毕 业 设 计 (论 文)

题 目 防终止的基于 Windows 透明加密过滤系统研究与实现

学院(系)： 计算机与电子信息学院

专　　业：

班　　级：

学　　号：

学生姓名：

指导教师：

二零一五年五月二十五日

防终止的基于 Windows 透明加密过滤系统研究与实现

摘　要

办公自动化飞速发展，极大的提高了我们工作的效率，在这样的工作模式下，我们每天都要生成、存储、传输大量的电子文档，而其中不乏企业机密、个人隐私，若不采取恰当的保护措施，极易泄露。如何保证电子文档的安全，成了备受关注的热点问题。然而目前较为成熟的防火墙技术和入侵检测技术只能阻断来自网络的攻击，却无法解决终端设备的文档安全问题。为了防止公司机密文档泄露，很多公司通过禁用 USB 接口、禁止连接网络等措施来防止员工泄露机密文档。虽然，这样的方式在一定程度上保证了存储在终端设备中的文档安全，却给人们的工作带来了极大的不便。在这样的背景下一种针对系统级文档安全问题的技术应运而生——透明加密技术。该技术有两种实施途径，即钩子技术和过滤驱动技术。其中钩子技术通过挂钩相关的 API 来实现拦截保护，工作于用户层，极易受到病毒、木马等的攻击。相比之下过滤驱动技术在操作系统的内核层实现过滤驱动功能来拦截发往目标设备的请求，进而达到保护文档的目的，由于过滤驱动工作于内核层，因此安全性高，不会轻易受到攻击，使得过滤驱动成为了目前实现透明加密的主流手段。

本文在深入研究 Windows 内核与驱动开发技术的基础上，采用微软提供的 Minifilter 框架实现防终止的基于 Windows 透明加密过滤系统，为文档安全提供了较为完备的解决方案。系统主要由三个部分组成，即服务器端、客户端用户层控制程序和客户端内核层透明加密过滤驱动。服务器主要进行策略的下发、监控客户端程序、密钥的分发与管理、信息审计等工作；客户端用户层控制程序作为服务器与客户端内核的桥梁；客户端内核模块实现了透明加密过滤驱动，对受保护文档进行自动加解密。服务器通过 Socket 与客户端进行通信，采用心跳机制监控客户端的在线情况。用户层与内核层通过 Minifilter 提供的端口通信机制实现高效的双向通信。在完成基本功能的基础上，还利用 HOOK API 技术实现了剪切板监控与进程防终止功能，进一步提高了系统的安全性。

关键词　Windows 内核；文件系统；过滤驱动；Minifilter；透明加解密

DESIGN AND APPLICATION OF PROCESS-PROTECTED TRANSPARENT ENCRYPTION FILTRATION SYSTEM BASED ON WINDOWS

Abstract

The rapid development of office automation greatly improves our work efficiency. In this rapid work mode, we generate, storage and transport a large amount of electronic documents, among which involve many business secrets and personal privacy. Without appropriate protection measures, these would be leaked easily. How to ensure the safety of electronic documents has become a hot issue. However, yet more mature firewall technology and intrusion detection technology can only block the attack from the Internet, but can not solve the problem of document security in the terminal equipment. In order to protect confidential documents, some companies prohibit employees from copying the documents away by disabling USB interfaces，others even ban connecting to the Internet. Although this method ensures documents safety in terminal equipment to a certain extent, it brings great inconvenience to people's work. Given this backdrop, the technology for the security of the system level documents is emerge, namely the transparent encryption technology. There are two ways to implement this technology, hook and filter driver. Hook technology achieves interception protection by hooking related API, working on user layer, vulnerable to viruses, Trojans and other attacks. In contrast, filter drive technology achieves filter driver in the operating system kernel layer to intercept the request to the target device so as to protect documents. As the filter driver works in the kernel layer, the security is high and will not be attacked easily, which enables the filter driver become the mainstream means of achieving a transparent encryption.

Based on the further study of Windows kernel and driven development technology, this paper realizes a process-protected transparent encryption filtration system based on Windows using Minifilter framework provided by Microsoft, providing a complete solution for the security of documents. The system is composed of three parts, the server, the control program of client on user layer and the transparent encryption filter driver of client on kernel layer.

The server is responsible for giving out the strategy, monitoring the client program, distributing and managing keys, auditing information and so on. The control program of client on user layer is the bridge between the server and the kernel layer filter driver. The kernel layer module realizes the transparent encryption and filtering, encrypting and decrypting the protected documents automatically. The server communicates with the client through the Socket, and uses the heartbeat mechanism to monitor the client's online situation. The user layer and the kernel module communicate effectively through the port communication mechanism provided by Minifilter. On the basis of the completion of basic functions, the API HOOK technology is also used in this paper to realize the function of monitoring clip board and preventing the process from being terminated, thus the security of the system is further improved.

KEYWORDS Windows kernel; File System; Filter Driver; Minifilter; Transparent encrypted and decrypted

目 录

第一章 绪论

1.1 课题研究背景

在这个信息化飞快发展的时代中，我们享受着信息给我们带来的便利与快捷。随着计算机大面积的普及，网络的高速发展，人们的办公也开始走向自动化，智能化，移动化。在这样的需求下，企业、单位、个人每天都需要生成、存储、处理、传输大量的电子文档。这些文档中不乏企业的机密，个人的隐私，若不采取一定的保护措施，极易被不法分子窃取利用，损害企业、公司、个人的利益。因此，如何保证电子文档在生成、存储、处理和传输过程中的安全，应该得到高度的重视。

2008 年初的“艳照门”事件，正是个人隐私文档泄露的典型，借助网络的开放性，隐私信息迅速在网络中蔓延，对明星本人和社会的影响至深。在 2011 年中，国际上 RSA、Sony 等一大批大型国际化公司接连受到黑客攻击，导致大量用户信息丢失，企业不得不耗费大量的人力、财力、物力进行处理，这不仅影响了大量用户的个人隐私生活，公司方面也损失惨重。此外还有国内的 CSDN 社区数百万用户的数据被窃取，同样给企业造成了巨大的损失。有关于信息泄露的安全事件远远不仅于此，信息安全问题已不容忽视，成为了现今热门的话题。

目前，个人和商用计算机上最常用的信息安全保护技术是：防火墙和入侵检测。这两种技术相结合，在一定程度上阻断了网络威胁，但却无法解决企业内部人员主动泄密的问题。由此可见，单纯依靠这两种技术不能完全地解决内网信息安全问题。再者，文件信息泄露很大一部分是因为管理不当引起的。据 FBI/CSI 的调查，文件信息安全的威胁 80%以上来自企业员工的不法操作[1]。因此，保证企业内部敏感信息的安全不仅要阻断来自网络的威胁，还要防范非法人员进入公司内部的终端设备，导致敏感信息被蓄意损坏或泄露[2][3]。

面对这样一个信息安全问题，很多企业采取的安全策略是封闭办公室电脑上的 USB 口来防止员工将敏感文件拷贝带走，有的企业还禁止办公电脑接入互联网[4]。虽说这些类似的做法在一定程度上降低了企业内部信息泄露的风险，但这也给日常办公带来极大的不便。于是一种全新的电子文档安全保障技术应运而生——透明加密技术。

对于利用透明加密技术解决内网操作系统级的文件安全问题，最常用的有两种技术：钩子技术和过滤驱动技术[5]。钩子技术是通过 HOOK API 来实现，即挂钩相关的用户层或内核函数，以达到拦截保护的目的，实现上较为简单，但由于其处于操作系统的上层，因此容易受到木马、病毒等的攻击。相比之下，过滤驱动技术是在操作系统的内核层实现过滤驱动功能，即在系统内核层加入我们自己实现的过滤层，该过滤层可拦截相关的操作系统请求来达到对保护机密文档的目的，并且不会影响其他驱动的工作。由于过滤驱动工作

在内核层，而 Windows 操作系统对内核模式程序有严格的保护机制，因此安全性高，不会轻易受到攻击，这也就使得过滤驱动成为了目前实现透明加密的主流手段。不过，也正是因为在内核模式下实现，很多内核模式下的开发文档和技术微软是没有公开，再者内核模式下的程序不受操作系统控制，并且内存共享，因此，开发难度大，系统出错得不到操作系统的保护，会直接引起系统崩溃或蓝屏。

目前，在 Windows 下实现文件系统过滤驱动的框架有：基于传统型过滤驱动模型的文件过滤驱动 Sfilter 和微软专门为文件系统过滤驱动引入的 Minifilter[6]。Minifilter 实际上是 Sfilter 的进一步封装，使用 Minifilter 进行开发，可以直接使用封装好的 API，减小驱动开发的难度。相对于 Sfilter 而言，在 Minifilter 中引入了过滤管理器模型（Filter Manager Model），使得 Minifilter 具备以下优点：更好的控制过滤器的加载顺序、能在系统运行时卸载的能力、只需要处理必要的操作、更高效的利用内核堆栈空间、稳定性提高、复杂程度减小、更容易添加新的操作、可移植性大大提高、更易于实现用户层和内核层的交互。虽然底层的 Sfilter 功能性比较强，但其所具备的功能通过 Minifilter 也是可以完全实现的，这就好比在 C 语言中嵌入汇编，只是会在一定程度上影响系统原有的可移植性[7]。

综上考虑，本文提出了一个相对完备的文档保护解决方案，并采用 Minifilter 框架实现了防终止的基于 Windows 透明加密过滤系统。

1.2 国内外研究状况

透明加密的概念最早由微软发布 Windows 2000 操作系统时提出，该操作系统实现了一个加密文件系统（Encrypting File System, EFS）提供加密功能[8] [9]。EFS 是基于 NTFS 文件系统的系统级的加解密软件，能实现对机密数据的透明加密保护。用户只有正常的打开 Windows 操作系统，才能获取磁盘上的文件信息，若用户将磁盘接入其他非 Windows 操作系统，则磁盘上的文档无法正常使用。

所谓“透明”是指在不改变用户使用习惯、应用程序和电子文件格式的前提下，对指定文档类型进行实时监控，实现对受保护文档的自动加解密功能。即受保护的文档是以密文的形式存储在磁盘中，当合法用户需要查看相关文档时，系统会对文档进行自动解密以受保护的明文形式呈现，而且用户是感受不到这一层变化的。一旦受保护文档脱离该透明加密环境，将会无法正常解密受保护文档，这样一来即使是文档被拷贝带走，攻击者也只能得到无法解密的密文，从而对机密的电子文档起到了一定的保护作用。

在此之后，许多信息安全企业纷纷推出自己的文件加密产品，其中使用较为广泛的有赛门铁克（Symantec）推出的企业版端点安全软件 SEP（Symantec Endpoint Security）[10]，趋势科技推出的云安全软件 Pc-cillin[11]，以及诺顿安全软件 Norton 中的文件加密工具 DISKREET 等。不过最为典型的，还是微软的 EFS 和 Office 系列办公软件中的加密功能。但微软 EFS 加密粒度是磁盘卷，也就是说加解密操作针对的是整个磁盘卷，包括非机密文

档在内，因此用户无法选择性的加密某个或某类文件，缺乏灵活性。此外，EFS 集成在 Windows 操作系统中，用户无法在其他操作系统或平台上使用这种技术，又由于微软没有公开 EFS 的源码，第三方开发者无法根据自身特殊要求对系统进行改造或升级。而 Office 系列办公软件中的加密功能，均在用户层实现，用户可以在编辑相应文档时选择对文档进行加密，相对于 EFS 而言，Office 的加密功能是比较细粒度的，较为灵活，可以针对单个文档进行操作。但其不足也是显而易见的，其内嵌在 Office 办公套件中，即工作于用户模式下，易受到攻击，并且无法处理 Office 办公套件之外的文档类型，再者用户需要自己手动的进行加解密操作，失去了透明加密的优点。

国内的文件加密技术起步较晚。2000 年开始，国内不断出现各种文件加密系统，早期较多采用基于应用层的文件加密技术，或是使用基于文件格式转换的加密技术。之后开始较多的尝试在调用层和内核层实现透明加密。但是，目前国内在文件系统过滤驱动领域的研究仍处在一个起步阶段，利用文件过滤驱动实现透明加密也大都停留在理论研究阶段。虽然有少部分学者已实现具有加密功能的过滤驱动，不过这些大多都是采用微软的传统型过滤驱动框架，使得所实现的系统稳定性和兼容性较差，开发过程繁琐，开发周期长。为弥补传统型过滤驱动框架的不足，微软新提出了一种新的微过滤驱动框架（Minifilter）。

在对加密算法的研究方面，目前较为常用的加密算法可分为对称加密算法、非对称加密算法和散列算法等。其中，对称加密算法主要包括 DES、3DES 和 AES 等。而 AES 算法是由美国 NIST 研究所提出的新一代高级加密标准规范，用以代替早期的 DES 加密算法。AES[12] [13]是分组加密算法，密钥可为 128、192 或 256 位，明文分组长度为 128 位，经过多轮的重复和替换实现加密过程。该算法具有安全性高、加密速度快等优点。

终上所述，继续探索 Windows 内核并实现一个更完善的透明加密过滤驱动以加强对机密文档的保护还是很有必要的。此外新的微过滤驱动框架，给本文的透明加密过滤驱动实现提供了支持，而 AES 算法为驱动加密模块的算法设计提供很好的支持。

1.3 课题研究内容

通过对文件安全问题的调查与研究，结合现有的文件加密技术，综合考虑了国内外文件加密产品的优缺点，本文拟设计并采用 Minifilter 过滤驱动框架实现一个防终止的基于 Windows 的透明加密过滤系统。本文深入研究 Windows 操作系统的层次结构与内核机制、Windows 内核编程、驱动开发技术、过滤驱动开发框架，包括传统型文件过滤驱动[14]（Sfilter）和新的 Minifilter 过滤驱动框架、文件加密技术、内核（驱动）调试技术等相关知识与技术，在此基础上提出了较为完备的文档保护方案。

以下是课题主要的研究内容：

（1） 对 Windows 内核组件，尤其是文件系统做深入研究，掌握内核驱动开发的流程、工具和技术要点。

（2） 由于内核编程的特殊性，开发过程中出现死机或蓝屏是常态。因此，搭建主机-虚拟机的双机调试环境，对文件的主要操作如读、写、打开、关闭、卷挂载等进行了跟踪与详细的分析，以进一步了解文件系统的内部工作机制。

（3） 深入学习并研究微软开发的新一代驱动开发框架 Minifilter，理解和掌握其实现的流程和技术要点。在此基础上实现内核透明加密过滤驱动模块。

（4） 充分研究应用层和内核层的通信机制，在此基础上实现内核层和用户层的信息交互。

（5） 构建 C/S 架构[15]。服务器模块，主要负责策略的制定与下发，客户端的认证、监控和审计，密钥管理与分发等功能。客户端应用层控制模块，主要任务是与服务器建立连接，作为服务器与内核加密模块通信的桥梁。在此基础上深入研究 socket 通信机制与心跳机制，实现服务器和客户端的通信功能。此外还制定较为完善的客户端-服务器认证机制，内网 IP 认证和口令认证。

（6） 深入研究进程防终止技术[16]，对用户层的控制进程进行保护，防止蓄意关闭操作。对内核模块实现自我保护功能，防止非法停止加密模块的运行。

（7） 深入研究 Windows 剪切板机制，在此基础上实现对剪切板的监控功能，防止机密数据被复制/剪切带走。

1.4 论文组织结构

本论文共分为六个章节：

第 1 章：绪论。介绍了本课题的研究背景、国内外研究状况、研究内容以及本文的组织结构。

第 2 章：相关理论基础和关键技术介绍。涉及 Windows 内核系统介绍、Windows 内核驱动开发技术、文件过滤驱动的开发框架——传统型框架和 Minifilter 框架的对比研究，主要对 Minifilter 框架进行详细的介绍、密码学相关基础知识概要。

第 3 章：需求分析与总体设计。从用户提出的需求着手，提出本文的文档保护方案。并给出本系统的总体设计方案以及模块划分方案。

第 4 章：系统详细设计与实现。主要对本系统核心部分进行详细的介绍，并详细的讨论涉及到的核心部分，以及控制流程。

第 5 章：系统测试与分析。对本系统实现的一系列功能进行全面的测试。

第 6 章：总结与展望。主要对已做的工作与不足进行一个总结，对后续的研究工作进行一个计划与展望。

第二章 相关理论基础和关键技术

本章主要对 Windows 操作系统的架构、驱动开发相关知识、Minifilter 框架、HOOK API 技术以及本文所涉及的密码学知识进行介绍。

2.1 Windows 操作系统概述

从最初的 Windows 95 到现今的 Windows 10，Windows 操作系统家族可分为两大分支，一是基于 Windows 95 的 Win9x，已基本退出应用市场。另一个系列则是基于 NT 技术，如今常见的版本有 WindowsXP、Windows 7、Windows 8、Windows 10 和 Windows Server 2008 等都属于这一系列。本系统核心模块——文件过滤驱动模块，就是运行于 Windows NT 系列操作系统的内核层。

Windows NT 系列操作[17]系统被开发时有以下几个关键准则，即：尽可能最小化内核模块、采用客户端-服务器模型实现操作系统部件、采用消息传递机制实现各模块之间的信息传递。再者，Windows NT 实现上基于分层思想，从硬件层到用户层，各个相邻层次之间均使用定义好的接口进行交互，通过这样面向对象的层次抽象方法，使得 Windows NT 系列操作系统具有很好的可移植性。

Windows NT 层次结构大体上可以划分为用户模式（User Mode）和内核模式（Kernel Mode），这类似于 CPU 的特权层划分。以 Intel 的 x86 CPU 为例，特权层可分为 4 层，即 Ring0、Ring1、Ring2、Ring3，从 Ring0 到 Ring3 特权层级依次递增，Ring0 特权层级最高，可执行任意代码，而 Ring3 层级最低，可执行的代码有限。相对应下，操作系统的用户模式就相当于 CPU 中的 Ring3 层，执行的代码受到操作系统的保护，可以实现的功能有限，若涉及敏感操作需要向内核模式中的相关组件提出请求； 而内核模式就相当于 CPU 中的 Ring0 层，脱离操作系统的管制，理论上可以实现任何功能，执行任何代码，但由于失去操作系统的保护，这一层的代码一旦发生错误就有可能导致整个系统的崩溃。

2.1.1 Windows 操作系统核心架构

Windows 的层次设计结构如图 2-1 所示[18]。大体上可分为内核模式和用户模式。其中又可分为若干层，层次之间通过定义好的接口进行交互。上层组件权限较低，其运行依赖于下层组件提供的支持，而越靠近底层，组件的权限越高，底层组件为上层组件提供服务，各组件之间遵循客户端——服务器模型进行组织，通过消息进行通信。采用这样层层抽象的方式，使得 Windows 具有严格的安全机制以及很好的可移植性。

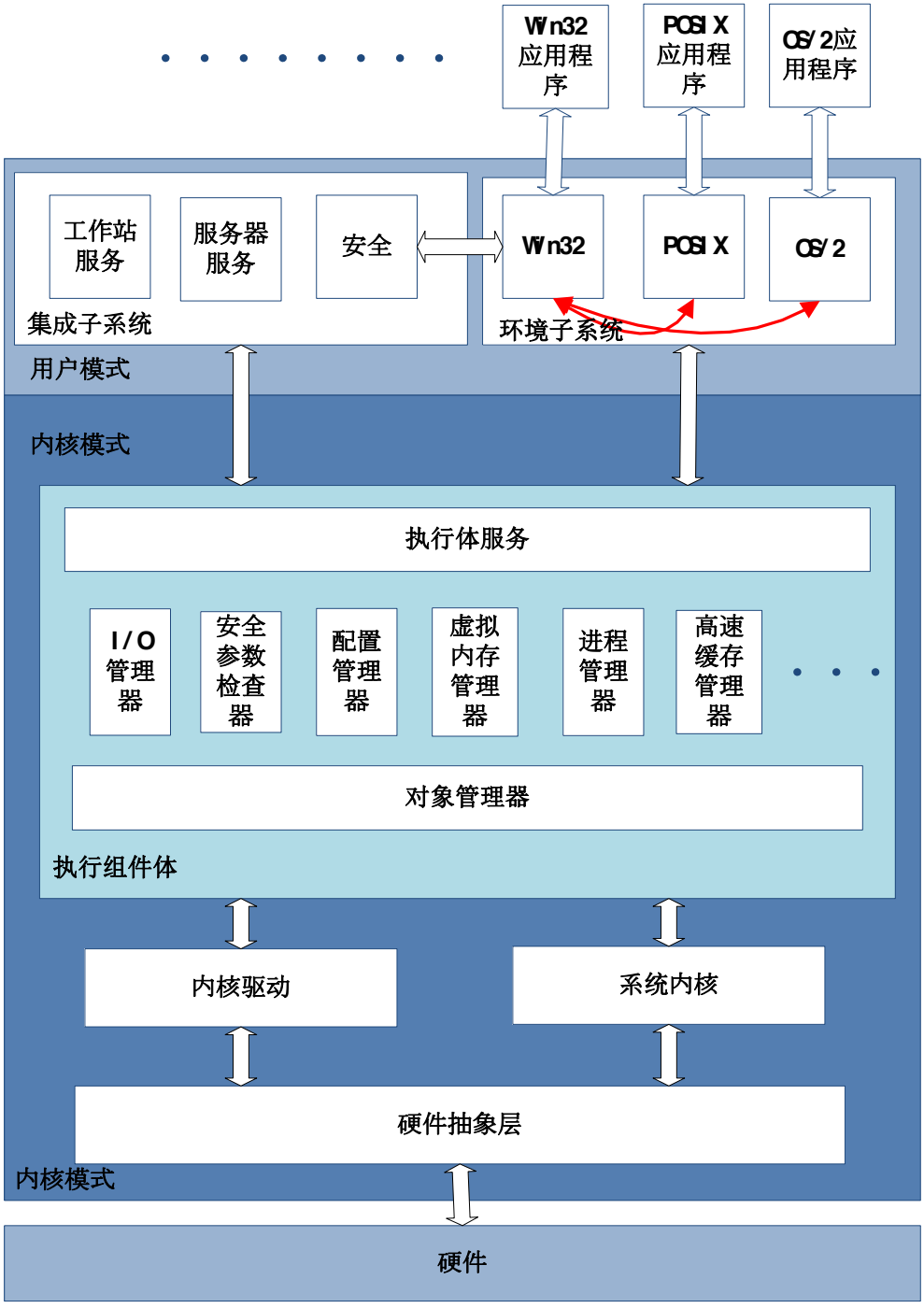


图2-1　Windows 层次结构图

在用户模式下，各个进程运行于自己的进程模块中，相互隔离，拥有一个独立且安全的运行环境，通过调用各自子系统的 API 接口以实现相关的功能。其中，Win32 子系统是最主要的子系统，他向用户层提供 API 接口，并且负责将 API 函数转化为 Native API 函数。Native API 层是连接着用户模式与内核模式的大门，它通过软中断的方式可进入内核模式，同时将用户模式的调用转化为对下层系统服务函数的调用。软中断会将 Native API 中的系统服务号和参数一同传入到内核模式中，这样一来就可以将系统服务号作为索引，在系统

服务描述符表（System Service Descriptor Table）中查到对应的系统服务函数地址，在这个过程中系统会检查参数的合法性，这就是操作系统的最后一道保护屏障，在此之后，任何操作不再受操作系统管制，在这样的情况下，出现任何错误都将会直接导致系统崩溃。在这里，系统服务函数会向执行体组件中的 I/O 管理器提出请求，之后由 I/O 管理器会以 IRP（I/O Request Package）的形式重新封装用户下发的请求，再将 IRP 请求包传递给内核驱动程序。驱动程序再继续将请求下发给硬件抽象层，硬件抽象层可以直接操作实际的硬件设备，在完成请求之后，就将操作结果依层向上层反馈。其中，执行组件体由若干个不同的功能组件构成，这些组件各自负责一块功能的实现和管理，而各个组件又是由大量的处理相应功能的内核函数构成。

由操作系统的层次结构划分可知，我们要实现的过滤驱动程序运行在内核模式下，拥有着操作系统的最高权限。因此，在进行驱动程序开发时候，要格外的谨慎小心，这里没有像用户模式那样的安全运行环境，任何的错误都将会直接引起系统崩溃蓝屏。而不同于用户模式下的应用程序，操作系统会对应用的参数进行严格的检察，对行为进行监控，对错误做出应急处理，若出错操作系统会立即终止该应用程序，以防对操作系统或其他部件造成影响。

2.1.2 用户模式

用户模式是由各子系统构成的，这些子系统可以通过 I/O 管理器（内核模式执行组件体中的一部分）将 I/O 请求发送给适当的内核模式驱动。如图 2-1 所示，构成用户模式的子系统可以划分为环境子系统（Environment subsystems）和集成子系统（Integral subsystem）。

其中，环境子系统给那些为不同操作系统类型编写的应用程序提供运行环境。没有一个环境子系统可以直接访问硬件，并且访问内存资源时必须向运行在内核模式下的虚拟内存管理器发出请求。再者，用户模式程序的权限低于内核模式程序，用户模式程序的运行会受到操作系统以及系统内核模块的管制。子系统不仅给应用程序提供了一个安全的运行环境，并且在子系统的支持下，应用程序只需要调用子系统提供的 API 接口，即可完成所需要的功能，而不需要考虑内存共享、操作系统性能、系统底层结构等复杂问题，极大的便利了用户的工作。

用户模式下的环境子系统，主要是以下三个：Win32 子系统、OS/2 子系统、POSIX 子系统。而最重要的子系统是 Win32 子系统，负责将子系统提供给应用程序的 API 转换成 Native API，通过 Native API 进入内核模式。以下是对这三个子系统的简要介绍：

（1） **Win32 子系统：**是 Windows NT Native 执行环境，可以运行 32-位的 Windows 应用程序。微软强烈建议应用程序开发者在它们开发的软件中使用 Win32 API 来获取操作系统的服务。Win32 独自负者管理与用户进行交互的设备，显示器、键盘和鼠标都由 Win32 子系统管理。它还是系统的窗口管理器，定义策略来控制图形用户接口的显示。

（2） **OS/2 子系统：**在 Intel x86 硬件平台上，为 16-位的 OS/2 应用程序提供 API 支持，但不支持 32 位字符或图形的 OS/2 应用程序。最后一个具有 OS/2 子系统的是

（3） **POSIX 子系统：**可移植操作系统结构（Portable Operating System），支持应用程序符合 POSIX1003.1 源码标准。该子系统已被 Interix 代替, 它是 Windows Services for UNIX 中的一部分。

用户模式下的集成子系统代替环境子系统负责操作系统的特定功能，包括：工作站服务、服务器服务和安全子系统。工作服务站子系统为系统提供访问网络的功能，服务子系统提供网络服务功能，安全子系统负责处理安全令牌。

2.1.3 内核模式

可执行代码指令集取决于 CPU 的硬件特权级。例如，虚拟内存管理分页表不能被用户模式进程访问，若是允许所有的用户程序进行这样的特权操作，将会很快导致系统混乱，甚至阻碍正在 CPU 上执行的关键任务，后果不堪设想。内核模式与 Intel x86 的 Ring0 对应，拥有最高权限，可执行任何代码。

Windows NT 的内核模式有下几个组成部分：

1) **执行体组件（Windows NT Executive）：**执行组件是 Windows NT 最大的组成部分。它使用内核和 HAL 提供的提供的服务，具有非常好的可移植性，并为用户模式的子系统提供丰富的系统服务。其主要的构成部分有如下几个：对象管理器、I/O 管理器、安全参数监视器、配置管理器、虚拟内存管理器、进程管理器、高速缓存管理器和本地过程调用设施等。执行体组件使用内核和 HAL 提供的服务，也正因如此，使得其在不同的框架和硬件平台之间很易于移植。
2) **内核层（Windows NT Kernel）：**内核层，即系统内核，位于 HAL 和执行体组件层之间，为操作系统的其它部分提供基本的操作系统功能。虽然内核层远小于执行组件体，但其可被视为操作系统的心脏。它主要负责中断处理和调度、陷阱（Trap）处理、通过自旋锁来支持多处理器同步、为调度进程和线程提供支持等。内核代码拥有 CPU 最高特权级，在多个处理器上并行执行，内核代码是否可以被抢占，取决于其硬件中断级（IRQL），IRQL 高的代码可以抢占 IRQL 低的代码执行。
3) **内核驱动（Kernel Driver）：**内核驱动，负责与硬件交互。内核驱动可分为三层：高层驱动程序、中间层驱动程序（功能驱动）和底层驱动程序（总线驱动）。内核驱动是操作系统与硬件设备的接口，硬件设备只有在操作系统成功安装与其对应的驱动程序后才能被操作系统识别。I/O 管理器在接收到用户模式程序发来的操作请求时，一般情况下，会将请求以 IRP 的形式进行重封装，然后将 IRP 依次顺着驱动层次向下传递，直到底层驱动，底层驱动直接与真实硬件进行交互完成指定操作之后又顺着驱动层次依次上相返回操作结果。

4) **硬件抽象层（Hardware Abstraction Layer，HAL）**：HAL 是一个相当小的软件层，它直接连接 CPU 和硬件，对底层硬件进行适当的抽象，屏蔽各种不同硬件的差异，并为操作系统上层提供统一支持。所有不同的硬件架构都在 HAL 内部进行管理，并由 HAL 导出相关的功能集，给操作系统的其它组件进行调用。由 HAL 层对硬件进行抽象之后，向操作系统其它组件屏蔽了硬件相关代码，给出统一的调用结构，使得程序不必直接访问硬件资源，从而也大大提高系统的可移植性。

2.2 驱动开发

2.2.1 相关数据结构

本小节将对驱动开发中涉及的通用结构进行介绍，即传统型过滤驱动和 Minifilter 框架都会涉及到的通用结构对象[19] [20]。

2.2.1.1 驱动对象

每个成功加载的内核驱动镜像都有且只有一个驱动对象表示，作为 I/O 管理器加载的一个实例。驱动对象用 DRIVER_OBJECT 结构组织，由对象管理器在驱动加载时创建，指向该对象的指针作为驱动入口函数（DriverEntry）的一个输入参数。驱动对象有部分域是不透明的，驱动开发者必须了解驱动对象所有域，以便初始化驱动或在需要时卸载驱动。以下是驱动对象中可访问域的列表：

1) PDEVICE_OBJECT DeviceObject：指向由驱动程序创建的设备对象指针。该域值会在 IoCreateDevice 例程被成功调用后自动更新。一个驱动可以使用这个域结合 DEVICE_OBJECT 中的 NextDevice 域来逐个遍历该驱动创建的所有设备对象。
2) PDRIVER_EXTENSION DriverExtension：指向驱动扩展的指针。驱动扩展结构中只有 AddDevice 域可以被直接访问，存储驱动的 AddDevice 例程。
3) PUNICODE_STRING HardwareDatabase：指向硬件配置信息在注册表中的路径：路径硬件配置信息在注册表中的路径：\Registry\Machine\Hardware。
4) PFAST_IO_DISPATCH FastIoDispatch：指向驱动的 Fast I/O 入口结构。
   PDRIVER_INITIALIZE DriverInit：DriverEnty 例程入口，由 I/O 管理器建立。
5) PDRIVER_STARTIO DriverStartIo：驱动的 StartIo 例程的入口，可为 NULL。
6) PDRIVER_UNLOAD DriverUnload：驱动卸载例程的入口，任何时候，它都是在驱动初始化时由 DriverEntry 例程设置。可为 NULL。
7) PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION+1]：这是存放驱动派遣函数（DispatchXxx）入口的数组。数组的索引值是 IRP_MJ_Xxx，这个值代表各个 IRP 主功能号。每个驱动都必须在这个数组中为驱所要处理的 IRP_MJ_Xxx 请求设置函数入口。

2.2.1.2 设备对象

一个驱动程序可以创建一个或多个设备对象，同一驱动程序创建的设备对象间通过 NextDevice 域连接成一个设备链表，该链表的头结点可以通过驱动对象的 DeviceObject 域获取，该链表由 I/O 管理器维护。此外，值得注意的是，在内核模式下，请求基本上均以 IRP 的方式进行传递，而设备对象是唯一能接收 IRP 的实体。内核中，设备对象用 DEVICE_OBJECT 结构来表示。DEVICE_OBJECT 结构的关键域定义如下：

```
typedef struct _DEVICE_OBJECT {
 . . .
 PDRIVER_OBJECT  DriverObject;
 PDEVICE_OBJECT  NextDevice;
 PDEVICE_OBJECT  AttachedDevice;
 PIRP  CurrentIrp;
. . .
 __volatile PVPB  Vpb;
 PVOID  DeviceExtension;
 CCHAR  StackSize;
. . .
 } DEVICE_OBJECT, *PDEVICE_OBJECT;
```

1) DriverObject：指向 DRIVER_OBJECT 结构的指针，通过这一个域，可以获取到创建该设备对象的驱动对象。这是一个只读域，在调用 IoCreateDevice 或 IoCreateDeviceSecure 时，自动设置。
2) NextDevice：指向设备链表中的下一设备对象。这是一个可读/写域。
3) AttachedDevice：指向附加设备对象，若没有则该域为 NULL。一般情况下，AttachedDevice 域指向的设备对象是一个过滤驱动的设备对象。过滤驱动可使用 IoAttachDevice 或 IoAttachDeviceByPointer 例程将自己创建的设备附加到目标设备上，以拦截发往目标设备（被附加的设备）的 I/O 请求，从而实现过滤的功能。
4) Vpb：指向与该设备对象相关的卷参数块（Volume Parameter Block，VPB）。VBP 可提供到任何一个没有名字的代表一个已挂载卷的逻辑设备对象的连接。
5) DeviceExtension：指向设备扩展的指针。这个结构由驱动开发者自行定义，是驱动创建设备对象时的一个参数，用于存储开发者指定的信息。
6) StackSize：指定发往该驱动的 IRP 在堆栈空间中历经的最小结点数。在新创建设备对象时，IoCreateDevice 和 IoCreateDeviceSecure 将这个值设置为 1；底层驱动可以忽略这个域。如果调用 IoAttachDevice 或 IoAttachDeviceToDeviceStack，I/O 管理器会自动修改堆栈空间中上层驱动设备对象 StackSize 的值。除以下情况外，在驱动将其自身绑定到其他驱动上之后调用 IoGetDeviciObjectPointer 时，必须显式的设置自己设备对象中的 StackSize 域为下一层驱动的设备对象的 StackSize 域值+1。

2.2.1.3 输入输出请求包（IRP）

输入输出请求包（I/O Request Packet，IRP）是内核模式结构，被 Windows Driver Model (WDM) 和 Windows NT 设备驱动用来实现彼此间的通信，以及与操作系统间的通信。该数据结构被用来描述 I/O 请求，可以将其看成是“I/O 请求描述符”。使用这样一个结构对通信过程中涉及的大量参数进行封装（例如缓冲地址、缓冲大小、I/O 函数类型等），使得通信过程变得很简单，各部件之间只需传递一个指向该持久性数据结构的指针即可。

驱动程序在接收到 IRP 请求后，主要有以下几种处理方式：

1) 当 I/O 请求不能被马上处理时，对应的 IRP 可以被挂载到请求队列中。
2) 根据 IRP 请求，直接操作硬件，完成 IRP 请求并返回，当然也可以不做任何处理直接返回该 IRP 请求。
3) 将 IRP 发送到更底层的驱动中，并等待底层驱动将处理结果返回。
4) 接收到 IRP 后，分配新的 IRP 并将其发送到其他驱动中去，并等待结果返回。
5) 在返回之前，可以修改 IRP 来改变请求原本的意图。

值得注意的是，对 IRP 的操作不仅仅局限于以上三种，并且一般情况下都不是单独操作，而是几种方式相结合。

一般情况下，IRP 由 I/O 管理器创建来响应用户模式的请求，有时候，也可由即插即用管理器（Plug-And-Play Manager）、电源管理器、以及其它系统组件创建，甚至可由驱动创建并传递到给其它驱动。IRP 数据结构的主要成员如下所示：

```
typedef struct _IRP {
 . . .
 PMDL  MdlAddress;
 ULONG  Flags;
union {
  struct _IRP  *MasterIrp;
     . . .
  PVOID  SystemBuffer;
 } AssociatedIrp;
 . . .
 IO_STATUS_BLOCK  IoStatus;
 KPROCESSOR_MODE  RequestorMode;
 BOOLEAN PendingReturned;
 . . .
 BOOLEAN  Cancel;
 KIRQL  CancelIrql;
 . . .
 PDRIVER_CANCEL  CancelRoutine;
 PVOID UserBuffer;
 union {
```

```
  struct {
    . . .
    union {
     KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
     struct {
      PVOID  DriverContext[4];
     };
    };
    . . .
    PETHREAD  Thread;
    . . .
    LIST_ENTRY  ListEntry;
    . . .
  } Overlay;
  . . .
 } Tail;
} IRP, *PIRP;
```

IRP 主要结构成员介绍：

1) MdlAddress ：内存描述符表（MemoryDeseriptionList，MDL）指针，MDL 描述一个与该请求相关的用户模式缓冲区。若驱动使用直接 I/O（Direct I/O），则 IRP 主功能号与 MDL 用途的对应关系如下：IRP_MJ_READ，MDL 描述一个待填充的空缓冲区；IRP_MJ_WRITE，MDL 描述一个放有数据的缓冲区；IRP_MJ_DEVICE_CONTROL 或 IRP_MJ_INTERNAL_DEVICE_CONTROL，若 IOCTL 码指定 METHOD_IN_DIRECT 传输类型，那么 MDL 描述一个放着数据的缓冲区，若 IOCLT 码指定 METHOD_OUT_DIRECT 传输类型，那么 MDL 描述一个待设备或驱动填充的空缓冲区。若驱动没有使用直接 I/O 方式，则该域为 NULL。
2) AssociatedIrp：三指针联合域，其中 AssociatedIrp.SystemBuffer 指向一个系统空间缓冲区。

   如果驱动使用缓冲 I/O（Buffer I/O），那么该缓冲的使用目的与 IRP 主功能号的对应关系如下：（1）IRP_MJ_READ，缓冲区从设备或驱动中获取数据，缓冲区的长度由驱动的 IO_STACK_LOCATION 结构的 Parameters.Read.Length 域指定。（2）IRP_MJ_WRITE，缓冲区为设备或驱动提供数据，缓冲区的长度由驱动的 IO_STATCK_LOCATION 结构的 Parameters.Write.Length 域指定。（3）IRP_MJ_DEVICE_CONTROL 或 IRP_MJ_INTERNAL_DEVICE_CONTROL，缓冲区同时代表提供给 DeviceIoControl 和 IoBuildDeviceIoControlRequest 的输入输出缓冲区，输出的数据会覆盖输入的数据。对于输入，缓冲区的长度由驱动的 IO_STACK_LOCATION 结构的 Parameters.DeviceIoControl.InputBufferLength 域指定。对于输出，缓冲区的长度

由驱动的IO_STACK_LOCATION结构的Parameters.DeviceIoControl.OutputBufferLength域指定。

如果驱动使用直接I/O（Direct I/O），缓冲区的使用意图与IRP主功能号的对应关系如下：（1） IRP_MJ_READ，该域为NULL。（2）IRP_MJ_WRITE,该域为NULL。（2）IRP_MJ_DEVICE_CONTROL 或IRP_MJ_INTERNAL_DEVICE_CONTROL，情况与缓冲I/O一样。

3) IoStatus：包含IO_STATUS_BLOCK结构，存放操作状态和信息存，在驱动调用IoCompleteRequest完成请求时返回。

4) UserBuffer：若主功能号为IRP_MJ_DEVICE_CONTROL或IRP_MJ_INTERNAL_DEVICE_CONTROL，且I/O控制码为METHOD_NEITHER时，该域包含一个输出缓冲区地址。

2.2.1.4 IO堆栈位置(IO_STACK_LOCATION)结构

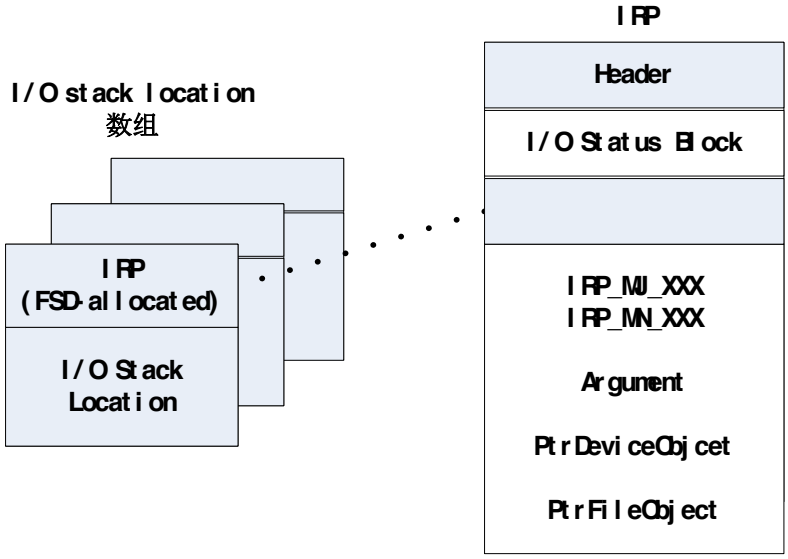


图2-2 IRP结构图

I/O管理器创建IRP时，为驱动层次链（A Chain Of Layered Drivers）中的每个驱动设置一个与其对应的I/O Stack Location。I/O Stack Location 由IO_STACK_LOCATION结构表示，这是一个与IRP相关的I/O堆栈的入口。I/O管理器为每个IRP创建一个I/O Stack Location数组（Stack Locations），存放将要处理该IRP请求的所有驱动对应的IO_STACK_LOCATION结构。每个驱动都拥有IRP中的一个I/O Stack Location，可以调用IoGetCurrentIrpStackLocation 来获取与该驱动相关的I/O Stack Location，并负责调用IoGetNextIrpStackLocation来设置相邻下一层驱动的I/O Stack Location。任意高层驱动的I/O Stack Location 都可以被用来存放与一个操作相关的上下文，便于驱动的IoCompletion例程可以执行它的清理操作。任意高层的驱动为低层驱动分配IRP时，要根据相邻下一层驱动

的设备对象的 StackSize 域值来决定这个新的 IRP 应该有多少个 I/O Stack Location。图 2-2 详细描绘了 IRP 中的内容以及 IRP 与 I/O Stack Location 的关系。

以下给出 IO_STACK_LOCATIONZ 的结构。与 IRP 主功能号对应的特定参数被组织在结构的一个联合体（union）中，也就是说确定的 IRP 主功能号会对应联合体中特定的结构域，其中存放与该 IRP 主功能号对应的参数列表。

```
typedef struct _IO_STACK_LOCATION {
 UCHAR  MajorFunction;
 UCHAR  MinorFunction;
 UCHAR  Flags;
 UCHAR  Control;
 union {
     // Parameters for IRP_MJ_CREATE
     struct {
        PIO_SECURITY_CONTEXT  SecurityContext;
        ULONG  Options;
        USHORT POINTER_ALIGNMENT  FileAttributes;
        USHORT  ShareAccess;
        ULONG POINTER_ALIGNMENT  EaLength;
     } Create;
    //      other request
           . . .
  PDEVICE_OBJECT DeviceObject;
  PFILE_OBJECT  FileObject;
} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

1) MajorFunction：IRP 主功能号，表明 I/O 请求类型。
2) MinorFunction：IRP 主功能号之下的次功能号。
3) Flags：几乎是专属于文件系统使用的面向特定请求类型的域。可移除媒体设备驱动通过检查 Flags 是否为读请求设置了 SL_OVERRIDE_VERIFY_VOLUME 来决定是否继续读操作，即使 Flags 的 DO_VERIFY_VOLUME 值被设置。在可移动媒体设备驱动之上的中间层驱动，对 IRP_MJ_READ 请求进行处理时，必须将该 Flags 复制到下一层驱动的 I/O Stack Location 中。
4) Control：驱动通过检查该只读域来判断是否要标记 SL_PENDING_RETURNED。
5) Parameters：联合体中的结构：struct {… … } Xxx。联合体的内容因 IRP 主、次功能号（MajorFunction、MinorFunction）的不同而不同，存放着指定请求的特殊参数。
6) DeviceObject：设备对象指针。
7) FileObject ：表示文件对象的指针。该域若存在，则与 DeviceObject 域相关。

2.2.2 驱动、设备层次框架结构与请求处理机制

介绍设备对象的时，曾提到设备对象中有这样两个域：AttachedDevice 和 StackSize。设备对象是唯一可以接收 IRP 请求的实体，设备对象接收到 IRP 之后，才交由创建它的驱动程序进行处理。

上文曾提到内核驱动分为三个层次，即高层驱动、中层驱动（功能驱动）、下层驱动（总线驱动）。IRP 顺着驱动层次依次向下传递请求，期间可以由任何一个驱动程序完成并返回。但是，IRP 是如何知道自己要交由哪个设备对象进行处理呢？而 IRP 在设备之间又是怎样传递的呢？设备和驱动之间又是什么关系呢？一个用户请求是怎样层层下发最后得到处理的呢？以下就设备、驱动之间的关系，以及各自的组织形式作进一步介绍，具体说明用户请求的下发与处理过程，IRP 的传递机制与组织方式。

2.2.2.1 驱动与设备组织架构

当我们将 U 盘或其他移动设备通过 USB 接口连接到电脑上时，电脑常常会显示“正在扫描安装设备驱动”类似的提示，若是这样一个过程失败，即安装对应于移动设备的驱动程序失败，我们的移动设备将无法被电脑识别。从这一角度看，内核驱动可以是操作系统和硬件设备的一个接口。而设备对象又相当于设备的一个抽象或者映射，组织和存放着设备的相关信息。这里需要注意的是，所谓的“设备”并不一定全都是物理设备，内核驱动可分为三层，其中只有最下层的总线驱动直接与硬件挂钩，而其他驱动是挂载在一些内核模块之上以实现类似于物理驱动的功能，往往这一部分驱动也被称作内核模块。本课题实现的透明加密过滤系统的加解密模块就是这样一个挂载在文件系统之上的过滤驱动，同样也是一个内核模块。

需要说明的是，本文所说的驱动与驱动对象相对应、设备与设备对象相对应，不作严格区分。后面篇幅不再重申。

如图 2-3 所示，由同一驱动创建的设备对象使用 NextDevice 指针域连接起来，每个设备对象都有一个 DriverObject 指针，指向创建该设备的驱动，而该设备链的头结点由驱动对象中的 DeviceObjcet 域指向。每个设备的 NextDevice 域和驱动的 DeviceObject 域可构成一个循环链表，这使得遍历一个驱动对象创建的所有设备对象变得很容易，只要知道一个驱动对象或是一个设备对象就可以遍历由同一个驱动对象所创建的所有设备对象。

不同驱动的设备对象之间通过“附加（Attache）”这样一个动作连接起来。驱动对象可以调用 IoAttachDevice 或 IoAttachDeviceToDeviceStack 例程将其创建的设备对象附加到某个指定的设备上，这样依次附加，就形成一个设备栈的结构，其中包含存在相互附加关系的设备对象，在这样一个堆栈结构中，附加设备在被附加设备之上，被附加设备通过 AttachedDevice 指向附加设备。设备栈的深度，由设备中的 StackSize 指定，最小值为 1，表明从当前设备到底层设备之间的节点数（包括当前设备）。

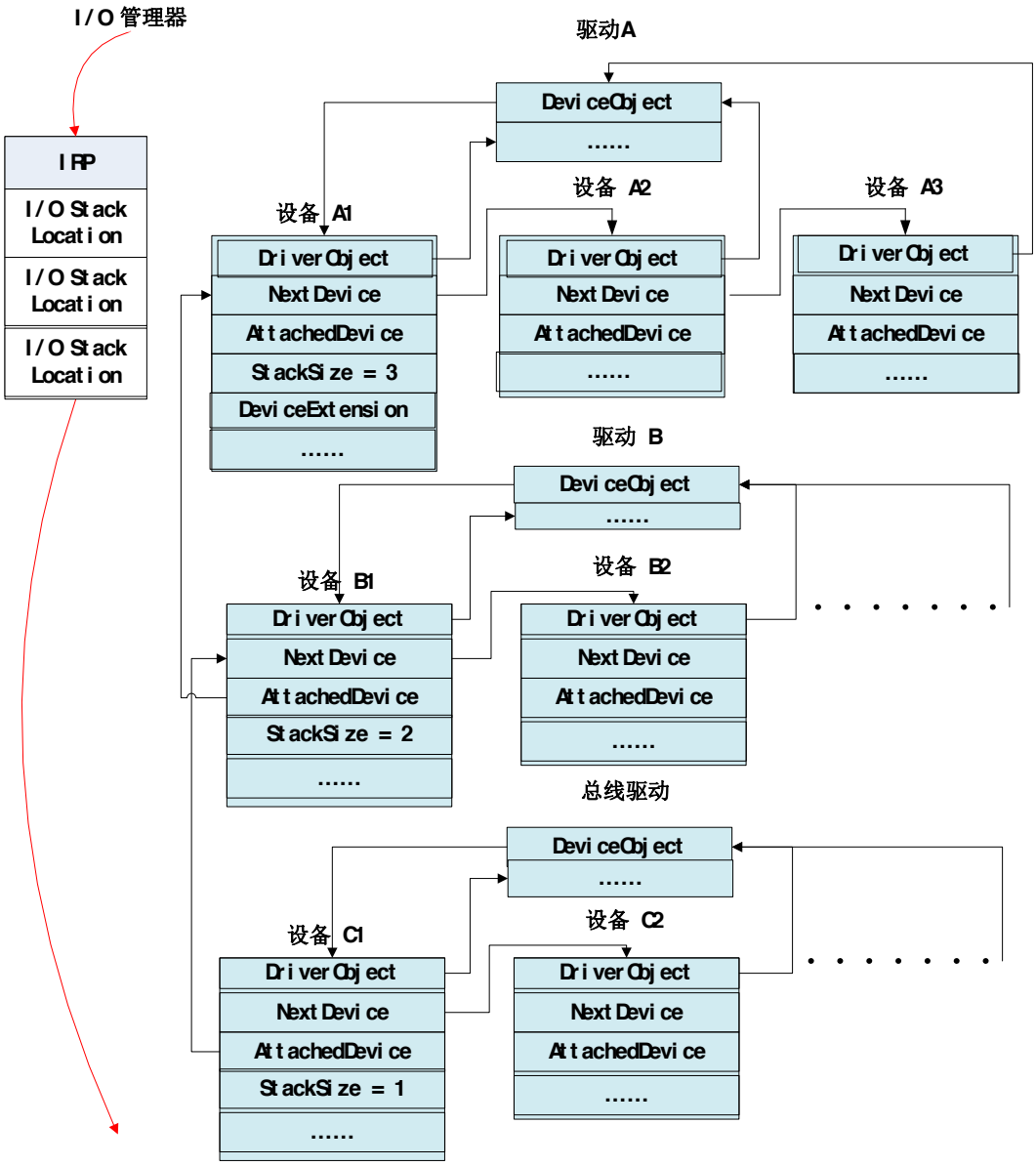


图2-3 驱动——设备架构图

I/O管理器以IRP的形式重新封装用户层下发的请求参数并将其下发到内核驱动层，IRP会顺着设备栈依次向下传递，直到被完成返回，再顺着设备栈依次向上返回请求处理结果。其中IRP中存储着一个Io Stack Location数组，每个数组元素与将要处理该IRP的驱动相对应。例如图2-3中有三层驱动设备对象将要处理这个IRP请求，因此数组对应着有3个元素。

由以上的介绍可以看出，我们只需创建一个自己的驱动，并且创建一个设备对象，将这个设备对象附加到我们需要监控的设备对象上，我们就可以在目标设备接收到发向它的IRP之前拦截到这些信息，从而实现我们预想的功能。

此外，观察图2-3中的设备栈不难看出，在当前设备中，我们只能通过AttachedDevice

域获取到附加在它之上的设备对象，但是却无法获取到其下层的设备对象。若我们有这样的需求，可以利用 DeviceExtension（设备扩展）域，指向我们自己定义的结构，在附加设备对象的同时，把指向下层设备的指针存储到该域中。

2.2.2.2 一个用户请求的处理过程

图 2-4 显示了一个子系统为某个应用程序打开表示一个数据文件的文件对象时，请求被处理并返回处理结果的一个过程。

1. I/O 系统服务被子系统调用来打开一个命名文件。
2. 对象管理器被 I/O 管理器调用来查找这个命名文件。I/O 管理器将会调用安全引用监控器，来检查这个子系统是否对该文件有访问权限。
3. 如果卷未被挂载，则打开请求将会被 I/O 管理器暂时地挂起，并继续访问一个或多个文件系统，直到能在这个文件系统中找到对应文件在存储设备中的映射为止。若文件系统有挂载的卷，I/O 管理器将继续处理这个请求。
4. I/O 管理器为打开请求分配 IRP 内存空间，并进行初始化。
5. I/O 管理器将 IRP 传递给文件系统驱动。文件系统驱动访问它在 IRP 中对应的 I/0 Stack Location 来决定它应该进行的操作，进行参数检查，确定被请求的文件是否在高速缓存中，如果没有，就需要为更下层的驱动建立其对应的 I/O Stack Location。
6. 驱动调用内核模式下的例程来处理并完成 IRP。
7. 请求操作是否成功的结果被存放到 IRP 的 I/O Status Block 结构，然后将 IRP 返回给 I/O 管理器。
8. I/O 管理器从 IRP 中读取请求操作结果，然后将结果信息通过子系统返回给最初的调用者。
9. I/O 管理器释放已经完成的 IRP。
10. 如果打开请求成功的话，将成果获取到文件对象的一个句柄并由 I/O 管理器返回给子系统。如果打开请求出错，则返回适当的错误信息。

子系统成功打开一个文件对象、一个设备、或一个卷后，会获得一个表示这些对象的一个句柄，那么在这之后，子系统就可以使用这个句柄来向已打开的对象发送 I/0 请求（通常是：读、写、或设备 I/0 控制请求），然后由 I/0 管理器将这些请求以 IRP 的形式路由到适当的驱动中进行处理。

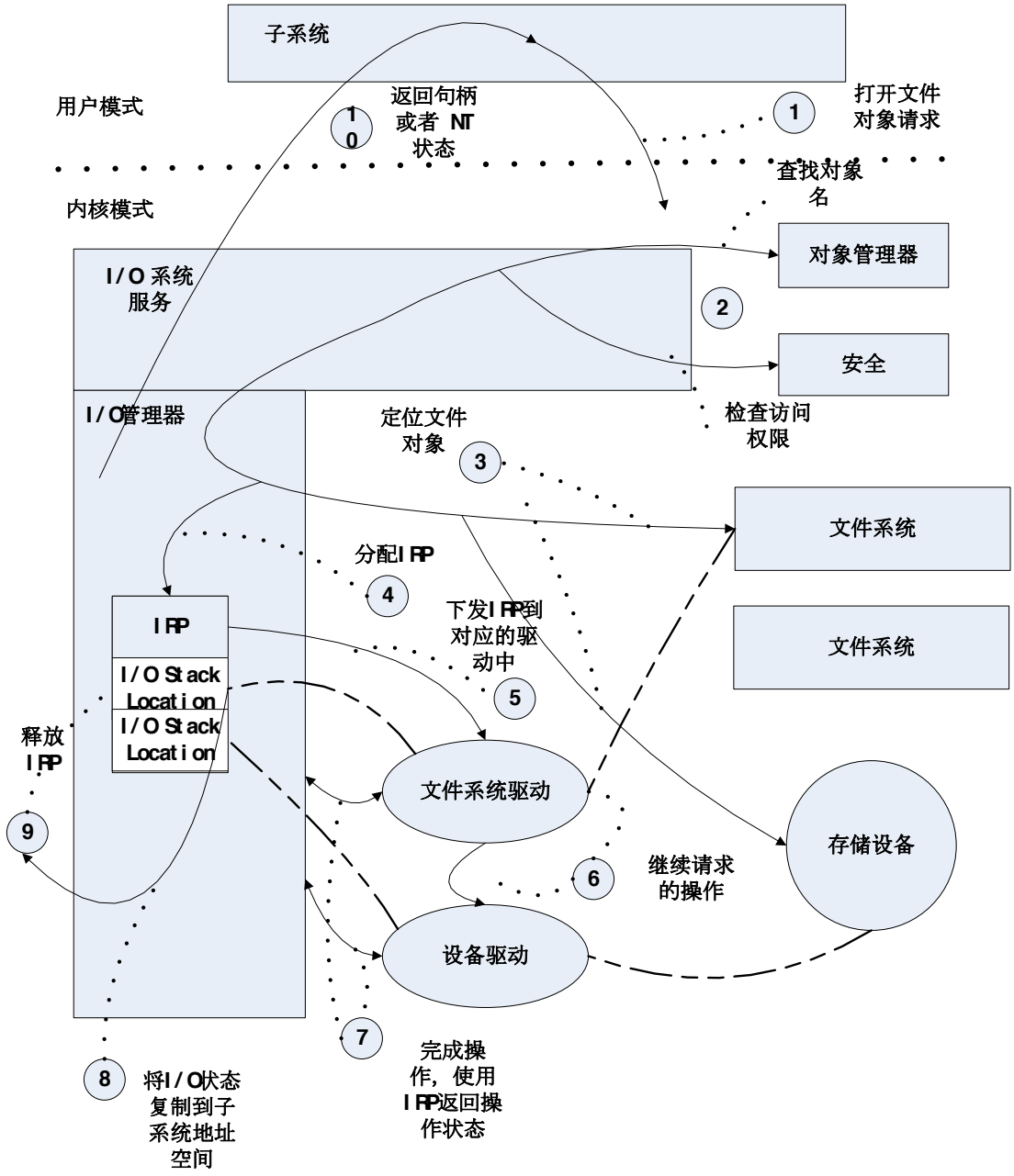


图2-4　一次打开文件操作的处理流程图

2.3　文件系统驱动与文件系统过滤驱动

在 Windows 中的文件系统，其实是在存储系统之上实现的文件系统驱动。在 Windows 中的每个文件系统都被设计来提供可靠的数据存储，其中还实现多种性能以解决用户的多种需求。Windows 中设计的文件类型通常是 NTFS 和 FAT，本文充分考虑这两种文件系统的异同点，使得过滤系统同时兼容这两种文件系统类型。Windows 的文件系统十分复杂，致使实现文件系统驱动很困难，然而，文件系统和文件系统驱动都可以提供任何自定义的行为来修改文件系统中原本存在的操作，因此重新开发一个文件系统驱动来实现某些功能需求是没有必要的。

文件系统过滤驱动[21]可以拦截发往文件系统或文件系统驱动的请求。文件系统过滤驱动在请求到达目的设备之前截获请求并进行适当地处理可扩充或代替原目标设备的功能。相比于文件系统和文件系统驱动，文件系统过滤驱动只需拦截所关心的请求，而不需要完全实现文件系统的功能，使得开发过程变得简单很多。常见的文件过滤驱动有：防毒过滤驱动，备份代理和加密过滤驱动等。加密过滤驱动加载在文件系统驱动之上并拦截发给它的请求，然后对请求进行分析和处理，以实现对机密文档的加密功能，本文实现的正是这样一个加密过滤驱动。

2.4 Minifilter 框架介绍

本文实现的透明加密过滤系统，主要的加密模块是采用 Minifilter[22]框架实现的一个文件系统过滤驱动，加载在文件系统之上，以拦截用户发往文件系统的请求，从而实现加/解密功能。Windows 的过滤管理器（Filter Manager,Fltmgr）[23]为文件系统过滤提供服务。过滤管理器为文件系统和文件过滤驱动的开发提供一个很好的框架，使得开发者不需要去管理复杂的文件 I/O。过滤管理器简化了第三方开发过滤驱动的过程，并解决了存在于传统型过滤驱动模型中的很多问题，例如，过滤管理器可以通过指定一个层级（Altitude）来控制过滤驱动的加载次序。使用过滤管理器模型来开发的过滤驱动被称为 Minifilter，即微过滤驱动。每个微过滤驱动均由指定的 Altitude 唯一标识，Altitude 决定该微过滤驱动被加载到 I/O 堆栈中的位置。Altitude 由微软分配和管理。以下将对 Minifilter 框架以及关键技术作进一步的介绍。

2.4.1 过滤管理器模型

微过滤驱动（Minifilter Driver）被加载时过滤管理器开始工作。Fltmgr 为一个目标卷挂载到文件系统堆栈中，一般来说，一个卷对应一个逻辑盘，也就是我们常说的 C：、D：、E：、F：盘，一个盘对应一个卷。Minifilter 驱动通过 Fltmgr 为自己注册需要过滤的 I/O 操作，并直接附加到文件系统堆栈中。

传统型过滤驱动（Legacy Filter Driver）的加载位置是在系统启动时，通过驱动所属的加载次序组（Load Order Group）来决定，它在文件系统 I/O 堆栈中的加载位置与其他过滤驱动有关。例如，一个防毒过滤驱动在堆栈中的位置就应该高于一个复制过滤驱动，因为只有这样防毒过滤驱动才能在复制过滤驱动复制数据之前对数据进行检查。因此，FSFilter Replication 加载次序组中的过滤驱动会晚于 FSFilter Anti-Virus 组被加载。每个加载次序组都有系统定义的类和类 GUID 与其对应，在安全驱动的 INF 文件中用到。

Minifilter 驱动与传统型过滤驱动一样也需要按照特定的次序附加。但 Minifilter 驱动附加的次序由一个唯一标识符 Altitude 来决定。一个 Minfinite 驱动在指定卷上按特定的 Altitude 进行的一次附加，被称作该 Minifilter 驱动的一个实例。Minifilter 驱动使用 Altitude 来保证自身的实例被加载到适当的位置，不影响其它的 Minifilter 驱动实例和传统型过滤驱

动。Altitude 还决定过滤管理器调用 Minifilter 驱动来处理 I/O 请求的次序。

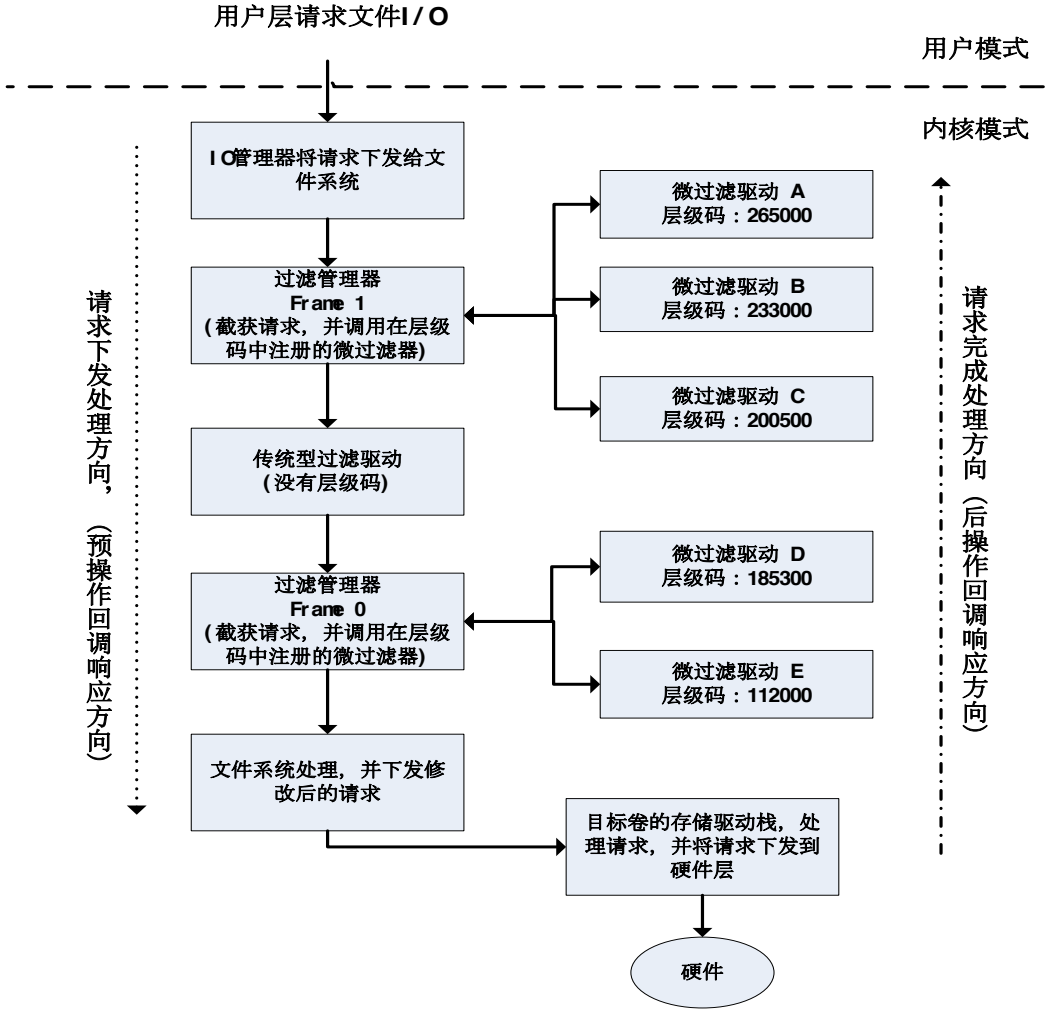


图2-5 Minifilter 驱动-传统型过滤驱动共存的文件系统 I/O 堆栈层次结构

图 2-5 给出了 I/O 堆栈结构示例，说明被加载的 Minifilter 驱动和传统型过滤驱动之间的层次关系，I/O 请求的处理方向。每个 Minifilter 驱动都可以为感兴趣的 I/O 操作注册预操作回调例程（Preoperation Callback Routine）和后操作回调例程（Postoperation Callback Routine），预操作回调例程在 I/O 请求下发时被触发，后操作回调例程在 I/O 请求被完成返回时被触发。过滤管理器可以在 I/O 堆栈多处附加过滤设备对象，以保持 Minifilter 驱动与传统型过滤驱动之间的互操作性。用帧（Frame）来代表一个过滤管理器管理的过滤设备对象。从传统型过滤驱动的角度来看，每个 Frame 就相当于另一个传统型过滤驱动。

2.4.2 关键数据结构

（1） FLT_REGISTRATION

FLT_REGISTRATION 结构作为 FltRegisterFilter 例程的一个参数，存放 Minifilter 驱动的注册信息，其中包括一些参数、上下文结构和回调例程。其结构如下：

```
typedef struct _FLT_REGISTRATION {
  USHORT  Size;
  USHORT  Version;
  FLT_REGISTRATION_FLAGS  Flags;
  CONST FLT_CONTEXT_REGISTRATION  *ContextRegistration;
  CONST FLT_OPERATION_REGISTRATION  *OperationRegistration;
  PFLT_FILTER_UNLOAD_CALLBACK  FilterUnloadCallback;
  PFLT_INSTANCE_SETUP_CALLBACK  InstanceSetupCallback;
  PFLT_INSTANCE_QUERY_TEARDOWN_CALLBACK  InstanceQueryTeardownCallback;
  PFLT_INSTANCE_TEARDOWN_CALLBACK  InstanceTeardownStartCallback;
  PFLT_INSTANCE_TEARDOWN_CALLBACK  InstanceTeardownCompleteCallback;
  PFLT_GENERATE_FILE_NAME  GenerateFileNameCallback;
  PFLT_NORMALIZE_NAME_COMPONENT  NormalizeNameComponentCallback;
  PFLT_NORMALIZE_CONTEXT_CLEANUP  NormalizeContextCleanupCallback;
  #if FLT_MGR_LONGHORN
  PFLT_TRANSACTION_NOTIFICATION_CALLBACK TransactionNotificationCallback;
  PFLT_NORMALIZE_NAME_COMPONENT_EX  NormalizeNameComponentExCallback;
  #endif // FLT_MGR_LONGHORN
} FLT_REGISTRATION, *PFLT_REGISTRATION;
```

1) Size：以字节为单位，是 FLT_REGISTRATION 结构的大小。Minifilter 驱动必须把这个域设置为：sizeof(FLT_REGISTRATION)。微软习惯在 Windows 内核的数据结构前面加上大小字段，以便于排错。
2) Version：FLT_REGISTRATION 结构的版本号。Minifilter 驱动必须将这个域设置为：FLT_REGISTRATION_VERSION。
3) Flags：Minifilter 注册标记掩码。Flags 只有两种设置情况，一种是为 NULL；另一种是设置为 FLTFL_REGISTRATION_DO_NOT_SUPPORT_SERVICE_STOP，这表示，当停止服务时，Minifilter 不会响应也不会调用卸载例程，即使 Minifilter 已经注册卸载例程，即 FilterUnloadCallback 域不为 NULL。
4) ContextRegistration：可变长的结构数组，用于注册上下文。
5) OperationRegistration：可变长的结构数组，用于注册过滤驱动关心的 I/O 操作。
6) FilterUnloadCallback：Minifilter 驱动卸载时触发的回调例程。可选。
7) InstanceSetupCallback：实例安装回调例程。可选。 InstanceQueryTeardownCallback：实例销毁回调例程，可选。
8) InstanceTeardownStartCallback：实例解除绑定回调例程，可选。
9) InstanceTeardownCompleteCallback：实例解除绑定完成例程。可选。
10) 余下其它域很少使用到，这里就不再作介绍，直接设置为 NULL 即可。

（2） FLT_CONTEXT_REGISTRATION

这里对前面介绍到的 FLT_REGISTRATION 结构中的 ContextRegistration 作进一步介绍。该结构在 Minifilter 驱动注册时被用来注册上下文类型。结构定义如下：

```
typedef struct _FLT_CONTEXT_REGISTRATION {
 FLT_CONTEXT_TYPE ContextType;
 FLT_CONTEXT_REGISTRATION_FLAGS Flags;
 PFLT_CONTEXT_CLEANUP_CALLBACK ContextCleanupCallback;
 SIZE_T Size;
 ULONG PoolTag;
 PFLT_CONTEXT_ALLOCATE_CALLBACK ContextAllocateCallback;
 PFLT_CONTEXT_FREE_CALLBACK ContextFreeCallback;
 PVOID Reserved1;
 } FLT_CONTEXT_REGISTRATION, *PFLT_CONTEXT_REGISTRATION;
```

1) ContextType：上下文类型，可取以下值：FLT_FILE_CONTEXT（Windows Vista 或之后的 Windows 版本）、FLT_INSTANCE_CONTEXT、FLT_STREAM_CONTEXT、FLT_STREAMHANDLE_CONTEXT、FLT_TRANSACTION_CONTEXT（Windows Vista 或之后的 Windows 版本）、FLT_VOLUME_CONTEXT。
2) Flags：标记位掩码，表明过滤管理器应该如何从一个上下文大小固定的后备链表（Lookaside List）中为新的上下文分配内存空间。
3) ContextCleanupCallback：指向 Minifilter 定义的上下文清理例程。可为 NULL。若指定清理例程，则在上下文被删除之前会触发该回调例程。
4) Size：若使用的上下文大小固定，该域以字节为单位指定上下文大小。0 也是有效值。若上下文大小可变，则该域必须设置为：FLT_VARIABLE_SIZED_CONTEXTS。
5) PoolTag：内存池标记。
6) ContextAllocateCallback：上下文内存分配回调例程，可为 NULL。
7) ContextFreeCallback：上下文释放回调例程，可为 NULL。
8) Reserved1：系统保留字段，必须设置为 NULL。

在 Minifilter 注册上下文时，使用 FLT_CONTEXT_REGISTRATION 结构数组存储必要参数。数组中的每个单元代表一个需要注册的上下文类型，按照 FLT_CONTEXT_REGISTRATION 结构域顺序进行必要设置，后面不需要的域可省略。数组一般定义如下：

```
const     FLT_OPERATION_REGISTRATION Callbacks[] = {
    {
        FLT_VOLUME_CONTEXT,     //    上下文类型
        0,                      //    标记 Flags
        CleanupContext,         //    上下文清除回调例
        VOLUME_CONTEXT_SIZE,    //    大小,Size
        VOLUME_CONTEXT_TAG      //    PoolTag
```

```
    },
    //    …… 填写其他类型上下文的信息 ……
    {    FLT_CONTEXT_END }// 结束标记必须有
};
```

（3） FLT_OPERATION_REGISTRATION

这里对前面介绍到的 FLT_REGISTRATION 结构中的 OperationRegistration 作进一步介绍。Minifilter 在注册时使用该结构来注册所关心的 I/O 操作。结构定义如下：

```
typedef struct _FLT_OPERATION_REGISTRATION {
  UCHAR  MajorFunction;                                  // 请求主功能号 IRP_MJ_Xxx
  FLT_OPERATION_REGISTRATION_FLAGS  Flags;     // 标志位掩码，可选
  PFLT_PRE_OPERATION_CALLBACK  PreOperation;   // 预操作回调函数
  PFLT_POST_OPERATION_CALLBACK  PostOperation; // 后操作回到函数
PVOID  Reserved1;                                      // 保留字段
} FLT_OPERATION_REGISTRATION, *PFLT_OPERATION_REGISTRATION;
```

1) MajorFunction:请求主功能号，IRP_MJ_Xxx。
2) Flags：标志位掩码，指明回调例程是否响应高速缓存 I/O 或者分页 I/O 操作。
3) PreOperation：预操作回调例程。
4) PostOperation：后操作回调例程。
5) Reserved1：系统保留字段，必须设置为 NULL。

Minifilter 在注册时，会使用 FLT_OPERATION_REGISTRATION 结构数组来注册关心的 I/O 操作。数组中的每个元素代表这一个 I/O 操作的注册参数。数组一般定义如下：

```
const     FLT_OPERATION_REGISTRATION Callbacks[] = {
      {
          IRP_MJ_CREATE,      //    主功能号 IRP_MJ_Xxx
          0,                  //    Flags
          XxxPreCreate,       //    预操作回调
          XxxPostCreate       //    后操作回调
      },
    //    . . . 其它请求 . . .
{    IRP_MJ_OPERATION_END} // 结束标志一定要有
};
```

（4） FTL_CALLBACK_DATA

在 Minifilter 中，IRP 指针的概念已被淡化，除非必要功能需要自己创建 IRP 之外，其他时候不会像传统型过滤驱动框架那样直接解析 IRP 数据包。本文只有在管理文件加密标识，文件的打开和读写等操作时通过自己构建 IRP 来实现，其它时候都是使用 Minifilter 中封装好的包含与请求相关全部信息的回调数据包，即一个 FLT_CALLBACK_DATA 结构。FLT_CALLBACK_DATA 与 IRP 类似，表示一个 I/O 请求，过滤管理器使用这个结构来初始化和处理 I/O 操作。正是因为有这个参数，使得 Minifilter 中的各回调例程不必直接读取

IRP 信息，在这里，与请求相关的全部请求信息已经被封装在回调数据包中，回调数据包在各回调例程中进行传递，即可完成与 IRP 一样的信息交互工作。此外，FLT_CALLBACK_DATA 内包含许多嵌套结构，这些可以在 WDK 头文件 fltkernel.h 中找到相关定义，这个结构可以说是 Minifilter 的基础。以下是回调数据包 FTL_CALLBACK_DATA 结构定义：

```
typedef struct _FLT_CALLBACK_DATA {
  FLT_CALLBACK_DATA_FLAGS  Flags;             // 标志位掩码
  PETHREAD CONST  Thread;                     // 指向创建该 I/O 请求的线程
  PFLT_IO_PARAMETER_BLOCK CONST  Iopb;        // 结构中最关键的域，包含 I/O 请求的相关参数
  IO_STATUS_BLOCK  IoStatus;                  // 包含 I/O 操作的状态和信息
  struct _FLT_TAG_DATA_BUFFER  *TagData;      // 包含 I/O 操作的数据重解析点
    union {
      struct {
          LIST_ENTRY  QueueLinks;             // 过滤管理器用来挂起回调的队列
          PVOID  QueueContext[2];             // 额外的上下文
      };
      PVOID  FilterContext[4];
  };
 KPROCESSOR_MODE  RequestorMode;              // 最初调用者的请求模式：内核模式或用户模式
} FLT_CALLBACK_DATA, *PFLT_CALLBACK_DATA;
```

FTL_CALLBACK_DATA 中的 Iopb 域，FLT_IO_PARAMETER_BLOCK 结构定义如下：在结构定义中可以看到很多与 IRP 相类似的域，意义相同，这里不再赘述。

```
typedef struct _FLT_IO_PARAMETER_BLOCK {
  ULONG  IrpFlags;
  UCHAR  MajorFunction;
  UCHAR  MinorFunction;
  UCHAR  OperationFlags;
  UCHAR  Reserved;
  PFILE_OBJECT  TargetFileObject;
  PFLT_INSTANCE  TargetInstance;
  FLT_PARAMETERS  Parameters;
} FLT_IO_PARAMETER_BLOCK, *PFLT_IO_PARAMETER_BLOCK;
```

FTL_CALLBACK_DATA 中的 Iopb 中的 Parameters 域，该结构存放特定请求的参数，与 IRP 的 IO_STACK_LOCATION 结构类似，在此不再赘述。关于写请求的 FLT_PARAMETERS 如下所示：

```
typedef   union     _FLT_PARAMETERS{
        . . .
     struct {
          ULONG Length;
          ULONG POINTER_ALIGNMENT Key;
```

```
        LARGE_INTEGER ByteOffset;
        PVOID WriteBuffer;
        PMDL MdlAddress;
    } Write;
    . . .
} FLT_PARAMETERS, *PFLT_PARAMETERS;
```

2.4.3 Minifilter 过滤驱动的入口例程与卸载回调例程

每个文件系统微过滤驱动都必须有一个 DriverEntry 入口例程，这相当于我们平时写控制台程序时的 main 函数，作为整个驱动程序的入口。DriverEntry 在 Minifilter 驱动加载时被调用。DriverEntry 在系统线程上下文中运行，IRQL 为 PASSIVE_LEVEL。以下是 DriverEntry 例程的定义：

```
NTSTATUS   (*PDRIVER_INITIALIZE) (
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath );
```

DriverEntry 有两个输入参数：DriverObject，是在 Minifilter 驱动被加载时创建的驱动对象；RegistryPath，指向一个 Unicode 字符串，存储着 Minifilter 驱动注册表键的路径。一般情况下，DriverEntry 必须完成以下几个工作，依次如下：

1) 完成 Minifilter 驱动必要的初始化工作。
2) 调用 FltRegisterFilter()例程完成 Minifilter 驱动的注册。
3) 调用 FltStartFiltering()例程开始过滤，只有该例程返回成功，Minifilter 驱动才真正工作。
4) 返回适当的 NTSTATUS 值，即返回操作状态信息。

这里有两个重要的函数。FltRegisterFilter()，原型如下：

```
NTSTATUS  FltRegisterFilter(
  IN PDRIVER_OBJECT  Driver,
    IN CONST FLT_REGISTRATION  *Registration,
    OUT PFLT_FILTER  *RetFilter );
```

1) Driver：指向该 Minifilter 驱动的驱动对象，由 DriverEntry 传入。
2) Registration：指向由调用者分配的 Minifilter 驱动注册结构（FLT_REGISTRATION）。
3) RetFilter:指向调用者分配的 FLT_FILTER 变量，用于接收不透明的过滤器指针。

FltStartFiltering()，原型如下：

```
NTSTATUS   FltStartFiltering(
  IN PFLT_FILTER  Filter    );
```

该函数只有一个输入参数，即从 FltRegisterFilter 返回的不透明的过滤器指针。只有该函数被成功调用后，过滤驱动才真正开始工作。

2.4.4 Pre/Post Operation 回调例程

在 DriverEntry 例程中，微过滤驱动可以为需要过滤的 I/O 操作注册一个或多个预操作回调例程（Preoperation Callback Routine）和后操作回调例程（Postoperation Callback

Routine）。与传统型过滤驱动不同的是，一个 Minifilter 驱动可以选择性的过滤某些类型的 I/O 操作。值得注意的是，对给定类型的 I/O 操作，可以注册预操作例程或后操作例程。这时，Minifilter 驱动只会接收那些已注册的 I/O 操作，并且由相应的回调例程进行处理，而其它 I/O 操作不接收。

一个预操作回调例程，其实相当于传统型过滤驱动模型中的派遣函数。当过滤管理器处理一个 I/O 请求时，它会依次调用 Minifilter 驱动实例堆栈中那些为该类型请求注册了预操作回调例程的驱动。调用次序由加载 Minifilter 驱动的 Altitude 决定，Altitude 越大，表明 Minifilter 驱动加载的位置越高，其注册的预操作回调例程会最先被调用，进而最先过滤到该 I/O 请求。也就是说 Altitude 大的，加载点最高的 Minifilter 驱动会最先接收到请求，当 Minifilter 驱动处理完接收到的请求后，会将这个请求返回给过滤管理器，此时，过滤管理器又会将请求下发给堆栈中下一个最高层的 Minifilter 驱动。如此依次下发请求，直到 Minifilter 实例堆栈中所有为该类型请求注册了预操作回调例程的 Minifilter 驱动都已处理 I/O 请求后，排除中途有某个 Minifilter 驱动完成 I/O 请求的情况，过滤管理器会将请求传递给传统型驱动和文件系统。

一个后操作回调例程等同于传统型过滤驱动模型中的完成函数。在 I/O 管理器把请求传递给文件系统和已为该请求注册完成函数的传统型过滤器时，I/O 操作将进行完成处理（Completion Processing）。完成函数结束返回后，过滤驱动将为这个请求进行完成处理。此后，过滤管理器会自下而上依次调用 Minifilter 实例堆栈中已为该 I/O 请求类型注册后操作回调例程的 Minifilter 驱动。在堆栈最底层的 Minifilter 驱动，也就是 Altitude 最小的实例，将会最先接到这个 I/O 操作。在 Minifilter 驱动结束请求处理后，会将这个 I/O 操作返回给过滤管理器，再由过滤管理器将这个 I/O 操作传递给下一个最底层的 Minifilter 驱动，如此依次传递，最后将结果返回给最初发出请求的调用者。

总之，当某类型请求下发时，会顺着微过滤驱动实例堆栈，自上而下依次触发为该类型请求注册了预操作回调例程的微过滤驱动，在请求被完成后，又依次自下而上返回，依次触发为该请求注册了后操作回调例程的微过滤驱动。这样一个请求的传递过程，可以参看图 2-5。以下给出预操作回调和后操作回调的原型定义：

```
typedef FLT_PREOP_CALLBACK_STATUS
 (*PFLT_PRE_OPERATION_CALLBACK) (
   __inout PFLT_CALLBACK_DATA Data,
   __in PCFLT_RELATED_OBJECTS FltObjects,
   __deref_out_opt PVOID *CompletionContext
  );
typedef FLT_POSTOP_CALLBACK_STATUS
 (FLTAPI *PFLT_POST_OPERATION_CALLBACK) (
   __inout PFLT_CALLBACK_DATA Data,
     __in PCFLT_RELATED_OBJECTS FltObjects,
```

```
    __in_opt PVOID CompletionContext,
    __in FLT_POST_OPERATION_FLAGS Flags
);
```

1) PFLT_CALBACK_DATA Data：回调数据包指针，封装有关请求的所有信息参数。
2) PCFLT_RELATED_OBJECTS FltObjects：存储与当前请求相关对象的不透明指针，例如卷指针、实例对象等。
3) PVOID CompletionContext：用户自定义的上下文指针，由预操作回调例程返回，传入到后操作回调例程中，实现预操作回调例程和后操作回调例程之间的参数传递。
4) FL_POST_OPERATION_FLAGS Flags：标明后操作回调应该被如何执行。

值得一提的是，CompletionContext 常用来实现预操作回调例程到后操作回调例程的参数传递。由于内核代码有严格的中断级（IRQL）限制，在特定的 IRQL 下有些代码不能执行。比如说，一般情况下，在后操作的 IRQL 中，不能获取上下文结构（例如流上下文），此时可以在预操作回调中获取到所需的上下文结构，并将其封装在 CompletionContext 中，请求返回时，CompletionContext 自动被传递到对应的后操作回调例程中。

2.4.5 用户层和内核通信接口

过滤管理器支持内核模式与用户模式通过非缓冲模式的通信端口实现高效的双向通信。微过滤驱动制定一个安全描述符（Security Descriptor）来控制通信端口的安全。Minifilter 驱动创建服务端口后，就开始接收发到该端口的连接。当一个用户模式程序尝试连接服务端口时，过滤管理器将调用 Minifilter 驱动的 ConnectNotifyCallback 例程，当过滤驱动重新得到控制权时，它将一个分离文件（Separate File）句柄返回给用户模式程序。此后，调用者就可以用这个句柄与内核模式 Minifilter 驱动进行通信。只有当用户模式调用者有足够的访问权限时，一个连接才会被接收，可以通过端口的安全描述符进行描述。每个到服务端口上的连接，都会有自己的消息队列和独立的端点，可以想象成服务端口有多个连接点，每个连接点对应一个来自用户模式的连接。关闭连接的任何一端（用户端或内核端）都将会终止当前的连接。当用户模式程序关闭连接句柄时，过滤管理器会调用 Minifilter 驱动的 DisconnectNotifyCallback 例程，Minifilter 驱动就可以在该例程中关闭连接的句柄。值得一提的是，服务端口关闭将会阻止新的连接，但不会终止当前存在的连接。过滤驱动卸载时，过滤管理器会终止所有存在的连接。当内核微过滤驱动和用户模式调用者建立连接后，可以相互发送消息，而当用户模式向微过滤驱动发送消息时，将触发 MessageNotifyCallback 回调例程，在该例程中可以对收到的消息进行解析，并做相应处理。

2.4.6 上下文结构

上下文是由 Minifilter 驱动定义的一个结构体，这个结构体可以关联过滤管理器对象相。Minifilter 驱动可以为以下对象创建和设置上下文：文件（Windows Vista 和以后版本的操作系统）、实例、卷、流、流句柄（文件对象）、事务（Windows Vista 和以后版本的操作系统）。

除卷上下文必须要分配在非分页内存中以外，其它上下文可以分配在分页内存中也可以分配在非分页内存中。当上下文附加的对象被删除时，在一个 Minifilter 驱动实例从一个卷上解绑后，或当 Minifilter 驱动卸载时，相应的上下文会自动被过滤管理器删除。

2.4.7 读写方式

一个 I/O 操作的 FLT_PARAMETERS 结构，包含与该操作相关的参数，包括缓冲区地址和任何在操作中被使用的缓冲区内存描述符列表（Memory Descriptor Lists MDL）。对于基于 IRP 的 I/O 操作，操作的缓冲区可以使用以下方式指定：只用 MDL（典型的，用于分页 I/O）；只使用缓冲区地址；缓冲区地址和 MDL 同时使用。对于 Fast I/O 操作，只指定用户空间下的缓冲区。Fast I/O 操作拥有的缓冲区通常既不使用缓冲 I/O 也不是直接 I/O，因此不会有 MDL 参数。

2.5 HOOK API 技术

HOOK（钩子，挂钩）是一种实现 Windows 平台下类似于中断的机制[24]。HOOK 机制允许应用程序拦截并处理 Windows 消息或指定事件，当指定的消息发出后，HOOK 程序就可以在消息到达目标窗口之前将其捕获，从而得到对消息的控制权，进而可以对该消息进行处理或修改，加入我们所需的功能。钩子按使用范围分，可分为线程钩子和系统钩子[25]，其中，系统钩子具有相当大的功能，几乎可以实现对所有 Windows 消息的拦截、处理和监控。这项技术涉及到两个重要的 API，一个是 SetWindowsHookEx，安装钩子；另一个是 UnHookWindowsHookEx，卸载钩子。

本文使用的 HOOK API 技术[26]，是指截获系统或进程对某个 API 函数的调用，使得 API 的执行流程转向我们指定的代码段，从而实现我们所需的功能。Windows 下的每个进程均拥有自己的地址空间，并且进程只能调用其地址空间内的函数，因此 HOOK API 尤为关键的一步是，设法将自己的代码段注入到目标进程中，才能进一步实现对该进程调用的 API 进行拦截。然而微软并没有提供 HOOK API 的调用接口，这就需要开发者自己编程实现，大家所熟知的防毒软件、防火墙软件等均采用 HOOK API 实现。

一般来说，HOOK API 由两个组成部分，即实现 HOOK API 的 DLL 文件，和启动注入的主调程序。本文采用 HOOK API 技术对剪切板相关的 API 函数进行拦截，从而实现对剪切板内容的监控功能，同样使用该技术实现进程防终止功能。其中 DLL 文件支持 HOOK API 的实现，而主调客户端程序将在初始化时把带有 HOOK API 功能的 DLL 随着鼠标钩子的加载注入到目标进程中，这里的鼠标钩子属于系统钩子。

2.6 密码学相关知识概述

密码学（Cryptology）[27]是结合数学、计算机科学、电子与通信等诸多学科于一体的交叉学科，是研究信息系统安全保密的一门科学。一个密码系统或一个密码体制是指明文（Plaintext, p）、密文（Ciphertext, c）、密钥（Key, k）、加密算法（E）和解密算法（D）的

五元组，其中，加密过程可表示为：$c = E_k(p)$；解密过程可表示为：$p = D_k(c)$。

（1）分组密码基础

密码体制是指实现加密和解密功能的密码方案，根据密钥的使用策略可分为对称密码体制（Symmetric Key Crytosystem）和非对称密码体制（Asymmetric Key Cryptosystem，也称为公钥密码体制）。其中对称密码算法主要用于保证数据的机密性，使用相当广泛[28]，它包括分组密码和序列密码。本文在加密模块所使用的 AES 加密算法属于对称密码体制中分组密码的范畴，其它典型的分组密码还有 DES、3DES、IDEA、RC6 等。

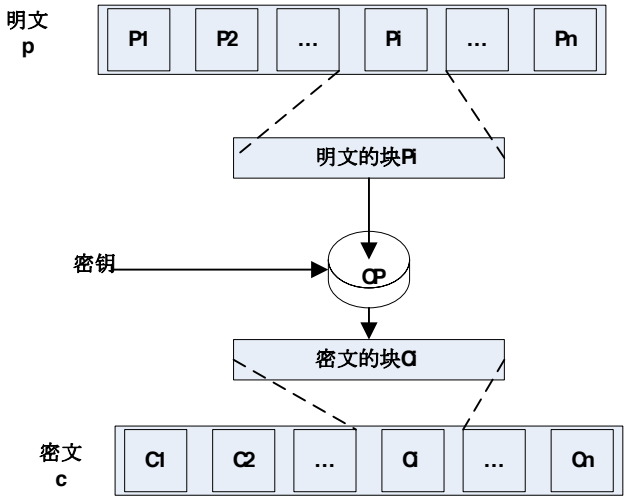


图2-6 分组加密常见加密结构

分组密码（Block Cipher）是现代密码学的重要算法之一，具有加密速度快、安全性好、易于标准化等特点，广泛的应用于数据的保密传输、加密存储等场合。顾名思义，分组密码是将明文消息的二进制序列划分成若干定长的块，每块分别在密钥的控制下，采用相同的加密算法，变换成等长的二进制块。如图 2-6 所示，将明文的二进制序列划分成 P1,…,Pn 的 n 个明文分组，然后对每个明文分组使用相同的加密算法 OP，可以得到 C1,…,Cn 的 n 个密文分组，这样就实现了对数据的加密过程。分组密码的解密过程与加密过程类似，首先将密文分成 C1,…,Cn 的 n 个小组，然后对每个密文分组执行解密算法，从而恢复出对应的 P1,…,Pn 的 n 个明文分组。分组大小由加密变换的输入长度确定，为了抵抗穷举明文攻击，通常分组长度较长（128 或 64 的倍数位）。

本文的重心在于研究并实现一个防终止的基于 Windows 透明加密过滤系统。因此本文并没有过多考虑对加密算法的设计与改进。只是对典型的分组加密算法进行了对比分析，进而选择适当的加密算法。DES 的主要缺陷有：互补性，DES 的互补性使得 DES 在选择明文攻击下需要的工作量减半；弱秘钥性；迭代轮数，对于 16 轮的 DES 的已知明文攻击、差分分析攻击比穷举攻击有效；密钥太短。针对 DES 的缺陷，人们提出了一种简单的方案就是使用多重 DES，在此之下就产生了 3DES 加密算法。但由于其是 DES 的改进版，3DES 也具有先天不足：3DES 的处理速度较慢，尤其是软件实现；虽然增加了密钥的长度，但是

明文分组的长度并没有变化，仍是 64bit，因此就效率和安全性而言，与密钥的增长不相匹配。在这样的背景下，2001 年 11 月 26 日，NIST 正式公布高级加密标准（Advanced Encryption Standard，AES），并于 2002 年 5 月 26 日正式生效。AES 分组长度只能是 128bit，密钥长度可以使用 128bit、192bit、256bit 三者中的任意一种，如表 2-1 所示：

表2-1　AES 的密钥长度和加密轮数关系表

	密钥长度（32 比特字）	分组长度（32 比特字）	加密轮数
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

本文选择 AES-256，即表中最后一行的情况。AES 的处理单位是字节，明文分组 P 为 128bit，密钥 K 分成 32 比特字。一般的，明文分组用被称为状态矩阵的以字节为单位的方阵来描述。在算法的每轮迭代中，状态矩阵的内容会不断变化，最后作为密文输出。

（2）Hash 函数与 Hash 算法：

Hash 函数是从一个消息空间到像空间的不可逆映射，也被称为杂凑函数、哈希函数、散列函数等[Error! Bookmark not defined.]。Hash 函数可以把“任意长度”的输入经过变换得到定长的输出。因此，Hash 函数是一种具有压缩特性的单向函数，它的像通常称为散列值（Hash Value）、消息摘要（Message Digest）或数字指纹（Digital Fingerprint）。散列值可表示为：h = H（M），其中 h 是定长的消息摘要，H 是 Hash 函数，M 是一个长度任意的消息。目前，典型的 Hash 函数算法有：MD5、SHA-1、SHA-256 等。

NIST 在 FIPS 180-1 的基础上发布了 FIPS 180-2，这个标准中除 SHA1 之外还新增加了 SHA256、SHA384 和 SHA512 三个摘要算法标准。为了便于与 AES 的使用相匹，这些算法的消息摘要长度分别是 160bit、256bit、384bit、512bit。这 4 种摘要算法的相关属性区别如下表所示：

表2-2　SHA 相关属性比较

	SHA1	SHA256	SHA384	SHA512
消息摘要长度	160	256	384	512
消息长度	$<2^{64}$	$<2^{64}$	$<2^{128}$	$<2^{128}$
分组长度	512	512	1024	1024
字长度	32	32	64	64
步数	80	64	80	80

本文选择使用 SHA-256 来计算加密密钥以及用户认证口令的摘要值。

2.7 本章小结

本章主要介绍本文所实现的透明加密过滤系统的相关理论基础和关键技术。首先介绍了 Windows 操作系统的核心框架，其主要分为用户模式和内核模式，用户模式受控于操作系统，拥有安全的运行环境；而内核模式处于系统底层，不受操作系统保护，拥有最高的特权级别，几乎可以实现任何功能，其中内核模式主要由执行体组件层、内核层、内核驱动层和硬件抽象层构成，相互之间通过定义好的接口进行消息传递。还介绍了驱动开发的相关知识，其中介绍了驱动开发中所涉及的重要数据结构，主要有驱动对象、设备对象、输入输入请求包（IRP）以及 IO_STACK_LOCATION；较为全面的介绍了驱动与设备的层次架构，以及请求处理过程。再者介绍了文件系统驱动与文件系统过滤驱动的区别，以及相互之间的关系。重点介绍了 Minifilter 框架，为本文内核层的透明加密过滤驱动的实现提供了很好的理论基础和技术支持，主要内容有：过滤管理器模型、关键数据结构、入口例程、预操作回调例程和后操作回调例程、用户层与内核层通信接口、上下文结构、读写方式。对 HOOK API 的技术原理进行了介绍，为实现剪切板监控模块与进程防终止模块提供了技术支持。最后对密码学相关知识进行了介绍，包括分组密码基础知识与摘要算法的相关内容，为本系统的加密模块选取了合适的算法。

第三章 透明加密过滤系统的总体设计

3.1 系统设计目标

本文实现防终止的基于 Windows 透明加密过滤系统，较为全面的提供一个保护机密文档的解决方案。此外本系统还考虑了系统的兼容性、稳定性，以及全面的自我保护方案，在一定程度上降低了被恶意关闭的危险。本系统设计的具体目标有如下几点：

1) 采用微软新提供的 Minifilter 驱动模型实现透明加密过滤驱动。利用 Minifilter 自身的优点（参见第一章）可以保证本系统具有很好的稳定性、兼容性，并且保证本系统的加载不会影响原有其他系统或驱动的工作。
2) 透明性好。内核加解密模块被加载在文件系统之上，监控机密文档的使用情况，写机密文档时进行加密，使得机密文档在磁盘上以密文存储。当授权用户需要阅读文档时，机密文档会被自动解密，以明文的形式存储在内存中。当用户不在使用文档后，会对内存缓存进行清除，即使用户将机密文档拷贝带走，也无法获取到文档的明文，从而对机密文档起到很好的保护作用。处理的全过程用户是感受不到的，也无需额外的操作，使得过程变得简单、安全。
3) 监控剪切板，进一步提升系统安全性。通过挂钩剪切板相关的 API，从而实现对剪切板内容的监控，可以防止用户通过剪切板复制粘贴的方式，将机密文档复制或剪切到非机密文档中带着。
4) 进程防终止。采用 HOOK API 技术对应用层控制程序进行保护，降低进程被恶意关闭的风险。
5) 制定符合实际且合理可行的加解密策略，使得文档保护过程更为流畅、易用、完整。
6) 保护粒度细化到文档，在一定程度上提高加解密的灵活性。
7) 实现 C/S 架构。服务器和客户端通过 Socket 进行通信，服务器端对客户端行为操作审计、对可疑行为进行报警，并且对密钥进行管理与分发。此外，服务器采用心跳机制来定时检查客户端用户层的在线情况。客户端应用层控制程序与内核驱动之间基于 Minifilter 通信端口机制实现双向通信，作为服务器端和内核驱动的桥梁。
8) 提供身份认证机制。考虑到安全性，将客户端所使用的文件加解密密钥统一存储在服务器端。并且实现基于内网 IP+口令的认证模式。

3.2 系统总体架构设计

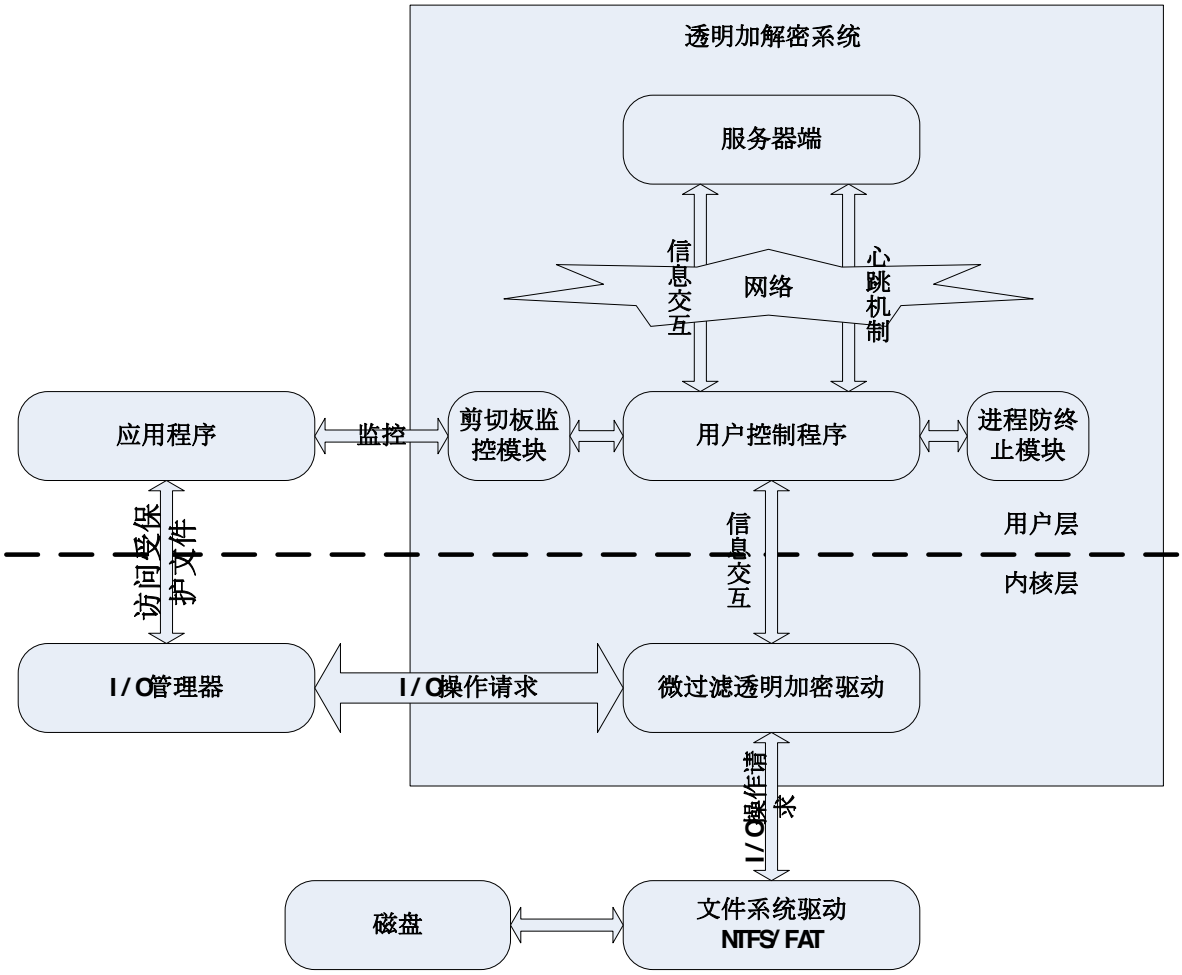


图3-1　系统总体框架结构

如图 3-1 所示，本文实现的透明加密过滤系统主要由三部分组成：服务器、客户端用户层控制程序和客户端内核层透明加密驱动程序。

1) 服务器：主要实现数据的存储、策略的制定与下发、密钥的管理与分发等功能。
2) 客户端：用户控制程序构成服务器与内核驱动之间的一个桥梁，主要辅助数据的传递，此外用户层控制程序还负责加载剪切板控制模块 DLL 和进程放终止模块的 DLL，即将实现 HOOK API 功能的 DLL 注入到目标进程中，以达到剪切板监控与进程防终止的目的，进一步提升系统的安全性。
3) 内核层透明加密微过滤驱动程序：是整个系统的核心部分，加载在文件系统驱动之上，拦截发往文件系统驱动的请求，以实现对机密进程的监控，对受保护文档进行自动加解密的处理。可以进行自我保护。

本文所实现的透明加密过滤系统主要由以上三个部分，为机密文档的保护提供一个较为完备的解决方案。下面将对系统各模块作进一步介绍。

3.3 系统模块划分

本透明加密过滤系统有如下几个模块，各模块主要完成的工作如图 3-2 中所示。具体实现过程将在下一章节进行详细的描述。

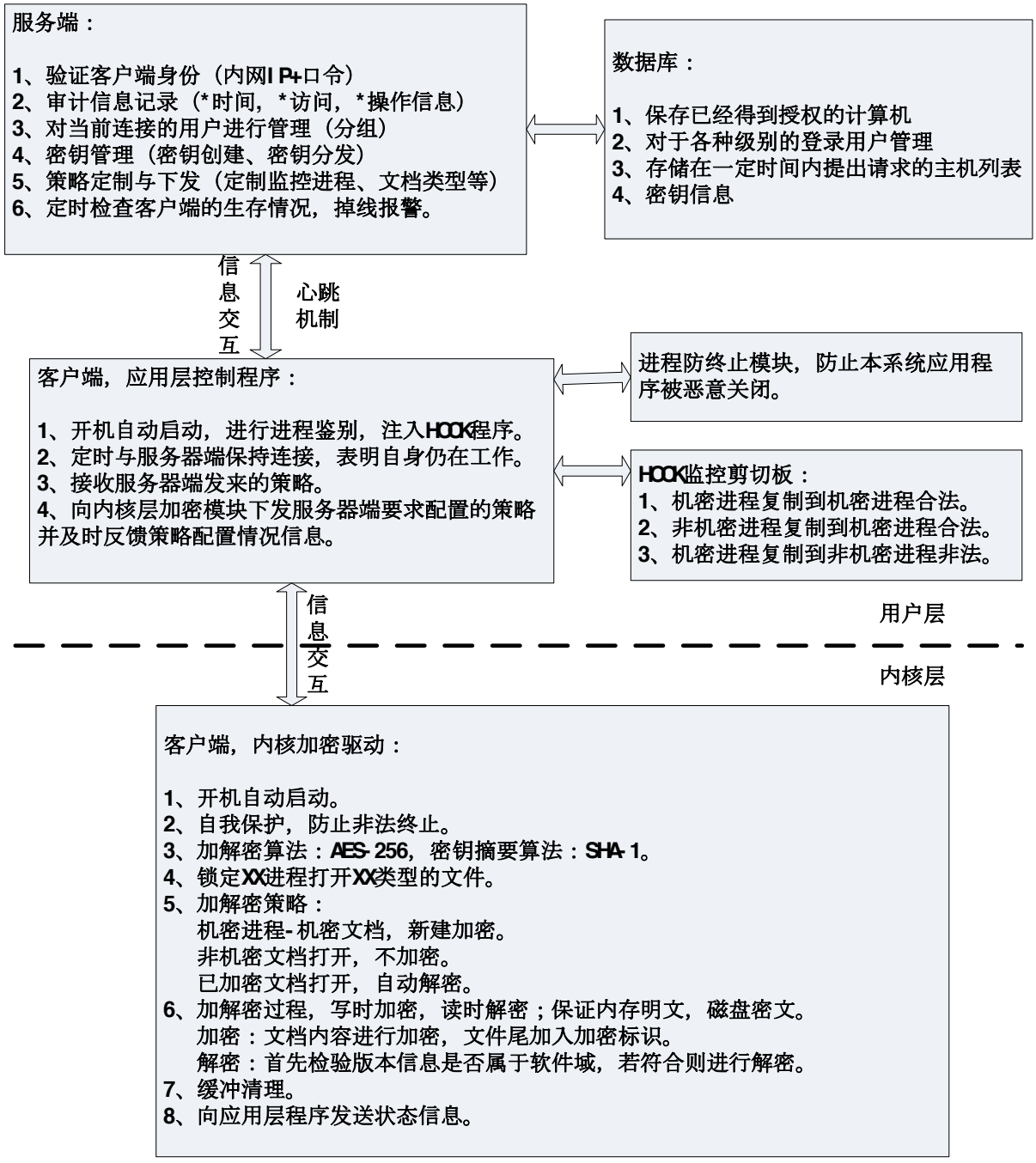


图3-2　系统模块划分图

3.4 系统开发环境及主要工具

表3-1　　系统开发主要工具

软件	作用
Visual Studio 2010	开发平台

WinDDK7600.16385.1	驱动开发包
Windebug	调试工具
VMware Workstation11	双机调试环境、测试环境
SRVINSTW.EXE	安装传统型驱动，Minifilter 使用 INF 文件进行安装。卸载指定服务。
DebugView.exe	调试工具，查看内核输出

表 3-1 为本文主要使用的软件。其中最重要的是 WinDDK，为驱动开发提供支持，带有相应操作系统的编译工具。Visual Studio 2010 提供了很好的编码环境，使得编码过程更为高效，但不提倡使用 Visual Studio 进行内核程序的编译。对驱动代码的编译，最好使用 WinDDK 配有的编译工具，其带有如 OACR 这样的检查工具，能检查出内核程序特有的错误或警告。开发内核程序时，只要出现错误将会直接导致系统崩溃，因此在主机中进行测试是不明智的，本文使用 Windebug 和虚拟机搭建了双机调试环境，给系统的调试工作提供很好的支持，即使虚拟机崩溃也不会殃及主机。另两个小工具，SRVINSTW.EXE，加载传统型过滤驱动，而微过滤驱动则通过 INF 文件进行安装，不过已安装服务可以使用该工具进行卸载；DebugView.exe 调试工具，查看内核输出。

3.5 本章小结

本章从总体上提出文档保护的解决方案。根据需求定制本系统的设计目标。并且从总体上给出本系统的设计框架，主要包括服务器端、客户端用户层控制程序和内核层透明加密过滤驱动程序三个模块。在此基础上对系统的各模块进行划分，并给出各模块需要实现的功能。

第四章 系统详细设计与实现

本章将对本系统所实现的关键模块进行详细的介绍，给出详细设计步骤、流程图。需要说明的是，本章主要介绍系统核心模块的具体实现过程，对于相关技术以及数据结构不再做详细说明，对于这一部分内容可以参看前文相关章节。

4.1 透明加密微过滤驱动注册与加载

4.1.1 重要数据结构定义

（1） 加密标识结构定义：

加密标识[29]用于标识受保护文档，其中存放着与文档相关的必要参数，在受保护文档关闭时被写入文件尾。并且本文采用了适当的数据隐藏手段，对该加密标识进行隐藏处理，使其对用户层应用程序来说是不可见的。

```
/*文件标识结构，每当文件被关闭时被写入文件尾。*/
typedef struct _FILE_TAIL{
    /*    使用 GUID 来区分加密文件和非加密文件。所有加密文件共用同一个 GUID。*/
    UCHAR FileGUID[FILE_GUID_LENGTH] ;
    /*   真实大小，即文本内容真实长度，不包括文件加密标识和填充长度。*/
    LONGLONG FileValidLength ;
    /* 文件加密密钥的摘要。*/
    UCHAR FileKeyDigest[HASH_SIZE] ;
    /* 保留字段，以便进一步使用或与扇区大小对准，其中 8 是 LONGLONG 的字节大小。*/
    UCHAR Reserved[SECTOR_DEFAULT_SIZE-HASH_SIZE-FILE_GUID_LENGTH-8] ;
}FILE_TAIL,*PFILE_TAIL;
```

（2） 卷上下文结构定义：

在卷挂载时将本文定义的卷上下文挂载带卷上，来记录必要的参数。必要时，可在操作回调例程中读取。

```
/* 卷上下文结构，被绑定到系统监视的每个卷上,用来存储与卷相关的参数。*/
typedef struct _VOLUME_CONTEXT {
    UNICODE_STRING usPathName;          // 用于显示的 DOS 名
    UNICODE_STRING usFileSystemName;    // 挂载卷的文件系统名称
    ULONG ulFileSystemType;             // 文件系统类型
    ULONG ulSectorSize;                 // 扇区大小
    BOOLEAN bIsSetKey;                  // 是否设置文件密钥
    UCHAR ucFileKey[MAX_KEY_LENGTH];// 文件密钥。该卷上存储的文件默认的加解密密钥
    /*文件密钥摘要。用来匹配文件尾的密钥摘要，以确定是否可直接使用当前默认密钥。*/
    UCHAR ucFileKeyDigest[HASH_SIZE] ;
    KSPIN_LOCK CryptSpinLock ;          // 用于同步加解密过程的自旋锁
} VOLUME_CONTEXT, *PVOLUME_CONTEXT;
```

（3） 流上下文结构定义：

流上下文与文件对应，用于记录与文件相关的参数。

```
typedef struct _STREAM_CONTEXT {
    UNICODE_STRING usFileName;                          // 当前打开的文件名
    WCHAR wcVolumeName[VOLUME_NAME_LENGTH]; // 文件从属卷的卷名
    BOOLEAN bIsSetKey;                                  //文件密钥是否设置
    /* 文件密钥，文件被打开时，根据存储在文件尾的密钥摘要在当前密钥链中匹配查找获取*/
    UCHAR ucFileKey[MAX_KEY_LENGTH] ;
    UCHAR ucFileKeyDigest[HASH_SIZE] ;                  // 文件密钥的摘要
    /*文件流被引用次数的计数器。当计数值为 0 时，表示所有文件对象都已被关闭，
      可以在文件尾写入文件标识，以及清除缓冲区。*/
    LONG RefCount;
    LARGE_INTEGER FileValidLength;  // 文件有效大小，不包括文件标识和填充的长度
    /*  文件实际存储大小，包括文件真实大小，填充长度，和文件标识长度 。*/
    LARGE_INTEGER FileSize ;
    ULONG uTrailLength ;            // 在文件尾加入的文件加密标识长度
    ULONG uAccess ;                 // 文件的访问权限
    /* 标志位定义 */
    BOOLEAN bIsFileCrypt ;
    BOOLEAN bEncryptOnWrite ;
    BOOLEAN bDecryptOnRead ;
    /* 用于保护这个上下文的锁( 锁定正在读写的 Stream Context ) */
    PERESOURCE Resource;
    /* 自旋锁，被用于保护这个上下文，当 IRQL 太高的时候使用。*/
    KSPIN_LOCK Resource1 ;
} STREAM_CONTEXT, *PSTREAM_CONTEXT;
```

（4） 本文的预操作回调例程到后操作回调例程的上下文结构定义：

有些内核对象不能再 DPC 中断级中被获取，但在 DPC 中断级中可安全释放。因此通常会在预操作回调中取得这些对象，再传给后操作回调例程，并由后操作回调例程完成释放工作。这个过程可通过操作回调例程中的 Completion 参数来实现。

```
typedef struct _COMPLETION_CONTEXT {
    PVOLUME_CONTEXT pVolumeContext;  // 卷上下文结构指针，记录挂载卷的相关信息
    PSTREAM_CONTEXT pStreamContext ;  // 流上下文结构指针，记录文件的相关信息
    /*
      后操作回调中接收到的参数，都是原始的参数（即从下层返回的参数，不带有我们设置的
      参数）。为此，我们必须将新设置的目标缓冲，通过该结构来封装，传入到后操作回调中
      以便于在后操作回调中完成对缓冲区进行的释放。
    */
    PVOID SwappedBuffer;
} COMPLETION_CONTEXT, *PCOMPLETION_CONTEXT;
```

4.1.2 填写 minifilter 注册结构

填写 FLT_OPERATION_REGISTRATION 结构数组，注册本系统需要过滤的 I/O 操作：

```
CONST FLT_OPERATION_REGISTRATION FltOperationRegistrationArray[] = {
    {   IRP_MJ_CREATE,                                          //    主功能号
        FLTFL_OPERATION_REGISTRATION_SKIP_PAGING_IO,            //    Flags
        EncrytingFilter_PreCreate,                              //    预操作回调
        EncrytingFilter_PostCreate                              //    后操作回调
    },
    { IRP_MJ_READ,
        0,
      EncrytingFilter_PreRead,
      EncrytingFilter_PostRead
    },
    {   IRP_MJ_WRITE,
        0,
        EncrytingFilter_PreWrite,
        EncrytingFilter_PostWrite
    },
    { IRP_MJ_CLOSE,
        0,
        EncrytingFilter_PreClose,
        NULL
    },
    { IRP_MJ_CLEANUP,
        FLTFL_OPERATION_REGISTRATION_SKIP_PAGING_IO,
        EncrytingFilter_PreCleanup,
        NULL
    },
    {   IRP_MJ_QUERY_INFORMATION,
        0,
        EncrytingFilter_PreQueryInfo,
        EncrytingFilter_PostQueryInfo
    },
    { IRP_MJ_SET_INFORMATION,
        0,
     EncrytingFilter_PreSetInfo,
     EncrytingFilter_PostSetInfo
    },
    { IRP_MJ_OPERATION_END }        // 结构结束标志
};
```

填写 FLT_CONTEXT_REGISTRATION 结构数组，注册上下文：

```
CONST FLT_CONTEXT_REGISTRATION FltContextRegistrationArray[] = {
```

```
    { FLT_VOLUME_CONTEXT,                   //    上下文类型
        0,                                  //    Flags
      EncrytingFilter_CleanupContext,       //    上下文清理回调例程
      VOLUME_CONTEXT_SIZE,                  //    上下文大小
      VOLUME_CONTEXT_TAG                    //    PoolTag
    },
    { FLT_STREAM_CONTEXT,
        0,
      EncrytingFilter_CleanupContext,
      STREAM_CONTEXT_SIZE,
      STREAM_CONTEXT_TAG
    },
    { FLT_CONTEXT_END }                     //   结束标记
};
```

FLT_REGISTRATION 注册结构填写：

```
CONST FLT_REGISTRATION FltRegistration = {

    sizeof( FLT_REGISTRATION ),             //   结构大小
    FLT_REGISTRATION_VERSION,               //   结构版本
    0,                                      //   微过滤器标志位
    FltContextRegistrationArray,            //   上下文结构数组
    FltOperationRegistrationArray,          //   操作回调例程数组
    EncrytingFilter_Unload,                 //   卸载回调例程
    EncrytingFilter_InstanceSetup,          //   实例加载回调例程
    NULL,                                   //   InstanceQueryTeardown
    NULL,                                   //   InstanceTeardownStart
    NULL,                                   //   InstanceTeardownComplete
    NULL,                                   //   GenerateFileName
    NULL,                                   //   GenerateDestinationFileName
    NULL                                    //   NormalizeNameComponent
};
```

4.1.3 DriverEntry 例程

```
/*++       过滤驱动入口函数。完成全局变量初始化，FltMgt 注册，启动过滤。        --*/
NTSTATUS
DriverEntry (__in PDRIVER_OBJECT DriverObject,__in PUNICODE_STRING RegistryPath )
{
    NTSTATUS status;
    ULONG processStatus;
    ReadConfigFile(RegistryPath);
    Key_InitKeyList(NULL);
    /* 初始化后备链表，用于开辟 COMPLETION_CONTEXT 内存空间 */
```

```
    ExInitializeNPagedLookasideList( &CompletionContextLookasideList, NULL, NULL, 0,
sizeof(COMPLETION_CONTEXT), COMPLETION_CONTEXT_TAG, 0 );
    g_ulProcessNameOffset = Ps_GetProcessNameOffset();    // 获取进程名 FCB 中的偏移位置
    InitializeListHead(&g_ProcessListHead);               // 初始化进程链表入口
    KeInitializeSpinLock(&g_ProcessListLock) ;            // 初始化自旋锁同步进程链表操作
    status = File_InitFileTail() ;                        // 初始化全局文件标识结构
    if (!NT_SUCCESS(status)){  return status ;}

    /*   启动默认监控状态   */
    processStatus = Ps_AddDefualtProcessInfo(PROCESS_NAME_SYSTEM,TRUE);
    if(processStatus!= MGDPI_RESULT_SUCCESS){……}

    /* 向 FltMgr 注册本过滤器   */
    status = FltRegisterFilter(DriverObject, &FltRegistration, &gFilterHandle);
    if (NT_SUCCESS(status))
    {    /* 为用户应用程序与 minifilter 驱动之间的通信创建服务端口 */
        status = Msg_CreateCommunicationPort(gFilterHandle) ;
        if (NT_SUCCESS(status))
        {
            status = FltStartFiltering(gFilterHandle);      // 开始过滤 I/O 请求。
            if (!NT_SUCCESS(status)) {
                FltUnregisterFilter(gFilterHandle);
            }
            else{ … … }
        }
    }
    return status;
}
```

4.2 Create 回调例程

本文为 Create 操作注册了预操作回调例程和后操作回调例程，对主功能号为 IRP_MJ_CREATE 的 IRP 请求进行拦截。预操作回调用于拦截下发的 IRP，即 Create 预操作回调拦截到 IRP 时，Create 操作还未得到完全处理；后操作回调用于拦截回发的 IRP，即拦截 Create 请求被下层驱动完后返回时的 IRP。当有新的目录或文件被创建时，当已经存在的卷、目录、设备或目录被打开时，I/O 管理器会发出 IRP_MJ_CREATE 请求。因此，我们需要在 Create 请求的预操作回调中对被打开的对象进行识别，以确定是否需要进一步处理，若为机密文档，则在 Create 请求的后操作回调中对保护文档的对象进行初始化，以便与该文档相关的其它请求使用。以下给出 Create 请求处理流程，以及关键代码实现。

1) Create 预操作回调例程实现：

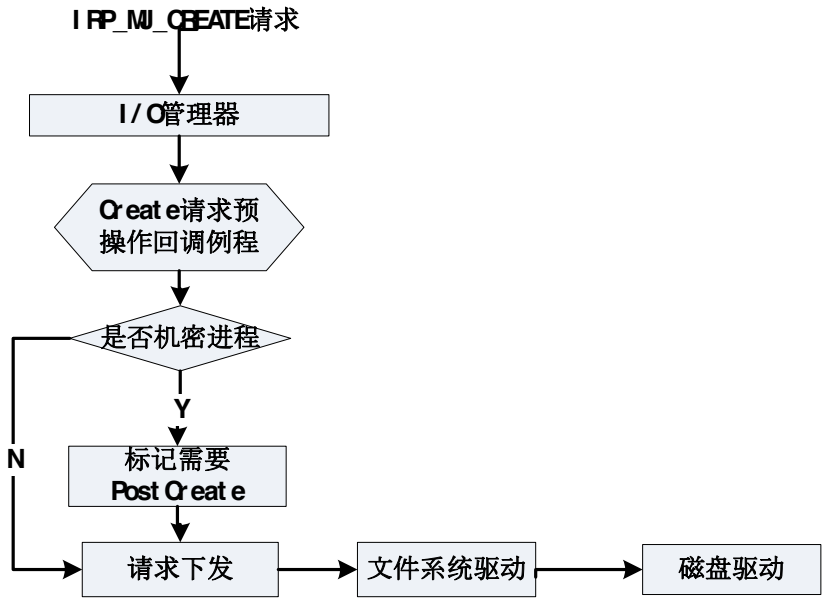


图4-1　Create 请求预操作处理流程

2)　Create 后操作回调例程实现：

Create 请求的后操作回调例程详细流程如图 4-2 所示，若打开失败，什么都不做。否则，首先获取文件名信息，再判断文件是否为监控的文件。若是，则尝试获取该文件的流上下文，不存在则创建新的流上下文，并根据读到的文件标识对文件进行初始化。

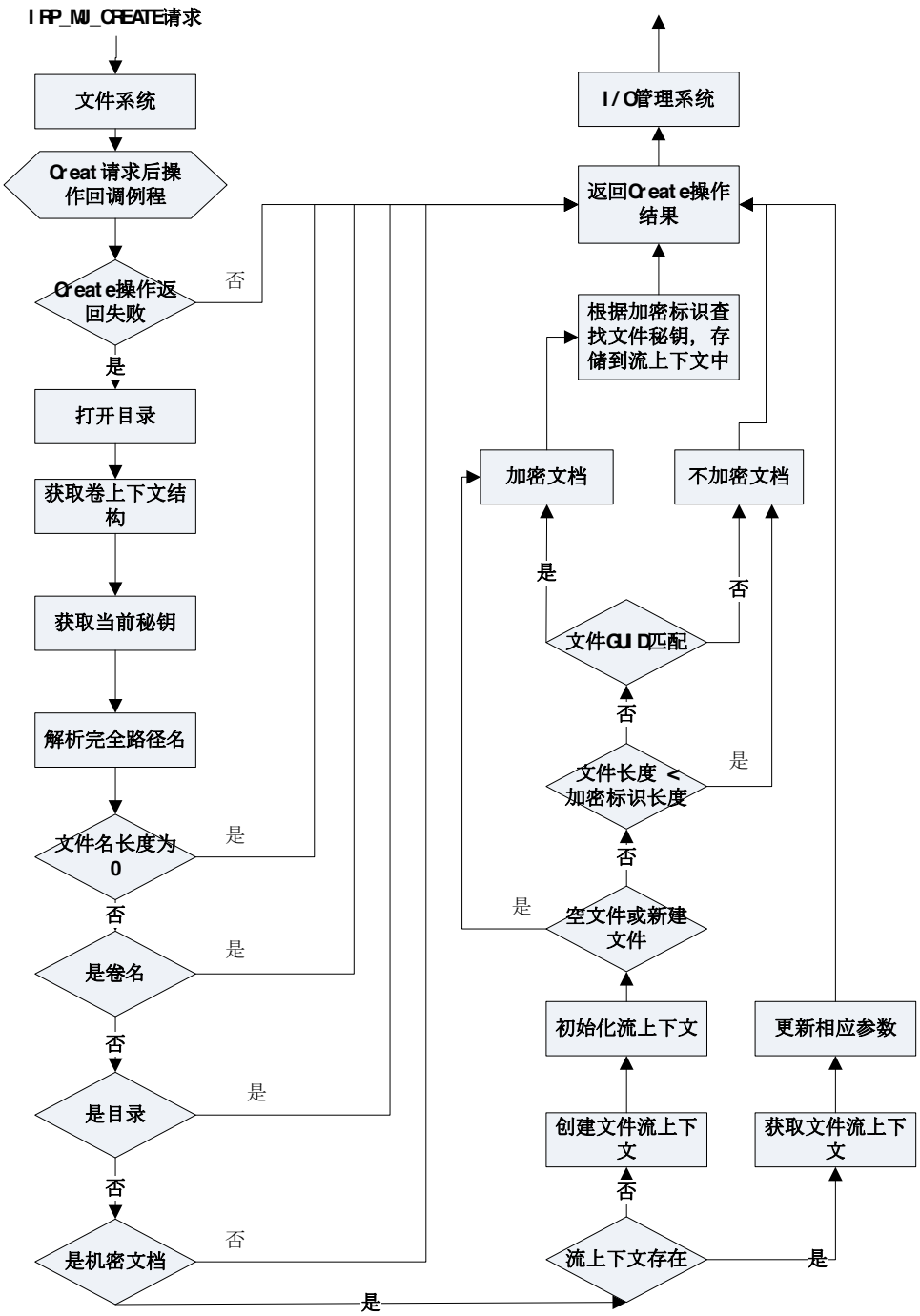


图4-2　Create 请求后操作处理流程

关键代码：

```
FLT_POSTOP_CALLBACK_STATUS
EncrytingFilter_PostCreate (
    __inout PFLT_CALLBACK_DATA Data,    __in PCFLT_RELATED_OBJECTS FltObjects,
    __inout_opt PVOID CompletionContext,    __in FLT_POST_OPERATION_FLAGS Flags)
{
    ......
```

```
    __try{
        if (!NT_SUCCESS( Data->IoStatus.Status )){__leave;}      // 如果 Create 失败，什么都不做
        status = FltGetVolumeContext( , , ) ;                    // 获取卷上下文

        /* 获取文件全路径( 例如\Device\HarddiskVolumeX\file\aa.txt */
        status = FltGetFileNameInformation( , , ) ;
        /* 解析 FLT_FILE_NAME_INFORMATION 结构 */
        status = FltParseFileNameInformation(pfNameInfo);

        if(文件名长度为 0 || 是卷名 || 是一个目录){   __leave;   }

        /* 查看当前进程是否被监控 */
        bReturn = Process_IsMonitoredProcess(pFileNameInfo->Name.Buffer,
                                             pFileNameInfo->Name.Length/sizeof(WCHAR),
                                             &bIsSystemProcess, NULL);
        if (!bReturn){ __leave;}         // 不是受监控进程，退出

        /* 创建或获取这个文件的流上下文 */
        status = Contex_FindOrCreateStreamContext(Data,FltObjects,TRUE,
                                             &pStreamContext,&bIsNewCreated);
        if (!bIsNewCreated){
            …… 更新流上下文参数，做特殊处理 ……;
            __leave;
        }

        /*  初始化新建的流上下文  */
        SC_LOCK(pStreamContext, &OldIrql) ;
        RtlCopyMemory(pStreamContext->wcVolumeName, pFileNameInfo->Volume.Buffer,
pFileNameInfo->Volume.Length) ;
        pStreamContext->bIsFileCrypt = FALSE ;
        pStreamContext->bDecryptOnRead = FALSE ;
        pStreamContext->bEncryptOnWrite= TRUE ;    //写操作的过程中加密
        pStreamContext->bHasWriteData = FALSE ;
        pStreamContext->bHasPPTWriteData = FALSE ;
        pStreamContext->RefCount = 1 ;
        pStreamContext->uAccess = ulDesiredAccess ;
        pStreamContext->uTrailLength = FILE_TAIL_SIZE ;
        pStreamContext->bIsSetKey = FALSE;           // 默认没有特定的秘钥
        SC_UNLOCK(pStreamContext, OldIrql) ;

        /*  读文件标识进行比较 */
        pFileTail = (PFILE_TAIL)ExAllocatePoolWithTag(NonPagedPool, FILE_TAIL_SIZE,
FILE_TAIL_POOLTAG) ;
```

```
        status = File_ReadFileTail(Data,FltObjects,pFileTail);

        /* 将文件标识数据中的 GUID 和全局文件标识中的 GUID 相比较 */
        if (FILE_GUID_LENGTH != RtlCompareMemory(g_psFileTail, pFileTail,
FILE_GUID_LENGTH)){   //不等，因此当前文件未被加密
          …… 修改一些参数 ……;
          __leave;
         }

        /* 文件已经被加密，重置一些域 */
        SC_LOCK(pStreamContext, &OldIrql) ;
        pStreamContext->FileValidLength.QuadPart = pFileTail->FileValidLength;
        pStreamContext->bIsFileCrypt = TRUE ;
        pStreamContext->bEncryptOnWrite= TRUE ;
        pStreamContext->bDecryptOnRead = TRUE ;
        pStreamContext->uTrailLength = FILE_TAIL_SIZE ;
        /* 获取文件加密/解密的密钥 */
                . . . . . .

    }__finally{
        . . . 清理工作 . . .
    }
    return FLT_POSTOP_FINISHED_PROCESSING;
}
```

4.3 Read 回调例程

IRP_MJ_READ 请求由 I/O 管理器或文件系统驱动发送。当用户模式应用程序调用了 Win32 函数，例如 ReadFile；或者当内核模式组件调用了类似 ZwReadFile 的文件读操作函数，那么主功能号为 IRP_MJ_READ 的请求就会被发出。因此，我们为 Read 请求注册预操作回调例程和后操作回调例程，以实现对文件读请求的拦截。在 Read 预操作回调例程中，获取文件相关上下文结构，并为文件读请求分配一个交换内存，最后将请求下发。在 Read 后操作回调例程中，对读到的机密文档进行解密，然后将明文以及操作状态返回给用户。使得用户读取机密文档的过程透明化。此外，值得重申的一点是：在 DPC 中断级中取上下文是不安全的，所以通常会在预操作回调中取得上下文，然后使用回调例程的 CompletionContext 参数进行封装，将这些参数传递到后操作回调中，并在 DPC 中断级的后操作回调中释放，在 DPC 中断级中释放上下文是安全的。以下是 Read 请求的预操作回调例程和后操作回调例程的详细设计以及关键实现。

1) Read 请求的预操作回调例程：

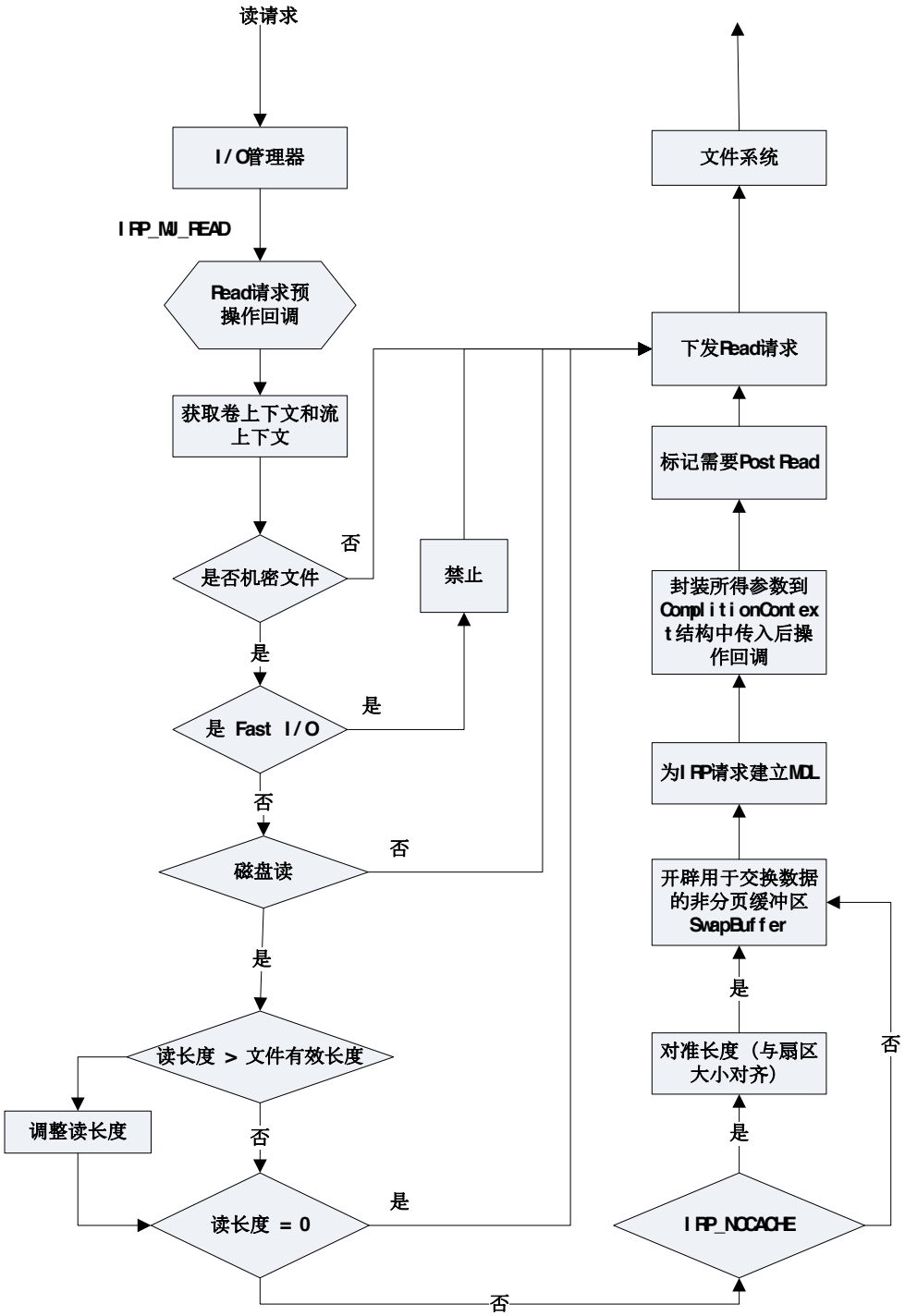


图4-3　Read 请求预操作处理流程

关键源码：

```
FLT_PREOP_CALLBACK_STATUS
EncrytingFilter_PreRead(
    __inout PFLT_CALLBACK_DATA Data,
    __in PCFLT_RELATED_OBJECTS FltObjects,
    __deref_out_opt PVOID *CompletionContext )
```

```
{
    NTSTATUS status;
    FLT_PREOP_CALLBACK_STATUS FltStatus = FLT_PREOP_SUCCESS_NO_CALLBACK;
    . . .  其它变量定义  . . .
    __try {
        /* 获取卷上下文 */
        status = FltGetVolumeContext( FltObjects->Filter, FltObjects->Volume, &pVolumeContext );
        /* 获取每个流上下文，传递给后操作 */
        status = Contex_FindOrCreateStreamContext(Data,FltObjects,FALSE,&pStreamContext,NULL) ;
        /* 若进程没有被监控，pass */
        bReturn =Process_IsMonitoredProcess
(pStreamContext->usFileName.Buffer,pStreamContext->usFileName.Length/sizeof(WCHAR), NULL,
NULL) ;
        if (!bReturn){__leave;}

        /* 快速 I/O 路径，不批准，这将会导致一个等价的 IRP 请求到来 */
        if (FLT_IS_FASTIO_OPERATION(Data))
        { FltStatus = FLT_PREOP_DISALLOW_FASTIO ; __leave; }

        /* 高速缓冲存储器存储的 I/O IRP 路径 */
        if (!(Data->Iopb->IrpFlags & (IRP_NOCACHE | IRP_PAGING_IO |
IRP_SYNCHRONOUS_PAGING_IO)))
        {    __leave; }

        /* 如果读偏移超过了文件真实大小，返回 EOF 并完成 IRP */
        if ( liReadOffset.QuadPart >= pStreamContext->FileValidLength.QuadPart)
        {
            Data->IoStatus.Status =   STATUS_END_OF_FILE;
            Data->IoStatus.Information = 0 ;
            FltStatus = FLT_PREOP_COMPLETE ;
            __leave;
        }
        . . .  . . .  . . .

        /*为我们用于交换的 buffer 开辟非分页内存*/
        newBuffer = ExAllocatePoolWithTag( NonPagedPool, ulReadLen,BUFFER_SWAP_TAG );
        if (FlagOn(Data->Flags,FLTFL_CALLBACK_DATA_IRP_OPERATION))
        {/*为 IRP 操作建立一个 MDL。 */
            newMdl=IoAllocateMdl(newBuffer,ulReadLen,FALSE,FALSE,NULL );
            MmBuildMdlForNonPagedPool( newMdl );
        }
        /* 获取一个 CompletionContext 上下文结构 */
```

```
            pMyCompletionContext =
(PCOMPLETION_CONTEXT)ExAllocateFromNPagedLookasideList( &CompletionContextLookasideList );
            /* 回调数据结构 */
            iopb->Parameters.Read.ReadBuffer = newBuffer;
            iopb->Parameters.Read.MdlAddress = newMdl;
            FltSetCallbackDataDirty( Data );
           /* 封装将上下文和 buffer，传给后操作回调*/
            pMyCompletionContext->SwappedBuffer = newBuffer;
            pMyCompletionContext->pVolumeContext = pVolumeContext;
            pMyCompletionContext->pStreamContext = pStreamContext ;
            *CompletionContext = pMyCompletionContext;
            FltStatus = FLT_PREOP_SUCCESS_WITH_CALLBACK;
      } __finally {
      . . . 清理工作  . . .
      return FltStatus;
}
```

2) Read 请求的后操作回调例程：

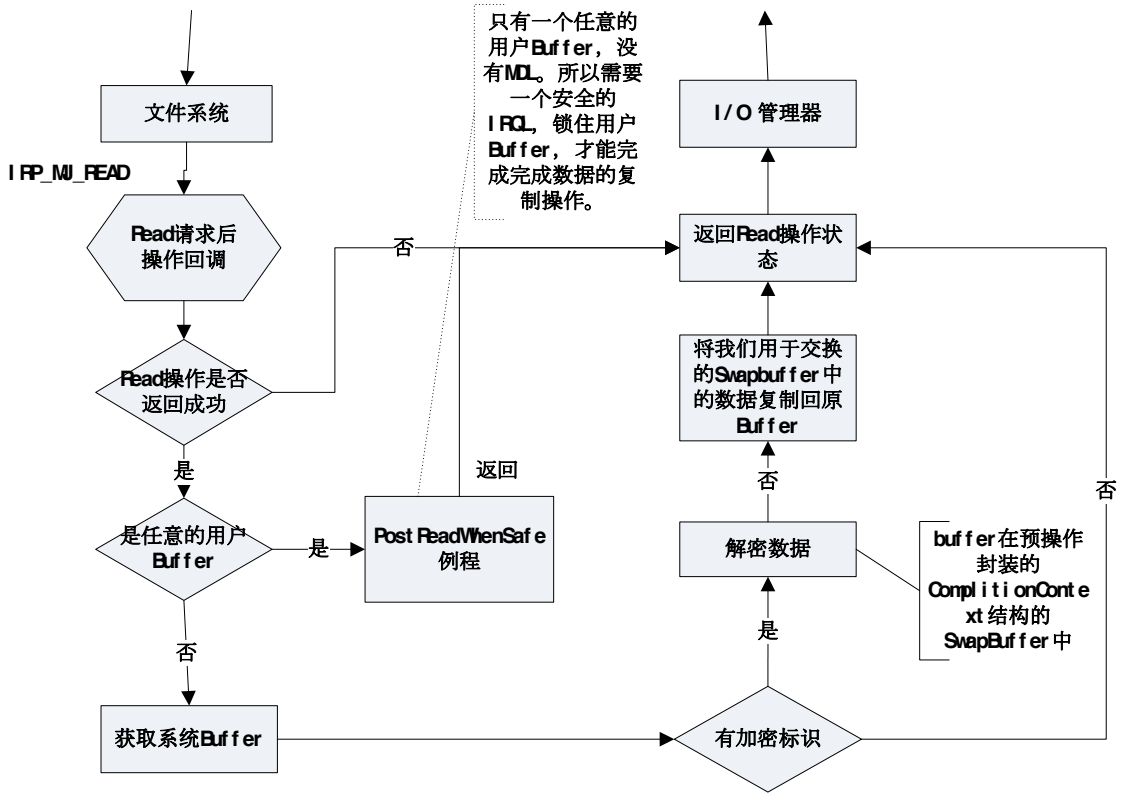


图4-4　Read 请求后操作处理流程

关键源码：

```
FLT_POSTOP_CALLBACK_STATUS
EncrytingFilter_PostRead(
      __inout PFLT_CALLBACK_DATA Data,
```

```
    __in PCFLT_RELATED_OBJECTS FltObjects,
    __in PVOID CompletionContext,
    __in FLT_POST_OPERATION_FLAGS Flags)
{
    NTSTATUS status = STATUS_SUCCESS ;
    FLT_POSTOP_CALLBACK_STATUS FltStatus = FLT_POSTOP_FINISHED_PROCESSING;
    …… ……  其它变量定义 …… ……
    __try {
        /*若操作读操作失败，或读长度为 0，表明没有数据要复制，只需马上返回即可*/
        if (!NT_SUCCESS(Data->IoStatus.Status) ||  (ulReadLen== 0))
        {    __leave; }
    /*  需要注意的是，传进来的参数是用户原来的 buffer，而不是预操作中我们所开辟的用于
        交换的 swapBuffer。因此，我们需要将读到的数据复制回用户 buffer。*/
        if (iopb->Parameters.Read.MdlAddress != NULL) {
            /*为原始的缓冲定义一个 MDL，并为它获取一个系统地址。*/
              originalBuffer = MmGetSystemAddressForMdlSafe( iopb->Parameters.Read.MdlAddress,
NormalPagePriority );
              if (originalBuffer == NULL) {
                  /*  如果获取 SYSTEM 地址失败，标记读失败并返回 */
                  Data->IoStatus.Status = STATUS_INSUFFICIENT_RESOURCES;
                  Data->IoStatus.Information = 0;
                  __leave;
              }
        }else if (FlagOn(Data->Flags,FLTFL_CALLBACK_DATA_SYSTEM_BUFFER) ||
FlagOn(Data->Flags,FLTFL_CALLBACK_DATA_FAST_IO_OPERATION))
        {
         /* 如果这是一个系统缓冲，只需要用所给的地址即可，因为它在所有的线程上下文中
            都是有效的。然而，如果这是一个 FASTIO 操作，只需在 （try/except）结构中使
            用这个缓冲即可，因为我们知道此时处在正确的线程上下文中。*/
            originalBuffer = iopb->Parameters.Read.ReadBuffer;
        } else {
        /*原始的用户缓冲没有 MDL，并且这不是一个系统缓冲或一个 FASTIO，由此可知这可
         能是一些任意的用户缓冲。我们不能在 DPC 中断级中对其进行处理，因此需要尝试获
         取一个安全的 IRQL 才能做这个处理。*/
            if (FltDoCompletionProcessingWhenSafe( Data,FltObjects,
                                                   CompletionContext,Flags,
                                               EncrytingFilter_PostReadWhenSafe,
                                                   &FltStatus )) {
              /*  此时，该操作已经被移到一个安全的 IRQL 中，并且被调用函数已经做完了
              清理工作，在此函数中不必再做清理工作。*/
                cleanupAllocatedBuffer = FALSE;
            } else {
           /*我们没有取得一个安全的 IRQL，并且没有一个 MDL。在这样一个情形下，我们
```

```
            没有办法安全的将数据复制回用户缓冲中，此时 post 是不安全的，我们应该获得一
            个 MDL，因此使操作失败并返回。*/
                __leave;
            }
/* 执行到这里，我们已经有一个系统缓冲或可确定这是一个 FASTIO 操作。因此，当前在一个
  适当的上下文中。下面将开始复制数据，并进行异常处理。*/
        __try {
            do{
                /* 若文件是机密文件，或者已标记在读的时候解密，那么需要解密文件数据 */
                if (pMyComletionContext->pStreamContext->bIsFileCrypt ||
pMyComletionContext->pStreamContext->bDecryptOnRead){
        /* 若读数据长度超过了文件真实长度，修改返回的数据长度，使其符合文件真实大小 */
                        if (pMyComletionContext->pStreamContext->FileValidLength.QuadPart <
(liReadOffset.QuadPart+ ulReadLen)){
                            Data->IoStatus.Information =
(ULONG)(pMyComletionContext->pStreamContext->FileValidLength.QuadPart -   liReadOffset.QuadPart) ;
                            FltSetCallbackDataDirty(Data) ;
                        }
                        KeEnterCriticalRegion() ;
                        if(pMyComletionContext->pStreamContext->bIsSetKey == FALSE )
                        {//   使用当前秘钥解密
    File_DecryptBuffer(pMyComletionContext->SwappedBuffer,ulReadLen,pMyComletionContext->pVolu
meContext->ucFileKey,&CryptIndex, liReadOffset.QuadPart);
                        }else{// 使用文件指定的秘钥去解密文件数据
    File_DecryptBuffer(pMyComletionContext->SwappedBuffer,ulReadLen,pMyComletionContext->pStrea
mContext->ucFileKey,&CryptIndex, liReadOffset.QuadPart);
                        }
                        KeLeaveCriticalRegion() ;}
            }while(FALSE) ;
        /* 将加密后的数据复制回用户 buffer */
        RtlCopyMemory( originalBuffer,pMyComletionContext->SwappedBuffer, ulReadLen);
        } __except (EXCEPTION_EXECUTE_HANDLER) {
            /* 复制失败，返回错误，使操作失败 */
            Data->IoStatus.Status = GetExceptionCode();
            Data->IoStatus.Information = 0;
        }
    } __finally { if (cleanupAllocatedBuffer) {. . . 清除工作 . . .}}
   return FltStatus;
}
```

4.4 Write 回调例程

IRP_MJ_WRITE 请求由 I/O 管理器或文件系统驱动发送。当用户模式应用程序调用 Win32 函数，例如 WriteFile 或当内核模式组件调用如 ZwWriteFile 的文件写操作函数时，

请求被发送。本文为 IRP_MJ_WRITE 请求注册了预操作回调例程和后操作回调例程。预操作回调例程主要是完成上下文的获取，以及交换缓冲区的分配，并对机密文档内容进行加密处理。后操作回到例程，主要是完成内存空间的释放，并继续向上返回操作状态。以下将对 Write 请求的回调例程进行详细的设计与实现。

1) Write 请求的预操作回调例程

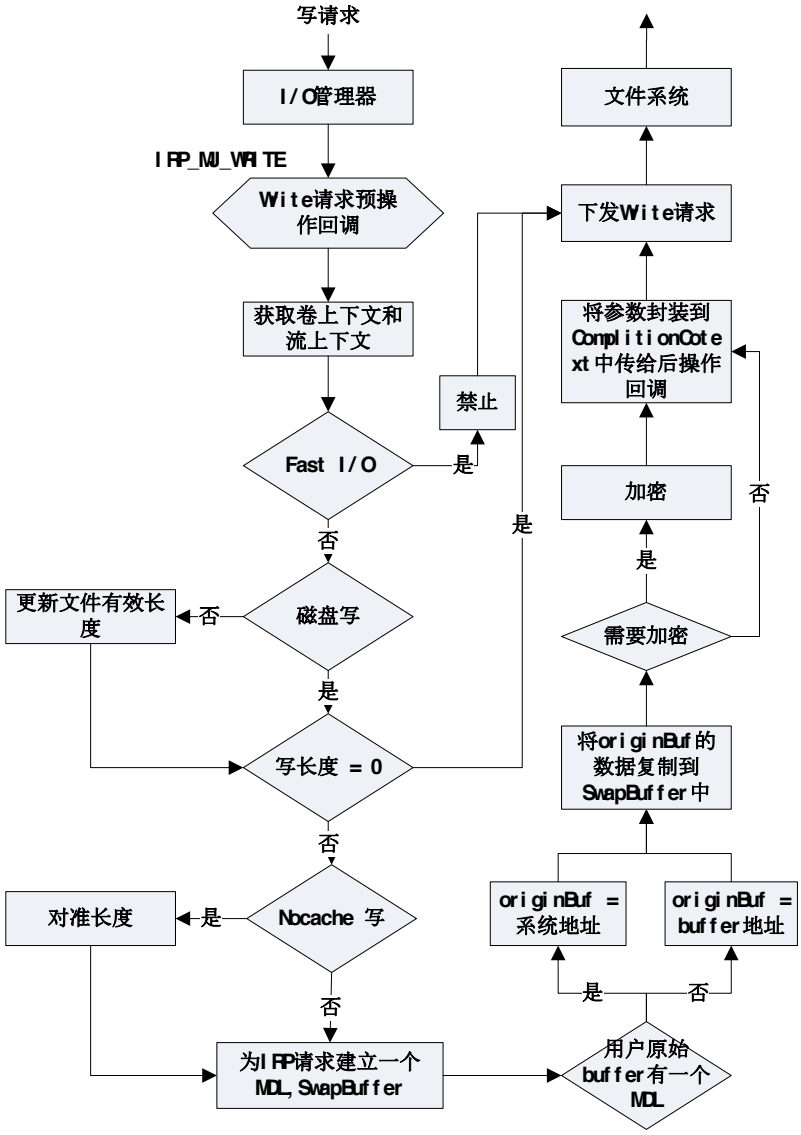


图4-5　Write 请求预操作回调例程流程图

关键源码：

```
FLT_PREOP_CALLBACK_STATUS
EncrytingFilter_PreWrite(
    __inout PFLT_CALLBACK_DATA Data,
    __in PCFLT_RELATED_OBJECTS FltObjects,
    __deref_out_opt PVOID *CompletionContext)
{
```

```
    NTSTATUS status;
    FLT_PREOP_CALLBACK_STATUS FltStatus = FLT_PREOP_SUCCESS_NO_CALLBACK;
    . . . 其它变量定义 . . .

    __try {
        status = FltGetVolumeContext( FltObjects->Filter,FltObjects->Volume,&pVolumeContext );
        status = Contex_FindOrCreateStreamContext(Data,FltObjects,FALSE,&pStreamContext,NULL) ;
        bReturn = Process_IsMonitoredProcess
(pStreamContext->usFileName.Buffer,pStreamContext->usFileName.Length/sizeof(WCHAR),
&bIsSystemProcess, &bIsPPTFile);
        if (!bReturn){__leave ;}
        if (!(Data->Iopb->IrpFlags & (IRP_NOCACHE | IRP_PAGING_IO |
IRP_SYNCHRONOUS_PAGING_IO))){ /* 非磁盘写，记下写长度 */
            SC_LOCK(pStreamContext, &OldIrql) ;
            if ((ulWriteLen + liWriteOffset.QuadPart) > pStreamContext->FileValidLength.QuadPart)
            {//扩大文件大小
                pStreamContext->FileValidLength.QuadPart = ulWriteLen + liWriteOffset.QuadPart ;
            }
            SC_UNLOCK(pStreamContext, OldIrql) ;
            __leave ;
        }
        newBuffer = ExAllocatePoolWithTag( NonPagedPool, ulWriteLen,BUFFER_SWAP_TAG );

        /*  为 IRP 操作建立一个 MDL。*/
        if (FlagOn(Data->Flags,FLTFL_CALLBACK_DATA_IRP_OPERATION)) {
            newMdl=IoAllocateMdl(newBuffer,ulWriteLen,FALSE,FALSE, NULL );
            MmBuildMdlForNonPagedPool( newMdl );
        }

        /* 如果用户原始的 buffer 有一个 MDL，获取一个系统地址 */
        if (iopb->Parameters.Write.MdlAddress != NULL){
            originalBuffer = MmGetSystemAddressForMdlSafe( iopb->Parameters.Write.MdlAddress,
NormalPagePriority );
            if (originalBuffer == NULL){
                Data->IoStatus.Status = STATUS_INSUFFICIENT_RESOURCES;
                Data->IoStatus.Information = 0;
                FltStatus = FLT_PREOP_COMPLETE;
                __leave;
            }
        } else{    /* 这里没有已定义的 MDL，使用所给的 buffer 地址。*/
            originalBuffer = iopb->Parameters.Write.WriteBuffer;
        }
```

```
            /*复制内存，必须在 try/except 结构中进行操作，因为这里可能会
            使用一个用户的 buffer 地址。*/
            try {RtlCopyMemory( newBuffer,originalBuffer,ulWriteLen );
                do {KeEnterCriticalRegion() ;
                    if( FALSE == pStreamContext->bIsSetKey){// 使用当前秘钥加密数据
                        File_EncryptBuffer(newBuffer, ulWriteLen,
pVolumeContext->ucFileKey,&CryptIndex,liWriteOffset.QuadPart);
                    }else{// 使用文件制定秘钥加密数据
                        File_EncryptBuffer(newBuffer,ulWriteLen,
pStreamContext->ucFileKey,&CryptIndex,liWriteOffset.QuadPart);
                    }
                    KeLeaveCriticalRegion() ;
              }while(FALSE) ;
            } __except (EXCEPTION_EXECUTE_HANDLER) {
                ……  ……
            }

            pMyCompletionContext =
ExAllocateFromNPagedLookasideList( &CompletionContextLookasideList );

            /* 设置新的 buffer 并将状态传递到后操作回调中 */
            iopb->Parameters.Write.WriteBuffer = newBuffer;
            iopb->Parameters.Write.MdlAddress = newMdl;
            FltSetCallbackDataDirty( Data );
            pMyCompletionContext->SwappedBuffer = newBuffer;
            pMyCompletionContext->pVolumeContext = pVolumeContext;
            pMyCompletionContext->pStreamContext = pStreamContext ;
            *CompletionContext = pMyCompletionContext;

            /*  返回，需要一个后操作回调 */
            FltStatus = FLT_PREOP_SUCCESS_WITH_CALLBACK;
        } finally {
          . . . 清理工作 . . .
        }

        return FltStatus;
}
```

2) Write 请求的后操作回调例程

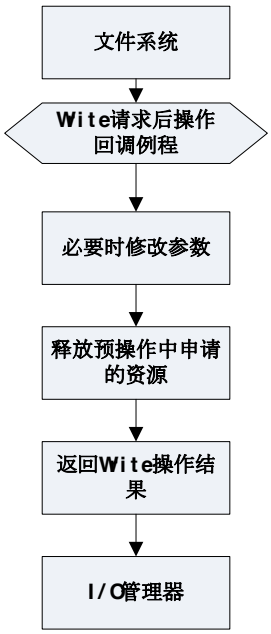


图4-6　Write 请求预操作回调例程流程图

4.5 其它回调例程

1) IRP_MJ_CLOSE：本系统为该请求注册预操作回调例程 EncrytingFilter_PreClose。当该回调例程被触发时，必须要减少流上下文的引用。若流上下文的引用减少到 0，文档被关闭，若该文档为受保护文档，需要将加密标识写入文件尾，并且清除缓存。
2) IRP_MJ_QUERY_INFORMATION：注册了 EncrytingFilter_PreQueryInfo 预操作回调例程和 EncrytingFilter_PostQueryInfo 后操作回调例程，实现对受保护文档的文件标识进行隐藏的工作。当用户需要查询文件信息时，要重新计算，返回除了填充和文件标识之外的，真正的文本内容长度与偏移。
3) IRP_MJ_SET_INFORMATION：注册了 EncrytingFilter_PreSetInfo 预操作回调例程和 EncrytingFilter_PostSetInfo 后操作回调例程。当需要设置文件信息时，必须要经过重新计算，以达到对文件标识以及额外填充内容进行隐藏的目的。
4) IRP_MJ_CLEANUP：注册了 EncrytingFilter_PreCleanup 预操作回调例程。当 IRP_MJ_CLEANUP 请求到来时，表示文件对象句柄的应用计数值已经减少到 0。换句话说，就是文件对象的所有句柄都已经被关闭。此时，需要对机密文档的缓冲进行清除，以防机密内容泄露。

4.6 内核层与用户层通信

本系统利用 Minifilter 对用户层与内核层通信的支持，很好的实现了用户层与内核层的信息交互。内核层在入口函数 DriverEntry 中调用端口的初始化函数 Msg_CreateServerPort()，创建服务端口。该函数调用成功后，微过滤驱动开始接收来自用户层的连接，用户层通过调用 FilterConnectCommunicationPort() API 来请求与内核模块建立连接，此时，将会触发内

核层 Msg_ConnectNotifyCallback()回调例程，内核模块在此回调例程中记录下用户层客户端的信息。成功建立连接后，内核层与用户层开始进行双向通信，此时，用户层可通过 FilterSendMessage() API 向内核模块发送消息，这将触发内核模块的 Msg_MessageNotifyCallback()回调例程，内核模块在此回调中对用户层发来的消息进行解析，并完成指定的操作，再通过该函数的输出缓冲参数，返回操作结果。当用户层端口的引用减少到 0 时，即用户层的相关程序被关闭时，会自动触发内核层回调例程 Msg_DisconnectNotifyCallback()，内核模块在此回调例程中关闭连接，并清除相关信息记录。其中 Msg_CreateServerPort 实现如下：

```
NTSTATUS Msg_CreateCommunicationPort( IN PFLT_FILTER pFilter )
{
    ......
    /*   建立安全描述符 */
    status = FltBuildDefaultSecurityDescriptor(&pSecurityDescriptor,FLT_PORT_ALL_ACCESS) ;
    /* 初始化服务端口 */
    RtlInitUnicodeString(&uPortName, SERVER_PORTNAME) ;
    InitializeObjectAttributes(&objectAttributes, &uPortName,
        OBJ_CASE_INSENSITIVE|OBJ_KERNEL_HANDLE, NULL,pSecurityDescriptor) ;
    /* 创建新的服务端口 */
    status = FltCreateCommunicationPort(
        pFilter,                        //   过滤驱动对象句柄
        &g_pServerPort,                 //   服务端口
        &objectAttributes,  NULL,
        Msg_ConnectNotifyCallback,      //   连接回调例程
        Msg_DisconnectNotifyCallback,   //   断开连接回调例程
        Msg_MessageNotifyCallback,      //   发送信息回调例程
        MAX_CONNECTIONS                 //   最大连接数
        ) ;
    …… ……
    return status ;
}
```

4.7 剪切板监控模块与进程防终止模块

剪切板监控模块与进程防终止模块主要基于 HOOK API 技术实现。HOOK API 的实现过程主要分为两个步奏：其一是实现一个与原 API 一样的函数接口，其中加入自己的控制代码，用新 API 的地址替换原 API 的地址；二是将实现 HOOK API 功能的 DLL 注入到目标进程中。DLL 可跟随系统钩子一起被注入到目标进程中，本系统利用鼠标钩子，实现自定义 HOOK DLL 的注入。这样一来只要 HOOK 的 API 被调用就会被重定向到我们的替换函数中。

4.7.1 剪切板监控模块

剪切板监控模块通过 HOOK 与剪切板相关的 API 来实现对复制、剪切和粘贴操作的控制。其中复制或剪切由 API SetClipboardData() 实现，粘贴由 API 实现 GetClipboardData()，这两个 API 均在 User32.dll 中。使用 g_bIsMonitor 共享变量来标记当前剪切板中的内容是否为机密内容，在替换的 SetClipboardData()中进行设置。当替换的 GetClipboardData()函数被调用时，表明有进程需要获取剪切板内容，这时候需要判断进程对联合 g_bIsMonitor 来判断操作的合法性。主要的实现过程如下：

1) 共享数据定义

```
#pragma data_seg("Share")
HWND g_hwnd = NULL;                    //    主窗口句柄，加载 HOOK 的时候传入
HINSTANCE hInstance = NULL;            //    本 DLL 的实例句柄
HHOOK hhook = NULL;                    //    鼠标钩子句柄
unsigned int g_bIsMonitor = 0;         //    剪切板中内容状态,1：机密信息，0：非机密信息
#pragma data_seg()
#pragma comment(linker,"/section:Share,rws")
```

2) API 重定义

```
typedef HANDLE ( WINAPI *TypeSetClipboardData )(_In_ UINT uFormat,_In_opt_ HANDLE hMem);
typedef HANDLE ( WINAPI *TypeGetClipboardData )(_In_ UINT uFormat);
```

3) 替换原 API 地址，以 GetClipboardData()为例

```
    // 加载 User32.dll
    HMODULE hmodleUser32;
    hmodleUser32 = ::LoadLibrary(_T("User32.dll"));
    //    获取原函数地址
    oldGetClipboardData = (TypeGetClipboardData)::GetProcAddress(hmodleUser32,"GetClipboardData");
    pfOldGetClipboardData = (FARPROC)oldGetClipboardData;
    /*    保存原函数入口  */
    _asm
    {
        lea edi, oldCodeGet
        mov esi,pfOldGetClipboardData
        cld
        mov ecx,CODE_LENGTH
        rep movsb
    }
```

4) 将替换地址写入内存，若恢复原 API 地址，则 lpAddress 指向原 API 入口，若替换原 API，实现 HOOK，则 lpAddress 为自己实现的，新的 API 入口。主要代码如下：

```
BOOL WriteMemory(LPVOID lpAddress,BYTE* pcode,size_t length)
{
    . . . . . .
    dwRet = VirtualProtectEx(hProcess,lpAddress,length,PAGE_READWRITE,&dwOldProtect);
```

```
    dwRet = WriteProcessMemory(hProcess,lpAddress,pcode,length,&dwWrited);
    dwRet = VirtualProtectEx(hProcess,lpAddress,length,dwOldProtect,&dwtemp);
    return TRUE;
}
```

5) 跟随鼠标钩子注入目标进程：

```
hhook = ::SetWindowsHookEx(WH_MOUSE,MouseProc,hInstance,0);
```

6) 自己编写的 API：

```
/*
    “粘贴”，操作会调用该函数完成。
    机密进程 -》 机密进程        ：   允许
    非机密进程 -》 机密进程      ：   允许
    机密进程 -》 非机密进程      ：   禁止
*/
HANDLE WINAPI MyGetClipboardData(_In_ UINT uFormat)
{
    . . . . . .
    if (1 == g_bIsMonitor){// 当前剪切板中的是密文
        int ret = con2Kernel.IsMonitorProcess(processName);
        if (ERROR_RET == ret){//   出错返回失败
            return NULL;
        }else if (NOMONITOR == ret){// 若不是机密进程，则返回失败
            return NULL;
        }
    }
    WriteMemory(pfOldGetClipboardData,oldCodeGet,CODE_LENGTH); /*   恢复原函数地址  */
    handle = oldGetClipboardData(uFormat);      /*   调用原 API 处理合法操作  */
    WriteMemory(pfOldGetClipboardData,newCodeGet,CODE_LENGTH); /*   继续 HOOK */
    return handle;
}
/*
    “剪切”或“复制”操作，会调用这个函数。
    设置全局标志：
        g_bIsMonitor = 1        ：        剪切板内容属机密内容
        g_bIsMonitor = 0        ：        剪切板内容属非机密内容
*/
HANDLE WINAPI MySetClipboardData(_In_ UINT uFormat,_In_opt_ HANDLE hMem)
{
    . . .   . . .
    int ret = con2Kernel.IsMonitorProcess(processName);
    if (MONITOR == ret || ERROR_RET == ret){//   是机密进程
        if (0 == g_bIsMonitor)
        { InterlockedIncrement(&g_bIsMonitor);}
```

```
    }
    else{//    非机密进程
        if (1 == g_bIsMonitor)
        {    InterlockedDecrement(&g_bIsMonitor);}
    }
    WriteMemory(pfOldSetClipboardData,oldCodeSet,CODE_LENGTH);
    handle = oldSetClipboardData(uFormat,hMem);
    WriteMemory(pfOldSetClipboardData,newCodeSet,CODE_LENGTH);
    return handle;
}
```

4.7.2 进程放终止模块

进程放终止模块通过 HOOK Kernel32.dll 中的 TerminateProcess()和 OpenProcess() API 实现进程保护。如果只 HOOK OpenProcess()其中一个，那么任务管理器将由于不能获取到受保护进程的信息而出错；若只 HOOK TerminateProcess()也是不可行的，因为一个进程句柄在本进程与其它进程中是不一样的，所以在不知道受保护进程在其他进程中的句柄时，是无法 HOOK TerminateProcess()的。本模块首先利用 OpenProcess()取得受保护进程在其它进程中的句柄，然后通过 TerminateProcess()来禁止其它进程非法终止受保护进程。其实现过程与剪切板监控模块类似，因此这里仅给出替换函数的实现。

```
#pragma data_seg("Share")
… …
DWORD g_dwProcessId = 0;            // 保护进程 id
HANDLE g_hProcess = NULL;           //保存本进程在远程进程中的句柄
#pragma data_seg()
#pragma comment(linker,"/section:Share,rws")
BOOL WINAPI MyTerminateProcess(_In_ HANDLE hProcess, _In_ UINT uExitCode)
{
    BOOL ret;
    DWORD dwRet;
    if (g_hProcess == hProcess){
        AfxMessageBox(_T("不能关闭受保护进程哦！！"));
        ret = TRUE;
    }else{
        WriteMemory(pfOldTerminateProcess,oldCodeTermPro,CODE_LENGTH);
        ret = oldTerminateProcess(hProcess,uExitCode);
        WriteMemory(pfOldTerminateProcess,newCodeTermpro,CODE_LENGTH);
    }
    return ret;
}
HANDLE WINAPI MyOpenProcess(_In_ DWORD dwDesiredAccess,_In_ BOOL   bInheritHandle,_In_
DWORD dwProcessId)
{
```

```
    HANDLE hProcess = NULL;
    WriteMemory(pfOldOpenProcess,oldCodeOpenPro,CODE_LENGTH);
    hProcess = oldOpenProcess(dwDesiredAccess,bInheritHandle,dwProcessId);
    if ( dwProcessId == g_dwProcessId){
        g_hProcess = hProcess;
    }
    WriteMemory(pfOldOpenProcess,newCodeOpenPro,CODE_LENGTH);
    return hProcess;
}
```

4.8 加密策略

本系统在对文档的保护过程中，遵循以下策略：

1) 机密文档在磁盘中是密文、在内存中是明文。
2) 机密进程新建属于保护类型的文档时，写时需要加密。
3) 机密进程打开属于保护类型的文档时，自动解密，修改后保存，自动加密。
4) 机密进程打开原有的属于保护文档类型的却没有被加密的文档，读不解密，关闭不加密。只有当该文档被修改，并进行保存时，对该文档进行加密。
5) 非机密进程操作的文档不需要进行加解密处理。
6) 非机密进程打开机密进程保护的文档时，只能看到密文，无法获取明文。
7) 剪切板监控，禁止复制/剪切机密文档内容到非机密文档。
8) 自定义文件加密标识，并将其写入已加密文档的结尾。文件加密标识存储该文档的相关信息，以及文件密钥的摘要。为了安全起见，本文使用多个文件加密密钥，因此需要在文档中存储对应的密钥摘要，以便于在加解密过程中匹配对应的文件密钥。采用数据隐藏技术，使文件尾多加的文件标识对应用程序来说是不可见的。
9) 采用分组加密算法 AES，分组长度为 128bit，密钥长度为 256。本系统在加密时对长度不足的明文加入空格作为填充，以达到分组要求。在解密过程中，会将填充空格去掉，再将实际明文内容返回给应用程序。

4.9 客户端-服务器通信模块

本文实现服务器与客户端之间稳定、安全的通信。以下给出该模块的详细设计图：

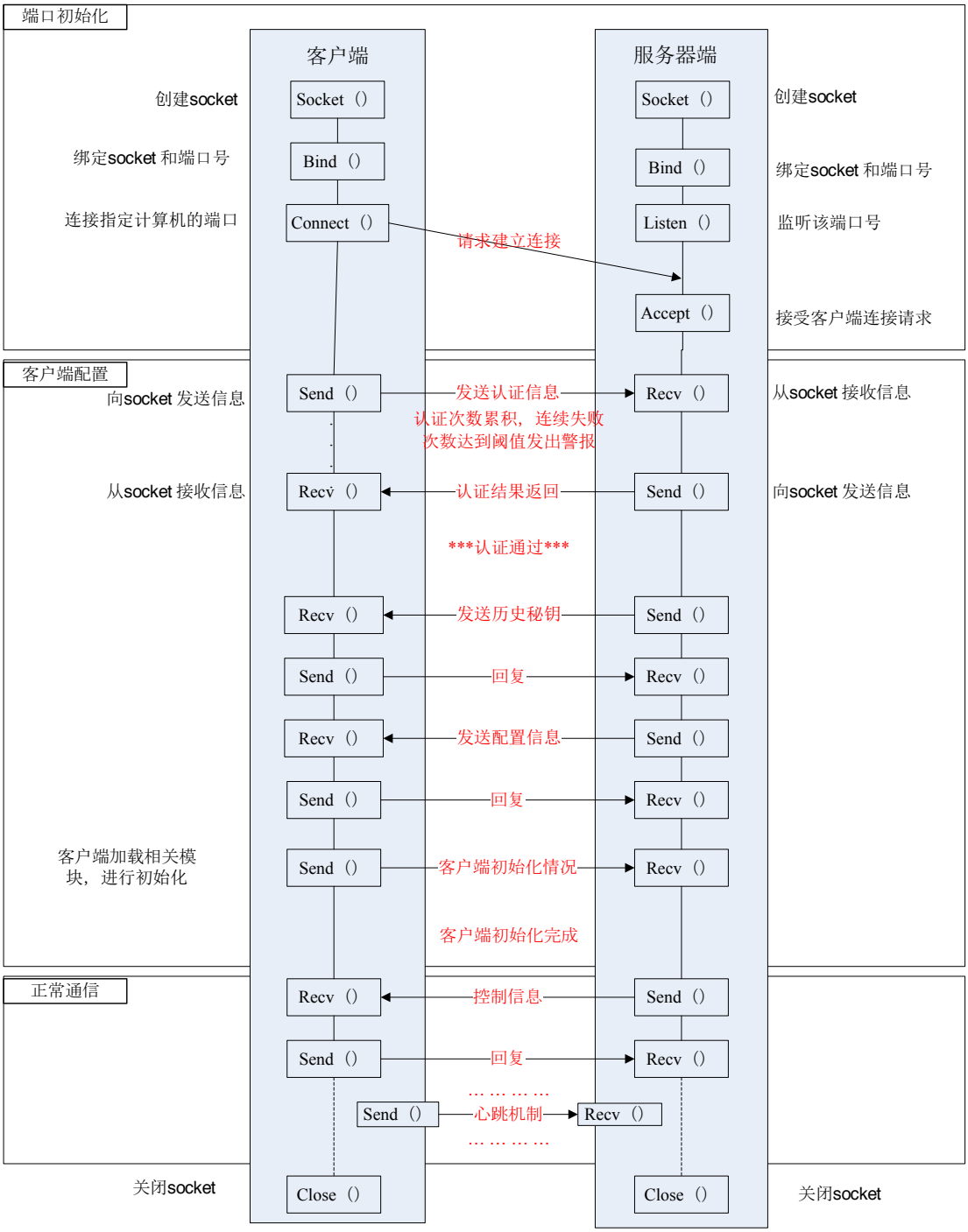


图4-7　服务器-客户端通信过程

4.10　本章小结

本章主要介绍系统核心模块的详细设计与实现，给出系统实现过程的主线。首先给出系统的加载过程，关键数据结构的定义，以及驱动入口函数 DriverEntry 的实现。其次，给出主要回调例程的实现过程，包括 Create、Read、Write 回调例程。还给出各模块之间通信过程的实现，其中包括内核层与用户层控制程序的通信过程，服务器端与客户端的通信过程。还实现剪切板的监控，进程防终止，完成加密策略的定制。

第五章 系统测试与分析

本章将对系统进行了较为全面的测试。由于各 Windows 平台下的测试情况多有类同，这里仅给出在同时安置了 NTFS 和 FAT 文件系统的 Windows Sp3 操作系统上的测试情况，其他系统的测试结果不再赘述。

5.1 系统测试环境

本文实现基于 Windows 操作系统的透明加密过滤系统，系统的测试环境为 Windows NT 系列操作系统。较为典型的有 Windows XP、Windows 7、Windows Vista、Windows Server 2003/2008 等。本文主要选择 Windows XP 与 Windows 7 作为测试环境，其它系统理论上应该可以兼容，但没有进行严格的测试。在 Windows 所支持的 FAT 和 NTFS 文件系统中，本系统的内核模块均能稳定运行，达到预期效果。表 5-1 为测试环境信息。

表5-1 系统测试环境

操作系统	Windows XP、Windows 7
处理器	AMD A6-4455M APU with Radeon(tm) HD Graphics ，MMX，2.1GHz
内存	1024MB RAM
文件系统类型	FAT、NTFS

接下来仅给出在 Windows XP SP3 中测试的结果。在 Windows7 操作系统中的测试结果相同。理论上兼容 Windows 系列系统，但除以上两种 Windows 操作系统版本之外，其他版本未经过严格的测试。

5.2 系统部署

编译工程得到以下程序：HFLEncryptingSystemServer.exe，服务器端程序；HFLEncryptingSystemClient.exe，客户端应用层控制程序；HFLEncryptingFilter.sys 客户端内核过滤驱动；DllModel.dll 用户层与内核层接口；MonitorDll.dll，实现挂钩剪切板的功能模块，这将被注入各进程中。运行 HFLEncryptingFilter.inf 文件，加载内核模块的透明加密过滤驱动；运行服务器程序 HFLEncryptingSystemServer.exe，进入监听状态，此时可以接收来自客户端的连接请求；运行客户端用户层控制程序 HFLEncryptingSystemClient.exe，完成 DLL 的加载，以及 MonitorDll.dll 的注入，与服务器完成认证连接；连接成功后服务器与客户端之间可以开始进行通信，可在服务器端获取客户端的监控状态。

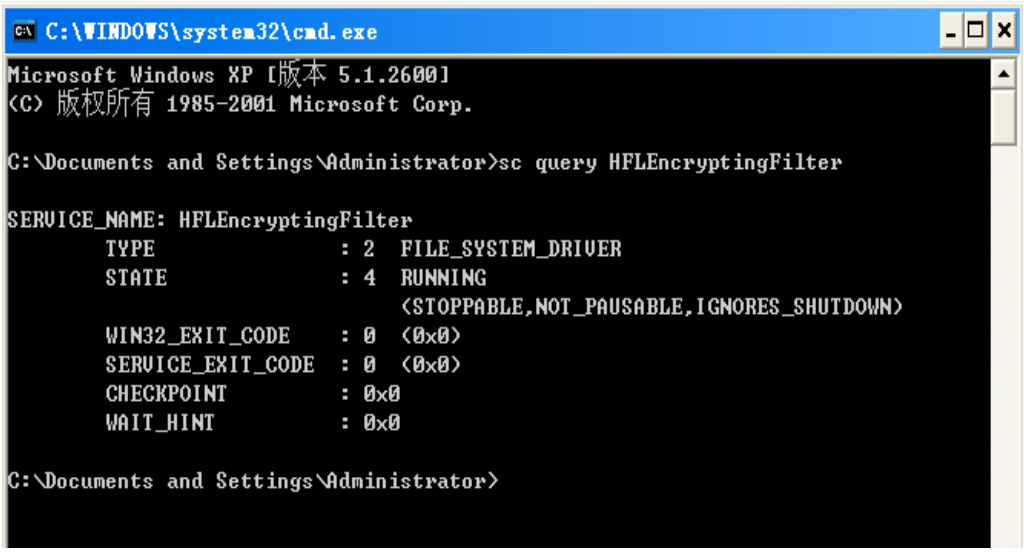


图5-1　　内核加密模块加载情况

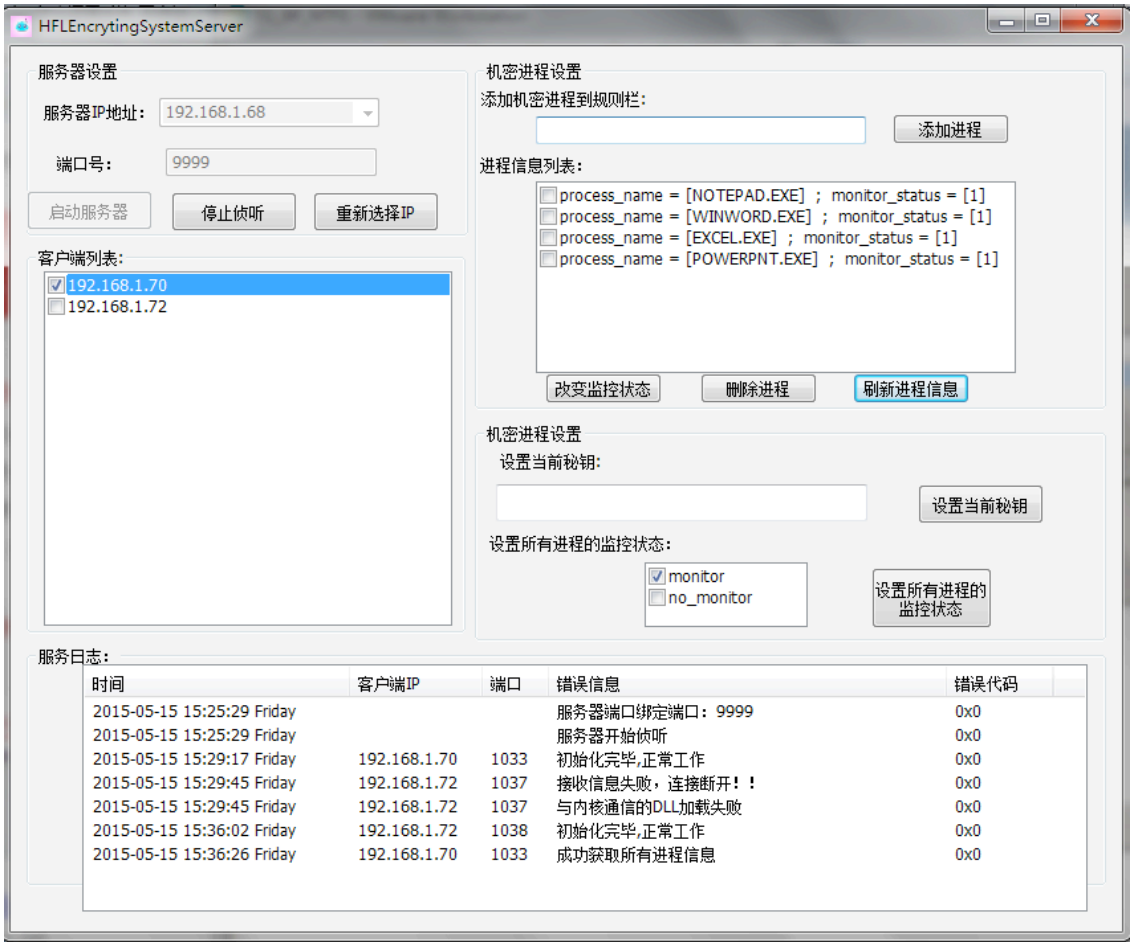


图5-2　　服务器接收连接，获取监控进

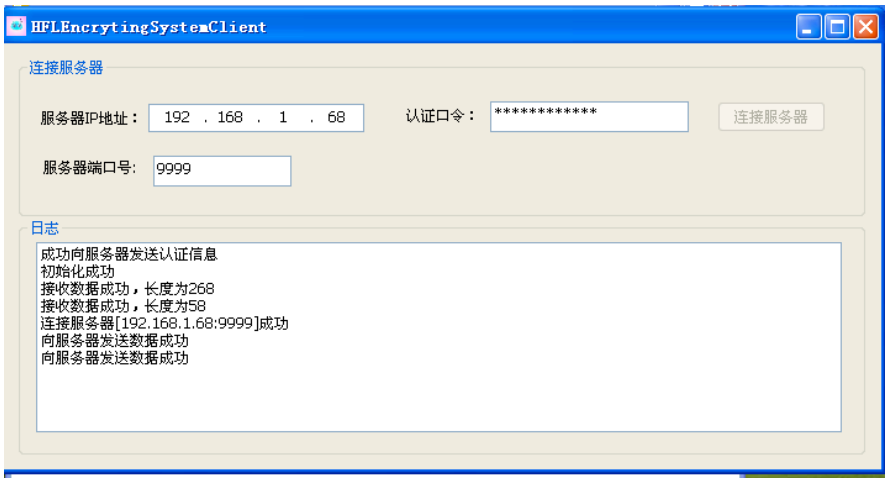


图5-3　客户端连接服务器

5.3 系统功能性验证

本系统基于机密进程与文档后缀名设置受保护的文档，由于测试过程多有重复。以下仅给出主要类型机密进程，以及其处理的典型的文件后缀名文档的测试情况。其中有WINWORD.EXE，.doc；EXCEL.EXE，.xls；POWERPNT.EXE，.ppt；NOTEPAD.EXE，.txt。

（1） WINWORD.EXE 进程，doc 文档测试

操作过程：1.新建文件 myfile.doc；2.输入明文信息；3. 保存 myfile.doc 文件；4.将 myfile.doc 文件从虚拟机中拷贝到物理主机中，物理主机中没有加解密环境；5. 在物理主机中打开 myfile.doc 文件，只能获取密文乱码；6.这时在部署了透明加密过滤系统的虚拟机中可正常打开 myfile.doc，阅读明文。整个测试符合预期要求。

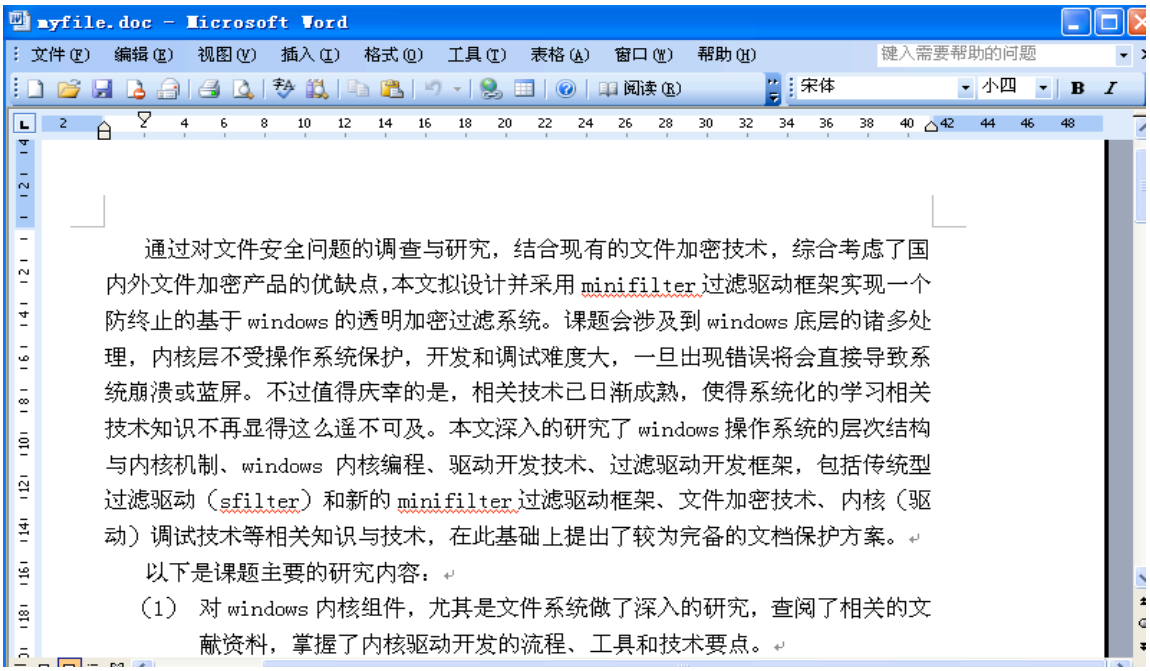


图5-4　myfile.doc 明文信息

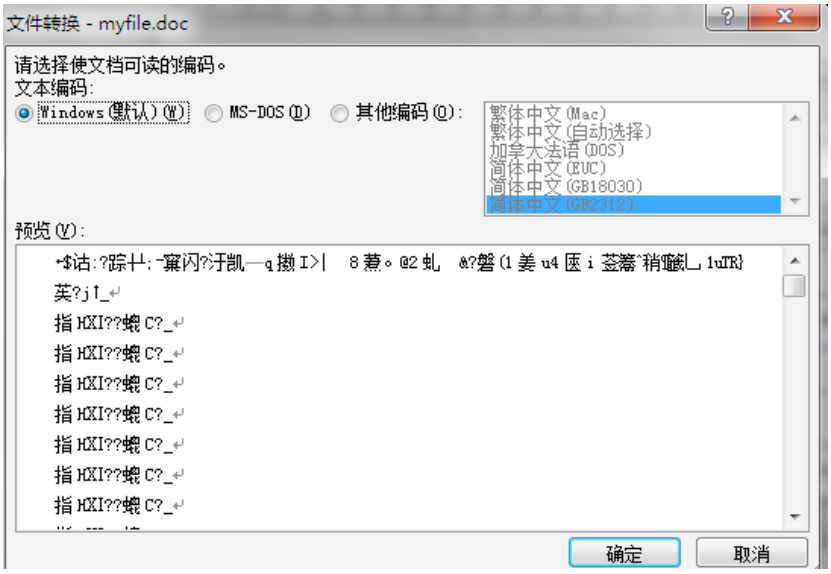


图5-5　物理机打开 myfile.doc 提示窗口

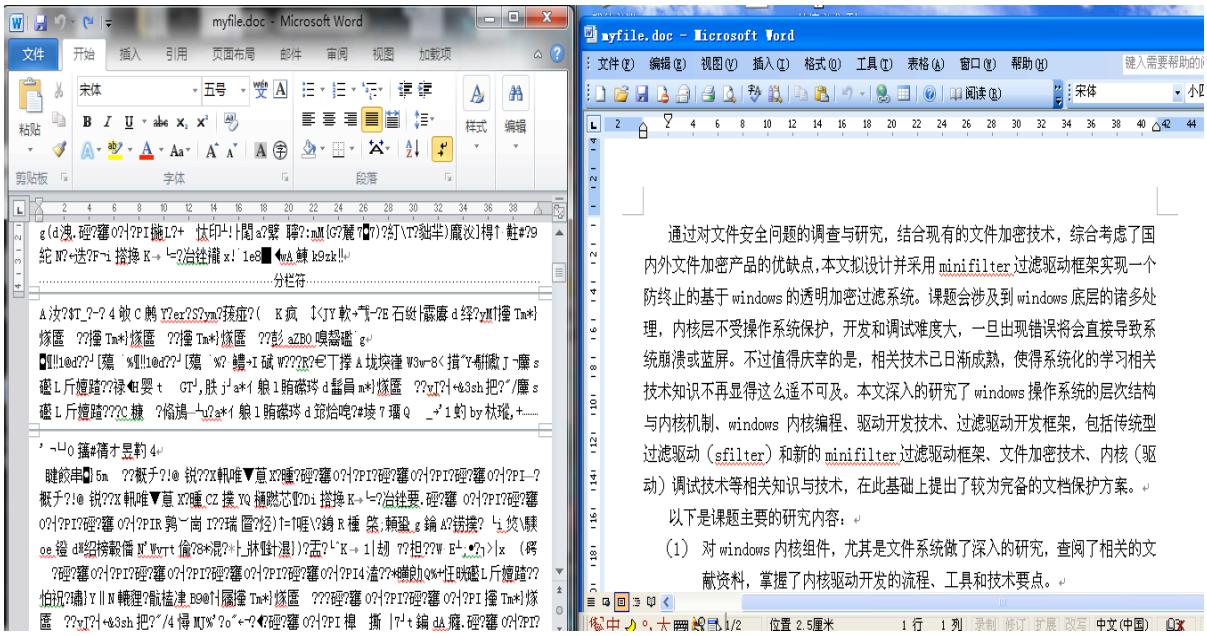
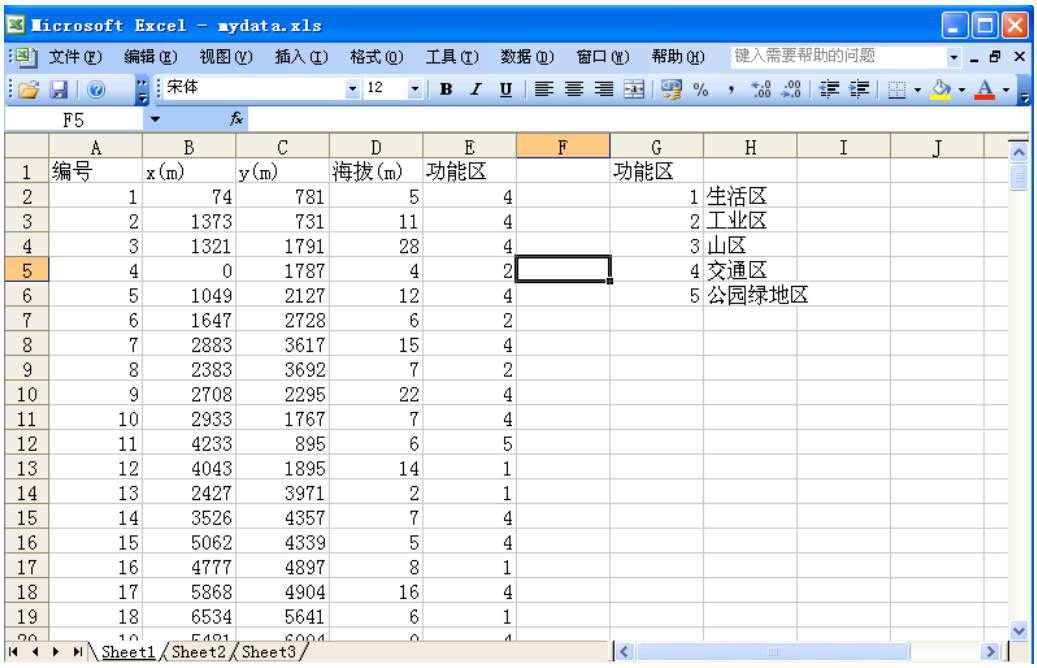


图5-6　主机与 VMware 的 myfile.doc 文档

（2） EXCEL.EXE

操作过程：1.新建 mydata.xls 文档；2.输入文档内容；3.保存 mydata.xls 文档；4. 将虚拟机中的 mydata.xls 文档复制到物理机中；5.打开 mydata.xls 文档，提示“格式不一致”；6.选择继续打开文档，只能得到密文文本，无法获知明文内容；7.此时虚拟机可以正常对 mydata.xls 文档进行操作。全部过程符合预期要求，达到了保护文档的目的。



	A	B	C	D	E	F	G	H	I	J
1	编号	x(m)	y(m)	海拔(m)	功能区		功能区			
2	1	74	781	5	4		1	生活区		
3	2	1373	731	11	4		2	工业区		
4	3	1321	1791	28	4		3	山区		
5	4	0	1787	4	2		4	交通区		
6	5	1049	2127	12	4		5	公园绿地区		
7	6	1647	2728	6	2					
8	7	2883	3617	15	4					
9	8	2383	3692	7	2					
10	9	2708	2295	22	4					
11	10	2933	1767	7	4					
12	11	4233	895	6	5					
13	12	4043	1895	14	1					
14	13	2427	3971	2	1					
15	14	3526	4357	7	4					
16	15	5062	4339	5	4					
17	16	4777	4897	8	1					
18	17	5868	4904	16	4					
19	18	6534	5641	6	1					

图5-7　　mydata.xls 明文信息

图5-8　　主机打开 mydata.xls 提示

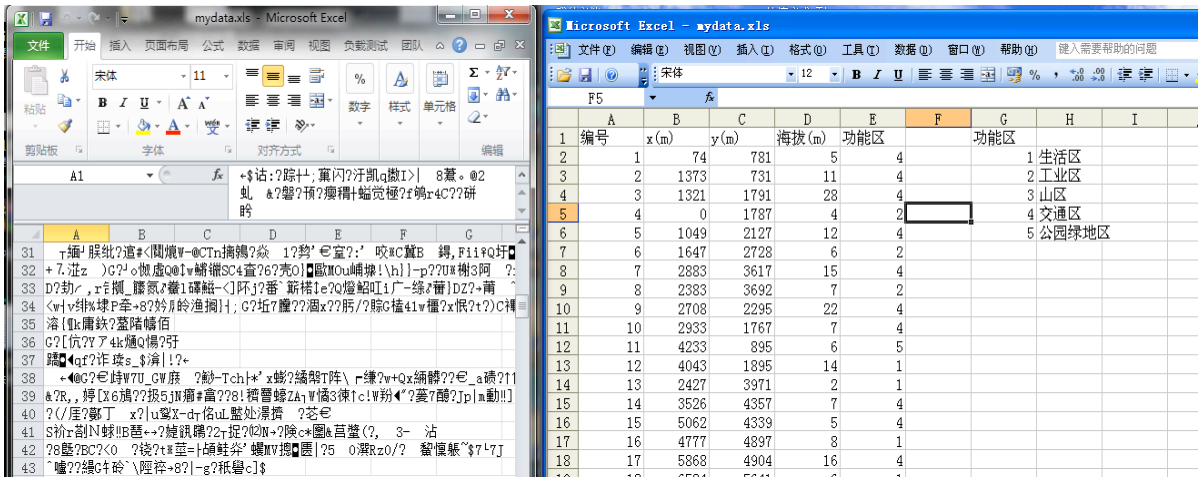


图5-9　　主机和 VMware 同时打开 mydata.xls

（3） POWERPNT.EXE

操作过程：1.新建 myppt.ppt 文档；2.输入文档内容；3.保存 myppt.ppt 文档；4. 将虚拟机中的 myppt.ppt 文档复制到物理机中；5.打开 myppt.ppt 文档，提示“无法打开”；全部过程符合预期要求，达到了保护文档的目的。

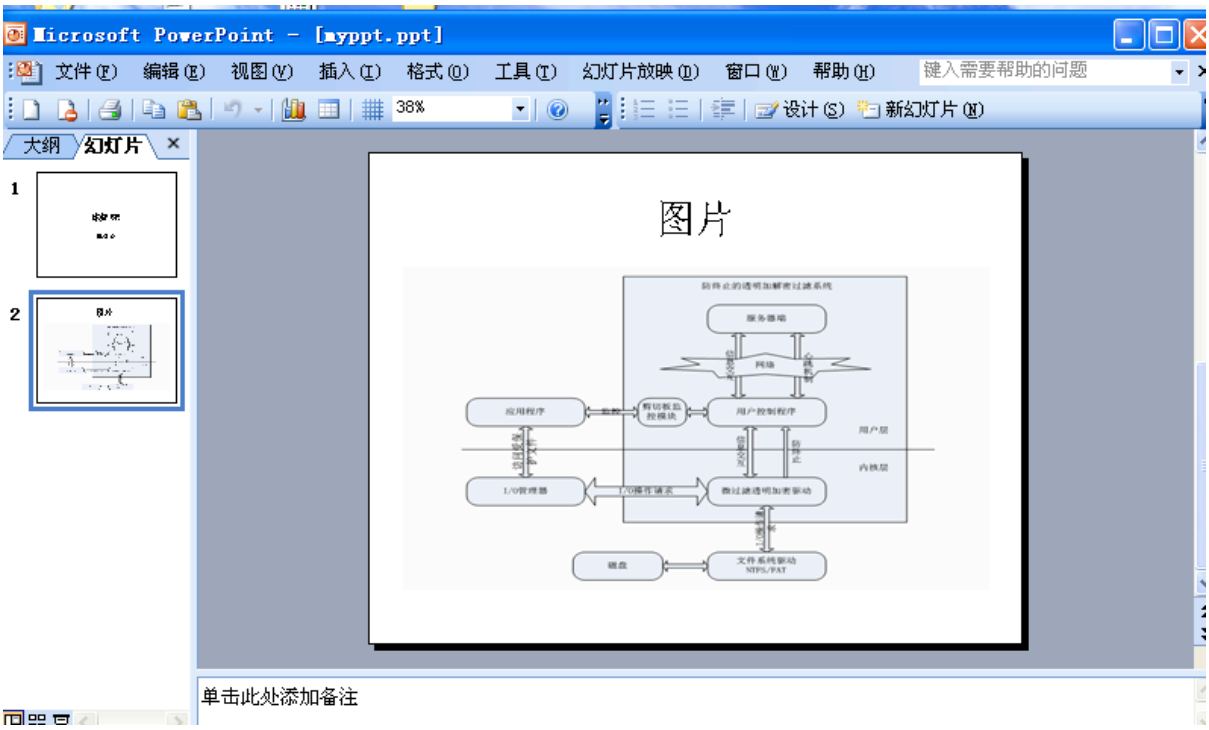


图5-10　myppt.ppt 文档的明文内容

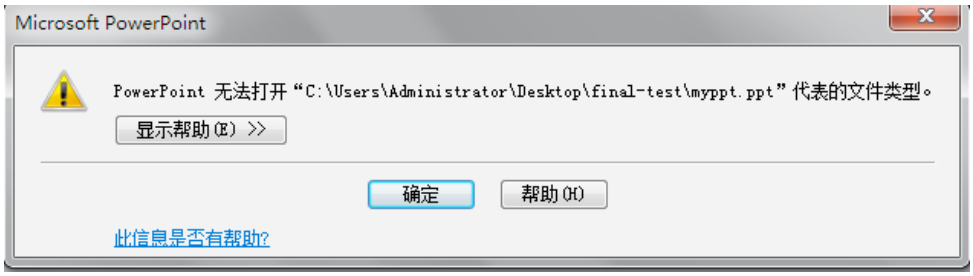


图5-11　物理机中打开 myppt.ppt 失败

（4） NOTEPAD.EXE

操作过程：1.新建 mytext.txt 文档；2.输入文档内容；3.保存 mytext.txt 文档；4. 将虚拟机中的 mytext.txt 文档复制到物理机中；5.打开 mytext.txt 文档，只能得到密文文本，无法获知明文内容；7.此时虚拟机可以正常对 mytext.txt 文档进行操作。全部过程符合预期要求，达到了保护文档的目的。

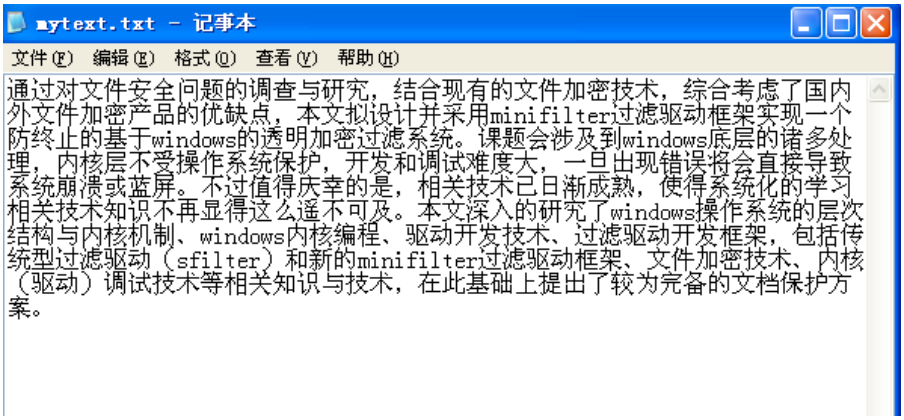


图5-12　mytext.txt 文档的明文内容

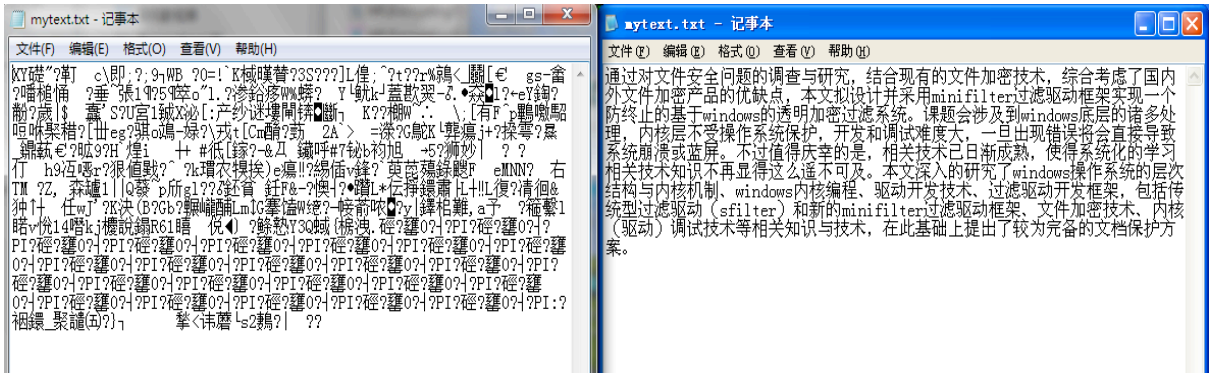


图5-13　主机和 VMware 同时打开 mytext.txt

（5） 非机密进程不能机密进程受保护文档

测试过程：1.在服务器端将 WINWORD.EXE 设置成非机密进程；2.使用非机密进程 WINWORD.EXE 打开受保护文档 mytext.txt，获得密文；3.使用机密进程 NOTEPAD.EXE 打开受保护文档mytext.txt，获得明文。整个测试过程符合预期要求，非机密进程无法打开受保护文档，起到了保护的目的。

图5-14　将 WINWORD.EXE 设置成非机密进程

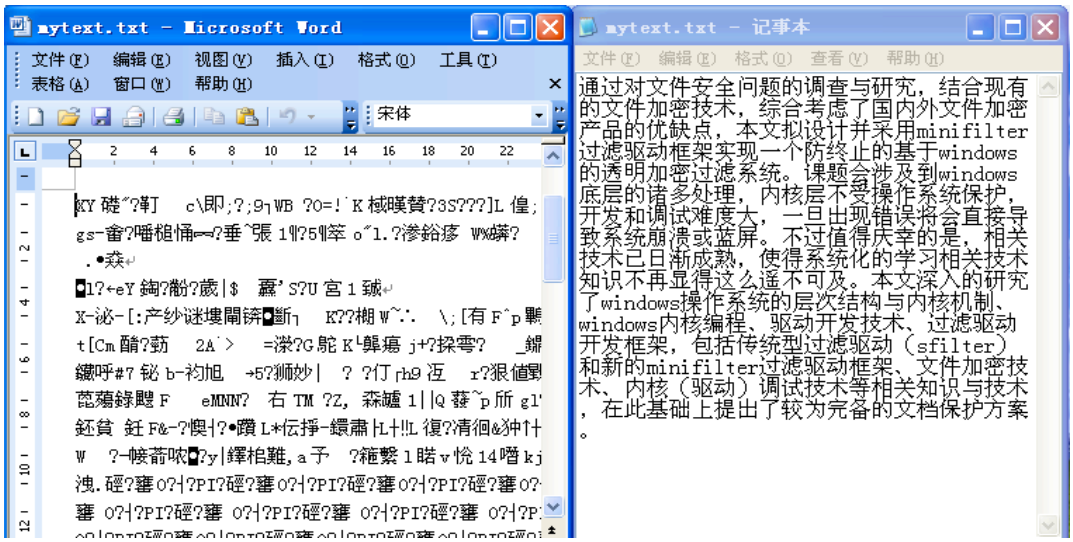


图5-15　非机密进程 WINWORD.EXE 打开机密受保护文档 mytext.txt

（6） 心跳机制

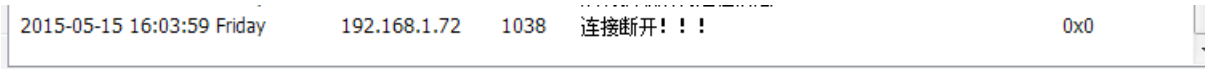


图5-16　服务器报警，192.168.1.72 连接断开

（7） 剪切板监控

测试过程：1.在服务器端将 NOTEPAD.EXE 设置成非机密进程；2.在客户端新建立 testHook.txt，输入明文信息“我是非机密文档”；3.在客户端建立 testHook.doc 文档，输入明文信息“我是机密文档”；4.试图将 testHook.txt 中的“我是非机密文档”内容复制到 textHook.doc 中，操作成功；5.试图将 testHook.doc 中的“我是机密文档”复制到 testHook.txt 中，操作失败，提示“粘贴操作非法！！！”；6. 试图将 testHook.txt 中的“我是非机密文档”内容复制到 testHook.txt 中，操作成功；8. 试图将 testHook.doc 中的“我是非机密文档”内容复制到 testHook.doc 中，操作成功。由测试结果可知，从受保护文档中复制内容是不允许的，这就防止了用户通过复制/剪切操作带走机密数据，而其他情况都不会影响机密数据的安全性，操作合法。测试结果符合预期要求，达到了保护机密数据的目的。

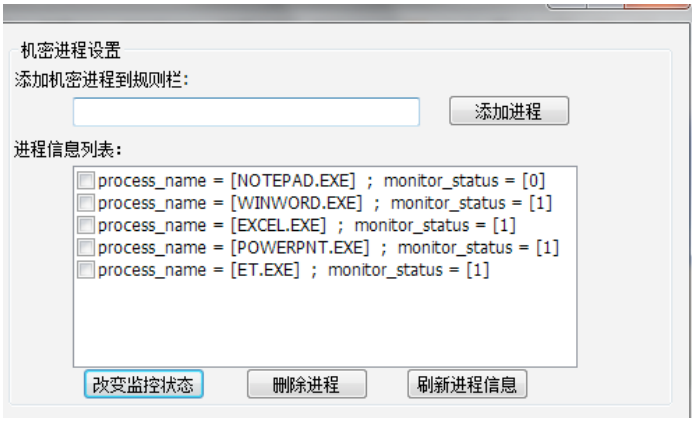


图5-17　将 NOTEPAD.EXE 设置为非机密进程

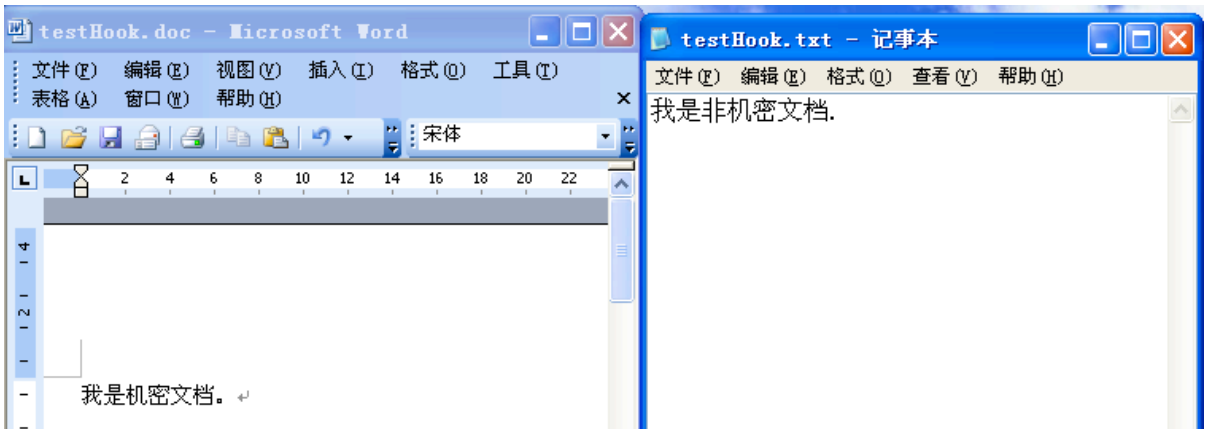


图5-18　测试文档初始状态

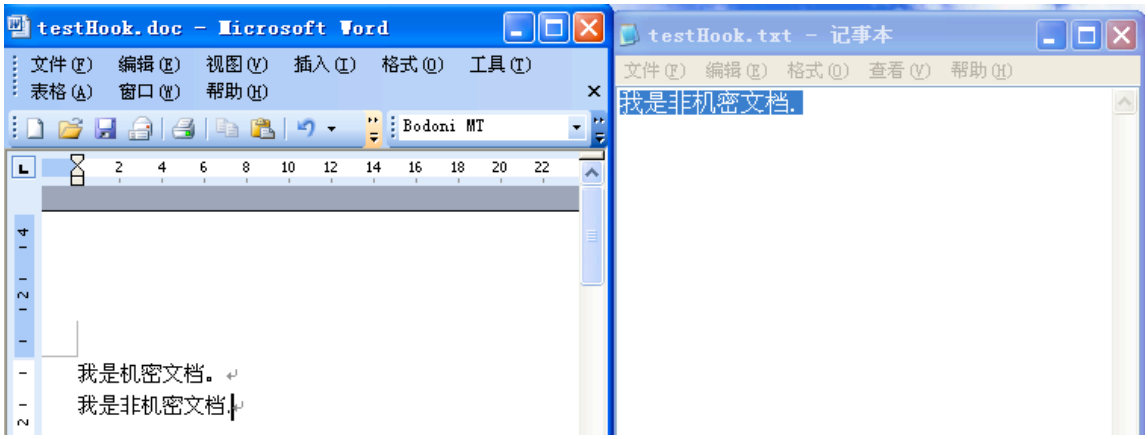


图5-19　从 testHook.txt 到 testHook.doc 的复制粘贴操作，合法

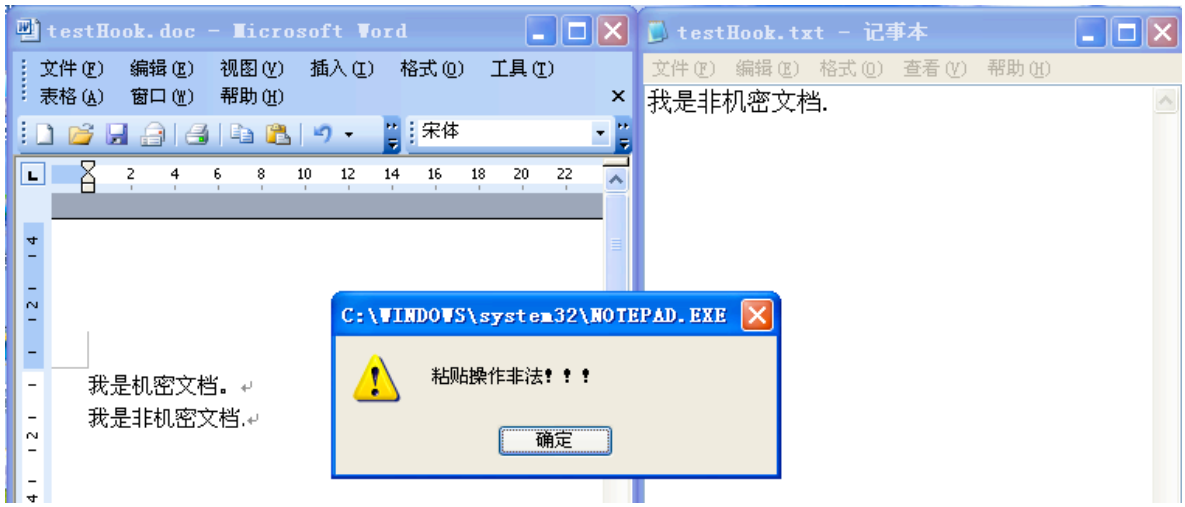


图5-20　从 testHook.doc 到 testHook.txt 的复制粘贴操作，非法

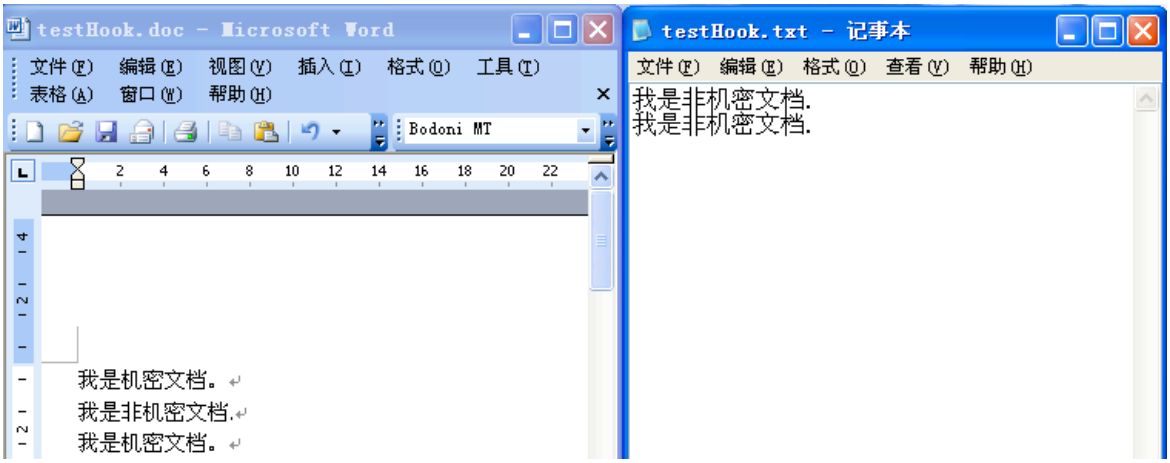


图5-21　testHook.doc 和 testHook.txt 到自身的复制粘贴操作，合法

（8） 进程防终止

图5-22　打开任务管理器试图强制关闭受保护进程

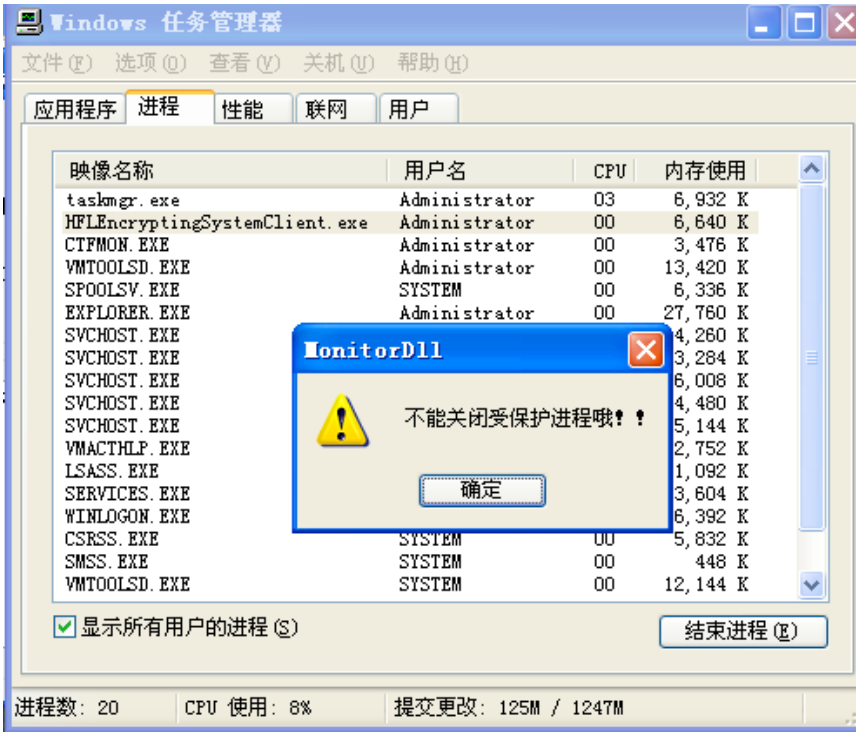


图5-23　强制关闭受保护进程失败

5.4 本章小结

本章对透明加密过滤系统进行较为全面的测试，由于测试过程多为重复，因此只给出具有代表性的测试过程。经过测试本系统可以正常的运行在 Windows XP 和 Windows 7 操作系统上，实现对 NTFS 和 FAT 文件系统上的机密进程相关的文档保护。理论上系统可以兼容 Windows 的系列版本，以及多种应用程序进程的文档操作，但本文没有对更多的情况进行测试。

第六章 总结与展望

随着计算机的高度发展，使得自动化办公已经成为了生活中不可或缺的一部分。这使得企业、单位和个人每日都需要操作大量的文档，其中不乏组织机密以及个人隐私。目前网络的安全方面，主要由防火墙技术和入侵检测技术加以保证，但是对于内网中的文档安全仍然是一个亟待解决的问题。如何保证文档在传输、存储、共享过程中的安全一直是一个热门话题，许多公司常常通过禁用公司的 USB 接口，或者移动设备来防止员工将机密文档带走，以防信息的泄露，但这无疑给工作造成了极大的不便。在这样的背景下，透明加密技术应允而生。本文对相关问题的背景以及国内外研究情况进行了全面的分析，拟定了较为全面的文档安全解决方案，并且对相关技术进行了深入的研究与学习，在此基础上主要参考了微软相关的开发文档以及 WDK 的示例源码，实现了防终止的基于 Windows 透明加密过滤系统。本系统主要的功能特点有：

1、 实现了对受保护文档的透明加解密功能。
2、 系统由服务器端、客户端应用层控制程序、客户端内核层透明加密过滤驱动三个主要模块构成，各模块之间相互制约，相互监控，为保证文档安全提供了很好的支持。
3、 采用多个文件密钥，这可由管理员进行设置，并且将密钥的摘要存放到受保护文档的加密标识中，这相对于单一密钥的安全性要高。
4、 深入的研究了进程防终止技术，对本系统的进程进行了保护，降低系统被恶意关闭的危险，提高了系统的安全性。在服务器端与客户端之间加入心跳机制，实现对客户端在线情况的监控。
5、 结合用户层的安全技术，实现对剪切板的监控，防止用户通过剪切板将机密信息拷贝带走，进一步增强系统的安全性。
6、 系统对用户的行为进行较为合理的跟踪审计，此外保证系统具有较好的稳定性、灵活性、方便性、健壮性。

本文已经基本达到预期目的，并实现了性能较好的原型系统。不过由于时间有限，系统仍存在一些不足，有待进一步的改进与完善。主要有以下几个方面：

1、 本系统基本上只为内网文档安全提供了保障，内网通过 IP 地址加口令进行认证。但这对于存在出差办公需求的用户显得不是很方便。虽然在出差情况下也可以通过网络进行口令认证，启动移动设备的加解密环境，但在没有网络的情况下，这个问题就无法解决。针对这样一个问题，本文拟定解决方案是，指定文档授权机制，在授权范围和时间内，用户可以通过单位分发的 USB Key 启动系统加解密环境，进行正常的工作。
2、 防止进程欺骗。在编写驱动代码时，主要是在 PEB 中获取进程名来判断当前进程是否

为机密进程。但如果有恶意用户将非机密进程名改为机密进程名，系统就无法鉴别，而将其当做机密进程来处理，这将可能导致受保护文档外泄。针对此问题，本文拟定解决方案为：计算机密进程版本信息的摘要值，若企业使用的应用程序是由管理员统一版本的，则这是一个唯一值，然后根据对可执行文件的完整性验证（HASH 验证）来判断当前进程是否为真实的机密进程。

3、 系统休眠处理。本文拟定解决方案为：系统进入休眠时，关闭所有打开着的机密文件对象并清除系统缓冲。

参考文献

[1] Zhang X, Liu F, Chen T, et al. Research and Application of the Transparent Data Encpryption in Intranet Data Leakage Prevention[C]//Computational Intelligence and Security, 2009. CIS'09. International Conference on. IEEE, 2009, 2: 376-379.

[2] 徐慧,孙世佳.基于 Minifilter 的实时备份系统的研究与实现.科技论坛.2012.01

[3] Mueller,Milton.Internet Security and Networked Governance in International Relations.INTERNATIONAL STUDIES REVIEW,Vol15,No.1,2013.

[4] Microsoft Corporation. Filter Driver Development Guide: File System Filter Manager.

[5] Chang C, Liang M, Kou H, et al. A Review of Encryption Storage[J]. Information Technology Journal, 2010, 9(7): 1517-1520.

[6] Zhao Yong,Liu Jiqian,Han Zhen,Shen Changxiang,"The Application of Information Leakage Defendable Model in Enterprise Intranet",In: Journal of Computer Research and Development,pp761-767 2007 44(5).

[7] 王全民,何明,苗雨等.基于微过滤驱动的文件透明加解密系统的研究与实现[J].计算机安全,2014,(3):28-32.DOI:10.3969/j.issn.1671-0428.2014.03.007.

[8] Zizzi S.Encrypting File System[EB/OL].http://en.wikipedia.org/wiki/Encrypting_File_System,2006

[9] 黄革新.Windows 加密文件系统核心技术分析[J].电脑与信息技术,2005,13(4): 1-4

[10] Jee Hun Park,Kyoung Nam Ha,Suk Lee and Kyung Chang Lee."Performance Evaluation of NDIS-based four-layer architecture with virtual scheduling algorithm for IEEE 802.lib" International Conference on Control,Automation and Systems.Oct.17-20,2007 in COEX,Seoul,Korea.Introduction to NDIS,MSDN Library.

[11] Tang Dongming,Li Juguang,Lu Xianliang.Efficient Packet Capture onWindows CE.Intelligent Control and Information Processing(ICICIP),2011 2nd International Conference on 25-28 July 2011,Volume(1):100-114.

[12] 何明星,林昊.AES 算法原理及其实现.计算机应用研究,2002,12.

[13] 谷利泽,郑世慧,杨义先.现代密码学教程[M].北京:北京邮电大学出版社,2009.

[14] 李珠峰.Windows 系统中基于文件过滤驱动的文件动态访问控制技术研究[J].电脑知识与技术,2012,08(9):2045-2047.DOI:10.3969/j.issn.1009-3044.2012.09.027.

[15] 杨少吟.基于微过滤驱动的透明加密文件系统的设计与实现.华南理工大学硕士学位论文.2013

[16]李诗松,陈伟,陈运等.Windows 平台下软件自身防护关键技术[J].计算机系统应用,2012,21(4):264-267.DOI:10.3969/j.issn.1003-3254.2012.04.060.
[17]Rajeev Nagar.Windows NT file system internals:a developers guide[M].Cambridge:O'Reilly,1997.
[18]Zizzi S.Architecture of Windows NT[EB/OL].http://en.wikipedia.org/wiki/Architecture_of_Windows_NT,2014
[19]张帆,史彩成等.Windows 驱动开发技术详解.北京:电子工业出版社 2009/2
[20]谭文,杨溟,邵坚磊.寒江独钓:Windows 内核安全编程[M].北京:电子工业出版社,2010:223-271.
[21]宋永军.基于文件系统过滤驱动的透明加解密系统的研究与实现.北京邮电大学硕士学位论文,2012
[22]OSR Online.http://osronline.com/
[23]Mirosoft MSDN.Windows.HardWare Dev Center.Develop. https://msdn.microsoft.com/en-us/library/windows/hardware/ff540402%28v=vs.85%29.aspx
[24]Shen Jianfang,Cheng Lianglun,Fu Xiufen. Implementation of program behavior anomaly detection and protection using hook technology [C].Kunming: International Conference on Communications and Mobile Computing,2009.
[25]徐江峰,邵向阳.基于 HOOK API 技术的进程监控系统设计与实现[J].计算机工程与设计,2011,04:1330-1333.
[26]Jeffrey Richter.Windows 核心编程[M]:清华大学出版社,2008.
[27]谷利泽, 郑世慧, 杨义先. 现代密码学教程[M].北京:北京邮电大学出版社, 2009.
[28]何明星,林昊.AES 算法原理及其实现.计算机应用研究, 2002,12.
[29]王煦,朱震,苗启广等.文件头中存储加密标识技术的研究与实现[J].计算机工程与设计,2012,33(10):3746-3750.DOI:10.3969/j.issn.1000-7024.2012.10.018.