毕业论文（设计）

设计(论文)题目 基于 Minifilter 框架的双缓冲透明加解密驱动

姓　　名： hkx3upper

学　　号：

学　　院：

专　　业： 计算机科学与技术

年　　级 2018 级

指导教师：

2022 年 05 月 10 日

目　录

摘　要

为防止数据外泄，公司通常会对重要数据进行加密。但这样会造成数据使用的不便利，为此，本课题设计一个基于 minifilter 的双缓冲透明加解密系统，实现对目标扩展名文件的加密保护功能，在保证数据安全性的前提下，方便数据的使用。

本课题通过一种 Windows 文件系统的 minifilter 过滤驱动，将重要文件加密后存在磁盘中，并且划分进程为授权进程和非授权进程。授权进程使用明文缓冲，非授权进程使用密文缓冲，进程本身察觉不到使用的是明文或密文。加密解密使用块密码算法 AES-128 ECB 模式，采用密文挪用的方式填充和块大小不对齐的明文。

本课题对 Office 类文件进行单独处理，从而适配 word，ppt，excel 等文件；注册相关的进程回调，校验进程代码段的完整性，实现对于进程的控制与保护；增加特权加密和特权解密功能，实现单独加密或解密文件。

公司在所有涉密电脑上安装本软件，设置相同解密密钥，文件会在修改操作以后自动加密。授权进程可以正常使用文件，而当文件经非授权进程以密文形式传送后，安装本软件的电脑可以透明加解密文件，未安装的电脑只能获得密文。

关 键 词

文件微过滤驱动；双缓冲；透明加解密；密文挪用

Abstract

To prevent data leakage, companies usually encrypt important data. However, it is inconvenience for the process to access data. Therefore, the thesis designs a double-cache transparent encryption and decryption system based on minifilter to encrypt target file with specific file extension. On the premise of ensuring data security, it is easy for the process to use data.

The thesis realizes a minifilter driver of Windows file system to encrypt important files and divide the processes into authorized processes and unauthorized processes. In addition, authorized processes use plaintext cache and unauthorized processes use ciphertext cache, and the process is unaware whether it is plaintext or ciphertext. AES-128 ECB mode is adopted to encrypt and decrypt file, and the ciphertext stealing method is used for padding the plaintext that is not aligned with the block size.

The Office files are processed separately, so as to adapt to word, ppt, excel files. Process notify routines are registered to control the process and the integrity of the process text segment is checked to protect the process. The paper use privilege encryption and privilege decryption to encrypt or decrypt files individually.

The project is installed on all secret-involving computers and the same decryption key is set on these computers. Files will be automatically encrypted after the modification operation. The authorized process can use the file normally, and when the file is transmitted in ciphertext by the unauthorized process, the computer with this software installed can transparently encrypt and decrypt the file, while the computer without the installation can only obtain the ciphertext.

Keywords

Minifilter; double cache map; transparent encryption and decryption; ciphertext stealing

一、 绪论

（一）课题研究背景和意义

为防止员工将代码以及数据外泄，公司通常会选择加密敏感数据。如果每次都提前将数据加密，使用时再解密，这种方式过于复杂和麻烦，而且也不能很好的保证数据的安全。

现在通常使用一种透明加密[1]的软件，这种软件可以使具有目标扩展名（在 minifilter 中设置，比如.txt，.docx 等）的文件在修改操作以后自动加密。透明加密是指员工正常的办公软件察觉不到数据已经被加密，而非办公软件，尤其是可能导致外泄的通讯软件等，就会无法解密而得到密文。在公司电脑和员工的办公电脑上同时安装软件，设置相同解密密钥，软件也可以保证已加密的代码和数据在办公电脑之间传输后的正常使用。

一般的透明加密软件会采用 AES，SM4 等块密码算法，而待加密文件的大小有时和块密码算法的块大小是不对齐的，这种情况下往往需要给文件加一块填充。但是在 Windows 文件系统内，minifilter 驱动为这一块填充而扩展文件大小时可能导致文件系统死锁的发生，想要避免死锁，就要额外处理文件。

基于上面的分析，本文的研究重点是使用微软的 minifilter 驱动[2]（文件微过滤驱动）模型，设计一个双缓冲[3]透明加密解密过滤驱动，同时为了解决在使用块密码算法加密文件时，待加密文件的大小和块密码算法的块大小不对齐的情况，本文使用密文挪用[4]的方法对文件进行处理。另外设计对文件的单独加密、解密操作，以及对进程的权限管理。

（二）国内外研究现状

国内外公司最初是采用传统文件驱动[5]实现的，当时微软并未在 Windows 的文件系统内添加 minifilter 组件。而传统文件驱动需要做很多额外的重复性的工作，使软件过于复杂，稳定性大打折扣。最开始关于缓冲的处理方式是在授权进程和非授权进程之间刷新并清除缓冲[6]，保证非授权进程无法看到缓冲中的明文，但是这样导致整个系统非常的不稳定。后来国内有很多公司在 minifilter 的基础上设计了 Layered fsd（分层文件系统驱动），通过单独设计一个分层的文件系统，创建两个 fcb（文件控制块），来实现双缓冲的功能，但这样要实现的代码量几乎与实现一个文件系统驱动的无异。

关于待加密文件的大小和块密码算法的块大小不对齐的情况，一般的解决方法是在应用程序本身修改文件大小时，扩展文件大小，或者在缓冲写操作时扩展文件大小。无论怎样，这些操作影响了系统的效率，而且对于文件本身的数据安全性和完整性来讲也不合理。

（三）本文研究内容

本文深入研究了 Windows 文件系统、缓冲管理器和内存管理器的协同方式，设计了一个双缓冲、密文挪用加解密的透明过滤驱动。

1.本文使用双缓冲，授权进程和非授权进程分别使用明文和密文缓冲，授权进程允许修改明文缓冲，非授权进程不允许修改密文缓冲。

2.本文使用 StreamContext[7]存放驱动运行时的文件信息，使用文件标识尾的方式，在文件的尾部 4KB 储存文件所需的解密信息；

3.加密算法使用 AES-128 ECB 模式，并且使用密文挪用（Ciphertext stealing）的方法，处理块大小不对齐的明文需要填充（padding）的问题；

4.本文的 Write（写操作）和 Read（读操作）使用 SwapBuffers[8]的方式进行透明加密解密；

5.对文件的特权加密和特权解密使用重入（Reentry）的方式，让 minifilter 重新进入自己的加密或解密代码中，使驱动加密解密文件；

6.为了适配 office 文件，本文在文件 FileRenameInformationEx 和 FileRenameInformation 时做特殊处理，因此可以自动加密解密 docx，doc，pptx，ppt，xlsx，xls 等使用 tmp 文件重命名方式读写的文件；

7.关于进程的控制与保护，本文使用双向链表存储进程策略，并注册相关进程回调函数，同时定时扫描授权进程代码段（.text 段）的完整性，实现进程的控制与保护。

（四）本文内容组织

本文一共有五部分，每一部分的摘要如下：

第一部分为绪论，介绍国内外的研究状况。

第二部分为问题描述与相关方法，对重要概念进行解释，描述项目的组成部分，简述双缓冲与密文挪用的实现方式。

第三部分为本文核心方法，对本文的研究内容以及项目实现做了详细说明。

第四部分为实验与分析，介绍了测试所需的系统环境，对测试结果进行分析。

第五部分为总结，阐述本文的总体设计和不足。

二、 本论

（一）问题描述与相关方法

1. 重要概念

本文使用了 Fcb，FileObject，IRP，缓冲，NonCachedIo 等概念，在正文开始之前，对这些概念进行描述。

（1）Fcb：文件控制块，由文件系统驱动维护的一块结构体，其中保存文件的一些信息，通常情况下一个文件只有一个 fcb。

（2）FileObject：文件对象，类似于文件的一个打开实例，一个文件的 fcb 可以对应多个 FileObject。文件的打开操作、写操作和读操作等都是操作的 FileObject。由 IO 管理器创建并初始化，由文件系统驱动填写 FileObject 一些文件的信息。

（3）IRP：即 IO 请求数据包，应用程序打开、读写、关闭文件时，通过系统调用进入操作系统内核，由 I/O 管理器做一些参数检查的工作，再把参数封装成 IRP 包，传到文件系统驱动中，文件系统驱动处理完（比如从磁盘或 Cache 中读取数据）以后，再将一些返回值填到 IRP 中，回传到 I/O 管理器。

（4）缓冲[9]：为了提高文件系统读写文件的效率，通常并不是每次读写都要去和磁盘交互，所以缓冲管理器维护了一块内存，由缓冲管理器去判断是否进行预读操作或者延迟写操作。

（5）NonCachedIo：非缓冲 IO，就是文件系统驱动不建立缓冲，不从缓冲中读取已有的数据，直接去磁盘中读。写操作则直接写入到磁盘中。该标识要求操作的偏移和大小都要和扇区大小对齐。
（6）CachedIo：缓冲 IO。读操作是从缓冲中读数据，如果缓冲中没有，就建立缓冲，并且由缓冲管理器以非缓冲 IO 发送 IRP 到文件系统驱动，从磁盘中读取数据。写操作则写入缓冲中，如果没有缓冲，就建立缓冲。
（7）PagingIo：分页 IO，通常由内存管理器设置该标识，分页 IO 一定是非缓冲 IO，优先级较高。在分页写请求和分页读请求时会提前取独占锁，而且分页写请求不允许扩展文件大小。
（8）内存映射读写是一种直接由内存管理器提供的接口，它读写的内存和缓冲读写的内存是一块内存，内存管理器和缓冲管理器都可以操作这块内存。如果内存中没有数据，同样由内存管理器以非缓冲 IO 发送 IRP 到文件系统驱动，从磁盘中读取数据。写操作同样由内存管理器负责，向磁盘中写入数据。

2. 文件微过滤驱动 minifilter 介绍

Minifilter，即文件微过滤驱动，是 Windows 操作系统文件系统内的一个组件，由 Filter Manager 统一管理。通过在 minifilter 中注册想要监听的事件，它可以过滤包括打开（Create）、读操作（Read）、写操作（Write）等由 IO 管理器[10]发往文件系统的 IRP 请求。

如图 1-1 所示，本次操作是由应用程序发起的缓冲读操作，此时缓冲还未建立，所以 IRP 由 IO 管理器建立，经过 minifilter 发往文件系统驱动，然后文件系统驱动建立缓冲，由缓冲管理器以及内存管理器协同发起非缓冲分页读操作（箭头 6），该请求被 IO 管理器封装为 IRP，经过 minifilter 发往再次文件系统驱动，文件系统驱动从磁盘中读取文件（箭头 7、8、9），然后逐步返回。

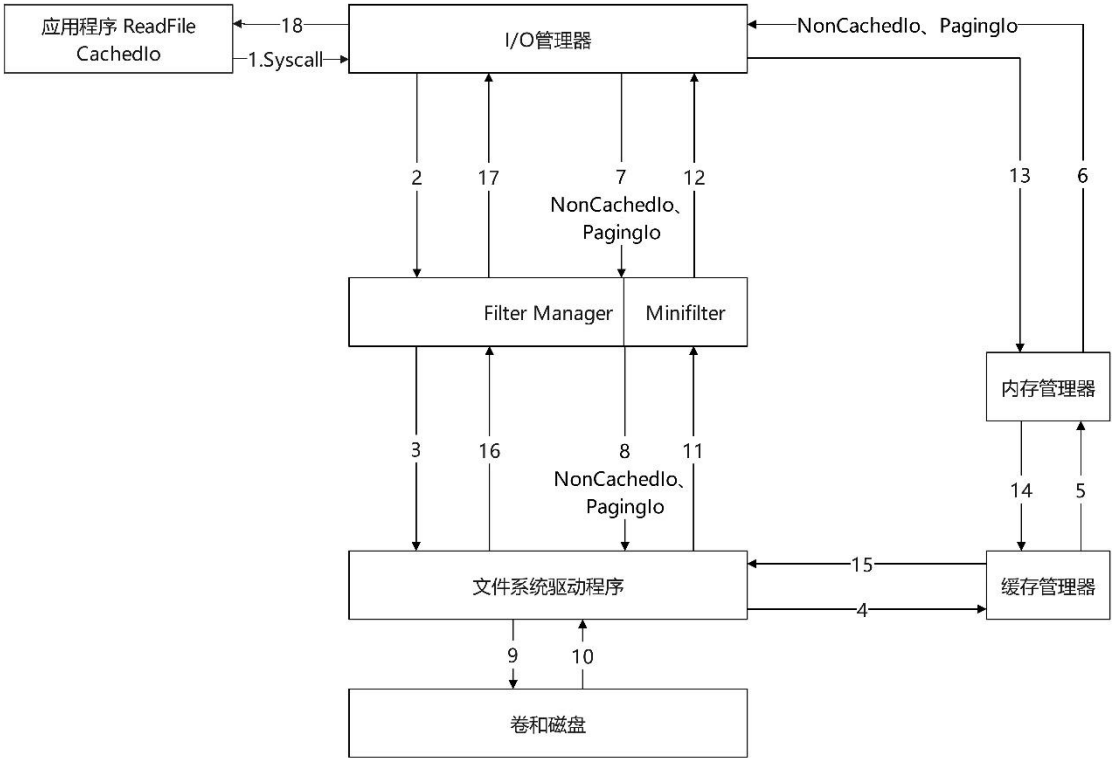

图 1-1 minifilter 在文件系统内的位置（以读操作为例）

可见，每次 IRP 从 IO 管理器发往文件系统驱动时会进入 minifilter 一次，从文件系统驱动返回 IO 管理器也会进入 minifilter，在读操作中，前者称为 PreRead，后者称为 PostRead。

3. 项目的组成部分

如图 1-2 所示，项目分为三个模块：在 Windows 内核的 minifilter 模块，大部分功能实现于 minifilter 中；另外两个在桌面端，分别是特权加密解密模块，负责从桌面发送单独加密或解密文件的命令到 minifilter；进程授权模块，负责添加或修改进程的权限。

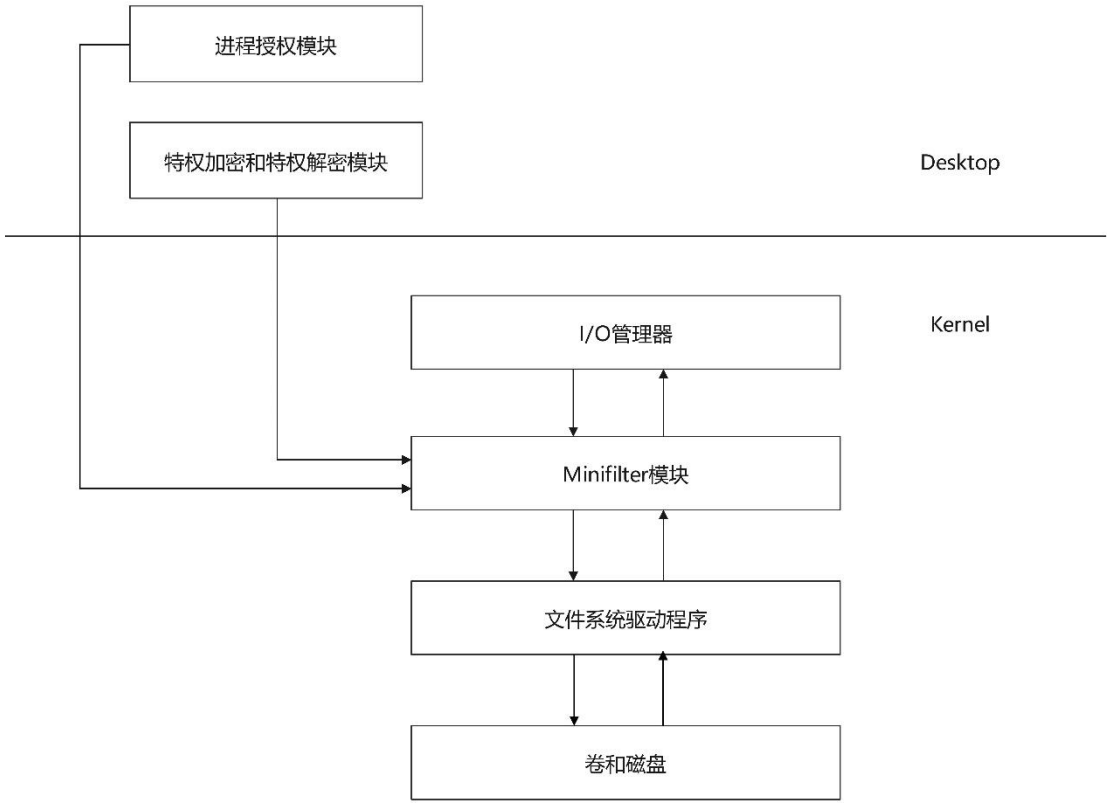

图 1-2 项目的组成部分

4. 双缓冲方法简述

缓冲是缓冲管理器和内存管理器共同维护的一块内存，授权进程一次缓冲读操作后，缓冲中就是明文，此时非授权进程再尝试缓冲读操作，也会读到明文，显然这样是错误的。一种方法是在这两个进程之间刷新并清除缓冲，这样对缓冲过于频繁的操作往往会导致系统不稳定；另一种可以通过两个 fcb（文件控制块）来维护两套缓冲，这样实现方式太麻烦。

本文采用另一种方法，通过将 FileObject（文件对象）中有关缓冲的结构体单独分配一块内存，在这块 minifilter 私有的缓冲结构体内存上建立密文缓冲，原先的文件系统驱动的缓冲结构体上建立明文缓冲，这样的代码量很小，实现的功能也完全相同。

5. 密文挪用方法简述

为解决待加密文件的大小和块密码算法的块大小不对齐的问题，本文在非缓冲写操作加密文件时，块密码使用密文挪用（Ciphertext stealing）的方式，明文不需要因为块对齐而填充，保证了密文和原始文件的大小一致，所以不需要为这块填充而扩展文件大小。在非缓冲读操作时，使用密文挪用解密文件。

但密文挪用也有一些限制，比如需要文件大小大于块密码算法的块大小，而且需要最后两个明文块或密文快相邻。如果一个文件的非缓冲写或读操作的倒数两次操作，正好从文件最后两个数据块之间分隔开，这种情况下，密文挪用无法使用。

本文在文件大小和块大小不对齐时，使用密文挪用的方式加密解密；当文件大小小于一个块时选择其他流式加密算法加密解密；在文件大小和块大小对齐时直接使用块密码算法加密解密；

在非缓冲分页写操作使用密文挪用，倒数两次写操作，正好从文件最后两个明文块之间分隔开时，把倒数第二次写操作的数据减掉一个扇区，保存在 Stream Context 中，倒数第一次写操作的数据之前加上这个扇区，让最后两个块相邻，使用密文挪用的方式对拼接后的数据进行加密；

在非缓冲读操作使用密文挪用，倒数两次读操作，正好从文件最后两个密文块之间分隔开时，两次读操作分别使用非重入非缓冲的读函数将待拼接的密文读出，再使用密文挪用的方式对拼接后的数据进行解密，然后将与本次读操作的偏移和数据大小均相同的明文拷贝回读操作的请求者分配的内存。

6. 使用的工具

（1）使用 Windbg 和 VMware Workstation Pro 中的 Windows 10 虚拟机进行串口调试[11]。
（2）使用 OsrLoader 加载驱动，使用 DebugView 打印驱动输出信息。
（3）项目开发过程中，使用 FileSpy 抓取 IRP，分析 IRP 参数。

（二）核心方法

本文的核心方法一共分为七部分，分别为双缓冲部分的设计，Stream Context 和文件标识尾，密文挪用方式加密解密文件，读和写的 SwapBuffers（交换缓冲区），重入和特权加密解密，重命名方式读写文件，授权进程的控制和保护，本章将对这些核心方法进行详细的描述。

1. 双缓冲部分的设计

双缓冲即使用原来的缓冲作为明文缓冲，新建的缓冲作为密文缓冲。主要涉及到密文缓冲的创建与销毁，以及密文缓冲与明文缓冲的同步问题。密文缓冲不允许下发非缓冲写请求到文件系统驱动，它的非缓冲读请求不解密，因而应用程序会显示密文。

(1)密文缓冲的创建和销毁：

从微软公开的 FastFat 的源码中可以看出，缓冲并不是在 Create 中建立的，而是在 Read 或者 Write 有缓冲需求下建立的，但是本文的密文缓冲为了统一管理，是在 PostCreate 建立的。

Minifilter 在文件 Create 时，首先判断是否为目标扩展文件，否则结束处理，是则判断文件是否已加密，否则会在有写入操作时加密，是则判断当前进程是否为授权进程，若是授权进程则将该 FileObject 指向明文缓冲，明文缓冲

的读操作会解密，写操作会加密；若是非授权进程则将该 FileObject 指向密文缓冲，密文缓冲的读操作不解密，写操作不执行。如图 2-1 所示。

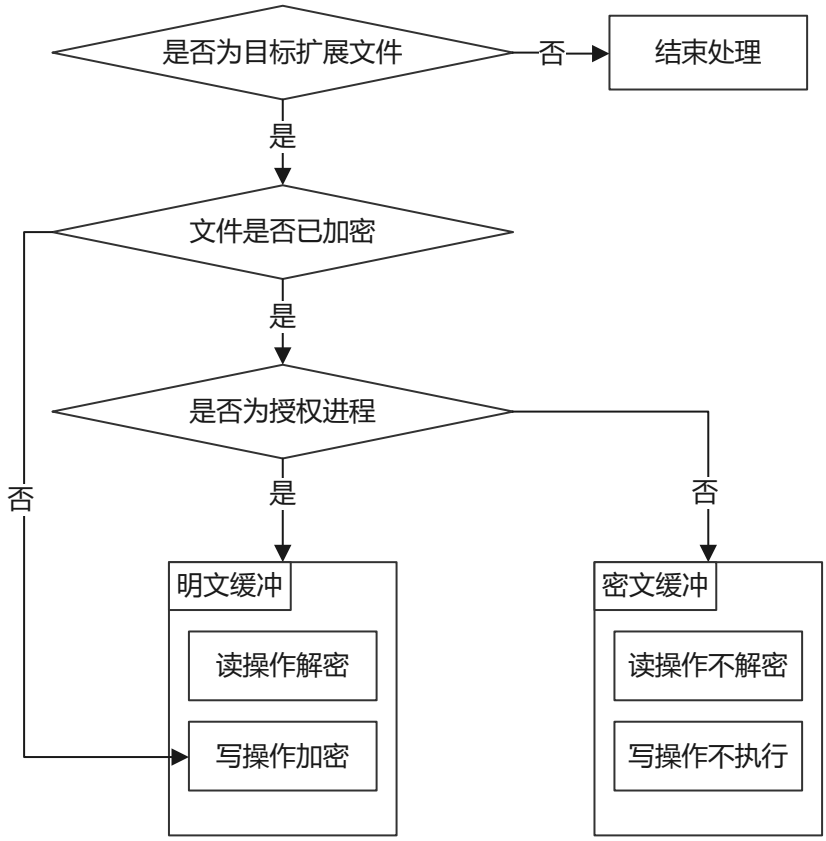

图 2-1 明文缓冲和密文缓冲

① FileObject（文件对象）中有关缓冲的结构如图 2-2 所示。

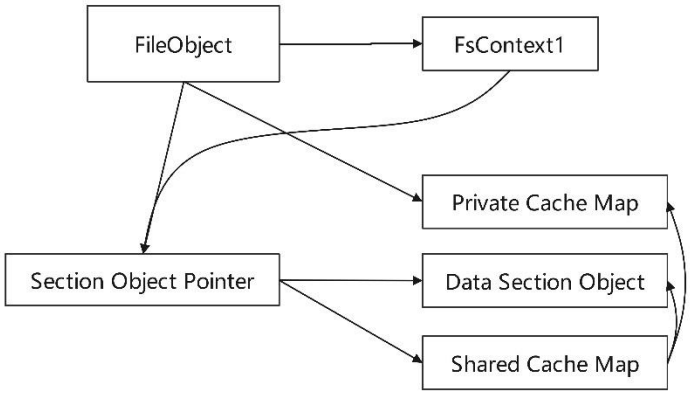

图 2-2 FileObject 中有关缓冲的结构

FileObject->PrivateCacheMap 和 FileObject->SectionObjectPointer 是 FileObject 中和缓冲相关的部分。

SectionObjectPointer 是指向 Fcb 中一块旁视链表分配的 NonPagedPool（非分页内存）的指针，这块内存是一个文件的所有 FileObject 共享的，也意味着同一个文件的所有 FileObject 都使用一个缓冲。SectionObjectPointer 的 DataSectionObject 和 SharedCacheMap 是文件缓冲使用的两个指针，DataSectionObject 即 ControlArea，是 Memory Manager 管理缓冲的部分，SharedCacheMap 是 Cache Manager 管理缓冲的部分。

PrivateCacheMap 是每个 FileObject 私有的，存放着该 FileObject 读操作的历史记录，给预读（Read Ahead）[12]算法提供预读位置和预读长度，它不存放实际的缓冲。我们不需要考虑 PrivateCacheMap，因为它本身指向的内存或者是 FileObject->SectionObjectPointer->SharedCacheMap 中一块提前预备的内存，或者是缓冲管理器新分配的内存，然后插入到对应 SharedCacheMap 的 PrivateList 链表中。

就是说 PrivateCacheMap 和 SharedCacheMap 有着多对一的对应关系。因为我们是在 PostCreate 中开始处理 FileObject，所以此时 PrivateCacheMap 并没有建立，只要我们在缓冲还没有建好时就替换 SectionObjectPointer，那么 PrivateCacheMap 就是与我们之后建立的 Shadow SectionObjectPointer（密文缓冲的指针）相对应。

② 密文缓冲的建立

本文把 FileObject->SectionObjectPointer 指向了 StreamContext 中事先分配的一块内存 Shadow SectionObjectPointer，然后调用 FltReadFileEx，使用缓冲 IO 向文件系统驱动（File System Driver）下发读请求，文件系统驱动会帮我们调用 CcInitializeCacheMap 建立缓冲，然后它会调用 CcCopyReadEx 尝试从缓冲中读数据。

如此一来，密文缓冲就建立好了，存放在 StreamContext->ShadowSectionObjectPointer 中，当非授权进程的其他 FileObject 也想要读写密文缓冲时，我们就把该 FileObject->SectionObjectPointer 指向已经建立好的密文缓冲。

③ 密文缓冲的销毁

通过密文缓冲读写的 FileObject，缓冲管理器都会使用 SharedCacheMap->FileObject 进行读写，即缓冲建立时的原始 FileObject，并增加该 FileObject 的引用数，保证该 FileObject 不会被提前清除。密文缓冲的销毁是由文件系统驱动在 Cleanup 中调用 CcUnitializeCacheMap 实现的，当 SharedCacheMap->OpenCount==0 等条件满足时就会被删除。

(2)密文缓冲和明文缓冲的同步问题：

当非授权进程打开的 FileObject 有读的倾向时，我们使明文缓存（如果有的话）尽快下刷到磁盘中，这样等到密文缓冲建立时，从磁盘中读到的数据是最新的。刷新缓存尽量要从上层去调接口，让文件系统驱动去判断条件，取相关的锁，然后刷新，尽可能地避免自己调用 CcFlushCache，所以这里使用函数 FltFlushBuffers 刷新明文缓冲。

2. StreamContext 和文件标识尾

StreamContext 是 minifilter 驱动运行时为每个文件维护的一块内存，用于存放运行时所需的文件信息，比如文件大小，是否已加密等。文件标识尾是写入文件结尾的一块数据，用于在 minifilter 驱动卸载时静态存储文件的信息。

(1)StreamContext：

StreamContext（流上下文）是我们的 minifilter 驱动为每个文件维护的一块数据，它和 Fcb 类似，不过 Fcb 是文件系统驱动维护的。我们可以在 StreamContext 中定义需要记录的数据，minifilter 卸载时它就被释放了。当然也可以为文件维护一张链表，但是，面对大量的文件时 StreamContext 比链表快，至少它省去了每次遍历查找链表的时间。StreamContext 的结构如下：

```
typedef struct _POC_STREAM_CONTEXT
{

   ULONG Flag;
   PWCHAR FileName;
   /*
   * FileSize 中存着明文==密文大小，因为写进去的尾是 NonCachedIo，所以有扇区对齐，不是紧接着密文写的
   * FileSize 主要是用于隐藏尾部，在 PostQueryInformation 和 PreRead，PostRead 中使用
   * FileSize 会在 PostWrite 中更新，并在 PostClose 中写进尾部，以便驱动启动后第一次打开文件时，从尾部中取出
   */
   ULONG FileSize;

   //明文缓冲
   PSECTION_OBJECT_POINTERS OriginSectionObjectPointers;
   //密文缓冲
   PSECTION_OBJECT_POINTERS ShadowSectionObjectPointers;

   //文件是否已加密
   BOOLEAN IsCipherText;

   //将倒数第二个扇区大小的块存在 StreamContext->PageNextToLastForWrite 中
   POC_PAGE_TEMP_BUFFER PageNextToLastForWrite;

   //读写锁，保证多线程的安全
   PERESOURCE Resource;

} POC_STREAM_CONTEXT, * PPOC_STREAM_CONTEXT;
```

StreamContext 中记录了文件名、文件大小、文件是否已加密、密文缓冲的指针、以及一块用于密文挪用的内存。

(2)文件标识尾：

文件标识尾：我们的 minifilter 加密一个文件以后，需要把一些数据写入已加密的文件，以便之后解密，比如密钥与一块密文的哈希，用于解密时验证密钥的正确性，或者文件的原始大小，以便解密以后对进程隐藏标识尾。

下面简单的列举标识头和标识尾的优缺点。

加密标识头优点是，只要写入标识头文件就是密文，而且头部不会被修改、覆盖；缺点在于既要在读和写时加上偏移跳过头部，而且还要修改 FileSize（文件大小），ValidDataLength（有效数据长度）等去隐藏头，对于文件系统驱动来说，各方面的影响比较多一点。

加密标识尾则不需要在读写时修改偏移，但是，写操作以后，需要重新添加标识尾，因为原始的标识尾被破坏了，所以，不能根据是否有尾去判断文件是否被加密，而且也需要修改 FileSize 去隐藏标识尾。本文使用了标识 IsCipherText 判断文件是否被加密。本文采用加密标识尾的方式，其中记录了文件名、文件大小、文件是否已加密、使用的加密算法等，结构如下：

```
typedef struct _POC_ENCRYPTION_TAILER
{
  //标识符
  CHAR Flag[32];
  //文件名
  WCHAR FileName[POC_MAX_NAME_LENGTH];
  //文件大小
  ULONG FileSize;
  //是否已加密
  BOOLEAN IsCipherText;
  //加密算法类型
  CHAR EncryptionAlgorithmType[32];
  CHAR KeyAndCiphertextHash[32];

}POC_ENCRYPTION_TAILER, * PPOC_ENCRYPTION_TAILER;
```

(3)文件标识尾读取和写入的时机：

标识尾是在 PostCreate 中读取并判断的，因为使用的读操作是非缓冲 IO，所以读偏移和读的数据大小需要和扇区对齐，然后从标识尾中读取相关数据，写入 StreamContext 中。

在非缓冲 IO 的写操作加密文件以后，我们设置 StreamContext->Flag 为 POC_TO_APPEND_ENCRYPTION_TAILER，PostClose 时，如果 Flag 有相应的标识，就写入标识尾，不过这种方式还是有死锁的可能。这里使用单独的函数刷一下原始的明文缓冲，促进文件系统的处理，避免在写入文件标识尾时缓冲管理器也写入数据而导致死锁。

(4)文件标识尾的隐藏：

因为应用程序分配读操作内存的方式不同，文件标识尾的隐藏需要针对不同的应用程序进行适配。

对于记事本程序来说，本文在 PostQueryInformation 把和文件大小（EndOfFile）相关的值都修改成原始大小（存放在 StreamContext 中）。使用如下代码：

```
case FileEndOfFileInformation:
{
   PFILE_END_OF_FILE_INFORMATION Info =
    (PFILE_END_OF_FILE_INFORMATION)InfoBuffer;
   //设置文件的 EOF 为明文大小
   Info->EndOfFile.LowPart = StreamContext->FileSize;
   break;
}
```

对于写字板和 notepad++来说，本文在 PostRead 中修改读取成功的返回值。将返回值 Data->InStatus.Information 设置为去掉文件标识尾的大小。这里是因为 wordpad.exe notepad++.exe 每次会读固定的大小，例如 EEF 等，直到 EOF（文件结尾），然后根据 Read 返回的结果值分配内存。代码如下：

```
if (STATUS_SUCCESS ==
     Data->IoStatus.Status)
{
   if (StartingVbo + Data->IoStatus.Information > FileSize)
   {
      //返回值
      Data->IoStatus.Information = FileSize - StartingVbo;
   }
}
else if (!NT_SUCCESS(Data->IoStatus.Status)
     || (Data->IoStatus.Information == 0))
{

   Status = FLT_POSTOP_FINISHED_PROCESSING;
   goto EXIT;

}
```

3. 密文挪用方式加密解密

对于分块的加密算法，如何在 PagingIo 下扩展文件大小是做透明加密比较重要的问题。对于文件大小本身就和块密码算法的块大小对齐的文件，我们直接加密解密。对于不对齐的文件，这里我们采用密文挪用的方式，这个方式有两个问题需要解决：文件大小小于一个块，不能用这种方式加密；非缓冲 IO 以扇区为单位读写数据，如果最后一次读或写操作的实际数据小于一个块，首先整个文件是需要做密文挪用加密解密的，其次需要倒数第二个数据块，才能正确加密或解密最后一个数据块。

密文挪用理论上可以用在其他块加密的算法中，这里使用的是 AES-ECB 模式，AES-CBC 模式同样可以。

(1) 密文挪用的实现

加密过程如图 2-3 所示，*Pn-1*，*Pn* 是明文，前者的数据大小和块大小是对齐

的，后者是需要填充的；*Cn-1*，*Cn* 是密文，*C'*是 *Pn* 待填充大小的密文；*Pn-1* 经过加密后的密文分为两部分，一部分后移成为 *Cn*，另一部分 *C'*和 *Pn* 补齐，加密后成为密文 *Cn-1*；最后成为 *Cn-1Cn* 的一块数据。

由以上可知，密文数据的长度与明文长度一致。

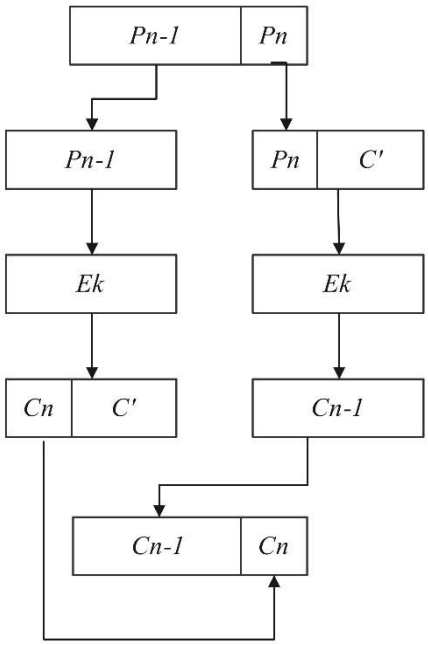

图 2-3 加密过程

解密过程如图 2-4 所示，*Cn-1*，*Cn* 是密文，前者的数据大小和块大小是对齐的，后者是需要填充的；*Pn-1*，*Pn* 是明文，*C'*是 *Cn* 待填充大小的密文；*Cn-1* 经过解密后的明文分为两部分，一部分后移成为 *Pn*，另一部分 *C'*和 *Cn* 补齐，解密后成为明文 *Pn-1*；最后成为 *Pn-1Pn* 的一块数据。可见密文挪用可以正常加密与解密数据。

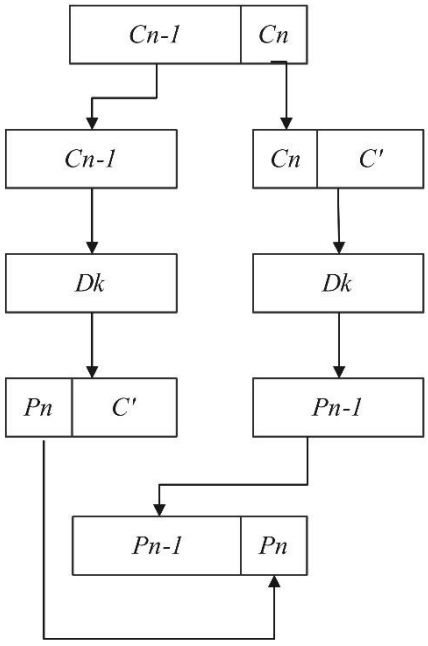

图 2-4 解密过程

通过以上方法就可以使用密文挪用加密解密和块大小不对齐的文件。

(2) 文件大小小于一个块大小时。因为密文挪用要求文件数据大小要大于块大小，所以这里选择使用其他的流式加密算法进行加密解密，不使用密文挪用。加密如图 2-5 所示，解密如图 2-6 所示。

(3) 文件大小和块大小对齐时，可以直接使用块密码算法加密解密，不需要做密文挪用。加密如图 2-5 所示，解密如图 2-6 所示。

(4) 文件大小和块大小不对齐时，且最后两个明文块或密文块相邻时，直接使用密文挪用加密解密。加密如图 2-5 所示，解密如图 2-6 所示。

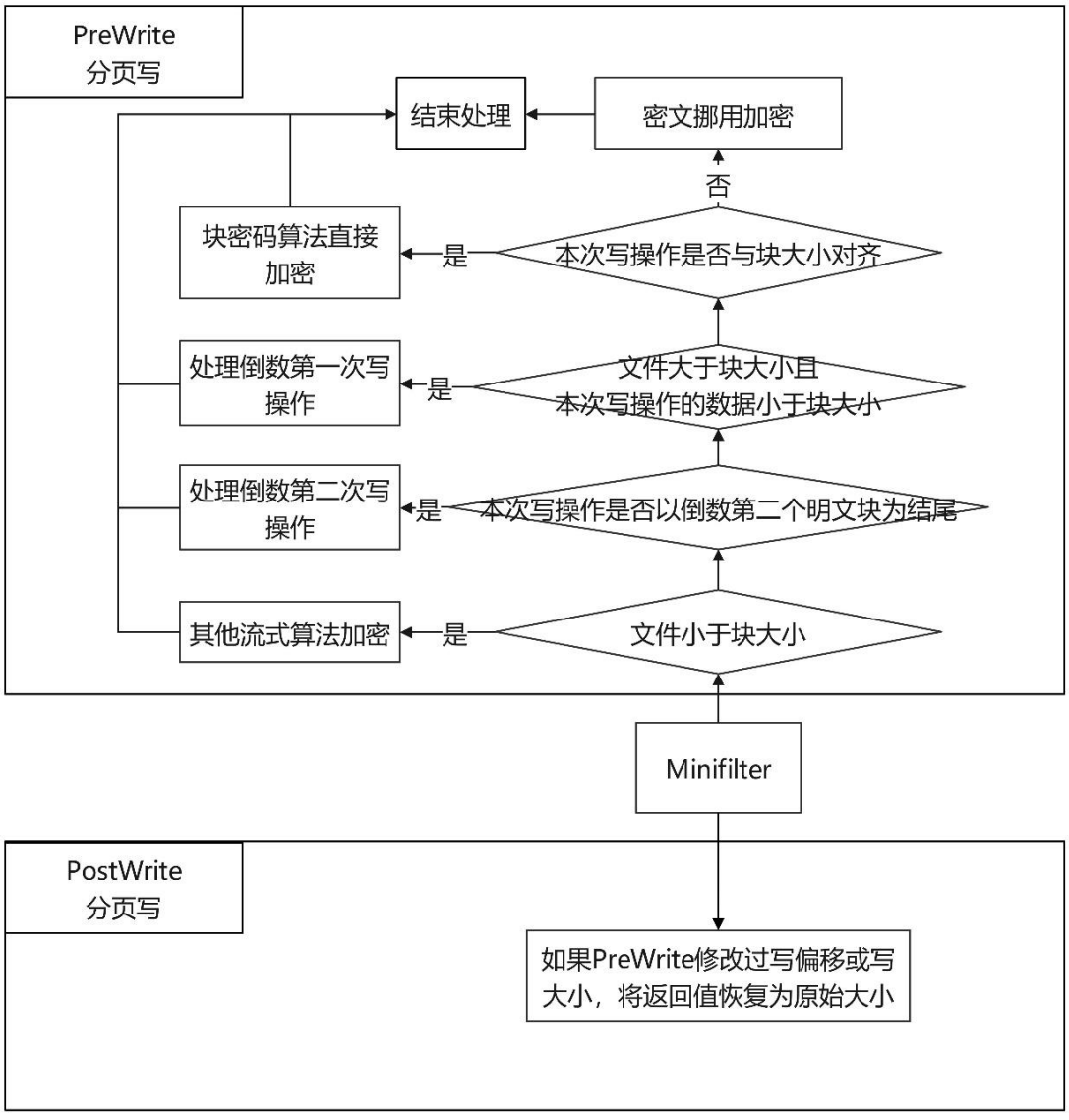

图 2-5 加密过程

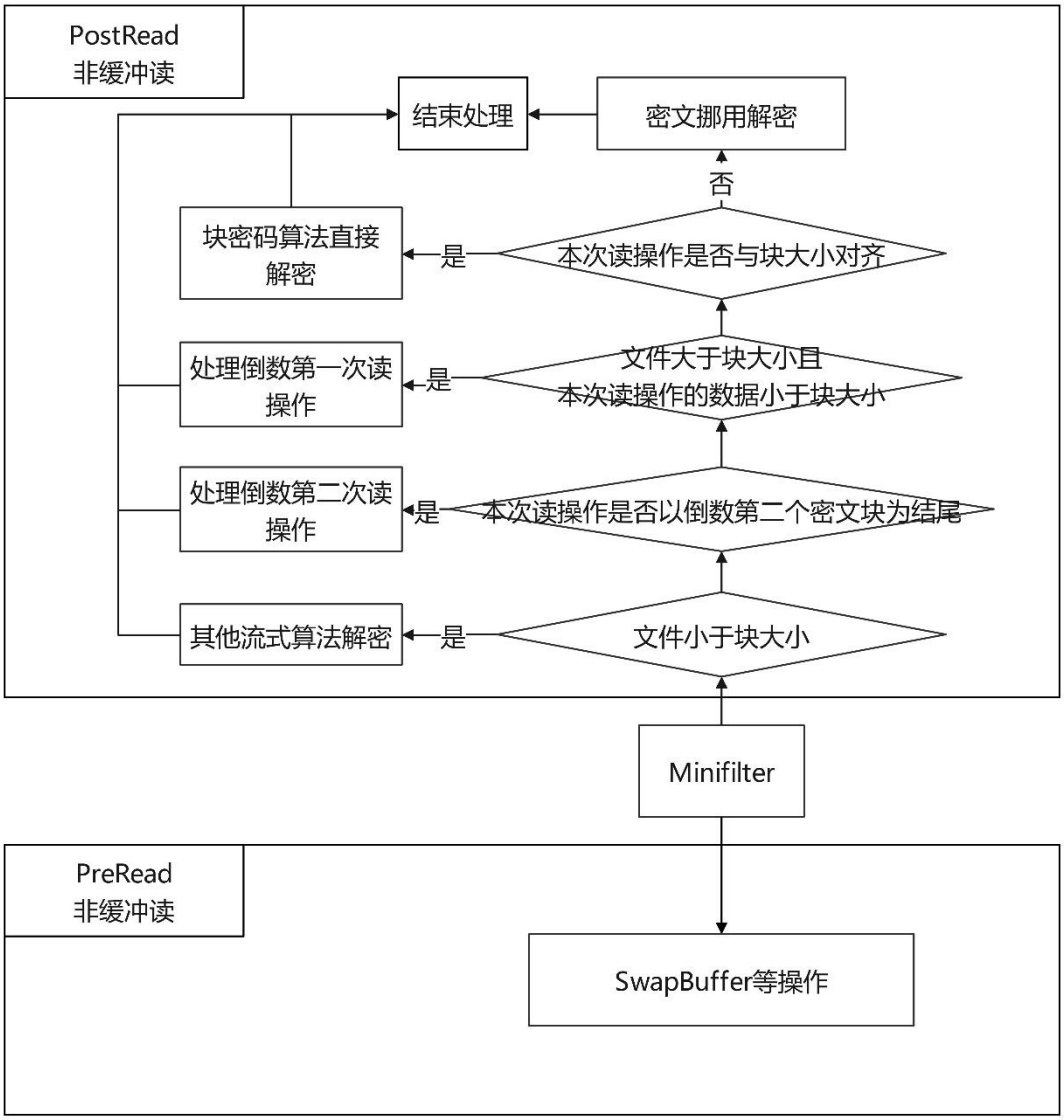

图 2-6 解密过程

(5) 使用密文挪用时，如果一个文件的非缓冲分页写操作的倒数两次操作，正好从文件最后两个明文块之间分隔开。

因为 Windows 文件系统要求非缓冲写操作的数据大小和写偏移必须与扇区大小对齐（这个值通常为 0x200），且不允许在非缓冲分页写操作时扩展文件的大小，所以通常会在缓冲写操作时扩展文件大小或者在写操作之前直接调用函数修改文件大小。驱动程序或者缓冲管理器非缓冲分页写操作时，超过文件大小的那部分数据会被截断。

在这种情况下，如果一个文件的非缓冲分页写操作的倒数两次操作，正好从文件最后两个明文块之间分隔开，此时最后一次的写操作的真正数据大小小于块大小（虽然写操作的大小是扇区对齐的，但超过文件大小的那部分数据被截断了），密文挪用会出现文件的最后两个明文块不相邻的问题。

根据倒数第二次写操作的数据大小是否为一个扇区大小，分为了两种处理方式，分别为图 2-7、2-8 和图 2-9、2-10。处理的方式简单来说就是把倒数第二次写操作的数据减掉一个扇区，保存在 StreamContext 中，倒数第一次写操作的数

据加上这个扇区，让最后两个块相邻，然后使用密文挪用的方式对拼接后的数据进行加密。另外需要分别修改两次的写操作的数据大小和写偏移。

① 倒数第二次写操作大小为一个扇区大小

倒数第二次写操作的数据大小为一个扇区大小；如图 2-7 所示，图中是正常的数据写入，待写入的是一块 0x207 大小的数据，因为非缓冲写操作的要求，这块数据需要和扇区对齐，所以整块被写入的内存大小是 0x400。

这块数据的写入分为了两次，倒数第二次写入的数据大小是 0x200，倒数第一次真正写入的数据大小是 0x7（0x1F9 的后缀会被截断），最后的一个明文块 0x7 和倒数第二个明文块（在其偏移 1F0 处，大小为 0x10）不相邻，所以无法使用密文挪用。

所以使用 StreamContext 将倒数第二次写操作的整个扇区大小的数据保存，并记录偏移和大小。

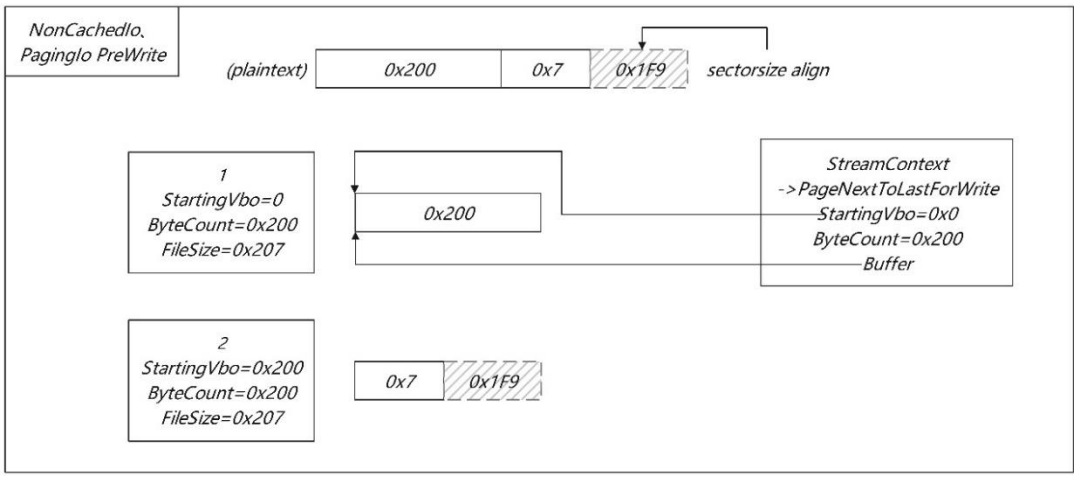

图 2-7 倒数第二次写操作数据大小为一个扇区大小的未修改数据图

如图 2-8 所示，图中是本文处理过的数据，因为倒数第二次写操作没有数据要写入了，直接返回 FLT_PREOP_COMPLETE，返回值 IoStatus.Information 恢复为未修改之前的 I/O 数据请求包中的写操作数据大小。

倒数第一次写操作的数据 0x7 之前加上 *StreamContext* 保存的一个扇区的数据 0x200，使用密文挪用的方式对拼接后的数据进行加密，然后修改写操作的数据大小（*ByteCount*）和写偏移（*StartingVbo*），将密文整体下发到文件系统驱动，由文件系统驱动写入磁盘。

最后在 PostWrite 时，将返回值 IoStatus.Information 恢复为未修改之前的 I/O 数据请求包中的写操作数据大小，避免写操作的请求者接收到错误的返回值，引发不必要的冲突。代码如下：

```
if (0 != SwapBufferContext->OriginalLength)
{
    Data->IoStatus.Information = SwapBufferContext->OriginalLength;
}
```

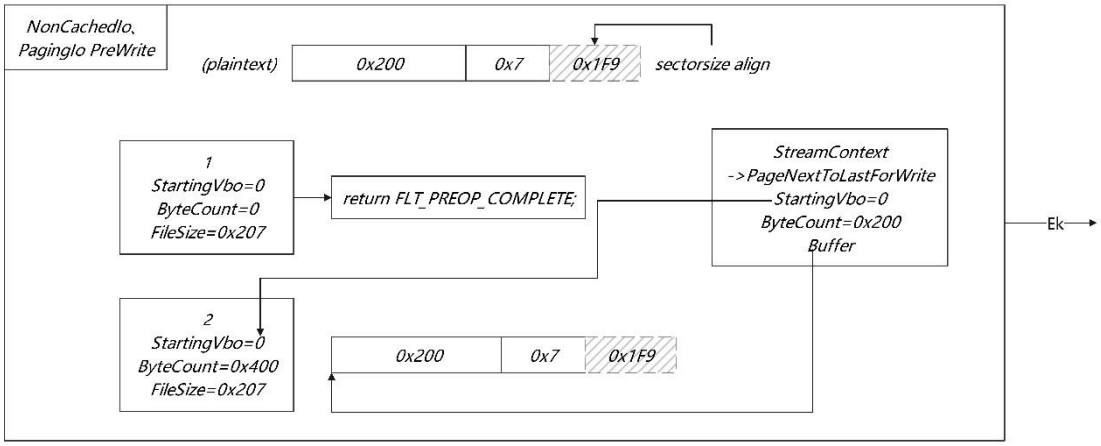

图 2-8 倒数第二次写操作数据大小为一个扇区大小的修改后数据图

② 倒数第二次写操作的数据大小大于一个扇区大小：

如图 2-9 所示，图中是正常的数据写入，使用 *StreamContext* 将倒数第二次写操作最后一个扇区大小的数据保存，并记录写偏移和写操作的数据大小。

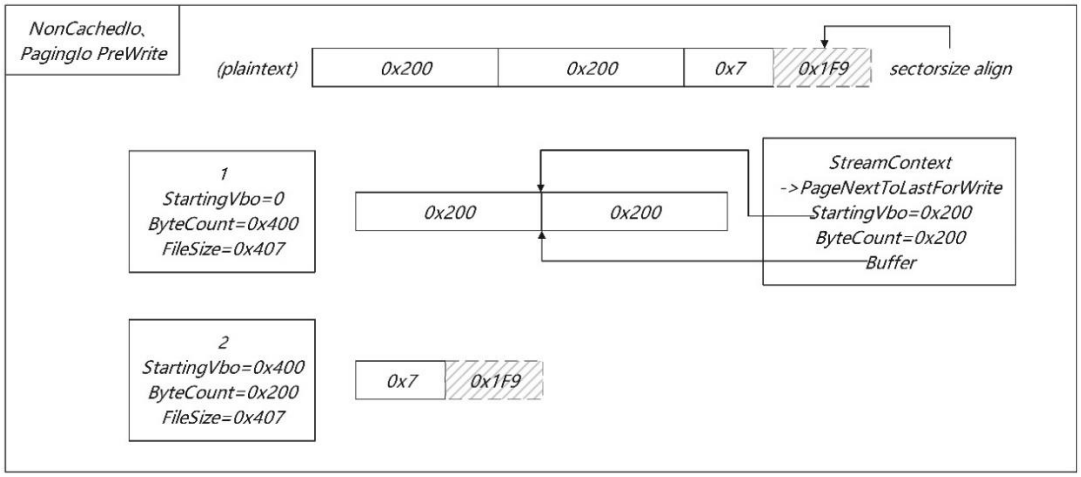

图 2-9 倒数第二次写操作数据大小大于一个扇区大小的未修改数据图

如图 2-10 所示，图中是本文处理过的数据，将倒数第二次的写操作的数据大小相应减掉一个扇区大小，然后将这块剩下的 0x200 的数据直接使用块密码算法加密后下发到文件系统驱动，由文件系统驱动写入磁盘。

倒数第一次写操作的数据之前加上 *StreamContext* 保存的一个扇区的数据，使用密文挪用的方式对拼接后的数据进行加密，然后修改写操作的数据大小和写偏移，将密文整体下发到文件系统驱动，由文件系统驱动写入磁盘。

最后在 PostWrite 时，分别将两次写操作的返回值 IoStatus.Information 恢复为未修改之前的 I/O 数据请求包中的写操作数据大小，避免写操作的请求者接收到错误的返回值，引发不必要的冲突。

通过对写操作的数据进行处理，使文件最后的两个明文块相邻，从而可以使用密文挪用的方式对文件进行加密。

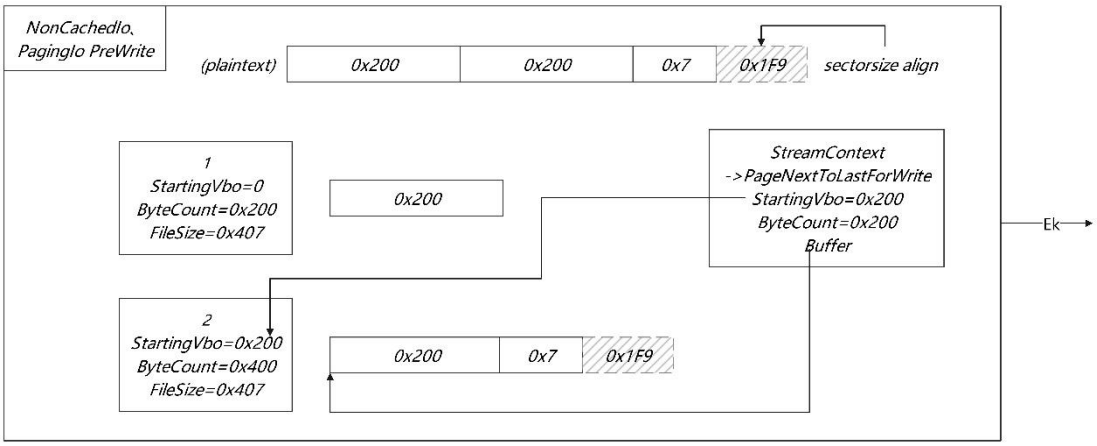

图 2-10 倒数第二次写操作数据大小大于一个扇区大小的修改后数据图

(6) 使用密文挪用时，如果一个文件的非缓冲读操作的倒数两次操作，正好从文件最后两个密文块之间分隔开时。

Windows 文件系统要求非缓冲读操作的数据大小和读偏移必须与扇区大小对齐（这个值通常为 0x200），所以，如图 2-11 所示，读操作的数据占磁盘大小是 0x400，超过文件大小（0x207）的数据将被截断。同样，如果两次非缓冲读操作，如图 2-11、2-12 所示，正好从文件最后两个密文块之间分隔开，此时最后一次的读操作的真正数据大小小于块大小，无法使用密文挪用解密。处理的方式是：使用非重入非缓冲的读函数将待拼接的密文读出，再使用密文挪用的方式对拼接后的数据进行解密，然后将解密后与读偏移和读操作数据大小均相同的明文拷贝回读操作的请求者分配的内存。

倒数第二次读操作的处理：如图 2-11 所示，使用 FltReadFileEx 函数，设置非缓冲标识，且 FileObject 是非重入的，将最后一个不对齐的块读出，与之前的密文拼接后整体使用密文挪用解密，然后把 0x200 的明文拷贝回读操作的请求者分配的内存。

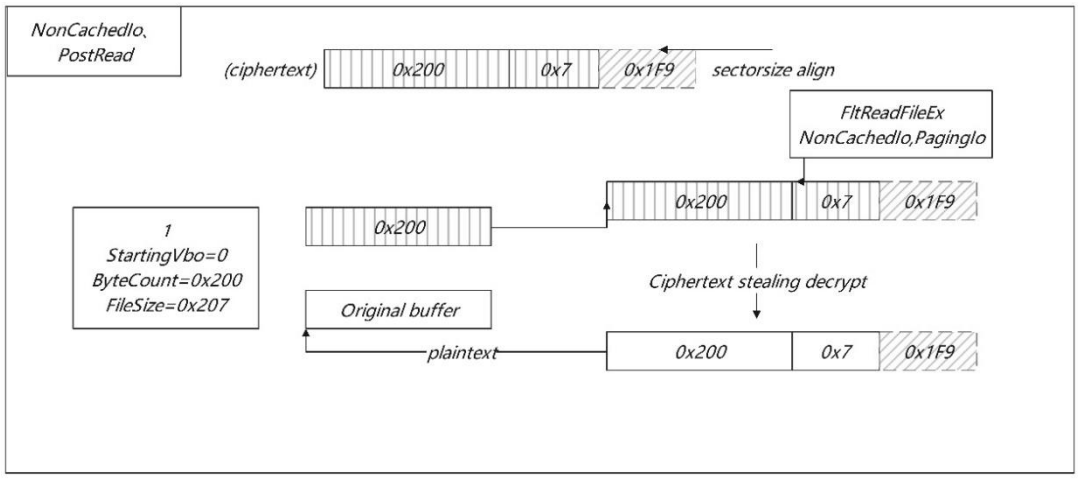

图 2-11 倒数第二次读操作数据处理流程图

倒数第一次读操作的处理：如图 2-12 所示，使用 FltReadFileEx 函数，设置非缓冲标识，且 FileObject 是非重入的，将倒数第二个扇区 0x200 的密文读出，与 0x7 的密文拼接后整体使用密文挪用解密，然后把 0x7 的明文拷贝回读操作的

请求者分配的内存。

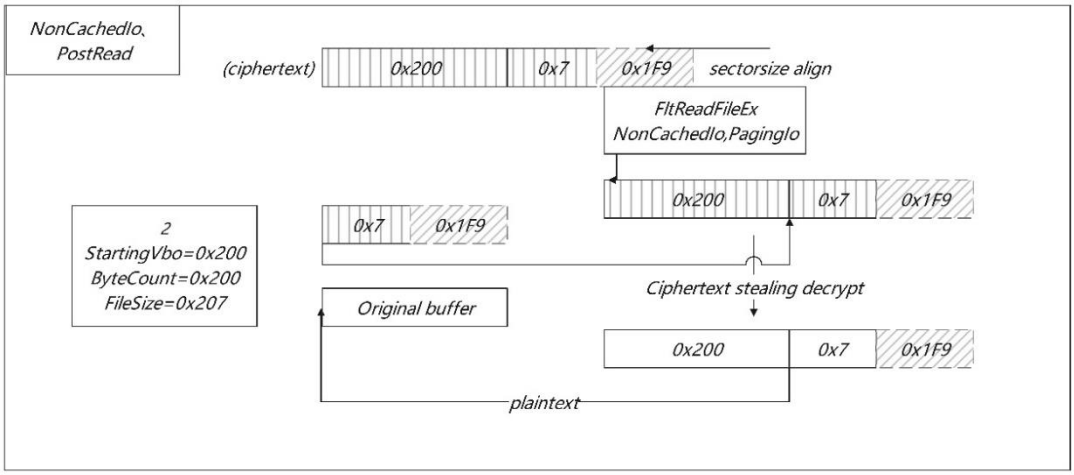

图 2-12 倒数第一次读操作数据处理流程图

4. 读和写的 SwapBuffers

通过 SwapBuffers（交换缓冲区）的方法，minifilter 可以实现透明加密解密的功能。应用程序无法察觉自身是授权进程还是非授权进程，也无法得知自己得到的是明文还是密文。

(1) Write：

如图 2-13 所示，该图是应用程序写操作后操作系统各部分的处理流程，当应用程序指定的是缓冲写操作时，I/O 管理器发送 IRP 到文件系统驱动，文件系统驱动先向缓冲管理器中写入数据（如图 2-13 箭头 4），由缓冲管理器和内存管理器协同再次发送 NonCachedIo 的写请求到文件系统驱动（如图 2-13 箭头 5、6、7、8），最后由文件系统驱动将数据写入磁盘。

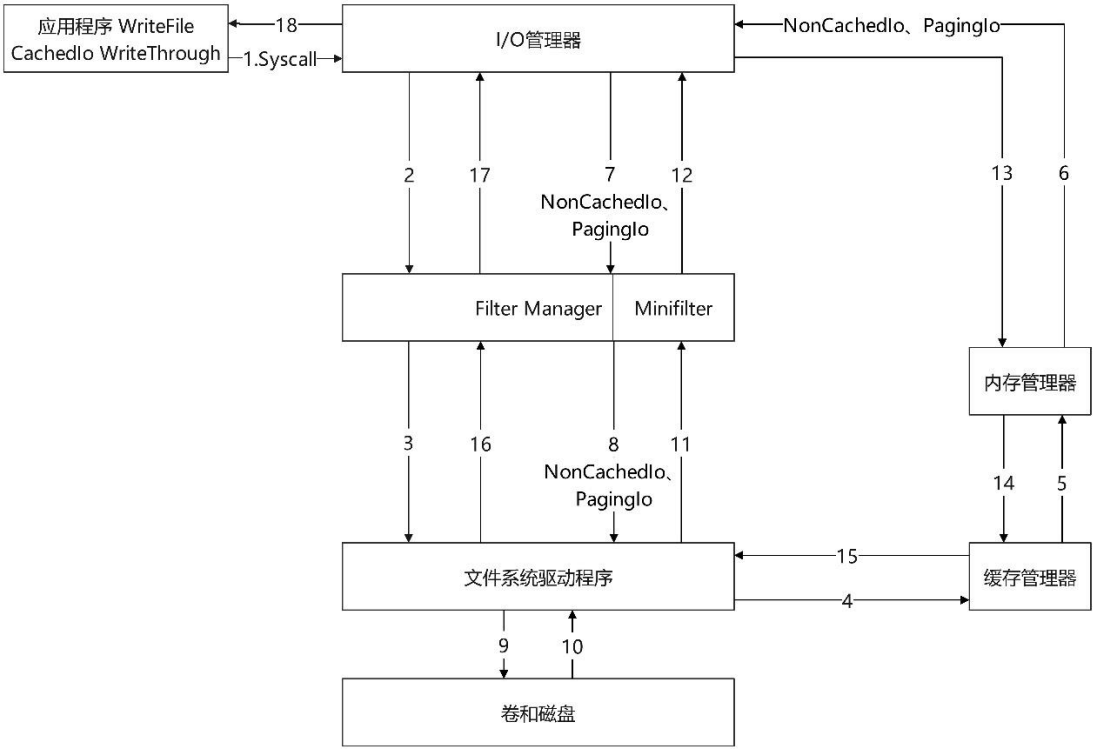

图 2-13 写操作的流程图

我们的 minifilter 拦截的是 NonCachedIo 的 IRP，这里主要是因为 NonCachedIo 的 IRP 是所有写操作的统一的点，只要处理了文件系统驱动的磁盘写操作，那么就可以处理所有的写操作。

如图 2-14 所示，该图是 minifilter 对于 NonCachedIo 的 SwapBuffers（交换缓冲区）流程[13]，在写操作从 IO 管理器把 IRP 发到本文的 minifilter 时，即 PreWrite，本文重新分配一块内存 NewBuffer，把原始内存 OrigBuffer 的数据按照特定的处理方式加密后写入 NewBuffer，把 IRP 的 WriteBuffer 等指针指向本文新分配的 NewBuffer，这样发下去的就是密文。然后在 PostWrite 时，把新分配的内存 NewBuffer 释放掉，并且更新 StreamContext->FileSize。

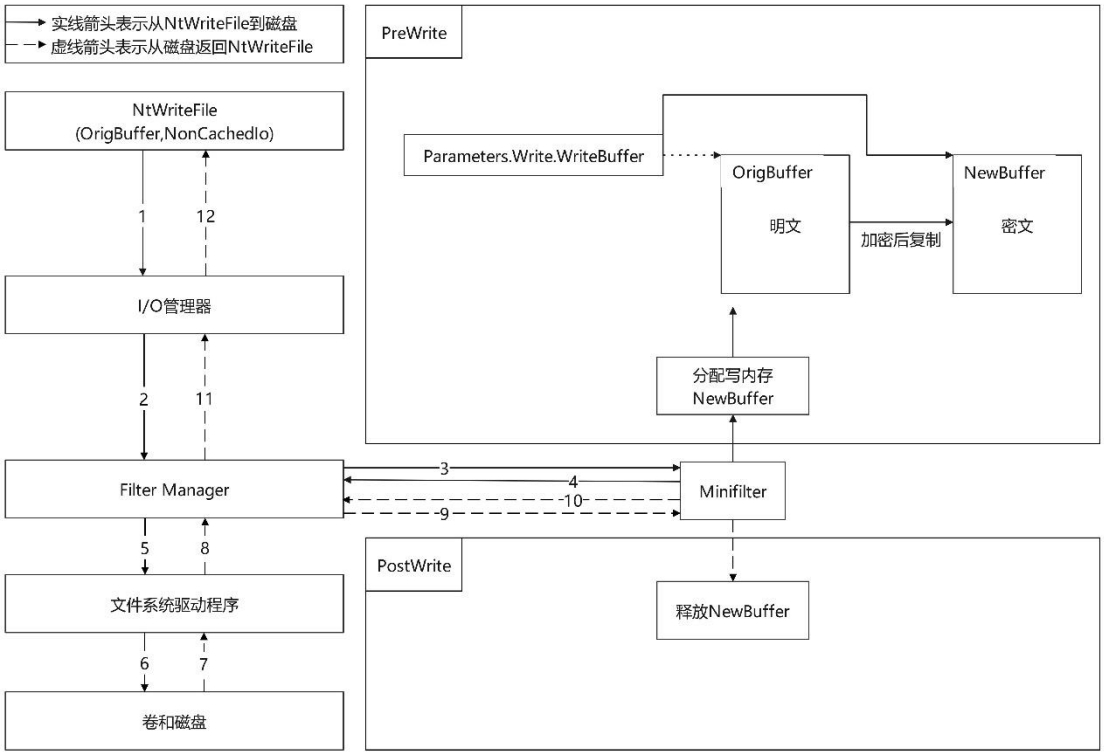

图 2-14 写操作 SwapBuffers 流程

这里的 SwapBuffers 是指写操作不使用原来的内存 OrigBuffer，单独分配一块新的内存 NewBuffer 用于写操作。

按照之前的设计，密文缓冲是不允许下发的，即非授权进程无法写入已加密文件，如下代码，本文在 PreWrite 直接结束 IRP，返回成功。

```
if (FltObjects->FileObject->SectionObjectPointer ==
   StreamContext->ShadowSectionObjectPointers
      && NonCachedIo)
{
   PT_DBG_PRINT(PTDBG_TRACE_ROUTINES,
    ("PocPreWriteOperation->Block
      StartingVbo = %d
      ProcessName = %ws
      File = %ws.\n",
```

```
        Data->Iopb->Parameters.Write.ByteOffset.LowPart,
        ProcessName,
        StreamContext->FileName
        ));

    Data->IoStatus.Status = STATUS_SUCCESS;
    Data->IoStatus.Information = Data->Iopb->Parameters.Write.Length;

    Status = FLT_PREOP_COMPLETE;
    goto ERROR;
}
```

(2) Read：

如图 2-15 所示，该图是应用程序缓冲读操作的流程，与 Write 一样，都是先由文件系统驱动去缓冲管理器中读数据，如果数据已经存在，那么读取后直接返回，如果不存在，那么由缓冲管理器和内存管理器协同请求文件系统驱动从磁盘中读数据。我们的 minifilter 也是只处理 NonCachedIo 读操作。

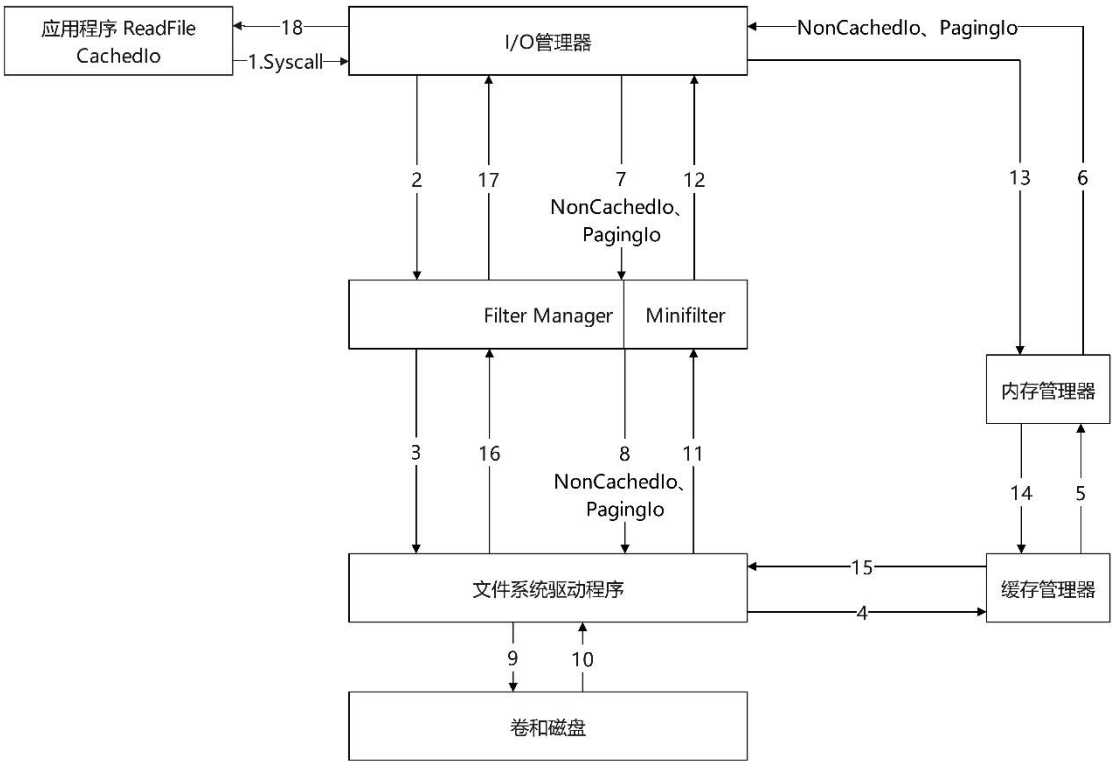

图 2-15 读操作的流程图

如图 2-16 所示，与 Write 稍有不同的是，Read 是在 PostRead 做的 SwapBuffers，因为 PostRead 时，数据才从文件系统驱动中读出。

本文也是在 PreRead 分配内存 NewBuffer，替换 IRP 中的 ReadBuffer 指针为本文的 NewBuffer，之后文件系统驱动读出的密文会存在 NewBuffer 中。

在 PostRead 中对 NewBuffer 中的密文按照对应方式解密，最后把明文拷贝回原始的内存 OrigBuffer 中。

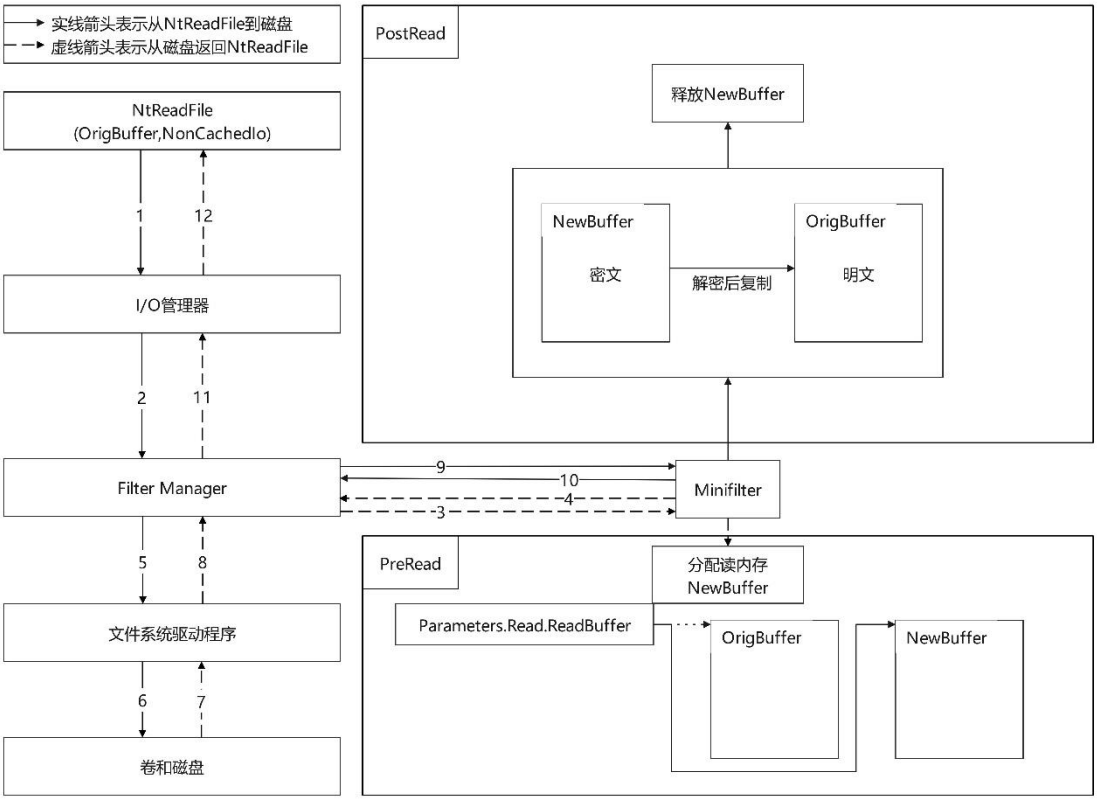

图 2-16 读操作 SwapBuffers 流程

非授权进程的密文缓冲在 PostRead 中不解密，直接返回成功。代码如下：

```
if (FltObjects->FileObject->SectionObjectPointer
   == StreamContext->ShadowSectionObjectPointers)
{
   PT_DBG_PRINT(PTDBG_TRACE_ROUTINES, ("PocPostReadOperation->Don't
 decrypt ciphertext cache map.\n"));
   Status = FLT_POSTOP_FINISHED_PROCESSING;
   goto EXIT;
}
```

5. 重入和特权加密、特权解密

正常的透明加密解密驱动需要实现单独加密或解密文件的功能，这里因为 minifilter 本身需要写入文件明文或密文，所以涉及到是否重入到 minifilter 的问题。

(1)重入与非重入[14]

关于 minifilter 函数 Create，Read，Write 的重入与非重入，这里的重入指的是函数由 minifilter 调用，然后函数执行过程中再次被 minifilter 拦截，从而实现一些操作的过程。

FltCreateFile 函数创建的 FileObject（文件对象），之后无论是 FltReadFile/FltWriteFile 还是 ZwReadFile/ZwWriteFile 操作，都会是非重入的方式。这是因为 IO 管理器内会调用 IoGetRelatedDeviceObject，解析出 FileObject

相关的设备对象，然后把 IRP 发给 Filter Manager，由 Filter Manager 决定下发的卷实例。

使用 ZwCreateFile 函数创建的 FileObject，必须使 FltReadFile/FltWriteFile 才能非重入。当然也可以使用 FltPerformSynchronousIo 直接下发 IRP，不经过 IO 管理器来非重入。

除了上述情况以外，如果在写之前已经有别的进程创建了缓冲，那么该缓冲下发时使用的 FileObject 是该进程创建的 FileObject，而不是我们写操作的非重入 FileObject，这样的写操作是重入的。也就是说，即便用了 FltWriteFile，如果是缓冲写操作，还是有可能是重入的，或者是其他最终调用 MmFlushCache 的函数，比如 FltSetInformationFile 设置文件的 EOF 时，也会导致重入的写操作。

特权加密和特权解密由桌面的应用程序发送命令[15]到 minifilter，因为此时 minifilter 的 Create，Read，Write 相关操作都已建立，所以特权加密，特权解密直接使读写操作重入，minifilter 调用自己的 Write 或 Read 部分代码来实现加密或解密，而不需要再写一份代码用来实现单独的加密和解密处理。

(2)特权加密

要实现特权加密，首先判断文件是否已经被加密过，通过 ZwCreateFile 函数重入（如图 2-17 的箭头 2 和 3）到 minifilter 的 PostCreate 中，然后 minifilter 的 PostCreate 会判断是否有标识尾等，如果未加密，取出明文数据，最后将明文数据以重入且缓冲写操作的方式写入到 minifilter 中，minifilter 的 PreWrite 会加密数据并由文件系统驱动写入磁盘。

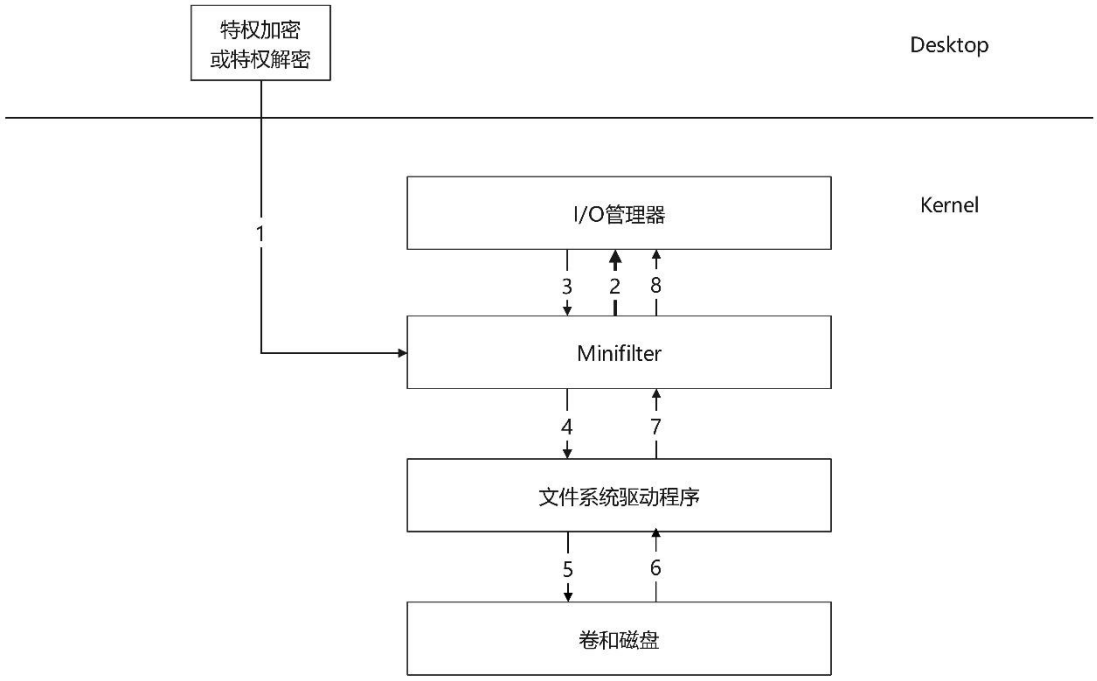

图 2-17 特权加密与特权解密的重入

(3)特权解密

直接的解密与加密一样，首先判断文件是否已加密，如果已加密，判断请求的进程是否为授权进程，非授权进程的话不允许解密，然后以重入（如图 2-17 的箭头 2 和 3）且缓冲读操作的方式读取文件，minifilter 的 PostRead 会解密

数据，然后修改文件大小（这里的 FltSetInformation 会以重入的方式修改文件大小），最后以非重入且非缓冲的写操作写入明文。

优势在于功能不依赖于单独的代码，由已经实现的 minifilter 完成包括判断标识尾，加密解密等操作，便于代码后期的统一维护。

6. 重命名方式读写文件[16]

有时用户会有重命名文件的操作，有的应用程序还会修改扩展名；像 WPS 这种软件，处理 docx，doc 等 office 文件时[17]，是通过两次重命名实现的：将原来的 docx 文件，重命名为 tmp 扩展名文件，将另一个真正写入新的数据的 tmp 文件最后重命名为 docx 文件，这样就实现了替换。

而且还会有已加密文件重命名后，加密标识尾中的文件名需要更新的问题。所以，我为重命名操作制定了几条规则，如图 2-18 所示。

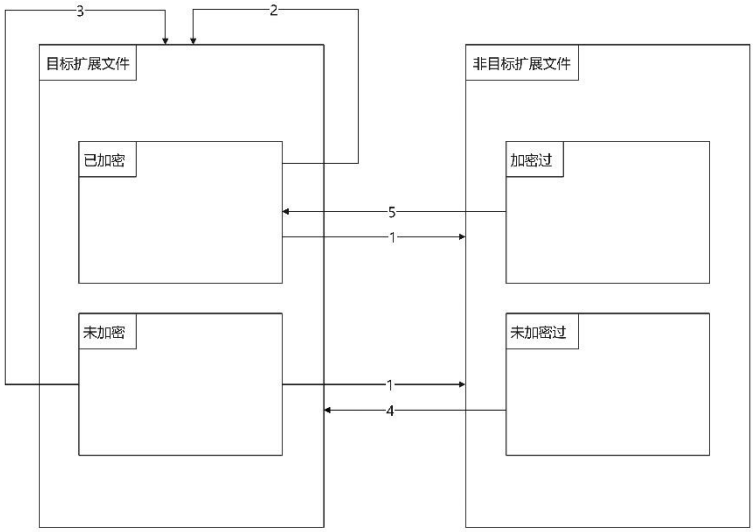

图 2-18 文件重命名的 5 条规则

(1) 已加密、未加密目标扩展名文件重命名为非目标扩展名文件：

删除 StreamContext 中的数据。

(2) 已加密目标扩展名文件重命名为目标扩展名文件：

删除 StreamContext 中的数据，之后下一次的 Create 时，会发现实际文件名与标识尾的文件名不同，设置 StreamContext->Flag，然后 PostClose 会重新写入标识尾。

(3) 未加密目标扩展名文件重命名为目标扩展名文件：

删除 StreamContext 中的数据。

(4) 非目标扩展名文件(且不是由加密文件重命名的)重命名为目标扩展名文件：

在 PostSetInformation 中重入 Create 中判断是否有标识尾，没有则设置 StreamContext->Flag，在 PostClose 中用重入的方式加密文件。

(5) 非目标扩展名文件且本身是密文重命名为目标扩展名文件：

删除 StreamContext 中的数据，之后下一次的 Create 时，会发现实际文件名与标识尾的文件名不同，设置 StreamContext->Flag，然后 PostClose 会重新写入标识尾

五条规则中，删除 StreamContext 中的数据就意味着，除非改成了目标扩展名文件，否则就不会再进入 PostCreate 中了，我们的 minifilter 就不会再处理该文件了。

7. 授权进程的控制与保护

对于进程的鉴别与保护也是本文一个比较重要的问题，如果一个非授权进程可以冒充授权进程而得到明文，那么就会导致数据的泄露。

(1) 进程的控制

使用两张链表 ProcessRules 和 ProcessInfo 分别保存进程规则和新创建的进程信息，在每一个 ProcessRules 中指向对应的 ProcessInfo 链表头 PocCreatedProcessListHead，这里 ProcessInfo 是一条动态的链表，而 ProcessRules 是在驱动开始之前配置的静态链表。两张链表 ProcessRules 和 ProcessInfo 中定义的各成员如下：

```
typedef struct _POC_PROCESS_RULES
{
   LIST_ENTRY ListEntry;
   PWCHAR ProcessName;
   ULONG Access;

   LIST_ENTRY PocCreatedProcessListHead;
   KSPIN_LOCK PocCreatedProcessListSpinLock;

}POC_PROCESS_RULES, * PPOC_PROCESS_RULES;

typedef struct _POC_CREATED_PROCESS_INFO
{
   LIST_ENTRY ListEntry;
   HANDLE ProcessId;
   BOOLEAN ThreadStartUp;

   PPOC_PROCESS_RULES OwnedProcessRule;

}POC_CREATED_PROCESS_INFO, * PPOC_CREATED_PROCESS_INFO;
```

使用函数 PsSetCreateProcessNotifyRoutineEx 注册进程创建回调，参数 CreateInfo 不为空表示新进程的创建，如果该进程在 ProcessRules 链表中，则将其加入到 ProcessInfo 中，而当 CreateInfo 为空表示该进程结束，将其从 ProcessInfo 链表摘除。并且调用函数 NtQuerySystemInformation 遍历系统内的 EPROCESS，将在回调注册之前就已经创建的相关进程加入到链表中。这样一来就可以按照链表中进程的权限（Access）项，判断当前进程的权限[18]，从而决定对于文件是解密还是不解密。

(2) 进程的保护

Minifilter 中创建线程，间隔一段时间对授权进程的.text（代码段）进行完整性校验[19]。首先读取进程的文本文件，将进程的各节映射到内存中，对代码按照重定位表进行重定位，然后计算进程文本文件.text（代码段）的 SHA-256 哈希值。

然后使用函数 KeStackAttachProcess 附加到目标进程的内存空间中，修改进程内存的保护属性为可读可执行，读取.text 代码段计算哈希。将两个哈希值进行比较即可判断进程.text 的完整性。通过这种方式可以一定程度上保护授权进程本身不被修改。

（三）实验与分析

本章是对项目的测试部分，主要测试以下四点：

首先测试对于.txt 文件写入时的加密功能，授权进程记事本读取文件时的解密功能；

其次测试授权进程和非授权进程的双缓冲功能以及进程的权限配置；

再测试对 docx 文件的加密与解密；

最后测试对 txt 文件的特权加密与特权解密。

1. 测试环境

开发环境是 Visual Studio 2019，Windows 10 SDK(10.0.19041.0)；

测试环境选择 Windows 10 x64 LTSC 1809；

安装在 VmWare workstation Pro 16 虚拟机中；

磁盘是 NTFS 文件系统[20]。

测试工具如下

软件	初始权限/作用
notepad.exe（记事本）	明文
notepad++.exe	密文
wordpad.exe（写字板）	密文
WPS（wps.exe）	明文
OsrLoader	加载驱动
DebugView	打印驱动输出信息

表 3-1 测试工具的权限或作用

2. 系统部署

使用 OsrLoader 加载驱动，使用时要选择 minifilter 类型。然后先注册驱动，再加载驱动，如图 3-1 所示。

图 3-1 使用 OsrLoader 加载 minifilter 驱动

驱动加载以后，打开界面，可以配置授权进程和非授权进程权限。如图 3-2 所示。

图 3-2 配置进程权限的界面

3. 功能测试

(1) 测试 txt 文件写入时的加密功能，授权进程记事本读取文件时的解密功能

目的：验证加密策略：对一个目标扩展名文件有修改操作时自动加密；以及测试能否成功使用密文挪用加密和解密文件。

① 授权进程记事本写入 11014 个字节的数据，该数据与 AES-128 的块大小（16 个字节）不对齐。DebugView 的日志显示文件加密成功。如图 3-3 所示，该图上方显示 PocPreWriteOperation->Encrypt success。

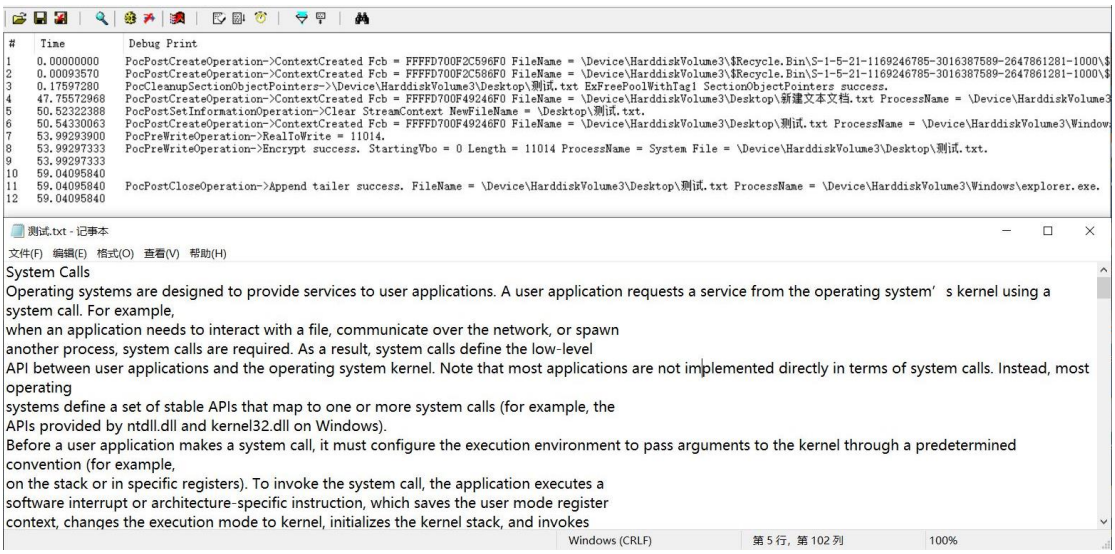

图 3-3 记事本写入数据后加密

② 测试.txt 文件加密以后，可通过非授权进程 notepad++.exe 打开文件是否显示密文判断加密是否成功，如图 3-4 所示，该图下方为密文（乱码），说明文件加密成功。

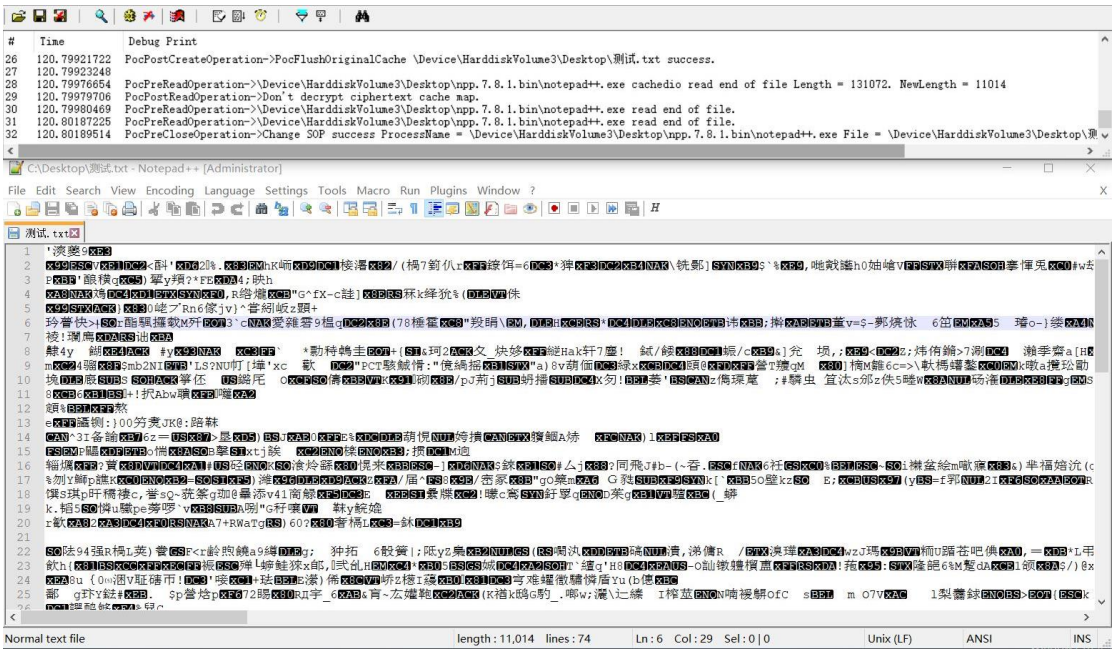

图 3-4 非授权进程 notepad++.exe 显示密文

③ 授权进程记事本读取数据显示明文，DebugView 显示解密成功。如图 3-5 所示，该图上方显示 PocPostReadOperation->Decrypt success。

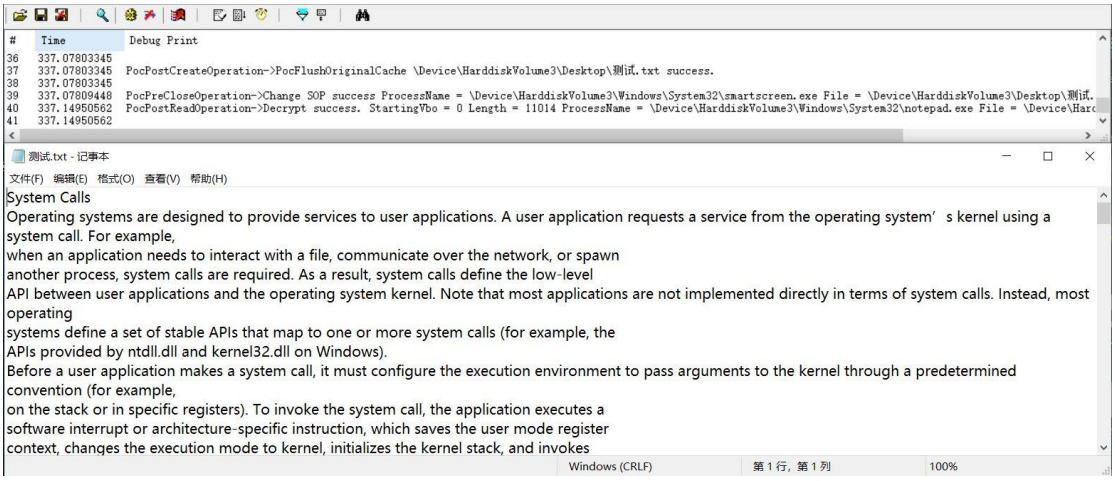

图 3-5 记事本读取数据时解密

通过以上的测试，可以说明 minifilter 的加密策略正常工作，加密和解密流程正常。

(2) 测试授权进程和非授权进程的双缓冲功能以及进程的权限配置

目的：测试明文缓冲与密文缓冲的功能

① 非授权进程访问的是密文缓冲，对应的读操作不解密，所以显示密文（乱码）。如图 3-6 所示，该图下方 notepad++.exe 的内容为乱码。

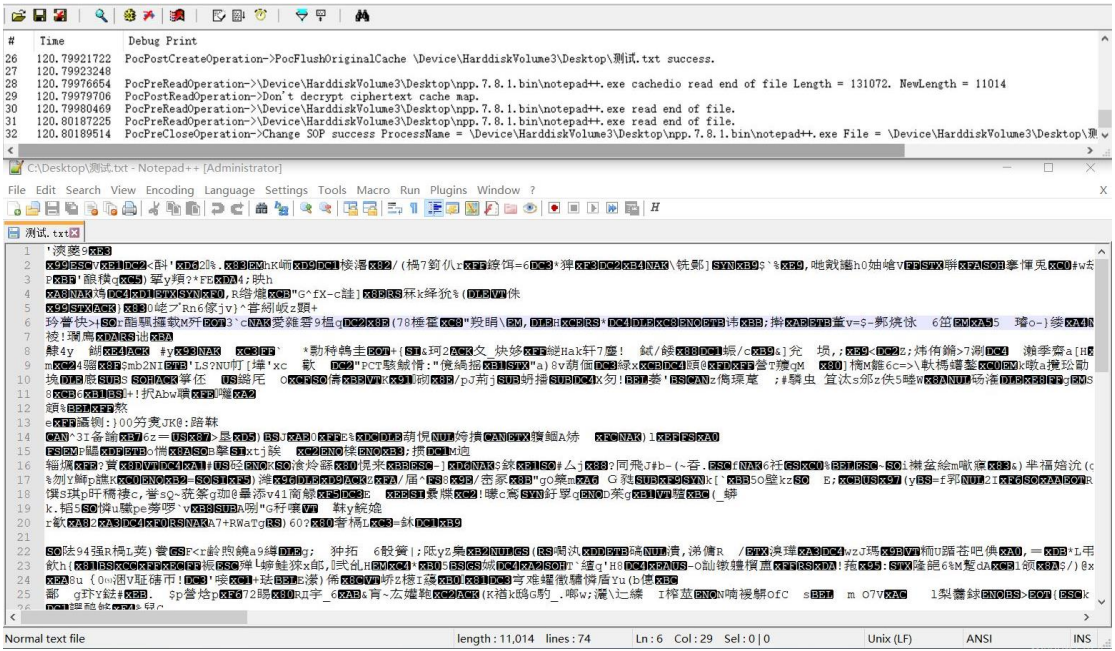

图 3-6 非授权进程 notapad++.exe 的密文缓冲

② 通过授予 notepad++.exe 明文权限，使它变成授权进程，访问的是明文缓冲，显示明文。如图 3-7 所示，该图下方 notepad++.exe 的内容为正常数据。

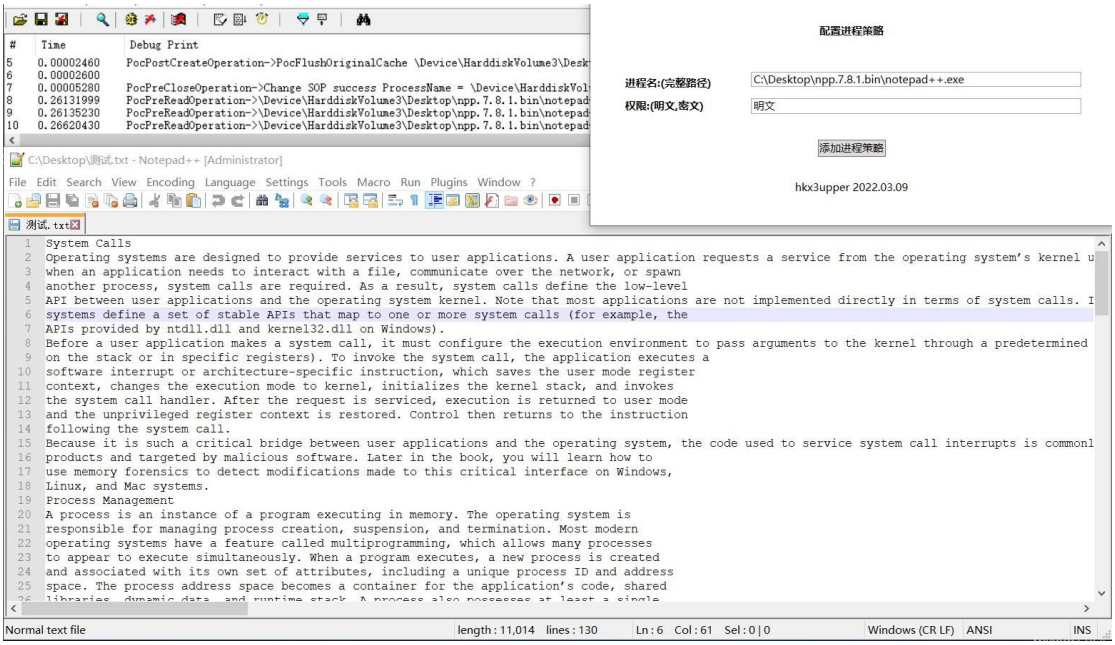

图 3-7 授予 notepad++.exe 明文权限

通过以上测试，可以说明本文为授权进程和非授权进程设置的双缓冲正常工作，通过双缓冲将两者隔离，而不用频繁清除缓冲，增强了系统的稳定性。

(3) 测试对 docx 文件的加密与解密

目的：测试使用 tmp 文件重命名方式读写的文件是否可以加密解密。

① 通过 DebugView 中打印的日志显示文件（docx 测试.docx）加密成功。如图 3-8 所示，该图上方输出 PocReentryToEncrypt->success。

图 3-8 授权进程 WPS 写入数据后加密

② 使用非授权进程写字板打开该文件，没有 docx 固定的文件格式，说明该文件是密文，文件已成功加密。如图 3-9 所示，写字板中是乱码。

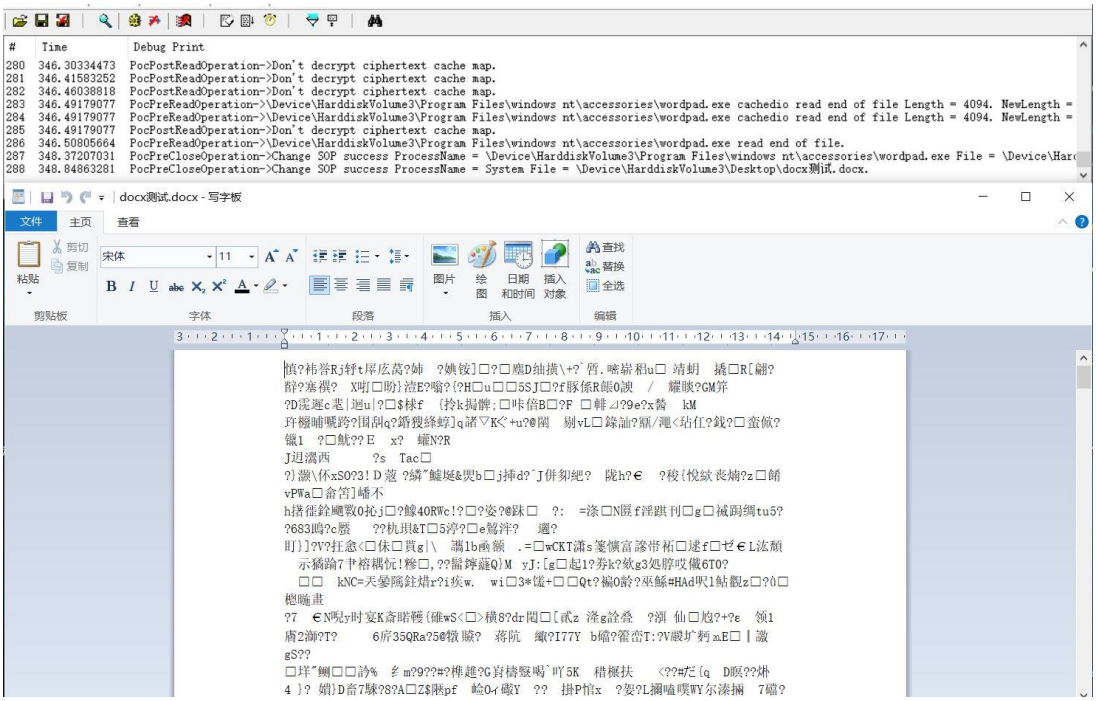

图 3-9 非授权进程写字板显示密文

③ 使用授权进程 WPS 读取数据，DebugView 中打印的日志显示文件解密成功，而且显示明文。如图 3-10 所示，该图上方显示 PocPostReadOperation ->Decrypt success，下方 WPS 中是正常数据。

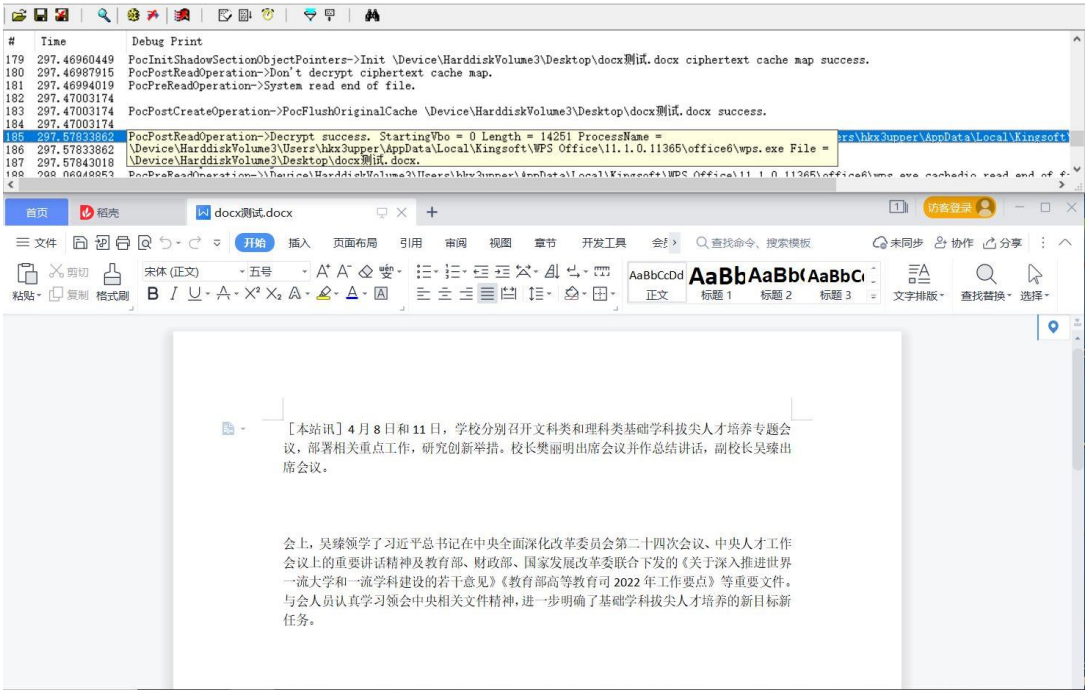

图 3-10 使用授权进程 WPS 读取数据时解密

通过以上测试，可以说明，使用 tmp 重命名方式读写的文件可以正常加密与解密。

(4) 测试对 txt 文件的特权加密与特权解密

目的：测试使用重入方式的特权加密与特权解密是否成功。

① 对已加密的文件（测试.txt）进行特权解密后，使用非授权进程写字板打开也会显示明文，说明文件解密成功。如图 3-11 所示，该图下方的写字板程序中是正常数据。

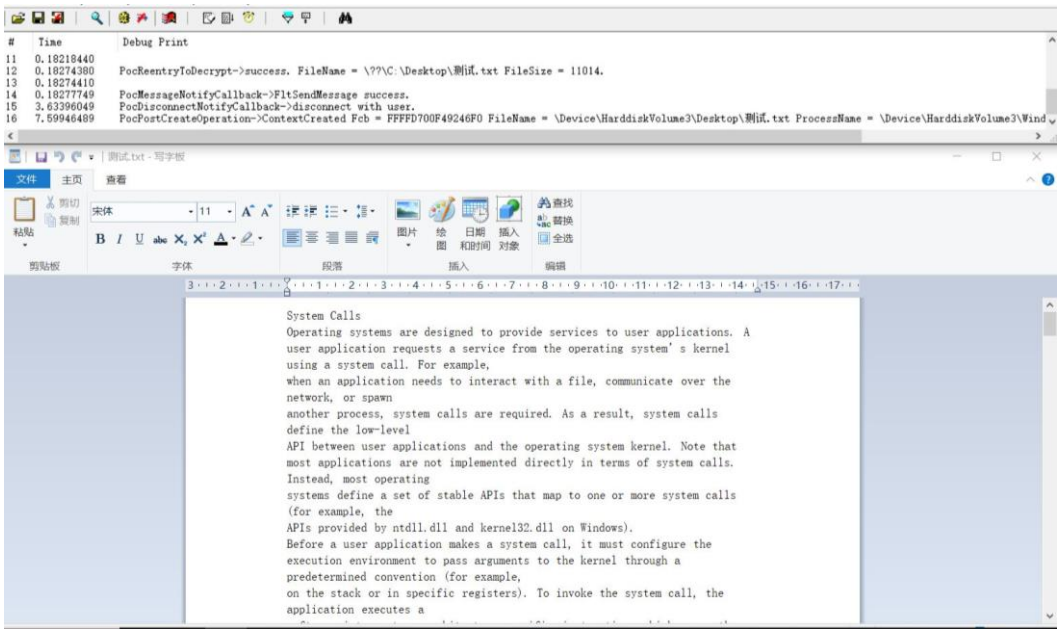

图 3-11 特权解密后非授权进程写字板显示明文

② 对文件进行特权加密后，使用非授权进程写字板打开显示密文，说明文件加密成功。如图 3-12 所示，该图下方的写字板程序中是乱码。

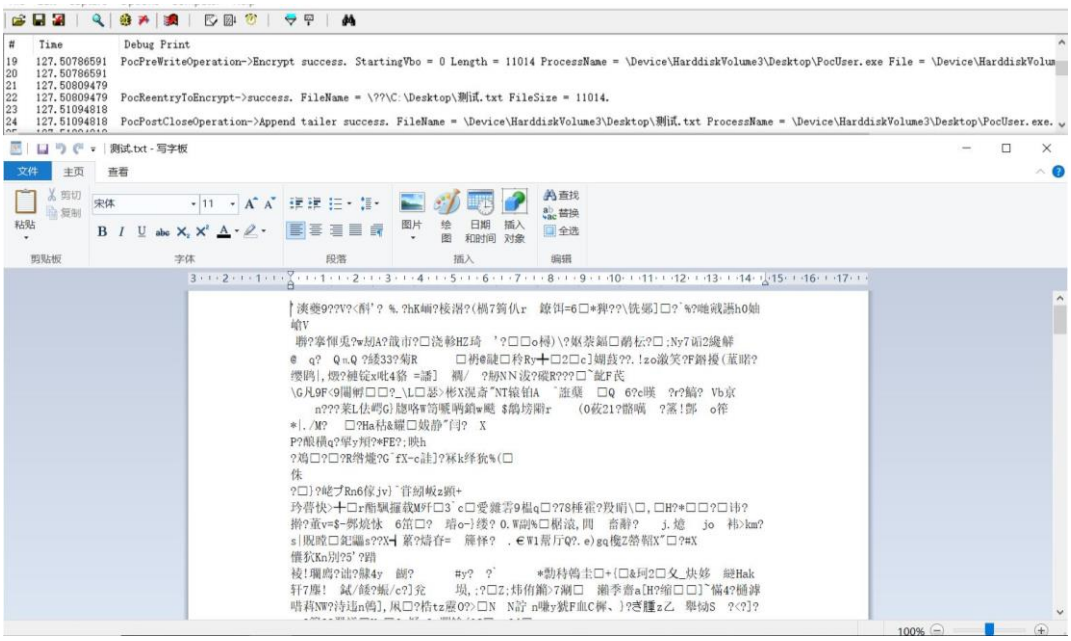

图 3-12 特权加密后非授权进程写字板显示密文

4. 实验分析

从测试结果来看，minifilter 驱动的各种功能达到预期结果，通过双缓冲将授权进程与非授权进程隔离，保护了数据的安全，同时兼顾对于 office 文件的加密和解密，以及对文件单独的加密和解密，另外还可以配置进程的权限。

目前项目应用在 office、wps 的某些版本上会有死锁的发生，主要发生在添加文件标识尾和对 tmp 文件的加密处，本文对这些情况进行了尽可能的修复，对于这些问题还需要在实际的生产环境下一步一步的完善。

三、 结论

目前国内透明加解密驱动发展很快，也有许多公司实现了成熟的项目，但相关的开源项目很少，大部分是基于传统过滤驱动，而且很多的重难点并没有实现，比如双缓冲和使用块加密算法，前者大多是在进程切换之间采用清除缓冲的方法来实现的，后者大多只是用简单的异或加密来模拟实际的文件加密与解密。大多数并没有实现对于 office 文件的兼容。

本课题实现了一个双缓冲透明加解密驱动，主要工作如下：

双缓冲的建立是调用上方接口 FltReadFileEx 实现的，没有直接使用更底层的 Cc 函数，比如 CcInitializeCacheMap，有更好的兼容性；

使用了密文挪用的方式解密解密数据，避免了块密码算法需要扩展文件大小的问题；

使用重入的方式单独加密解密文件，精简了代码了逻辑，也提升了整体的效率；

解决了重命名的问题，可以实现对于 docx 等文件的加解密。

对进程进行鉴别和控制，通过对授权进程的代码段进行校验，实现了对其的保护。

关于密文挪用以及重入进行加密解密是本项目比较创新的点，而关于双缓冲部分的建立则是本项目的重难点，从当时对于 Windows 的文件系统一无所知到项目的实现，我查阅了大量的书籍和资料。

参考文献

[1] 邵昱,萧蕴诗.基于文件系统过滤驱动器的加密软件设计[J], 计算机应用, 2005(05):1151-1152.

[2] 谭文,陈铭霖.Windows 内核安全与驱动开发[M], 北京:电子工业出版社, 2015. 297-316.

[3] 刘晗.基于双缓冲过滤驱动的透明加密系统研究与实现[D],重庆:重庆大学, 2010. 9-14.

[4] Matthew V. Ball,Cyril Guyot,James P. Hughes,Luther Martin,Landon Curt Noll.The XTS-AES Disk Encryption Algorithm and the Security of Ciphertext Stealing[J].Cryptologia,2012,Volume 36, Issue1:70-79.

[5] 毛德操.Windwos 内核情景分析：采用开源代码 ReactOS[M], 北京:电子工业出版社, 2009.1099-1276.

[6] 何明.基于 Minifilter 微框架的文件加解密系统的设计与实现[D], 北京:北京工业大学, 2014.43-45.

[7] Microsoft Corporation. About minifilter contexts[EB/OL], https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/managing-contexts-in-a-minifilter-driver, 2021.

[8] Microsoft Corporation. Swapbuffer File System Minifilter Driver[EB/OL], https://docs.microsoft.com/en-us/samples/microsoft/windows-driver-samples/swapbuffer-file-system-minifilter-driver, 2021.

[9] Rajeev Nagar.Windows NT File System Internals:A Developer's Guide, O'REILLY, 2007.243-352.

[10] 徐翔.文件保护系统中透明加解密技术的设计与实现[D], 北京:北京化工大学, 2009.10-15.

[11] 安亮.基于文件微过滤驱动的文件监控系统的设计与实现[D], 上海:上海交通大学, 2020.47-50.

[12] 潘爱民.Windows 内核原理与实现[M], 北京:电子工业出版社, 2010.493-496.

[13] 李凡,刘学照,卢安,谢四江. WindowsNT 内核下文件系统过滤驱动程序开发[J].华中科技大学学报(自然科学版), 2003(01):19-21.

[14] Issuing IO in minifilters: Part 2 – Flt vs. Zw[EB/OL]. https://fsfilters.blogspot.com/2010/02/issuing-io-in-minifilters-part-2-flt-vs.html?m=1, 2010.

[15] 谢佳岩.基于 Minifilter 框架的文件保护方法及关键技术研究[D], 秦皇岛:燕山大学, 2016.25-27.

[16] 俞晓光.基于文件系统过滤驱动的文件保护系统设计与实现[D], 西安:西安电子科技大, 2012.32-36.

[17] 孟钰鑫.基于透明加密的动漫素材保护软件的设计和开发[D], 青岛:中国海洋大学, 2013.19-33.

[18] 孙莹莹,郑扣根.基于微过滤驱动的文件监控系统[J].计算机应用,2010(11): 3115-3117.

[19] 薛胜军,曹凤艳.基于微过滤驱动的安全软件文件保护[J].武汉理工大学学报, 2011,33(4):130-133.

[20] 孙思良.基于 NTFS 的文件加密系统[J].网络安全技术与应用,2005(12):52-54.

谢　辞

我向你们，和我曾在你们中度过的青春告别了。我们是那坚牢植物的种子，在我们的心成熟丰满的时候，就交给大风纷纷吹散。