell tool

2.2.2　保护壳

随着脱壳技术的发展，压缩壳的保护性能已经不能满足人们的需要，上述 ASPack、PECompact、UPX 等压缩壳软件已经形成专门针对的软件，使用压缩壳的应用程序都可以被一键脱壳，调试技术也不断提高，产生了诸多的反汇编工具 ODdbg、IDA Pro 等、调试工具 SoftICE、OllyDbg 等，随着硬件设备的性能不断提高，应用程序的占用空间和运行时间已经不是安全问题的瓶颈，防止软件被逆向、反调试技术已经成为研究的重点。于是，在加壳软件中会加入反调试的代码，比如化指令、结构异常处理机制、加密算法、代码混淆等技术，抗逆向技术在不断发展。

保护壳可以自动给应用程序添加安装包保护性外壳程序，并强制在应用程序运行之前确认安全令牌的存在和状态。保护壳还会对软件中的代码和数据进行加密，以防止进行逆向功能，使用这种保护方法，可以在不修改源代码的情况下应用软件保护。有代表性的保护壳有 ASPprotect、ACProtect、Armadillo 等。

ASProtect 是主要面向 Windows 用户的软件加壳保护软件，这款软件通过 RSA 加密算法生成唯一 KEY，通过对称加密的方式，为用户分配私钥，只有得到私钥的用户，才能在程序运行时对加密功能代码进行正确解密。内置反调试引擎，自己有一套反调试系统，在运行时会开启一个线程，检测自身是否被调试，如果被调试，则直接中断程序，给破解者的破解带来难度。其运行界面如图2-7

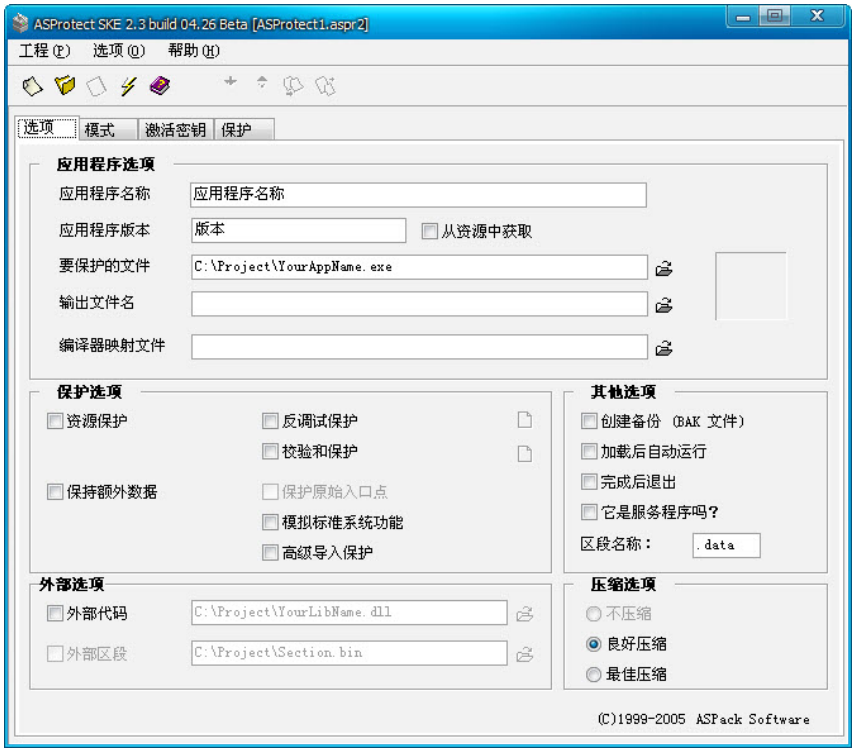

图 2-7　ASProtect 加壳工具

Figure 2-7　ASProtect packing tool

2.2.3　工具脱壳

被加壳软件在运行时是需要脱壳后才会正常执行，软件脱壳就是软件加壳的逆向操作。根据脱壳方式的不同，脱壳方式分为手工脱壳和工具脱壳，工具脱壳通常只适应于流行的加壳软件，越是流行的加壳工具，研究它的人越多，拥有高技术的破解者则可以完全研究明白加壳机制后，写出自动化脱壳机，但是这种自动化脱壳机通常只针对压缩壳。

工具脱壳通常主要分为两类：专业脱壳软件和通用脱壳软件。专业脱壳软件通常只针对一种或两种加壳软件，由于是针对性的软件，脱壳成功率也相对较高。通用脱壳软件通常针对保护性能低的压缩壳，它具有通用性，可以脱掉多种不同的壳。常见的脱壳软件有 UnPECompact、UnASPack、UPXShell 等。

2.2.4 手工脱壳

通常的压缩壳，都有对应的自动脱壳机，但只针对于压缩壳，由于保护壳的机制太过复杂，很难写出自动脱壳机，所有保护壳通常需要手工脱壳。

要想手工脱壳需要对 PE 文件以及程序加载后在内存中的数据非常了解。软件加壳技术的发展总是伴随着脱壳技术的进步，要研究脱壳技术，首先要分析壳的加载过程。壳的加载过程可以分为五个步骤，定位壳需要的所有 API 地址、解密函数对块 (Section) 的操作、重定位、EIP 设置为 OEP(Original Entry Point)、HOOK-API, 在程序被脱壳之后，加壳进程将会把 CPU 的控制权还给原程序，此时原程序就会像没有加壳时的状态正常运行，原本被加密过的各种节区块都会被还原。而此时，则是脱壳程序最关键的时机，此时的程序在内存中的形态和程序没有被加壳过的形态几乎完全一致[31]。

手动脱壳通常分为三个步骤：一是找到程序真正入口点 OEP; 二是 dump 下内存映像文件；三是输入表的重建过程。

2.2.5 壳的类型分析

在手工脱壳之前，通常要对软件进行加壳分析，一是分析是否加了壳，二是可以分析出此程序使用了哪种壳或者本程序是用哪种语言编写。这时通常使用市面上流行的 PEiD 或者 FileInfo 等工具。PEiD 的图形界面如图2-8所示。

将程序加载到 PEiD 中，此程序通常会给出很多有用的信息，比如壳的类型和本程序使用的编译器。此程序使用特征码做出识别判断，每种壳都有固定的特征码，只要文件搜索特征码，大部分情况下都可以识别成功，但是对于那种小众加壳或者未被公布的加密壳却无能为力，这也是本课题最终的研究目标。本软件提供了一个扩展接口 USERDB.txt, 用户可以自定义一些已知的特征码，通过这种方式可以识别出大部分的壳。

加壳与脱壳总是在不断博弈中，正因为 PEiD 是使用特征码方式识别加壳

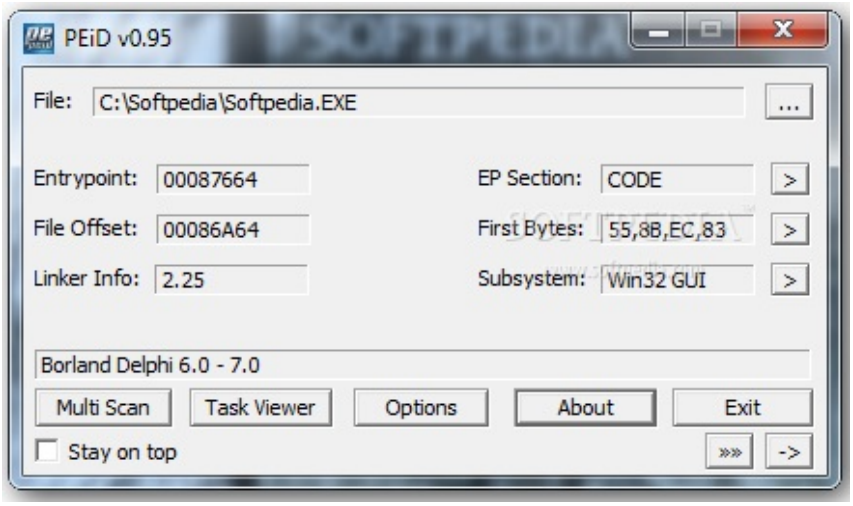

图 2-8　PEiD 侦壳工具

Figure 2-8　PEiD debugger

类型，加壳软件也可以伪造特征码，比如加壳软件在壳中放入多种不同类型的加壳软件特征码，这时 PEiD 就会失去它的作用，这种方式也被称为”PEiD 欺骗”。

2.3　本章小节

本章节对虚拟机加壳和二进制代码混淆的相关技术进行了阐述，并对 PE 文件结构进行了分析，对其中的 DOS 头、PE Header、相对虚拟地址等重要结构进行了详细阐述，并简要说明了反调试的原理，说明了在汇编语言和硬编码语言的层面上对 Windows 下的执行文件进行修改的相关技术，从而可以保证修改文件的正确性和可重用性。

第 3 章　多样化 Handler 虚拟机加壳的研究与实现

在软件反逆向工程中的虚拟机和虚拟机模拟软件是完全不同的东西，本章描述的虚拟机保护方法是类似于一种解释执行系统，它和解释型语言如 Python、Lua、Ruby 等有相似的地方。

对于 PE 文件结构，在2.1节有详细的分析结果，对于虚拟机加壳比较重要的位置是节表 Section Table，原始 PE 程序的二进制硬编码通常是按照顺序排列在多个节表中的，在通过分析 PE 文件的 DOS Header 和 PE Header 后，可以通过里面的相关信息，计算出 Section Table 的起始偏移地址，得到代码区域的地址后，将此部分标记为待保护汇编的指令流的起始区域，由次区域开始到下一个.rdata 节截止，对原始 PE 程序的二进制硬编码进行加密保护。

本章节将会设计一个在 PE 程序中的虚拟机壳，程序的 OEP(Original Entry Point) 程序的入口点将会被改变为虚拟机的入口点，进行虚拟机的执行区域。虚拟机将会进行一系列的初始化工作：对原有的汇编代码进行整理、提取出其中的导入表信息、建立 Handler 等。建立调度器 dispatcher，调度器会将原本的寄存器符号统统压入栈中，把 esi 指向虚拟机字节码的起始地址，ebp 指向当前的真实栈区域，edi 指向 VMContext 虚拟环境结构，在执行上述过程后，虚拟机初始化环境将会加载完毕。

3.1　虚拟机保护方法基本原理

首先虚拟机保护系统会将可执行程序反汇编为 CUP 可以理解的 X86 指令流，做出一套 X86 指令和字节码之间的映射，再讲 X86 指令流的汇编代码转换为字节码，此时可执行程序已经完成了压缩阶段。最后，虚拟机保护会系统会将解释器内嵌到可执行程序中，使得被保护的代码在程序执行时可以被正确翻译后执行，整个流程类似于 Java 的虚拟机。

具体保护步骤如下:

Step1 将带保护程序使用反汇编引擎转换为汇编指令;

Step2 将汇编指令转换为一对一的字节码;

Step3 建立 Handler 表和跳转表;

Step4 将上述三个步骤产生的文件打包为可执行文件，使用虚拟环境结构

VMContext 存储寄存器中的值，将字节码、虚拟环境结、Handlers、Dispather 打包为一个新节，并添加到可执行程序中，虚拟机代码保护机制的流程如图3-1所示。

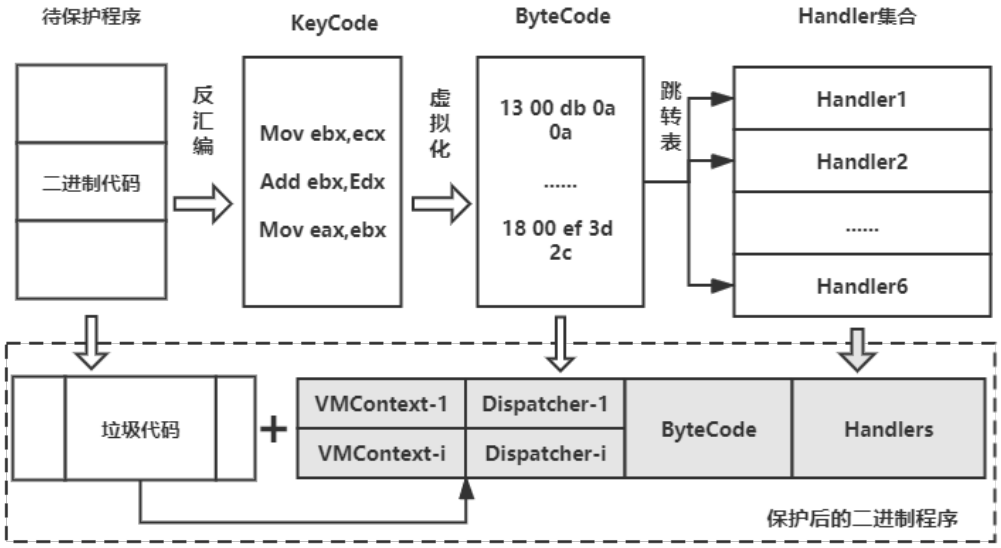

图 3-1　虚拟机代码保护机制

Figure 3-1　Virtual machine code protection mechanism

3.1.1　相关反汇编工具

反汇编器是把高级语言写成的程序翻译成汇编语言的工具，反汇编程序的目标是高级语言如 C/C++、VB，而不是汇编语言，它通常输出为了便于阅读而不是适合汇编程序输入而设置的格式，反汇编器会给阅读反汇编代码者提供大量的可参考数据[50]。

本课题采用的是 OllyDbg 引擎提供的开源反汇编器，在进行反汇编过程中，它可以自动跟踪寄存器、识别函数、API 调用，给阅读者带来很大的方便，由于其易用性和可用性，此软件在分析恶意代码时也很有用[51]。

3.1.2 X86 指令

按照 x86 体系结构，按照功能分为流指令、普通指令、栈指令、和不可模拟指令 4 类[52]。

流指令是指跳转指令、函数调用指令、返回指令等改变文件执行流程的指令。

普通指令指 add、sub、mov 等加减运算和数据传输指令。

栈指令指压栈指令弹出栈指令等操作栈空间的指令。

不可模拟指令指无法被模拟的指令，这类指令通常硬件相关，比如 int3、in、out 等指令。

指令分类如表3-1所示。

表 3-1 X86 汇编指令分类

Table 3-1 X86 assembly instruction classification

无操作数指令		单操作数指令	双操作数指令		多操作数指令
AAA	STC	CALL	ADC	XADD	IMUL
AAD	STD	JMP	ADD	XCHG	SHLD
AAM	STI	JCC	AND	CMP	SHRD
AAS	INSB	LOOP	MOV	CMPS	-
CBW	INSW	LOOPE	OR	LEA	-
CDQ	INSD	LOOPNE	RCL	MOVSX	-
CLD	LAHF	INC	RCR	MOVZX	-
CLI	LODSB	DEC	ROL	-	-

基于虚拟机 Handler 动态加密的情况大致如图3-2所示。其中 VStartVM 部分初始化虚拟机，VMDispatcher 调度每一个 Handler。在程序运行时，字节码就是操作系统需要识别的二进制代码，VMregisters 就是操作系统的调度器，不同的 Handler 对应不同的汇编代码，相对应的就是机器码。

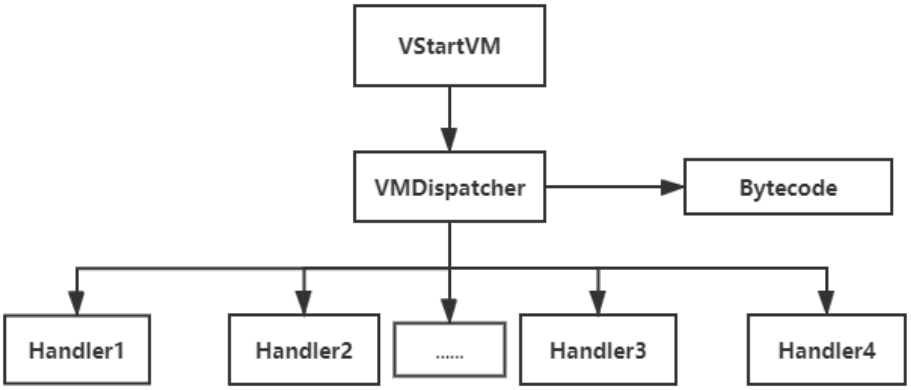

图 3-2　虚拟机执行时的情况

Figure 3-2　When the virtual machine executes

3.2　调用约定和框架

3.2.1　虚拟环境

解释器在执行时识别的是字节码，所以需要使用一个结构来保存 CPU 中寄存器的值，其结构设计具体如图3-3所示[53]。

```
struct VMregisters
{
    DWORD v_eax;
    DWORD v_ebx;
    DWORD v_ecx;
    DWORD v_edx;
    DWORD v_esi;
    DWORD v_edi;
    DWORD v_ebp;
    DWORD v_efl;
}
```

图 3-3　虚拟环境结构

Figure 3-3　Virtual environment structure

使用 VMregisters 结构体来存储 eax,ebx 等重要的寄存器，需要注意的是，这个结构体中不需要存放 esp 这个寄存器的，esp 一直指向栈顶，而程序的执

行本身需要栈空间的，esp 寄存器的值就是 ebp 寄存器的值。

3.2.2 调度器

MbeginVM 将原程序的环境中大部分寄存器的值压栈之后会建立一个 VMDispatcher 标签，Handler 会循环执行这部分代码，如图3-4所示的代码 MbeginVM 部分也称为调度器。

```
VbeginVM:
    push eax
    push ebx
    push ecx
    push edx
    push esi
    push edi
    push ebp
    pushfd
    mov esi,[esp+0x20]                  ;参数，字节码开始的地址
    mov ebp,esp                         ;ebp就是栈
    sub esp,0x200
    mov edi,esp                         ;edi就是VMContext
    sub esp,0x40                        ;这时的esp就是VM使用的栈
VMDispatcher:
    movzx eax,byte ptr [esi]            ;获得Bytecode
    lea esi,[esi+1]                     ;跳过这个字节
    jmp dword ptr [eax*4+JUMPADDR]      ;调到Handler执行处
```

图 3-4　Handler 核心执行代码

Figure 3-4　The Handler core executes the code

首先，调度器会将所有的 CPU 寄存器压栈，esi 是设计的字节码的开始地址，ebg 是原程序的栈空间，edi 是虚拟环境结构,esp 减去 50h（不固定）就是虚拟机使用的栈空间地址。将虚拟机的虚拟环境结构和栈空间都放在了当前栈的 300h 处。

由于栈空间是变化的，这块区域可能在执行时被使用栈空间的语句覆盖，所以要设定一个机制保护本区域，当虚拟机在执行有关栈空间的指令时，需要有代码检查使用的空间是否覆盖到这块存放的数据，如果覆盖，则将这块结构向下移动。从“movze eax,byte ptr[esi]”指令开始读字节码。读取一个字节，就到跳转表中找到相应的 Handler 并执行相关语句。

这里会产生三个寄存器和某些值一直是对应关系，edi 始终和虚拟环境结构的值相同，esi 是当前字节码的地址，ebp 一直是真实栈的地址。这三个对应关系在整个虚拟机执行过程中是一直成立的。通常情况下，这三个寄存器只能用作上述三个功能，不能使用做其他用途，如果必须使用则需要使用 pushad 来保护起所有的寄存器，在使用后需要使用 popad 恢复所有寄存器的值。

3.3 多样化 Handler 设计

本文所说 Handler 与 Windows 中的句柄 Handler 不同，本文的 Handler 指把某一段代码表示的功能给包装起来，形成可重复使用的功能结构块，它通常是被虚拟机保护中的调度器使用。Handler 结构是虚拟机保护系统的核心部分，通常也是逆向攻击者的重点攻击对象，Handler 结构中存放着字节码和汇编代码之间的转化过程，虽然可能这个转化过程不是通过一个 Handler 实现的，可能是由多个有着不同功能的 Handler 共同实现的，但是保护 Handler 等于保护了核心的翻译模块。所以，对 Handler 进行保护显得尤为重要，本节将会描述一种通过设计多样化的 Handler 使得逆向攻击者难以逆向分析字节码转化为汇编代码之间的翻译过程。

3.3.1 辅助和普通 Handler 的实现

Handler 根据指令功能的不同可以分为两大类：一类是辅助 Handler，通常存放着维护栈帧结构、CPU 寄存器、堆结构等比较重要的 Handler。另一类是普通 Handler，通常存储着算术指令和跳转等基础指令。

3.3.1.1 辅助 Handler

辅助 Handler 的示例代码如图3-5所示，实现了 PUSH 寄存器指令、PUSH 标志位指令、POP 寄存器指令。

3.3.1.2 普通 Handler

在图3-5的基础上，可以实现普通指令 Handler，其实现代码如图3-6所示。

将一个普通的汇编指令分为两个部分进行，指令的实现由普通 Handler

```
vPushReg32:
    mov eax,dword ptr[esi][
    add esi,4
    mov eax,dword ptr [edi+eax]
    push eax
    jmp VMDispatcher
vPushImm32:
    mov eax,dword ptr [esi]
    add esi,4
    push eax
    jmp VMDispatcher
vPopReg32:
    mov eax,dword ptr[esi]
    add esi,4
    pop dword ptr [edi+eax]
    jmp VMDispatcher
```

图 3-5　辅助 Handler 执行核心代码

Figure 3-5　The helper Handler executes the core code

```
vadd:
    mov eax,[esp+4]          ;取源操作数
    mov ebx,[esp]            ;取目的操作数
    add ebx,eax
    add esp,8                ;平衡栈
    push ebx                 ;压入栈
```

图 3-6　普通 Handler 执行核心代码

Figure 3-6　The normal Handler executes core code

处理，目的操作数和源操作数都交由栈 Handler 处理。这样可以用更少的模拟 Handler 实现真正的汇编指令。

举一个例子便于理解。例如，sub 指令的形式通常有“sub reg,imm”，“sub reg,reg”，“sub mem,reg”等，这里采用现将两个操作数使用栈 Handler 处理，将两个操作数存放在栈中，然后再调用“vsub Handler”，在这时两个操作数已经以立即数的形式在栈中，则“vsub Handler”可以直接默认栈顶就存放着它需要处理的数据，接着可以直接对栈顶这两个立即数进行减操作。

3.3.2 多样化 Hanlder 实现

本课题的核心保护模块则是使用了多样化 Handler 模块，所谓多样化，就是将同一个 Handler 功能的模块采用多种设计方案，比如转移指令、算术指令、CALL 指令等，由于汇编语言的特性，往往相同的功能可以采用多用方式实现，一旦相同功能采用不同汇编代码实现，逆向攻击者在分析虚拟机时会得到不同的 Handler 结果，并且每次分析的结果可能都不一样，使得逆向攻击者不能获得正确的 Handler，也就无法掌握由汇编代码到字节码的翻译过程，通过这种方式，可以有效加大逆向攻击者的逆向分析难度。

3.3.2.1 转移指令多样化

转移指令一共包括四种指令，分别是无条件转移、条件转移、CALL 和 RETN。

由于无条件转移在汇编语句中就是 JMP 指令，而 JMP 指令的核心实现就是改变寄存器 EIP 的值无条件跳转指令 JMP 的 Handler 比较简单，如图3-7所示。

```
vJmp:
    mov esi,dword ptr [esp]      ;[esp]指向跳转的目的地址
    add esp,4
    jmp VMDispatcher
```

图 3-7 jmp 的 Handler 实现代码

Figure 3-7 The Handler implementation code for JMP

JMP 指令的功能太过单一，多样化的实现意义不大，所以本节将会介绍条件转移的多样化方法。X86 的条件转移指令和条件传输指令是可以通过某

种方式对应起来的，其比较如表3-2所示。

表 3-2　条件转移指令和条件传输指令

Table 3-2　Conditional transfer instruction and conditional transfer instruction

条件转移指令	条件传输指令
jne	cmovne
ja	cmova
jb	comvae
jbe	comvbe
je	comve
jg	comvg

所有条件跳转指令都有对应的条件传输指令，图3-8是通过设计之后，条件跳转指令和条件传输指令的实现方法。

```
vjne:
    cmovne esi,[esp]
    add esp,4
    jmp VMDispatcher
vja:
    cmova esi,[esp]
    add esp,4
    jmp VMDispatcher
vjae:
    cmovae esi,[esp]
    add esp,4
    jmp VMDispatcher
....:
    ................
    ................
    ................
```

图 3-8　条件跳转指令的另一种实现

Figure 3-8　Another implementation of conditional jump instruction

这里再提出一种方式实现转移指令多样化，可以使用 X86 平台下的标志寄存器和无条件转移指令实现另一种跳转命令模拟跳转，原理是条件转移命令本身就是判断标志寄存器中的值实现跳转的，而存在其他指令可以直接读

取标志寄存器的值，直接判断标志寄存器中的值，加上无条件跳转 JMP 的配合，就可以实现条件跳转。

3.3.2.2 算术指令多样化

在 X86 指令体系中存在一些不同指令可以用同一种指令去实现的情况，它们的目的都是将 esi 指令加一，但是却是 inc 指令和 add 指令的不同实现方式, 示例代码图3-9所示。

```
inc esi
add esi,1

dec esi
sub esi,1
```

图 3-9 add 指令和 sub 指令的另一种形式

Figure 3-9 Another form of the Add and SUB directives

X86 体系中存在很多这种指令，实现原理于图3-9类似，在 Handler 多样化实现时都可以利用这种机制，实现汇编到字节码翻译过程中的 Hanlder 指令多样化。

具体操作方式则是如3.3.1.2节所述，将操作指令、目的操作数、源操作数分别使用不同的 Handler 实现。

3.3.2.3 CALL 指令多样化

CALL 指令虽然作为跳转指令的一种，但在功能实现方面与 JMP 衍生出来的指令大不相同，CALL 指令通常是调用一个函数，虚拟机中的代码运行通常在一个栈中进行，但是 CALL 指令由于是进到另外一个函数中执行，所以会需要把控制权交给真实的 CPU，举例代码如图3-10所示。

除第三条指令外，其他指令都是在一个栈层次上进行，不需要更换栈，但是第三条指令“CALL anotherfunc”是对其他函数的调用，它的栈帧会改变，所以必须将控制权交给其他代码。

汇编语言中，CALL 指令先把当前执行指令的下一条指令压入栈中，然后再跳转到目标函数的首地址处，代码如图3-11所示，将控制权交给其他代码之后必须再跳回虚拟机中，所以使用代码如图3-12所示，这是返回虚拟的

```
    mov eax,1234
    push 1234
    call anotherfunc
theNext:
    add esp,4
```

图 3-10　call 指令示例汇编代码

Figure 3-10　Call instruction sample assembly code

一段代码，NextCode 代表 THEnext 之后代码的字节码。只需要将 thenext 的地址修改为 nextVM 的地址，便可以再次模拟一个 CALL 指令，vcall 指令的伪代码如图3-13所示。

```
push the Next
jmp anotherfunc
```

图 3-11　目标函数的首地址

Figure 3-11　The starting address of the target function

```
theNextVM:
    push theNextByteCode
    Jmp VStartVM
```

图 3-12　跳回虚拟机代码

Figure 3-12　Jump back to the virtual machine code

3.4　本章小结

本章采用虚拟机保护的方法对 Windows 下可执行文件进行了保护，并提出一种多样化 Handler 设计思路，实现了该方法，对转移指令、算术指令、CALL 指令进行了多样化设计，由于 X86 架构和汇编语言的特性，指令本身的实现不是唯一的，将 Handler 分为辅助 Handler 和普通 Handler，以低耦合

```
vcall:
    push all vreg
    pop all reg
    push 返回地址
    push 要调用的函数的地址
    retn
```

图 3-13　vcall 指令的伪代码

Figure 3-13　The pseudocode for the VCall directive

的特点设计了汇编语言和字节码之间的转换协议，本章的试验将会在后文进行详细描述。

第 4 章　静态分析混淆技术研究与实现

逆向攻击者在分析软件时通常会通过动态分析工具如 OllyDbg、WinDbg、SoftICE 等工具和静态分析工具如 IDA 工具来分析软件，动态分析工具是在运行是分析可执行程序，可以观察到程序执行时的寄存器、堆栈、内存空间等的所有数据。静态分析工具则是直接反汇编源码，它的优点是对代码阅读者提供更多的信息，高级功能的静态分析工具比如 IDA 甚至可以直接由汇编代码得到与源码类似的 C 语言代码，函数的所有跳转都给出详细信息。

本节将针对静态分析工具对软件进行反调试保护，提出一种建立索引进行函数间基本块交换的混淆算法，最后根据提出的算法实现了一个 PE 文件保护器。

4.1　常用的反调试技术

常见的反调试技术有使用花指令、文件完整性检验、代码与数据结合、调试器检测等。

逆向工程是研究程序以获得有关其工作方式和使用的算法的封闭信息的过程，尽管软件逆向可用于合法的研究目的，尤其是恶意软件分析或未记录的系统研究，但通常认为是黑客将其用于非法活动，调试与反调试一直都是软件保护者和软件破解者之间的相互博弈，虽然理论上不存在不能被逆向分析破解的程序，但是软件保护者们依然采用各种各样的方式来实现对软件的保护，逆向工程的工作中也没有最优解。本课题也只是在加大破解难度的原则上对软件进行保护，并加入自己的加密算法，最终加大破解难度或者引导破解者到错误的方向。本节将简述常见的的基本保护措施，并会加以改进。

常见的分析软件的方法有：一、使用数据包嗅探器进行数据交换分析，以分析通过网络交换的数据。二、软件二进制代码反汇编以汇编语言形式列出其代码。三、反编译二进制或字节代码，以高级编程语言重新创建源代码。本课题考虑了流行的反破解和反逆向工程保护技术，即 Windows 中的反调试方法。我们一开始就应该提到，不可能完全保护软件免受逆向工程。各种反逆向工程技术的主要目标只是使过程尽可能复杂。

4.1.1 花指令

花指令是一种隐藏不需要进行反向工程的代码块或其他功能的方法。在实际代码中插入一些垃圾代码还可以确保原始程序的正确执行，并且程序无法很好地进行反编译，难以理解程序的内容并难以达到使代码混乱的效果，使用某种方式排布数据和代码，在汇编代码中插入一些“数据垃圾”，对反汇编软件进行干扰。

第一种花指令利用了汇编指令的长度。不同的体系结构的机器指令长度并不相同，有多操作码指令和单操作码指令的区分，多操作码指令需要确定这条指令第一个字节的起始位置，由于长度固定，所以指令的结尾也是固定的，否则可能会被反编译器翻译为其他的或者错误的指令。图4-1是一段汇编代码。

```
start_:
    xor eax,eax
    test eax,eax
    jz label1
    jnz label1
    db 04f7h
label1:
    xor eax,3
    add eax,4
    xor eax,5
    ret
```

图 4-1　花指令汇编源码

Figure 4-1　Flower instruction assembly source code

对源程序使用 nasm 编译，使用 W32Dasm 反汇编，结果图4-2。

```
:00401000 33C0       xor eax,eax
:00401002 85C0       test eax,eax
:00401004 7403       je 00401009
:00401006 7501       jne 00401009
:00401008 E883F00383 call 83440090
:0040100D C00483F0   rol byte ptr [ebx_4*eax],F0
:00401011 05C3000000 add eax,000000C3
```

图 4-2　反汇编硬编码代码

Figure 4-2　Disassemble hardcoded code

W32Dasm 使用的反汇编器是逐行反汇编的，代码中的 04F7h 不是任何汇编指令，干扰了 W32Dasm 使用的 Linear Sweep 反汇编引擎，对指令的起始位置做出了错误的判断，使反汇编的跳转指令的跳转位置无效。比如 00401009h 这个位置属于指令的内部，不具有反花指令引擎的反汇编器很容易给调试者带来无效的分析信息。

反汇编引擎最关键的问题之一是代码与数据的区分，由于汇编代码的指令长度是不同的，并且代码之间存在绝对地址和相对地址，跳转命令可能使用这些地址来直接跳转或间接跳转，所有反汇编引擎要对所有指令的长度和功能有确切的认知，从而保证反汇编结果正确性。

第二种花指令是利用了将 JMP 指令修改为 JE+JNE（JX+JNX 或者 JZ+JNZ 等）实现的。

在汇编语言中存在着大量的条件跳转，比如 JNE 是比较两个操作数，如果不相同，则跳转，如果不同，则不跳转，JE 与上述指令功能相反。条件跳转都是成对出现的，比如 JZ/JNZ、JX/JNX 等。而 JMP 指令则是无条件跳转，使用方法为 JMP XXXX，指令运行时永远会跳转到 XXXX 处，JMP 这条指令的地址和 XXXX 代表的地址中间的代码可能永远不会被执行，这部分代码叫做 DEAD CODE。JE+JNE 型代码与 JMP 代码同理，如果将 JE 和 JNE 代码同时使用，将会实现无条件跳转。

```
:00402665 jz short near ptr loc_402669+1
:00402667 jnz short near ptr loc_402669+1
:00402669 loc_402669:
:00402669 db47h
:00402669 xor eax,eax
:00402669 cmp dword ptr [ebp-0ch],0
```

图 4-3　跳转花指令

Figure 4-3　Jump flower instruction

如图4-3所示，无论标志位是否为 1，代码都会跳转到 00402669 处执行，代码都会跳转到 00402669 处执行 47h 指令，也就是无用代码。db 47h 指令则会给反编译器带来困扰，因为这行代码是在代码区域，但是不能被识别为指令。

第三种花指令的抗逆向效果较低，但是这部分指令不会影响其他部分有用程序，对程序的影响较小，植入的成本更低，程序不容易出现错误。这种

花指令通常构造一部分没有意义的算术运算。

4.1.2 文件完整性检验

在抗逆向方案中，文件完整性检验是最基础的方案，通常也被最先考虑并采用。文件完整性检验就是在软件运行过程中，对本文件进行自身完整性的检查，抵御逆向分析者对文件进行修改，可以达到一定的抗分析目的。

文件完整性检验原理就是在发布时预先计算文件的哈希值，存放在程序的启动代码中，在文件每次运行时，首先对当前文件的某个模块或者全部模块进行哈希运算得出哈希值，与预先计算好的哈希值对比，如果相同则成功执行，如果不同则直接终止程序。

文件完整性检验通常使用三种方式，分别是：磁盘文件校验的实现、内存映像校验和校验和。

其中的校验和方式安全性能较差，在 PE 文件的 IMAGE_OPTIONAL_HEADER 中有一个校验和 (Checksum) 字段，这个数据结构中已经存放着整个文件的校验和，相当于 PE 文件对自身的一种保护，但由于这种保护广为人知，现在几乎已经失去它的作用，只要逆向分析者发现有校验和的检查，自己手动修复校验和或者使用工具修复都可以成功绕开这个限制。

4.1.3 调试器检测

4.1.3.1 使用 IsDebuggerPresent

提出的反调试方法是基于 IsDebuggerPresent 函数。此函数检测调用模式是否正在由用户模式调试器调试。图4-4的代码显示了基本保护的示例：

4.2 建立索引进行函数间基本块交换的混淆算法

本节将会提出一种对混淆算法，此算法对二进制函数分块，进行交换，并建立函数块之间交换记录的索引。目前市面上大部分混淆算法都是对一个函数的二进制进行加密处理，比如对函数内部的汇编代码进行花指令处理或进行分块然后进行交换。由于函数内部的二进制代码可能极其复杂，函数之间

```
int main()
{
    if (IsDebuggerPresent())
    {
        cout << "Stop debugging program!" << endl;
        exit(-1);
    }
    return 0;
}
```

图 4-4　IsDebuggerPresent 函数

Figure 4-4　IsDebuggerPresent function

进行块交换的主要困难在这几个方面：一、编程难度大，由于只能看到函数的二进制代码；二、虽然通过反汇编处理后可以看到汇编代码，但是在编程时需要分析各条指令的长度、函数执行时的栈帧结构的保护、执行时内存堆空间的变化情况。

本节将会提出相对简单的二进制混淆方式，只考虑移动函数内部二进制命令中较易处理的汇编命令，比如 MOV、SUB、ADD 等不影响栈和堆的汇编指令，且这些指令的长度大部分是固定的，后文将会实现这种算法，并进行分析。

4.2.1　基本思想

一个可执行程序中包含多个函数，函数之间可能进行相互调用，首先要对函数进行分块处理，把每个函数切分为可交换块和不可交换块，可交换块数量大于等于一，在移动时只会移动可交换块，故不考虑不可交换块块数。通常，函数的指令都是连续的，函数的调用流程反映了程序的执行时的运行结构。

在记录交换索引的基础上，对函数之间的可交换块进行了的交换，在可执行文件的尾部建立一个新节来存储索引信息。

如图4-5所示，图中展示了三个函数块的交换过程，并在左侧建立索引，在恢复时程序会根据此索引进行函数执行流程恢复。函数 X、Y、Z 分别包含了 4 个基本块，函数 X 的基本块 X_Block_2、X_Block_3、函数 Y 的基本块 Y_Block_1、Y_Block_3、函数 Z 的基本块 Z_Block_1 和 Z_Block_4 参与了交换，基本块 Y_Block_3 和 C_Block_1 被交换到了函数 X 中、基本块 X_Block_3 和 Z_Block_4 被交换到了函数 Y 中、基本块 B_Block_2 和 A_Block_2 被交换

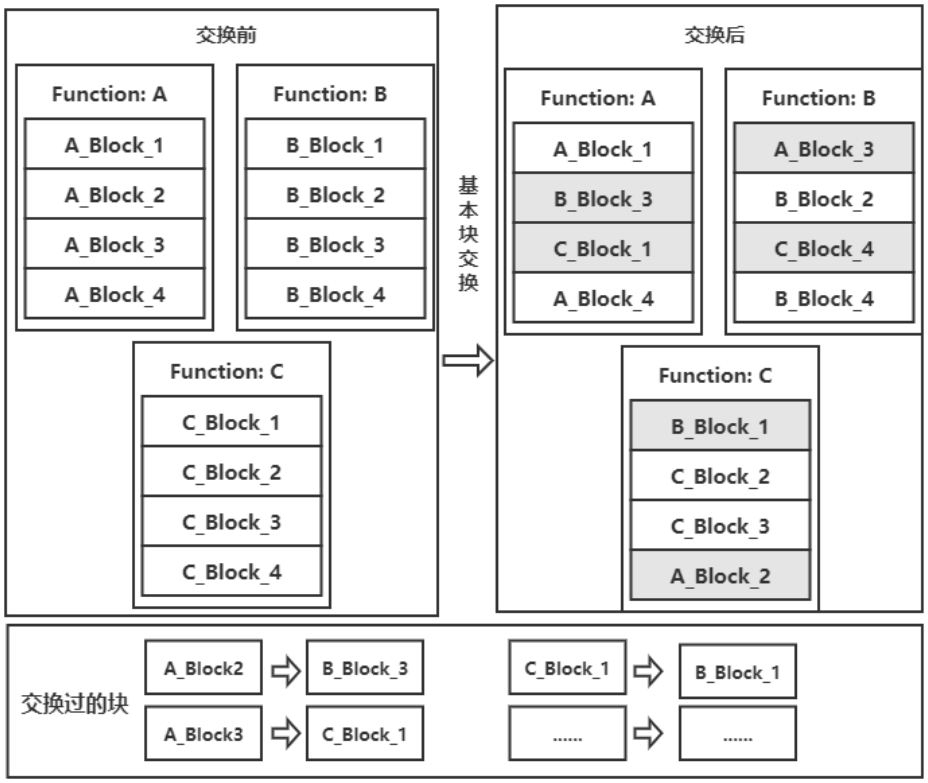

图 4-5　建立索引进行函数间基本块交换过程

Figure 4-5　Establish an index for the basic block exchange process between functions

到了函数 Z 中。根据图4-5右半部分所示，在程序执行时，解释器需要根据索引情况确定被交换过的块。

建立索引进行函数间基本块交换的混淆算法的基本思想是将某个函数的可交换块，通常是 MOV、ADD、SUB 等不影响堆栈和内存的指令块和其他函数的可交换块进行交换，并使用索引记录的方式记录交换情况，在索引方面，也要使用相关的指令使程序在正常运行时保证在正常运行时可以正常还原。在索引的位置也要进行相关的操作，索引的开头位置使用 PUSHAD 指令进行寄存器的保护，在索引的结束位置使用 POPAD 进行寄存器的恢复操作。PUSHAD 指令的功能是将目前所有的寄存器指令压入栈中，POPAD 指令则是 PUSHAD 指令的逆操作。混淆后的函数控制信息被隐藏到了索引中，所以每一条索引都包含了两个函数的交换块信息。

静态分析工具都是以函数为基本单位进行二进制分析，当使用静态分析工具分析经过此种方式加密的程序时，会产生分析函数失败的各种不可预料的错误，大大的加大了静态分析难度，增加了逆向分析者的分析时间和代价。

4.2.2 算法描述

假设加密程序 P 由多个函数 F1、F2...Fn 组成，对程序 P 进行反汇编处理，得到程序 P 中的一系列函数信息。在这些函数中选取部分函数，并将函数进行基本块分割，将函数分为 A、B、C、D、...，根据每个组内的函数数量，将随机选择函数放入多个小组中。A0 中包含 m 个函数 f_0、f_1、…、f_i、…、f_{m-1}，从函数 f_i(i∈[0,m-1]) 中选取 n 个基本块，然后将参与混淆的基本块进行随机交换。

基本块交换的形式化定义：参与的函数为 f_i(i∈[0,m-1]), 每个函数内参与的基本块为 B_{ij}(i∈[0,m-1],j∈[0,n-1]), 然后对这些基本块进行随机处理，最初基本块 B_{ij}(i∈[0,m-1],j∈[0,n-1]), 随机处理改为 $B_{k,l}$(k ∈[0,m-1],l∈[0,n-1])。

根据上述形式化描述可知，基本块的交换相当于对二维数组的随机打乱处理。假设参与的函数有四个，所有函数参数的函数数量也为四个，那么交换之前基本块的位置如表4-1所示。

其中基本块的序号表示此基本块的函数在所处函数参与交换的基本块的索引位置，设定 0 为基本块的起始位置。混淆算法的目的是将上表中 $B_{i,j}$(i∈[0,3],j∈[0,3]) 进行随机打乱，得到一个新的二维数组，可以假设打乱后的结果如表4-2所示。

交换前位置	0	1	2	3
f0	B0	B0,1	B0,2	B0,3
f1	B1,0	B1,1	B1,2	B1,3
f2	B2,0	B2,1	B2,2	B2,3
f3	B3,0	B3,1	B2,3	B3,3

表 4-1　基本块交换前位置

Table 4-1　Position before basic block exchange

表 4-2　基本块交换后位置

Table 4-2　Position after basic block exchange

交换前位置	0	1	2	3
f0	B2,1	B2,1	B1,3	B1,1
f1	B3,0	B0,2	B1,0	B3,2
f2	B1,2	B0,3	B0,0	B3,1
f3	B0,1	B3,3	B2,3	B2,2

这里使用 Knuth-DurstenfeldShuffle 一维数据随机化算法，这里首先将二维数组一维化，将数组 a[m][n] 转化为 a[m*n], 通过上述的方式，将二维数组内的数据转化为一维数组内的随机数据。在最后，再进行一维数组二维化，也就是上述操作的逆操作。

4.3　二进制混淆器设计与实现

根据上一节提出的二进制混淆算法，本节设计并实现二进制混淆器。

4.3.1　总体设计

改加壳机制总体架构如图4-6所示。加壳器主要由四个模块构成，分别为反汇编引擎、控制流程、索引建立和加密混淆引擎、文件重建。

加壳器的输入为 PE 文件，输出为加壳后的 PE 文件。反汇编引擎模块对输入的 PE 文件进行反汇编，得到汇编代码；索引和控制流程把通过反汇编引擎得到的数据进行分析，处理程序的可交换函数、函数之间的调用情况、基本块的数量等信息；加密混淆引擎是整个处理流程的核心，根据索引和控制流程的信息，对程序进行函数混淆加密；文件重建是对混淆后的数据进行重

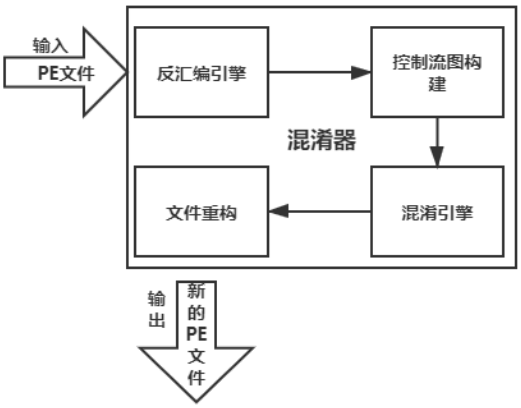

图 4-6　混淆器总体结构

Figure 4-6　The overall structure of the obfuscator

构输出新的 PE 文件。

4.3.2　二进制分析器设计与实现

本节主要的内容是使用反汇编引擎对 PE 文件进行静态分析，为下节混淆器进行加密做准备。反汇编引擎使用 OllyDbg 的反汇编引擎，这个反汇编器是开源的，速度快，易于使用且稳定。由于本文提出的二进制混淆算法只处理 MOV、SUB、ADD 等不涉及堆栈和内存的混淆，属于非常基本混淆机制，没有复杂的内存处理，所有使用本反汇编引擎是最好的选择。

控制流程最主要的任务是建立基本块，如图4-7所示。

基本块的识别流程为：1. 读取所有指令，标记起始指令；2. 遍历指令列表，判断当前指令是否为返回指令或者跳转指令；3. 如果不是跳转或返回指令则执行步骤 2，进行下一条指令的遍历，否则将标记为基本块的结束；4. 将下一条指令标记为另一个基本块的开始，跳转到 2 继续遍历其他所有指令。

4.3.3　混淆器的设计与实现

混淆器的编程实现难度偏高，故本文的实现中所参与交换的函数，不涉及处理内存、堆和栈中的数据，只处理交换函数之间的 MOV、SUB、ADD 等寄存器变换的指令。本文为了方便实现，将算法的某些细节进行简化，算法将不对函数进行详细分组，直接使用随机算法进行满足要求的函数，其中可

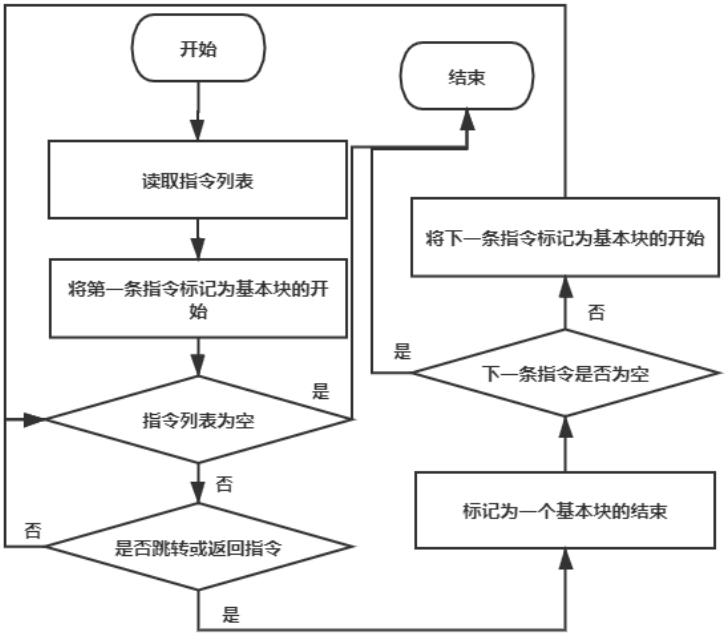

图 4-7　基本块的识别过程

Figure 4-7　Basic block recognition process

能包含大量的 MOV、SUB、ADD 等基础指令并进行混淆。

混淆器的工作流程如图4-8, 具体流程为：1. 加密引擎首先根据二进制分析器提供的构建信息，从满足要求的函数中随机选取 m 个函数；2. 将 m 个中选出的函数进行 n 个基本块之间的交换；3. 建立一个 long 型数组 arr[m*n], 并进行 1-m*n 的初始化。4. 使用 Knuth_Durstenfeld Shuffle 算法对数组 arr 随机化；5. 将函数产生的随机化数据进行如实交换；6. 遍历参与混淆函数列表，如果为空，转到步骤 11；7. 判断函数的占用空间变化情况，如果没有变大，跳转到步骤 9，否则在 PE 文件的最后加一个 add 节，并申请所需空间；9. 计算出各个基本块交换后的新的位置，同时将交换索引放到 PE 文件中的 add 节中；10. 保存处理过的指令，这里只处理了 MOV 指令；11. 将 arr 数组的结果放到新的位置；12. 修正函数的相对地址；13. 保存混淆结果更新 PE 文件头，计算新添加节表的大小，并添加新 add 节表。

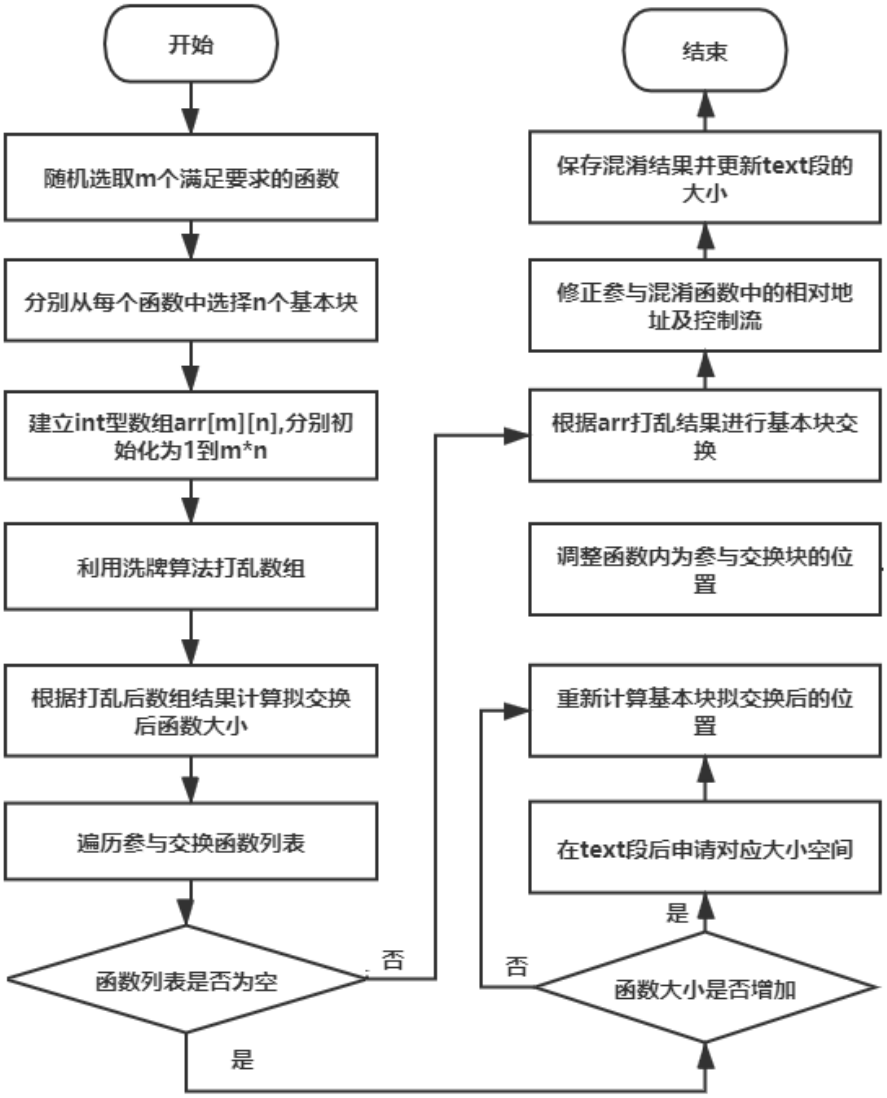

图 4-8　混淆引擎工作原理

Figure 4-8　How the obfuscation engine works

4.3.4 二进制文件重建

PE 文件的重建工作是设计加壳器的重要一环，也是编程的瓶颈所在，PE 文件重建，是根据加密结构重新建立一个 PE 文件，需要用到第二章提到过的大量内容。首先，肯定不能将混淆之后的汇编代码直接放到 PE 文件的各个段中，因为 PE 文件的各个段在经过加密之后占用空间大小会有所增加，PE 文件头部的各个段大小需要修改。另外，各个段起始位置也都需要修改，否则可能会出现两个段重合。段中的跳转地址也需要相应的修复操作。将以上的信息进行修正之后，才能将修改后的汇编代码翻译成机器码写入到 text 段中重新生成一个 PE 文件。

PE 文件重建步骤为：1. 将混淆后的结果翻译成机器码，计算混合后的各个段的占用空间；2. 更新 PE 文件头 PE 可选头表；3. 检查混淆后代码段是否产生重叠，如果不重叠跳转到步骤 5；4. 将重叠的段向后移动；5. 按照 PE 文件的布局依次将混淆后的结果依次写入到新的 PE 文件中，保存文件并退出。

4.4 本章小结

本章对常见的静态分析混淆技术进行了研究，对抗逆向中容易出现的问题进行了总结，分析了已有方法的优缺点，并提出一种建立函数索引进行函数间基本块交换的一种混淆方式，基于随机算法的二进制代码混淆算法，对算法的细节进行了详细的描述。最后根据提出的算法设计并实现了一个 PE 文件加壳器，本文实现了 MOV、SUB、ADD 等指令的交换方式在处理内存和堆栈区域上，在进行混淆之后，合理地还原程序的运行环境，就可以实现其他指令转化方法。

第 5 章　系统实现与测试

本章将实现的 PE 文件虚拟机加壳和 PE 文件二进制混淆器组装成一个 PE 文件保护系统，并分别对本系统的保护壳和加密壳进行了功能验证、加密性能评估。其中本系统虚拟机壳的多样化 Handler 是在已有的虚拟机壳进行修改，并进行了 Handler 多样化的改进，增加了加壳后文件二进制编码的多样性，对逆向攻击者进行了有效的误导，二进制混淆器针对静态调试器进行防御，提出一种对函数间基本块交换的混淆算法，并记录交换块的索引放到文件的最后节表中，但由于交换函数间指令后修复程序的运行环境如内存、堆、栈、寄存器等难度太高，本文将针对 MOV、SUB、ADD 等不影响堆栈的指令进行交换，其余指令在保证能够恢复运行时的环境也可以实现，本章将会对以上的动态和静态保护措施进行实现和测试。

5.1　总体设计方案

5.1.1　编程语言的选择

通常情况下 PE 文件二进制保护软件使用 C 语言或者汇编来编写，市面流行的加密壳的外壳部分通常使用汇编代码，使用汇编语言编写，可以实现更加复杂的算法并且对 PE 文件的二进制混淆算法更容易实现，但是汇编语言难以掌握，并且通常情况下不具有可移植性，所以本保护系统将会使用 C++ 和汇编语言编译器为 X86nasm 混合编程的方式，在需要对 PE 文件加壳的复杂位置使用汇编语言，程序的整体架构采用 C++, 这极大地提高了开发效率和降低了开发难度。

5.1.2　软件的系统构架

本系统采用模块化设计有利于系统的开发和维护，采用模块化结构有利于对软件功能进行细分，模块出现问题更利于修改。本系统有以下几个模块构成：外壳添加模块、反调试模块、特殊数据处理模块、区块压缩模块、输入表构建模块、反静态分析模块、虚拟机模块、PE 文件读取模块组成。其中反调试模块、区块压缩模块、虚拟机模块采用汇编语言编写，采用 X86nasm

编译器。

5.2 功能验证

功能测试部分将对系统中的虚拟机加密和二进制混淆器的功能分别进行测试。在 32 位 Windows 10 系统下磁盘空间为 120G、内存为 2G 的环境下进行功能测试，选择 32 位 PE 文件 notepad.exe 作为测试文件。在未进行虚拟机加密或未经二进制混淆之前，程序 Aseprite.exe 的运行情况如图5-1所示。

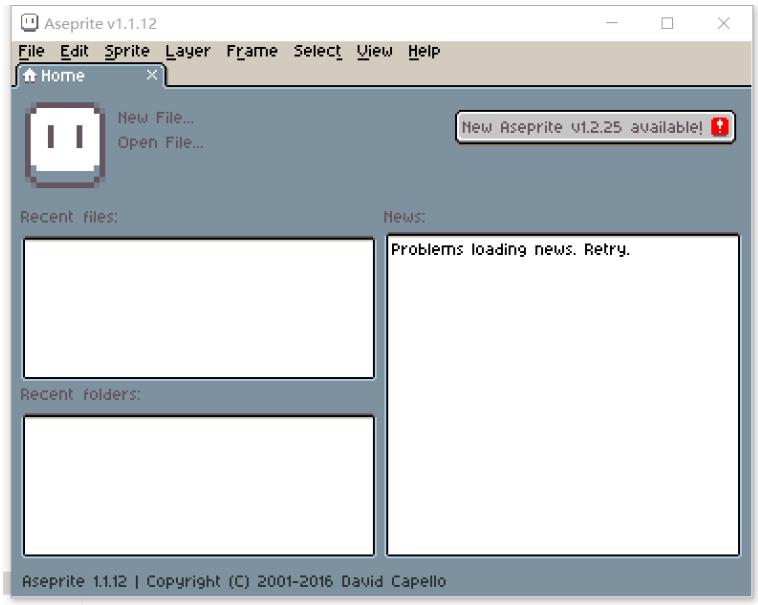

图 5-1　程序 Aseprite 运行时

Figure 5-1　When the program Aseprite is running

5.2.1 虚拟机加壳功能验证

加密壳由一个可执行文件 aseprite.exe 和一个加密脚本 protect.bat 组成。通过对脚本 protect.bat 调用虚拟机壳对 aseprite.exe 进行加壳的运行时如图5-1, 加密密钥为 0x3275923A。

使用 PEiD 对 aseprite 进行加壳分析，加壳前节区如图5-2，加壳后节区如图5-3。

通过对图5-2和图5-3中的红色框出数据分析，在加壳后，PE 文件中的节

名称	V. 偏移	V. 大小	R. 偏移	R. 大小	标志
.text	00001000	00807EF3	00000400	00808000	60000020
.rdata	00809000	00122978	00808400	00122A00	40000040
.data	0092C000	00024638	0092AE00	0001CA00	C0000040
.tls	00951000	0000012D	00947800	00000200	C0000040
.gfids	00952000	0000150C	00947A00	00001600	40000040
_RDATA	00954000	00000120	00949000	00000200	40000040
.rsrc	00955000	000048B0	00949200	00004A00	40000040
.reloc	0095A000	0004824C	0094DC00	00048400	42000040

图 5-2　Aseprite 加壳前节表

Figure 5-2　Aseprite Packing Front Section Table

名称	V. 偏移	V. 大小	R. 偏移	R. 大小	标志
.rdata	00809000	00123000	00316C00	0005E200	C0000040
.data	0092C000	00025000	00374E00	00007600	C0000040
.tls	00951000	00001000	0037C400	00000200	C0000040
.gfids	00952000	00002000	0037C600	00001600	C0000040
_RDATA	00954000	00001000	0037DC00	00000200	C0000040
.rsrc	00955000	00005000	0037DE00	00004A00	C0000040
.reloc	0095A000	00049000	00382800	00012400	C2000040
.pediy	009A3000	00001000	00394C00	00000400	E0000040

图 5-3　Aseprite 加壳后节表

Figure 5-3　Aseprite packed back section table

表数据已经被修改，并且添加了加壳用到的新节.pediy，此时大部分节表数据已经被压缩，成功添加了.pediy 新节。aseprite 程序加壳后的运行时情况成功打开，与加壳功能一致，表明加壳成功。运行结果一致，加壳后的程序能够正常运行，表明加密壳的功能正常，能够对 PE 文件的有效加壳。

5.2.2　二进制混淆器功能验证

二进制混淆器只有一个可执行文件 shell.exe，对文件 Calculator.exe 进行混淆，Calculator 为 Windows10 中自带的计算器，下面通过查看 Calculator.exe 程序在混淆前后的函数间的调用变化在说明保护的有效性。使用二进制混淆器对 Calculator 加密后的运行情况如图5-4所示。

计算器的功能正常运行，为了更深入的测试二进制混淆效果，下面使用 IDA PRO 对该程序进行反汇编，通过查看 Calculator 程序在混淆前后的函数控制流来观察加密的有效性。在加密前后 Calculator 的函数 sub_403339 的控制流图分别如图5-5和图5-5所示，图中序号表示函数内部的基本块序号。

由图5-5和图5-6可知，加密前后函数 sub_403339 分别包含了 6 个和 7 个基本块。加密后，原函数的 1 和 3 基本块加密前后基本没有发生变化，分别

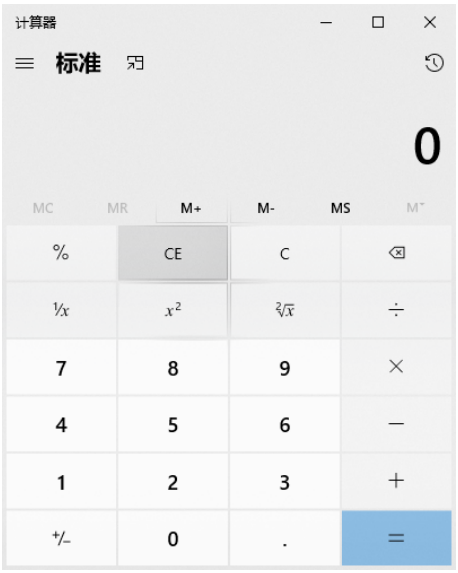

图 5-4　Calculator.exe 使用二进制混淆后的运行情况

Figure 5-4　Operation of Calculator.exe after using binary obfuscation

对应加密后的 1 和 4 基本块；原函数中 5 和 6 基本块内容也没有变化，加密后被合成了基本块 7；原函数中的 2 和 4 基本块被交换到了其他函数中；加密后函数中多了两个其他函数的 3 和 6 基本块，且基本块最后是一条 jmp 指令，通过该 jmp 指令将控制流转移到下一个基本块；原函数中对于基本块 2 和 4 的控制流转移通过基本块 2 和 5 中 push+retn 型指令实现。

加密后，参与加密函数内部的部分基本块被交换到了其他函数中，被交换的基本块通常有其他的函数的基本块，加大了使用 IDA 等静态分析器的分析难度，整个程序的控制流信息被打乱。

综上可知，二进制混淆器对程序进行加密后，原有的控制流信息得到了隐藏，达到了对 PE 文件加密的目的，二进制混淆器功能正常。

5.3　性能评估

性能评估在 32 位 Windows10 系统下，内存为 2G，磁盘空间为 60G 下进行，评估内容包括静态混淆技术加密保护和虚拟机保护有效性。使用的测试样本为 Windwos10 下自带程序和目前流行软件。

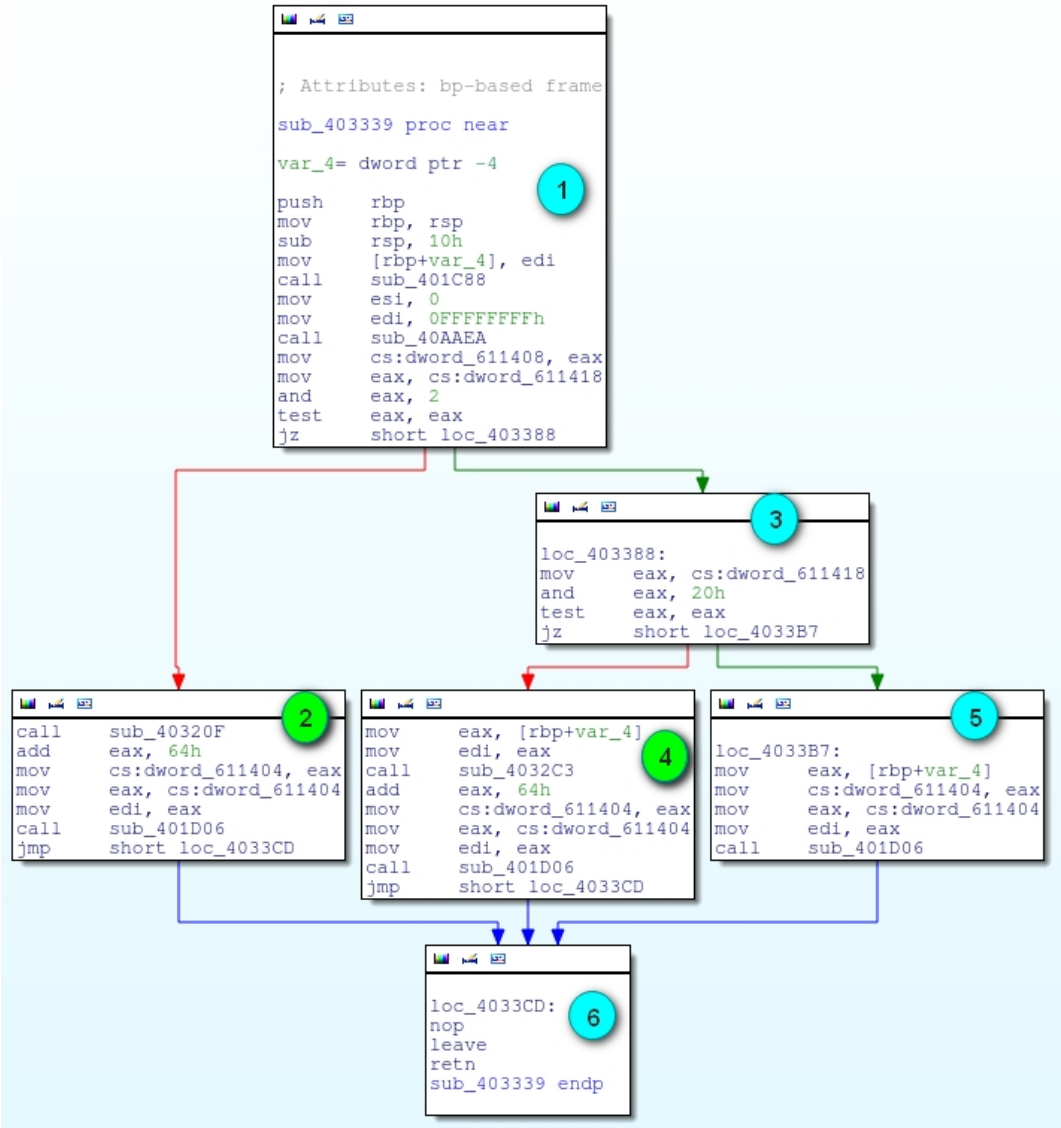

图 5-5　加密前 sub_403339 函数的控制流图 IDA 截图

Figure 5-5　Control flow diagram of sub_403339 function before encryption

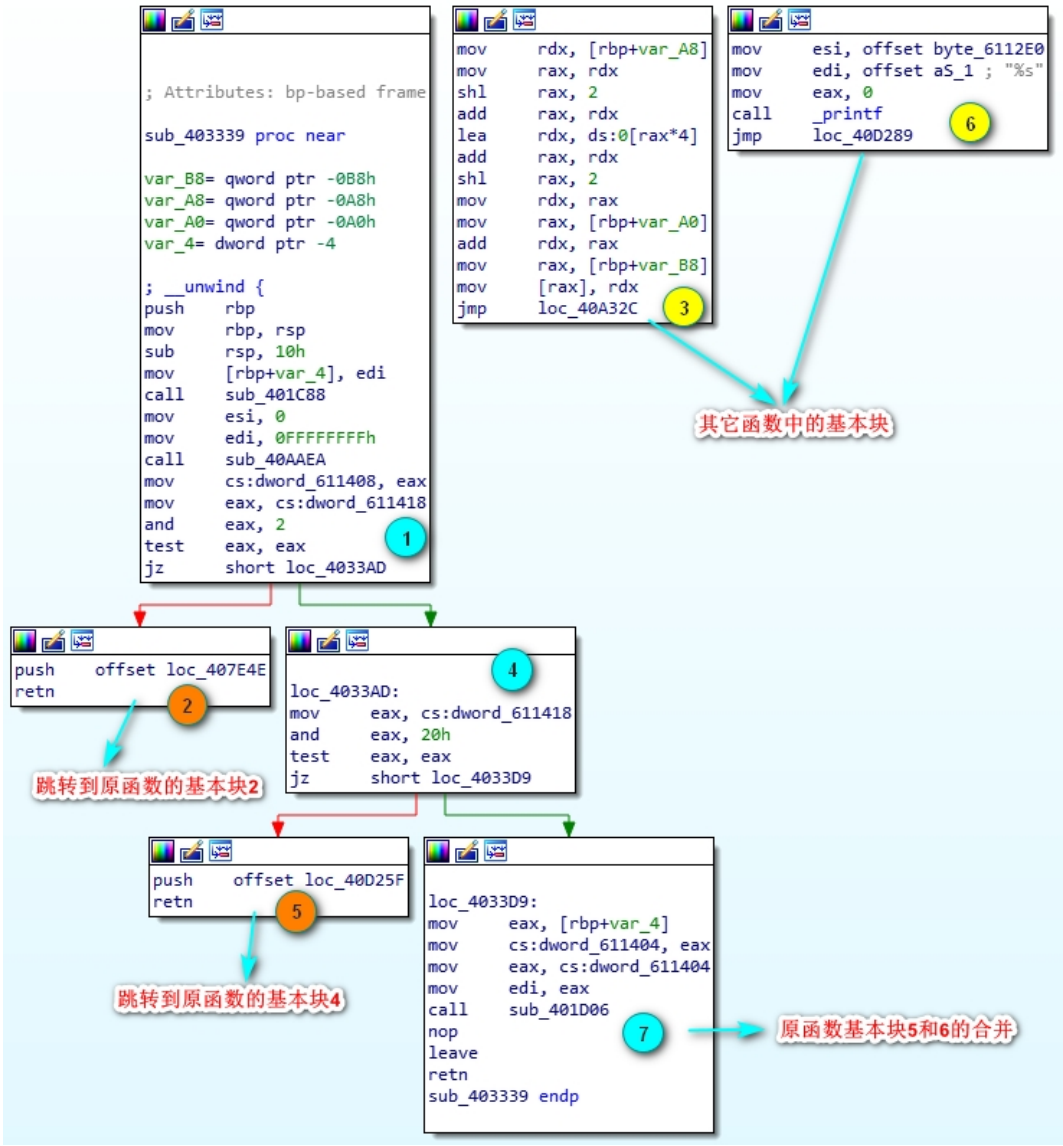

图 5-6　加密后 sub_403339 函数的控制流图 IDA 截图

Figure 5-6　Control flow diagram of sub_403339 function after encryption

5.3.1 虚拟机加壳性能评估

本节将对虚拟机加壳与流行加壳软件进行性能对比，其中包括压缩壳和加密壳，对比的变量为不同的测试程序，对比的输出数据为程序加壳后的大小、开启用时、和执行一次任务所用时间对比，通过计算得出压缩率、启动时间延迟百分比，额外运行时间开销百分比。

使用 UPX 软件这款压缩壳作为对照组。这里说明一下，使用 UPX 压缩壳软件作为对照组，是因为软件虚拟机加壳下，使用虚拟机加壳，相当于软件内置一个虚拟机解释器，在执行过程中采用边执行二进制代码边进行解释，程序的运行时间是最重要的性能指标，所以，在能保证程序正常运行的情况下，程序的运行时间是最重要的参考指标。至于加密壳，其运行时间远远小于虚拟机加壳，在这一方便，虚拟机加壳性能与加密壳性能差距太大，所以不能作为对照组。尽管 UPX 是一个压缩壳，但是现在未公布的 Windows 压缩壳大部分是参考 UPX 进行修改，所以使用 UPX 作为加壳对照组具有一定意义。

表 5-1　测试程序加壳前后大小变化

Table 5-1　The size change of the test program before and after packing

测试程序	原始程序	UPX 加壳后		加密壳加壳后	
	大小 (KB)	大小 (KB)	压缩率 (%)	大小 (KB)	额外空间开销 (%)
maps	21	8	40.9	40	93.0
cacls	33	8	26.9	45	38.7
mkdir	72	32	45.7	110	54.0
bscmake	95	36	37.9	116	23.0
ls	146	35	24.5	156	7.6
aseprite	3668	656	17.9	3851	0.5

从表5-1可以看出，在测试程序大小小于 100KB 时，加壳导致的额外空间开销明显降低了，程序 aseprite 加壳后产生的额外空间开销甚至只有 0.5%。

从表5-2可以看出，使用本文加密壳加壳后的程序额外运行时间比较少，和 UPX 加壳软件加壳后产生的额外运行时间开销接近。时间开销相对不大的一个原因是加密壳只进行了函数间基本块的交换，使用随机算法，没有对程序汇编代码进行压缩，时间复杂度较低，如果需要对加密性能进行强化，可

表 5-2　测试程序加壳前后时间变化

Table 5-2　Time changes before and after the test program is packed

测试程序	原始程序	UPX 加壳后		加密壳加壳后	
	平均运行时间 (ms)	平均运行时间 (ms)	额外时间开销 (%)	平均运行时间 (ms）	额外时间开销 (%)
maps	5	8	66.7%	6	33.3
cacls	120	127	6.2	131	9.8
mkdir	44	47	8.1	51	16.3
bscmake	253	272	7.6	275	9.0
ls	30	31	4.9	32	8.5
aseprite	540	561	3.9	564	4.5

以使用更复杂或使用自定义的加密算法。所以，加密壳加壳后程序产生的额外时间和空间开销比较小，可以较好地满足实际应用。

5.3.2　二进制混淆器功能评估

本节通过二进制混淆器的指令执行效率进行评估来判断本文提出的混淆算法的有效性，评估的指标包括程序混淆后的混淆强度、运行时间开销、稳定性、隐蔽性。

为对比系统中混淆器的性能情况，使用市面上流行的二进制代码混淆工具 Virbox Protector 对程序混淆后的结果作为对照。

5.3.2.1　混淆强度

混淆强度的评估使用赵玉洁等提出的控制流循环复杂度和单条指令执行效率来度量。其中指令执行效率的公式为:$IE = Id/Is$，其中 Is、Id 分别表示被测试程序所包含的总指令数量和被测试程序中被运行过的指令数量；控制流循环复杂度的公式为:$V(G) = e - n + 2$，其中 e 表示边的总数，n 表示节点（基本块）的总数, 测试程序经过混淆器混淆前后的指令执行率如表5-3所示。

程序在混淆之后，静态指令和动态指令都有所增加，这是因为采用了建立索引进行函数间基本块交换的混淆算法，在原有程序的基础上，增加了一些指令数据来实现函数间的跳转，虽然跳转的只是一些 MOV、ADD、SUB 等不影响寄存器和堆栈的指令，但是其指令数量还是有所提高，由原始程序

表 5-3　混淆器混淆前后测试程序的指令执行率

Table 5-3　The instruction execution rate of the test program before and after obfuscation

测试程序	原始程序			混淆后程序		
	静态指令数 (条)	动态指令数 (条)	执行率 (%)	静态指令数 (条)	动态指令数 (条)	执行率 (%)
maps	4755	1059	22.27	4899	1210	24.69
cacls	3306	761	23.01	3514	810	23.05
mkdir	21230	4097	19.29	22019	4600	20.89
bscmake	8185	1975	24.12	9100	2216	24.35
ls	12523	2439	19.47	12699	3010	23.70
aseprite	193966	32374	16.68	196109	41409	21.11

和混淆程序的指令执行率可知，程序中每条执行被执行概率增加了，一定程序上反映出了程序的复杂性增加了。使用 Virbox Protector 混淆后程序的执行执行效率如表5-4。

表 5-4　Virbox Protector 混淆后程序的指令执行率

Table 5-4　The instruction execution rate of the program after Virbox Protector confusion

测试程序	Ftotal	Fobf	静态指令数	动态指令数	指令执行率
maps	126	14	4518	1198	26.51
cacls	39	7	3214	943	29.34
mkdir	234	15	14981	4709	31.43
bscmake	157	15	4910	1598	32.54
ls	164	14	46910	6102	13.00
aseprite	6019	99	259838	34631	13.32

其中 FOBF、FTOTAL 分别表示参与混淆的函数个数和程序的函数总数。从表5-4 可以看出几点：1. 经过 Virbox Protector 混淆之后，不同程序的静态执行数量出现了增加和减少两种情况，这是 Virbox Protector 的混淆机制导致的，使用 IDA Pro 查看混淆后的程序的二进制汇编代码可知，在反汇编之后存在大量的数据，在静态反汇编时是以 db 指令存在的，被反汇编当做数据对待，但在执行时，其实在被解密后是以普通汇编指令执行的；2. 当程序增加较多的静态指令数量时，程序的指令执行率未混淆之前甚至高于混淆之后，

当增加较少的静态指令数量时，程序的执行效率比混淆前要高很多。所测试的程序指令执行率有减少也有增加，这是因为 Virbox Protector 内部使用了不同的混淆机制, 本课题只使用一种，比如某些垃圾指令不被执行会降低程序的指令执行效率，而将一条会被执行的指令拆分成多条执行或添加被执行的垃圾执行会提高程序执行率。

测试程序在混淆前后控制流循环复杂度如表5-5所示。

表 5-5　混淆前后测试程序控制流循环复杂度

Table 5-5　Test program control flow loop complexity before and after confusion

测试程序	原始复杂度	Virbox Protector 混淆后		混淆器混淆后	
		复杂度	增长率 (%)	复杂度	增长率 (%)
maps	267	184	-31.1	279	4.49
cacls	318	123	-61.3	329	3.45
mkdir	778	1690	117.2	920	18.25
bscmake	378	298	-21.2	451	19.31
ls	934	3890	316.4	1090	16.70
aseprite	34509	40109	16.2	37909	9.85

使用 Virbox Protector 混淆后的程序的的控制流循环复杂度没有固定的规则，控制流循环复杂度有的数据特别高，有的甚至出现负数。而经过本课题的混淆器混淆之后，基本控制流的复杂度和增长率可以呈现一定的规律，程序的增长率基本维持在 7%-11%，这说明在不考虑垃圾指令优化时，使用单种混淆方式通常情况下复杂度是增加的。

值得注意的是，Virbox Protector 处理过后的程序的控制流循环复杂度降低了，但是并不代表程序的控制减少了，由于 Virbox Protector 包含多种混淆机制，在执行时有大量的数据需要解密后执行，所以程序的动态控制流会更复杂。

根据上述测试结果，可以得到二进制混淆器加密前后指令复杂性和控制流循环复杂度的变化以及与 Virbox Protector 软件的混淆结果对比，可以看到，经过二进制混淆器混淆后的程序的指令复杂度得到一定提高，表明混淆算法的混淆强度指标合格。同时也反映出了通过指令的执行率和控制流循环夫再度来衡量代码混淆程序的局限性。

5.3.2.2 运行时间开销的对比

二进制混淆器和 Virbox Protector 对测试程序混淆后，程序的运行时间开销的对比如表5-6

表 5-6 混淆后测试程序的额外开销

Table 5-6 Additional overhead of test program after obfuscation

测试程序	原始程序		混淆器混淆后		额外开销	
	大小 (KB)	平均运行时间 (ms)	大小 (KB)	平均运行时间 (ms)	空间 (%)	时间 (%)
maps	40	3	43	5	7.5	66.7
cacls	78	157	85	170	8.9	8.2
mkdir	124	223	130	131	5.6	5.6
bscmake	68	28	72	32	5.8	14.2
ls	120	91	128	101	6.7	10.9
aseprite	5093	450	5609	480	10.1	6.7

使用二进制混淆器后，程序的额外占用空间随着程序的占用空间增大而增大，空间开销在程序小于 5M 时空间开销低于 10%；程序的额外运行时间开销随着运行时间增加比例大约在 8% 左右。由以上可得，二进制混淆器产生的额外开销所占比率并不高。由表5-7可知，

表 5-7 Virbox Protector 混淆后测试程序的额外开销

Table 5-7 The extra cost of the test program after Virbox Protector obfuscation

测试程序	大小 (KB)	空间开销 (倍)	时间 (ms)	时间开销 (倍)
maps	484	11.0	10.3	3.44
cacls	1723.8	22.1	282.6	1.8
mkdir	1277.2	10.3	691.3	3.10
bscmake	632.4	9.3	81.4	2.91
ls	588	4.9	171.9	1.89
aseprite	6111.6	1.2	49.5	1.10

使用 Virbox Protector 混淆后的程序的占用空间和运行时间开销都很高，这样会程序在实际使用起来的体验不如本课题的二进制混淆器。

5.3.2.3 隐蔽性

本课题隐蔽性通过计算经过二进制混淆前后的文件相似度来判断。测试样本的混淆前后文件相似度使用 radare 中的 radiff2 工具来计算，经过混淆前后的相似度如表5-5所示。使用 Virbox Protector 加密过的程序的变化非常明显，可以很明确看出程序是经过混淆处理的，这样的程序隐蔽性较低。

表 5-8　混淆前后程序的相似度

Table 5-8　Program similarity before and after confusion

测试程序	maps	cacls	mkdir	bscmake	ls	aseprite
相似度 (%)	91	93	95	92	93	92
平均相似度 (%)	93					

由表5-8可知，混淆前后，程序的相似度比较高，都在 80% 以上，平均相似度为 89%。从上面可以得出，使用本课题的虚拟机的混淆程序不易被发现该程序是经过混淆的，这样降低了程序被逆向的风险，所以具有一定的抗逆向意义。

二进制混淆器与 Virbox Protector 对比，本文设计的二进制混淆器的混淆强度较低，同时在被逆向的情况下，本课题的保护壳更容易被成功逆向破解，这是因为本混淆器只实现了普通汇编指令如 MOV、SUB、ADD 等不影响寄存器和堆栈空间的指令的交换和混淆，如果将所有汇编指令都实现交换和混淆，产生的二进制混淆器的混淆强度将会高出一个级别。但是本文的二进制混淆器的运行时间、占用磁盘空间和隐蔽性能比较良好。

综上可知, 加密后的程序的运行时间、磁盘占用空间和隐蔽性都要优于 Virbox Protector，具有较好的实用性。这表明本文二进制混淆器的抵抗静态分析的能力较强，但是抵御动态分析方面存在不足。由上文可知，本混淆器可以与其他加壳工具兼容，实现混用，本文的二进制混淆器与其他保护壳共同使用时，可以产生较高的保护力。由此表明本文所提出的二进制混淆算法可以提高软件的抗逆向性能。

5.4 本章小结

本章首先对系统的实现和所选择的测试环境进行介绍，然后分别对虚拟机加壳器和二进制混淆器的功能进行了验证和性能进行评估，由实验结果可知，加壳器和二进制混淆器的功能正常、性能评估结果表明使用上述保护方法对二进制可执行程序进行保护具有一定实用性，总体实验结果表明所提出的虚拟机加壳方法和二进制混淆器组成的保护系统能够有效增强 Windows 下可执行程序的抗逆向能力。

结论

恶意的逆向分析技术严重威胁着软件的版权安全，本文针对 Windows 平台下 x86 架构下的可执行文件进行了软件保护技术方面的探索，提出了一种基于虚拟机加壳的多样化 Handler 动态保护方法和一种建立索引的方式对函数间基本块进行交换的静态保护算法。本文从代码层面上对算法进行了阐述，描述了软件保护机制的实现方法，给出了虚拟机加壳技术和静态二进制混淆技术的核心汇编源码，并对设计的核心数据结构进行了说明，本论文的工作总结如下：

首先，PE 文件加壳技术的研究。首先对 PE 文件的加载过程和 PE 文件的文件结构进行分析，在虚拟机加壳的基础上，提出了多样化 Handler 虚拟机加壳技术，对同一个 Handler 进行多样化的指令实现，使得同一个功能模块有不同的二进制指令实现，有效了增加逆向分析者的分析成本。

其次，PE 二进制混淆技术的研究。首先对二进制混淆技术进行了分析，总结常见二进制混淆方法的不足，提出了建立索引进行函数间基本块交换的混淆算法，提出了对 PE 文件进行基本块交换后的重构方法。

最后，根据提出的虚拟机加壳技术和二进制混淆算法设计并实现了一个 PE 文件保护系统，系统包括 PE 文件虚拟机加密壳和 PE 文件二进制混淆器。对该系统的设计与实现进行了详细阐述，并对 PE 文件虚拟机加密壳和 PE 文件二进制混淆器进行了功能的验证和性能评估。实验结果表明提出的多样化 Handler 虚拟机加壳方式和二进制代码混淆算法均有效，有效增强了 Windows 的 PE 文件的抗逆向分析能力。

参考文献

[1] 芦天亮, 李国友, 吴警, 等. 计算机病毒中的密码算法应用及防御方法综述[J]. 科技管理研究, 2020, 040(002):207-215.

[2] 王欢. 构建高效, 可靠, 安全的移动应用安全体系[J]. 通讯世界, 2020(7).

[3] 徐君锋, 吴世忠, 张利. Android 软件安全攻防对抗技术及发展[J]. 北京理工大学学报, 2017, 037(002):163-167.

[4] 简容, 黎桐辛, 周渊, 等. 一种多层次的自动化通用 Android 脱壳系统及其应用[J]. 北京理工大学学报, 2019(7):725-731.

[5] 钟林辉, 扶丽娟, 叶海涛, 等. 软件演化历史的逆向工程生成方法研究[J]. 计算机科学, 2020(S2).

[6] 夏学云. 软件逆向工程技术分析[J]. 科学技术创新, 2019, 000(027):P.74-75.

[7] Rauf M A A A, Asraf S M H, Idrus S Z S. Malware behaviour analysis and classification via windows dll and system call[J]. Journal of Physics Conference Series, 2020, 1529: 022097.

[8] Ntantogian C, Poulios G, Karopoulos G, et al. Transforming malicious code to rop gadgets for antivirus evasion[J]. IET Information Security, 2019, 13(6):570-578.

[9] Jin H, Park M C, Lee D H. Analysis of detection ability impact of clang static analysis tool by source code obfuscation technique[J]. Journal of the Korea Institute of Information Security and Cryptology, 2018, 28.

[10] Ceccato M, Penta M D, Falcarin P, et al. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques[J]. Empirical Software Engineering, 2014, 19(4):1040-1074.

[11] Zhang F, Li L, Liu C, et al. Flow chart generation-based source code similarity detection using process mining[J]. entific Programming, 2020, 2020(11):1-15.

[12] Nellaivadivelu G, Troia F D, Stamp M. Black box analysis of android malware detectors [J]. Array, 6.

[13] Liu X, Lin Y, Li H, et al. A novel method for malware detection on ml-based visualization technique[J]. Computers and Security, 2020, 89(Feb.):101682.1-101682.12.

[14] 秦飞. 逆向工程和 Geomagic 软件在医学中的应用[J]. 黑龙江科技信息, 2018, 000 (016):71-72.

[15] 马宁. 应用虚拟机杀毒引擎解决病毒加壳难题[J]. 中国金融电脑, 2007(1):23-23.

[16] 曾强. Linux 二进制应用软件漏洞自动挖掘研究与实现[J]. 电力信息与通信技术, 2020(9).

[17] Badhani S, Muttoo S K. Analyzing android code graphs against code obfuscation and app hiding techniques[J]. Journal of Applied Security Research, 2019:1-22.

[18] Nunez-Musa Y, Rodriguez-Veliz M, Sepulveda-Lima R. Call graph obfuscation and diversification: an approach[J]. IET Information Security, 2020, 14(7).

[19] Roos D R. Access code obfuscation using speech input[J]. 2017.

[20] Wang Y, Wu H, Zhang H, et al. Orlis: obfuscation-resilient library detection for android [C]//the 5th International Conference. [S.l.: s.n.], 2018.

[21] 段钢. 加密与解密[J]. 程序员, 2003, 000(009):53-53.

[22] Xuan H, Lu C, Zhou Y, et al. Saturable and reverse saturable absorption in molybdenum disulfide dispersion and film by defect engineering[J]. Photonics Research, 2020.

[23] A Y Z, A Z T, A G Y, et al. Compile-time code virtualization for android applications[J]. Computers and Security, 94.

[24] Lee G, Yu J, Kim I, et al. Implementation of software source code obfuscation tool for weapon system anti-tampering[J]. Journal of KIISE, 2019, 46(5):448-456.

[25] Fels U, Gevaert K, Damme P V. Bacterial genetic engineering by means of recombineering for reverse genetics[J]. Frontiers in Microbiology, 2020, 11.

[26] Gong C, Zhang J, Yang Y, et al. Detecting fingerprints of audio steganography software [J]. Forensic ence International Reports, 2020, 2:100075.

[27] Shim K, Goo Y, Lee M, et al. Clustering method in protocol reverse engineering for industrial protocols[J]. International Journal of Network Management, 2020(82).

[28] Marin L. White box implementations using non-commutative cryptography[J]. Sensors, 2019, 19(5).

[29] Lim K, Kim B, Cho S J. Implementing a technique for detecting obfuscated open-source android apps[J]. KIISE Transactions on Computing Practices, 2019, 25(2):106-112.

[30] Roberto B, Roberto G, Roberta G. Code obfuscation against abstraction refinement attacks [J]. Formal Aspects of Computing, 2018.

[31] Jianga W, Wanga H, Wua K. Method for detecting javascript code obfuscation based on convolutional neural network[J]. International Journal of Performability Engineering, 2018, 14(12).

[32] Tang Z, Xue M, Meng G, et al. Securing android applications via edge assistant third-party library detection[J]. Computers and Security, 2018, 80.

[33] Sevinc A. Model parameters of electric motors for desired operating conditions[J]. Advances in Electrical and Computer Engineering, 2019, 19(2):29-36.

[34] Proy J, Heydemann K, Berzati A, et al. Compiler-assisted loop hardening against fault attacks[J]. Acm Transactions on Architecture and Code Optimization, 2017, 14(4):1-25.

[35] Gautam P, Saini H. A novel software protection approach for code obfuscation to enhance software security[J]. International Journal of Mobile Computing and Multimedia Communications, 2017, 8(1):34-47.

[36] Huda S, Islam R, Abawajy J, et al. A hybrid-multi filter-wrapper framework to identify run-time behaviour for fast malware detection[J]. Future Generation Computer Systems, 2018:S0167739X17325049.

[37] Zhanyong T, Meng L, Guixin Y, et al. Vmguards: A novel virtual machine based code protection system with vm security as the first class design concern[J]. Applied ences, 2018, 8(5):771.

[38] Fujieda N, Tanaka T, Ichikawa S. Design and implementation of instruction indirection for embedded software obfuscation[J]. Microprocessors and Microsystems, 2016, 45(aug.): 115-128.

[39] 李路鹿, 张峰, 李国繁. 代码混淆技术研究综述[J]. 软件, 2020(2).

[40] 向飞, 巩道福, 刘粉林. 一种基于 ROP 技术的代码混淆方法[J]. 计算机应用与软件, 2019, 036(009):293-301.

[41] 吕苗苗. 基于 JAVA 的安卓应用代码混淆技术研究[J]. 山东农业大学学报 (自然科学版), 2019(4).

[42] Kuang K, Tang Z, Gong X, et al. Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling[J]. Computers and Security, 2018, 74(MAY):202-220.

[43] Karetnikov V V, Shahnov S F, Ageeva A A. Construction method of telecommunication system for corrective information distribution[J]. Iop Conference, 2018, 171(1):012010.

[44] 张晓寒, 张源, 池信坚, 等. 基于指令虚拟化的安卓本地代码加固方法[J]. 电子与信息学报, 2020(9).

[45] 何永瑾, 郭肖旺, 赵德政. 基于注册码的软件授权保护系统的设计与实现[J]. 微型机与应用, 2020, 039(005):42-45,50.

[46] Basile C, Canavese D, Regano L, et al. A meta-model for software protections and reverse engineering attacks[J]. Journal of Systems and Software, 2018.

[47] 马雪婷, 李宏图. 一种共享软件保护机制的完整实现[J]. 科技创新与应用, 2020, 000(010):25-27,30.

[48] Bill V, Fayard A L. Building a makerspace to nurture the innovation culture at the nyu tandon school of engineering[C]//International Symposium on Academic MakerSpaces. [S.l.: s.n.], 2016.

[49] Zejda D, Zelenka J. The concept of comprehensive tracking software to support sustainable tourism in protected areas[J]. Sustainability, 2019, 11.

[50] 张泉, 舒辉, 李婧睿. Win32 平台下基于 LLVM 的代码混淆技术研究[J]. 信息工程大学学报, 2018, 19(04):498-502.

[51] 乐德广, 赵杰, 龚声蓉. 一种抵御逆向工程的安卓应用混淆技术研究[J]. 小型微型计算机系统, 2018, 39(07):138-143.

[52] 曹宏盛, 焦健, 李登辉. 一种用于 Android 应用的反控制混淆系统[J]. 计算机应用研究, 2019, 036(005):1544-1548.

[53] 胡浔惠, 葛王飞, 段文强, 等. 一种应用随机森林的代码混淆路径分支技术[J]. 信息技术, 2019(8).

攻读硕士学位期间发表的学术论文与研究成果

[1] 哈尔滨理工大学. Windows 下可执行文件压缩加密程序软件 V1.0: 中国，(软件著作权登记号为 2020SR1581640) .

致谢

本人软件保护工程方面做了些微小的工作，仅仅围绕了抗逆向技术做了边缘性的优化工作，并没有做其技术的核心优化。不过从没有太多参考的当下从 0 到 1 建立了一个完整系统，且结果也恰好说明这个系统确实有效，也是一件颇为有趣的事情。当然这里少不了我的家人、老师和朋友们的功劳。

首先致谢我的父母，支持我完成本科教育后继续学习，使得我满足自己的好奇和求知欲。其次感谢我的导师李老师，接触的四年里共同奋斗过很多事情，有欢乐也有忧愁，所有教导我都铭记于心，并且在毕业设计之际允许我完成自己的想法。接着感谢我的朋友们，在我的毕业设计之际提供了无数关怀与帮助，使我完成之。最后感谢在哈尔滨这个城市直接或间接教导我的所有人，谢谢。

最后，再次一并感谢。