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

Enabling read-only access to LUKS2 partitions on Windows and comparing the performance to other disk encryption technologies

Max Ihlenfeldt

Universität Augsburg
Lehrstuhl für Organic Computing
Bachelorarbeit im Studiengang Informatik

Abstract goes here

Contents

1 Introduction

Explain use case etc.

Note that in this thesis the terms *disk*, *drive*, *volume* and *partition* are used somewhat loosely and probably mean roughly the same.

2 Background

todo?

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 [1] 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 together with the Linux dm-crypt subsystem, but that is not mandatory[1].

The differences between the original LUKS and LUKS2 are minor. According to [1], 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 chapter 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 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).

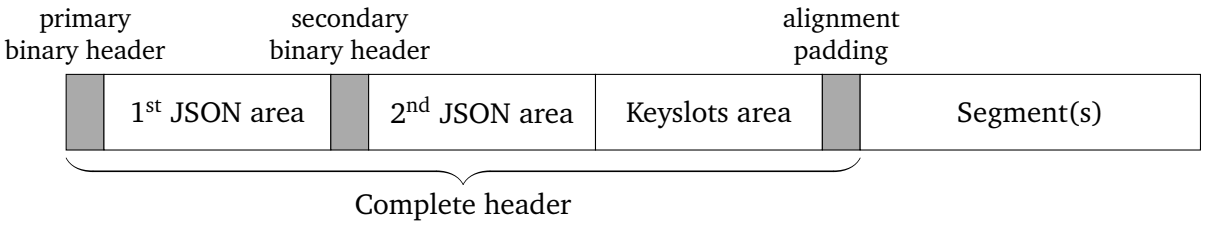

Figure 2.1: LUKS2 on-disk format (modified after [1]). The complete header consists of three areas: a binary header of exactly one 4096-byte sector, JSON metadata, and the binary keyslots data. A *keyslot* is an "encrypted area on disk that contains a key" [1]. For redundancy, the binary header and the JSON metadata are stored twice. After that follow one or areas containing encrypted user data. The specification calls these areas *segments*.

2.1.2 On-Disk Format

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 [1] 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 [1], Table 3). The default PBKDF used by LUKS2 is `argon2i` [2]. In the following paragraphs, it will become apparent what PBKDFs are used for in LUKS2[5].

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

[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`
[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.

```
#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 [1]. 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.

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 [3]. 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 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. 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.

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

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.

[3] presents two templates for storing keys, TKS1 and TKS2. The difference is

0000	4C	55	4B	53	BA	BE	00	02	00	00	00	00	00	00	40	00	LUKS°¾.....@.
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.

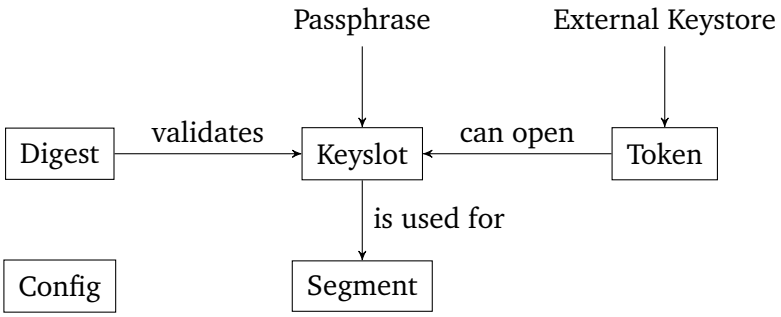

Figure 2.4: LUKS2 object schema from [1]. The most important objects are the following: *keyslots*, which describe the details of how cryptographic keys are stored and encrypted; *digests*, which can be used to verify that one has successfully extracted a key from a keyslot; and *segments*, which describe 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.

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.

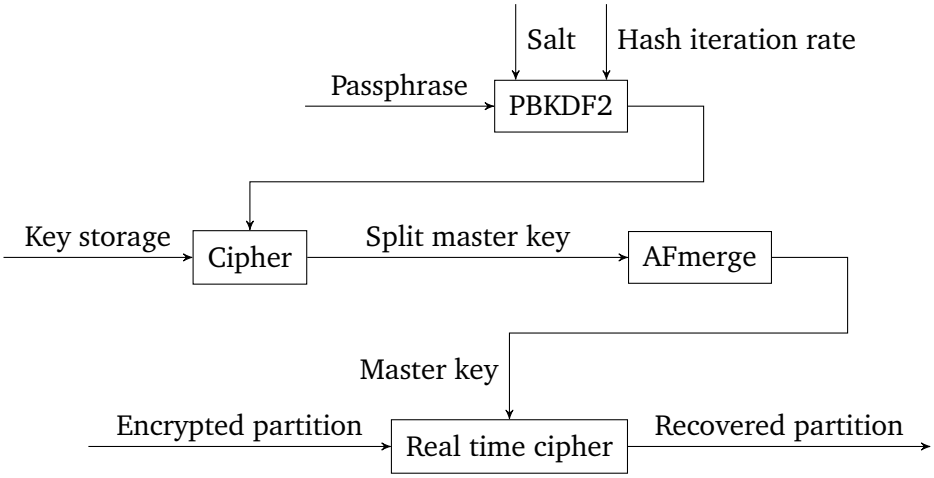

Figure 2.5: TKS2 scheme (modified after [3]).

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.

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 [1]). The `dm-crypt` algorithm notation is described in chapter 4.1.1. See [5] for the CBC mode and the Serpent and Twofish ciphers, and [3] for the ESSIV IV mode.

The AES encryption algorithm is a block cipher for processing 128 bit data blocks [6]. 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.

```rust
fn decrypt_keyslot(
    password: &[u8], keyslot: &LuksKeyslot, json: &LuksJson, /* ... */
) -> Result<Vec<u8>, LuksError> {
    let mut k = vec![
        0; keyslot.key_size() as usize * keyslot.af.stripes() as usize
    ];
    // read keyslot data from disk into k...

    let mut pw_hash = vec![0; area.key_size() as usize];
    match keyslot.kdf() {
        // hash into pw_hash using pbkdf2, argon2i, or argon2id...
    }

    // decrypt keyslot area using the password hash as key
    match area.key_size() {
        32 => {
            let key1 = Aes128::new_varkey(&pw_hash[..16]).unwrap();
            let key2 = Aes128::new_varkey(&pw_hash[16..]).unwrap();
            let xts = Xts128::<Aes128>::new(key1, key2);
            xts.decrypt_area(&mut k, sector_size, 0, get_tweak_default);
        },
        // 64 byte key uses AES256 instead...
    }

    // merge and hash master key
    let master_key = af::merge(
        &k, keyslot.key_size() as usize, af.stripes() as usize
    );
    let digest_actual = base64::decode(json.digests[&0].digest())?;
    let mut digest_computed = vec![0; digest_actual.len()];
    let salt = base64::decode(json.digests[&0].salt())?;
    pbkdf2::pbkdf2::<Hmac<Sha256>>(
        &master_key, &salt, json.digests[&0].iterations(), &mut digest_computed
    );

    // compare digests
    if digest_computed == digest_actual {
        Ok(master_key)
    } else {
        Err(LuksError::InvalidPassword)
    }
}
```

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.

To encrypt data longer than one block, a *block cipher mode* is needed [5]. These are encryption functions that build on a existing block cipher. When using aes-xts-plain64, LUKS2 uses the XTS block cipher mode. The defining IEEE standard [7] 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. 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 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 [4].

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" [8]. 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 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 [5].

All this theory may sound complicated, but in chapter 5.2.4 we will see that the practical usage of cryptography in our driver is quite simple[6].

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.

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.

[6] 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.

The definitive reference for all detailed information on the inner workings of Windows is [9]. The online documentation of the Win32 API [10] and the Windows Driver Kit (WDK) [11] 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) [9].

There are also some invaluable tools to directly take a look at OS internals. Most of them are from the Sysinternals[7] Suite, others are included with Windows. See table 1-4 in [9] 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 [9]. 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 [10]. 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 chapter 2.2.2 for more) [9].

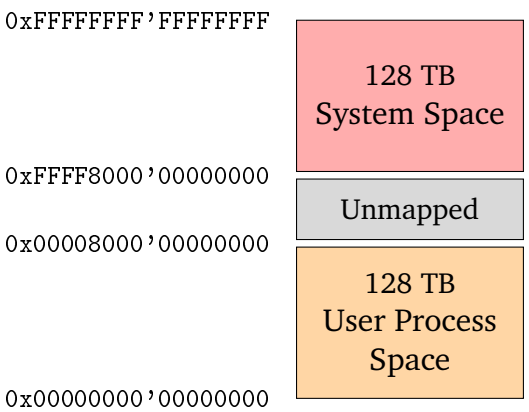

Figure 2.8: Address space layout (not to scale) for 64-bit Windows 10 (modified after [9]). Each process has its own copy of the user process space.
(Sven Peter has a nice quote on this topic on his blog: “One of the kernel’s tasks is to lie to each application running in userland and to tell them that they’re the only one in the address space.”).

Figure 2.9 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,

[7] The executables can be downloaded from `https://live.sysinternals.com` and are accompanied by documentation in [12].

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 [9].

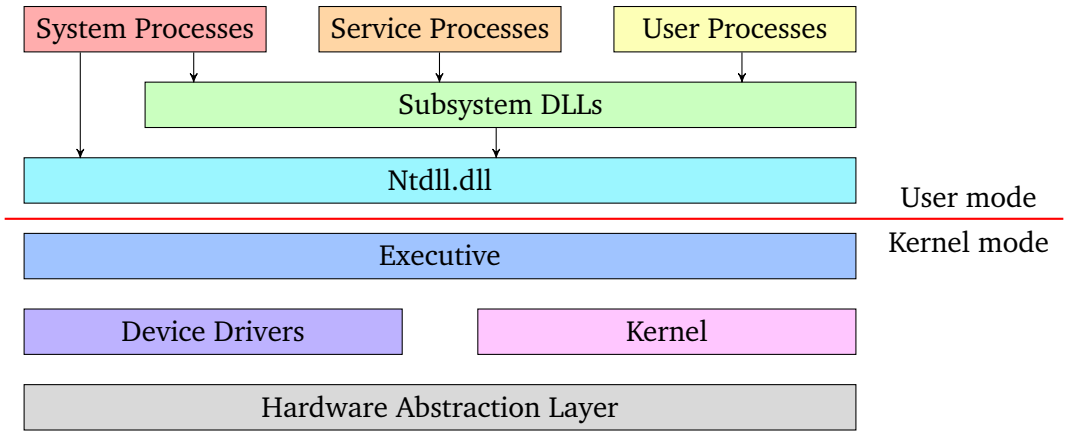

Figure 2.9: Simplified Windows architecture (modified after [9]). User processes are regular applications. The service process are responsible for hosting Windows services, such as the printer spooler or DNS client. System processes are fixed processes that are not started by the Service Control Manager. 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.

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[8]. 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 been completed, the OS orders the processor to switch back into user mode and hands control back to the application [9].

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[9] [9].

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 [9].

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

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 [9].

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* [9].

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 [9]:

- 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 [9] 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[10]. 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 [9]:

- 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.
- 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.

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

– 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) [13]. This moves Windows in the direction of a *microkernel* system. [14] 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 [9]. The functionalities of the HAL that are intended to be used by drivers (i.e. the ones that are documented in the WDK) include [11]:

– 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[11] and calling the function it points to;

– utilities for direct memory access (DMA) (see also footnote 15). There exist old methods that use the `Hal` prefix, but they have been deprecated in favour of other 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.

[11] Using IRPs, which are described in chapter 2.2.3.

2.2.2 Windows Kernel Drivers and the Windows Driver Model

Windows kernel drivers are modules that can be loaded and run kernel-mode. They are normally found in the form of files with the `.sys` extension. Drivers can be categorized as follows [9]:

- **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)* [9]. We will focus on WDM in this section, because that is the framework our driver uses (see chapter 5.2). But there also exist other models and frameworks for Windows driver development, such as the Windows Driver Frameworks (WDF) [13]. These try to simplify things and consist of the Kernel-Mode Driver Framework (KMDF) and the User-Mode Driver Framework (UMDF) [9]. We will explore some of the differences between them and WDM in chapter 5.1.

WDM distinguishes between three types of drivers [9]:

- **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." [9] 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 [9]. See Figure 2.10 for an example of how such a device stack can look.

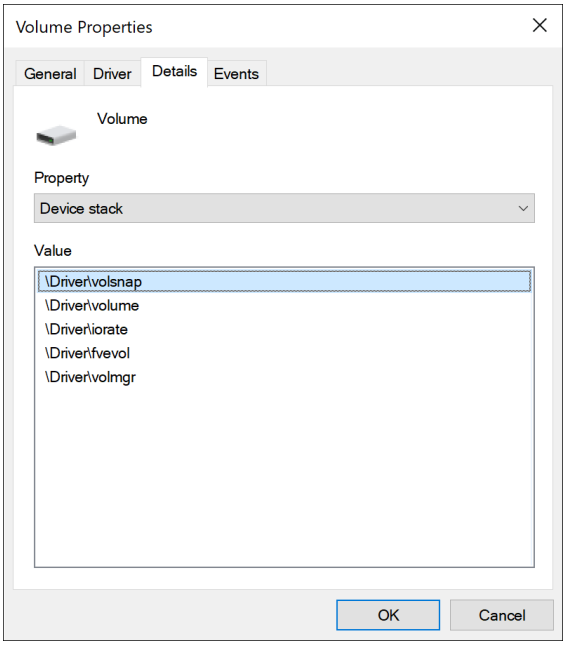

Figure 2.10: Example of a device stack for a storage volume (Screenshot from device properties in Windows Device Manager). The official descriptions of the drivers listed in this example are, from top to bottom: "Volume Shadow Copy driver", "Volume driver", "I/O rate control filter", "BitLocker Drive 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 [9].

For all the drivers of a device, the PnP manager creates objects that are used for managing and representing the relationships between the drivers. These objects form 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 [9]:

- 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 places 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." [9]

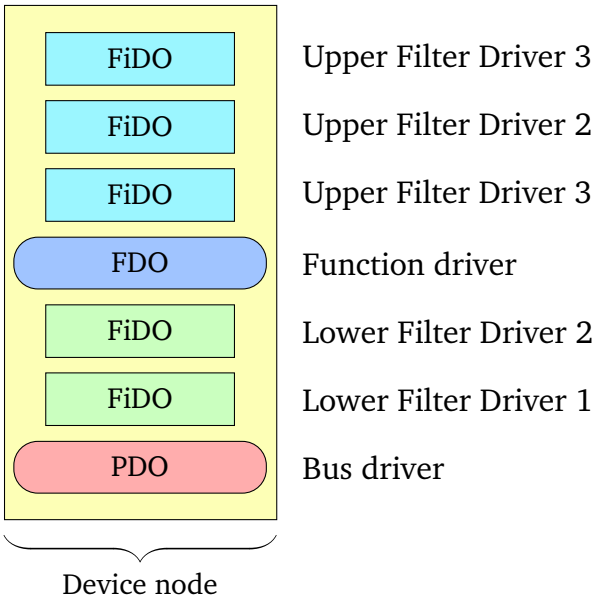

Figure 2.11: A device node and its device stack (modified after [9]). See Figure 2.10 for a concrete example of a device stack.

As already hinted at in the description of Figure 2.10, the registry is responsible for storing the information that governs driver loading. Both which drivers are loaded and the order they are loaded in are 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. mention that chapter 5.2 explains adding the require

The keys from Table 2.2 each have different roles [9]:

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 [9]. 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[12] 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[13]. 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 [9]:

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

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

[13] 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.

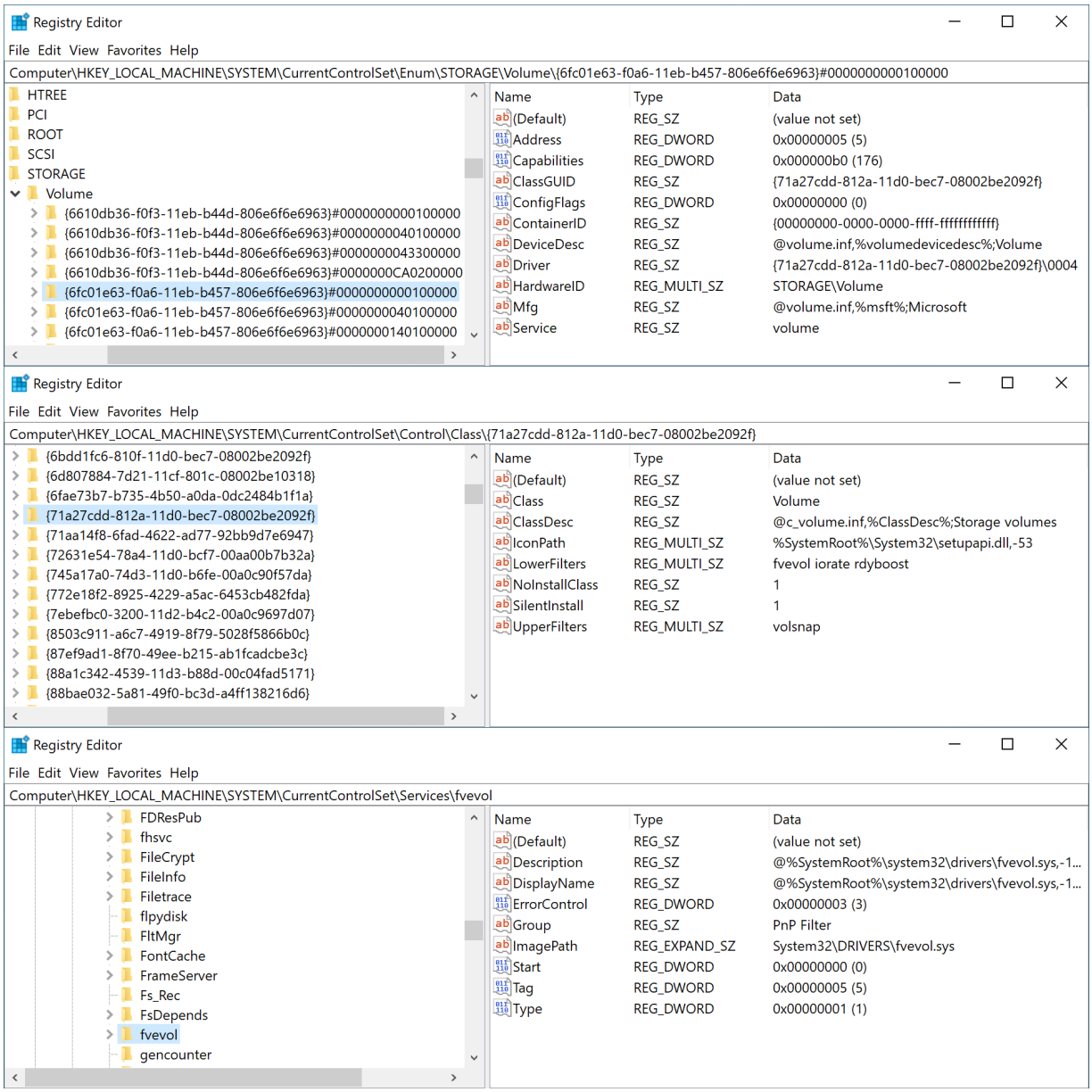

Figure 2.12: Example contents of a Hardware, Class, and Software subkey (Screenshots from the Windows Registry Editor). These are all relevant for the device whose device stack is shown in Figure 2.10. As already partly discussed there, the configured values match the final device stack: the `Service` 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 [9] 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 [9].

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 [9]:

- 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*[14].

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 [9]:

- 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.

[14] 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 [9]:

– **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[15].

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[16]. 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 [9].

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 [9] 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.

[15] 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." [9]

[16] 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 [10] for information on how to use them.

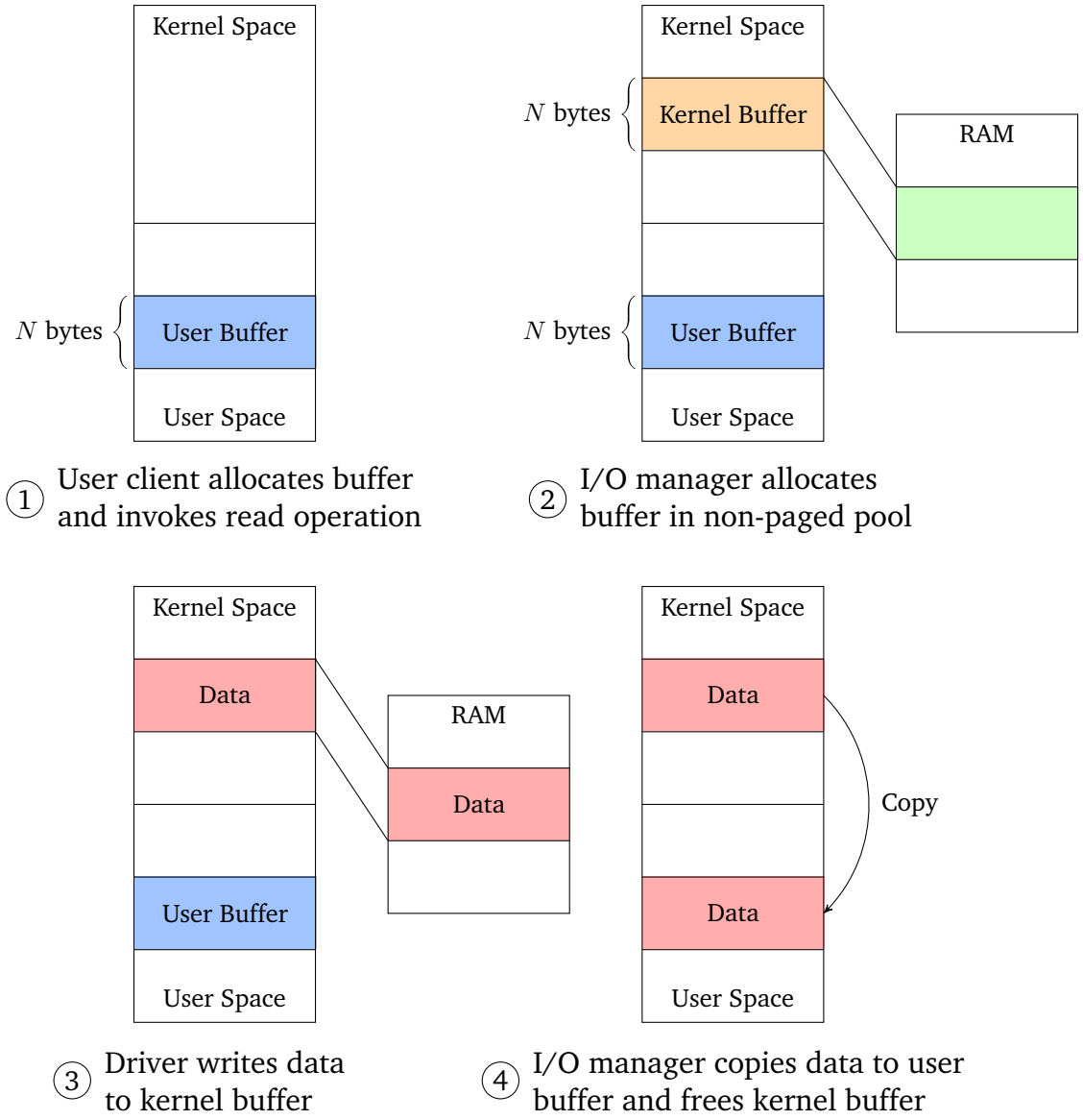

Figure 2.13: Buffered I/O for a read operation (modified after [9]). For write operations, the copying is done before the driver is called to handle the request.

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[17]. This abstraction allows the I/O manager to only care about files, leaving the translation of file commands to device-specific operations to drivers [9].

Secondly, Windows' internal I/O system is packet-driven. Almost all I/O requests[18] 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[19], 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

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

[18] The exception is so-called *Fast I/O*, which works differently [9]. It is not supported by our driver and will therefore not be covered.

[19] The PnP or power manager also sometimes allocate IRPs [9]. For simplicity, we will write "the I/O manager" instead of "the I/O, PnP, or power manager" in the following paragraphs.

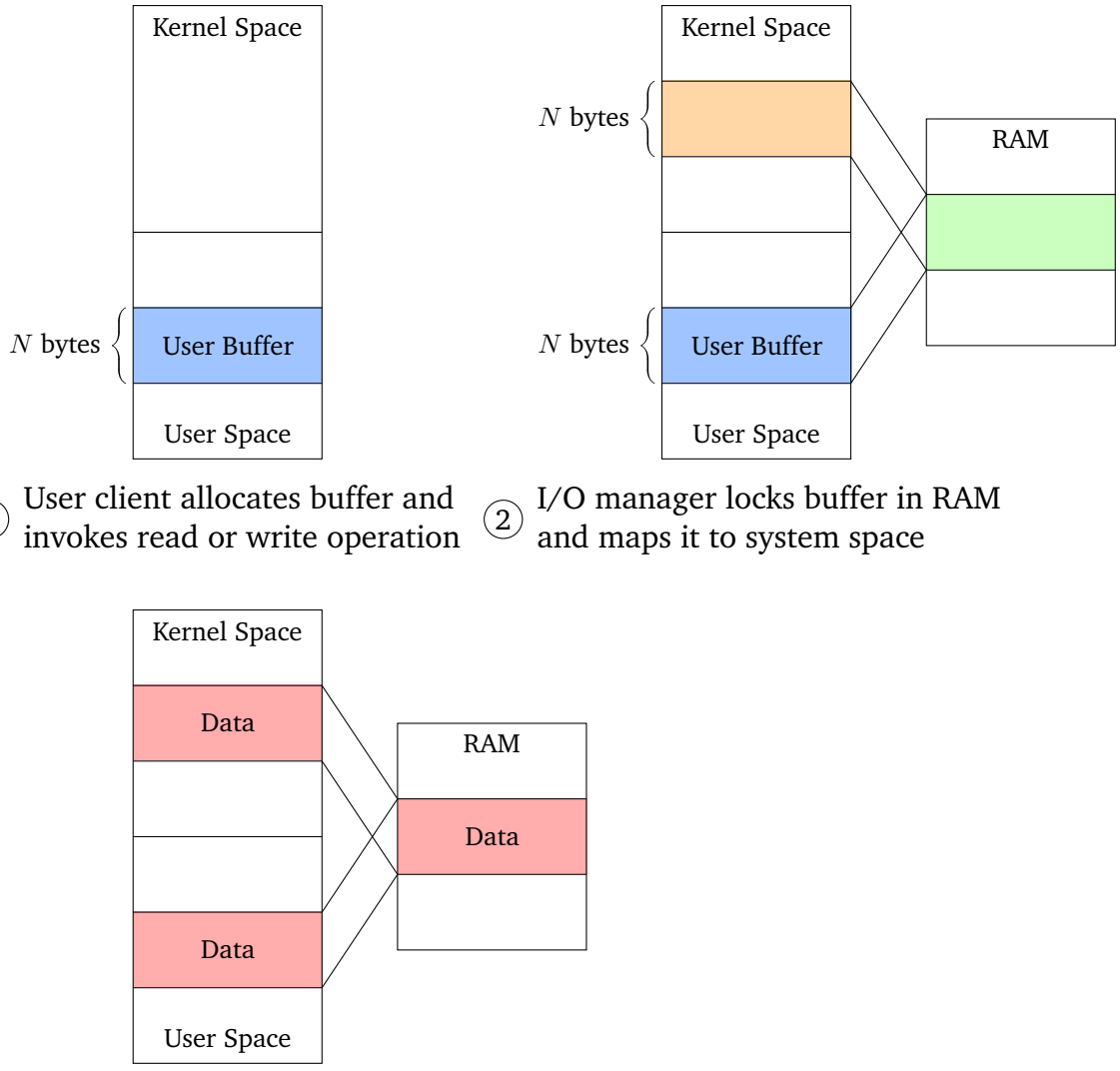

Figure 2.14: Direct I/O for a read or write operation (modified after [9]).

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 [9].

An IRP holds a lot of information. For this thesis, it suffices to know about the following fields[20] [9]:

- 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

[20] See [9] and its documentation in [11] for more details.

the number of drivers this IRP will pass through[21]. 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 [9].

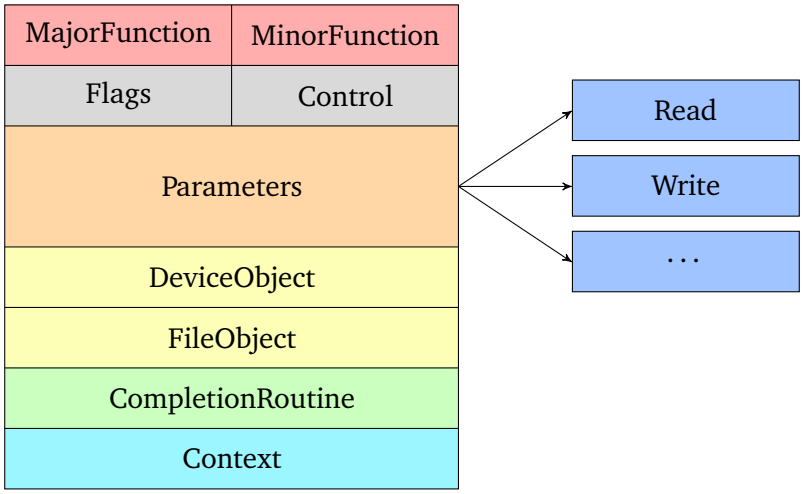

Figure 2.15: Layout of the `IO_STACK_LOCATION` structure (modified after [9] using information from [11]). The `MajorFunction` field has one of 28 possible values, denoting the request type, e.g. `IRP_MJ_READ` or `IRP_MJ_WRITE`. `MinorFunction` is used to further specify the type for some of the major 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." 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. 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. This initialization can be done in two ways [9]:

- 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 mo-

[21] 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 [9]. We will discuss the filter manager in chapter 5.1.

mentarily) counts as a change, and a driver must copy the stack location instead of skipping it in that case [11].

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 [9]. Completion routines may run at an IRQL equal to `DISPATCH_LEVEL` [11].

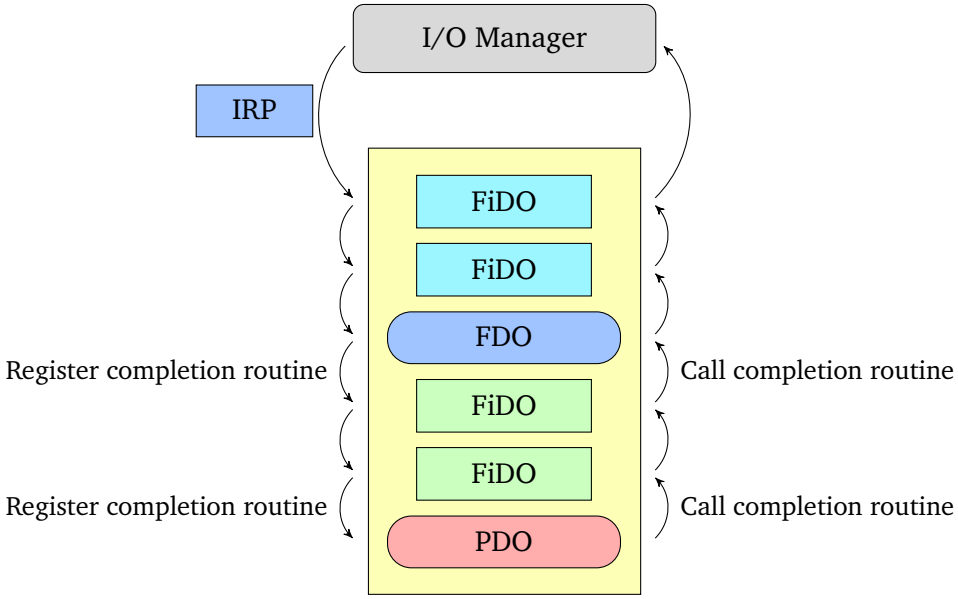

Figure 2.16: Example IRP flow (modified after [9]). The I/O manager sends an IRP down the device stack of a devnode. It first passes through two upper filter driverss and the function driver, with the latter registering a completion routine. It then passes through two lower filter drivers, with the second of those also registering a completion routing. 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 [9]. It is also responsible for calling the completion routines [11].

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. [9] also uses

WinDbg for exploring the internals of Windows, including a lot of screenshots and explaining many useful commands[22]. You can also find screenshots of WinDbg from the process of writi debugging our driver in chapter 5.

There are different ways to use WinDbg [9]:

- 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[23]. 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[24].

screenshot

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

[23] 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.

[24] 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.

[15] 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. optimizations also mentioned in chapter 4.1.1, remove or rewrite there?

An implementation of a kernel driver for purposes that are similar to ours, but for Linux instead of Windows, is presented in [16]. 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 chapter 4.1.1.

A comparison and performance study of various file encryption systems can be found in [17]. 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 [18]), but not implemented in our driver.

[19] 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's built-in BitLocker (described in chapter 4.3), the authors describe the results of its deployment and the lessons learned during the process.

[20] 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 [21]. 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. [22] 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 [23]. 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.

Relevant for chapter 6 of this thesis, [24] introduces criteria for good benchmarks of file systems and storage. The authors explicitly mention encrypted storage and have useful suggestions for benchmarking these systems. These include comparing the encrypted file systems performance to a modified version, which works exactly the same except that no cryptographic actions are performed (sometimes called the null cipher).

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] better selection? present, describe, and analyse different disk drive workloads collected on real hardware.

search for workload descriptions for more specific use cases? e.g. web browsing, gaming, ...

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 chapter 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 [15].

4.1.1 The Linux Kernel Device Mapper

The reference implementation uses the Linux kernel's Device Mapper [8]. 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 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 [16].

[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 [8]). 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 [8]). `cipher` specifies the encryption cipher that is used, in the following format: `cipher-chainmode-ivmode[:ivopts]`. See chapter 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 [8]. See chapter 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. [15] 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 [8].

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 [18]:

- LUKS and LUKS2 (see chapter 2.1);
- TrueCrypt and VeraCrypt (see chapter 4.2);
- BitLocker (see chapter 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 chapter 2.1) [18].

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 [18] for more detailed information. See Appendix A for information on which version of `cryptsetup` the following paragraphs refer to.

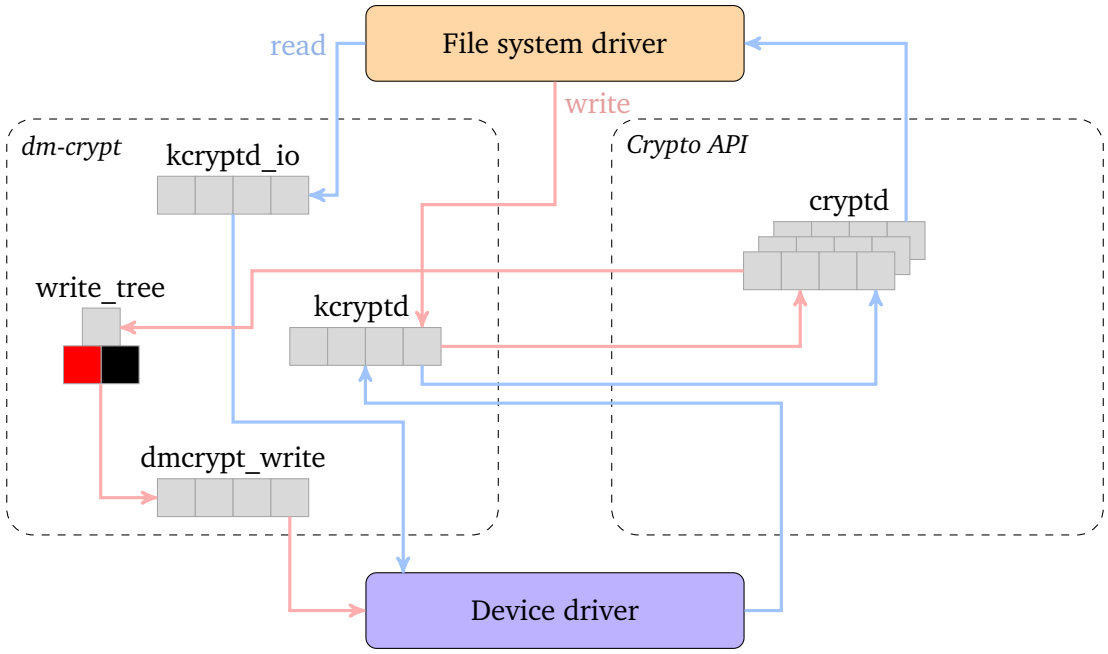

Figure 4.3: `dm-crypt` and Linux kernel crypto IO traverse path (modified after [15]). A write request from the file system driver first moves into the `kcryptd` workqueue. This queue takes care of doing the encryption at some later, more convenient point in time (see [30] for more details). The Linux 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 [18].

[4] Recall from chapter 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.

VeraCrypt 1.18 Security Assessment, BSI Security Evaluation of VeraCrypt

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] [33]. 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 [33].

VeraCrypt supports different block ciphers, including AES, Serpent, and Twofish (see [5] 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 [33].

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 [33].

Using an unlocked partition is practically equivalent to what is described in chapter 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

[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. [33] mentions a volume that has been securely wiped as a plausible example for why a volume would only contain random data.

components are not used when a VeraCrypt volume is unlocked and mounted in portable (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].

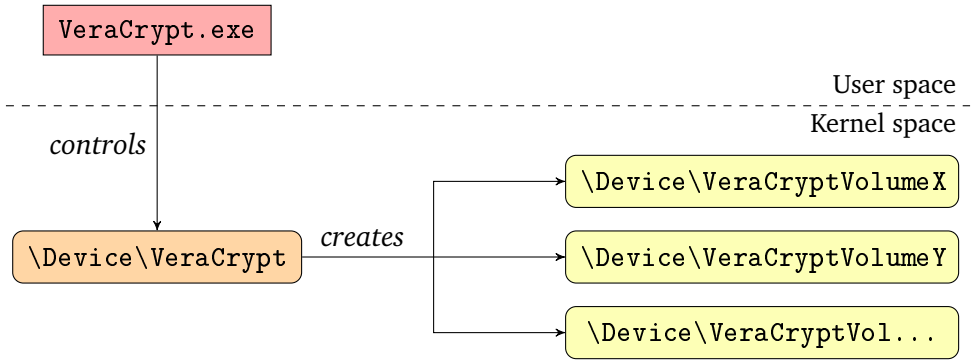

Figure 4.4: VeraCrypt Architecture. The user space application sends commands to the root device, `\Device\VeraCrypt`, via custom IOCTLs. One possible command is `TC_IOCTL_MOUNT_VOLUME`, which instructs the root device to create a new device and mount a VeraCrypt volume using that device.

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 then inserted into the I/O queue[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` and l. 886, 673f., 822f. and 844 in `EncryptedIoQueue.c`.

- **I/O queue** This queue is responsible for initiating the I/O for incoming requests. It also adds read requests to the completion queue[15].
- **Completion queue** This is where the decryption of successful 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 [11].

appendix on how to compile veracrypt in debug mode using VS 2019?

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 chapter 2.1. If more details are of interest, [34] 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 [35]. Because the basics did not change, we still mostly rely on the findings presented in [34].

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 [34]:

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

[15] See l. 456f. and 499 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 [36] for more information on TPMs. They are not really relevant for this chapter.
[19] [34] 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."

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 [34].

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 [34].

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 [34]:

- 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 [34]. The algorithms available for encryption are AES-CBC and AES-XTS (the latter since Windows 10, build 1511) [35].

Refer to chapter 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.

Looking at `cryptsetup`'s implementation of its support for BitLocker volumes may also be interesting. This is left as an exercise for the curious reader.

Figure 4.6 shows that the BitLocker driver filters only IRPs with certain major functions, skipping all other requests. We employ a similar tactic for our driver.

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 [34], "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.

[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`.

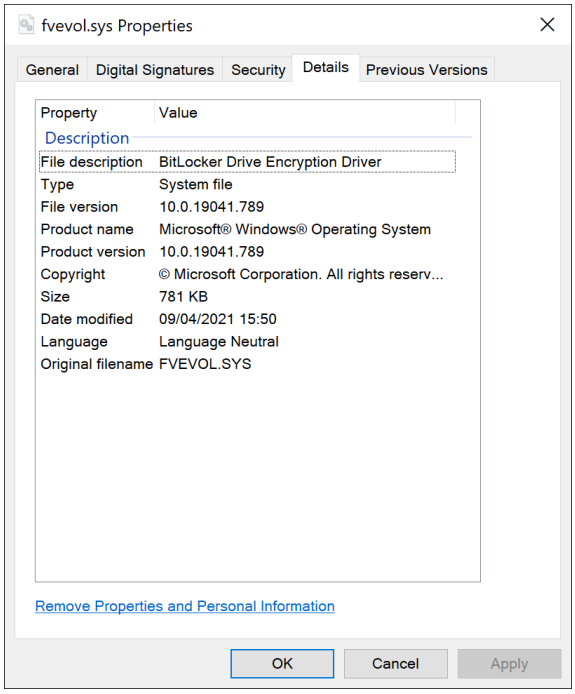

Figure 4.5: Properties of `fvevol.sys` (Screenshot from Windows Explorer). Note especially its description. This file is usually located at `C:\Windows\System32\drivers\fvevol.sys`.

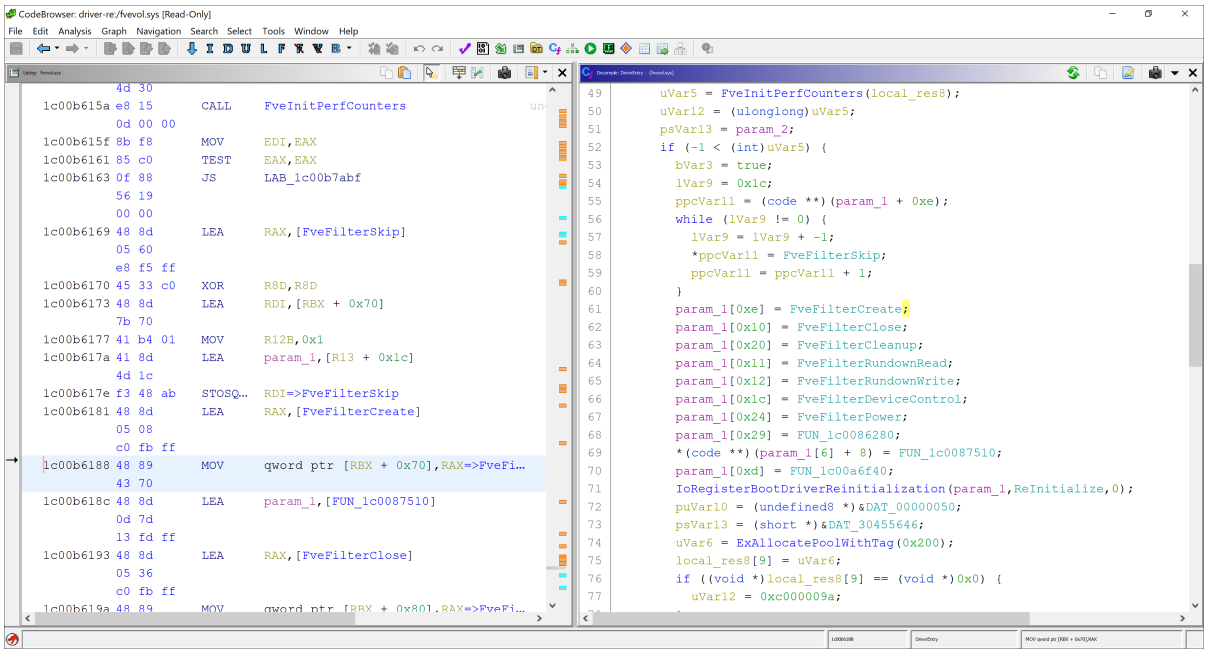

Figure 4.6: Assembly and disassembly of parts of the `DriverEntry` function (Screenshot from Ghidra). Shown here is the definition of which function of the driver will be called when an IRP passes through it. These are called dispatch routines. The driver has to specify the different dispatch routine for each IRP major function. See chapter 5.2.3 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 other major functions use the generic `FveFilterSkip` routine, which usually just forwards the IRP to the next driver in the stack (except for some error cases, where it fails the request).

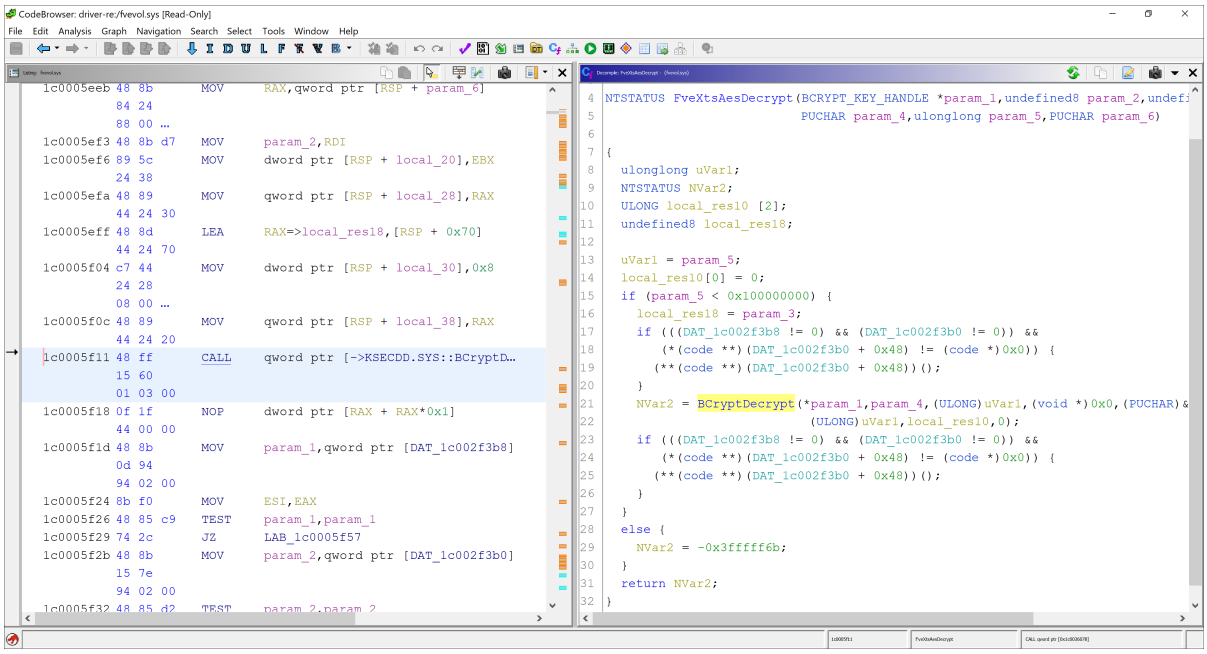

Figure 4.7: Assembly and disassembly of the `FveAesXtsDecrypt` function (Screenshot from Ghidra). `fvevol.sys` uses functions from `ksecdd.sys`, which is the "Kernel Mode Security Support Provider Interface", for all cryptographic functionality. Shown here is the implementation of AES-XTS decryption,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

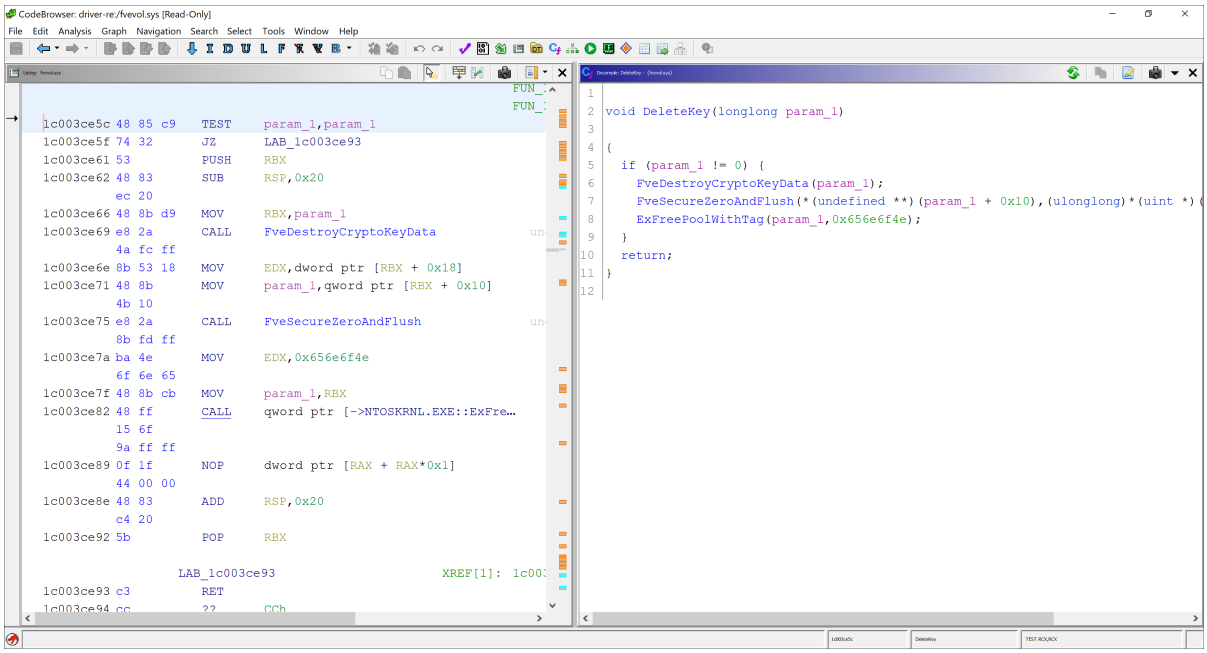

Figure 4.8: Assembly and disassembly of the `DeleteKey` function (Screenshot from Ghidra). The `FveDestroyCryptoKeyData` function notifies the `ksecdd.sys` driver that the key is no longer needed. `FveSecureZeroAndFlush` zeros out the key data and also calls `KeSweepLocalCaches`, which is not documented (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 chapter 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 [37].

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] [37].

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.

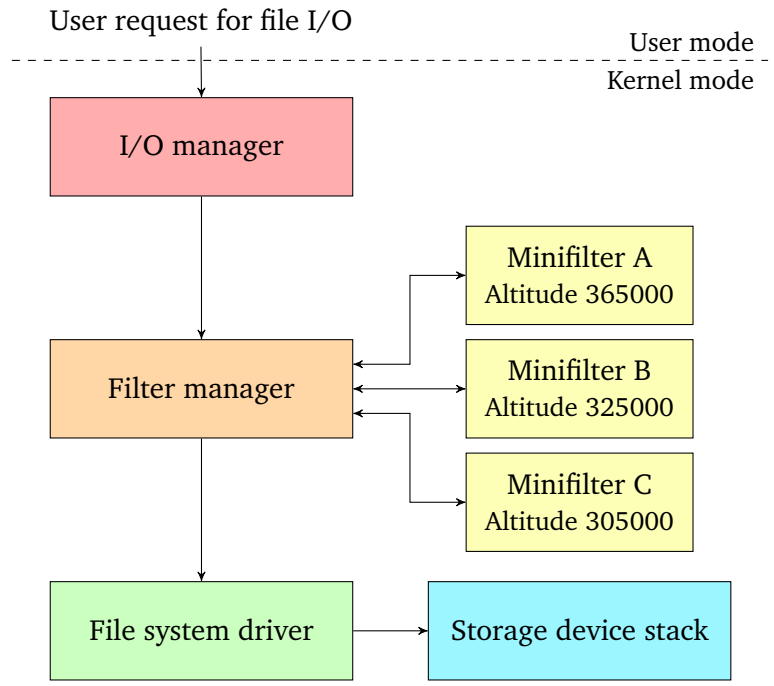

Figure 5.1: Example of a filter manager I/O stack (modified after [37]). A user initiates an I/O request, which the I/O manager receives first and forwards to the file system. The filter manager intercepts the request and lets it pass through the registered minifilters, in order according to their altitude (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 [37] for more information on the filter manager and minifilter drivers.

this and this?

5.1.2 The Windows Driver Frameworks

As already mentioned in chapter 2.2.2, another possibility to develop a driver are the Kernel Mode Driver Framework (KMDF) and User Mode Driver Framework (UMDF),

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

together called the Windows Driver Frameworks (WDF). These try to simplify things by 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 [9].

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 [9].

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 though abstraction (see chapter 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 [13]. 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 filter drivers, though (see chapter 5.2.5 for how our final 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; chapter 5.2 should be a good starting point.

[9] contains detailed information on the architecture and inner workings of both KMDF and UMDF. The official documentation [13] 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 Final WDM Driver

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

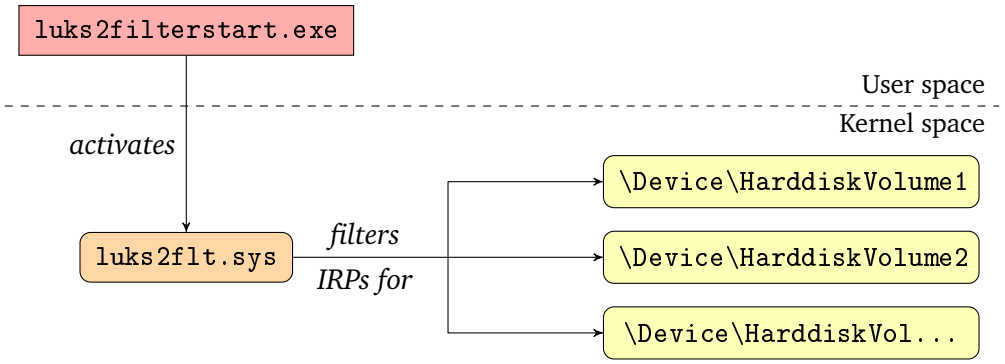

Figure 5.2: Basic communication and architecture of <PROJECT NAME>. To activate the filtering, `luks2filterstart` sends a custom IOCTL to a device, in response to which `luks2flt` starts filtering requests.

considerations during its design in detail.

5.2.1 Architecture

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

`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 chapter 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.

mention that luks2filterstart is written in Rust

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. 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[4] and it became clear that compiling this for the Windows kernel was not feasible.

- **libsodium** Chosen for its promising 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

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

process we realized that this library does not support the AES-XTS mode[5], which is crucial for our driver's functionality. Therefore, we had to reject this library.

In the end, we decided that for a scientific project it would be okay to implement our own cryptographic functionality. We adapted the AES and AES-XTS implementations of two existing Rust libraries[6], translating them to C code. More on this topic is described in chapter 5.2.4.

here and/or in other sections: screenshots from WinDbg showing e.g. a device stack and luks2flt's device object

explain "by the way"? or move to chapter 2.2.3

– look-aside lists (page 331)

5.2.2 Installation

[9] and [11] describe to ways to install a driver: either using setup information (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 chapter 4.2.2, chose the `CreateService` API, and BitLocker already ships with Windows. Because we did not want an extra executable for installing our driver during development, we chose to use an INF file to install `luks2flt`.

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. [11] 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 one wants them to do.

The most important parts of the INF file that we wrote for `luks2flt` is shown in Figure 5.3.

Screenshots from the registry showing the modified volume class' lower filters?

5.2.3 Initialization and Configuration

`luks2filterstart.exe`

Because 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], `luks2flt` attaches to all devices in the Volume class. However, the default is let every incoming request pass through, and only start filtering after the appropriate IOCTL from `luks2filterstart.exe` was received. more detail about what other information is contained in the I

[5] 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.

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

[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.

mention possibility of configuring via registry which devices luks2flt should attach to
we still need luks2filterstart.exe for sending the password, but all other config could be in the registr
`DriverEntry` function, keep Figure 4.6 in mind

5.2.4 De-/encrypting Reads and Writes

custom AES implementation, mention failed attempts of making existing crypto libraries work in ker
make sure this chapter and chapter 2.1.4 are not too similar

LUKS2 supports many encryption algorithms (see [1], Table 4), but `luks2flt` only supports `aes-xts-plain64`.

```
VOID
EncryptWriteBuffer(
    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->Encrypt(
            &CryptoInfo->Xts, Buffer + Offset,
            VolInfo->SectorSize, Tweak
        );
        Offset += VolInfo->SectorSize;
        Sector += 1;
    }
}
```

5.2.5 Handling Other Request Types

5.3 Security Considerations

reference chapter 4.1?
https://crates.io/crates/secrecy
[38] [39] [40] [41]
https://tails.boum.org/contribute/design/memory_erasure/
https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-rtlsecurez
This is the generic solution written in C for clearing memory of the OpenSSL project (as of version 1.1.1l) [citation needed], explain `volatile`?: the doc for it

```
/*
 * Pointer to memset is volatile so that compiler must de-reference
 * the pointer and can't assume that it points to any function in
 * particular (such as memset, which it then might further "optimize")
 */
typedef void *(*memset_t)(void *, int, size_t);

static volatile memset_t memset_func = memset;

void OPENSSL_cleanse(void *ptr, size_t len)
{
    memset_func(ptr, 0, len);
}
```

The most popular CPU architectures, including x86, x64, ARM, and SPARC, have a hand-written assembly version of `OPENSSL_cleanse`. This avoids the compiler from optimizing the zeroing away.

6 Performance of Our Driver

keep [24] in mind

6.1 First Experiments

6.2 Final Experimental Setup

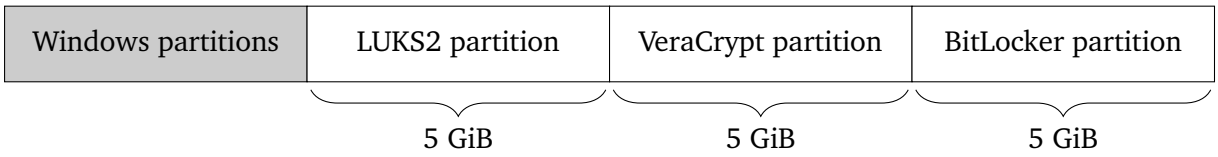

Figure 6.1: Disk layout of the real hardware SSD.

6.3 Results

For testing purposes, we tried optimizing the performance by restricting support to AES256-XTS. This enabled removing some if-else constructs that dispatched de-/encryption functions based on whether AES128 or AES256 was used. Even though these conditionals were located in a performance critical section, we saw no speed improvements. Our theory for why this was the case is the following: our driver was only ever used for one LUKS2 partition and therefore always took the same path through the if-else (either always AES128 or always AES256). 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 default compiler optimization settings in Visual Studio were a little bit conservative and also optimized for small code size rather than high speed/performance. After tuning these settings to enable more aggressive optimizations and also focus on speed rather than code size, we found that this is a little bit complicated....

7 Discussion

8 Conclusion

A Notes On Online References

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

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

The VeraCrypt documentation [33] has been archived at `https://web.archive.org` and is also available at VeraCrypt's GitHub repository.

todo

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

List of Figures

List of Tables

Bibliography

[1] 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

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

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

[4] 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

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

[6] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)," U.S. Department of Commerce, Tech. Rep., 2001.

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

[8] *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

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

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

[11] *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

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

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

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

[15] 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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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

[22] 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.

[23] 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.

[24] 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.

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

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

[35] 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

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

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

[38] 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

[39] 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.

[40] C. Li, L. Guan, J. Lin, B. Luo, Q. Cai, J. Jing, and J. Wang, "Mimosa: Protecting private keys against memory disclosure attacks using hardware transactional memory," *IEEE Trans. Dependable Secur. Comput.*, vol. 18, no. 3, pp. 1196–1213, 2021.

[41] L. Simon, D. Chisnall, and R. J. Anderson, "What you get is what you C: controlling side effects in mainstream C compilers," in *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 2018, pp. 1–15.