Design and Implementation of a Windows Kernel Driver for LUKS2-encrypted Volumes

Enabling Read-Only Access to LUKS2 Volumes on Windows and Comparing the Read Rate to Other Disk Encryption Technologies

MAX IHLENFELDT

Universität Augsburg
Lehrstuhl für Organic Computing
Bachelorarbeit im Studiengang Informatik

This thesis presents an implementation of read-only support for LUKS2-encrypted volumes on the Windows operating system. To our knowledge, this is the first solution that enables access to LUKS2 volumes using Windows. We implemented a WDM filter driver that intercepts all read requests from an encrypted volume, transparently decrypting them such that the contained file system can be recognized. The configuration of the driver is done using our user mode command line tool. To test and classify our driver's read rate, we compared it to BitLocker's and VeraCrypt's. Using the `fio` tool, we measured both sequential and random-access read performance. We found that for large block sizes our driver reaches up to 91% of BitLocker's and 101% of VeraCrypt's sequential read rate; and up to 92% of BitLocker's and 134% of VeraCrypt's random-access read rate.

Contents

1 Introduction

Data privacy is an important topic in today's increasingly digital world. Deciding and controlling who has access to what information is a huge part of one's digital autonomy. This is not only relevant for data made available via the Internet, but also for offline storage.

If third parties' access to permanently stored data cannot be completely ruled out, preventing them from reading it may be desirable. Depending on the type of data, it may even be absolutely essential, for example if trade or state secrets are concerned. An everyday example for a scenario like this is a laptop: it may get lost or stolen, giving other people physical access to the contents of its data storage. Accidentally giving others access to data can also happen when disposing of old hard disks or SSDs: overwriting data in such a way that it cannot be recovered is a non-trivial process.[1]

One way of preventing information leaks even in the case of physical access is to encrypt the partition containing the data, also known as full disk encryption. This method of data protection has become more and more popular as hardware and thus encryption have become faster. There are different standards and technologies for full disk encryption. One of them is Microsoft's BitLocker technology, which is a built-in feature of the Windows operating system. The fact that Microsoft created their own disk encryption solution shows the importance and popularity of this technology. Another possibility is the Linux Unified Key Setup (LUKS), which currently exists in its second version called LUKS2. As the name suggests, it was designed for usage with operating systems from the Linux family, and is quite popular in that domain: the repositories of all major Linux distributions contain the LUKS2 packages, and most describe how to use it in their wikis. As the usage of Linux continues to grow, LUKS2 probably accounts for a considerable portion of fully encrypted disks.

However, as far as we know, it is currently not possible to use LUKS2-encrypted volumes on Windows.[2] This is a significant drawback compared to other full disc encryption solutions that are cross-platform, such as the VeraCrypt project, which works on both Linux and Windows. Because we think that competition between multiple standards for disk encryption on Windows is desirable, we want to try and eliminate this drawback. The purpose and contribution of this thesis therefore is to see if it is feasible, with reasonable effort, to bring LUKS2 support to Windows, and if yes, check whether the performance can hold up to practical usage standards.

When measuring I/O performance, one needs to consider the two possible access patterns, sequential and random access. According to [1], random-access I/O "is typical of on-line transaction processing (OLTP) systems, database systems, or mail server applications." Examples for sequential access are "large file processing (scientific computing, large-scale financial processing), large database queries (data mining, business intelligence), and on-demand video." For a storage solution to be usable in practice, both its sequential and random-access throughput need to be high enough.

[1] We will come across this fact again in the next chapter and discuss it in more detail.

[2] There is the LibreCrypt project, but it only supports the first version of LUKS. It also seems not to be maintained any more.

In Chapter 2 we describe how the LUKS2 encryption standard works, as understanding this is crucial for implementing it. The on-disk format, how to unlock an encrypted partition, and how to use an unlocked partition are outlined. We chose to write a Windows kernel driver to realize our endeavour, and therefore also detail the knowledge required to comprehend the mechanics and development of such a driver. This includes information on Windows' inner workings, an introduction to its driver models and concepts, and driver debugging techniques.

Full disk encryption, kernel drivers, and benchmarking storage systems are all well-researched topics. In Chapter 3, we lay out important work related to this thesis.

Microsoft's BitLocker technology and the VeraCrypt project both have already been mentioned. To understand the state of full disk encryption and the currently available solutions, Chapter 4 is devoted to these other approaches. Additional to the two already mentioned implementations, LUKS2's reference implementation and the subsystem of the Linux kernel that it uses are described in detail. This includes all relevant cryptographic and theoretical details as well as implementation specifics.

Chapter 5 is all about our implementation of LUKS2 on Windows, which consists of two components, together forming the Oknolynx project. Our choice for a driver framework, the architecture and initialization process, as well as the details of its operation are all described.

As mentioned above, we do not only want to implement LUKS2 support on Windows; we also want to quantify how well it performs. These performance measurements, including comparisons to BitLocker and VeraCrypt, are the contents of Chapter 6. We perform preliminary tests in a virtual machine, and afterwards more extensive measurements on real hardware.

Our conclusions on the success of our feasibility study are laid out it Chapter 7, together with suggestions for possible further work and research.

Finally, one note on terminology: in this thesis, we use the terms *volume* and *partition* equivalently.

2 Background

This chapter introduces information and concepts that are needed to understand both the problem this thesis solves and the details of our solution. It describes the encryption standard implemented by our driver and details the concepts needed for Windows kernel driver development.

2.1 LUKS2 Disk Encryption

Linux Unified Key Setup 2, or short LUKS2, is the second version of a disk encryption standard. It provides a specification [2] for a on-disk format for storing the encryption metadata as well as the encrypted user data. Unlocking an encrypted disk is achieved by providing one of possibly multiple passphrases or keyfiles. The intended usage of LUKS2 is in combination with the Linux device mapper subsystem (described in Section 4.1.1), but that is not mandatory.[1]

The differences between the original LUKS and LUKS2 are minor. According to [2], LUKS2 adds "more flexible ways of storing metadata, redundant information to provide recovery in the case of corruption in a metadata area, and an interface to store externally managed metadata for integration with other tools." Practically, this means that LUKS2 has a different on-disk layout and, among other things, supports more password hashing algorithms (more precisely, password-based key derivation functions).

2.1.1 Our LUKS2 Cryptographic Helper Library

The reference implementation[2] is designed only for usage on Linux, which is why we developed a new library in Rust for interacting with LUKS2 partitions. This is not a full equivalent, but only a cross-platform helper. Its task is to take care of all the cryptographic work needed before actually decrypting and encrypting data (more precisely, the process described in Section 2.1.3). Notably, it lacks the following features of the reference implementation:

- formatting new LUKS2 partitions,
- modifying or repairing existing LUKS2 partitions,
- converting a LUKS partition to a LUKS2 partition,
- actually mounting a LUKS2 partition for read/write usage (this is what our kernel driver and its userspace configuration tool is for).

Our library does provide access to the raw decrypted user data, but the practical use of this is very limited: the decrypted data is in the format of a filesystem, e.g. FAT32, btrfs, or Ext4. Therefore, a filesystem driver is needed to actually access the stored files. One

[1] As we show in this thesis, it is possible to make the combination of LUKS2 and Windows work.
[2] `https://gitlab.com/cryptsetup/cryptsetup`, see Section 4.1.2.

way of exposing the decrypted data to the system's filesystem drivers is by transparently decrypting the data directly in the kernel, which is what our driver does (see Chapter 5).

2.1.2 On-Disk Format

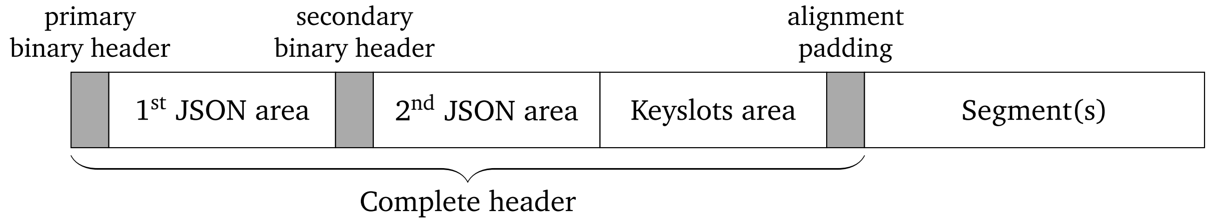

Figure 2.1: LUKS2 on-disk format (modified after [2]). The complete header consists of three areas: a binary header of exactly one 4096-byte sector, JSON metadata, and the binary keyslots data. A *keyslot* is an "encrypted area on disk that contains a key" [2]. For redundancy, the binary header and the JSON metadata are stored twice. After that follow one or more areas containing encrypted user data. The specification calls these areas *segments*.

Figure 2.1 shows the high-level layout of a LUKS2-encrypted disk.

The two binary headers have a size of exactly one sector, so that they are always written atomically. Only the first 512 bytes are actually used. The header marks the disk as following the LUKS2 specification, and contains metadata such as labels, a UUID, and a header checksum. The labels and UUID can be accessed using the `blkid`[3] command-line tool and also be used in the `udev`[4] Linux subsystem. For the detailed contents, see Figure 2.2. Figure 2.3 also contains an example hexdump of a binary header.

The sector containing the binary header is followed by the JSON area. This area contains the metadata that is arguably most relevant for decryption and encryption. Figure 2.4 contains an overview of the objects stored in JSON and their relationships. For this thesis' brevity's sake, please refer to chapter 3.1 in [2] for an example of a LUKS2 JSON area.

After the JSON area, the keyslots area is stored on the disk. This is space reserved for storing encrypted cryptographic keys. The metadata from the JSON keyslot objects describe the position of a key on the disk as well as information on how to decrypt it.

2.1.3 Unlocking a Partition

For simplicity, our LUKS2 Rust library does not support unlocking a keyslot using an external keystore defined by a token, even though Figure 2.4 mentions this possibility. Only unlocking via password is implemented. The library does however include support for different *password-based key derivation functions (PBKDFs)*, namely `pbkdf2` with SHA-256, `argon2i`, and `argon2id`. These are all the PBKDF algorithms that are listed in the LUKS2 specification (see [2], Table 3). The default PBKDF used by LUKS2 is `argon2i` [3]. In the following paragraphs, it will become apparent what PBKDFs are used for in LUKS2.[5]

[3] `https://www.man7.org/linux/man-pages/man8/blkid.8.html`
[4] `https://man7.org/linux/man-pages/man7/udev.7.html`
[5] It will also become apparent that they are quite aptly named.

```c
#define MAGIC_1ST "LUKS\xba\xbe"
#define MAGIC_2ND "SKUL\xba\xbe"
#define MAGIC_L     6
#define UUID_L     40
#define LABEL_L    48
#define SALT_L     64
#define CSUM_ALG_L 32
#define CSUM_L     64

struct luks2_hdr_disk {
    char     magic[MAGIC_L];       // MAGIC_1ST or MAGIC_2ND
    uint16_t version;              // Version 2
    uint64_t hdr_size;             // size including JSON area [bytes]
    uint64_t seqid;                // sequence ID, increased on update
    char     label[LABEL_L];       // ASCII label or empty
    char     csum_alg[CSUM_ALG_L]; // checksum algorithm, "sha256"
    uint8_t  salt[SALT_L];         // salt, unique for every header
    char     uuid[UUID_L];         // UUID of device
    char     subsystem[LABEL_L];   // owner subsystem label or empty
    uint64_t hdr_offset;           // offset from device start [bytes]
    char     _padding[184];        // must be zeroed
    uint8_t  csum[CSUM_L];         // header checksum
    char     _padding4096[7*512];  // Padding, must be zeroed
} __attribute__((packed));
```

Figure 2.2: LUKS2 binary header structure from [2]. Integers are stored in big-endian format, and all strings have to be null-terminated. The `magic`, `version`, and `uuid` fields are also present in the LUKS1 binary header and were placed at the same offsets as there.

The LUKS2 specification allows for multiple segments in one partition. To make things easier, our driver only supports unlocking one segment. Therefore, in this thesis we may speak of unlocking a partition and mean unlocking one of the partition's segments.

To unlock a segment means to derive the cryptographic key that is needed for reading decrypted or writing encrypted data. This key is called the segment's *master key*.

LUKS2 uses a process called *anti-forensic splitting* to store the master key on disk. This method was introduced in [4]. It is used to diffuse the key's bytes into a longer sequence of bytes that has the following property: if at least one bit of the diffused sequence is changed, the key cannot be recovered. This is achieved by a clever combination of XOR and a hash function. The motivation behind this is to make it easier (or possible) to dispose of an old key in such a way that it cannot be recovered from the disk. This is because it is much more feasible to partially erase a long sequence of bytes than to completely erase a short sequence.[6] Erasing here means to overwrite the data in such a way that it cannot be recovered, which is not as trivial as one might think.

[4] calls the operation that splits data anti-forensically AFsplit and the recovery operation AFmerge. We will adhere to this convention.

[6] "If the probability to destroy a certain block of data is $0 \le p \le 1$, then the probability that the block survives is $1 - p$. Given a set of data consisting of n data items, the probability to erase the whole data set becomes worse since p^n becomes smaller as n becomes larger. But the probability that the whole set survives, $(1-p)^n$, becomes smaller as well with increasing n. If $(1-p)^n$ becomes smaller, than [sic] $1-(1-p)^n$ must become larger, which is exactly the chance that the whole block does not survive." [4]

```
0000  4C 55 4B 53 BA BE 00 02 00 00 00 00 00 00 40 00  LUKSº¾........@.
0010  00 00 00 00 00 00 00 03 54 68 69 73 20 69 73 20  ........This is 
0020  61 6E 20 41 53 43 49 49 20 6C 61 62 65 6C 00 00  an ASCII label..
0030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040  00 00 00 00 00 00 00 00 73 68 61 32 35 36 00 00  ........sha256..
0050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0060  00 00 00 00 00 00 00 00 EB 0F D2 C6 E3 D2 8D 4B  ........ë.ÒÆãÒ.K
0070  BB 2B 8A 49 E6 2E 4E B7 04 2F A9 39 76 71 8F 8A  »+ŠIæ.N·./©9vq.Š
0080  33 E8 F3 90 FF DC 4D 3D E8 30 7B 37 01 30 E7 5D  3èó.ÿÜM=è0{7.0ç]
0090  AD A0 57 1C 0E 63 BC D4 DD 3C EC F5 DE 67 F8 D8  ..W..c¼ÔÝ<ìõÞgøØ
00A0  F2 7E 82 CD B9 DD 77 10 65 39 33 64 63 61 66 61  ò~,Í¹Ýw.e93dcafa
00B0  2D 65 65 30 62 2D 34 31 36 38 2D 61 61 37 63 2D  -ee0b-4168-aa7c-
00C0  66 33 30 34 37 34 38 38 36 61 32 65 00 00 00 00  f30474886a2e....
00D0  54 68 69 73 20 69 73 20 61 6E 20 6F 70 74 69 6F  This is an optio
00E0  6E 61 6C 20 73 65 63 6F 6E 64 61 72 79 20 6C 61  nal secondary la
00F0  62 65 6C 00 00 00 00 00 00 00 00 00 00 00 00 00  bel.............
0100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01C0  91 A4 A9 83 03 FF FB 68 4E C2 94 6F 4C 78 71 AF  ‘¤©f.ÿûhNÂ”oLxq¯
01D0  AE 1A 91 F8 E0 2C F3 71 D5 17 CB 60 E5 2F D6 36  ®.‘øà,óqÕ.Ë‘å/Ö6
01E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figure 2.3: LUKS2 binary header example. The fields, as described in Figure 2.2, were coloured differently to be easily distinguishable. A similar header, although with different salt and hash, can be generated by executing `fallocate -l 16M luks2.img && cryptsetup luksFormat --label 'This is an ASCII label' --subsystem 'This is an optional secondary label' --uuid e93dcafa-ee0b-4168-aa7c-f30474886a2e luks2.img` in a Linux shell.

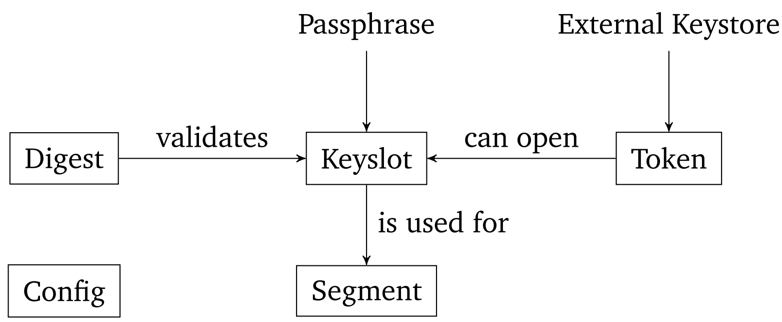

Figure 2.4: LUKS2 object schema from [2]. The most important objects are the following: *keyslots*, which describe the details of how cryptographic keys are stored and encrypted; *digests*, which can be used to verify that one has successfully extracted a key from a keyslot; and *segments*, which describe the disk areas where the encrypted user data is stored. Figure 2.1 shows where the areas described by the keyslot and segment objects actually lie on disk.

To necessitate the need of a password to recover the key, the data is also encrypted before it gets written to the disk. The encryption key is a hash of the password obtained by a PBKDF.

The properties of anti-forensic splitting can be used when the user wants to change the password: the master key is derived using the old password and then re-encrypted with the new password. The key as it was encrypted with the old password can then be destroyed.

[4] presents two templates for storing keys, TKS1 and TKS2. The difference is whether the key is encrypted before or after splitting it. LUKS and LUKS2 use TKS2, which is schematically explained in Figure 2.5. The hash function used by LUKS2 for anti-forensic splitting is SHA-256. Figure 2.6 shows the outline of an implementation of TKS2 in Rust.

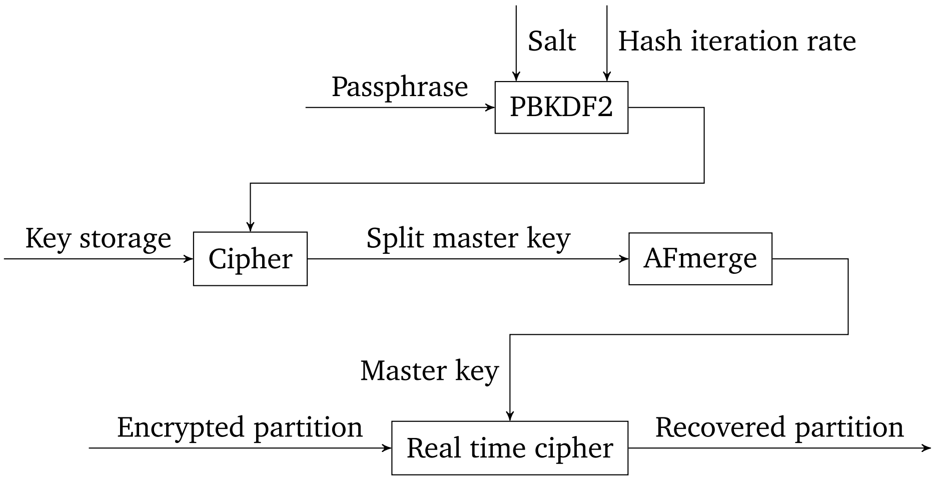

Figure 2.5: TKS2 scheme (modified after [4]).

The keyslots area on the disk is large enough to store multiple split and encrypted master keys. Thus one can configure a LUKS2 partition to be unlocked by different passwords. Unlocking then works as described in Figure 2.7.

2.1.4 Using an Unlocked Partition

After a partition has been unlocked, i.e. after the master key of one of its segments has been decrypted, the partition is ready to be read from and written to. This happens using what Figure 2.5 calls a real time cipher. LUKS2 supports different encryption algorithms for this purpose, see Table 2.1 for a selection of them. Our driver only supports the default aes-xts-plain64 encryption. Therefore, we will focus on that in this section.

The AES encryption algorithm is a block cipher for processing 128 bit data blocks [7]. This means that AES takes a key and 128 bits, or 16 bytes, of data, and generates 16 bytes of encrypted data, called ciphertext. Using the same key, this ciphertext can be decrypted back to the original plaintext data. The key can be 128, 192, or 256 bits long. The three variants of AES, each using a different key size, are called AES-128, AES-192, and AES-256.

To encrypt data longer than one block, a *block cipher mode* is needed [6]. These are encryption functions that build on an existing block cipher. When using aes-xts-plain64, LUKS2 uses the XTS block cipher mode. The defining IEEE standard [8] describes it as follows: “XTS-AES is a tweakable block cipher that acts on data units of 128 b[its] or more and uses the AES block cipher as a subroutine. The key material for XTS-AES consists of a data encryption key (used by the AES block cipher) as well as a ‘tweak key’

```
1  fn decrypt_keyslot(
2      password: &[u8], keyslot: &LuksKeyslot, json: &LuksJson, /* ... */
3  ) -> Result<Vec<u8>, LuksError> {
4      let mut k = vec![
5          0; keyslot.key_size() as usize * keyslot.af.stripes() as usize
6      ];
7      // read keyslot data from disk into k...
8
9      let mut pw_hash = vec![0; area.key_size() as usize];
10     match keyslot.kdf() {
11         // hash into pw_hash using pbkdf2, argon2i, or argon2id...
12     }
13
14     // decrypt keyslot area using the password hash as key
15     match area.key_size() {
16         32 => {
17             let key1 = Aes128::new_varkey(&pw_hash[..16]).unwrap();
18             let key2 = Aes128::new_varkey(&pw_hash[16..]).unwrap();
19             let xts = Xts128::<Aes128>::new(key1, key2);
20             xts.decrypt_area(&mut k, sector_size, 0, get_tweak_default);
21         },
22         // 64 byte key uses AES256 instead...
23     }
24
25     // merge and hash master key
26     let master_key = af::merge(
27         &k, keyslot.key_size() as usize, af.stripes() as usize
28     );
29     let digest_actual = base64::decode(json.digests[&0].digest())?;
30     let mut digest_computed = vec![0; digest_actual.len()];
31     let salt = base64::decode(json.digests[&0].salt())?;
32     pbkdf2::pbkdf2::<Hmac<Sha256>>(
33         &master_key, &salt, json.digests[&0].iterations(), &mut digest_computed
34     );
35
36     // compare digests
37     if digest_computed == digest_actual {
38         Ok(master_key)
39     } else {
40         Err(LuksError::InvalidPassword)
41     }
42 }
```

Figure 2.6: LUKS2 master key decryption in Rust. Some values are hardcoded: only the digest with index 0 is used (lines 29, 31, 33), and it is assumed that the digest algorithm is always `pbkdf2` with SHA-256 (line 32). The latter is compliant with the specification, which lists this digest algorithm as the only option, but not optimal in the sense of input validation.

that is used to incorporate the logical position of the data block into the encryption." This tweak key is called the *initialization vector*, or IV, in the context of LUKS2. This is what the "plain64" in aes-xts-plain64 means: "the initial vector is the 64-bit little-endian version of the sector number, padded with zeros if necessary" [9]. This sector number is relative to the first sector of the segment, i.e. the first sector uses an IV of 0.

The need for an IV arises from a critical problem that occurs when encrypting each

1. The user supplies a password.
2. One of the available keyslots is selected.
3. Using the password and keyslot, the master key is decrypted as described above.
4. The derived master key is hashed and the result compared to the corresponding digest. Which digest and what hash parameters to be used is defined in the JSON section.
5. If the digests match, the master key has been successfully decrypted. Else go to step 2 and select a keyslot that has not been used yet.
6. If the master key could not be decrypted with all available keyslots, the supplied password was not correct.

Figure 2.7: LUKS2 master key decryption with multiple available keyslots. LUKS2 also allows defining priorities that govern the order in which the available keyslots are tried. For a more detailed pseudocode see Figure 5 in [5].

Algorithm in `dm-crypt` notation	**Description**
aes-xts-plain64	AES in XTS mode with sequential IV
aes-cbc-essiv:sha256	AES in CBC mode with ESSIV IV
serpent-xts-plain64	Serpent cipher with sequential IV
twofish-xts-plain64	Twofish cipher with sequential IV

Table 2.1: Selection of LUKS2 encryption algorithms (modified after [2]). The `dm-crypt` algorithm notation is described in Section 4.1.1. See [6] for the CBC mode and the Serpent and Twofish ciphers, and [4] for the ESSIV IV mode.

block separately with the same key: if some unencrypted blocks are identical, then so will be their encrypted counterparts. This can lead to leaked information about structure and contents of the plaintext [6].

All this theory may sound complicated, but in Section 5.2.5 we will see that the practical usage of cryptography in our driver is quite simple.[7]

2.2 Introduction to Windows Kernel Driver Development

Writing a Windows kernel driver is a non-trivial endeavour. Therefore, this chapter will describe relevant parts of Windows' internal architecture, answer the question of what a kernel driver is in the first place, and introduce the most important concepts for writing kernel drivers.

[7] The implementation of cryptographic algorithms like the XTS mode of course remains non-trivial. Implementing AES itself has been made much easier using the AES-NI CPU instruction set, though.

2.2.1 Structure and Hierarchy of the Windows Operating System

First, we will introduce some basic concepts of the Windows operating system that will be relevant in the following sections. Please note that we will focus on Windows 10 running on the x64 architecture, as that is the target of our driver. Some details may be different in other versions of Windows, though most information should still be valid.

The definitive reference for all detailed information on the inner workings of Windows is [10]. The online documentation of the Win32 API [11] and the Windows Driver Kit (WDK) [12] both provide valuable information. It may also sometimes be useful to directly look at the data structure definitions provided by the WDK header files (`ntddk.h`, `ntifs.h`, and `wdm.h` are the most useful) [10].

There are also some invaluable tools to directly take a look at OS internals. Most of them are from the Sysinternals Suite,[8] others are included with Windows. See table 1-4 in [10] for a more detailed compilation.

The Windows *registry* is a database for storing system-wide and per-user configuration. It can also be used to query the current state of the system, e.g. performance counters or information on loaded device drivers [10]. Each data entry in the registry, called a *value*, has a path, also known as its *key*, and a name. This name is used to distinguish different entries stored under the same key. The keys are organized hierarchically in a tree, similar to file paths [11]. In the context of registry keys, `HKLM` stands for `HKEY_LOCAL_MACHINE`.

Windows uses the physically available RAM to back its 64-bit address space of virtual memory. See Figure 2.8 for how the address range is used. The addresses are grouped into so-called *pages*, which are typically 4KB large. Most systems have less physical RAM available than the sum of virtual memory used by all processes. The memory manager solves this problem by transferring pages currently not in use to disk. This is called *paging*. When a virtual address in a page that currently resides on disk is accessed, a *page fault* occurs. In that case, the needed page is loaded back into memory. This process is completely transparent to applications, aside from latency introduced by paging. However, this will become relevant when writing drivers, as driver code may be called in situations where page faults are not allowed (see Section 2.2.2) [10].

Figure 2.9 shows a simplified view of the architecture of Windows. It also differentiates between *user mode* and *kernel mode*. The difference is that when a process is running in kernel mode, it has access to all CPU instructions, the whole system memory. Kernel mode also allows direct access to hardware. User mode, on the other hand, only allows access to a limited subset of all that. To protect the OS from user applications and to also isolate different programs from each other, user applications run in user mode. Kernel mode is reserved for OS code, such as drivers and system services. This separation ensures that an incorrectly programmed or malicious program cannot jeopardize the whole system's stability and/or data integrity [10].

The permission checks for user mode are enforced by the processor. Switching between user and kernel mode is achieved by executing a special CPU instruction, on x64 usually `syscall`. Prior to that, the user code specifies which system service it wants the kernel to execute.[9] When kernel mode is activated, control is transferred from the application to a special part of the operating system. Its purpose is to dispatch control to the part of the OS that implements the requested system service. When that has

[8] The executables can be downloaded from `https://live.sysinternals.com` and are accompanied by documentation in [13].

[9] These system services are also referred to as system calls or syscalls, especially in the context of Linux.

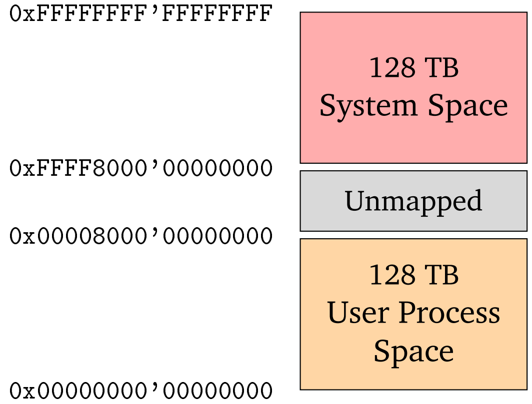

Figure 2.8: Address space layout (not to scale) for 64-bit Windows 10 (modified after [10]). Each process has its own copy of the user process space.
(Sven Peter has a nice quote on this topic on his blog: "One of the kernel's tasks is to lie to each application running in userland and to tell them that they're the only one in the address space.").

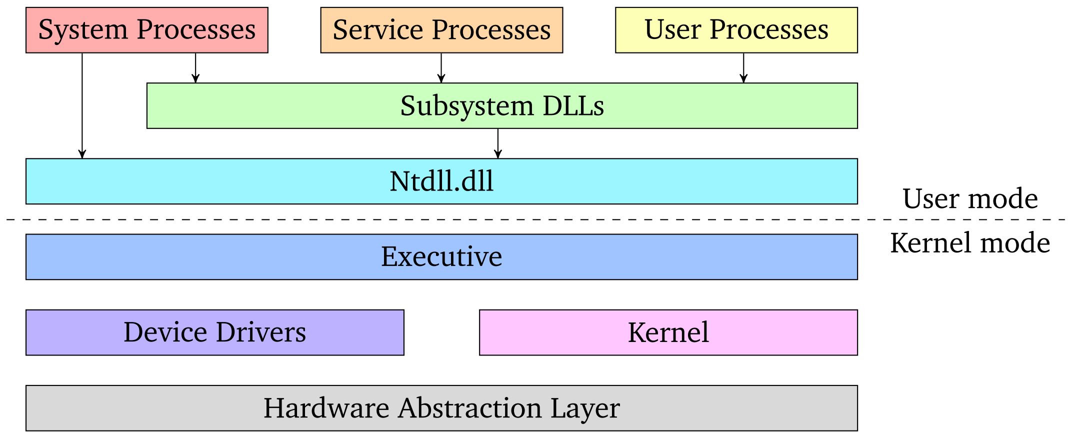

Figure 2.9: Simplified Windows architecture (modified after [10]). User processes are regular applications. The service processes are responsible for hosting Windows services, such as the printer spooler or DNS client. System processes are fixed processes that are not started by the Service Control Manager. The Session Manager and the logon process are examples for system processes. The subsystem dynamically linked libraries (DLLs) are responsible for translating public API functions into their corresponding internal system service calls. The latter are mostly implemented in Ntdll.dll. The executive, kernel, and Hardware Abstraction Layer are described in detail later on.

been completed, the OS orders the processor to switch back into user mode and hands control back to the application [10].

Developers of regular applications will not come into contact with syscalls. Normally, these are the responsibility of Windows API subroutines. For example, the `CreateProcess` function will instruct the OS to create a new process by executing the `NtCreateUserProcess` syscall. Because they are not meant to be used by non-OS code, these system calls are not (officially) documented[10] [10].

[10] There are unofficial resources that attempt to map out the space of Windows syscalls, even across different versions, e.g. `https://j00ru.vexillium.org/syscalls/nt/64`.

As mentioned before, each user process has its own private address space. In contrast to that, all kernel mode OS components and device drivers share one area of memory. Furthermore, all user mode addresses can also be accessed from kernel mode. Pages can however be marked as read-only, a restriction which not even the kernel can circumvent [10].

The fact that there exists only one global kernel address space means that special care must be taken when writing code that runs in kernel mode. A single malfunctioning driver can destabilize the complete system or even introduce critical security vulnerabilities. To aid in discovering bugs in drivers, Windows provides a *Driver Verifier* tool [10].

It is also necessary to carefully choose which code is allowed to run in kernel mode. This is why Microsoft mandates that all third-party drivers for Windows 10 must be signed by one of two accepted certification authorities. They also have to be reviewed and signed by Microsoft. For testing purposes, Windows can be configured to load self-signed drivers. This is known as *test mode* [10].

One part of the OS that runs in kernel mode is the *Windows executive*, which was already mentioned in Figure 2.9. It is responsible for multiple things, including [10]:

- The implementations of the aforementioned system calls reside in the executive.
- The executive also provides implementations of functions that can be used by other kernel mode components. Some of them are documented, others are not. They generally fall into one of four categories: managing Windows executive objects that represent OS resources; message passing between client and server processes on the same machine; run-time helpers for e.g. processing strings or converting datatypes; and general support routines for allocating and accessing memory as well as some synchronization mechanisms, e.g. fast mutexes.
- The executive can be categorized into different components, of which the following are most relevant when developing a device driver: the I/O manager, the Plug and Play (PnP) manager, and the Power manager. The functions exported by these components generally follow a naming scheme were their names start with `Io`, `Pp`, and `Po`, respectively. Please refer to table 2-5 in [10] for a more exhaustive list of commonly used prefixes of function names.
- Shareable resources, e.g. files, processes, and events, are represented as objects by the executive to user mode code. The state of the represented resource is captured in data fields called the object's attributes. Objects are *opaque*, i.e. their attributes can only be accessed through object methods. This makes the internal implementations of objects easily changeable.[11] The object methods may also implement certain security and access checks. A reference to a object is called a *handle*. Keeping track of these references allows the system to automatically recognize and deallocate objects that are no longer in use.

Another major kernel mode OS component is the *Windows kernel*, also already presented in Figure 2.9. Its tasks include the following [10]:

- It takes over the task of thread scheduling and (multi-processor) synchronization and also dispatches interrupts and exceptions. Aside from that, it avoids making policy decisions, and leaves this to the executive.

[11] One can see why these representations were named objects, as they follow some of the core concepts of object-oriented programming.

- In general, one of the kernel's jobs is to conceal some of the differences of the different hardware architectures that Windows supports. It is assisted in this task by the HAL.

- Intended for usage by other, higher-level components, including the executive, it provides low-level routines and basic data structures. These are partly documented and may be used in device drivers. The function names all start with the `Ke` prefix.

- The exposed data structures are called *kernel objects*. These are simpler than the executive's objects and therefore have no overhead for e.g. manipulation via handles or security checks. They are the building blocks for most executive-level objects.

Looking at Figure 2.9, one can see that quite a lot of the OS runs in kernel mode. Moving parts of it to user mode has some advantages: the OS becomes more stable, because a crash in user mode does not crash the whole system (which is what happens when kernel mode code crashes); the OS becomes more secure, because a user mode process does not have access to kernel space memory; and programming the OS becomes simpler, because user mode code does not have to think about as many things as kernel mode code (see the following two sections for examples of what kernel mode drivers need to keep in mind). Windows allows implementing certain parts of the OS in user mode using the User-Mode Driver Framework (see the following section) [14]. This moves Windows in the direction of a *microkernel* system. [15] explains this concept as follows: "The basic idea behind the microkernel design is to achieve high reliability by splitting the operating system up into small, well-defined modules, only one of which – the microkernel – runs in kernel mode and the rest run as relatively powerless ordinary user processes."

Finally, the last component from Figure 2.9 to be described is the *Hardware Abstraction Layer (HAL)*. Its purpose is to separate the kernel and device drivers from hardware specifics, e.g. differences between motherboards. It is implemented as a kernel module that is loaded at system boot. In the case of the x64 and ARM architecture, all systems of these respective types are similar enough that one module can support them all: there is just one `Hal.dll` for x64, and one for ARM. For other architectures, such as x86 and different embedded platforms, multiple HAL implementations may exist. Windows decides at boot which specific HAL implementation will be loaded, depending on the detected architecture and hardware. It is also possible to extend a basic HAL implementation using HAL extensions, which the boot loader loads if it notices hardware that requires an extension [10]. The functionalities of the HAL that are intended to be used by drivers (i.e. the ones that are documented in the WDK) include [12]:

- working with hardware performance counters to monitor the system's performance;

- utilities for directly accessing system buses, although the methods for this are mostly deprecated. The new, supported way is getting a function pointer by other means[12] and calling the function it points to;

- utilities for direct memory access (DMA) (see also footnote 16). There exist old methods that use the `Hal` prefix, but they have been deprecated in favour of other

[12] Using IRPs, which are described in Section 2.2.3.

methods without this prefix or with other prefixes. It is possible, or maybe even likely, that these newer functions internally still make use of the HAL.

2.2.2 Windows Kernel Drivers and the Windows Driver Model

Windows kernel drivers are modules that can be loaded and run in kernel mode. They are normally found in the form of files with the `.sys` extension. Drivers can be categorized as follows [10]:

- **Device drivers** As an interface between hardware and the I/O manager, these make use of the HAL to control retrieving input from or writing output to hardware.
- **File system drivers** These translate file-oriented requests into raw I/O requests.
- **File system filter drivers** These process incoming I/O requests or outgoing responses before passing them to the next driver. They can also reject an operation. The driver that this thesis is about falls into this category.
- **Network drivers** These either implement networking protocols, like TCP/IP, or handle the transport of file system I/O between machines on a network.
- **Streaming filter drivers** These are for data stream signal processing.
- **Software drivers** These are helper kernel modules for user mode programs. They are used when a desired operation, e.g. retrieving internal system information, is only available in kernel mode.

Drivers can be written using different models or frameworks. Windows NT 3.1 introduced the first driver model, which lacked PnP functionality. Windows 2000 introduced PnP support as well as an extended driver model known as the *Windows Driver Model (WDM)* [10]. We will focus on WDM in this section, because that is the framework our driver uses (see Section 5.2). But there also exist other models and frameworks for Windows driver development, such as the Windows Driver Frameworks (WDF) [14]. These try to simplify things and consist of the Kernel-Mode Driver Framework (KMDF) and the User-Mode Driver Framework (UMDF) [10]. We will explore some of the differences between them and WDM in Section 5.1.

WDM distinguishes between three types of drivers [10]:

- **Bus drivers** These are responsible for managing devices that have child devices, such as bus controllers, bridges, or adapters. They are generally provided by Microsoft because they are required for widely used buses like PCI and USB. It is nevertheless also possible for third parties to implement support for new bus types by supplying a bus driver.
- **Function drivers** "A function driver is the main device driver and provides the operational interface for its device." [10] Every device needs a function driver, if it is not used raw. This is a mode of operation where the bus driver manages I/O together with bus filter drivers.
- **Filter drivers** These enhance an already existing driver or device by either adding functionality or modifying I/O requests and/or responses from other drivers. They are optional and the number of filter drivers for a device is not limited.

In this model, the control over a device is not concentrated at one single driver, but rather shared between multiple drivers. A bus driver only informs the PnP manager about the devices connected to its bus, while the device's function driver handles the actual device manipulation and control. The drivers responsible for a device form a hierarchy called the *device stack*. This defines the order in which the drivers process incoming and outgoing messages to or from the device: the uppermost driver receives incoming requests first and the lowermost one receives them last. For outgoing driver responses, this order is reversed [10]. See Figure 2.10 for an example of how such a device stack can look.

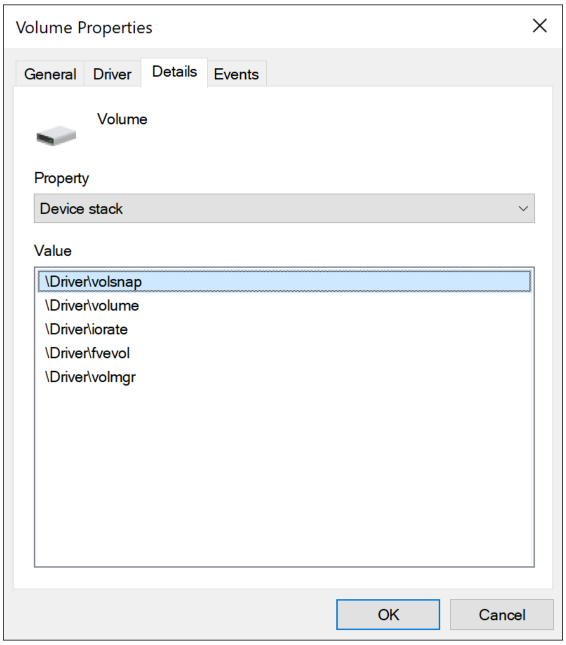

Figure 2.10: Example of a device stack for a storage volume (screenshot from device properties in Windows Device Manager). The official descriptions of the drivers listed in this example are, from top to bottom: "Volume Shadow Copy driver", "Volume driver", "I/O rate control filter", "BitLocker Drive Encryption Driver", and "Volume Manager Driver". As can be seen under the appropriate registry key, `fvevol` and `iorate` are lower filter drivers and `volsnap` is an upper filter driver. This leaves `volume` as the function driver and `volmgr` as the bus driver.
Later on in this section we will talk about the registry's role in driver management in more detail. See Figure 2.12 for what key to use and how the values stored under that key may look.

The device stack is built by the PnP manager during a process called *device enumeration*. This happens at system boot or when resuming from hibernation, but it can also be triggered manually. During enumeration, the PnP manager constructs a so-called *device tree* that captures the parent-child relationships of devices. A node in this tree represents a physical device and is appropriately called a *device node*, or short *devnode*. When a new device is discovered, its required drivers are loaded (if they have not already been loaded). The drivers are then informed about the new device, and can decide whether they want to be part of the device stack or not [10].

For all the drivers of a device, the PnP manager creates objects that are used for managing and representing the relationships between the drivers, i.e. the device stack.

They are also a form of device object, however they don't represent a real, physical device, but rather a virtual one. See Figure 2.11 for an example device node and the objects it is made of. There are the following types of objects in a device node, ordered from bottom to top by their occurrence in a devnode [10]:

- exactly one *physical device object (PDO)*, created by the bus driver. It is responsible for the physical device interface.
- zero or more *filter device objects (FiDOs)*, created by lower filter drivers (lower being relative to the FDO).
- exactly one *functional device object (FDO)*, created by the device's function driver. It is responsible for the logical device interface.
- zero or more FiDOs, created by upper filter drivers.

This shows that filter drivers can be placed either above the function driver (upper filter drivers) or between the function driver and the bus driver (lower filter drivers). These serve different purposes: "In most cases, lower-level filter drivers modify the behavior of device hardware. For example, if a device reports to its bus driver that it requires 4 I/O ports when it actually requires 16 I/O ports, a lower-level, device-specific function filter driver could intercept the list of hardware resources reported by the bus driver to the PnP manager and update the count of I/O ports. Upper-level filter drivers usually provide added-value features for a device. For example, an upper-level device filter driver for a disk can enforce additional security checks." [10]

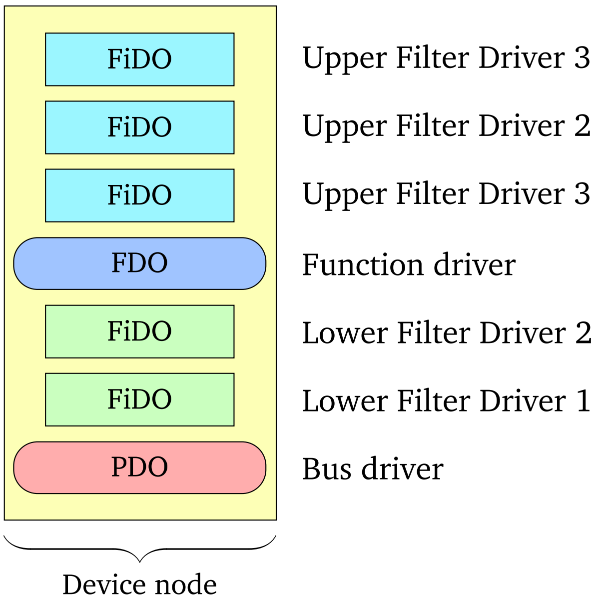

Figure 2.11: A device node and its device stack (modified after [10]). See Figure 2.10 for a concrete example of a device stack.

As already hinted at in the description of Figure 2.10, the registry is responsible for storing the information that governs driver loading. Which drivers are loaded and the order they are loaded in are both configured this way. Basically, it is described how exactly the device nodes should be built. Table 2.2 lists the registry keys that, together with their subkeys, are relevant for the configuration. In Section 5.2.2, we will see how some of these registry entries are set up at driver installation.

The keys from Table 2.2 each have different roles [10]:

Registry Key	Short Name	Description
`HKLM\System\CCS\Enum`	Hardware key	Settings for known hardware devices
`HKLM\System\CCS\Control\Class`	Class key	Settings for device types
`HKLM\System\CCS\Services`	Software key	Settings for drivers

Table 2.2: Important registry keys for driver loading [10]. Note that `CCS` stands for `CurrentControlSet`.

- The Hardware key configures driver loading for individual hardware devices. Its subkeys all follow the pattern of `<Enumerator>\<Device ID>\<Instance ID>`. The enumerator is the name of the responsible bus driver; the device ID identifies a particular type of hardware, e.g. a specific model from one manufacturer; and the instance ID distinguishes different devices with the same device ID, e.g. the device's location on the bus or its serial number. The most important values for each device are: `Service`, which contains the name of the device's function driver (this is used as a subkey in the Software key); `LowerFilters` and `UpperFilters`, which each contain an ordered list of lower and upper filter driver names, respectively; and `ClassGUID`, which identifies the device's class (this is used as a subkey in the Class key).
- The Class key contains class GUIDs[13] as subkeys. The configuration stored here is similar to the subkeys of the Hardware key, but applies to a whole class of devices. For example, the lower and upper filters listed under the GUID for the "Volume" class are loaded for all volumes.
- The Software key contains relevant information about drivers, most importantly the `ImagePath` value, which stores the path to the driver's `.sys` file. The subkey that contains the values for a driver is the driver's name.

Examples of subkeys of each of these keys are shown in Figure 2.12.

2.2.3 Important Concepts for Kernel Driver Development

The goal of this section is to explain some concepts that are relevant when developing a file system filter driver. They concern execution contexts, memory access, and the kernel's I/O system.

Thread Contexts and IRQLs The code of a driver can run in different thread contexts, depending on how it was called. This is mostly relevant for accessing memory, because the translation of virtual memory addresses into physical addresses depends on the thread context.[14] We will discuss what exactly to keep in mind when accessing memory shortly. The three different contexts that driver code can run in are the following [10]:

- when called to handle an I/O request initiated by a user thread, the driver code runs in this thread's context;

[13] Globally Unique Identifiers, also known as Universally Unique Identifiers (UUIDs). See RFC 4122 for more information.

[14] This is an inherent property of the concept of virtual memory and applies not only to Windows, but to all virtual memory OSes, including Linux.

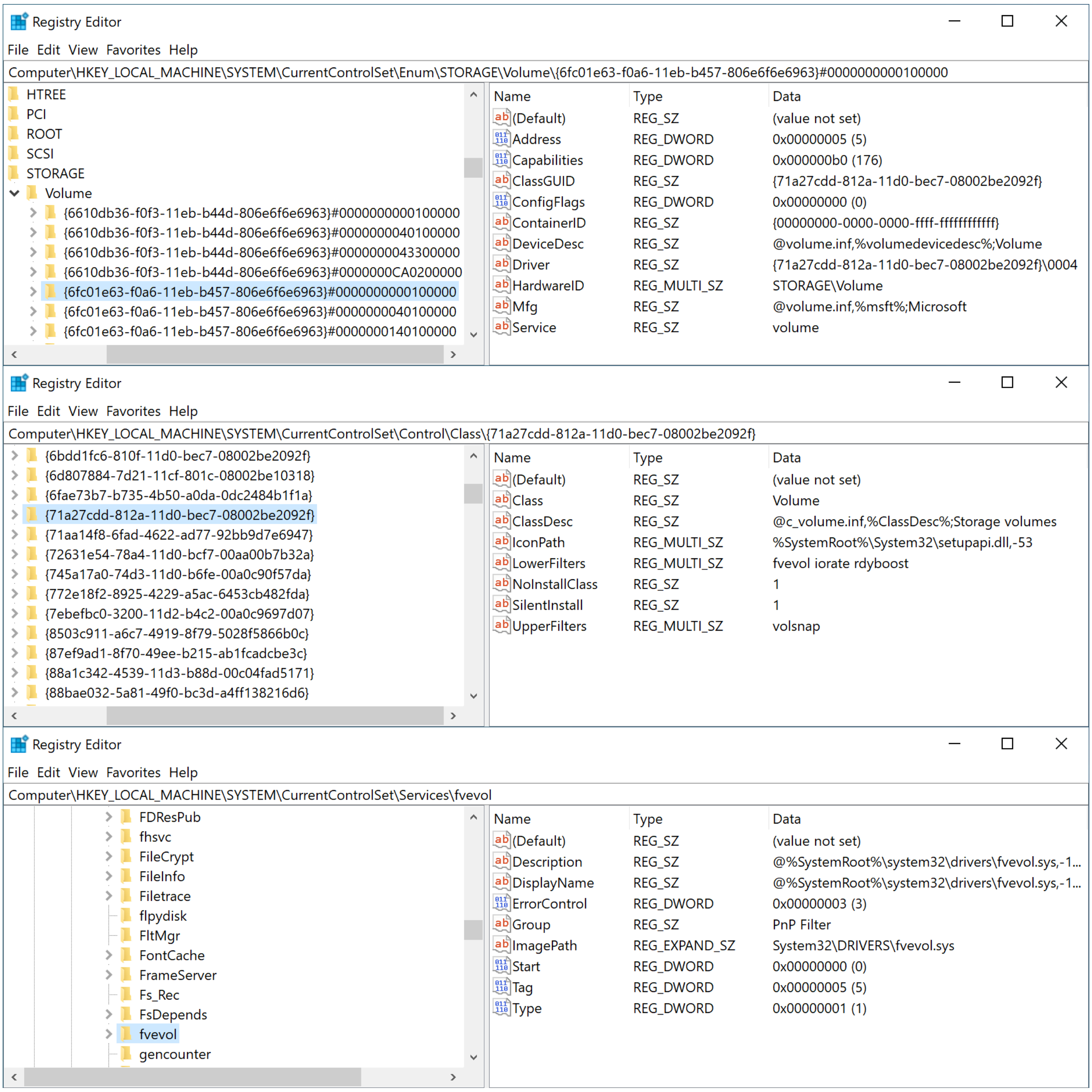

Figure 2.12: Example contents of a Hardware, Class, and Software subkey (screenshots from the Windows Registry Editor). These are all relevant for the device whose device stack is shown in Figure 2.10. As already partly discussed there, the configured values match the final device stack: the `Service` value under the Hardware subkey designates `volume` as the function driver; the drivers listed in the `LowerFilters` and `UpperFilters` values of the Class subkey are indeed loaded before or after `volume`, respectively (also note that they are loaded, from bottom to top, in the order listed); the `ImagePath` under the Software subkey shows the path to one of the driver's `.sys` file.
The `readyboost` driver is listed as a lower filter, but does not appear in the device stack, probably because it decided not to be part of it. This makes sense, because it is responsible for the ReadyBoost technology, which is designed for SD cards and USB sticks, and not hard disk volumes. See [10] for more information on this technology.

– when called by a OS component, e.g. the PnP manager, the driver code runs in the context of the calling system thread;

– when called via an interrupt, it is not possible to predict in which context the driver code will run. The context of whatever thread was currently executing when the interrupt arrived will be used. The interrupt may stem from hardware, for example when a device informs about available data, or from software, for example when a timer fires.

Note that, regardless of the context, the code always executes in kernel mode. When running in the context of a user mode thread, the CPU switches to kernel mode before running the driver code [10].

Another important parameter of code execution is the so-called *interrupt request level (IRQL)*. As the name suggests, this has something to do with interrupts, but for this thesis it suffices to talk about one important property of this numerical value: code running at a lower IRQL cannot interrupt code running at a higher IRQL. We will discuss the implications of this in a moment. The normal IRQL of the CPU is 0, also called `PASSIVE_LEVEL`. This is also the only possible value when in user mode. Another possible value for kernel mode code, more or less the only other one non-device drivers (such as filter drivers) will come in contact with, is an IRQL of 2. This is known as `DISPATCH_LEVEL`. We will present a concrete example of code that can run at `DISPATCH_LEVEL` when talking about the I/O system. The important thing is that the kernel's thread scheduler runs at this IRQL. Because of the property mentioned earlier, this means that the scheduler cannot interrupt code running at `DISPATCH_LEVEL`. The immediate consequences of this are [10]:

– Synchronization objects that rely on the scheduler's support, such as semaphores or mutexes, no longer work. The problem is that when waiting on one of these objects, the thread goes to sleep. But the scheduler can't run, because that would mean interrupting the thread, and thus it can never wake up the thread again. Windows prevents this by checking the IRQL and crashing if the current IRQL has a value of 2.

– Page faults must not occur. Handling them requires a context switch, but for that the scheduler would need to work. Therefore, code at `DISPATCH_LEVEL` can only access memory that is not currently paged to disk. This can be ensured by allocating the memory in a special pool which, by definition, always is resident in memory: the *non-paged pool*.[15]

In general, when running at an IRQL of 2, every called function needs to be checked for the ability to be called at this level. The supported IRQL is documented for all WDK functions.

User Memory Access and I/O Modes As mentioned before, access to memory in userspace generally requires special care. To be precise, the following problems can occur [10]:

– Recall that the translation of user mode virtual addresses to physical addresses depends on the thread context. Referencing an address from a different context than it belongs to either leads to an access violation or accesses random data from another process.

[15] There also exists an allocation pool without this special guarantee, called the *paged pool*.

– Userspace memory always has the risk of being paged out. Code running at an IRQL of 2 must ensure that this is not the case before accessing memory. As discussed before, page faults are illegal in this state.

One situation where a driver needs to access user memory is doing I/O, i.e. read or write operations. Because these are so common, the I/O manager presents a solution, namely the *Buffered I/O* and the *Direct I/O* modes [10]:

– **Buffered I/O** When using this mode, the I/O manager copies the data from the user buffer to a newly allocated kernel buffer from the non-paged pool (or vice versa, depending on whether it is a read or write operation). This means the driver does not have to care about accessing the user buffer at all and can just work with the kernel buffer. Because it was allocated from non-paged pool, no page fault can occur, and as a system space address, the kernel buffer address is valid in any thread context. See Figure 2.13 for how this works when reading data. Buffered I/O is commonly used for small buffers (up to one page, or 4KB).

– **Direct I/O** In this mode, the I/O manager locks the user buffer in RAM, so that it cannot be paged out to disk. The driver can then create a mapping of the buffer in kernel memory and just use that, without any copying. To do so, the I/O manager provides the driver with a MDL, which stands for *memory descriptor list*. It holds information about the physical memory occupied by the user buffer, and it is needed to map the buffer into system space. At this point, the buffer's physical memory is mapped twice: once into user memory, and once into kernel memory. Both mappings can be accessed at any IRQL, because the buffer cannot be paged out. However, while the user mapping is only valid in one thread context, the kernel mapping is always valid. After the operation has completed, the I/O manager removes the kernel mapping and unlocks the buffer. Figure 2.14 illustrates this process. Direct I/O is useful when using large buffers (larger than one page, or 4KB), because no copying is needed.[16]

It is up to the driver to decide which I/O mode to use. Their choice is signalled to the I/O manager using a flag of their virtual device's object (the one that is part of the device node). This applies to all I/O operations, the exception being so-called *device I/O control requests*. These are the Windows equivalent to Unix' `ioctl` syscall and are used to communicate with a device from user mode.[17] A device control code, specifying what exactly the user wants from the device, also defines the used I/O mode, and drivers must adhere to this [10].

In situations where it is not too complicated, drivers can choose to handle the user buffer access completely by themselves. This mode is known as *Neither I/O*. It has the advantage of zero overhead, because the I/O manager does no extra work. It does however increase the risk of bugs and possible security risks. See [10] for more information.

The Windows Kernel's I/O System I/O operations have been mentioned a lot in this section. In the following paragraphs, the important details of the kernel's I/O system will be laid out.

[16] It is also more suitable than buffered I/O for devices with direct memory access (DMA), "because DMA is used to transfer data from a device to RAM or vice versa without CPU intervention – but with buffered I/O, there is always copying done with the CPU, which makes DMA pointless." [10]

[17] An example is `IOCTL_DISK_GET_DRIVE_GEOMETRY`, which is used get the physical layout of a disk. See `www.ioctls.net` for a comprehensive (unofficial) list of almost all relevant control codes and the documentation for `DeviceIoControl` in [11] for information on how to use them.

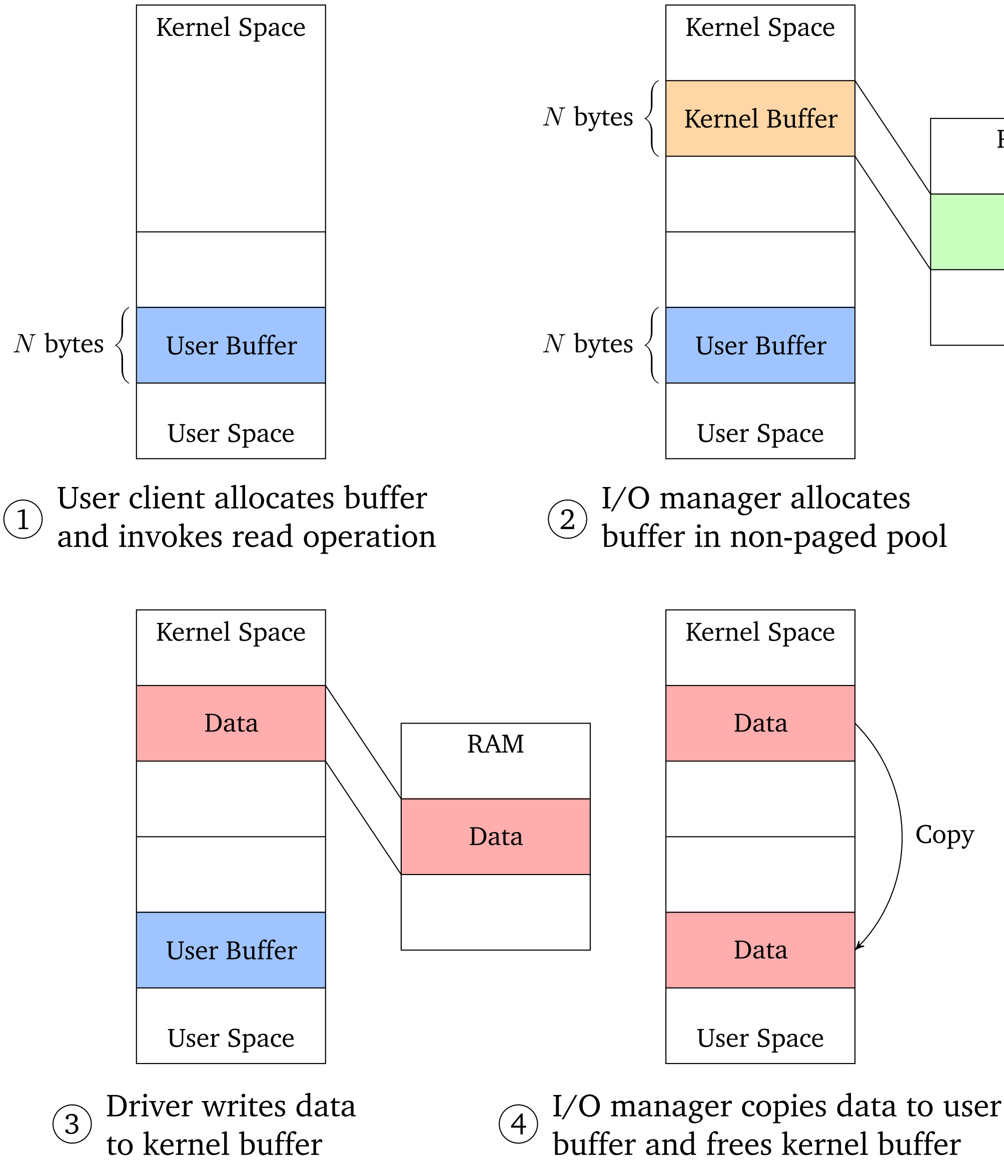

Figure 2.13: Buffered I/O for a read operation (modified after [10]). For write operations, the copying is done before the driver is called to handle the request.

Firstly, all I/O has a *virtual file* as its target. This may be backed by a "real" file or by something different, like a device.[18] This abstraction allows the I/O manager to only care about files, leaving the translation of file commands to device-specific operations to drivers [10].

Secondly, Windows' internal I/O system is packet-driven. Almost all I/O requests[19] travel through the different relevant parts of the system in the form of an *I/O request packet (IRP)*, which contains all relevant information about the request. IRPs are allocated in non-paged pool, usually by the I/O manager[20], but it is also possible for drivers to initiate I/O by creating an IRP. After an IRP's allocation and initialization, a pointer to it is passed to the driver that shall handle the request. The driver then performs its

[18] This concept will be familiar to Linux users.

[19] The exception is so-called *Fast I/O*, which works differently [10]. It is not supported by our driver and will therefore not be covered.

[20] The PnP or power manager also sometimes allocate IRPs [10]. For simplicity, we will write "the I/O manager" instead of "the I/O, PnP, or power manager" in the following paragraphs.

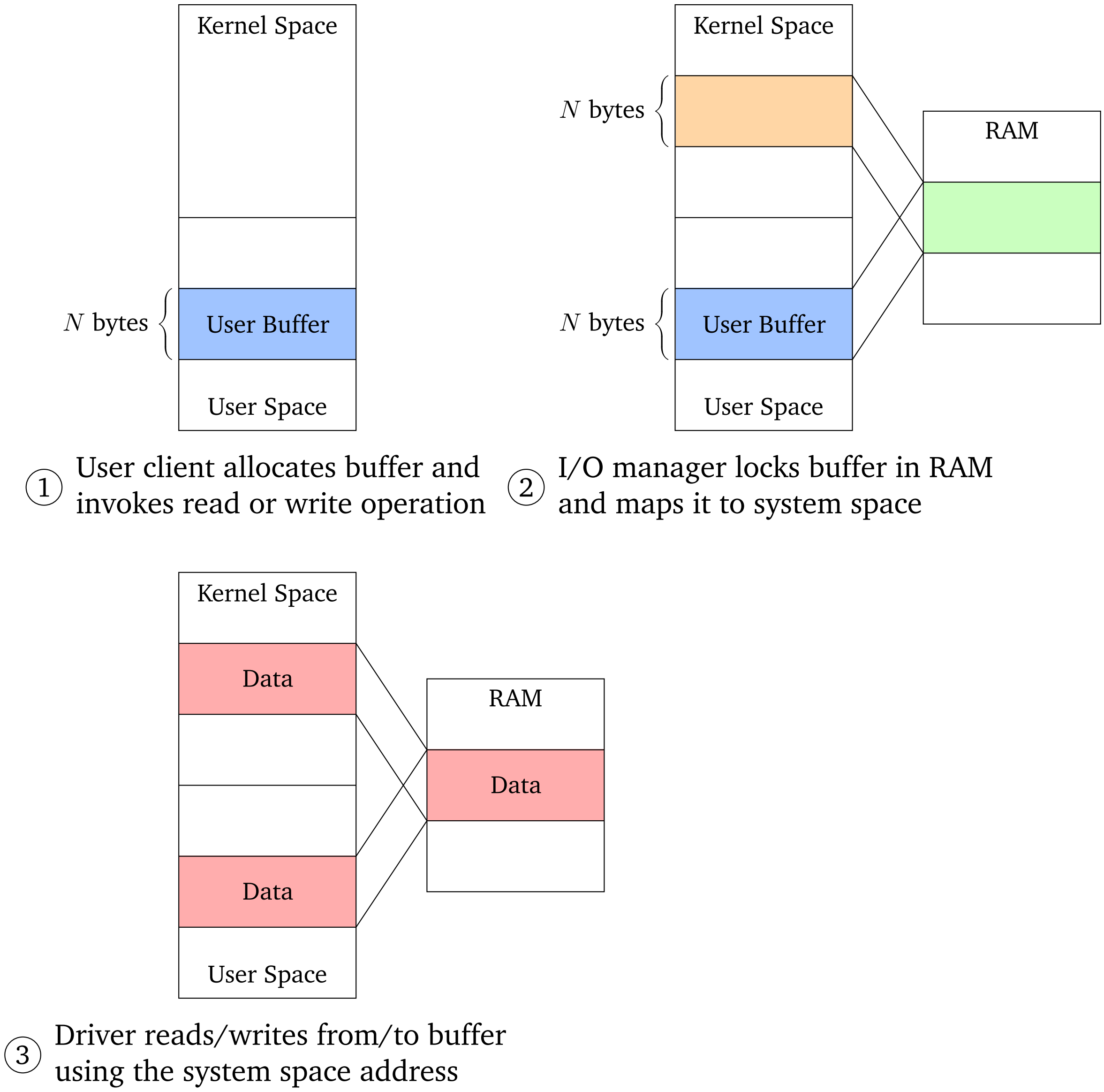

Figure 2.14: Direct I/O for a read or write operation (modified after [10]).

operation and returns the IRP to the I/O manager. It also reports whether it was able to complete the requested operation or whether further work by another driver needs to be done [10].

An IRP holds a lot of information. For this thesis, it suffices to know about the following fields[21] [10]:

- a pointer to a MDL, if the driver uses the direct I/O method;
- a pointer to a system buffer, if the driver uses the buffered I/O method;
- information about an IRP's final status, if it has been completed;
- information about the IRP's stack locations, which will be explained momentarily.

In memory, the IRP is followed by one or more *stack locations*, their count matching

[21] See [10] and its documentation in [12] for more details.

the number of drivers this IRP will pass through.[22] These stack locations are represented by `IO_STACK_LOCATION` structures, whose contents are shown and explained in Figure 2.15. Because the information about the request is split between the IRP and its stack location, drivers can modify the stack location parameters for the next driver, while keeping the current set of parameters. The original parameters may for example be useful in a completion routine [10].

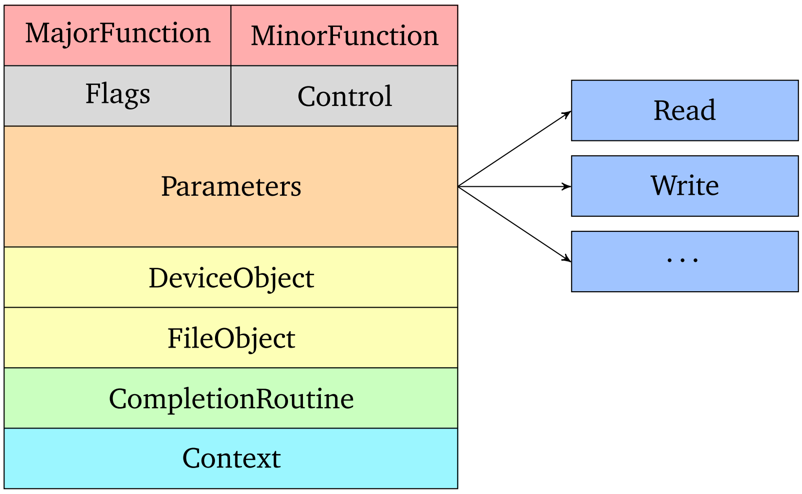

Figure 2.15: Layout of the `IO_STACK_LOCATION` structure (modified after [10] using information from [12]). The `MajorFunction` field has one of 28 possible values, denoting the request type, e.g. `IRP_MJ_READ` or `IRP_MJ_WRITE`. `MinorFunction` is used to further specify the type for some of the major functions. We will not discuss `Flags` and `Control`, because they are only rarely relevant. The `Parameters` are "a monstrous union of structures, each of which valid for a particular major function code or a combination of major/minor codes." [10] They contain information further specifying the request, e.g. the byte offset when reading from a file. `DeviceObject` and `FileObject` are pointers to the related device and file objects. `CompletionRoutine` holds a pointer to a completion routine, which will be explained separately. `Context` is a driver-defined parameter passed to the completion routine.

The I/O manager only initializes the first stack location before sending the IRP over to the driver at the top of the device stack. Each driver, except the lowest in the stack, is then responsible for initializing the stack location for the next driver. This initialization can be done in two ways [10]:

– if no changes to the stack location are needed, the driver can "skip" the current stack location. This means that the next driver will receive the same parameters in its stack location as the current driver.

– if changes to the stack location need to be made, the driver can copy its stack location and modify the copy. Registering a completion routine (explained momentarily) counts as a change, and a driver must copy the stack location instead of skipping it in that case [12].

[22] This value is known when allocating the IRP, because it is equal to the size of the device stack. In some scenarios, most of which involve the filter manager, an IRP might be sent over to a new device stack. This might necessitate readjusting the count of stack locations, if the new device stack has a different size than the one before [10]. We will discuss the filter manager in Section 5.1.

If a driver can complete an IRP and no other drivers need to be called, initializing the next stack location is of course not necessary.

The already mentioned completion routines are a way for drivers to intercept the flow of an IRP after its completion by another driver. Completion routines are called in the opposite order that they were registered in, i.e. those registered by lower drivers first. See Figure 2.16 for an example. In such a routine, a driver can check the status of the completed IRP and do necessary post-processing. It is even possibly to mark the IRP as uncompleted again and resend it down the stack [10]. Completion routines may run at an IRQL equal to `DISPATCH_LEVEL` [12].

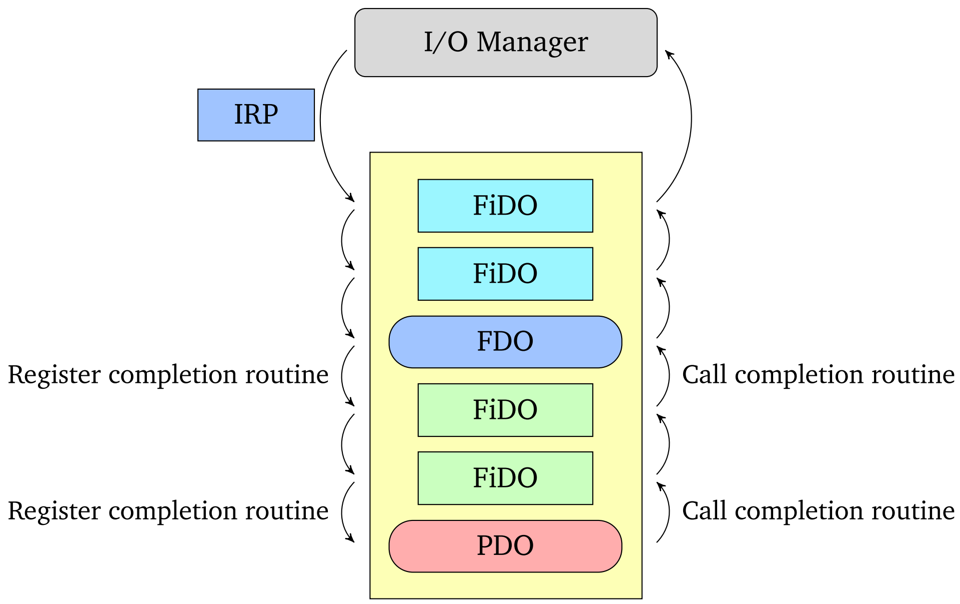

Figure 2.16: Example IRP flow (modified after [10]). The I/O manager sends an IRP down the device stack of a devnode. It first passes through two upper filter drivers and the function driver, with the latter registering a completion routine. It then passes through two lower filter drivers, with the second of those also registering a completion routine. After the bus driver completes the IRP, it travels back up the device stack. Because only the second lower filter driver and the function driver registered a completion routine, they are the only drivers that the IRP passes through on its way in this direction. The real path of the IRP is slightly more complex than shown in this figure, because normally the I/O manager takes care of routing the IRP from one driver to another [10]. It is also responsible for calling the completion routines [12].

2.2.4 Debugging Kernel Drivers

Debugging a program in kernel mode is not as simple as debugging most user mode applications, because it is part of the OS. For example, stopping the execution, e.g. via break points, halts the whole computer. This section will introduce the tools necessary to debug a Windows kernel component.

There are two official programs for kernel debugging, kd and WinDbg. They are essentially equivalent, with the former being a console debugger and the latter a GUI debugger. Out of personal preference, we will use WinDbg in this thesis. [10] also uses WinDbg for exploring the internals of Windows, including a lot of screenshots and explaining many useful commands.[23] See Figure 2.17 and Figure 5.4 for screenshots from WinDbg.

otherapproaches.veracrypt.devstack.png but for a volume with luks2flt attached?

Figure 2.17: WinDbg in remote kernel debugging mode. todo

There are different ways to use WinDbg [10]:

- as a debugger for user mode programs. It can be used for viewing and changing the contents of the program's memory, setting breakpoints, and other common debugging techniques. WinDbg can also attach to an already running process.
- to view a crash dump file that Windows created during a system crash. This contains parts of the RAM contents at the time of a crash.[24] Using this, one can find out more about what exactly caused the crash. The available commands are a subset of what would have been available if WinDbg had been debugging the operating system when the crash happened. For example, the call stack of the function that caused the crash can be displayed. Everything that involves running code or reading and writing from or to certain locations, e.g. I/O ports or some CPU registers, is not possible in this mode.
- as a local kernel mode debugger. If booted in debug mode, WinDbg can attach to a currently running instance of Windows. Only a subset of the normally available commands can be used, and some advanced debugging such as setting breakpoints and dumping memory do not work. An alternative for local kernel debugging is the Sysinternals LiveKd program. This simulates a crash dump file using the contents of physical memory, and in contrast to WinDbg, it can dump memory.
- as a remote kernel mode debugger. This is the mode of operation with the most possibilities, but also the one that is the most complicated and expensive to set up. It requires two computers, a target computer, running the OS to be debugged, and a host computer, that uses WinDbg to debug the target. Because WinDbg does not run under the OS that is being debugged, the target computer can be completely halted and the current system state can be examined. The connection between the two computers can be via USB or the local network.[25]

[23] A list of common commands can also be found at `http://windbg.info/doc/1-common-cmds.html`.

[24] If enabled, the dump file is usually located at `C:\Windows\MEMORY.DMP` It can also be configured which parts of the RAM shall be included in the file, e.g. all used physical memory or only all used kernel memory.

[25] Debugging via network can also be used to debug a virtual machine, if the hardware for a second real computer is not available. See the Windows Debugging Tools documentation for more.

3 Related Work

To our knowledge, we are the first to implement parts of the LUKS2 specification for the Windows operating system. However, there are of course other file encryption systems and implementations of kernel drivers for similar purposes. In this chapter we present a selection of these, and additionally list related work in the fields of driver testing and file system benchmarking. In addition to the following paragraphs, Chapter 4 describes three other implementations of encryption systems for partitions, including the reference implementation of LUKS2.

[16] partially describes the inner workings of the Linux kernel framework that LUKS2's reference implementation uses. They implemented a new feature to configure whether read and write requests should be queued in the kernel. Turning the queueing off resulted in significantly increased throughput for certain hardware configurations and use cases.

An implementation of a kernel driver for purposes that are similar to ours, but for Linux instead of Windows, is presented in [17]. This driver adds support for a plausibly deniable file system to Linux, i.e. the data written to the disk by the driver is "indistinguishable from other pseudorandom blocks on the disk". Like the LUKS2 reference implementation, the driver makes use of the Linux kernel device mapper infrastructure, which is described in Section 4.1.1.

A comparison and performance study of various file encryption systems can be found in [18]. The result is that the performance of encrypted hard disks is good enough for practical usage, in part thanks to good usage of caches. They also identify problems in hard disk encryption, including a lack of common standards. LUKS2, the standard that our driver implements, tries to solve that particular problem. Another problem is the lack of ensuring the integrity of the encrypted data. This is experimentally supported by LUKS2 and its reference implementation (see the "Authenticated data encryption" section of [19]), but not implemented in our driver.

[20] also compares different encryption systems, but only ones for full disk encryption. The comparison is mainly focused on the deployment process and user experience. After laying out the reasons for choosing Windows' built-in BitLocker (described in Section 4.3), the authors describe the results of its deployment and the lessons learned during the process.

[21] features a description of the Windows Driver Model, and also compares it to the kernel driver architecture of Linux. The comparison is accompanied by the implementation of a simple kernel driver with equivalent functionality for both operating systems. The example driver performs I/O from user to kernel buffers, showcasing important concepts for driver development.

The implementation of a Windows kernel driver is described in [22]. This driver is used as the logging component of an intrusion detection system. The events that are logged are gathered by the driver through intercepted syscalls. Similarly to our driver, this driver can be configured by a user space application that sends its messages via device I/O control requests.

As an addition to the Driver Verifier tool provided by Microsoft, other solutions for testing implementations of Windows kernel drivers have been developed. [23] implements a static analysis tool, that, in contrast to the Driver Verifier, does not require executing the driver for testing. Instead, it applies a set of rules to prove correct usage of the API exposed by Windows to device drivers. Rules for both the WDM and the KMDF frameworks are included, and the authors report a high rate of correctly identified errors in drivers (up to 80%). Another, more specialized tool, is the one presented in [24]. It can be used for testing the security of a driver's dispatch routine for device I/O control requests, achieving a high coverage of all possible code paths. Usage of this tool has already lead to finding several previously unknown vulnerabilities.

[1] contains a comparison and analysis of different benchmarks of file systems and storage. They categorize the benchmarks, and describe the surveyed types of benchmarks.

Finally, to thoroughly evaluate the performance of an encrypted storage system, one needs to test it under conditions that resemble the real world as good as possible. [25], [26], [27], [28], and [29] present, describe, and analyse different disk drive workloads collected on real hardware.

4 Other Approaches

In this chapter, we describe both the Linux reference implementation of LUKS2 and the two disk encryption technologies that are featured in the performance comparison in Chapter 6.

4.1 Linux Kernel Implementation of LUKS2

To compare the architecture of our driver (see Section 5.2) to that of the Linux reference implementation, the latter will be described in this section. To see how the user space part of this implementation works, we will take a look at the source code of the `cryptsetup` tool. The work of looking through the Linux kernel source code and creating a high-level overview has already been done in [16].

4.1.1 The Linux Kernel Device Mapper

The reference implementation uses the Linux kernel's device mapper [9]. This is a kernel driver that allows creating virtual devices in layers. One such layer is known as a *target*. A layer can be thought of as the Linux equivalent of a Windows file system filter driver.[1] The device mapper comes with many different available targets, including [30]:

– `zero` This target always returns zeroes for all reads and silently discards all writes. It is similar to Linux' `/dev/zero`, but in contrast to that this is a block device, not a character device.[2]

– `linear` This target maps a range of blocks of the virtual device to an existing device. Described in more detail: map the block range N to $N + L$ of the virtual device to the block range M to $M + L$ of device X.

– `snapshot` This target creates a copy-on-write snapshot of a device. These snapshots are "mountable, saved states of the block device which are also writable without interfering with the original content." [30]

– `crypt` This target performs transparent encryption of a device (writes are encrypted, reads are decrypted). This is what the LUKS2 reference implementation uses. The encryption is done via the Linux kernel's crypto API, which offers a variety of different encryption schemes. This interface is described in more detail in [30].

[1] This comparison holds as far as that the functionality of target is implemented by a kernel driver. It is also possible to implement new targets, as done e.g. in [17].

[2] The difference is that character devices read and write a stream of individual byes, and block devices read and write blocks (usually 512 bytes) [31].

The configuration of a virtual device happens via a *mapping table*. The general format is described in Figure 4.1, and Figure 4.2 describes the parameter format for the `crypt` target.

Outside of the mapping table, the device mapper's targets are usually written with the `dm-` prefix, e.g. we write `dm-crypt` for the `crypt` target.

```
<start block> <size> <target name> <target parameters>
```

Figure 4.1: Linux device mapper mapping table entry format (modified after [9]). The `start block` is the first block of the virtual device that this mapping applies to. `size` indicates the number of blocks, including the start block, that this mapping is for. `target name` specifies the device mapper target that will be used. The `target parameters` are specific to each of the different targets. The `crypt` target's parameters are described in Figure 4.2.
The full mapping for a virtual device can contain multiple entries, each in its own line. It is possible to combine different targets. The following example shows a 1024 block virtual device, whose first half is mapped using the `linear` target, and whose second half is mapped using the `crypt` target:

```
0   512 linear <linear parameters>
512 512 crypt  <crypt parameters>
```

```
<cipher> <key> <iv offset> <device path> <offset> [<#opt> <opt> <...>]
```

Figure 4.2: `dm-crypt` target parameter format (modified after [9]). `cipher` specifies the encryption cipher that is used, in the following format: `cipher-chainmode-ivmode[:ivopts]`. See Section 2.1.4 for more on ciphers, chain modes (also known as block cipher modes), and IVs. Table 2.1 also lists examples of available encryption algorithms. `key` is the key used for the specified cipher. It can either be in hexadecimal notation, or a reference to a key in the kernel keyring (more on this shortly). The `iv offset` is a constant offset that is added to the sector number to create the IV. The `device path` determines the device the encrypted data is stored on, either by giving its path (e.g. `/dev/sda1`) or its major and minor number (e.g. `8:16`, see [31] for more on these). The `offset` gives the first sector on the specified device containing the encrypted data. Additionally, optional parameters can be specified after these required parameters, e.g. the disk's sector size, preceded by the count of optional parameters.

Figure 4.2 mentioned the *kernel keyring* (also known as the kernel key retention service). It is provided by the Linux kernel to cache keys and other secrets for usage by other kernel components. Keys have different attributes, including a key type, a name (also called a description), and a unique serial number. Depending on the key type and a user's permissions, keys can also be accessed from user space using that serial. The serial can be obtained by searching for a key using its type and name. Kernel services can register new key types, supplementary to some already defined types. These include [30][32]:

– `keyring` A key of this type contains a list of links to other keys, including other keyrings.

– `user` A key of this type contains a payload of arbitrary binary data (up to 32767 bytes) and can be accessed and modified from user space. Because of this, keys of this type are not intended to be used by the kernel.

– `logon` This is similar to the `user` type, but keys of this type can only be read by the kernel. It is however possible to create or update them from user space. A `logon` key's description must start with a prefix followed by a colon, with the prefix denoting which service the key belongs to.

The format of the `key` parameter from Figure 4.2 when using the kernel keyring is: `:<size>:<type>:<description>`. `size` is the key size in bytes, and `type` and `description` are the key's type and description, respectively [9]. See Section 4.1.2 for an example value of this parameter, created by the `cryptsetup` tool.

As already mentioned, `dm-crypt` uses the Linux kernel's crypto API for the de-/encryption of reads and writes. Figure 4.3 shows the flow of I/O operations through different parts of `dm-crypt` and the crypto API. [16] experimented with skipping some of the queues shown in the figure, and found significantly increased throughput for some hardware configurations and use cases. They therefore implemented the possibility to configure `dm-crypt` to queue less operations. This is done via optional parameters (see Figure 4.2), and can be used since Linux 5.9 [9].

4.1.2 The `cryptsetup` Command Line Utility

Manually configuring the device mapper to work with encrypted partitions is impractical. Therefore, `dm-crypt` is accompanied by the `cryptsetup` command line utility. Its purpose is, given an encrypted volume, to automatically generate the appropriate `dm-crypt` configuration and send it to the kernel. It supports multiple different encryption technologies, including [19]:

– LUKS and LUKS2 (see Section 2.1);

– TrueCrypt and VeraCrypt (see Section 4.2);

– BitLocker (see Section 4.3). The support for it is relatively new and still in experimental status.

For LUKS and LUKS2, there is additional functionality available, including formatting new partitions, upgrading a LUKS partition to LUKS2, and adding new keyslots (which have been described in Section 2.1) [19].

To illustrate the usual workflow and explain some details "hands-on", we will use the following example: suppose we want to use the LUKS2-encrypted partition located at `/dev/sda1`. We will describe how to make the decrypted partition available, how to mount the contained file system, and how to unmount it and relock the partition again. For the first step, we will also take a look at parts of `cryptsetup`'s source code. This allows us to see what exactly is happening behind the scenes. We assume that no errors occur and that all supplied parameters, including the entered password, are correct. For the steps involving `cryptsetup`, please refer to [19] for more detailed information. See Appendix A for information on which version of `cryptsetup` the following paragraphs refer to.

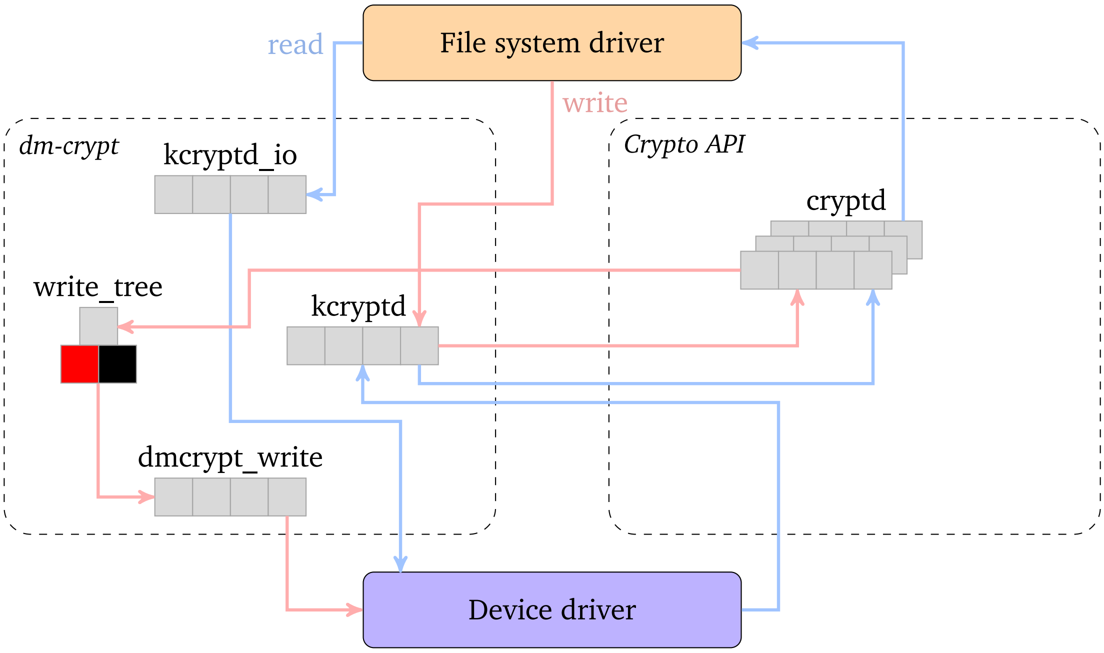

Figure 4.3: `dm-crypt` and Linux kernel crypto IO traverse path (modified after [16]). A write request from the file system driver first moves into the `kcryptd` workqueue. This queue takes care of doing the encryption at some later, more convenient point in time (see [30] for more details). The Linux kernel crypto API does the actual encryption work, and it also may use internal queues. The encrypted write requests are then sorted by `dm-crypt` using a red-black tree and are sent to the device driver via another workqueue.
For read requests, the encrypted data is retrieved from the device driver using the `kcryptd_io` workqueue. When this data arrives, it is scheduled for decryption in the already mentioned `kcryptd` queue. When the kernel crypto API has done its work, the decrypted data is ready for the file system driver.

Making the decrypted partition available From the user's perspective, this is not complex: we run `cryptsetup open /dev/sda1 crypto`. This instructs `cryptsetup` to decrypt the partition and make it available under the name `crypto` (we will see in a moment what role exactly this name plays). During the process, the user has to enter the password for one of the available keyslots. If the entered password matches a keyslot, the decrypted partition will be available at `/dev/mapper/crypto`. The last part of the path is specified by the name that we supplied when invoking the command.

Under the hood however, a lot of work gets done. After command line argument parsing, reading and verifying the on-disk header, and reading the password from the user, these are important parts of the source code:

1. `luks2_keyslot_get_key()` in `lib/luks2/luks2_keyslot_luks2.c`: this hashes the given password (l. 353), decrypts the keyslot area using the hash (l. 365), and merges the decrypted area (l. 370). The merged decrypted area is the derived master key.

2. `_open_and_verify()` in `lib/luks2/luks2_keyslot.c`: this verifies the derived master key (l. 340).

3. `get_key_description_by_digest()` in `lib/luks2/luks_digest.c`: this creates

the description for loading the key into the keyring. The description is always of the format cryptsetup:<uuid>-<digest>, where uuid is the UUID of the encrypted partition, and digest is the index of the digest (in the JSON array of digests) (l. 411).

4. _open_and_activate() in lib/setup.c: this adds the master key to the kernel keyring, if instructed to do so[3] (l. 4003). Note that the type of the key is always logon, as can be seen in crypt_volume_key_load_in_keyring() (l. 6081).

5. get_dm_crypt_params() in lib/libdevmapper.c: this creates the dm-crypt parameter string as described in Figure 4.2 (l. 684). The key is either formatted as a keyring reference (l. 671) or hexadecimally (l. 675).

6. _dm_create_device() in lib/libdevmapper.c: this sends the parameters to the device mapper (l. 1390).

7. LUKS2_activate() in lib/luks2/luks2_json_metadata.c: this frees the memory containing the parameters sent to the device mapper (l. 2221), including the master key. The key gets overwritten with zeroes before being freed, with special care taken to avoid the compiler to optimize the zeroing out (see crypt_safe_memzero() in lib/utils_safe_memory.c).

Mounting the contained file system This works like mounting any non-encrypted partition. The idea of dm-crypt is that the encryption is transparent, i.e. the newly created virtual device behaves just like a regular, non-encrypted partition. Mounting can therefore be achieved via mount /dev/mapper/crypto <mount point>.

Unmounting and relocking Unmounting, too, works as usual: umount <mount point>. Relocking the partition, i.e. removing the /dev/mapper/crypto mapping, can be achieved via cryptsetup close crypto.

Example of generated parameters Finally, we present an example of complete dm-crypt parameters, as generated by cryptsetup. These are the parameters for a device encrypted with AES-XTS with two 256 bit keys:[4] aes-xts-plain64 <key> 0 /dev/sda1 32768. The key parameter format depends on whether the kernel keyring is used or not:

– :64:logon:cryptsetup:59691c6c-a764-4c59-a876-cbc0c55802d5-d0 is a possible reference to the key in the keyring;

– 91bc1104b514053ae2420a09d0ba331fe8fa13acad6f9145697a5d28f2b7cb4a... is the hexadecimal representation of the key (only the first 64 of the 128 characters are shown).

[3] This is the default behaviour (see lib/libcryptsetup.h, l. 733) and can be disabled with the --disable-keyring command line option [19].

[4] Recall from Section 2.1.4 that the XTS mode needs two keys: a data encryption key and a tweak key.

4.2 VeraCrypt

VeraCrypt is an open source disk encryption program that is based on the discontinued TrueCrypt project. There have been several security assessments of both projects, see e.g. [33] and [34].

Because VeraCrypt is an open source project, one can always just take a look at the code to see how everything works. Pleasantly, a lot of VeraCrypt is also described in its documentation[5] [35]. In this chapter, we mostly rely on the documentation, but we also peek into the source code to gain a deeper understanding of VeraCrypt's inner workings.

4.2.1 Relevant Technical and Cryptographic Details

VeraCrypt stores all metadata in an encrypted header on disk. Because of the encryption, the header and the encrypted user data is indistinguishable from random bytes. This means a third party cannot tell a VeraCrypt volume apart from one that only contains random data.[6] The key to decrypt the header is derived from a password entered by the user. The master key, used to decrypt the user data stored on disk, is part of the header. This key is generated at volume creation and cannot be changed. It is however possible to change the password of the volume, achieved by decrypting the header and re-encrypting it using the key derived from the new password [35].

VeraCrypt supports different block ciphers, including AES, Serpent, and Twofish (see [6] for more on them). It is also possible to cascade ciphers by first encrypting data using one cipher and then encrypting the ciphertext again, using another cipher. For example, VeraCrypt supports AES-Twofish encryption, which means that data is first encrypted with Twofish and then with AES. As for block cipher modes, only the XTS mode is supported [35].

Unlocking a volume always requires a password from the user. The user can also specify the parameters needed for the decryption of the header, including parameters for key derivation and the encryption scheme. If they are not specified, VeraCrypt tries all possible combinations of parameters. The first four bytes of a decrypted header are always "VERA" (in ASCII). The plaintext header also contains a checksum of its last 256 bytes. If both the first four bytes and the checksum match their respective expected value, decryption is considered successful [35].

Using an unlocked partition is mostly equivalent to what is described in Section 2.1.4.

4.2.2 Peeking Into VeraCrypt's Source Code

In this chapter we will take a look at the source code of the kernel driver that implements most of VeraCrypt's functionality. Our claims about these inner workings are supported by references to the source code. All file paths that do not start with `src/` are relative to the `src/Driver` subdirectory of the VeraCrypt source code. See Appendix A for notes regarding the version of the inspected source code.

It should be noted that VeraCrypt's source code includes computer that we will not discuss, namely a "DeviceFilter" and "VolumeFilter". The reason for this is that these components are not used when a VeraCrypt volume is unlocked and mounted in portable

[5] In fact, the information provided in the documentation should suffice for most regular users. However, to understand how VeraCrypt works internally, one still needs to read the source code.

[6] This property is called plausible deniability. [35] mentions a volume that has been securely wiped as a plausible example for why a volume would only contain random data.

(i.e. non-install) mode,[7] which is how we conducted our experiments with VeraCrypt. We will therefore focus on the components that are used in this scenario.

The basic architecture of VeraCrypt is shown in Figure 4.4. When started, the user space executable installs a kernel driver using the `CreateService` Win32 API. VeraCrypt can operate in portable mode, which has the effect that the driver installation is only temporary.[8] This driver creates the so-called root device,[9] `\Device\VeraCrypt`, with which the user space program can communicate via IOCTLs.

To unlock and mount a VeraCrypt volume, the user space application sends the `TC_IOCTL_MOUNT_VOLUME` device control code to the root device. Its driver then creates a new device,[10] which will be used to mount the volume. All work of unlocking the volume and processing IRPs for the newly created device is done in its own thread.[11]

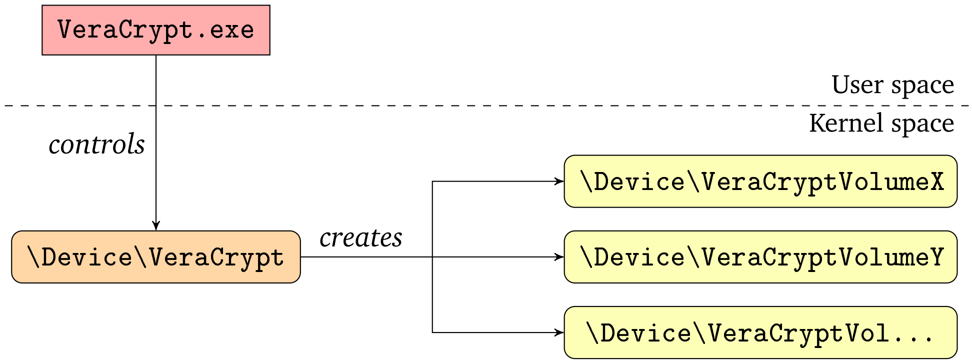

Figure 4.4: VeraCrypt Architecture. The user space application sends commands to the root device, `\Device\VeraCrypt`, via custom IOCTLs. One possible command is `TC_IOCTL_MOUNT_VOLUME`, which instructs the root device to create a new device and mount a VeraCrypt volume using that device.

The device for an unlocked volume holds a file handle to the device object for the encrypted volume (e.g. `\Device\HarddiskVolume6`). This is used for reading from and writing to the volume.[12]

Similar to `dm-crypt`, the VeraCrypt driver uses queues to manage incoming and pending IRPs. All I/O operations for a volume pass through up to three queues, each serviced by its own thread:[13]

– **Main queue** All Read and Write IRPs are inserted into this “encrypted I/O queue”. The thread responsible for this queue checks and adjusts the request parameters, if needed, and encrypts the data for write requests. Requests that were not rejected because of invalid parameters or other errors are fragemented into blocks of up to 256 KiB, and these fragments are inserted into the I/O queue. This is done “to enable efficient overlapping of encryption and IO operations.”[14]

[7] At least not in the setup that we used, which were the default settings except that the volume was mounted with read-only access.
[8] See l. 4614f., 4355f., and 4455f. in `src/Common/Dlgcode.c`.
[9] See l. 375 in `Ntdriver.c`.
[10] See l. 2622f. and 4076 in `Ntdriver.c`.
[11] See l. 4123 and 3074f. in `Ntdriver.c`.
[12] See l. 280f. in `Ntvol.c` and l. 457, 520, and 345 in `EncryptedIoQueue.c`.
[13] See l. 970f. in `EncryptedIoQueue.c`.
[14] See l. 639 in `Ntdriver.c`, l. 886, 673f., 771f., 822f., and 844 in `EncryptedIoQueue.c`, and l. 23 in `EncryptedIoQueue.h`.

- **I/O queue** This queue is responsible for initiating the I/O for incoming requests, and adding successful read requests to the completion queue. If the last read succeeded and there are no outstanding I/O requests, it also performs another read operation. This reads the same amount of bytes as the last operation and stores it in a read ahead buffer. For each read request, VeraCrypt checks whether it can fulfil it using the read ahead buffer, before falling back to reading from the encrypted volume.[15]

- **Completion queue** This is where the decryption of read requests happens.[16]

When no longer needed, sensitive data in memory, such as a volume's password, is overwritten with zeros. This is done using the `RtlSecureZeroMemory` function,[17] which is guaranteed not to be optimized out by the compiler [12].

4.3 BitLocker

Since Windows Vista, the professional versions of Windows (i.e. Professional and Enterprise) already ship with a proprietary volume encryption functionality, called BitLocker. This chapter describes the basic principles of BitLocker. It will not be as detailed as Section 2.1. If more details are of interest, [36] is a quite comprehensive reference. Note however that since its publication, some changes have been made to BitLocker: Windows 10 build 1511 introduced a new mode that uses the XTS block cipher mode, and the optional Elephant Diffuser was removed in Windows 8 [37]. Because the basics did not change, we still mostly rely on the findings presented in [36].

There are also other works that describe BitLocker and discuss relevant security aspects, e.g. [38] and [39].

We will also look at some implementation details by decompiling the source of the kernel driver that implements BitLocker's functionality.

4.3.1 Relevant Technical and Cryptographic Details

BitLocker has three kinds of keys that are needed to unlock and use an encrypted volume [36]:

- **FVEK** the *Full Volume Encryption Key (FVEK)* is used to encrypt the data stored on the volume (this is what LUKS2 calls the master key).

- **VMK** the *Volume Master Key (VMK)* is used to encrypt the FVEK so it can be safely stored in the volume's metadata.

- **External** this is a key used to encrypt the VMK so it, too, can be safely stored in the volume's metadata. An external key can be stored on an external USB device, released by a *Trusted Platform Module (TPM)*,[18] or entered by the user in the form of a password. It is also possible to store an external key directly in the volume's

[15] See l. 456f., 490f., 502f., and 329f. in `EncryptedIoQueue.c`.
[16] See l. 292f. in `EncryptedIoQueue.c`.
[17] See l. 400 in `src/Common/Tcdefs.h` and l. 2647f. in `Ntdriver.c`.
[18] See e.g. their specification [40] for more information on TPMs. They are not really relevant for this chapter.

metadata, effectively disabling BitLocker, but without storing the data in plaintext form.[19]

Similarly to LUKS2, BitLocker stores multiple encrypted versions of the VMK on disk, each using a different external key for the encryption. This means that the VMK can be decrypted using any one of the existing external keys. An example where this is useful is when the TPM detects a system modification and does not release the key: the plaintext VMK can then be obtained using a recovery password, which is always created when formatting the volume. Because the VMK can be used to decrypt the FVEK, which in turn decrypts the data stored on disk, one of the available external keys is all that is needed for unlocking a volume [36].

Also similarly to LUKS2, to verify that both the VMK and the FVEK were decrypted successfully, a hash of each decrypted key is stored together with its ciphertext. Additionally, the metadata for an external key also contains the GUID of the BitLocker volume whose VMK this key can decrypt [36].

The FVEK is always 512 bits long. BitLocker supports 128 and 256 bit keys for data encryption, and depending on which is used, different parts of the FVEK are utilized [36]:

- when using 128 bit keys, the first 128 bits of the FVEK are used for data encryption, and the third group of 128 bits, i.e. bits 256 to 383, are used as the sector key (explained momentarily). The other parts of the FVEK remain unused.
- when using 256 bit keys, the first 256 bits of the FVEK are used for data encryption, and the remaining 256 bits are used as the sector key.

The mentioned *sector key* is XORed with data to be written to disk before the encryption [36]. The algorithms available for encryption are AES-CBC and AES-XTS (the latter since Windows 10, build 1511) [37].

Refer to Section 2.1.4 for the usage of an unlocked partition. Most of what is described there also applies to BitLocker.

4.3.2 Decompiling Parts of BitLocker's Kernel Driver

In this section we will take a look at parts of `fvevol.sys`, which is the "BitLocker Drive Encryption Driver" (see Figure 4.5). `fvevol.sys` has already been mentioned in Figure 2.10 and Figure 2.12. From there it is apparent that it is a lower filter driver that gets loaded for all volumes.

All disassembled code shown in this section is also reproduced in Appendix B.

Figure 4.6 shows that the BitLocker driver filters only IRPs with certain major functions, skipping all other requests.

From Figure 4.7 (and looking at more disassembled code, which is omitted here) it is clear that the BitLocker driver does not implement cryptographic functionality itself. Instead, it relies on the interface of another kernel driver.[20]

According to [36], "Reverse engineering parts of the BitLocker system indicate that the system zeros out sensitive data as soon as they are no longer needed, consistent with good security practices." This is consistent with our findings presented in Figure 4.8.

[19] [36] has a nice metaphor for this: "Like leaving a house key under the doormat, the volume is still protected, but by knowing where to look it's trivial to bypass the protection."

[20] The documentation for the Windows Vista version of this interface can be found at `https://csrc.nist.gov/csrc/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp891.pdf`.

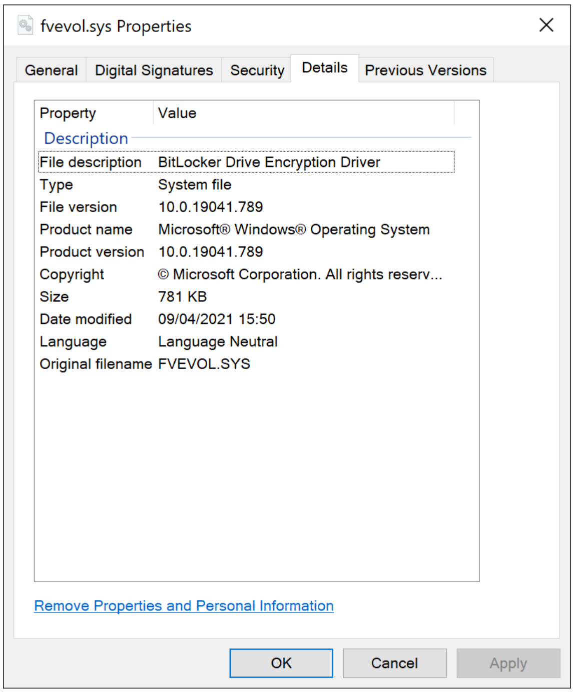

Figure 4.5: Properties of `fvevol.sys` (screenshot from Windows Explorer). This file is usually located at `C:\Windows\System32\drivers\fvevol.sys`.

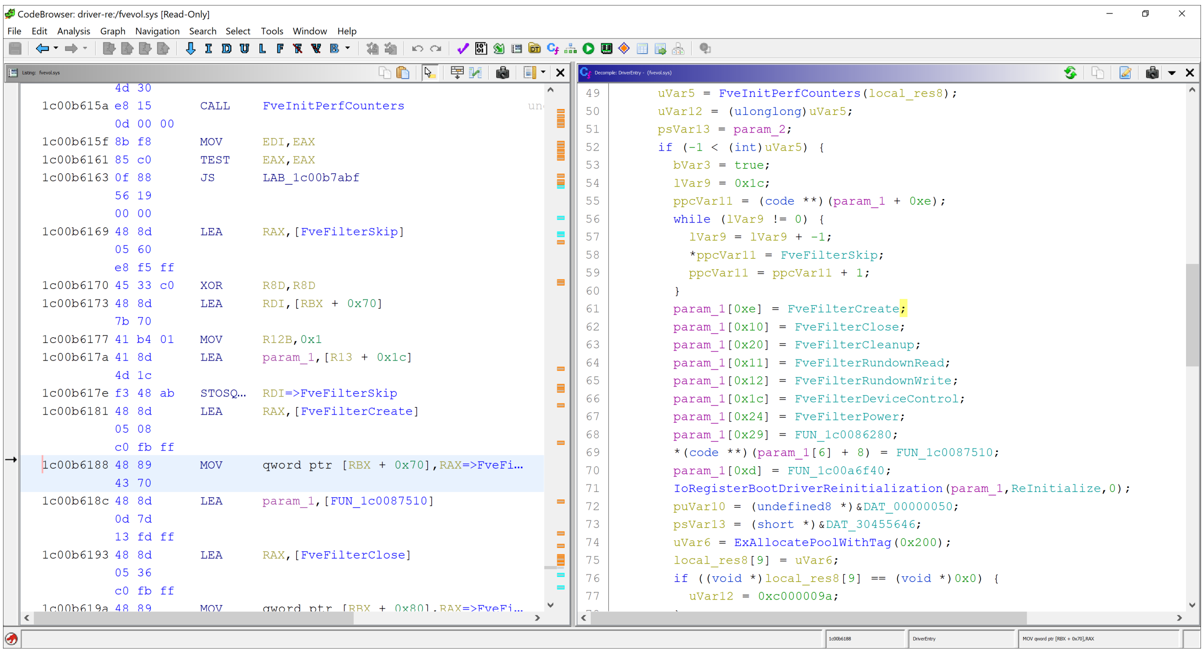

Figure 4.6: Assembly and disassembly of parts of the `DriverEntry` function (screenshot from Ghidra). Shown here is the definition of which function of the driver will be called when an IRP passes through it. These are called dispatch routines. The driver has to specify the different dispatch routine for each IRP major function. See Section 5.2.4 for more details on this initialization process.
`fvevol.sys` only defines specific dispatch routines for the Create, Close, Cleanup, Read, Write, DeviceControl, Power, and PnP major functions. All others use a generic routine, which usually just forwards the IRP to the next driver in the stack (except for some error cases, where it fails the request).

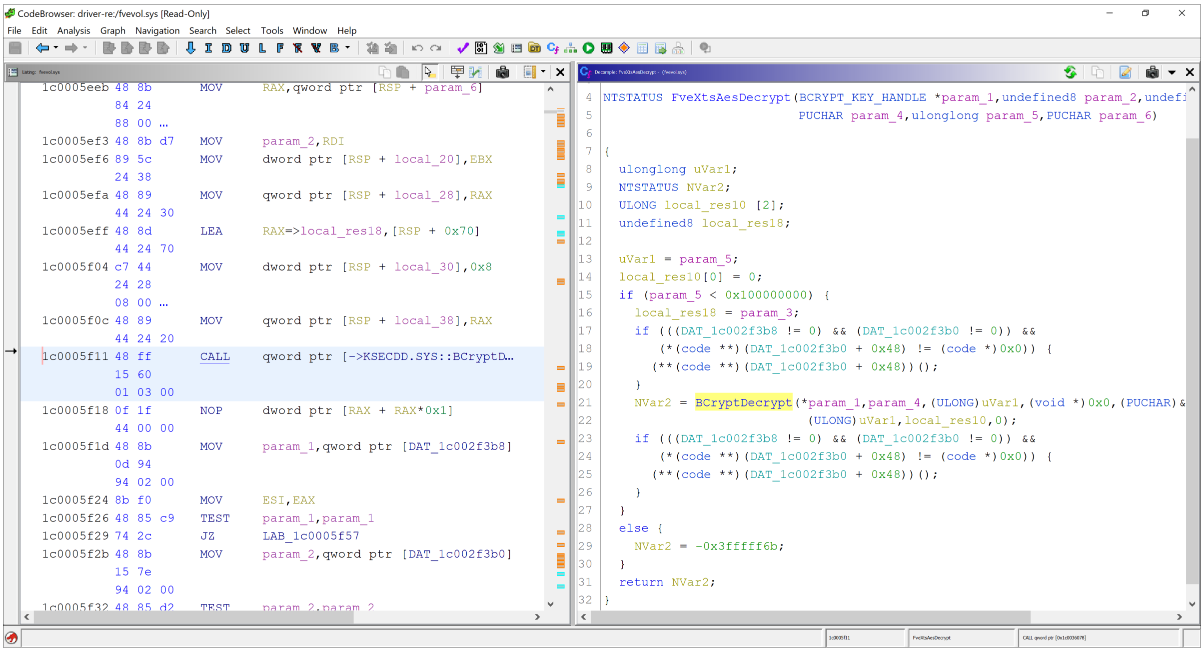

Figure 4.7: Assembly and disassembly of the `FveAesXtsDecrypt` function (screenshot from Ghidra). `fvevol.sys` uses functions from `ksecdd.sys`, which is the “Kernel Mode Security Support Provider Interface”, for all cryptographic functionality. Shown here is the implementation of AES-XTS decryption, which also makes use of this interface.

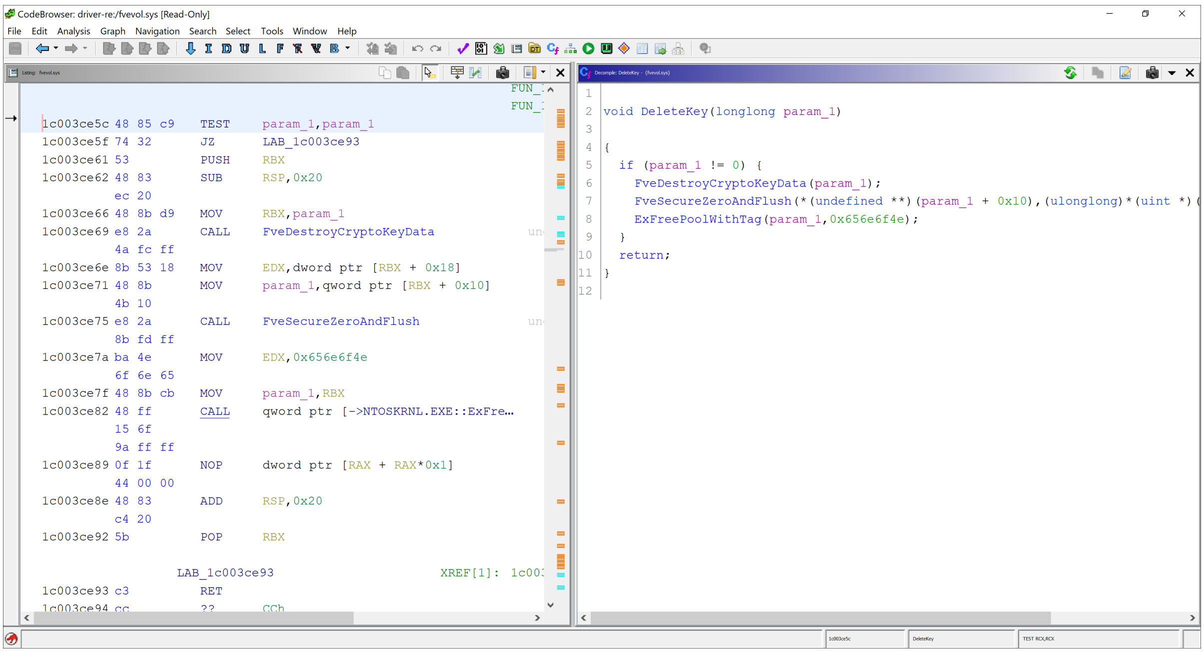

Figure 4.8: Assembly and disassembly of the `DeleteKey` function (screenshot from Ghidra). The `FveDestroyCryptoKeyData` function notifies the `ksecdd.sys` driver that the key is no longer needed. `FveSecureZeroAndFlush` zeros out the key data and also calls `KeSweepLocalCaches`, which is not documented (but probably flushes the key data from the CPU’s caches). `ExFreePoolWithTag` is basically a regular C `free` but for the Windows kernel.

5 Design and Implementation of Our Approach

This chapter lays out the design and architecture of our Windows kernel driver. Where there were multiple ways to do things, we will provide the reasons behind our choice. This includes the choice for WDM as the used framework for the driver.

5.1 Rejected Driver Frameworks

As already mentioned in Section 2.2.2, there are different frameworks to choose from when writing a driver. In this section, we will discuss some other frameworks that we ultimately did not use for our driver, even though we implemented an early prototype using one of them.

5.1.1 The Filter Manager

The filter manager is a kernel driver that ships with Windows, implementing commonly required functionality to simplify the development of third-party file system filter drivers. Filter drivers that make use of the filter manager are called *minifilter drivers*. They can, among other things, filter IRPs by registering a pre- and/or postoperation callback routine. These are the equivalents to WDM's dispatch and completion routines. Just like the I/O manager for WDM drivers, the filter manager is responsible for calling a minifilter's appropriate callback routine when an IRP arrives [41].

Also similar to WDM drivers, the order in which an IRP passes through multiple minifilters is deterministic and configurable. It depends on the minifilters' *altitude*, which is a value between 40,000 and 429,999.[1] For preoperation callbacks, a driver with a higher altitude is called before a driver with a lower altitude, and vice versa for postoperation callbacks. There are pre-defined groups of altitude values, called *load order groups*. Examples are the Anti-Virus group (using an altitude between 320,000 and 329,999) and the Encryption group (140,000 to 149,999). To ensure that all minifilters have an appropriate altitude, Microsoft is responsible for assigning altitude values[2] [41].

During the development process of our driver, we implemented a first prototype which used the filter manager. It worked to some extent, but ultimately it was not fit for its purpose. The architecture of an I/O stack that includes the filter manager shown in Figure 5.1 explains why this is the case: the filter manager and therefore all minifilters always sit above the file system driver. LUKS2 encryption however sits below the file system, meaning that despite our minifilter the file system driver can only read the encrypted data. We did not notice this during development because we used raw I/O to test our driver. This means that we created a handle to a path like

[1] Lower altitudes are also possible, but are reserved by Windows for internal use.

[2] At `https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/allocated-altitudes` one can browse a list of all currently assigned altitudes.

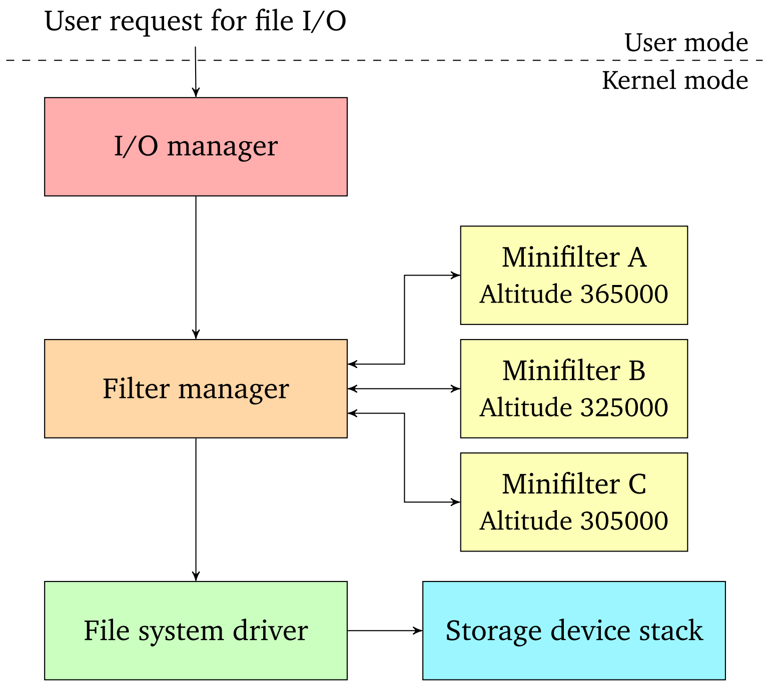

Figure 5.1: Example of a filter manager I/O stack (modified after [41]). A user initiates an I/O request, which the I/O manager receives first and forwards to the file system. The filter manager intercepts the request and lets it pass through the registered minifilters, in order according to their altitude (their altitude places them in the following groups, ordered from A to C: Activity Monitor, Anti-Virus, and Replication). After this, the file system driver processes the request and translates it from file-based to something that a driver in the storage device's stack can understand. Finally, this device stack receives the request from the file system driver.

`\Device\HarddiskVolume6` and performed I/O using that. Interestingly, in this case our minifilter successfully decrypted reads from the volume.

We figured out why the decrypted file system was not recognized by Windows through debugging with WinDbg. Online research yielded the name of the driver responsible for recognizing FAT filesystems (our test LUKS2 volume contained a FAT32 filesystem): `fsrec.sys`. Using information from unofficial online documentation,[3] we set a breakpoint at `fsrec`'s `IsFatVolume` routine. It receives the first sector of a volume and determines whether this matches the format of a FAT boot sector. When hitting the breakpoint and dumping the value of the function's parameter in WinDbg, it became clear that this routine received the encrypted first sector of the LUKS2 volume, consequently (and correctly) classifying it as a non-FAT volume.

See the official documentation [41] for more information on the filter manager and minifilter drivers.

5.1.2 The Windows Driver Frameworks

As already mentioned in Section 2.2.2, another possibility to develop a driver are the Kernel Mode Driver Framework (KMDF) and User Mode Driver Framework (UMDF), together called the Windows Driver Frameworks (WDF). These try to simplify things by

[3] `http://www.codewarrior.cn/ntdoc/win2k/fsrec/index.htm`

abstracting away some of the details that WDM drivers have to care about. The basic principles (e.g. IRP-based I/O) remain the same [10].

We decided against using UMDF out of performance concerns; as the framework's name suggests, UMDF drivers run in user mode. This has certain advantages, such as increased system stability. But most of the I/O ecosystem runs in kernel mode, which means that switches between kernel and user mode are necessary when an I/O driver runs in user mode. These induce latency, together with some additional communication with the kernel [10].

KMDF does not have this problems, but there were other factors that led to our decision against it:

– We had previously tried another framework that simplified things through abstraction (see Section 5.1.1) and it had not worked. Therefore it seemed safer to do all the work ourselves, and in return be sure that the framework will not be a reason for failure.

– By default, KMDF filter drivers pass all received IRPs to the next driver in the stack [14]. However, our planned approach was as follows: don't pass through anything at first, and then gradually during the development process allow more and more requests to pass through. This is still possible with KMDF, but more work than for a WDM driver.

– One of the biggest selling points of using KMDF is that it handles PnP and Power IRPs for the driver. These are not really relevant to our driver, though (see Section 5.2.6 for more information on these requests and how our driver handles them). Therefore we would not gain that much by sacrificing total control for reduced complexity, because the hidden complexity is not that relevant in the first place.

– Finally, writing a WDM driver requires a more complete understanding of how the kernel's IRP-based I/O system works. We wanted to learn as much as possible, and for this purpose WDM seemed the better fit.

Now that our WDM driver is finished, it would be interesting to compare it to an equivalent that was written using KMDF, both its architecture and performance. This is left as an exercise to the curious reader; Section 5.2 should be a good starting point.

[10] contains detailed information on the architecture and inner workings of both KMDF and UMDF. The official documentation [14] is also a valuable source of information.

5.1.3 Other User Space Frameworks

There also exist third-party user mode libraries for developing file system drivers, e.g. `https://github.com/billziss-gh/winfsp` or `https://github.com/dokan-dev/dokany`. However, these probably suffer the same drawbacks as UMDF drivers, and were therefore rejected.

5.2 The Oknolynx Project

After deciding against the frameworks mentioned in the previous section, only WDM was left, which we thus settled for. This section will describe our approach and our

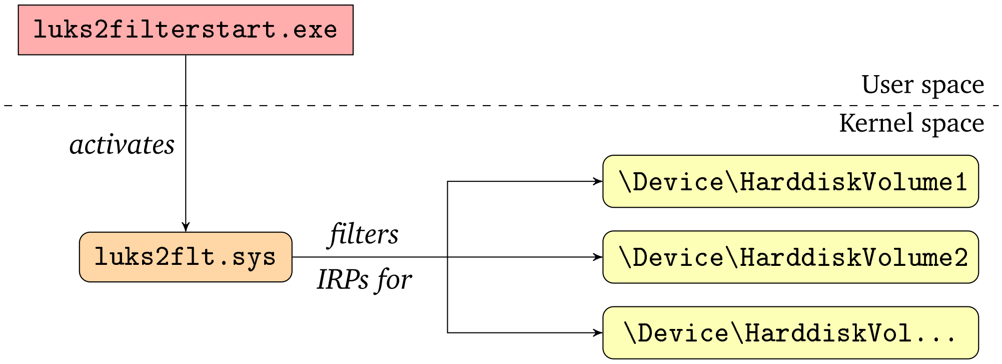

Figure 5.2: Basic communication and architecture of Oknolynx. To activate the filtering, `luks2filterstart.exe` sends a custom IOCTL to a device, in response to which `luks2flt` starts filtering requests.

considerations during its design in detail. Where applicable, both read and write operations will be taken into consideration. See Section 5.2.7 for why we ultimately decided to make all filtered partitions read-only.

As explained in the following section, there are two components that work together to enable LUKS2 support on Windows. Together, the two components form the *Oknolynx* project.[4] Appendix A explains how to obtain a copy of Oknolynx' source code.

5.2.1 Architecture

Oknolynx consists of two components, one in kernel and one in user mode: the `luks2flt` kernel driver, and the `luks2filterstart.exe` program to activate the filter for a volume. Figure 5.2 shows this basic communication and architecture.

`luks2filterstart.exe` is a program written in Rust that reads a password from the user and tries to derive the LUKS2 master key from it. For the key derivation, as described in Section 2.1.3, it uses our helper library (see Section 2.1.1). The decrypted master key, together with other relevant metadata, is then sent to `luks2flt` using a custom IOCTL, `IOCTL_DISK_SET_LUKS2_INFO`. This process is described in more detail in Section 5.2.3.

`luks2flt` is a WDM lower filter driver that attaches to all devices in the Volume class. This is the same mechanism that the BitLocker filter driver uses. It attaches to all volumes at boot time, but only starts filtering after receiving the command to do so from `luks2filterstart.exe` (see Section 5.2.3 for the reasons behind this choice and other details). Compared to VeraCrypt and the Linux kernel implementation of LUKS2, this is a completely different approach, as these both create a new device instead of filtering requests for an existing one.

Another important part of the architecture is where the actual cryptographic work happens. Both LUKS2's reference implementation and BitLocker let the OS's crypto providers do the work, VeraCrypt ships their own implementations. Because the API that BitLocker uses is not publicly documented (and there seems to be no public alternative), we decided against using it. When researching the topic of implementing AES, we found that the general consensus seems to be that it is seldom a good idea to "roll your own crypto." This is why we tried to link `luks2flt` against existing cryptography libraries.

[4] "Okno" means window in Czech, and the German word for lynx is "Luchs". The name therefore symbolizes the combination of Windows and LUKS(2).

To complicate things, these would need to be compiled specifically for being called from Windows kernel mode; the distributed library binaries do not work. We tried getting the following two libraries to work:

– **OpenSSL** This was our first choice because of the project's popularity, but the attempt was abandoned quickly. OpenSSL has a horribly complicated build system[5] and it became clear that compiling this for the Windows kernel was not feasible.

– **libsodium** Chosen because of its promisingly simple build process using Microsoft Visual Studio, which we also used to develop and compile our driver, this library could be compiled in such a way that our driver could use it. However, after the initial setup process we realized that this library does not support the AES-XTS mode.[6] Because this is crucial for our driver's functionality, we could not use this library.

In the end, we decided that for a scientific project it would be okay to implement our own cryptographic functionality. Details on this topic can be found in Section 5.2.5.

5.2.2 Installation

[10] and [12] describe to ways to install a driver: either using *setup information files (INF files)*, or using the `CreateService` Win32 API. The latter probably makes for a better user experience, because the user does not have to manually install the driver using its INF file. It is also possible to install a driver with an INF file without user interaction, but to our knowledge there is no easy-to-use API for it.[7] VeraCrypt, as described in Section 4.2.2, uses the `CreateService` API, and BitLocker already ships with Windows and thus needs no installation. Because we did not want an extra executable for installation during development, we chose using an INF file to install `luks2flt`, but it would not require much work to switch to using the `CreateService` API.

INF files are text files containing the information needed to install a driver. This includes version information, details about the driver's location, load time, and error handling, and updates to registry values that need to be executed at installation. Writing an INF file is not completely trivial, even for a simple use case as a WDM filter driver. [12] includes, in addition to the information that we just presented, more details about the structure of INF files, plus best practices and examples for common use cases. Our recommendation is to also use the `InfVerif` tool[8] to test INF files and verify that they do what they are expected to do.

Figure 5.3 shows the most important parts of the INF file that we wrote for `luks2flt`.

5.2.3 Initialization and Configuration

The device stack for volumes is built at boot time, and it is not possible to scan the data stored on the device for a LUKS2 header at this point.[9] Attaching to the device stack

[5] One of the build steps is to run Perl scripts which emit assembly, which is later assembled and linked together with C code.

[6] In fact, it doesn't even support AES on its own. The only possibility is to use AES with the Galois Counter Mode (GCM). A mode-agnostic implementation of AES would have simplified our work of implementing AES-XTS, because half of the work would have been done already.

[7] One possibility would be to execute `pnputil /add-driver <path to INF> /install`.

[8] `https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/infverif`

[9] We do not have a source for this claim, but we tried to make it possible and did not succeed.

```
1  [DefaultInstall.NTamd64]
2  CopyFiles = Luks2CopyFiles
3  AddReg    = Luks2AddReg
4
5  [DefaultInstall.NTamd64.Services]
6  AddService = %ServiceName%,,Luks2Service
7
8  [Luks2Service]
9  DisplayName    = %ServiceName%
10 Description    = %ServiceDescription%
11 ServiceBinary  = %12%\luks2flt.sys
12 ServiceType    = 1 ; SERVICE_KERNEL_DRIVER
13 StartType      = 0 ; SERVICE_BOOT_START
14 ErrorControl   = 1 ; SERVICE_ERROR_NORMAL
15 LoadOrderGroup = "Filter"
16
17 [Luks2AddReg]
18 HKLM,%VolumeClassPath%,"LowerFilters",0x00010008,"luks2flt"
19
20 [Luks2CopyFiles]
21 luks2flt.sys
```

Figure 5.3: Excerpt from `luks2flt`'s INF file. Values of the form `%val%` get replaced with the value of the variable `val`. These values are either defined in an extra section of the INF file or are predefined. E.g. the `%12%` in the `ServiceBinary` value in line 11 will always expand to `%SystemRoot%\System32\drivers`, where `%SystemRoot%` is the Windows installation directory (in most cases `C:\Windows`).
The sections starting with `DefaultInstall` specify which other sections will be executed. In this case three actions will be performed: copying the driver's `.sys` file, adding a registry entry, and adding a service.
The information in the `Luks2Service` section will end up in the Software registry key for `luks2flt` (see Table 2.2). The orientation for the chosen values were mostly the ones that the BitLocker driver uses. They can be found at `HKLM\SYSTEM\CCS\Services\fvevol`.
The `Luks2AddReg` section specifies that upon installation the string `luks2flt` will be appended to the `LowerFilters` value of the Volume Class key, `System\CCS\Control\Class\{71a27cdd-812a-11d0-bec7-08002be2092f}`. See Figure 2.12 for what the values under the Volume Class key usually look like.

must be done when it is built, which is why `luks2flt` attaches to all devices in the Volume class. However, because it is possible that the device whose stack it attached to is not a LUKS2 volume, the default is to let every incoming request pass through. Filtering only starts after the driver receives the appropriate IOCTL from `luks2filterstart.exe`.

To do its work when actually filtering a volume, `luks2flt` needs some information:

– The volume's sector size, to calculate the sector number given a byte offset into the volume. This is needed because our driver only supports the plain64 IV generator for AES-XTS, which means that the IV for the encryption is the 64-bit little-endian representation of the sector number. Read and Write IRPs only contain the read or write offset, and we therefore need to calculate the sector number ourselves.

- The number of the LUKS2 segment's first sector, relative to the volume's first sector. This value is also part of the JSON section of the volume's LUKS2 header. It is needed to shift the offset of a Read or Write IRP, as explained in Section 5.2.5.

- The key size for the encryption, i.e. whether AES-128-XTS (128-bit or 32-byte key) or AES-256-XTS (256-bit or 64-byte key) is used.

- The master key for decrypting reads and encrypting writes.

This data is sent along with the IOCTL in an input buffer. The first byte of the buffer signals whether filtering should be enabled, and if this byte is non-zero, the rest of the buffer contains the values listed above.

To keep track of these values, `luks2flt` stores them in something called the *device extension*. Every device object contains a pointer to this driver-defined data structure. Its purpose is to store information or objects that all driver routines operating on the device need access to [42].

At this point we want to note the following: it would also be possible to store most of the information sent with `IOCTL_DISK_SET_LUKS2_INFO` in the registry. Storing configuration this way is encouraged by Microsoft: at startup, every driver receives a string containing the path to its Software key [12]. One could even store a list of the system's LUKS2 volumes in the registry. This would eliminate the aforementioned problem that `luks2flt` does not know yet which volumes it should attach to when the device stack is built. However, `luks2filterstart.exe` would still need to send the key to the driver, as this information should not be stored in the registry. The key should always be freshly derived from a password entered by the user.

5.2.4 The `DriverEntry` Routine and IRP Dispatching

Every WDM driver must implement a `DriverEntry` routine. It is called when the driver gets loaded and is responsible for driver initialization [12]. In addition to initializing global driver state, this routine must store pointers to three types of other routines in its driver object [42]:

- **An `AddDevice` routine** This gets called during device enumeration (see Section 2.2.2), when the PnP manager finds a new device whose stack the driver may want to attach to. Its responsibilities are creating a new device object, initializing it, and attaching it to the device stack.

- **An Unload routine** This gets called just before the driver is unloaded. Here, all memory and other resources that were allocated in the `DriverEntry` routine are freed.

- **IRP Dispatch routines** These are responsible for handling IRPs. The dispatch routine is specified separately for each IRP major function.

The specification of the BitLocker driver's dispatch routines can be found in Figure 4.6. Figure 5.4 lists the routines that `luks2flt` registers and Figure 5.5 describes the driver's IRP dispatching technique.

The handling of IRPs with allowed major functions for filtered devices will be described in the following two sections.

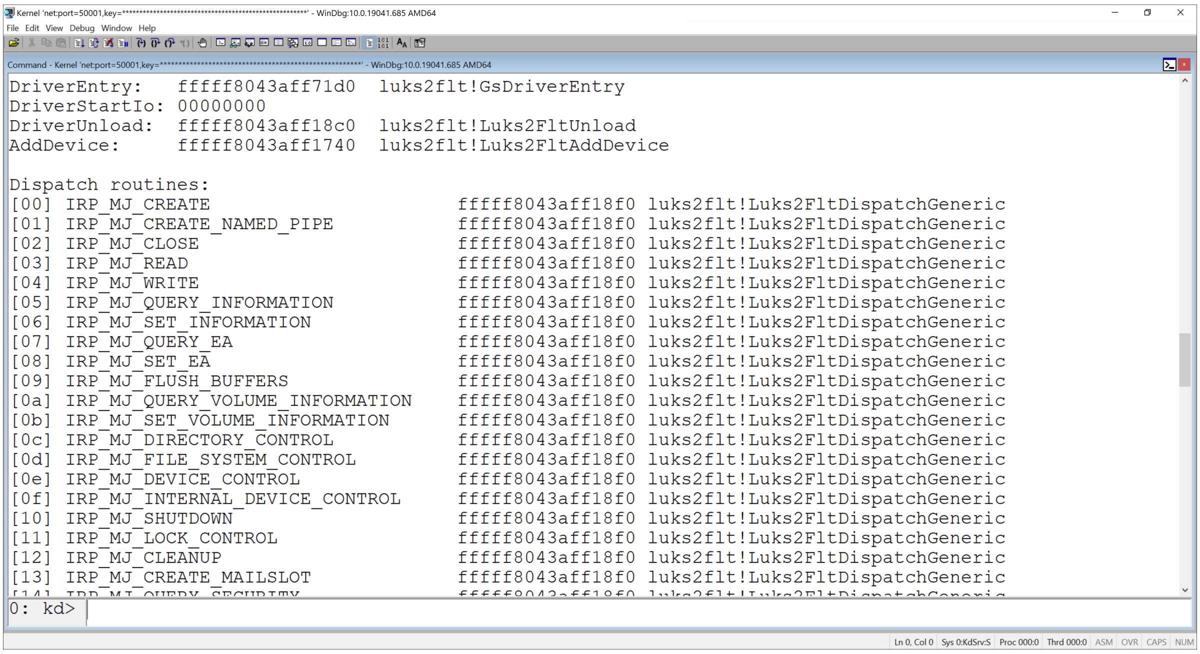

Figure 5.4: `luks2flt`'s registered routines (screenshot from WinDbg). This is part of the output that WinDbg shows for the `!drvobj luks2flt 7` command. The routines that were registered during initialization are shown: its unload, `AddDevice`, and IRP dispatch routines. IRPs of all major functions are handled by a generic dispatch routine, whose logic is described in Figure 5.5.

5.2.5 Handling Read and Write Requests

The basic idea of handling Read and Write IRPs for filtered volumes is, as we will see, almost the same. We will first describe the commonalities, then the specifics for Write requests, and then those for Read requests (which only require a little modification of the Write process).

Both Read and Write IRPs need adjusting to the read/write offset: all drivers above `luks2flt` and all user mode applications use offsets relative to the segment's first sector. The drivers below however use offsets relative to the volume's first sector. Therefore, `luks2flt` needs to translate between those two kinds of offsets. This is as simple as shifting the offset by the position of the encrypted segment on disk.

An example: a program in user mode wants to read the first sector of an unlocked LUKS2 volume. This request reaches `luks2flt` as a Read IRP with an offset of zero bytes. On disk however there are (in this example) 16 mebibytes of LUKS2 metadata before the first sector of encrypted data. The read offset therefore needs to be set to 16,777,216 bytes, so that the lower-level drivers read the correct, wanted data.

For Write requests, only one more task needs to be done: encrypting the received plaintext data before it is written to disk. Recall from Section 2.1.4 that LUKS2 supports many encryption algorithms, but `luks2flt` only supports `aes-xts-plain64`. The encryption is therefore always done using our own implementation of AES-XTS (see the end of this section).

Read requests require one more step, because when the IRP first arrives at our driver, no data has actually been read yet. That is the duty of the drivers below `luks2flt` in the volume's device stack. We therefore need to wait until these have completed all their work for this IRP. Completion routines, which were introduced in Section 2.2.3, were explicitly designed for this purpose. After shifting the read offset as explained above, the

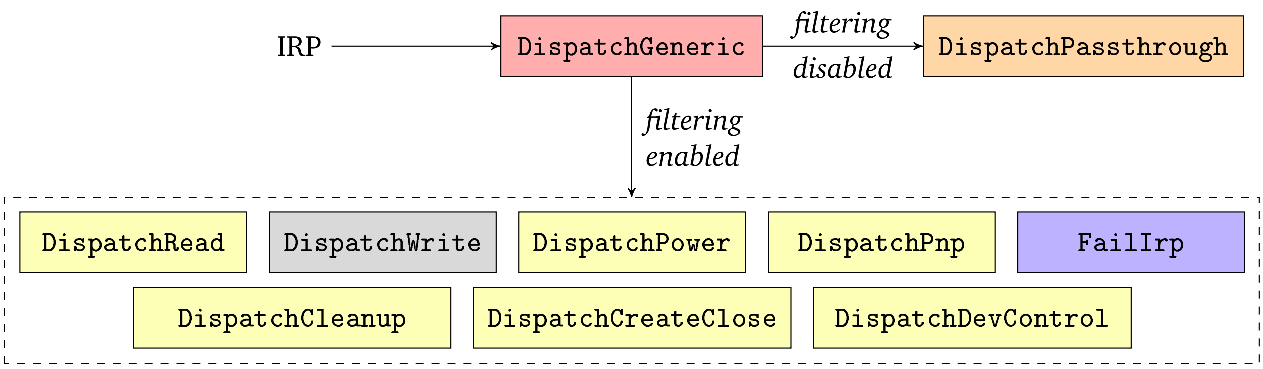

Figure 5.5: IRP flow through `luks2flt`'s dispatch routines. All IRPs first arrive at the generic dispatch routine. If filtering has not been enabled for the device, all IRPs (except for two special cases described shortly) are passed on to the next-lower driver in the device stack. If it has been enabled, IRPs that have a dispatch routine corresponding to their major function are sent to this routine. All other IRPs are rejected.
`DispatchWrite` is greyed out because while an implementation exists, we disabled write access and fail Write IRPs. See Section 5.2.7 for the reasons why we made this decision.
If a PnP remove request or `IOCTL_DISK_SET_LUKS2_INFO` arrives, it is always handled by `DispatchPnp` or `DispatchDevControl`, respectively. These routines for filtered devices already implement handling those requests, and their implementations work just as well for non-filtered devices.

Read dispatch routine registers a completion routine. It will be called when the lower-level drivers have successfully completed the IRP.[10] The only job of the completion routine is to decrypt the read data. Again, this is accomplished using our AES-XTS implementation.

To minimize the probability of errors, we did not implement all cryptographic algorithms from scratch. Instead, we adapted the AES and AES-XTS implementations of two existing Rust libraries,[11] translating them to C code.

The AES implementation makes use of specific hardware support for AES in CPUs, which was first introduced in 2010 [43]. This makes it faster than a software-only implementation of AES. Because LUKS2 only uses AES-128 or AES-256, we only implemented these two, and did not implement AES-192.

The AES-XTS implementation builds on AES and provides an interface to encrypt or decrypt one sector of data[12] at a time, together with a provided tweak.

Using the AES-XTS functions requires previous initialization of an `Xts` struct, which stores metadata required for en- and decryption. This includes the key, transformed into a representation that can be used by the AES algorithm. Because this struct only needs to be initialized once, this happens when the IOCTL that tells `luks2flt` to start filtering arrives. Figure 5.6 shows the usage of the implementation for decrypting one or more sectors read from disk.

[10] Should a lower-level driver fail the IRP, our completion routine will not be called, as there is nothing to decrypt.

[11] `https://github.com/RustCrypto/Block-ciphers` and `https://github.com/pheki/Xts-mode`

[12] Recall from Section 2.1.4 that AES operates on blocks of 16 bytes. AES-XTS however operates on sectors of arbitrary (but constant for a given device) length; the only requirement is that the sector size is greater than or equal to the block size [8].

```
VOID
DecryptReadBuffer(
    PUINT8 Buffer,
    PLUKS2_VOLUME_INFO VolInfo,
    PLUKS2_VOLUME_CRYPTO CryptoInfo,
    UINT64 OrigByteOffset,
    UINT64 Length
) {
    UINT64 Sector = OrigByteOffset / VolInfo->SectorSize;
    UINT64 Offset = 0;
    UINT8 Tweak[16];

    while (Offset < Length) {
        ToLeBytes(Sector, Tweak);
        CryptoInfo->Decrypt(
            &CryptoInfo->Xts, Buffer + Offset,
            VolInfo->SectorSize, Tweak
        );
        Offset += VolInfo->SectorSize;
        Sector += 1;
    }
}
```

Figure 5.6: Using our AES-XTS implementation to decrypt read sectors. `VolInfo` stores the information received in `luks2filterstart.exe`'s IOCTL, as described in Section 5.2.3. `CryptoInfo` stores the aforementioned `Xts` structure, which holds all data required for cryptographic operations with a given key, plus two function pointers to a `Decrypt` and `Encrypt` routine. The values of these pointers (i.e. the functions they point to) depend on the key size (128 or 256 bits). `OrigByteOffset` is the offset that the Read IRP originally contained (as opposed to the offset stored after `luks2flt`'s modifications). The `ToLeBytes` function converts the sector number to its 64-bit little-endian representation (this was introduced as the "plain64" IV generator in Section 2.1.4).

5.2.6 Handling Other Request Types

Although Read and Write IRPs are the most important and interesting, some other IRP types also need to be supported for the driver to work. In this section, we will describe our approach to handling these other requests when filtering is active. All IRPs with major functions that are not explicitly mentioned in the following are rejected by `luks2flt`.

Device Control Requests The dispatch routine for device control requests handles and responds to all IOCTLs that might be sent to a volume. There are a lot of different IOCTLs[13], which makes this a tedious task.[14] Aside from handling the aforementioned `IOCTL_DISK_SET_LUKS2_INFO`, we only implemented support for the IOCTLs that seemed to be necessary for the volume to function correctly.

If `luks2flt` receives `IOCTL_DISK_GET_LENGTH_INFO`, `IOCTL_DISK_GET_PARTITION_INFO`, `IOCTL_DISK_GET_PARTITION_INFO_EX`, or `IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS`, it registers a completion routine. The responses to all of these include information about the starting offset and/or length of the volume. The offset and length of the decrypted data on disk are different from the values for the underlying volume because of the LUKS2 metadata. The completion routine therefore modifies these values in the responses sent by the lower-level drivers.

We also investigated part of the other IOCTLs that our driver received, and identified some that we believe do not require intervention or modification.[15] These are therefore allowed to pass through. All other IOCTLs are blocked to prevent false information being sent by a lower-level driver.

PnP Requests The PnP manager notifies drivers about PnP events through IRPs with the `IRP_MJ_PNP` major function. These are sent e.g. during device enumeration or when a device is (about to be) removed from the computer. The exact request type is specified in the IRP's minor function code [42].

Some of the existing minor function codes must be supported by all (non-legacy) WDM drivers, others are optional [42]. We chose to ignore all optional requests (i.e. just pass them to the next driver), and handle the required ones as follows:

- Devices can be started, stopped, and restarted by the PnP manager. This has to do with hardware initialization and resource allocation, e.g. allocating I/O ports. Because `luks2flt` does not allocate any hardware resources, it more or less ignores the minor functions of this category.[16]

- Devices can also be removed, e.g. when the user ejects hardware or removes a device without informing the OS about it before.[17] Drivers can fail remove requests, but `luks2flt` has no reason to do so. It more or less ignores the requests

[13] See `http://ioctls.net` for an unofficial list. The IOCTLs starting with `IOCTL_DISK`, `IOCTL_STORAGE`, `IOCTL_VOLUME`, and `IOCTL_MOUNTDEV` are relevant for volume devices.

[14] The Device Control dispatch routine of the VeraCrypt driver stretches over 1,000 lines of code and handles 38 different IOCTLs (l. 833 to 1889 in `src/Driver/Ntdriver.c`).

[15] See the `Luks2FltDispatchDeviceControl` routine in the source code.

[16] Which are `START_DEVICE`, `QUERY_STOP_DEVICE`, `CANCEL_STOP_DEVICE`, and `STOP_DEVICE`. Also note that between Stop and Start requests, drivers theoretically need to queue all incoming IRPs that must not be dropped. Because our driver has no queuing functionality, we rely on the lower-level drivers to do that.

[17] Devices are also removed for driver updates and re-added after the new driver version has been loaded.

that warn about imminent removal[18] and only in reaction to the REMOVE_DEVICE request it detaches from the device stack and deletes its device object.

– The last required minor function is the DEVICE_USAGE_NOTIFICATION, which lets a driver know that a device contains a paging, dump, or hibernation file. This may have implications for Stop or Remove requests. luks2flt does not care about this and lets the lower-level drivers decide how such a usage impacts device operation.

Create, Close, and Cleanup Requests A Create request is sent when a handle to a file representing a device (e.g. \Device\HarddiskVolume1) is requested e.g. by a user mode program. Conversely, the Close and Cleanup requests are sent when the last handle to a device is closed, so that drivers may cancel outstanding requests and deallocate resources. The difference between these two is that Cleanup requests are sent regardless of whether there are any outstanding I/O requests. Close requests are only sent after all outstanding I/O requests have either been cancelled or completed [42]. luks2flt does not impose any restrictions on opening handles and does not allocate any resources for handles. It just passes these requests on to the next driver, which is therefore also responsible for cancelling any outstanding operations. Decompilation of the BitLocker driver shows that, except for some special cases, does the same.

Power, Flush Buffers, Shutdown, and System Control Requests These are not relevant to our driver and are therefore just passed on to the next driver [42]:

– Power requests are sent by the power manager to control the power status of devices.

– A Flush Buffer request instructs a driver to flush its internal buffer or the cache of a device.

– File system drivers send Shutdown requests to storage device drivers to notify them that the system is about to shut down. In response to this, the data stored in internal data caches will be written to disk before shutdown.

– System Control requests are implemented by drivers that use the *Windows Management Instrumentation (WMI)* to provide instrumentation and measurement data.

5.2.7 The Problem With Writing to a Volume

As already hinted at, enabling write access in our driver caused some problems. After about a minute of working with the decrypted files, they disappeared and instead files with strange-looking names and incorrect sizes were displayed. An example of a corrupted volume can be seen in Figure 5.7. It also describes a possible set of steps to reproduce the corruption. However, in our tests it was enough to just wait for about a minute without interacting with the volume at all for the corruption to appear.

To limit the number of possible causes, we disabled every IRP major function that was not needed for basic volume operation. This left mostly those described in Section 5.2.5 and Section 5.2.6. We also changed luks2flt to reject all device control requests, except for a few basic ones (again, mostly those described in Section 5.2.6). However,

[18] QUERY_REMOVE_DEVICE, CANCEL_REMOVE_DEVICE, and SURPRISE_REMOVAL. The last one is not really a warning but occurs when hardware is suddenly removed, but it is ignored as well.

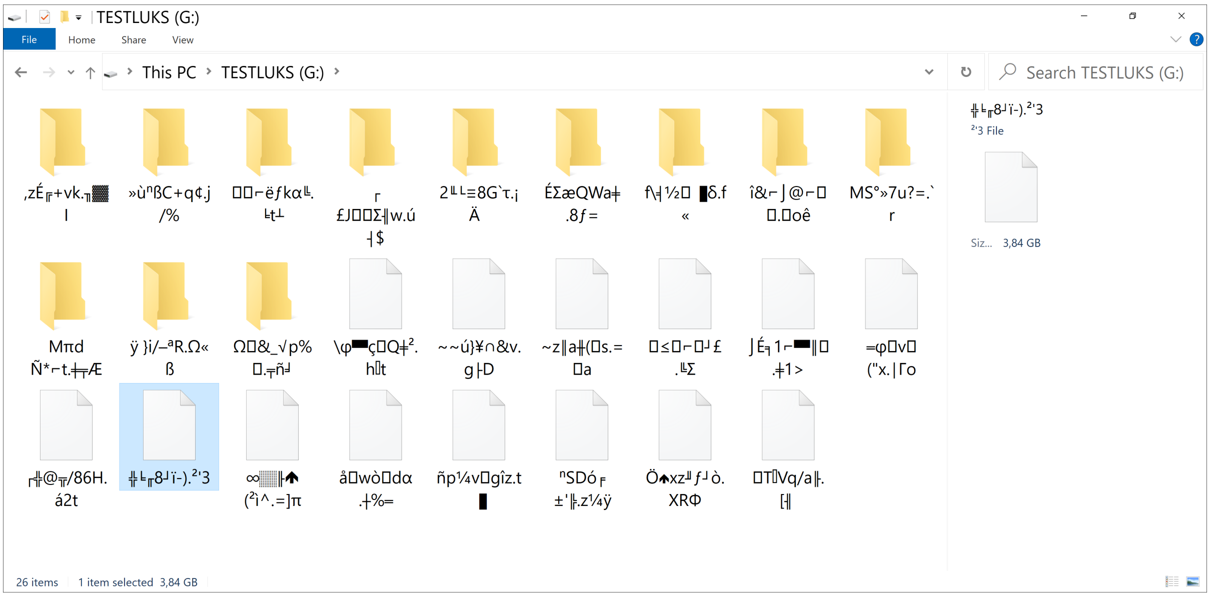

Figure 5.7: Corrupted volume after enabling write access (screenshot from Windows Explorer). The volume originally contained one small text file. We then changed the content of this file, created a new folder, and there created another text file. After closing and reopening the text files and refreshing the view in Windows Explorer, the files and folders disappeared. Instead, multiple folders and files with seemingly random names, sizes, and creation times were displayed. The volume label however remained correct. Trying to open a file resulted in the following error message: "The specified path does not exist." Trying to open a folder failed with "The filename, directory name, or volume label syntax is incorrect."

even with this most restricted set of allowed operations, the corruption still occurred. As soon as we disabled Write access, we could not reproduce the corruption.

Interestingly, the corruption seems to follow a pattern. In tests using a virtual machine we always used the same virtual disk, backing it up before corruption and restoring it afterwards. Every time the volume got corrupted, there were multiple folders and files with the same name and, in case of the files, the same size. For example, the file highlighted in Figure 5.7 was always there and always allegedly 3.84 gigabytes large. However, there were also other files and folders that were different each time.

The findings from Figure 5.8 give a clue for what might be the problem: the actual files seem to still be there, but something causes other, actually non-existent files to be shown. Our suspicion is that only the filesystem's metadata gets corrupted, leading to confusion of what the contents of the volume are. This would fit with the observation that the "real" files still seem to be there.

This theory is also backed by findings obtained using the `fatcat` tool.[19] This is a forensic tool for examining FAT filesystems, because we used FAT32 inside LUKS2 for our tests. First explorative usage of it showed to things: firstly, it was able to recover the files (including content, not only names) and folders we created before the corruption happened (similar to Figure 5.8); and secondly, it reported that the two copies of filesystem metadata blocks stored on the volume did not match.[20]

[19] `https://github.com/Gregwar/fatcat`

[20] It printed out the contents of both blocks in hexadecimal form, and in fact, they were completely different.

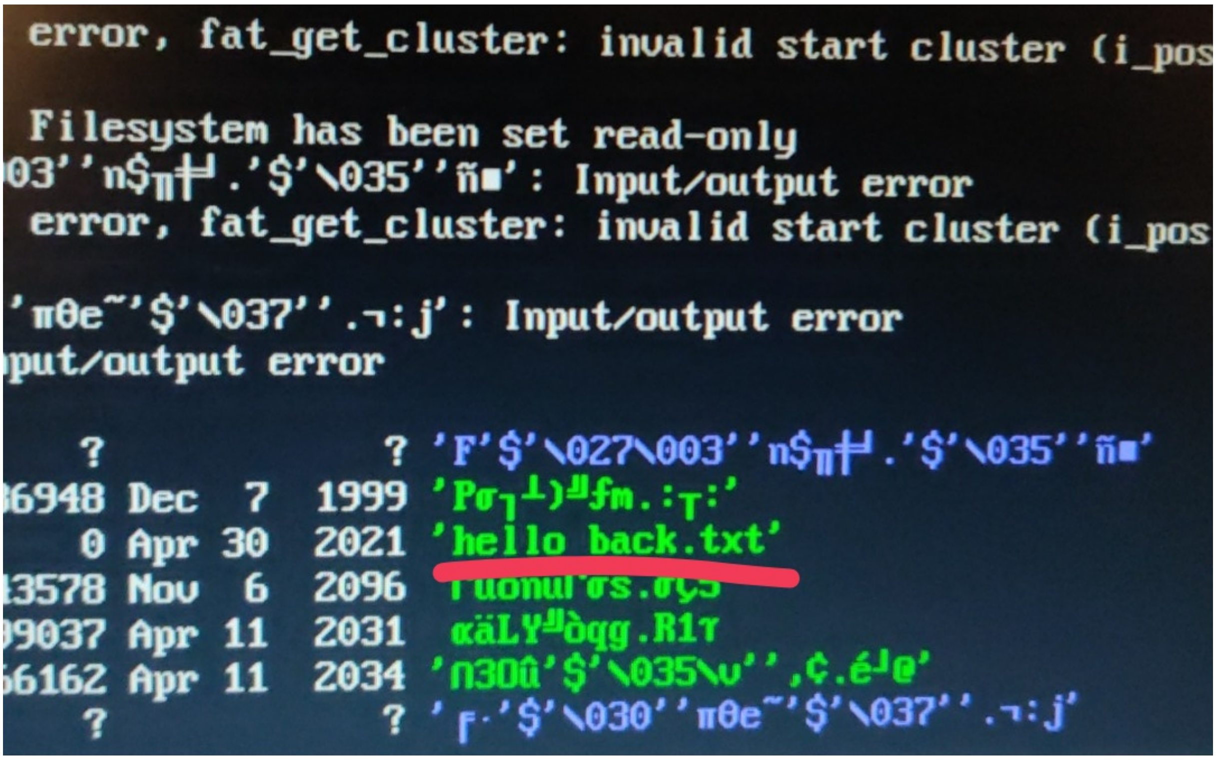

Figure 5.8: Corrupted but mountable volume after enabling write access (photographed output of the Linux `mount` command). After one of our tests that lead to the corruption of a LUKS2 volume, we tried unlocking and mounting the volume in Linux. The unlocking worked without a problem, but during the mount process multiple warnings and errors occurred. These can partially be seen in the photo. Listing the contents of the volume showed several corrupted files, similar to Windows Explorer, but also one text file that we created. Its file size was reported as zero bytes, which was correct, as we only created the file and did not write any content to it. The file creation date as well as the file name were also correct.

As to why the metadata might get corrupted, we still lack a theory. It could be that some IRP major functions or IOCTLs that we categorized as harmless actually are the cause of the problem. Maybe the filesystem driver obtains wrong information about e.g. the volume's size or offset through one of the still allowed requests. If the cause is a required type of IRP or IOCTL, it cannot be ruled out by exclusion.

To completely verify our assumption that only the metadata becomes corrupted and to hopefully find the error cause, we would need to take a more thorough look at the metadata. A first step would be parsing the metadata format of the corrupted FAT32 partition and inspecting the stored values. Regrettably, we lack the time for such an extensive research. Therefore, we note that this issue needs more investigation, and for now leave the write support disabled.

5.3 Security and Privacy Considerations

Security and privacy of confidential data was not a focus in our work, but we still thought about these topics. Following are our considerations on them.

Security As every kernel component has access to the complete physical RAM and the highest possible privileges, bugs in it can lead to the whole system crashing, or worse, being fully compromised. Especially the handling of untrusted input must be done with care.

In the case of `luks2flt`, there are two sources of user input: the data sent via `IOCTL_DISK_SET_LUKS2_INFO` and the data for Write IRPs.

To send IOCTLs, a user must open a device handle, for which they need administrator privileges. A theoretical attacker would thus need to already have those access rights. Therefore, any possibly existing security vulnerability in the handling of this IOCTL would be of low impact. If an adversary has admin rights, the system is already more or less completely compromised. Of course, we still were careful when implementing this functionality.

As we understand it, when a Write IRP arrives at our driver, the values of parameters like the buffer size have already been checked by Windows. The data to be written is only touched by the encryption routines. The implementations of the algorithms are not our own and, assuming we translated them correctly from Rust to C, should not contain vulnerabilities.

Of course, these claims must be verified by an independent party, should certainty be required (i.e. if someone wanted to use `luks2flt` in a production environment).

Privacy As we have discussed in Chapter 4, all described implementations take care to overwrite memory containing sensitive data before deallocating it. Most importantly, they all zero out any cryptographic keys when not needed any more.

`luks2flt` does this as well, on two occasions: after copying the data into the device extension, the buffer used for `IOCTL_DISK_SET_LUKS2_INFO` is overwritten with zeros;[21] and when deleting a device object in response to a PnP remove request, the device extension is also zeroed. Both times, the `RtlSecureZeroMemory` routine is used to clear the buffer. This is the same as VeraCrypt, and as explained in Section 4.2.2, a call to it is guaranteed not to be optimized out by the compiler.

`luks2filterstart.exe` also handles sensitive data and is responsible for overwriting it with zeros. However, we did not implement this, because Rust makes it non-trivial and this was not our focus.[22]

It should be noted that despite these efforts, cryptographic key material may still be recovered by attackers from a running system with unlocked volumes. [44] details successful coldboot attacks against BitLocker and TrueCrypt (VeraCrypt's predecessor). [45] describes a possible defence strategy, however requiring hardware support.

[21] This IOCTL always uses buffered I/O, therefore we receive a copy of the user buffer from the I/O manager, which will free it after our usage.

[22] There are partial solutions to this problem, e.g. the `secrecy` crate, but there are still some problems left to solve (for example, this crate cannot be easily used with the cryptography crates that `luks2filterstart.exe` uses).

6 Performance of Our Driver: Results and Discussion

To compare the performance of our driver to BitLocker and VeraCrypt, we conducted several experiments measuring the read rates for both sequential and random access. Preliminary tests were done in a virtual machine, and after some optimizations, we tested more extensively on real hardware. Finally, we also measured how the read rates behave over time.

Appendix A describes how to obtain copies of both the data collected in our measurements and the code that generated the figures presented in this section.

6.1 Experimental Setup

Both the setup for the virtual machine and the real hardware will be described in this section. They are mostly the same, so unless stated otherwise, the contents in this section apply to both.

Both configurations used a fresh Windows 10 build 19043.928 install. The virtual machine ran in Oracle VirtualBox 6.1.22 with 8 GiB of virtual RAM (backed by 16 GiB DDR4 RAM), 4 virtual cores (backed by a Intel Core i7-7700K), and two virtual disks (backed by a 232 GiB Samsung SSD 850 EVO). The real machine had 8 GiB DDR3 RAM, an Intel Core i7-3770 quad core with simultaneous multithreading, and a 233 GiB SanDisk SDSSDH3250G.

For each encryption technique featured in our benchmarks we created a separate 5 GiB FAT32 volume on the same disk, as shown in Figure 6.1. Every encrypted volume contained the same file, `random.bin` (4 GiB of random data). We configured BitLocker to use the new AES-XTS mode, all other settings were left unchanged. For VeraCrypt, we also used AES-XTS with SHA-256 and also mounted the volume as read-only after copying the `random.bin` file to it.

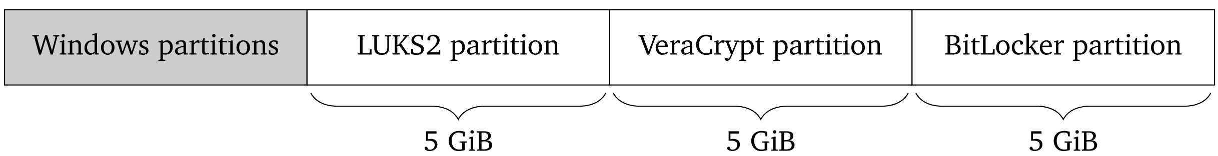

Figure 6.1: Volume layout for our experiments. This is the layout of the real hardware SSD. The virtual machine had a virtual disk for the Windows installation and another one for the three encrypted volumes.

To compare the performance of the different drivers, we used version 3.27 of the `fio` command line tool [46] to read from the three copies of `random.bin`, using different access patterns. Each run of `fio` is called a *job*, and the configuration for a job is stored in a *job file*. We maintained a different job file for each of the three encryption technologies. Figure 6.2 describes the different parameters that we used and shows the configuration for `luks2flt`. `fio` allows the user to specify which I/O engine to use. The default on Windows is the native asynchronous I/O API, which is what we used.

```
[luks2flt]
filename=L\:\random.bin
thread
iodepth=${IODEPTH}
numjobs=${NUMJOBS}
rw=${MODE}
bs=${BLOCKSIZE}
```

Figure 6.2: `fio` job file for `luks2flt`. It defines one job called `luks2flt`. The `filename` option's value is the only one that depends on the used driver; the rest of the configuration is the same for BitLocker and VeraCrypt. The `thread` option means that `fio` will create threads instead of processes to run jobs (which is always done on Windows, but when not specifying this option, the user is warned about this behaviour). The `iodepth` parameter controls how many I/O operations `fio` will try to submit before waiting for the completion of submitted requests (i.e. the size of the internal queue of submitted I/O requests). `numjobs` controls how many threads will concurrently execute the job (note that each thread does a full run of the job, i.e. each thread reads the complete file and has its own queue with the size specified by `numjobs`). `rw` holds the access mode, in our case it is always `read` (sequential reading) or `randread` (random-access reading). The blocksize of read operations, i.e. how the number of bytes read in a single I/O operation, is given by the `bs` option. All parameter values of the form `${PARAM}` are configured by environment variables [46].

We also looked at other I/O benchmarking programs, but found that they all required write access. This unfortunately made them unfit for our cause.

We wrote a PowerShell script to automate running jobs with different parameters. Its task was to set the environment variables used in the job files to their respective values and to invoke `fio`, pointing it to the correct job file. A sample invocation looks like this: `fio --readonly --output-format=json --output luks2flt.json luks2flt.fio`. The `--readonly` option is necessary because even though we only specified read operations, execution would fail for read-only volumes. The data visualized by the graphs in the following sections was taken from `fio`'s JSON output.

6.2 Virtual Machine Experiments

To get a first impression of `luks2flt`'s performance, we ran a first series of experiments in a virtual machine.[1] These experiments were only conducted once, which leaves room for variation in read rates. Therefore the results in this section need to be taken with a grain of salt. We will address this issue in Section 6.3.

When we looked at the results of our measurements, we were convinced that there was still some optimization possible. We first checked the compiler's optimization settings, and found that the default compiler optimization settings in Visual Studio were

[1] Note that this virtual machine had not been started in debug mode and was not connected to a WinDbg kernel debugging session, so as to not impact performance.

a bit conservative. They were also set to optimize for small code size rather than high speed/performance. We therefore tuned these settings to enable more aggressive optimizations and also focus on speed rather than code size. This led to a more optimized version of `luks2flt`. The results presented in this section feature both the first and the more optimized version.

Results and Observations Figure 6.3 shows a comparison of the read rates for sequential access, with block sizes ranging from 4 to 8192 KiB. The featured drivers are BitLocker, VeraCrypt and the two versions of `luks2flt` (before and after enabling more optimizations). BitLocker consistently has the highest read rate, starting at 171 MiB/s for 4 KiB blocks, rising with bigger blocks, until peaking at 469 MiB/s for a block size of 4096 KiB. VeraCrypt generally has the second-highest rate, starting at 154 MiB/s, with an upwards trend until the maximum of 341 MiB/s is reached for 256 KiB blocks. It then slowly falls back down to 329 MiB/s. The optimized `luks2flt` version comes third, starting at 144 MiB/s and rising to 344 MiB/s, beating VeraCrypt at the largest block size. `luks2flt`'s less optimized variant always has the lowest read rate, beginning at 119 MiB/s and climbing to 217 MiB/s.

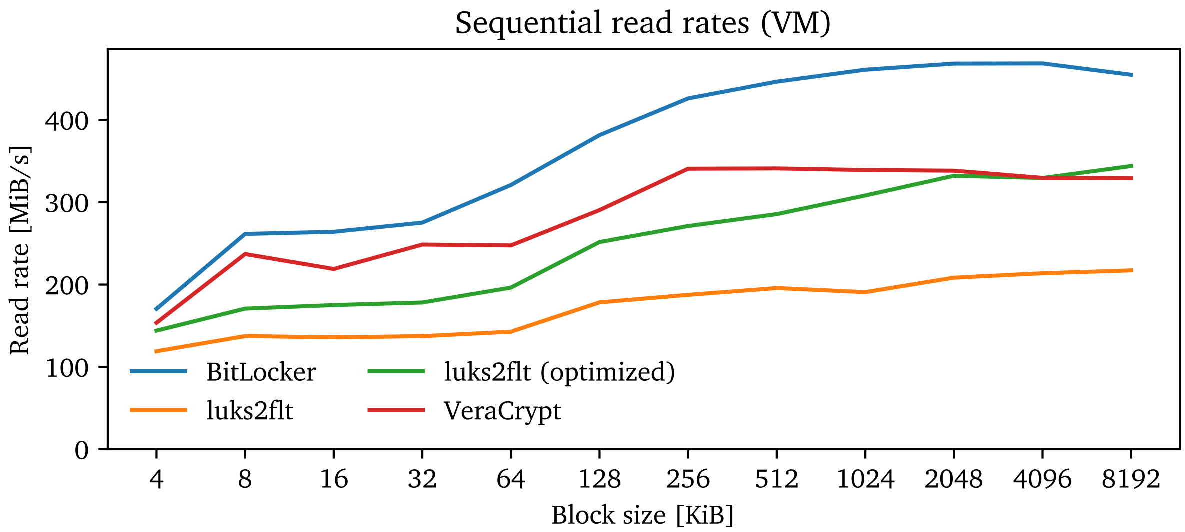

Figure 6.3: Sequential read rates in the virtual machine. The data was obtained by running jobs back to back: one job per combination of driver and block size, grouped by driver and ordered by increasing block size. There was a reboot after the BitLocker, VeraCrypt, and optimized `luks2flt` jobs to load the less optimized version of `luks2flt`. Both the `iodepth` and `numjobs` parameters were set to a value of 1.

We observe the following:

– BitLocker's sequential read rate is greater than our driver's and VeraCrypt's for all tested block sizes.

– VeraCrypt performs noticeably better at sequential reading than `luks2flt`'s optimized version, but the difference decreases with increasing block size. VeraCrypt's read rate peaks at 256 KiB, but our driver's rate continues to grow after that.

– The more optimized version of `luks2flt` has a significantly higher sequential read rate than the first version, especially at larger block sizes. At 4 KiB, the increase in performance is about 21%; at 8192 KiB the more optimized version has about a 59% higher read rate.

Figure 6.4 shows the random-access read rates of the four drivers for different block sizes. BitLocker always has the highest rate, starting at 16 MiB/s and climbing up to 388 MiB/s at 8192 KiB blocks, with a little stagnation and decrease in performance at 256 and 512 KiB. VeraCrypt has the lowest read rate of 10 MiB/s at 4 KiB, but it steadily grows: it rises to third place at 256 KiB, second place at 1024 KiB, and falls back to third at 8192 KiB, with a maximum of 291 MiB/s. The two versions of `luks2flt` both start at 16 MiB/s and rise for block sizes up to 128 KiB, switching their roles of second and third place. After that, the optimized version continues to improve its read rate up to the final value of 300 MiB/s, temporarily falling behind VeraCrypt. The less optimized version's rate first decreases a bit, then rises abruptly, and then decreasing a bit again, with a maximum value of 144 MiB/s at 1024 KiB.

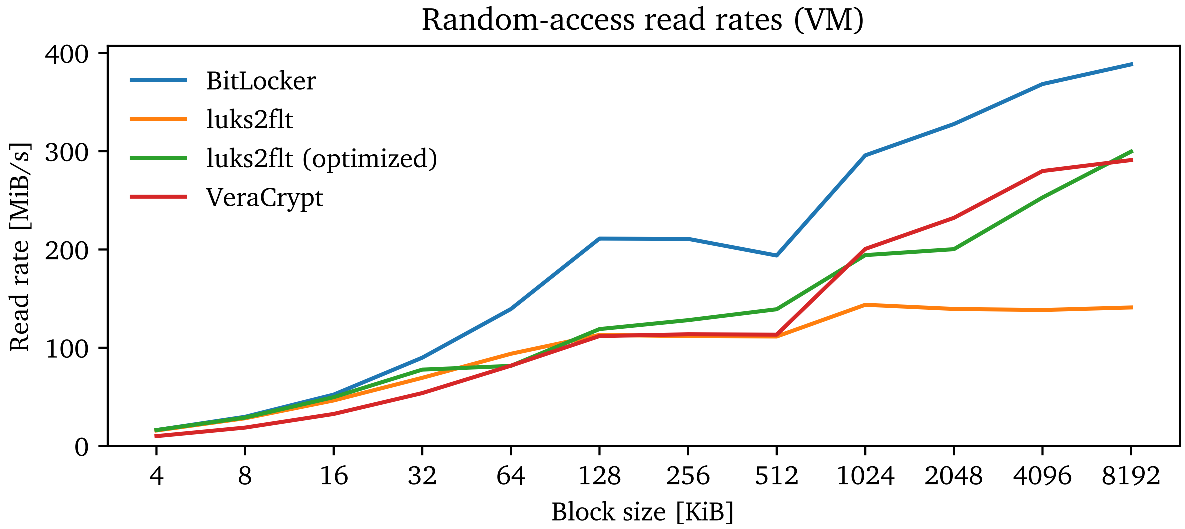

Figure 6.4: Random-access read rates in the virtual machine. See Figure 6.3 for how we collected the data.

We observe the following:

– BitLocker has the highest random-access read rate, with sometimes almost double the rate of the others (e.g. at 128 KiB).

– The optimized version of `luks2flt` beats VeraCrypt's random-access read rate at two thirds of the tested block sizes. For example, its read rate is 60% higher at 4 KiB, 52% higher at 16 KiB, and 14% lower at 2048 KiB.

– Starting from a block size of 128 KiB, the more optimized version has a significantly higher random-access read rate than the first version. For the biggest block size, the more optimized version has a 113% higher read rate.

Discussion Combining our observations from both access patterns, we close with the following thoughts:

– BitLocker is the best-performing driver, both at sequential and random-access reading. This is not really surprising, as it is not a fair competition: one driver is the proprietary product of Microsoft, who also wrote the operating system running the driver; and the other two are open source projects by third parties. Microsoft's driver profits from their expertise in Windows' inner workings, and can also take advantage of undocumented internals.

- VeraCrypt has a higher sequential read rate than `luks2flt`, with the difference decreasing for larger block sizes. The former is probably because of VeraCrypt's read ahead caching, as `luks2flt` does not perform any caching. The latter may be because of VeraCrypt's fragmenting of larger blocks into 256 KiB blocks. This means that the I/O performed by the driver can only profit so much from block sizes larger than 256 KiB. This theory is supported by the fact that VeraCrypt reaches its peak sequential read rate at exactly this block size. As already mentioned in Section 4.2.2, the fragmenting is done to overlap I/O and cryptographic operations. However, the alleged gains from this overlapping do not seem to be able to cancel out the losses by capping the effective block size at 256 KiB.

- For random-access reads, the performance difference between VeraCrypt and `luks2flt` is smaller, with our driver showing higher read rates most of the time. This supports the thesis that VeraCrypt's performance advantage for sequential access stems from its caching: for random-access reads the cache is not effective, and therefore the advantage vanishes.

- The changed compiler optimization settings have a significant impact on our driver's read performance, regardless of the access pattern.

Another Optimization Attempt We also tried optimizing `luks2flt`'s performance by restricting its cryptographic capabilities to AES-256-XTS and dropping support for AES-128-XTS. This enabled removing some `if-else` constructs that dispatched de-/encryption functions based on whether AES-128 or AES-256 was used. Even though these conditionals were located in a performance critical section, we found no significant read rate changes. Our theory for why this was the case is the following: in this setup, our driver was only ever used for one LUKS2 partition and therefore always took the same path through the `if-else` (either always AES-128 or always AES-256). This trained the CPU's branch prediction on this one specific path. Thus, after a short training phase, the CPU always speculatively executed the correct path, resulting in the same performance as without the `if-else`. The same reasoning applies for a use case where `luks2flt` is filtering multiple volumes that all use the same AES-XTS variant, although we did not test this.

6.3 Real Hardware Experiments

After gaining first insights through the virtual machine experiments, we turned to real hardware for a deeper understanding. To combat the issue of variations in read rates due to external circumstances, we did not rely on measurements from single runs. Instead, we ran each job three times and calculated the average read rate. The same job was not run three times back to back; instead, we ran a first series of all jobs, then a second series, and finally a third. Thus, there was always quite some time between running the same job again, which hopefully eliminated most of the external influences.

6.3.1 Measuring Read Rates for Different Block Sizes

The further optimizations of `luks2flt`, as described in Section 6.2, unfortunately led to reproducible crashes on real hardware.[2] We disabled some of the optimizations again (Whole program optimization, Spectre mitigations, Link-time code generation, and Frame-pointer omission), and did not find any more crashes. Thus, we used this slightly less optimized version for the real hardware experiments.

We also ran our experiments on a (real hardware) HDD. This generally showed the same trend as on the SSD, or `luks2flt`'s read rate could keep up with the other drivers' even better.

Additional to the parameter combinations already used for testing in the virtual machine, we also ran experiments where either one or both of the `iodepth` and `numjobs` parameters had a value of 16 instead of one. For jobs with `numjobs=16`, we took the average of all threads as the read rate for one run of this job. As we ran each job three times, these thread averages were then averaged again to obtain the final read rates used in the diagrams.

Results and Observations Figure 6.5 shows the sequential read rates of BitLocker, VeraCrypt, and `luks2flt`, for block sizes from 4 KiB to 8192 KiB. BitLocker's rate starts at 384 MiB/s, and after a small decrease rises to its maximum value of 473 MiB/s at a block size of 1024 KiB. It then decreases, and finally increases again, finishing at 425 MiB/s. VeraCrypt's read rate for 4 KiB blocks is 250 MiB/s, and after a slight decrease the rate climbs to a maximum of 432 MiB/s at 512 KiB. Alternating between decrease and increase, its read rate at 8192 KiB is 384 MiB/s. `luks2flt` starts with a read rate of 155 MiB/s and, with minor drops in between, the rate grows to a value of 228 MiB/s for the largest block size.

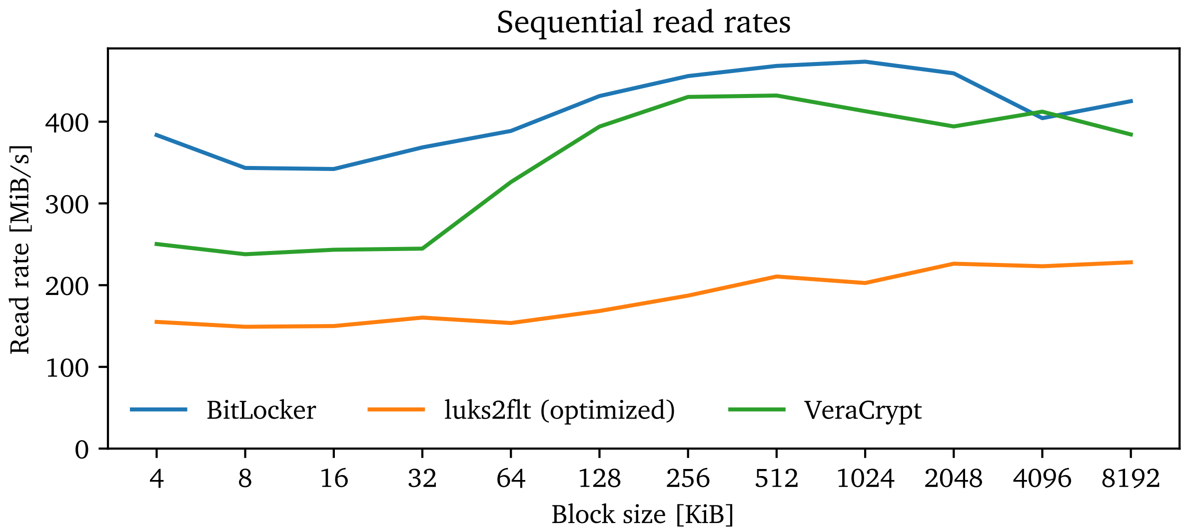

Figure 6.5: Sequential read rates. Each block size value is the average of three separate runs. Both the `iodepth` and `numjobs` parameters were set to a value of 1.

[2] Temporarily enabling debug mode and attaching a WinDbg remote kernel debugging session showed that the `fastfat` driver crashed. This is Windows' built-in FAT filesystem driver. We did not investigate this further, as the crashes did not persist after retuning the compiler optimization settings.

We observe the following:

– Except for a block size of 4096 KiB, where VeraCrypt has a 2% higher rate, BitLocker always has the highest sequential read rate. It is up to 156.0% higher than `luks2flt`'s (at 128 KiB), and up to 53.0% higher than VeraCrypt's (at 4 KiB).

– BitLocker reaches its maximum at a block size of 1024 KiB, VeraCrypt at 512, `luks2flt` at 8192.

– `luks2flt` never reaches the performance of the other two drivers. BitLocker has an up to 156% higher read rate (at 128 KiB), VeraCrypt up to 134% (at 128 KiB).

Figure 6.6 contains the random-access read rates of the three drivers. BitLocker starts with a read rate of 19 MiB/s and, except for a dent at 4096 KiB, continually rises to a maximum of 335 MiB/s. `luks2flt`'s rate also begins at 19 MiB/s and steadily climbs to 195 MiB/s. VeraCrypt's rate at 4 KiB is 13 MiB/s, and then rises up to a value of 301 MiB/s at 8192 KiB.

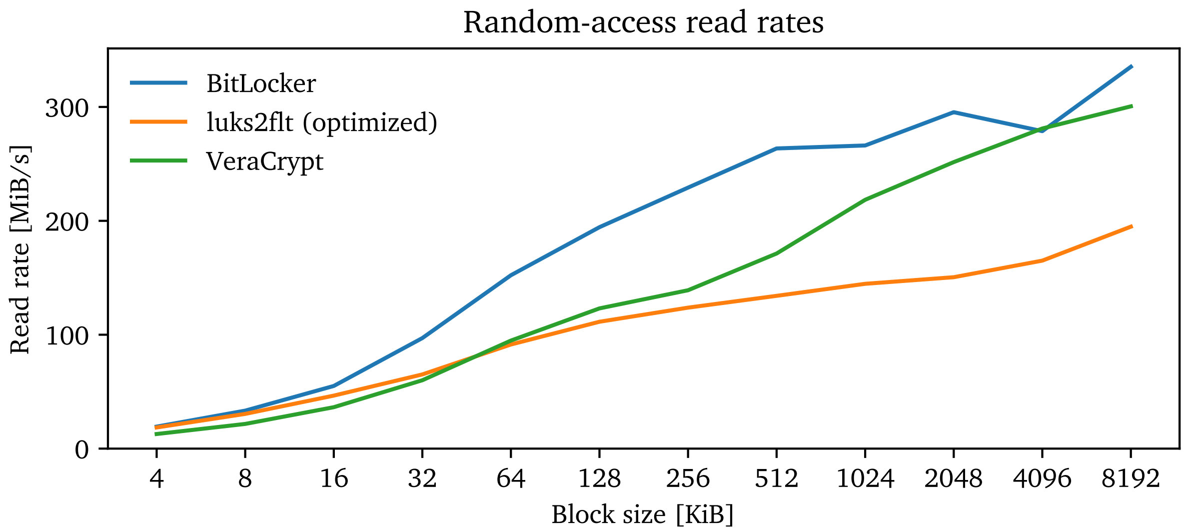

Figure 6.6: Random-access read rates. See Figure 6.5 for how we collected the data.

We observe the following:

– Except for a block size of 4096 KiB, where VeraCrypt has a 1% higher rate, BitLocker always has the highest random-access read rate.

– For blocks smaller than 64 KiB, `luks2flt` has a higher random-access read rate than VeraCrypt. For large block sizes, the other two drivers have significantly higher read rates: BitLocker up to 96% (at 512 KiB), VeraCrypt up to 70% (at 4096 KiB).

Figure 6.7 contains the sequential read rates of the three drivers with the `iodepth` parameter set to a value of 16. This means that `fio` submitted up to 16 I/O requests before waiting on the completion of requests. BitLocker's read rate starts at 344 MiB/s and, after a descent, rises to 521 MiB/s. VeraCrypt's read rate at 4 KiB is 241 MiB/s, and its maximum is 433 MiB/s at 8192. Its graph has a small dent from 1024 to 4096 KiB. `luks2flt` begins with a read rate of 149 MiB/s and rises to its maximum of 391 MiB/s at 4096 KiB.

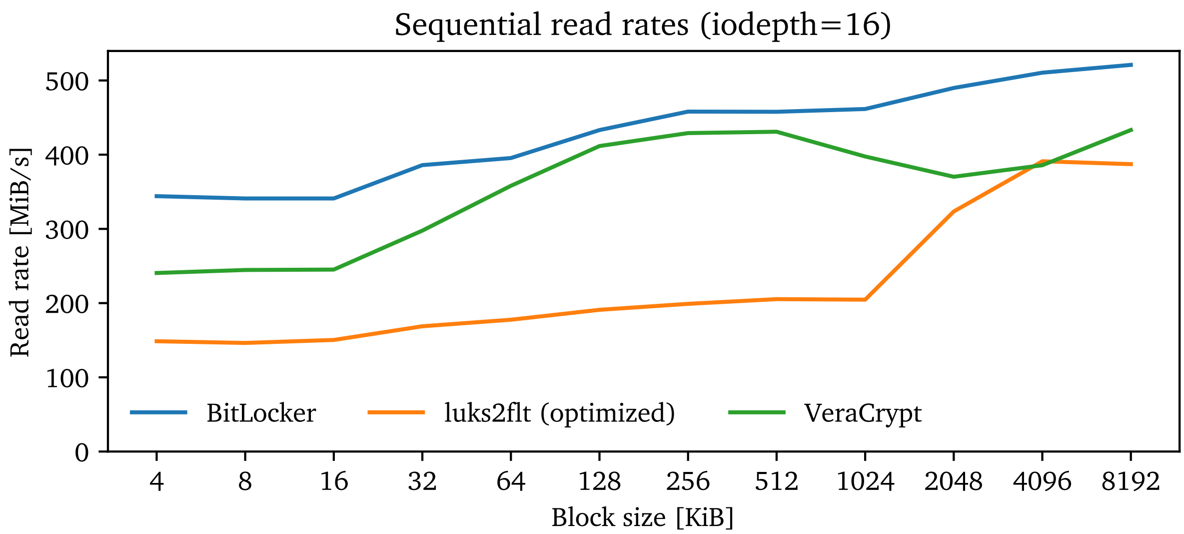

Figure 6.7: Sequential read rates with `iodepth=16`. Each block size value is the average of three separate runs. The `numjobs` parameter was set to a value of 1.

We observe the following:

– BitLocker always has the highest sequential read rate, but in the range of 64 to 512 KiB blocks VeraCrypt's rate is relatively close to BitLocker's. At 8 KiB, BitLocker has a maximum read rate advantage over `luks2flt` of 133%.

– At 4096 KiB, our driver has a slightly higher (1%) read rate than VeraCrypt, but for smaller blocks, VeraCrypt is up to 116% faster (at 256 KiB).

Figure 6.8 shows the random-access read rates of the three drivers for an `iodepth` of 16. BitLocker's read rate at 4 KiB is 61 MiB/s. It rises continuously, although only very slowly for blocks larger than 256 KiB. Its maximum is 505 MiB/s at 8192 KiB. `luks2flt`'s rate starts at 58 MiB/s and rises more or less continuously until it reaches it maximum of 414 MiB/s. VeraCrypt begins with a rate of 25 MiB/s and climbs to its highest value of 364 MiB/s for the largest block size.

We observe the following:

– BitLocker always has the highest random-access read rate, but for small block sizes our driver's rate is relatively close to BitLocker's. However, at 256 KiB BitLocker has a 55% higher read rate compared to `luks2flt`, which is the highest advantage across all block sizes.

– `luks2flt` always has a higher read rate than VeraCrypt, up to a 133% advantage (at 4 KiB).

Figure 6.9 contains the sequential read rates of the three drivers with the `numjobs` parameter set to a value of 16. This means that 16 threads concurrently executed the same job. BitLocker begins with a read rate of 107 MiB/s, which, with a small dent at 32 KiB, climbs to its maximum of 492 MiB/s at 512 KiB. It then falls back down to 187 MiB/s at 8192 KiB. VeraCrypt's read rate starts at 146 MiB/s, then jaggedly rises to a maximum of 431 MiB/s at 256 KiB, then also jaggedly goes back down to 253 MiB/s. Our driver's read rate for 4 KiB blocks is 106 MiB/s, and, with some fluctuations, it

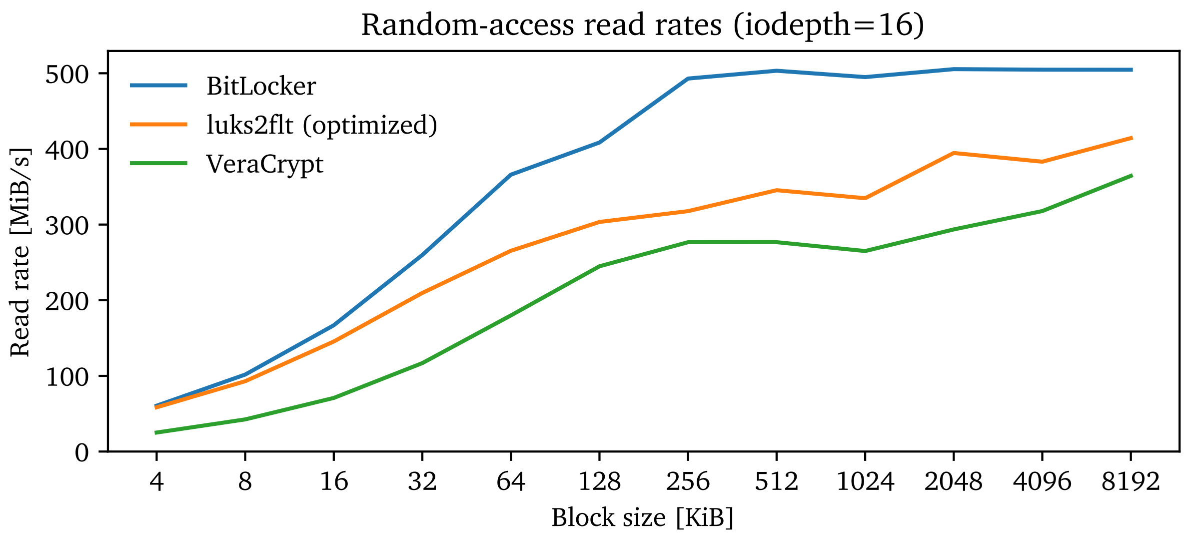

Figure 6.8: Random-access read rates with `iodepth=16`. See Figure 6.7 for how we collected the data.

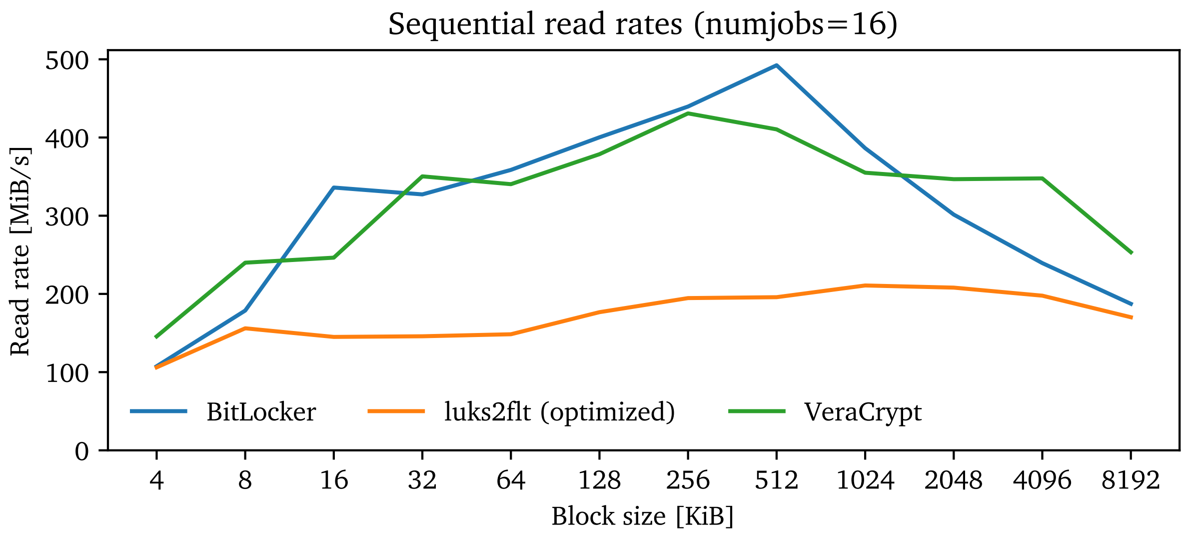

Figure 6.9: Sequential read rates with `numjobs=16`. Each block size value is the average of three separate runs, with the value used for each run being the average read rate of all threads. The `iodepth` parameter was set to a value of 1.

climbs to its maximum value of 211 MiB/s at 1024 KiB. It then slowly falls back down to 170 MiB/s.

We observe the following:

– VeraCrypt and BitLocker both have the highest sequential read rate for respectively six of the twelve tested block sizes.

– VeraCrypt reaches its maximum rate at 256 KiB, BitLocker at 512, our driver at 1024.

– Both other drivers perform significantly better than `luks2flt` for most block sizes. BitLocker has an up to 151% higher rate (at 512 KiB), VeraCrypt up to 140% (at 32 KiB).

Figure 6.10 shows the three drivers' random-access read rates for multi-threaded job execution. BitLocker's read rate starts at 32 MiB/s, reaches its maximum at 25 KiB with a value of 294 MiB/s, and then goes back down to 236 MiB/s. `luks2flt`'s rate begins at 32 MiB/s, has its maximum value of 220 MiB/s at 256 KiB, and, with a dent at 1024 KiB, slowly decreases to 212 MiB/s at 8192 KiB. The VeraCrypt driver's read rate is 19 MiB/s for 4 KiB blocks and a maximum of 203 MiB/s for 256 KiB blocks. After a dent in the range from 512 to 4096 KiB, its rate for 8192 KiB is 201 MiB/s.

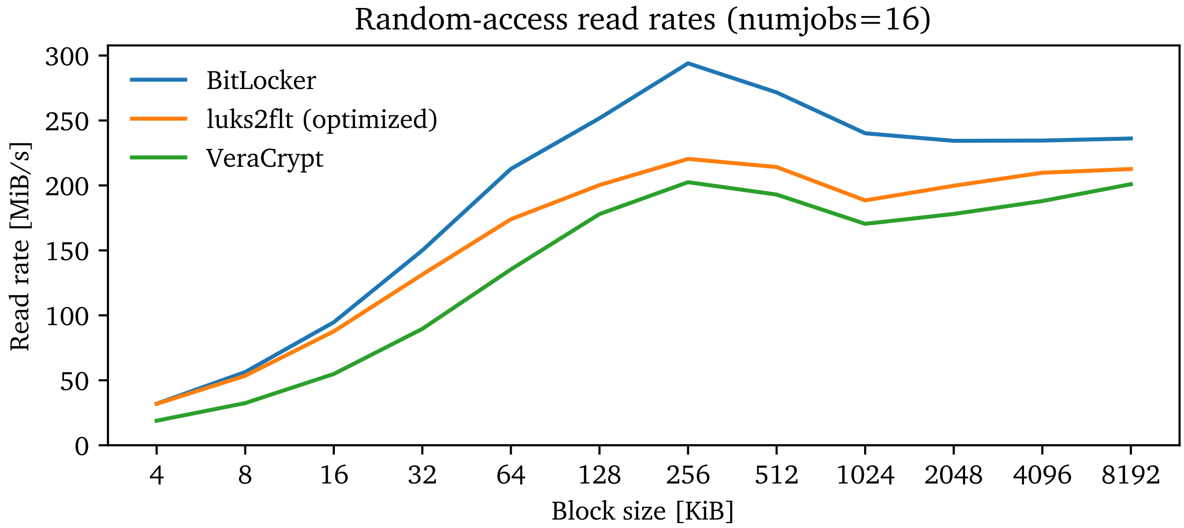

Figure 6.10: Random-access read rates with `numjobs=16`. See Figure 6.9 for how we collected the data.

We observe the following:

– BitLocker is always the fastest driver, with an up to 33% higher random-access read rate than luks2flt (at 256 KiB), and up to 74% higher than VeraCrypt (at 8 KiB).

– All drivers reach their respective maximum rate at a block size of 256 KiB.

– `luks2flt` always has a higher read rate than VeraCrypt, with an up to 69% advantage (at 4 KiB).

Figure 6.11 shows the tested drivers' sequential read rates for multi-thread job execution with an `iodepth` of 16. BitLocker starts with a read rate of 131 MiB/s, which then grows to 395 MiB/s for 32 KiB blocks. It then jaggedly decreases to 328 MiB/s at 4096 KiB, and finally falls down to 87 MiB/s. VeraCrypt's rate begins at 128 MiB/s, rises to 404 MiB/s at 32 KiB, and, after a dent at 64 KiB, slowly decreases to 317 MiB/s at 4096 KiB. The read rate at 8192 KiB is 90 MiB/s. `luks2flt`'s read rate starts at 132 MiB/s, rises mostly continuously to 275 MiB/s at 4096 KiB, and then falls down to 38 MiB/s at 8192 KiB.

We observe the following:

– Except for 16, 32, and 8192 KiB blocks, where VeraCrypt is the fastest, BitLocker has the highest sequential read rate. Compared to our driver, BitLocker has an up to 135% advantage (at 32 KiB), and VeraCrypt has an up to 165% higher rate (at 16 KiB).

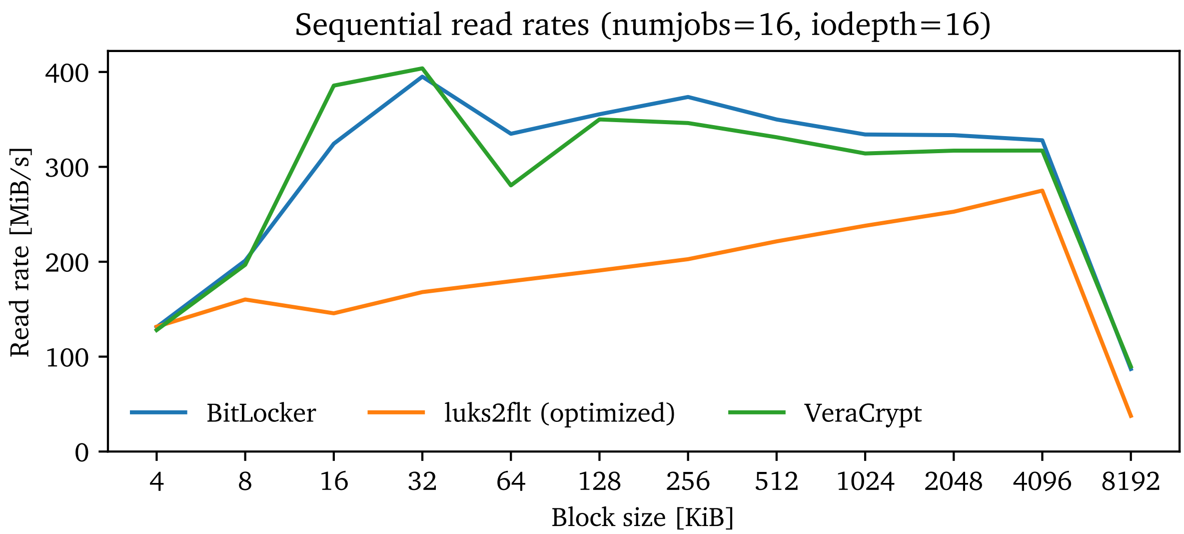

Figure 6.11: Sequential read rates with `numjobs=16` and `iodepth=16`. Each block size value is the average of three separate runs, with the value used for each run being the average read rate of all threads.

- BitLocker and VeraCrypt have their respective maximum read rates at 32 KiB, `luks2flt` at 4096.

- All drivers' read rates suddenly plummet down at 8192 KiB.

Figure 6.12 has the random-access read rates for multi-thread job execution with an `iodepth` of 16. BitLocker's read rate starts at 31 MiB/s and rises to its maximum of 241 MiB/s at 4096 KiB, although only slowly starting from a block size of 256 KiB. `luks2flt` has a rate of 31 MiB at 4 KiB, which then rises to a maximum of 222 MiB/s at 4096 KiB, with a dent at 1024 and 2048 KiB. VeraCrypt starts with a read rate of 19 MiB/s, which mostly continuously grows to its maximum of 202 MiB/s at 8192 KiB.

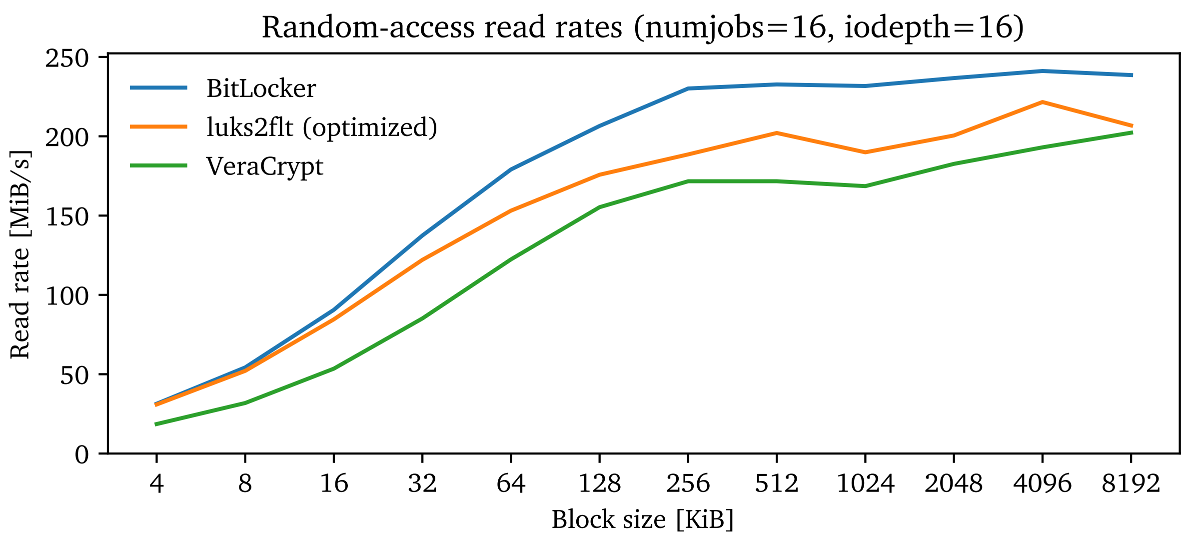

Figure 6.12: Random-access read rates with `numjobs=16` and `iodepth=16`. See Figure 6.11 for how we collected the data.

We observe the following:

- BitLocker always has the highest random-access read rate, with an up to 22% advantage over our driver (at 256 KiB) and a 70% advantage over VeraCrypt (at 8 KiB).
- `luks2flt` always has a higher read rate than VeraCrypt, with an up to 66% advantage (at 4 KiB).

Discussion Combining our observations from all measurements, we close with the following thoughts:

- In almost all cases, BitLocker has the highest read rate, regardless of the access pattern. As already detailed in Section 6.2, this is not really surprising, because it is not a fair comparison. Nonetheless, from time to time VeraCrypt's sequential read rate is quite close to BitLocker's, and our driver's multi-threaded random-access read rate can almost keep up with BitLocker's.
- Regrettably, the disabled optimization options led to a performance regression. Comparing the measurements with the same parameters (i.e. `iodepth=1` and `numjobs=1`), we can see that on real hardware `luks2flt` cannot compete with VeraCrypt as well as in the virtual machine. This is the case for both sequential and random access.
- Our driver's sequential read rate is consistently lower than that of the other two drivers. As discussed in the previous section, VeraCrypt probably has better sequential performance because of its read ahead caching. The maximum advantage the other drivers have over `luks2flt` mostly lies between 130% and 150%. However, when at least one of the `iodepth` or `numjobs` parameter has a value of 16, `luks2flt` can almost catch up with one of the other drivers for large block sizes.
- Even though `luks2flt` cannot compete at random access when only one I/O request is sent at a time (at least not for large block sizes), the situation looks differently for `iodepth=16` and/or `numjobs=16`: here, our driver always has a higher read rate than VeraCrypt, and even BitLocker has an at most 55% higher rate. BitLocker as a proprietary driver is out of reach for theorizing. However, we have a suspicion why VeraCrypts manages multiple requests at the same time not as well as `luks2flt`: every read request has to pass through three queues (Main, I/O, and Completion), and every queue only has one thread working on its requests. [16] showed that removing some queues in `dm-crypt` can lead to significant read rate improvements, and that may be the case for VeraCrypt, too.
- In the last section, we noted that VeraCrypt's sequential read rate peaked at 256 KiB blocks. We also highlighted that this was the size of the fragments that larger blocks get broken up into. In this section's measurements, this generally was also the case: VeraCrypt's sequential read rate either peaked at a block size of 256 or 512 KiB or did not improve by much for larger blocks.

6.3.2 Measuring Read Rates over Time

`fio` comes with an option to enable the logging of current read rates, `write_bw_log`. By default, one entry in the log file corresponds to one completed I/O request. This is not what we wanted for two reasons: firstly, this does not scale well with larger block sizes, as there are fewer I/O operations when using larger blocks; and secondly, we are not interested in single I/O requests, but the general trend over time (the logs with one line per completed request contain large read rate spikes which we do not care about). Fortunately, there is also an option to log the average bandwidth of the last N milliseconds, where the value of N is configurable: `log_avg_msec`. We found that 32 ms is a good value for this option, yielding data with reasonable high time resolution, but not too much spikes from the completion of individual I/O requests.

Note that all measurements in this section are values of single runs, and not averages of three runs of the same job like in the section before.

Results and Observations Figure 6.13 shows how BitLocker's, VeraCrypt's, and `luks2flt`'s read rates behave over time for a block size of 32 KiB. BitLocker's rate mostly stays between 367 and 375 MiB/s, with two short sections where it is between 375 and 383 MiB/s and some spikes, mostly downward. VeraCrypt's read rate generally falls between 226 and 290 MiB/s, but heavily fluctuates in that range. Except for the first 2.5 seconds, where it fluctuates between 149 and 169 MiB/s, `luks2flt`'s rate mostly stays between 161 and 173 MiB/s.

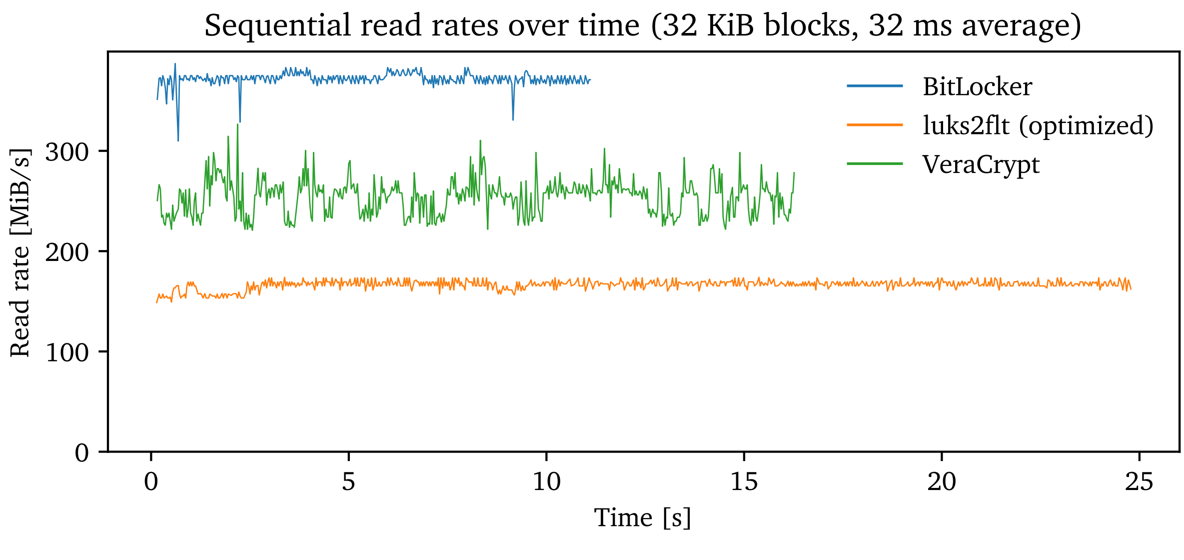

Figure 6.13: Sequential read rates over time. We used a block size of 32 KiB and set `fio`'s `log_avg_msec` parameter to 32 ms. The `iodepth` was set to a value of 1.

Figure 6.14 shows the distribution of the read rates from Figure 6.13 in a box plot. The median of BitLocker's read rate is 372 MiB/s, the lower quartile is 370 MiB/s, and the upper quartile is 375 MiB/s. The whiskers are at 363 and 382 MiB/s, and there are some outliers between 310 and 387 MiB/s. Our driver's median rate is 166 MiB/s, its lower quartile is 165 MiB/s, and its upper quartile is 169 MiB/s. The whiskers are at 160 and 174 MiB/s, and there are some outliers in the 149 to 159 MiB/s range. The median rate of VeraCrypt is 256 MiB/s, the lower quartile is 238 MiB/s, and the upper quartile is 262 MiB/s. The whiskers are at 221 and 298 MiB/s, and there are some outliers between 300 and 327 MiB/s.

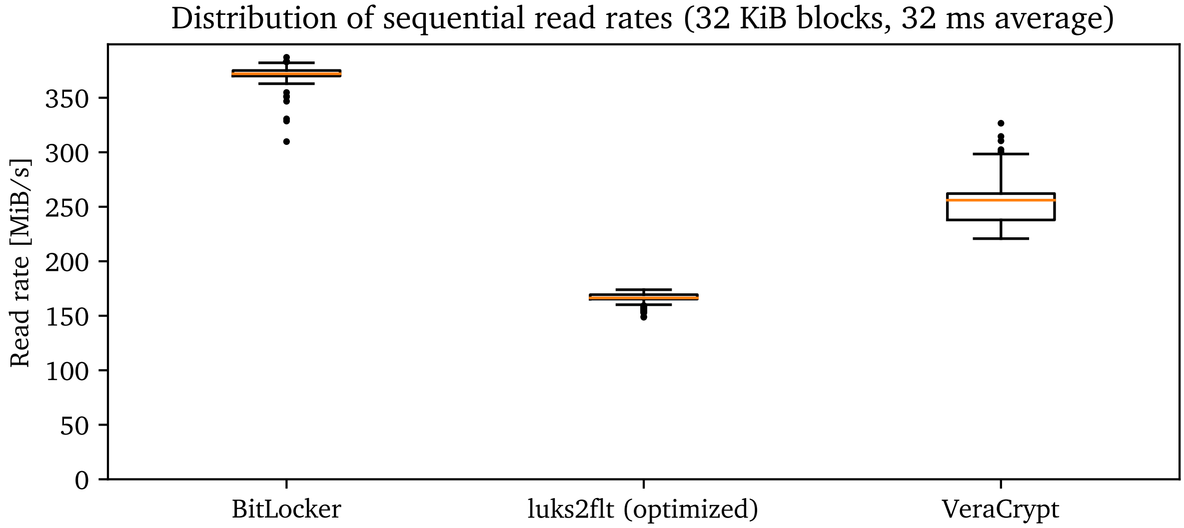

Figure 6.14: Distribution of sequential read rates over time. This is a different visualization of the data presented in Figure 6.13.

We observe the following:

– BitLocker has the highest sequential read rate, VeraCrypt comes second, and our driver comes third.

– `luks2flt` has the least variation in read rate values. Bitlocker comes close, but its outliers are further away from the mean. VeraCrypt's read rate has significantly more variation over time than the other drivers'.

Figure 6.15 shows the drivers' random-access read rates over time. BitLocker's read rates switches between lying in the 98 to 100 MiB/s range and the 94 to 96 MiB/s range, with some mostly downward spikes. `luks2flt` has a read rate that concentrates around either 64 MiB/s or 69 MiB/s, with some spikes in both directions. VeraCrypt's rate generally oscillates around 61 MiB/s, with exceptions, mostly at the beginning and the end of the run.

Figure 6.16 shows the distribution of the read rates from Figure 6.15. BitLocker's median read rate is 99 MiB/s, the lower quartile is 95 MiB/s, and the upper quartile is 99 MiB/s. The whiskers are at 91 and 101 MiB/s. The 69 to 113 MiB/s range contains all outliers, except for one at 35 MiB/s. The median of `luks2flt`'s read rate is 65 MiB/s, the lower quartile is 64 MiB/s, and the upper quartile is 69 MiB/s. The whiskers are at 57 and 75 MiB/s, and there are some outliers between 44 and 80 MiB/s. VeraCrypt has a median rate of 61 MiB/s, with the lower quartile being 60 MiB/s and the upper quartile being 61 MiB/s. The whiskers are at 57 and 75 MiB/s, and there are some outliers in the 32 to 71 MiB/s range.

We observe the following:

– BitLocker has the highest radnom-access read rate, our driver comes second, and VeraCrypt closely follows behind.

Figure 6.15: Random-access read rates over time. See Figure 6.13 for how we collected the data.

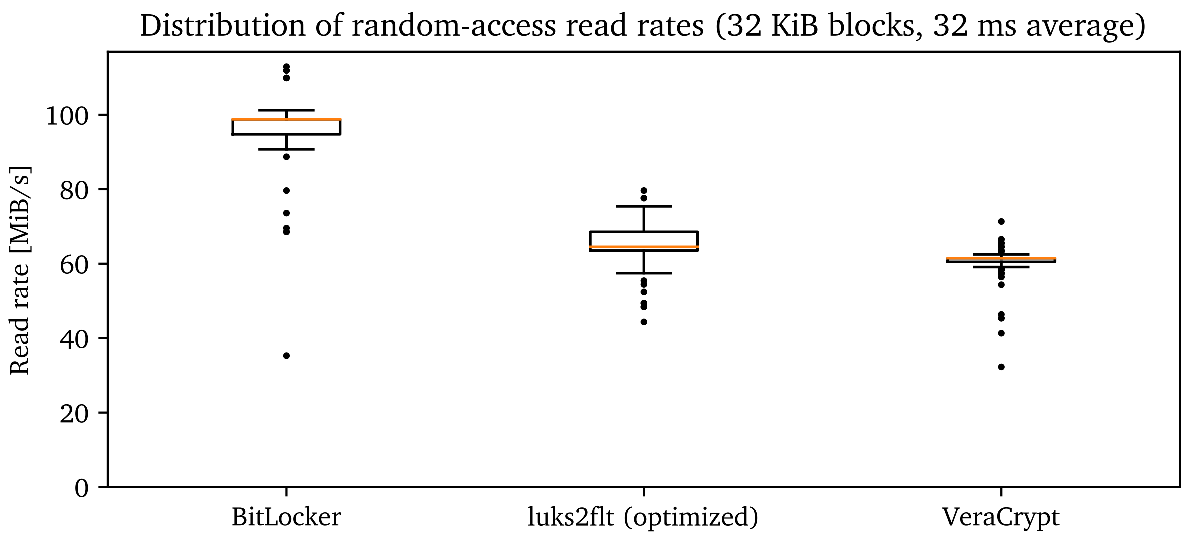

Figure 6.16: Distribution of random-access read rates over time. This is a different visualization of the data presented in Figure 6.15.

– VeraCrypt's read rate has the least variation, and BitLocker's and `luks2flt`'s both alternate between two small ranges. The rates of all three drivers contain occasional outliers.

We also looked at other block sizes than 32 KiB (all that we tested in the last section), and found this general trend: the larger the blocks, the more variations in the read rate (i.e. the boxes and whiskers stretched out farther, and there were also less outliers). We chose to display the graphs for 32 KiB blocks because they seemed a good compromise between being neither completely on one end of the block size range nor too jagged.

Discussion Combining our observations, we close with the following thoughts:

– The read rates in our measurements match what we saw in the last section (see Figure 6.5 and Figure 6.6).

- VeraCrypt had the most variation in read rate for sequential reading, and the least for random-access reading. Looking at our measurements over time for other block rates however we could not find a general trend; no driver regularly showed more or less variation in read rate than the others. However, these were all measurements of single runs. To find or rule out trends with more certainty, measurements from multiple runs should be combined.
- The trend that higher block sizes lead to more variation in the read rate probably occurs because the completion of a single I/O request has more impact on the short-time read rate the bigger the read block is.

7 Conclusion

The previous chapters described the design, implementation, and functionality of our driver as well as its I/O performance compared to other disk encryption technologies. Therefore, it is now time to evaluate whether we reached our goal as introduced in Chapter 1: implementing LUKS2 support on Windows, with reasonable effort and usable performance.

We made it possible to read the contents of LUKS2 volumes on Windows. Unfortunately, we had to disable write support because of it corrupting volumes. The goal of supporting LUKS2 on Windows can therefore be considered as only partially achieved. Then again, even more debugging than we already did to fix the corruption problem would have clashed with the resolution of reasonable effort.

The performance evaluation of driver also paints a mixed picture: while it falls behind both compared technologies for sequential reading, it overtakes VeraCrypt in random-access reading (except for single-threaded sequential requests). As this is only a prototype with no extra focus on optimized performance, compared to long-existing big projects, we view the goal of usable performance as successfully met. Furthermore, the version of `luk2flt` used in the virtual machine experiments had more compiler optimizations enabled and showed even better performance: its sequential read rate was comparable to VeraCrypt's for large block sizes. Thus it seems that further work in trying to gradually enable more compiler optimizations again, without introducing crashes, may improve the read rates on real hardware, too.

All in all, we consider our goal reached, but suggest the following improvements (some of them have already been mentioned in previous chapters):

- To find the cause of write support corrupting volumes, examining the filesystem metadata may help. We already presented our theory that this is the part of the volume that gets corrupted in Section 5.2.7 and justified how it matches with our observations.

- As explained above, fine-tuning the compiler optimization settings may further improve performance.

- Another way to improve the sequential read performance may be introducing a read ahead cache similar to VeraCrypt.

- The current way of attaching to all volumes at boot is not optimal. As noted in Section 5.2.3, using the Windows registry to configure which volumes to attach to may be desirable.

- Section 5.3 stated the possible problem that `luks2filterstart.exe` does not clear sensitive information from memory. An implementation meant for production usage should fix this issue.

We also identify the following possible further research topics:

- As mentioned in Section 5.1.2, comparing the process of implementing a driver with the same functionality using the KMDF would be interesting. Both from an architecture and performance point of view, this may yield useful insights.
- In Section 6.3.1, we formulated the theory that VeraCrypt's fragmenting of larger blocks into 256 KiB blocks may hinder its sequential read performance. We also theorized that its queuing is responsible for its worse performance random-access read performance when multiple requests at once come in. Both suspicions could be tested by appropriately modifying VeraCrypt and re-examining its performance
- The behaviour of read rates over time was only briefly analysed by us. It may be of interest to study this in more detail. Evaluating statistics on the latencies of the different drivers may be a fitting companion topic (the `fio` tool already provides such statistics, but we did not evaluate them).

A Notes On Online and Source Code References

All references to the source code of `cryptsetup` in Section 4.1.2 refer to the version of commit `ec946b17eb4aec48cbd4b3fc4b77324d4f5f8770`.

All references to the source code of VeraCrypt in Section 4.2.2 refer to VeraCrypt 1.24-Update9-Beta-21-08-15 (commit `b1323cabae32a2b0a9a96d7f228c1257c6188058`).

The VeraCrypt documentation [35] has been archived at `https://web.archive.org` and is also available at VeraCrypt's GitHub repository.

All referenced documentation by Microsoft is available/archived on GitHub:

- [11] at `https://github.com/MicrosoftDocs/sdk-api`;
- [12] at `https://github.com/MicrosoftDocs/windows-driver-docs-ddi`;
- [14], [41], and [42] at `https://github.com/MicrosoftDocs/windows-driver-docs`.

All cited online references have also been archived at `https://web.archive.org`.

archive `luks2flt`'s and `luks2filterstart.exe`'s source code and mention it here graphs-and-data repo

B Disassembly Code Listings

```
void DeleteKey(longlong param_1) {
    if (param_1 != 0) {
        FveDestroyCryptoKeyData(param_1);
        FveSecureZeroAndFlush(
            *(undefined **)(param_1 + 0x10),
            (ulonglong)*(uint *)(param_1 + 0x18)
        );
        ExFreePoolWithTag(param_1, 0x656e6f4e);
    }
    return;
}
```

Figure B.1: Disassembly of the `DeleteKey` function

```
uVar5 = FveInitPerfCounters(local_res8);
uVar12 = (ulonglong)uVar5;
psVar13 = param_2;
if (-1 < (int)uVar5) {
bVar3 = true;
lVar9 = 0x1c;
ppcVar11 = (code **)(param_1 + 0xe);
while (lVar9 != 0) {
    lVar9 = lVar9 + -1;
    *ppcVar11 = FveFilterSkip;
    ppcVar11 = ppcVar11 + 1;
}
param_1[0xe] = FveFilterCreate;
param_1[0x10] = FveFilterClose;
param_1[0x20] = FveFilterCleanup;
param_1[0x11] = FveFilterRundownRead;
param_1[0x12] = FveFilterRundownWrite;
param_1[0x1c] = FveFilterDeviceControl;
param_1[0x24] = FveFilterPower;
param_1[0x29] = FUN_1c0086280;
*(code **)(param_1[6] + 8) = FUN_1c0087510;
param_1[0xd] = FUN_1c00a6f40;
IoRegisterBootDriverReinitialization(param_1, ReInitialize, 0);
puVar10 = (undefined8 *)&DAT_00000050;
psVar13 = (short *)&DAT_30455646;
uVar6 = ExAllocatePoolWithTag(0x200);
local_res8[9] = uVar6;
if ((void *)local_res8[9] == (void *)0x0) {
    uVar12 = 0xc000009a;
```

Figure B.2: Disassembly of parts of the `DriverEntry` function

```
NTSTATUS FveXtsAesDecrypt(
    BCRYPT_KEY_HANDLE *param_1,
    undefined8 param_2,
    undefined8 param_3,
    PUCHAR param_4,
    ulonglong param_5,
    PUCHAR param_6
) {
    ulonglong uVar1;
    NTSTATUS NVar2;
    ULONG local_res10[2];
    undefined8 local_res18;

    uVar1 = param_5;
    local_res10[0] = 0;
    if (param_5 < 0x100000000) {
        local_res18 = param_3;
        if (((DAT_1c002f3b8 != 0) && (DAT_1c002f3b0 != 0)) &&
            (*(code **)(DAT_1c002f3b0 + 0x48) != (code *)0x0))
        {
            (**(code **)(DAT_1c002f3b0 + 0x48))();
        }
        NVar2 = BCryptDecrypt(
            *param_1,
            param_4,
            (ULONG)uVar1,
            (void *)0x0,
            (PUCHAR)&local_res18,
            8,
            param_6,
            (ULONG)uVar1,
            local_res10,
            0
        );
        if (((DAT_1c002f3b8 != 0) && (DAT_1c002f3b0 != 0)) &&
            (*(code **)(DAT_1c002f3b0 + 0x48) != (code *)0x0))
        {
            (**(code **)(DAT_1c002f3b0 + 0x48))();
        }
    }
    else {
        NVar2 = -0x3fffff6b;
    }
    return NVar2;
}
```

Figure B.3: Disassembly of the `FveXtsAesDecrypt` function

C Acknowledgements

I want to thank the following people for their support:

- Richard Sailer for his continuous feedback, ideas, and recommendations.
- Jonas Bühler for the suggestion of making LUKS2 on Windows the topic of my thesis.
- Paul Hennig for the Oknolynx name suggestion and his feedback on the thesis.
- My friends and family who gave me feedback on the thesis.

Bibliography

[1] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright, "A nine year study of file system and storage benchmarking," *ACM Trans. Storage*, vol. 4, no. 2, pp. 5:1–5:56, 2008.

[2] M. Broz, *LUKS2 On-Disk Format Specification Version 1.0.0*, 2018, visited on 2021-07-31. [Online]. Available: https://gitlab.com/cryptsetup/LUKS2-docs/-/raw/861197a9de9cba9cc3231ad15da858c9f88b0252/luks2_doc_wip.pdf

[3] *Frequently Asked Questions Cryptsetup/LUKS*, 2020, visited on 2021-08-10. [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup/-/wikis/FrequentlyAskedQuestions?version_id=6297db166f8ae73c6a616ba874a12f5d43d37fd9

[4] C. Fruhwirth, "New methods in hard disk encryption," Ph.D. dissertation, Vienna University of Technology, 2005.

[5] C. Fruwirth, *LUKS1 On-Disk Format Specification Version 1.2.3*, 2018, visited on 2021-07-31. [Online]. Available: https://mirrors.edge.kernel.org/pub/linux/utils/cryptsetup/LUKS_docs/on-disk-format.pdf

[6] N. Ferguson, B. Schneier, and T. Kohno, *Cryptography Engineering – Design Principles and Practical Applications*. Wiley, 2010.

[7] M. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. Bassham, E. Roback, and J. Dray, "Advanced Encryption Standard (AES)," National Institute of Standards and Technology, Tech. Rep., 2001.

[8] *IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices*, IEEE Std. 1619-2018 (Revision of IEEE Std. 1619-2007), 2019.

[9] *dm-crypt: Linux device-mapper crypto target*, 2020, visited on 2021-08-06. [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMCrypt?version_id=ee336a2c1c7ce51d1b09c10472bc777fa1aa18cd

[10] P. Yosifovich, D. A. Solomon, and A. Ionescu, *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more*. Microsoft Press, 2017.

[11] *Programming reference for the Win32 API*, visited on 2021-08-19. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api

[12] *API reference docs for Windows Driver Kit (WDK)*, visited on 2021-08-19. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi

[13] M. E. Russinovich and A. Margosis, *Troubleshooting with the Windows Sysinternals Tools*. Microsoft Press, 2016.

[14] *Documentation for the Windows Driver Frameworks (WDF)*, visited on 2021-08-25. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/wdf

[15] A. S. Tanenbaum and H. Bos, *Modern operating systems*. Pearson, 2015.

[16] I. Korchagin, "Speeding up Linux disk encryption," Cloudflare, Tech. Rep., 2020, visited on 2021-08-23. [Online]. Available: https://blog.cloudflare.com/speeding-up-linux-disk-encryption

[17] A. Barker, S. Sample, Y. Gupta, A. McTaggart, E. L. Miller, and D. D. E. Long, "Artifice: A deniable steganographic file system," in *Proceedings of the 9th USENIX Workshop on Free and Open Communications on the Internet (FOCI '19)*, 2019.

[18] C. P. Wright, J. Dave, and E. Zadok, "Cryptographic file systems performance: What you don't know can hurt you," in *Second IEEE International Security in Storage Workshop*. IEEE, 2003, pp. 47–47.

[19] *cryptsetup(8) – Linux manual page*, visited on 2021-08-27. [Online]. Available: https://man7.org/linux/man-pages/man8/cryptsetup.8.html

[20] S. G. Lewis and T. Palumbo, "BitLocker full-disk encryption: Four years later," in *Proceedings of the 2018 ACM SIGUCCS Annual Conference*, 2018, pp. 147–150.

[21] M. Tsegaye and R. Foss, "A comparison of the Linux and Windows device driver architectures," *ACM SIGOPS Oper. Syst. Rev.*, vol. 38, no. 2, pp. 8–33, 2004.

[22] R. Battistoni, A. D. Biagio, R. D. Pietro, M. Formica, and L. V. Mancini, "A live digital forensic system for Windows networks," in *Proceedings of The IFIP TC-11 23rd International Information Security Conference, IFIP 20th World Computer Congress, IFIP SEC 2008, September 7-10, 2008, Milano, Italy*, ser. IFIP, S. Jajodia, P. Samarati, and S. Cimato, Eds., vol. 278. Springer, 2008, pp. 653–667. [Online]. Available: https://doi.org/10.1007/978-0-387-09699-5_42

[23] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006*, Y. Berbers and W. Zwaenepoel, Eds. ACM, 2006, pp. 73–85.

[24] T. Ni, Z. Yin, Q. Wei, and Q. Wang, "High-coverage security testing for Windows kernel drivers," in *2012 Fourth International Conference on Multimedia Information Networking and Security*, 2012, pp. 905–908.

[25] A. Riska and E. Riedel, "Disk drive level workload characterization," in *Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USA, May 30 - June 3, 2006*, A. Adya and E. M. Nahum, Eds. USENIX, 2006, pp. 97–102. [Online]. Available: http://www.usenix.org/events/usenix06/tech/riska.html

[26] ——, "Long-range dependence at the disk drive level," in *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA*. IEEE Computer Society, 2006, pp. 41–50.

[27] A. W. Leung, S. Pasupathy, G. R. Goodson, and E. L. Miller, "Measurement and analysis of large-scale network file system workloads," in *2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings*, R. Isaacs and Y. Zhou, Eds. USENIX Association, 2008, pp. 213–226. [Online]. Available: http://www.usenix.org/events/usenix08/tech/full_papers/leung/leung.pdf

[28] A. Riska and E. Riedel, "Evaluation of disk-level workloads at different time-scales," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*. IEEE Computer Society, 2009, pp. 158–167.

[29] P. Sehgal, V. Tarasov, and E. Zadok, "Evaluating performance and energy in file system server workloads," in *FAST*, 2010, pp. 253–266.

[30] *The Linux Kernel 5.13 documentation*, visited on 2021-08-26. [Online]. Available: https://www.kernel.org/doc/html/v5.13

[31] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux device drivers*, 3rd ed. O'Reilly, 2005.

[32] *keyrings(7) – Linux manual page*, visited on 2021-08-27. [Online]. Available: https://man7.org/linux/man-pages/man7/keyrings.7.html

[33] M. Videau and J. Bédrune, "VeraCrypt 1.18 security assessment," Quarkslab, Tech. Rep., 2016.

[34] A. Poller, H. Evkan, N. Lahr, R. Niederhagen, R. Petri, P. Roskosch, and M. Tröger, "Security evaluation of VeraCrypt," Fraunhofer Institute for Secure Information Technology, Tech. Rep., 2020.

[35] *VeraCrypt Documentation*, visited on 2021-09-04. [Online]. Available: https://veracrypt.fr/en/Documentation.html

[36] J. D. Kornblum, "Implementing BitLocker drive encryption for forensic analysis," *Digital Investigation*, vol. 5, no. 3-4, pp. 75–84, 2009.

[37] R. Sosnowski, *Bitlocker: AES-XTS (new encryption type)*, 2016, visited on 2021-09-01. [Online]. Available: https://docs.microsoft.com/en-gb/archive/blogs/dubaisec/bitlocker-aes-xts-new-encryption-type

[38] S. Türpe, A. Poller, J. Steffan, J.-P. Stotz, and J. Trukenmüller, "Attacking the BitLocker boot process," in *International Conference on Trusted Computing*. Springer, 2009, pp. 183–196.

[39] C. Tan, L. Zhang, and L. Bao, "A deep exploration of BitLocker encryption and security analysis," in *2020 IEEE 20th International Conference on Communication Technology (ICCT)*. IEEE, 2020, pp. 1070–1074.

[40] Trusted Computing Group, *Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.59*, 2019.

[41] *File systems driver design guide*, visited on 2021-09-05. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs

[42] *Kernel-Mode Driver Architecture Design Guide*, visited on 2021-09-07. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel

[43] J. K. Rott, *Intel Advanced Encryption Standard Instructions (AES-NI)*, 2012, visited on 2021-09-11. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-advanced-encryption-standard-instructions-aes-ni.html

[44] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold boot attacks on encryption keys," in *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, P. C. van Oorschot, Ed. USENIX Association, 2008, pp. 45–60. [Online]. Available: http://www.usenix.org/events/sec08/tech/full_papers/halderman/halderman.pdf

[45] L. Guan, J. Lin, Z. Ma, B. Luo, L. Xia, and J. Jing, "Copker: A cryptographic engine against cold-boot attacks," *IEEE Trans. Dependable Secur. Comput.*, vol. 15, no. 5, pp. 742–754, 2018.

[46] *fio - Flexible I/O tester rev. 3.27*, visited on 2021-09-18. [Online]. Available: https://fio.readthedocs.io/en/latest/fio_doc.html

List of Figures

List of Tables