BITS DON'T LIE

DETECTING NTFS DRIVER FINGERPRINTS

by

Nick Borchers

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, Faculty of Science
Master Software Engineering
to be defended publicly on Monday, 18 December 2023, at 4:00 PM.

Student number:	852454271	
Course code:	IM9906	
Thesis committee:	dr. ir. Hugo Jonker (chairman & supervisor),	Open Universiteit
	dr. Greg Alpár (supervisor),	Open Universiteit
	ir. Vincent van der Meer (reviewer),	Zuyd University of Applied Science

Contents

ABSTRACT

Devices contain lots of data from their users. Many people have a smartphone or personal computer nowadays that collects personal data: i.e., documents, images, and such but also their metadata. Devices used by criminals often contain digital traces to their criminal activities. Digital forensic practitioners uncover and analyze this data to be used in court. Data is often stored in file systems by file system drivers. NTFS is a popular file system for Windows users, but there are also drivers for UNIX-like operating systems. These drivers differ in terms of how they write to storage media: they have their own fingerprints. Up until now, these fingerprints were an untapped source of forensic information. We introduce a novel method to discover NTFS driver fingerprints and use them to show the use of a specific driver on a storage medium: our black-box testing technique uncovers the telltale differences that NTFS drivers exhibit when interacting with storage media. We test drivers for three OSes used in everyday life: Windows, MacOS, and Ubuntu. We additionally introduce a proof-of-concept implementation of NTFS driver detection based on their fingerprints. Digital forensics practitioners should use this detection method to know what operating systems and what drivers have touched storage media they analyze to extract more evidence from them.

1

INTRODUCTION

Context In this modern world, software is a core part of people's daily personal and professional lives. It is everywhere, and its use will only increase. This omnipresence means that many data is stored. Petroc Taylor [Tay22] estimates that in 2023 alone, 120 zettabytes have been stored. Frequently, a file system arranges stored data on a persistent storage medium, allowing the information to be accessed later. A popular file system is NTFS. It is a proprietary file system by Microsoft for use in Windows, but there are publicly available drivers for other OSes.

Personal devices criminals use for illegal purposes often contain digital traces of their crimes. Timestamps, pictures, documents, and other digital artifacts can link a person to an unlawful activity. Perhaps not unsurprisingly, digital forensics practitioners can use this data to gather evidence. Nowadays, this is a more and more common practice. Digital artifacts extracted from storage media can mean the difference between a suspect receiving a punishment or not. Furthermore, using digital forensics (DF) techniques, even more data can be recovered from storage media. Even in the case of a corrupt file system or partially overwritten files, artifacts can sometimes be recovered through the use of DF techniques. In addition, DF also entails tackling the large volume of data. Practitioners cannot analyze disks by hand time-wise; they need automated techniques.

Digital evidence is used in court. Data extracted by DF practitioners can mean the difference between a suspect getting a sentence or being forgiven. To be admissible in court, evidence has to be sound. The techniques used to acquire should be verified and explainable. Conclusions drawn by forensics practitioners should be clear and understandable by the judge so they can weigh the value of the evidence and file an appropriate indictment.

Problem Criminals hide what devices operated on their storage media. Investigators must first prove the suspect is hiding data to use hidden data for evidence. They hide their own devices, but also the fact that accomplices touched their storage media. This poses a problem, as this previously untapped source of evidence might mean the difference between the suspect receiving a punishment or not. To hide digital evidence, suspects have some techniques at their disposal. First and foremost, suspects can simply withhold the fact that they own another computer. Second, more technologically advanced suspects can use encryption and anti-forensics (AF) techniques. These techniques are, according to

Rogers [Rog06] "Data hiding, artifact wiping, trail obfuscation, and attacks against the CF process/tools."

Solution NTFS drives contain traces to devices that wrote to them due to detectable differences in driver behavior. 1.1 This previously untapped source of information helps forensics investigators to squeeze even more evidence out of a storage medium. Our experiments show that the Windows 11 driver, the Paragon [1] driver for MacOS Catalina and the kernel NTFS driver used with Ubuntu 22.04 all exhibit unique characteristics in terms of how they operate a drive.

Contributions

1. Novel method for detecting telltale differences between NTFS drivers.
2. NTFS driver signatures.
3. Detection technique to show what OSes have touched a drive (+proof-of-concept implementation).

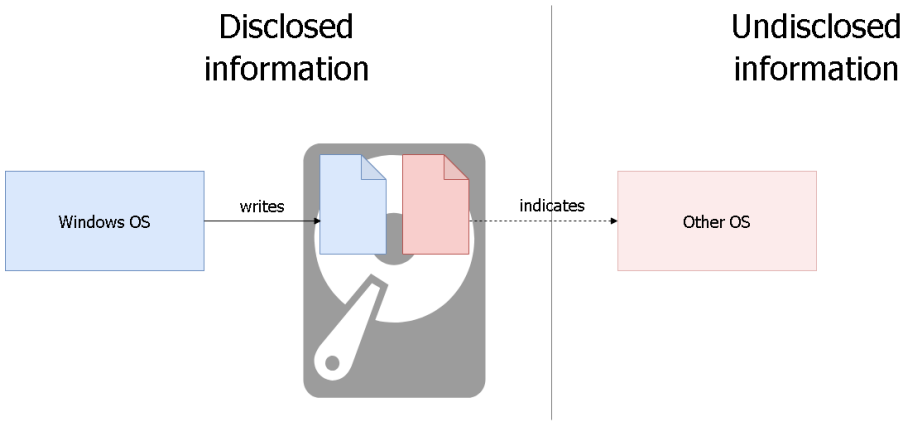


Figure 1.1: Example: a file on a storage device indicates it was created by a driver belonging to an operating system that investigators were unaware of up until now.

Steganography Steganography and other data-hiding techniques are a problem for investigators. They need robust tools & techniques to gather as much data as possible from their storage media. Knowing what devices a storage medium has interacted with can provide valuable insight for the investigators. A Deniable File System (DFS) is a steganography technique where the defender can plausibly deny its existence. An attacker can persuade the defender through forceful methods like violence to hand over the decryption key for a regular encrypted file system. However, in the case of a DFS, the defender can plausibly deny there is a file system in the first place. Only the so-called decoy OS is decrypted when the attacker persuades the defender to decrypt the volume. The other segment looks like free space to the naked eye. The other segment containing the hidden OS can be decrypted

[1] https://paragon-software.com

and used only when presented with another key. Using VeraCrypt [2] or its predecessor TrueCrypt [3], anyone can create such a partition.

[2] https://veracrypt.fr/en/Home.html
[3] http://truecrypt.org/downloads

2

BACKGROUND

2.1. TERMINOLOGY

Volume. A volume is a storage unit. It can be an entire storage device or a segment. The term has more precise definitions, depending on the context: On Linux-based systems, a volume means something else than on Windows. For this work, we use the general definition 'storage unit.'

Partition A partition is a logical division of a storage medium. It is in one of the following categories: primary partition, extended partition, and logical partition. The most well-known and often used partition table types are GUID Partition Table (GPT) and Master Boot Record (MBR) (sometimes called MS-DOS partition tables). While GPT was introduced to address some of the shortcomings of MBR, both are still widely used.

File system driver Bytes on a storage medium have to be organized such that files can be retrieved, updated, moved, deleted, and so on. This organization is the responsibility of a file system driver. In other words, it acts like a librarian where the files are the books, and the library is the storage medium.

2.2. NTFS

New Technology File System (NTFS) is a closed-source file system introduced by Windows in the early 1993s and has been used in all Windows versions since the early 2000s. Because this file system is closed-source, most publicly available documentation, third-party implementations, and other tooling for NTFS are created through reverse engineering. Software companies and smaller groups developed many third-party implementations of this file system for MacOS and Linux.

Master File Table (MFT) A key aspect of NTFS is the Master File Table (MFT). It keeps track of all files on the file system. Small files (< 1 kB) are stored entirely in the MFT. This file is also called a 'resident' file. Bigger files, also called non-resident files are stored as

clusters outside the MFT in the $DATA attribute. By convention, system attributes, and files are prepended with a '$.'

The MFT also contains some particular files, of which we highlight a few files that are critical to this work:

- The **$Bitmap** keeps track of cluster allocation. It stores one bit for every cluster. If a cluster is allocated, its corresponding bit is '1'. Otherwise, it is '0'.
- **The USN Journal** is a special NTFS file that tracks file changes. It records what type of file operations occurred on a file and the associated timestamps.
- **The $LogFile** is used to recover the state of the file system when something goes wrong during writing to prevent corruption.

File record The MFT is a list of file records [1]. And while most metadata is stored in file attributes, the file record also stores some. The Logfile Sequence Nr. (LSN) is stored in the file record itself. It references the $Logfile entry for this file.

File attributes Metadata and file content are stored in file attributes in NTFS. An attribute has a name and stores file data or metadata. Some attributes are stored entirely in the MFT, and some are not. Now, we introduce some specific attributes that are key to this work.

- The **$STANDARD_INFORMATION (SI)** attribute stores the file owner, permissions, and **m**odified, **a**ccessed, **c**reated, and **e**dited time. (MACE timestamps). A 'Security ID' (SID) is also stored in the SI attribute. This field is an identifier for a user or group in Windows. From now on, we use the terms $SI.M, $SI.A, $SI.C, $SI.E for the SI timestamps modified, accessed, created, and edited, respectively.
- The **$FILE_NAME (FN)** attribute stores a file's name and size. Also, it stores the four MACE timestamps. A file typically has these two attributes and has eight timestamps stored in attributes. Additional timestamps are in the $LogFile and in the USN Journal. From now on, we use the terms $FN.M, $FN.A, $FN.C, $FN.E for the FN timestamps modified, accessed, created, and edited, respectively.
- The **$DATA** attribute stores the file content. This attribute is stored in the MFT or elsewhere on disk, depending on the file content size.
- The **$EA** and **$EA_INFORMATION** attributes also exist. For this work, the content is left out of scope. These attributes are used to implement the concept of extended attributes for HPFS [2].

[1] https://dubeyko.com/development/FileSystems/NTFS/ntfsdoc.pdf
[2] https://pages.cs.wisc.edu/~bolo/shipyard/hpfs.html

2.3. HIDDEN FILE SYSTEM

Hidden file systems are a steganography technique where the file system is encrypted and hidden. It is based on the principle of Deniable File Systems (DFS). This type of file system allows the defender to deny the existence of the hidden data plausibly. An implementation of hidden file systems and deniable file systems are 'Hidden Volumes' by VeraCrypt [3]. VeraCrypt is available on Windows, UNIX/Linux, and MacOS program. It is the successor of TrueCrypt [4].

If an attacker (law enforcement or anyone wanting to break the encryption for that matter) forces the defender to hand over the key to the outer volume, an attacker can decrypt it. Observing the data after decrypting the outer volume, the attacker will see a decoy OS and data that looks like free space, but actually is a hidden OS. 2.1

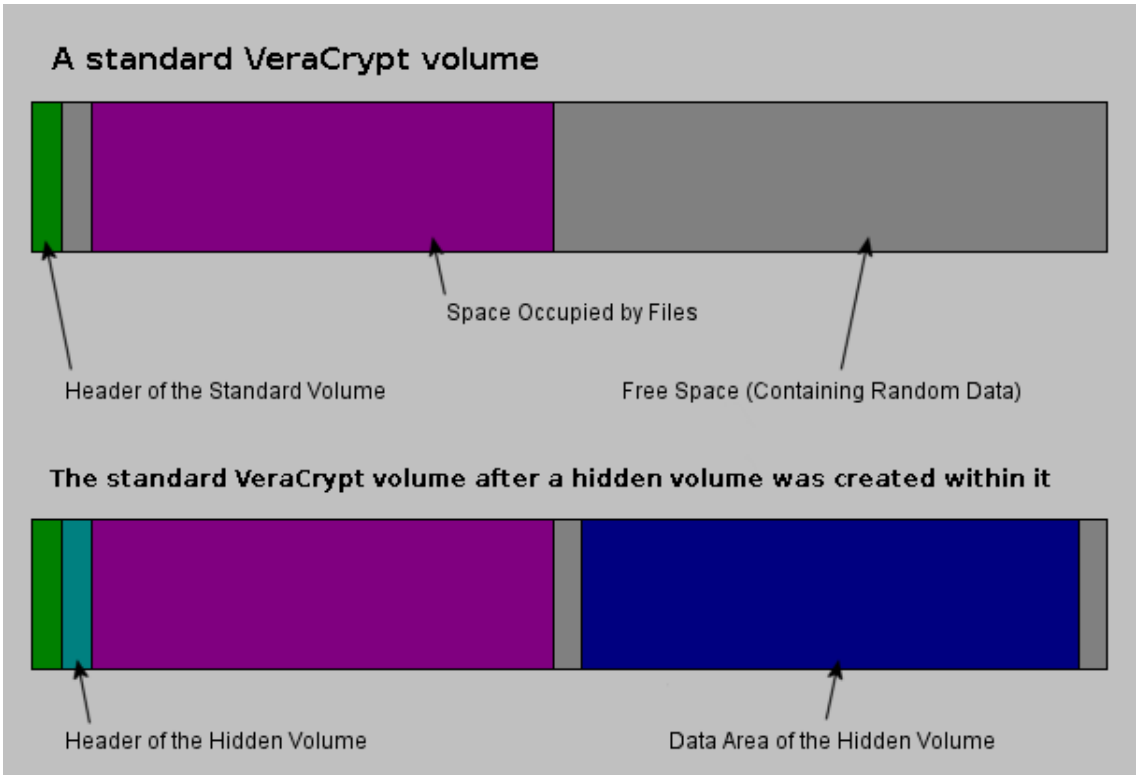


Figure 2.1: A graphical depiction of the physical layout of VeraCrypt hidden volumes https://www.veracrypt.fr

[3] https://www.veracrypt.fr
[4] https://truecrypt.sourceforge.net/

3

DOMAIN ANALYSIS

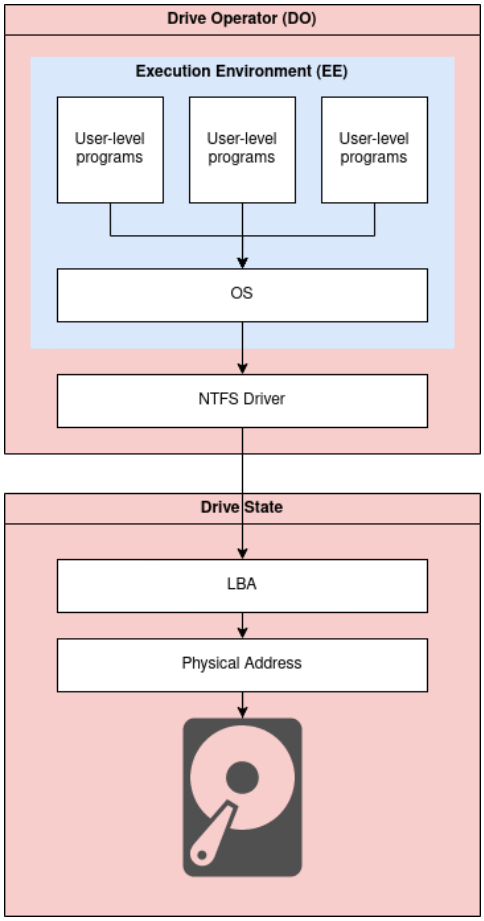
Drive State The bytes returned from the disk in logical order do not match the physical order. This is abstracted away from the OS using Logical Block Addressing (LBA). It is a way to address the storage medium such that the bytes are logically addressable, but where those bytes are physically is up to the device's firmware. We hereby define the term drive state as being the bits that are on the drive in logical order.

File Entry State (FES) A file has a couple of aspects on disk that can be measured. We divide these aspects into metadata, file content, and allocation behavior. Together, these categories make up the File Entry State (FES). We now introduce the term.

Execution Environment (EE) A driver does not operate on its own. Instead, it runs in the context of hardware and other software like the OS and user-level programs. In other words, it runs in an environment. While this is the case for all software, it is essential to make this distinction. We introduce the term Execution Environment (EE). It signifies the environment a file system driver runs in.

Drive Operator (DO) Operating systems create/read/update/remove all kinds of files while they run. They create swap files, administration files, and many more types. This also holds for users. They make code, text, pictures, and many more. A file system driver manages files on persistent storage for OSes and for users. The idea is that these files remain there after the computer has been restarted. A driver is the software interface to the storage device.

To this end, we introduce 'Drive Operator' (DO) as the driver + its execution environment. The drive operator is the only thing responsible for a drive's state. All changes concerning the drive's state and a FES can be attributed to the DO.



(a) A graphical depiction of important domain concepts

4

RELATED WORK

Finding undisclosed/hidden operating systems. Kedziora et al. [KCS17] introduce attacks against the plausible deniability of hidden operating systems. The authors categorize the attacks into three threat models: One-Time Access, Multiple Access, and Live Response Access. In the One-Time Access case, the attacker can observe one binary snapshot of the drive. Similarly, in the Multiple Access case, the attacker can observe the drive multiple times. Lastly, in the Live Response Access case, the attacker can access the machine or network the hidden OS is on.

First and foremost, the authors identify a One-Time access thread. When a hidden OS runs on a hidden volume; they found that three bytes change from one specific value to another when the hidden OS is booted. [KCS17]. The authors conclude that investigators can look for these three bytes when inspecting a drive and raise suspicion of the existence of a hidden OS. More One-Time access threats are presented by Czeskis et al. [CHK+08]. The authors empirically showed that hidden volumes leak information through links and registry entries. They also show that programs like Microsoft Word leak information. The 'safe save.' functionality creates a temporary copy of a file on the hidden volume on the outer volume that is recoverable using publicly available tooling. They finally conclude that successfully using hidden volumes requires knowledge of the operating system and the programs that run on it.

Second, Kedziora et al. [KCS17] also uncover Multiple-Access vulnerabilities. They conclude that hidden volumes/hidden OSes are vulnerable to cross-drive analysis. Cross-drive analysis means multiple drive snapshots are compared to each other and analyzed. The authors show that only the first segment of the drive changes when using the decoy OS, and the last segment only changes when the hidden OS is used.

Third, Kedziora et al. [KCS17] show that VeraCrypt [1] configuration files in the hidden OS and the decoy OS could raise suspicion about the presence of a hidden OS due to the presence/absence of some lines. In addition, they identify that a hidden OS connected to the internet also leaks information.

The works by Kedziora et al. [KCS17] and Czeskis et al. [CHK+08] are helpful to our research mainly because the research methods used for drive analysis apply to our work. Similarly, we draw inspiration for this work from their use of DF tools and techniques.

[1] https://veracrypt.fr/en/Home.html

The authors introduce black-box testing for identifying telltale signs of hidden file systems. Our technique extends this technique in that it can uncover not only steganographically hidden but also physically hidden traces.

Metadata. An essential part of what a file system driver does is operating on metadata. As such, driver differences can be present in metadata artifacts, and we consider related works in this area.

Nordvik et al. [NA22] employ black box testing to uncover exFAT behavior regarding timestamps for Windows, MacOS, and Linux. Furthermore, they found that forensic tools like EnCase [2], Autopsy [3], and FTK Imager [4] handle timestamps differently. These differences are important to consider, as DF tools will be used extensively throughout this work.

Furthermore, James Habben [5] as cited by Nordvik et al., [NTA19] showed that it could be detected on what volume a file was created based on the $ObjectId attribute. The authors' work is useful in that it identifies techniques with which to compare timestamps. Furthermore, the identified artifact, $ObjectId, can be used to identify different (virtual) machines.

In the work by Thierry et al. [TM22], the authors analyzed how applications, middleware, and the kernel adhered to the POSIX standard's timestamp rules. They found several mismatches and unexpected behavior. They also report their findings on timestamp granularity in detail. Furthermore, the authors provide detailed tables to be used by practitioners. Lastly, we draw inspiration from the authors' automated experiment techniques.

Aside from forensic techniques, we specifically consider anti-forensic (AF) techniques to discover timestamp tampering. Artifact wiping, a term introduced by Kessler et al. [Kes07], is altering or removing artifacts to hinder forensics. A couple of works in this category will be discussed now. First, Palmbach et al. [PB20] compare possibly altered timestamps to the $LogFile, Prefetch Files, $USNjrnl, Link files, and Windows event log. These artifacts and the corresponding tooling to read/extract them are useful to this work. The $LogFile was also used by Cho [Cho13] as an artifact to detect timestamp forgery. Furthermore, Galhuber [GL21] showed how traces of several anti-forensic tools could be discovered on NTFS file systems. Lastly, Mohamed et al. [MK19] used binary logistic regression to discover timestamp tampering.

Meanwhile, the other authors focus on a specific piece of metadata to show some sign or behavior. Our approach is more holistic in that it looks at multiple signals. This is such that we provide more robust data to forensic practitioners, and it is also more robust against anti-forensic techniques since the techniques would have to mask all signals we detect.

Allocation patterns. Van der Meer et al. [vdMJvdB21] research file fragmentation patterns for NTFS in 2021. Previous results dated back to 2007 and were severely outdated. The authors compiled a dataset from the metadata of the disks of about 220 computers. This data set, 'Wildfrag,' is available for follow-up research. They identify a till-then undiscussed variant of fragmentation: out-of-order'ness. This is the degree to which a file's allocated clusters are out of order. The work concludes that some fragmentation measures are

[2] https://www.opentext.com/products/encase-forensic
[3] https://www.autopsy.com/
[4] https://www.exterro.com/ftk-imager
[5] https://4n6ir.com/2018/09/20/ntfs-object-ids-in-encase/

incorrectly addressed by current tooling. The authors' work is important for this research in multiple ways. First, the newly identified fragmentation type helps describe the fragmentation degree for NTFS driver types. In addition, the dataset could also be used to research fragmentation patterns.

Additionally, Karresand et al. [KAD19] empirically tested NTFS fragmentation. The authors use black-box analysis to empirically test and report the allocation behavior of NTFS on different Windows versions using virtual machines. The authors tried to determine if it was possible to create a precomputed map of the probability of finding new data at various locations in NTFS formatted partitions. [KAD19]. Using this precomputed map, the authors list common user data locations to prioritize certain parts of a drive in the digital forensics field. The authors also find a disruption in the results and propose a solution in future work. They see that not the best-fit algorithm is used but the worst-fit instead. This contradicts NTFS documentation. Since differences in allocation behavior could also be used to distinguish drivers from each other, the methods this work used apply to our work. In addition, Karresand et al. [KAD20] also empirically tested cluster allocation behavior more generally. They test Windows cluster allocation behavior for various Windows versions and file operations. They conclude that allocation behavior differs for different Windows versions. This observation is relevant to this work, as this is a clear difference that could be detected. These allocation behavior differences could be used to distinguish different OS versions from each other.

The authors uncover the allocation behavior of NTFS drivers but do not use it for forensic purposes. Our work is novel in that it uses these data to identify driver fingerprints.

5

RESEARCH QUESTIONS

The main problem under consideration is

Given a drive, how do you tell which NTFS drivers have interacted with it?

To this end, we consider the following aspects of a drive's state:

- File metadata, including NTFS attributes and information from $LogFile, the MFT, and the file record.
- File allocation behavior, that is, which blocks have the driver chosen to allocate to files.
- File content.

First, we determine which features in each of these categories distinguish one NTFS driver from another. Additionally, we consider that a driver does not run in isolation but in an execution environment. It interacts with an operating system, a drive, and other software. These additional factors have to be taken into account. Since this is also how a driver is operated in practice. This leads us to the following subquestion.

RQ1. How to identify differences between NTFS drivers...?

This question can be further divided into the following subquestions:

- **RQ1.1** ...in terms of metadata?
- **RQ1.2** ...in terms of allocation behavior?
- **RQ1.3** ...in terms of file content?

Second, given that we have found telltale differences between drivers, we use them to distinguish them from one another. This leads to the following subquestion.

RQ2. How to distinguish NTFS drivers, given a drive they have interacted with?

- in terms of metadata?
- in terms of allocation behavior?
- in terms of file content?

6

METHODOLOGY

6.1. RQ1: FINDING DIFFERENCES

DOCUMENTATION

To find differences between how drivers represent NTFS, we compare the drivers. First, we consider official design documentation/specifications. It has the advantage of being relatively simple to compare high-level descriptions. It also has downsides, however. Specifications or design documents are not available for most drivers. And if they are available, they are severely outdated. For example, the Windows NTFS driver documentation reports that it uses the best-fit strategy, but Karresand et al. show this is not the case [KDA20]. Therefore, software documentation gives no guarantees on actual behavior.

SOURCE-CODE ANALYSIS

Next, we consider source code analysis. It has the advantage in that it is a very detailed description of driver behavior. Nonetheless, one of its drawbacks is that most drivers are closed-source. Note that source code could be reconstructed through the manually expensive reverse-engineering process. Regardless, making meaningful statements about a file system driver while taking the execution environment into account (Operating system and user-level programs) is a challenge for programs of 1000+ lines of code (LOC) and a tremendous human effort for programs of 10000+ LOC. For context, the NTFS Linux kernel driver has 30k+ LOC. Furthermore, convincing a judge that a driver works a certain way is unreliable because the source code says so. Lastly, we need automated techniques to apply this work to broader use cases. We do not use source-code analysis for this work as we have much better options.

BLACK-BOX ANALYSIS

Another approach is black-box analysis. It involves subjecting the System Under Test (SUT) to specific inputs and observing its outputs while omitting its internal workings. This information is then used to model the behavior of the SUT.

Performing black-box analysis for a file system driver would roughly include the following steps:

1. **Measure drive state.** Initially, the state is recorded and persisted.
2. **Perform file operation.** Subject the driver to some input.
3. **Measure drive state.** The changed state is recorded and persisted.
4. **Compare before vs. after.** Compare and analyze the drive recorded before and after the operation to find differences.

Black-box analysis has the upside in capturing how a driver will function in practice. The system's behavior as a whole is measured, even interactions with the OS and other programs. In addition, since this method requires only the executable software, there is no need for reverse engineering to acquire the source code.

On the other hand, it might be hard to distinguish if some effect is due to the driver, a user-level program, the OS, or any program. Since we only consider the bytes that eventually arrive on the drive, we cannot be sure what has ultimately led to those bytes. We do not measure what happens inside of the system; it is treated as a black box.

Another downside of black-box analysis is that it is a testing technique. These techniques have the downside that they never test every code path for reasonably sized programs. But completeness is not the point. Finding clear differences between file system, driver implementations do not require testing every control-flow path.

AUTOMATA LEARNING

Also worth considering is automata learning [Ang87] (also called model mining). It is a machine learning technique that results in a model of a software system. A Mealy Machine [Mea55] is a specific type of this technique, and they suit this task since they consider their current state (the storage medium) and the input (the file operation) to determine their next state.

Automata learning has all of the advantages of black-box testing and more. Creating individual automatons for each driver and then comparing them allows for a more detailed analysis of a larger space of driver inputs compared to black-box testing: the model is an abstract representation of the specific inputs, outputs, and transitions in the data.

Nevertheless, automata learning has more downsides as well. Mainly, there are some technical and theoretical hurdles to overcome before this technique can be applied.

1. **State explosion problem.** **[CKNZ11][Val96]** The state explosion problem occurs in software system analysis when the space of states is too big, even for a moderately sized system. A typical advanced solution is to devise some abstraction for a state such that multiple states can be represented as one.

2. **Performance.** To ensure the file system changes are flushed to the drive and not in a cache, experiments are usually done by stopping and starting the OS. This works but costs a lot of time. An efficient technique is needed to ensure bytes reach the drive.

Conclusion

To conclude, in this work, we opt for black-box testing. This decision is mainly motivated by the fact that specific technical challenges must be addressed before implementing more complex approaches like automata learning. Other methods like documentation or source code comparison are lacking because we cannot conclude actual software behavior.

6.2. RQ2: DETECTION

Given that we know what unique effects the drivers exhibit on a storage medium, some of these could be used to distinguish them from one another. Like the other research question, a file's state is divided into metadata, file content, and allocation behavior.

Of course, this highly depends on the differences these drivers exhibit. If there are a lot of apparent differences, this could be a trivial feat. Conversely, it is harder to draw firm conclusions if the differences are non-existent or subtle. Regardless, we discuss, on a high level, how to tell drivers apart based on their behavior.

Parsing the storage medium is necessary to determine a file's state. The unorganized bytes have to be organized in some way such that we know what they mean. There are popular forensic tools like The Sleuth Kit (TSK) [1] that can parse NTFS, and there are also open-source libraries for various programming languages. Parsing a file system is no easy feat, so it is preferable to use an existing implementation.

To strengthen this argument, we give an example. For example, in our preliminary research, we noticed that files created by a Windows NTFS driver had a $Logfile entry number. On the other hand, drivers for other OSes did not have this. This means that when we subsequently encounter a file that has $Logfile entry number, we can conclude that this file was created using a Windows NTFS driver on a Windows machine.

Apart from files in isolation, we also consider the drive as a whole. Mounting also affects a drive. For example, the Windows 10 and 11 NTFS drivers will create a 'System Volume Information' directory in the root file system. This means that mounting induces changes on a drive not because a user performed some operation on a file but simply because the drive was mounted.

Parsing NTFS induces a risk. NTFS implementations and parsers behave differently. Different Windows versions already implement some unique flavors of NTFS. Implementations for other OSes and programming languages probably behave even more differently. Also, some researchers have shown that forensic tools contain inconsistencies in file system parsing [NA22]. For this work, we use an open-source NTFS parser to analyze the file state with our software. We do not implement our NTFS parser since this is a significant effort and not interesting for a master thesis; we create new knowledge. So, since we know that these parsers have limitations, we have to discuss how to address them. We address them by cross-checking the results of various open-source NTFS parsers and performing manual checks.

6.3. VALIDATION

In principle, validation is not necessary. Since the research method guarantees that the results are actual driver behavior, we can conclude that they behave as such in practice. Regardless, we validate our findings using a storage medium on which different drivers have operated. To this end, there are two options: Synthetic or open-source. We discuss them below.

Synthetic disks. First, there is the option of synthetic disks. This entails creating our drive and manually executing file operations. This is called data simulation. The objective is to create a storage medium commonly faced by forensic practitioners in practical settings. A successful simulation entails creating a functional drive featuring a range of files,

[1] https://www.sleuthkit.org/

file operations, and mount-unmount cycles across different operating systems. Furthermore, the storage medium experiences file creation and modification using various programs to replicate realistic access patterns. Although automated synthetic data simulation methods exist, their lack of a proven track record deems them unnecessary for our specific case.

Open-source drive images. An alternative is to leverage open-source databases with drive images. However, the available options that align with our requirements are limited. While databases featuring Windows NTFS images are prevalent, equivalent databases for Ubuntu and MacOS NTFS images are scarce. As a result, synthetic data is our choice. This validation method enables us to identify obvious pitfalls in our method.

7

EXPERIMENT DESIGN & SOFTWARE DEVELOPMENT

Since we will be writing software to automate (parts of) experiments, we consider the experiment design and software design in tandem rather than in isolation. This automation is necessary for a few reasons:

- **reproducability.** Researchers and forensic practitioners can use the software to rerun experiments and reproduce results.
- **extendability.** Researchers and forensic practitioners can extend the software to include other OSes, file systems, drivers, or file operations.
- **a flexible research process.** When encountering issues or possible extensions during the writing process, parameters are tweaked, or the program is extended. The experiments can be rerun automatically rather than manually executing every step.

7.1. EXPERIMENT DESIGN

The effect of a file operation on a file's state is tested using a black-box testing. On a high level, the research method comprises the following steps. We explain these steps in more detail in dedicated subsections.

- **capture the drive state (A).** First, to ensure all effects are recorded, we capture the state of the whole drive. The state encompasses the bits that comprise both the partition and file system. Alternatively, one could extract a subset of the drive to save time/space, but we keep partitions sufficiently small to avoid this issue.
- **perform file operation.** We perform a file operation such as creating or deleting a file. Apart from work by Bouma et al. [BJvdMVDA23] constructed a canonical list of file operations in their work. Apart from this, to our knowledge, no such list is available in academic literature.
- **capture the drive state (B).** The state of the whole drive is captured again in the same fashion as step one.

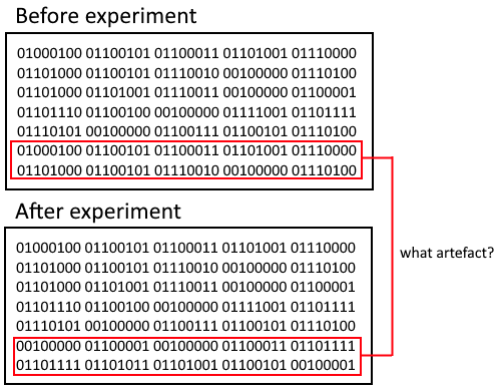


Figure 7.1: Differing bits on a drive, what artifact do they represent?

- **examine differences between A, and B.** To draw conclusions on the effect of a file operation, we compare differences between the drive from before the operation vs. after the operation.

  First, a possible approach is to find drive differences by comparing them bit by bit. This technique benefits from the fact that it is thorough: bits are the most detailed representation. However, a bit-by-bit comparison lacks significance without understanding the representation of each bit. An arbitrary bit could mean anything, such as a timestamp, the content of a file, or any other artifact. So, to make these bit-by-bit differences meaningful, one must identify the specific artifact to which each bit belongs. 7.1 exemplifies this phenomenon.

  A second approach is converting the fine-grained bits to a higher-level representation and comparing them. Any differences found are then already at a high level, so they require no additional processing. Digital forensic tools are made with this purpose in mind. They interpret the bits on a storage medium and convert them to some other format that is often higher level.

  The second approach is more straightforward as it allows the use of existing techniques. Also, findings acquired while writing the research proposal indicated exciting results that do not require applying a more complex approach.

Measuring the effect of a file operation Given that we have the drive state before the operation, and after the operation we can compare them. The goal is to find differences in how drivers implement some file operation: these can be used for their fingerprints. Differences that are present across the space of operations are the easiest to detect. Encountering a file that exhibits this difference can be fingerprinted immediately since it does not have to be deduced what operation last occurred on it. On the other hand, differences that are not present for all file operations are harder to fingerprint as we then have to know what operation last occurred on a file.

VM vs. bare-metal An obvious decision we first make is whether or not to run the DO in a VM or run it bare metal. Running it in a VM has the main advantage of allowing for easy automation. VMs like VirtualBox [1] offer a Command-Line Interface (CLI) or Application

[1] https://www.virtualbox.org/

Programming Interface (API). This is convenient if one wants to automate experiments.

On the other hand, this same level of automation can also be reached on bare-metal machines, but the automation then is platform-specific. Windows, as an OS, works differently than UNIX-based systems in this regard. However, this approach has the downside that does not reflect practice. In practice, most OSes used by 'regular' users will not run in a VM; they run on the hardware itself without any virtualization.

Ensuring bytes reach drive Ensuring the bytes actually reach the physical drive is notoriously hard. PostgreSQL recently discovered that it had been using the fsync syscall wrong. [2]. Furthermore, LWN has a long post on how to ensure bytes reach disk.[3]. Much happens before the actual bytes reach the physical storage medium when saving a file. Changes may linger in the cache and are likely not immediately flushed to drive. Then, when the file system decides to write the bytes to drive, the OS also has an in-memory cache for drives. It is hard to ensure that bytes reach the physical storage medium, and we have to design our software for this.

There is a general consensus that shutting down the device, or the whole OS for that matter, ensures that bytes reach the drive. The shutdown signal ensures that all caches are flushed. File-system caches, storage device caches, and firmware caches. This is a problem when performing experiments on a live OS. After a file operation, the OS has to be shut down after every operation. This adds an overhead of possibly minutes for every file operation. This is a severe performance impact when testing the effect of thousands of file operations.

An alternative is unmounting and shutting down the devices without shutting down the OS. This requires that the OS runs on another storage medium. This means the device can be unmounted and shut down and later turned on and remounted again without requiring a full OS restart. Among academia, this method is seldom used. It is unclear why, but we assume this is because many researchers focus on extracting user data from a file system containing an OS on a fixed storage medium.

We decided to restart the OS after every file operation since this guarantees increased reliability of experiments. The main downside is that this severely degrades the speed & rate at which we can do experiments, but we accept that performance cost.

Record state There are two options to record the state. First, capture the entire drive and extract the artifacts later. Second, extract the artifacts immediately and discard the rest of the drive. In the case of a big file system, option one quickly loses favor. Creating copies of a 10GB drive for every file operation costs a lot of storage and space.

However, option one does offer a lot of flexibility. Any changes to the analysis of drive images can be easily rerun. This means we don't have to rerun the whole experiment if only the artifact analysis is updated. Therefore, we opted for option one by default. If time & space become an issue, we use option two.

To conclude, we record and store the entire drive state (also called an image or snapshot). For this, we use the 'dd' [4] command.

[2] https://wiki.postgresql.org/wiki/Fsync_Errors
[3] https://lwn.net/Articles/457667/
[4] https://man7.org/linux/man-pages/man1/dd.1.html

Defining file operation File operations differ per OS. In fact, it differs per API. User-level programs offer an API to the user to interact with files. In turn, OSes have their own API. UNIX-like systems has VFS [5], together with the concept of an inode. The VFS hides the underlying file system API, which is a completely different one in and of itself. In turn, Windows also has its own API for interacting with files. There is no one-to-one mapping between these APIs.

Therefore, we do not define our experiments in terms of these low-level calls. Rather, we introduce the higher-level notion of file operation. Bouma et al. [BJvdMVDA23] introduce this notion for conducting NTFS timestamp experiments. We provide an example of FS API calls to exemplify that they differ. 7.3 We extend this notion in that we provide bash [6] and batch [7] definitions of these file operations. 7.2

Figure 7.2: List of file operations

File operation/OS	Windows	Ubuntu	MacOS
Create	`type nul > a`	`touch a.txt`	`touch a.txt`
Access	`type a`	`cat a`	`cat a`
Update	`echo "a" > b`	`echo "a" > b`	`echo "a" > b`
Delete	`del a`	`rm a`	`rm a`
Rename	`move a b`	`mv a b`	`mv a b`
Attribute change	`type nul > a`	`touch a`	`touch a`
Copy	`copy a b`	`cp a b`	`cp a b`
Copy (overwrite)	`copy /Y a b`	`cp a b`	`cp a b`
Move within volume	`move a dir-a/b`	`mv a dir-a/b`	`mv a dir-a/b`
Overwriting move from other partition	`move C:\a D:\b`	`mv a b`	`mv a b`

Automating file operations How to automate file operations. First, we must have a partition as a starting point that works on all OSes that we test. This is to reduce the number of interfering effects at play. Some testing with various drive partitioning and formatting tools eventually led us to GParted [10], a Ubuntu [11] tool. Luckily, the host OS on which all the guest OSes run supports this tool. Using GParted partitioning and file formatting functionality, we created a small testing partition that could be mounted on all OSes we tested.

Second, we need some way to do file operations on the guest OS. VirtualBox features an excellent interface for sending commands to the guest machine. We use this interface to send batch/bash commands to the system under test. This means that the source code that performs the file operations is kept in version control in the same repository as the testing workbench. This way, we have all the code in one place to track changes.

[5] https://www.kernel.org/doc/html/next/filesystems/vfs.html
[6] https://www.gnu.org/software/bash/
[7] https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-xp/bb490869(v%3dtechnet.10)
[10] https://gparted.org/
[11] https://ubuntu/com

VFS inode API
open(2) and creat(2)
lookup
link
unlink
symlink
mkdir
rmdir
mknod
rename
get_link
readlink
...

(a) VFS inode system calls[8]

Win32 API
AreFileApisANSI
AreShortNamesEnabled
CompareFileTime
CreateDirectoryA
CreateDirectoryW
CreateFile2
CreateFileA
CreateFileW
DefineDosDeviceW
DeleteFileA
DeleteFileW
...

(b) Win32 API[9]

File operations
Create
Access
Update
Delete
Rename
Attribute change
Copy
Overwriting copy
Move within volume
Move from another volume
Overwriting move from another volume
Overwriting move from other NTFS volume

(c) Canonical file operations [BJvdMVDA23]

Figure 7.3: There is no one-to-one mapping between Win32, VFS, and the notion of file operation

METADATA EXTRACTION

To extract file metadata, the file system has to be parsed. To this end, there are a few options.

1. **Write a parser** Writing a parser for NTFS would leave us unable to complete any other work required during the course of this thesis. We hereby decide that it is not a viable option.

2. **Forensics tools** Forensics researchers use these tools to a great extent. TheSleuthkit (TSK) [12] has a set of command-line tools and APIs for parsing various file system (meta)data. The 'istat' command from TSK outputs file metadata. The 'ntfsinfo' command also outputs file metadata [13]

3. **Libraries** There are also libraries in various programming languages in case the parser should be accessed through an API.

File system parsers have inaccuracies and shortcomings. Therefore, we do not select just one. For the experiments, we will use one tool to parse the results but subsequently verify them using another to ensure such shortcomings do not influence our results.

To conclude, there is a range of options for metadata extraction. For our experiments, we picked a library that is accessible through an API. In addition, we use the TSK CLI for verification.

[12] https://www.sleuthkit.org/
[13] https://manpages.ubuntu.com/manpages/focal/en/man8/ntfsinfo.8.html

OS AND DRIVERS

There are many operating systems and many NTFS drivers. We don't test all OSes and their versions as it would require too much time and effort with only marginal gain since we rarely encounter these in practice. Instead, we select a few popular OSes to prove this method generally works. The most obvious choices are Windows, some flavors of Linux, and MacOS since these are most often used for personal computing. For Microsoft Windows, we use the latest version, Windows 11, since from now on, the use of older Windows versions will only diminish. For a Unix-like system, we pick Ubuntu 22.04. First of all, Ubuntu is a pretty standard choice for Unix users. 22.04 is the LTS right now, so the use of older versions will only diminish from now on.

Lastly, for MacOS, the choice was not so clear-cut. Preferably, we use the latest version. However, not every MacOS version is efficiently run in VirtualBox [14] so we base our choice on the availability of a VBox image. For MacOS Catalina, there is a publicly available VirtualBox image. [15]

SELECTING DRIVERS FOR TESTING

Now that we know what OSes we will perform experiments on, we will decide on drivers to test.

- **Windows 11:** As Microsoft, the creator of NTFS, has NTFS support ingrained into the roots of the OS, similar to other Windows versions, we naturally opt for this driver in our experiments. There are no third-party Windows NTFS drivers. We use the latest driver of the latest Windows 11 version.

- **Ubuntu 22.04:** Ubuntu also has NTFS support. There were two NTFS drivers throughout the years: ntfs-3g and ntfs3. First, ntfs-3g is a user-space driver developed by Tuxera[16]. Being a user-space driver, it had some performance issues. Furthermore, functionality-wise, it was also subpar compared to the alternative. As of the kernel version 5.15 [17] the ntfs3 driver by "Paragon Software" [18]. This driver is the most feature-complete NTFS driver for Ubuntu and the most widely used one since it is the default. The 'ntfs-3g' driver by Tuxera is available in Ubuntu [19]. We base our tests on the ntfs3 driver because it's more feature-complete. So we use the Ubuntu LTS and the latest driver.

- **MacOS Catalina:** While the above previous choices are clear, the following is not. Regardless, we make an informed decision. There are two reputable drivers for MacOS - Tuxera [20] and Paragon [21] - both supporting NTFS format, mounting, and basic file operations. Notably, we encountered challenges mounting a drive with Tuxera during preliminary experiments. Throughout these initial tests, considerable effort was

[14] https://virtualbox.org
[15] https://github.com/myspaghetti/macos-virtualbox
[16] https://www.tuxera.com
[17] https://cdn.kernel.org/pub/linux/kernel/v5.x/ChangeLog-5.15
[18] https://www.paragon-software.com
[19] https://manpages.ubuntu.com/manpages/trusty/man8/ntfs-3g.8.html
[20] https://ntfsformac.tuxera.com/
[21] https://www.paragon-software.com/

invested in creating a single partition that can be mounted on all System Under Tests (SUTs). We use the latest version of this driver.

SSD vs. HDD

The experiments will be performed on a physical device to emulate real-world scenarios. All modern storage devices use Logical Block Addressing (LBA), which hides the physical location of data on the drive from the OS. Despite no differences in the actual bytes returned from HDD as opposed to SSD, there are substantial variations in performance and reliability. It is essential to highlight that these differences do not impact the bytes returned from the drive. Therefore, our choice of the device is based on convenience. Given the availability of a removable SSD for conducting the experiments and considering its superior speed, we execute the experiments on the SSD.

NTFS features

NTFS has an extensive array of features; however, many of these functionalities are exclusive to the Windows operating system. Third-party drivers lack comprehensive support for features like compression or alternate data streams, rendering testing of such scenarios unnecessary. Consequently, we use the default NTFS settings across all drivers.

External vs. internal storage media

The OS performs many file operations while it runs. This poses a problem for testing the effect of a file operation on the same storage medium as a running OS because a change can be attributed to either the OS or the actual file operation. Furthermore, detecting the usage of a driver on a drive with a running OS is not necessary for practice since the drive has the clearest possible hint: it contains the actual OS.

To solve this, we perform experiments on a removable storage medium like USB sticks and external drives without a running OS. This way, file changes can be attributed to a file operation.

SOFTWARE VERSIONS

For the sake of completeness and reproducibility, we list all versions of the used software for this work.

- Ubuntu host Machine [22]: 22.04, kernel: 6.2.0-37-generic
- Ubuntu guest Machine [23]: 22.04, kernel: 6.2.0-32-generic
- Windows 11 [24]: 10.0.22621
- MacOS Catalina [25] 10.15.7, kernel: 19.6.0
- Paragon for Mac [26], unable to find the version number.
- TheSleuthKit [27]: 4.12.0
- Oracle VirtualBox [28]: 7.0.12 r159484
- Python [29] 3.10.12
- dfir_ntfs [30] 1.1.18

UBIQUITOUS LANGUAGE

We introduce a ubiquitous language for the software domain. 7.4

[22] https://ubuntu.com/
[23] https://ubuntu.com/
[24] https://www.microsoft.com/nl-nl/windows/windows-11
[25] https://apps.apple.com/us/app/macos-catalina/id1466841314?mt=12
[26] https://www.paragon-software.com/home/ntfs-mac/
[27] https://www.sleuthkit.org/
[28] https://www.virtualbox.org
[29] https://www.python.org/
[30] https://github.com/msuhanov/dfir_ntfs

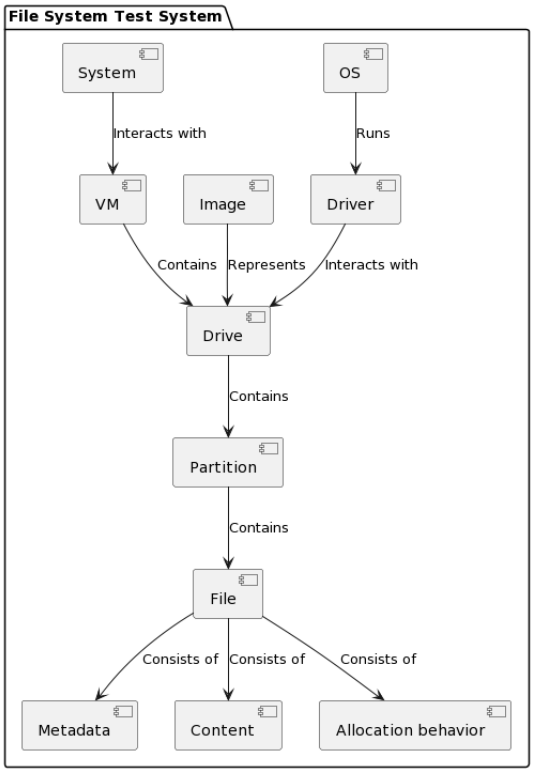


Figure 7.4: Ubiquitous Language for the experiment System

8

RESULTS

8.1. RQ1: FINDING DIFFERENCES

We performed a range of experiments to uncover behavioral differences of NTFS drivers in how they operate a disk together with the OS and other programs. We split these experiments into the various aspects of FES: metadata, file content, and allocation behavior. The behavior resulting from mounting can also be split up into the categories of FES. However, we consider all effects (metadata, allocation behavior, and file content) since the mount operation is an operation on the disk as a whole.

PARTITIONING, FORMATTING & MOUNTING

The life of a file system starts with formatting. It is imposing the basic structure of a file system on a drive. Any differences in on-drive structure could already arise from differences in formatting behavior between drivers. In practice, one hardly encounters a storage medium that is only formatted, but the differences that formatting imposes on a drive could be present after performing operations and are therefore worth examining. Also, to test the effect of file operations, we must mount it and be aware of it.

Format & mount experiment We performed an experiment where we formatted & mounted an NTFS partition on one OS and subsequently mounted it on another OS to see if it had an effect. We extracted and visualized the $Bitmap in a heatmap. Windows automatically mounts a drive as soon as one format. Therefore, we consider the format & mount operation as a single unit in our experiments since these operations cannot be distinguished from one another. Figure 8.1 shows the performed steps. The results are in figure 8.2.

Ubuntu In Ubuntu, the process of creating NTFS partitions is straightforward, allowing us to generate partitions of various sizes without encountering any issues. Notably, these NTFS partitions are mountable on both Windows and MacOS. Additionally, it's worth highlighting that, post-formatting, the file system was not automatically mounted by the operating system, mitigating potential interference and contributing to a cleaner evaluation of driver behavior.

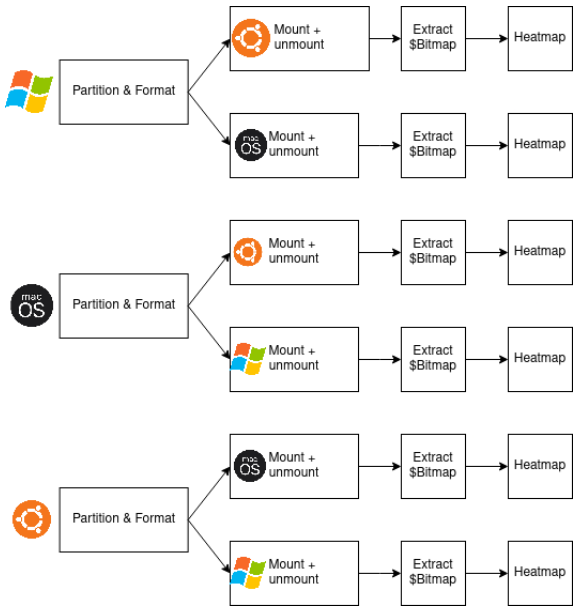


Figure 8.1: The experiment setup to test the effect of mount operations.

MacOS While we use the Paragon [1] For most experiments, we did uncover some behavior of both the Paragon and Tuxera [2] driver that is worth documenting.

MacOS Paragon Formatting a drive to NTFS in Paragon works, and it can also mount MBR partitions with NTFS formatted by other drivers. 8.3 It was unable to mount GPT partitions, but this could also be due to a shortcoming of the MacOS 'Disk Utility' program.

MacOS Tuxera Just like Paragon, Tuxera was unable to format and mount GPT partitions. In addition, the Tuxera driver could not format partitions that did not comprise the whole volume. While this could technically be a shortcoming of Disk Utility: Paragon's driver did not have this limitation.

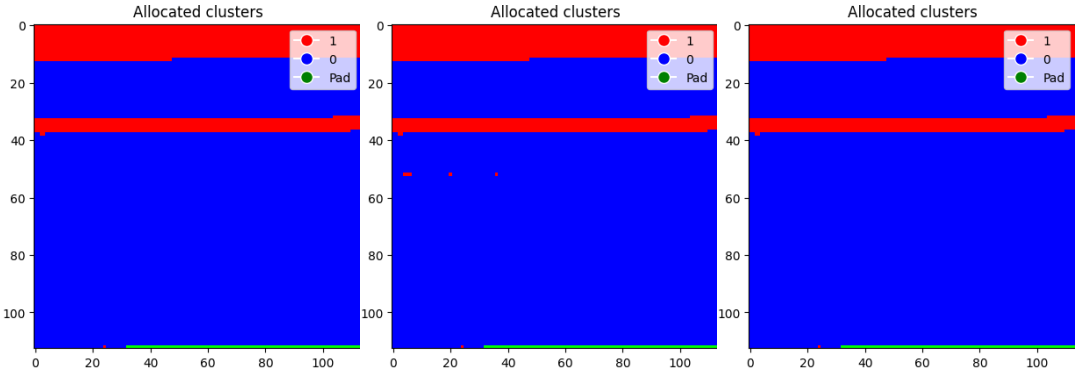
Apart from the Disk Utility interface, the following command can be used to create an NTFS file system in MacOS Catalina with the Paragon driver installed:

```
hdiutil create -size 10m -fs NTFS -layout SPUD -type SPARSEBUNDLE -volname "My NTFS
```

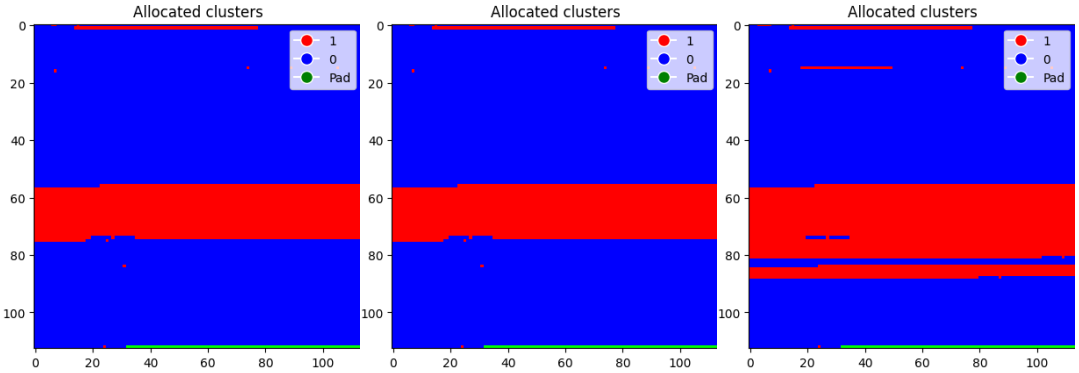
MacOS Both the Paragon and Tuxera NTFS driver for MacOS creates a folder titled '.fseventsd' when mounting. This folder is a system event log of some sort. Regardless, our experiments show that it is always created when mounting. We cannot be sure that the driver created this file. This behavior can also be attributed to the OS. To conclude, the MacOS NTFS drivers also emit characteristic files that could give their usage away.

MacOS creates another particular file named '$UGM.' This file is created upon mounting. Since the other drivers do not create such a file, it is a unique characteristic. To con-

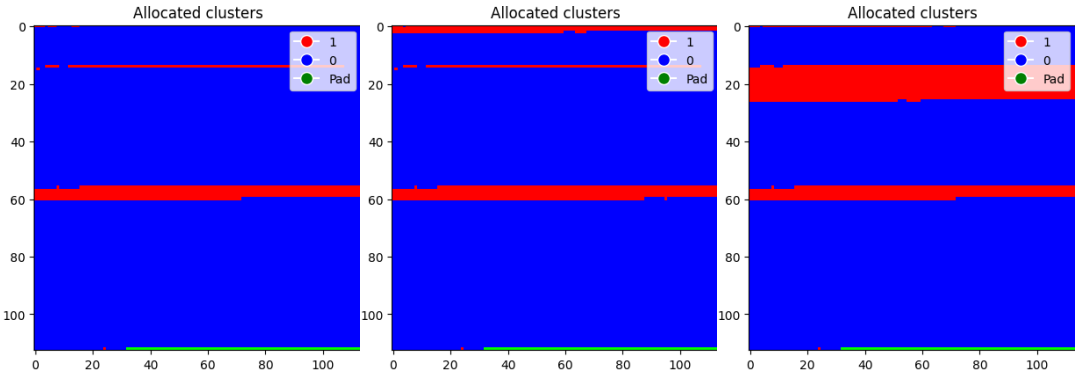
[1] https://www.paragon-software.com/home/ntfs-mac/#
[2] https://ntfsformac.tuxera.com/



(a) Windows (b) Windows-MacOS interaction (c) Windows-Ubuntu interaction



(d) MacOS (e) MacOS-Ubuntu interaction (f) MacOS-Windows interaction



(g) Ubuntu (h) Ubuntu-MacOS interaction (i) Ubuntu-Windows interaction

Figure 8.2: Mounting the NTFS file system already imposes changes on a drive

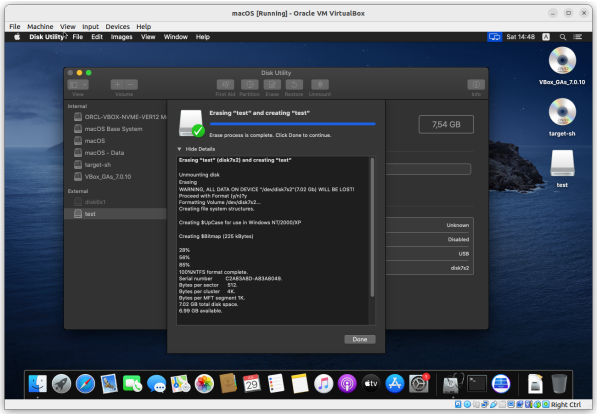


Figure 8.3: Paragon's driver can format a drive as NTFS

clude, every driver we tested had characteristic files that they emit. The presence, or lack thereof, strongly indicates the usage of specific drivers.

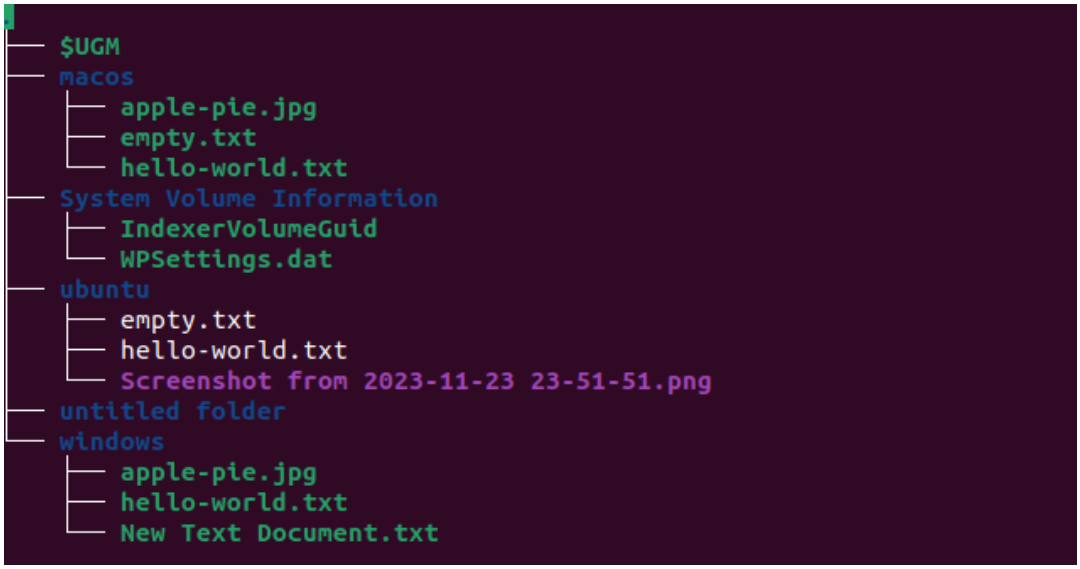


Figure 8.4: A storage medium mounted in MacOS contains a '*UGM*' file in the file system root

Ubuntu Mounting an NTFS drive with the ntfs3 driver introduces no telltale files on the drive.

METADATA

This section explains the experiments we performed to test the effect of file operations on file metadata and the results we acquired. To test this effect, we first devise a list of operations that we test. To our knowledge, there is no consensus on a single list of file operations in academia. However, Bouma et al. [BJvdMVDA23] introduce a canonical list of file operations. We implemented these operations as bash/batch commands. 7.2. Executing these operations with our experiment setup yielded the following results. 8.5 8.6 8.7 8.8 8.9 8.10 8.11 8.12

Figure 8.5: Metadata after the 'create' operation. Evidently, no baseline file exists in the case of the file creation operation. Consequently, a file is created out of thin air, so we list the entire file metadata to present a comprehensive picture. Conversely, we give a more concise representation of the metadata changes.

	Windows	MacOS	Ubuntu
Filename	a.txt	a.txt	a.txt
LSN	1058205	0	0
SID	266	259	258
$SI.M	13:59:11.823474500	15:59:20.000000000	12:45:24.077676700
$SI.A	13:59:11.823474500	15:59:20.000000000	12:45:24.089676700
$SI.C	13:59:11.823474500	15:59:21.000000000	12:45:24.089676700
$SI.E	13:59:11.823474500	15:59:20.000000000	12:45:24.089676700
$FN.M	13:59:11.823474500	15:59:20.000000000	12:45:24.077676700
$FN.A	13:59:11.823474500	15:59:20.000000000	12:45:24.089676700
$FN.C	13:59:11.823474500	15:59:21.000000000	12:45:24.089676700
$FN.E	13:59:11.823474500	15:59:20.000000000	12:45:24.089676700
$DATA	Present	Missing	Present
$FN	Present	Missing	Present
$SI	Present	Missing	Present
$EA_I	Present	Missing	Present
$EA	Present	Missing	Present

Figure 8.6: Metadata after the 'access' operation

	Windows	MacOS	Ubuntu
Filename	a.txt	a.txt	a.txt
LSN	1057314	0	0
SID	266	259	258
$SI.M	13:59:42.809894300	15:45:34.000000000	12:39:38.654808000
$SI.A	13:59:42.809894300	15:45:34.000000000	12:39:38.670816000
$SI.C	13:59:42.809894300	15:45:34.000000000	12:39:38.670816000
$SI.E	13:59:42.809894300	15:45:34.000000000	12:45:24.097676700
$FN.M	13:59:42.809894300	15:45:34.000000000	12:39:38.654808000
$FN.A	13:59:42.809894300	15:45:34.000000000	12:39:38.670816000
$FN.C	13:59:42.809894300	15:45:34.000000000	12:39:38.670816000
$FN.E	13:59:42.809894300	15:45:34.000000000	12:45:24.097676700
$DATA	Present	Missing	Present
$FN	Present	Missing	Present
$SI	Present	Missing	Present
$EA_I	Present	Missing	Present
$EA	Present	Missing	Present

Figure 8.7: Metadata after the 'attribute change' operation

	Windows	MacOS	Ubuntu
Filename	a.txt	a.txt	a.txt
LSN	1057333	0	0
SID	266	259	258
$SI.M	14:03:22.727036500	15:45:34.000000000	12:39:46.110534300
$SI.A	14:03:22.727036500	15:59:22.000000000	12:45:24.105676700
$SI.C	14:03:22.727036500	15:59:22.000000000	12:45:24.105676700
$SI.E	14:03:22.727036500	15:59:22.000000000	12:45:24.105676700
$FN.M	14:03:22.727036500	15:45:34.000000000	12:39:46.110534300
$FN.A	14:03:22.727036500	15:59:22.000000000	12:45:24.105676700
$FN.C	14:03:22.727036500	15:59:22.000000000	12:45:24.105676700
$FN.E	14:03:22.727036500	15:59:22.000000000	12:45:24.105676700
$DATA	Present	Missing	Present
$FN	Present	Missing	Present
$SI	Present	Missing	Present
$EA_I	Present	Missing	Present
$EA	Present	Missing	Present

Figure 8.8: Metadata after the 'copy' operation

	Windows	MacOS	Ubuntu
Filename	b.txt	a.txt	a.txt
LSN	1058685	0	0
SID	266	259	258
$SI.M	14:04:38.908108600	15:45:37.000000000	12:39:47.619288300
$SI.A	14:04:15.889805800	15:45:39.000000000	12:39:47.631294300
$SI.C	14:04:15.889805800	15:45:40.000000000	12:39:47.631294300
$SI.E	14:04:38.908108600	15:59:24.000000000	12:45:24.117676700
$FN.M	14:04:38.908108600	15:45:37.000000000	12:39:47.619288300
$FN.A	14:04:38.908108600	15:45:39.000000000	12:39:47.631294300
$FN.C	14:04:38.908108600	15:45:40.000000000	12:39:47.631294300
$FN.E	14:04:38.908108600	15:59:24.000000000	12:45:24.117676700
$DATA	Present	Missing	Present
$FN	Present	Missing	Present
$SI	Present	Missing	Present
$EA_I	Present	Missing	Present
$EA	Present	Missing	Present

Figure 8.9: Metadata after the 'copy overwrite' operation

	Windows	MacOS	Ubuntu
Filename	b.txt	b.txt	b.txt
LSN	1059247	0	0
SID	266	259	258
$SI.M	14:05:09.861588100	15:45:40.000000000	12:39:52.121538500
$SI.A	14:05:09.849854100	15:45:40.000000000	12:45:24.121676700
$SI.C	14:05:09.849854100	15:59:24.000000000	12:45:24.121676700
$SI.E	14:05:31.895704100	15:59:24.000000000	12:39:52.121538500
$FN.M	14:05:09.861588100	15:45:40.000000000	12:39:52.121538500
$FN.A	14:05:09.861588100	15:45:40.000000000	12:45:24.121676700
$FN.C	14:05:09.861588100	15:59:24.000000000	12:45:24.121676700
$FN.E	14:05:09.861588100	15:59:24.000000000	12:39:52.121538500
$DATA	Present	Missing	Present
$FN	Present	Missing	Present
$SI	Present	Missing	Present
$EA_I	Present	Missing	Present
$EA	Present	Missing	Present

Figure 8.10: Metadata after the 'move other' operation

	Windows	MacOS	Ubuntu
Filename	a.txt	a.txt	a.txt
LSN	1058346	0	0
SID	266	259	258
$SI.M	14:07:18.970044200	15:59:27.000000000	12:45:24.125676700
$SI.A	14:06:56.737218200	15:45:46.000000000	12:45:22.533676700
$SI.C	14:07:18.970044200	15:59:28.000000000	12:45:24.133676700
$SI.E	14:07:18.970044200	15:59:15.000000000	12:39:55.195074600
$FN.M	14:07:18.970044200	15:59:27.000000000	12:45:24.125676700
$FN.A	14:07:18.970044200	15:45:46.000000000	12:45:22.533676700
$FN.C	14:07:18.970044200	15:59:28.000000000	12:45:24.133676700
$FN.E	14:07:18.970044200	15:59:15.000000000	12:39:55.195074600
$DATA	Present	Missing	Present
$FN	Present	Missing	Present
$SI	Present	Missing	Present
$EA_I	Present	Missing	Present
$EA	Present	Missing	Present

Figure 8.11: Metadata after the 'move within' operation

	Windows	MacOS	Ubuntu
Filename	a.txt	a.txt	a.txt
LSN	1058750	0	0
SID	267	259	258
$SI.M	14:06:02.839179000	15:45:40.000000000	12:39:53.654304500
$SI.A	14:06:02.839179000	15:45:41.000000000	12:39:53.674314500
$SI.C	14:06:24.860266500	15:45:43.000000000	12:39:53.674314500
$SI.E	14:06:02.839179000	15:59:25.000000000	12:45:24.121676700
$FN.M	14:06:02.839179000	15:45:40.000000000	12:39:53.654304500
$FN.A	14:06:02.839179000	15:45:41.000000000	12:39:53.674314500
$FN.C	14:06:02.839179000	15:45:43.000000000	12:39:53.674314500
$FN.E	14:06:02.839179000	15:59:25.000000000	12:45:24.121676700
$DATA	Present	Missing	Present
$FN	Present	Missing	Present
$SI	Present	Missing	Present
$EA_I	Present	Missing	Present
$EA	Present	Missing	Present

Figure 8.12: Metadata after the 'rename' operation

	Windows	MacOS	Ubuntu
Filename	b.txt	b.txt	b.txt
LSN	1058390	0	0
SID	266	259	258
$SI.M	14:01:35.839722900	15:45:34.000000000	12:39:44.565762200
$SI.A	14:01:35.839722900	15:45:34.000000000	12:39:44.577768200
$SI.C	14:01:57.905684800	15:45:34.000000000	12:45:24.101676700
$SI.E	14:01:35.839722900	15:45:34.000000000	12:39:44.577768200
$FN.M	14:01:35.839722900	15:45:34.000000000	12:39:44.565762200
$FN.A	14:01:35.839722900	15:45:34.000000000	12:39:44.577768200
$FN.C	14:01:35.839722900	15:45:34.000000000	12:45:24.101676700
$FN.E	14:01:35.839722900	15:45:34.000000000	12:39:44.577768200
$DATA	Present	Missing	Present
$FN	Present	Missing	Present
$SI	Present	Missing	Present
$EA_I	Present	Missing	Present
$EA	Present	Missing	Present

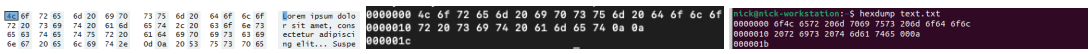
Figure 8.13: Metadata after the 'update' operation

	Windows	MacOS	Ubuntu
Filename	a.txt	a.txt	a.txt
LSN	1058445	0	0
SID	266	259	258
$SI.M	14:02:29.729294300	15:45:34.000000000	12:39:40.175568000
$SI.A	14:02:50.884392200	15:59:21.000000000	12:45:24.097676700
$SI.C	14:02:50.884392200	15:59:21.000000000	12:45:24.097676700
$SI.E	14:02:50.884392200	15:45:34.000000000	12:39:40.187574000
$FN.M	14:02:29.729294300	15:45:34.000000000	12:39:40.175568000
$FN.A	14:02:29.729294300	15:59:21.000000000	12:45:24.097676700
$FN.C	14:02:29.729294300	15:59:21.000000000	12:45:24.097676700
$FN.E	14:02:29.729294300	15:45:34.000000000	12:39:40.187574000
$DATA	Present	Missing	Present
$FN	Present	Missing	Present
$SI	Present	Missing	Present
$EA_I	Present	Missing	Present
$EA	Present	Missing	Present

File content

File content refers to the fundamental bytes that are the actual content of a file. For instance, when saving a picture to a storage medium, the file content comprises the specific bytes representing the pixels. Similarly, when storing a text file, the file content consists of the actual letters forming the text. The responsibility of the file system is to store these bytes in their original form, allowing higher-level programs to interact with them seamlessly. If a driver were to manipulate this binary data, it could render the file unusable for the program. Therefore, we hypothesize that any changes in file content do not stem from driver behavior but are more likely attributed to other factors, such as the operating system and higher-level programs.

Experiment: line endings A clear example of file content differences is line endings. .txt files and other text formats encode single/multiple characters to signify the end of a line. This is commonly known. Regardless, we perform a simple experiment that exemplifies the differences. We saved a text file on all three OSes with a text editor program on all OSes and examined the file's raw bytes. Figure 8.14 shows that the three drive operators we tested



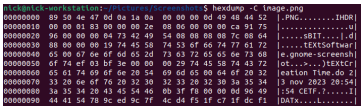
(a) Windows 11 (b) MacOS (c) Ubuntu

Figure 8.14: Line endings for various OSes

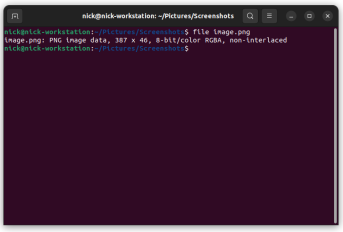
File extensions + magic numbers In addition to line endings, there are more OS-specific hints that a file exhibits. On UNIX-based systems, a file header/magic number, the first 16 bits, often indicate the file type as explained by McDaniel et al. [MH03]. A program uses this

magic number to decode the bytes of the file. Next to magic numbers, an OS has additional hints to determine the file type. The file extension can also be used. It is often used by Windows [MH03]. The file extension is a part of the filename that serves as a hint to the OS with what program this file can be opened. Many programs are OS-specific or are not on a specific OS. Detecting a program's usage also hints at another file's presence. Again, this difference does not originate from a file system driver.

We do a small experiment to show the presence of magic numbers for programs on all OSes.



(a) PNG images start with a 'magic number' to indicate how they should be decoded.



(b) Using the 'file' command to display a file's content type.

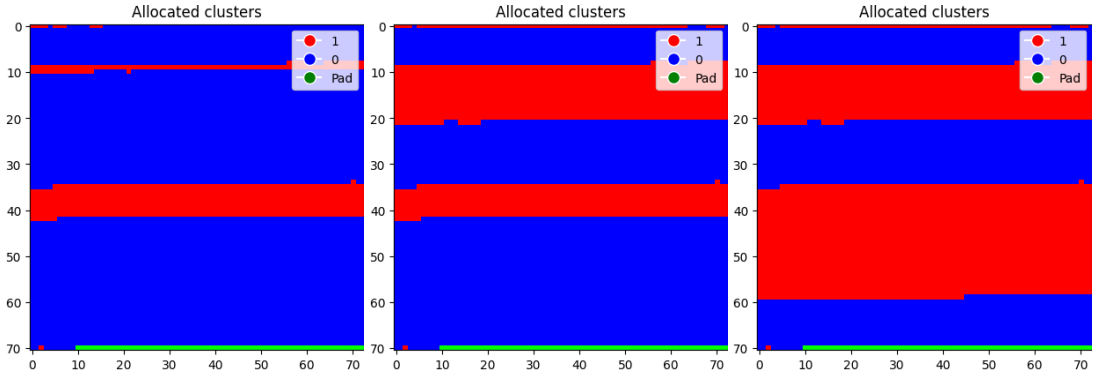
In conclusion, the NTFS drivers we examined consistently return file content without any alterations that can be attributed to the driver. Since preserving file content is a fundamental aspect of a file system, it would be unusual to see significant differences. Nevertheless, distinctions emerge when using different operating systems, primarily influenced by specific programs and text encodings. Notably, Windows, MacOS and Ubuntu employ distinct line endings. Furthermore, identifying a particular file header associated with a file type lacking support on other operating systems may suggest using a specific OS.

ALLOCATION BEHAVIOR

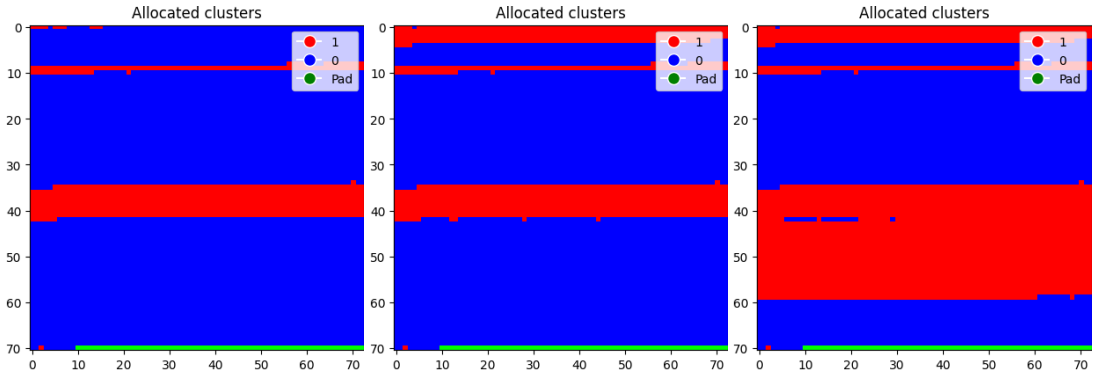
For allocation behavior, we conducted two experiments to get an overview of allocation behavior differences for the various execution environments that we tested. First, we conducted an experiment that allocates a large file on a cleanly formatted & partitioned NTFS driver to get an idea of the fit strategy of all drivers we test. Subsequently, we conducted an experiment that allocates large files on the same volume using different drivers to see how they interact.

Experiment: isolated allocation For this experiment, we formatted an NTFS partition of 50MB and wrote a 5MB file on all three OSes to test the allocation behavior. We render the $Bitmap as a heatmap to visualize the allocated clusters. The results are in figure 8.16. As you can see, all drivers give the clusters right after the MFT, at approximately 2/3 of the file system.

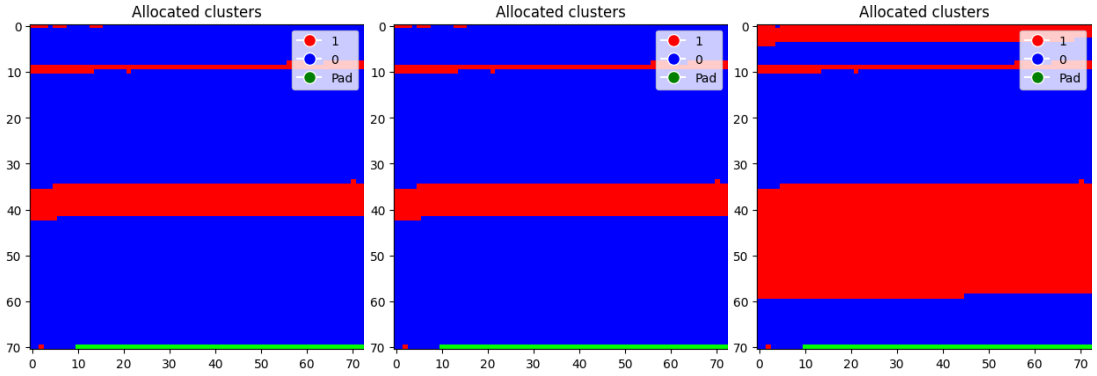
Experiment: mixed allocation We created a 50MB NTFS file system for this experiment and wrote three large files to it such that it spans multiple clusters. For MacOS and Ubuntu, we create a file of 5MB. For Windows, we create a file of 2MB since there was not enough space. We render the $Bitmap as a heatmap to visualize the allocated clusters. 8.17



(a) Baseline (b) Windows, mounted (c) Windows, mounted + allocated

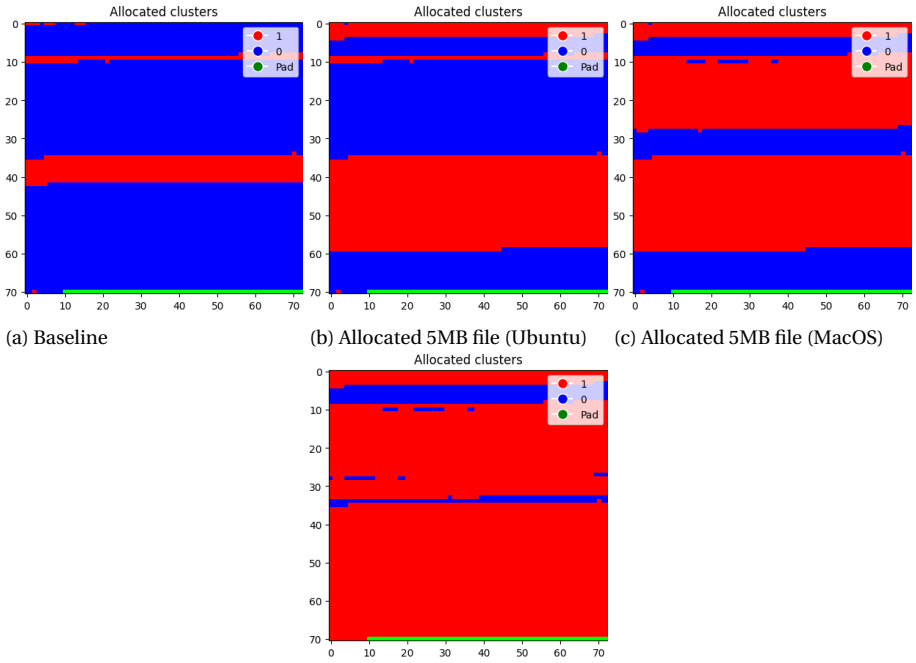


(d) Baseline (e) MacOS, mounted (f) MacOS, mounted + allocated



(g) Baseline (h) Ubuntu, mounted (i) Ubuntu, mounted + allocated

Figure 8.16: Allocated clusters for three OSes



(a) Baseline (b) Allocated 5MB file (Ubuntu) (c) Allocated 5MB file (MacOS)

(d) Allocated 2MB file (Windows)

Figure 8.17: Allocated clusters for three OSes

8.2. RQ2: Detection

Now that we know the differences between driver behavior, we distinguish them based on these differences. To this end, we first establish some rules that identify a driver.

- **Presence of a $LogFile Sequence Nr.** The presence of this metadata artifact is a telltale signature of the Windows NTFS driver. Both the MacOS and Ubuntu drivers don't seem to implement this artifact. Therefore, if a file has this metadata, it is touched by the Windows driver. On the other hand, if it is 0, it warrants a strong indication that the Windows driver did not touch it.

- **SecurityID** The security ID also exhibits telltale signs. For files written by the MacOS, it is 259; for Ubuntu, it is 258. This integer represents a Windows security ID.

- **Timestamps rounded to the second** the MacOS driver exhibits characteristic signs through its timestamps, regardless of the operation performed. All timestamps have a precision of whole seconds. Other NTFS drivers we measured do not exhibit this behavior. Therefore, when encountering a file with timestamps of exact seconds, it is likely touched by MacOS. Of course, a timestamp can be precisely rounded to the second purely by chance.

- **Presence of characteristic files** The presence of the characteristic files 'System Volume Information,' '.fseventsd,' and '$UGM.' can also be used to show that a driver has touched a drive. A simple version of this rule checks for files that have this filename. A more thorough but complex rule is checking if these folders have the expected content, but we consider this an exercise for the reader. It does not advance our point any further.

- **Presence of the $EA and $EA_INFORMATION attributes** these two attributes are only present for Windows and Ubuntu. This means that when we encounter files that have these attributes, it indicates that the MacOS driver did not touch it and that Windows touched it or Ubuntu.

8.2.1. PROOF-OF-CONCEPT IMPLEMENTATION

To show the feasibility of the detection rules discussed in the previous section, we introduce a prototype for driver detection that implements them. It concludes by what OSes, a storage medium, was touched. It returns a response of A where $A = \{x \mid x \text{ is one of Windows, Ubuntu, MacOS}\}$

Some hints are stronger than others. e.g., timestamp modification is a weaker signal than the SecurityID. Why are some signals weaker than others? This is because some Windows NTFS behavior is challenging to mimic for execution environments that know nothing about the Windows environment. A Windows NTFS driver can create a securityID that reflects an actual Group on the Windows system. Also, the Windows driver can add a sensible LSN since it knows that logfile. On the other hand, the other drivers do not have this knowledge, so we consider this a strong signal.

We use this information to provide a solid analysis to the end-user of the detection tool. It will help in court when these arguments are made backed up by data.

Validation To validate and evaluate the effectiveness of the detection tool, we run it on a manually created NTFS drive that reflects actual use. There are no publicly available test images for NTFS that are touched by a non-Windows execution environment.

On this drive, we perform the following operations for all three OSes:

- download a picture and copy it to the drive.
- create an empty .txt file using the command line.
- create a .txt file with content using a text editor.

We use these higher-level programs because a drive is often touched by higher-level programs rather than lower-level ones. They will, in practice, not be touched by a Python script or bash script. Rather, normal people use text editors, photo/video editors, web browsers, and other programs. These higher-level programs have more complex access patterns on a drive than lower-level. They often perform operations that result in a sequence of syscalls that result in drive modification. 8.18

The idea is to simulate some primary use-case of a drive while being simple enough to do with little to no manual effort. Our prototype detection method can use the rules above to show precisely what drivers are used on the drive.

Conclusion As you can see, it indicates the conclusion, arguments for the conclusion, and an indication of its strength. 8.19

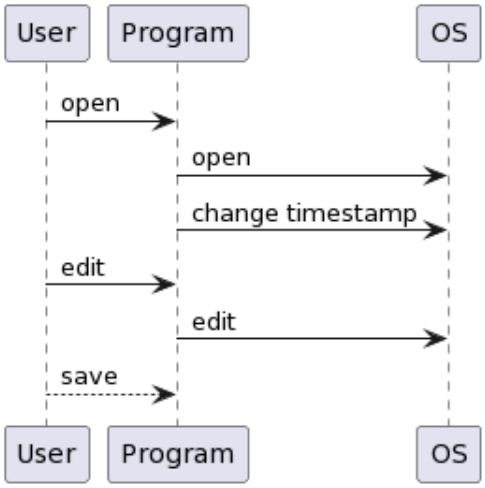


Figure 8.18: User actions on higher-level programs can result in multiple file operations

```
OS              Driver      # hints
--------------  --------  ---------
Windows 11      native            0
MacOS Catalina  Paragon           1
Ubuntu 22.04    ntfs3             0
```

Figure 8.19: The proof-of-concept detection implementation detects a MacOS hint on a drive that was solely mounted on MacOS

9

Discussion

9.1. RQ1: Finding Differences

Now that we performed the experiments, we now discuss them and to what degree they can be attributed to a driver or other effects.

Mounting

The goal of the experiments was to uncover any differences in behavior when mounting.

Windows 11 creates a telltale folder named 'System Volume Information.' Karresand et al. [KAD20] already note the existence of this folder on Windows 10. This folder can be used as a telltale sign of the driver. Other drivers we tested do not exhibit this behavior. Testing for the presence of this artifact on older Windows versions would undoubtedly be worthwhile such that we know it this can be considered a general detection method for Windows.

MacOS with the Paragon driver creates a file and directory upon mounting an NTFS file system.

- .fseventsd cannot be attributed to the Paragon NTFS driver. It is an artifact of MacOS Spotlight [1], an application that helps users search & find files. Work has been done by Atwal et al. [ASLK19] to understand its the structure for forensic purposes.
- $UGM is a file that can be attributed to the driver. The Paragon driver uses this file to map Windows file permissions to UNIX-like file permissions according to a comment in a fork of the Paragon [2] source code [3]. According to the Paragon source code in the kernel, that driver also knows this concept: [4] but we have not found a way to trigger the driver to create it.

For Ubuntu, the driver did not create any new files while mounting.

Notably, these driver-specific files can easily be removed through the file browser on any OS or via command-line tools. Regardless, their presence can be used as a hint.

[1] https://developer.apple.com/library/archive/documentation/Carbon/Conceptual/MetadataIntro/MetadataIntro.html
[2] https://www.paragon-software.com/
[3] https://github.com/ondr3j/paragon-ufsd/tree/master
[4] https://lore.kernel.org/lkml/0911041fee4649f5bbd76cca7cb225cc@paragon-software.com/

METADATA

The experiments conducted for metadata show the following results in summary. Since metadata is stored in file attributes in NTFS, we discuss them grouped by attribute.

- **$STANDARD_INFORMATION** First, timestamps written by the MacOS drivers in attributes are all in whole seconds in terms of granularity. The Ubuntu and Windows files have equal granularity, meaning they are indistinguishable in this regard. Furthermore, the MacOS files' SID is always 259. For Ubuntu, this is 258. The Logfile sequence nr. is never 0 for Windows and is always 0 for Ubuntu and MacOS.

- **$FILE_NAME** First, timestamps written by the MacOS driver in attributes are all in whole seconds in terms of granularity. The Ubuntu and Windows files have equal granularity, meaning they are indistinguishable in this regard. Apart from fiwalk, we also used ntfsinfo to cross-check the results , and it turns out that this tool returns some additional fields that the other tools did not offer. Our tests have shown that Windows files always have namespace: 'Win32 & DOS' and MacOS and Ubuntu files have namespace: 'POSIX' [5].

- **$EA** This attribute was only present for Windows & Ubuntu. Therefore, based on our experiments, a file with these artifacts is unlikely to stem from a MacOS file operation.

- **$EA_INFORMATION** This attribute was only present for Windows & Ubuntu. Therefore, based on our experiments, a file with these artifacts is unlikely to stem from a MacOS file operation.

[5] http://dubeyko.com/development/FileSystems/NTFS/ntfsdoc.pdf

File content

Differences in file content do occur. In terms of line endings, Windows handles this differently than MacOS and Ubuntu. The same holds for specific file types and file headers. Encountering content that can only be written by one particular user-level program on another OS is a clear hint of that OS. McDaniel et al. [MH03] introduce file content detection on a statistical analysis of allocated clusters for forensic purposes.

While there are differences in file content, they can only be attributed to user-level programs. So, the differences we encountered cannot be attributed to a driver. Instead, they are the result of the workings of a user-level program.

Allocation behavior

Regarding allocation behavior, all three DOs we tested allocated the clusters right after the MFT at around 66% of the logical address space. The experiment aimed to spot any obvious differences in the allocation algorithm. Based on our experiments, it is insufficient to draw any conclusions on fit strategy or whether these artifacts can distinguish drivers from one another.

To conclude, no clear-cut allocation behavior differences can be used for detection. Our experiments show no apparent differences in fit strategy, but more research is needed.

1. SID is non-zero
2. LSN is always non-zero
3. Timestamps have a granularity of 100 nanoseconds
4. Attributes $\$EA$ and $\$EA\_INFORMATION$ present

Figure 9.1: Windows metadata fingerprint

9.2. RQ2: DETECTION

Now that we have found the driver fingerprints, we discuss to what degree they can be used to show the presence of a specific driver.

Mounting For both Windows and MacOS, uniquely named files and folders are created on the drive as soon as it is mounted. First, for Windows, the 'System Volume Information' folder is characteristic. Given this information, we establish a rule that is the first part of the Windows driver fingerprint. Note that the rule can also be used in negated form; the absence of the file is also a hint. 9.1

$$windows(d) = \begin{cases} 1 & \text{if } d \text{ has a folder named 'System Volume Information',} \\ 0 & \text{otherwise.} \end{cases} \tag{9.1}$$

Second, for MacOS we establish a similar rule. 9.2. Since the source code of the Paragon driver shows that it can also create the '$UGM' file [6], we do not consider this file as telltale for the MacOS Paragon driver.

$$mac\_os(d) = \begin{cases} 1 & \text{if } d \text{ contains a folder named '.fseventsd',} \\ 0 & \text{otherwise.} \end{cases} \tag{9.2}$$

While the Ubuntu driver does not create any characteristic files, this very fact can be used for its fingerprint. We therefore introduce the following rule. 9.3

$$ubuntu(d) = \begin{cases} 0 & \text{if } d \text{ contains a folder named '.fseventsd' or 'System Volume Information',} \\ 1 & \text{otherwise.} \end{cases} \tag{9.3}$$

Metadata All OSes we tested exhibited unique behaviors in terms of how they write metadata. We now discuss to what degree these are characteristic and introduce rules to detect them. Since our NTFS parser [7] [8] does not expose the 'namespace' for a file, and we did not implement this for our detection method.

For Windows, we establish the following rules. 9.1

For MacOS, we establish the following rules. 9.2

For Ubuntu, we establish the following rules. 9.3

[6] https://lore.kernel.org/lkml/0911041fee4649f5bbd76cca7cb225cc@paragon-software.com/
[7] https://github.com/msuhanov/dfir_ntfs
[8] https://www.sleuthkit.org/

1. SID is always set to 259
2. LSN is always 0
3. Timestamps are rounded to the whole second
4. Attributes $\$EA$ and $\$EA\_INFORMATION$ missing

Figure 9.2: MacOS metadata fingerprint

1. SID is always set to 258
2. LSN is always 0
3. Timestamps have a granularity of 100 nanoseconds
4. Attributes $\$EA$ and $\$EA\_INFORMATION$ present

Figure 9.3: Ubuntu metadata fingerprint

File content Based on our experiments, no conclusive evidence exists of differences between how drivers write files to a drive that can be attributed to the driver or the OS. Rather, we attribute them to user-level files since the file headers, line endings, and other file content are all well-known phenomena. We, therefore, omit this information from the driver fingerprints.

Allocation behavior The experiments we performed do not uncover any signs that we deem fit to serve as a fingerprint for the NTFS drivers. This does not mean there are none.

Conclusion To conclude, the Windows and MacOS drivers generate telltale files and folders when mounting a drive, and all drivers also have detectable differences with how they operate on metadata. To conclude, we identify that metadata is the most promising aspect of driver differences. File content is different for OSes, but not as a result of drivers but rather other programs. For allocation, more work is needed. When a driver performs a file operation, it generates telltale differences that can be used to show the presence of the OS, both on the file state and on other parts of the disk.

10

CONCLUSIONS & RECOMMENDATIONS

10.1. CONCLUSIONS

Mounting NTFS drivers handle mounting differently. After mounting, the Windows and MacOS drivers create telltale files on the storage medium that can be used to show the usage of these drivers. Similarly, the absence of these files can also be used to show that these drivers were not used.

Metadata When identifying telltale signs, metadata stands out as the most promising artifact. Numerous aspects of metadata display unique characteristics that are a telltale sign of the driver that wrote them. Our conducted experiments and detection prototype show that each operating system can be reliably distinguished even on a drive utilized across three distinct operating systems featuring a range of programs, media types, and drivers. In summary, the following differences can be used to distinguish NTFS drivers.

- **$LogFile Sequence Nr.**
- **SecurityID**
- **Timestamps rounded to the second**
- **Presence of the $EA and $EA_INFORMATION attributes**

File content There are no differences in terms of file content between different NTFS drivers. Although programs running on the same OS will produce telltale signs like line endings, magic numbers, and file extensions, the drivers treat file content the same. This is to be expected since this is precisely the job of a file system: persist file content. To conclude, file content cannot be used as a telltale sign for the usage of a specific driver.

Notably, this does not mean there are no telltale signs. They are less clear-cut than the other artifacts we discuss. Some drivers may have subtle bugs in how they handle corner cases. e.g., they store huge or small files differently, and this is a difference that can be tested. But not as far as we know: we have not found them.

Allocation behavior Drivers differ in allocation behavior, but distinguishing them based on this behavior is not trivial. The exact state of the disk has to be known to determine with what fit strategy a cluster was allocated and this information cannot be retrieved in most cases.

However, this does not mean that this artifact is a dead-end. Byte-frequency analysis or other statistical methods could help in using this artifact for driver detection. This has parallels with the work by McDaniel et al. [MH03] where they use similar techniques for file content-type detection.

Conclusion To conclude, file metadata exhibits many telltale signs that can be used for detecting if a particular NTFS driver has touched the partition or not. This insight greatly helps practitioners who use this information to gather more information about a case and other researchers willing to extend this work of driver behavior modeling using black-box testing.

Metadata contains the most telltale signatures and is, therefore, more detectable. Elements such as timestamps, security ID, USN journal number, and $LogFile Sequence Number constitute distinctive metadata components of a file, each handled uniquely by different NTFS drivers.

These differences are explicit signs that forensic software and practitioners should use to identify a specific driver. This method can be used to extract previously untapped evidence from storage devices.

10.2. RECOMMENDATIONS

Automata learning A more rigorous testing approach should be employed to encompass a diverse range of NTFS driver differences and attain a more thorough understanding of these differences. Automata learning is a promising but involved technique for this purpose. While this work takes a lot of steps towards the application of automata learning to file system driver testing, there are still existing challenges that require attention.

Primarily, the state-explosion problem demands the creation of an abstract representation for an NTFS file system. This step is crucial in reducing the number of states generated during the execution of automata learning.

Flexibility It is infeasible to test every single driver, OS, and file operation in advance. Rather, this method should be used as a stepping stone for generating driver fingerprints 'on-the-fly'. If forensics practitioners encounter specific scenarios in practice that raises their suspicion on the use of other drivers they should be able to launch a series of experiments using this technique, such that their fingerprints can be generated and subsequently detected on the respective storage medium.

More drivers While this work identifies telltale signs of the tested drivers, there are more NTFS drivers. Up until Linux kernel patch 5.15 the 'ntfs-3g' driver by Tuxera [1] was available for UNIX-based systems. In addition, Tuxera also sells an NTFS driver for MacOS. We cannot estimate to what degree these drivers are used. Nor can we estimate to what degree they differ in terms of the behavior of drivers we tested.

Version detection This work should be extended to cover other use cases. Detecting a specific driver version would help squeeze even more digital traces out of a storage medium. Using our technique, if two unique computers with the same driver have touched a storage medium, they are indistinguishable from one another. However, if driver versions are detectable, they can be distinguished.

Timelines Apart from saying what driver has touched this drive, this can also be used to create a timeline of events (when a drive was mounted and with what driver + timestamps) by also using the timestamps in the metadata and other hints in the metadata like $UsnJournal and $LogFile. Bouma et al. have employed a similar technique for exFat timestamps. This same method of reasoning backward is applicable to this work. However, more data on file timestamps for NTFS has to be gathered for this to apply.

[1] https://tuxera.com

11

REFLECTION

A BLACK BOX IS A BLACK BOX

The most obvious side note to this work is that the method is black-box testing. For reasonably sized programs, it is impossible to test all code paths and this means we are unable to test all behaviors. The real question is if the black-box testing can be assumed to cover a reasonable amount of behavior that overlaps a lot with actual driver behavior in practice.

THE NOTION OF FILE OPERATION IS INSUFFICIENT

The concept of file operation is inadequate and vague. What is a file operation? The idea of file operation differs per operating system. In Unix-like systems files are accessed through POSIX syscalls, so for Ubuntu and MacOS, they are equal. However a one-to-one mapping cannot be made to Windows. Windows has its API for interacting with the file system. It is essential to realize what 'file operations' lead to what syscalls (in Windows: windows API functions) and, in turn, what syscalls lead to what drive changes. When doing experiments, the actual syscalls performed on a file should be monitored to determine what 'input' the driver receives.

SIMULATING FILE SYSTEM USE IS HARD

Simulating realistic file operations is challenging. Our experiments rely on bash/batch/python scripts to perform some file operations automatically on the system under test. However, actual use cases are more complicated. Files are often interacted with through higher-level programs like text/code editors, image editing programs, and web browsers. These programs use more complex access patterns than one file operation. Word has a 'file tunneling' feature that writes data to a new file and deletes the old file to prevent file corruption [BJvdMVDA23]. These usage patterns can create more complicated file states we have not tested with our method.

PARSERS ARE UNRELIABLE

This method relies on NTFS parsing to work. The very reason for doing this work is based on the fact that multiple drivers implement NTFS differently. In fact, not a single one of them is identical. In turn, we use an NTFS parser to gather results. The NTFS parser that we use might be incomplete. This incomplete implementation might lead to inconsistencies

in results. We cross-check the results with other forensic tools to reduce the risk of this occurring.

VIRTUAL ENVIRONMENT

The experiments were performed in a virtual environment, in VirtualBox. A virtual environment virtualizes the underlying machine to work.

While virtualization aims to simulate the underlying hardware and is successful in doing so in many cases, it differs from the practical use of modern drives. In this regard, this could result in certain scenarios differing from practice.

Second, it adds an extra moving part that unnecessarily complicates experiments. As noted earlier, making sure bytes reach the physical drive is hard and it is necessary to do so because we want to parse the file system without mounting to prevent interfering with experiments. Adding extra software layers only introduces complexity. The design decision to use VMs was mainly such that we have a uniform interface for all experiments and require less platform-specific automation. Furthermore, it also means that we can do the drive snapshotting on one environment for all experiments. We now conclude that these advantages do not outweigh the cons. The experiment setup can definitely be improved in this regard.

ANTI-FORENSICS TECHNIQUES

Anti-forensics is hiding traces on digital artifacts so they are unusable by forensics practitioners. [Kes07] Defenders can overwrite (parts of) files, delete them through regular file operations file operations, or use advanced anti-forensics tools like "Timestomp" [1] to alter timestamps such that the use of anti-forensics is not apparent. Every single byte on a disk can be changed, and our method relies on these bytes. So, in principle, if a defender can mask all signals we use for detection, they hide their steps successfully. On the other hand, this requires that they know every single signal we test for. So, in practice, this is unlikely to pose a general issue for this method.

SOURCE-CODE ADDS SEMANTICS

Differences in drivers arise for various reasons: different interfaces of the underlying OS, bugs, alternative design decisions, and more. Any structural design decisions that result in unique behavior are unlikely to change in the future. For this reason, consulting the source code is a useful endeavor. Although we initially dismissed this approach, we occasionally resorted to exploring the available source code as a means of gaining deeper insights into the rationale behind specific driver behaviors and estimating the likelihood of future changes.

MANUAL CONSTRUCTION OF DRIVER FINGERPRINTS

We manually constructed rules that constitute a driver fingerprint based on the differences we found. This requires manually observing all results and looking for patterns. This is necessary since comparing artifacts is a non-trivial task: Timestamps, for example, are almost always unequal because they are so fine-grained. This technique works but is error-prone and inefficient at a large scale. Rather, we should devise an automated technique that is

[1] http://forensicswiki.org/wiki/Timestomp

able to interpret driver differences as being meaningful or not by generating i.e., an abstract model such that two artifacts can be meaningfully compared.

BIBLIOGRAPHY

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987. ISBN: 0890-5401 Publisher: Elsevier. 15

[ASLK19] Tajvinder Singh Atwal, Mark Scanlon, and Nhien-An Le-Khac. Shining a light on spotlight: Leveraging apple's desktop search utility to recover deleted file metadata on macos. *Digital Investigation*, 28:S105–S115, 2019. 42

[BJvdMVDA23] Jelle Bouma, Hugo Jonker, Vincent van der Meer, and Eddy Van Den Aker. Reconstructing Timelines: From NTFS Timestamps to File Histories. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, pages 1–9, 2023. 19, 22, 23, 31, 50

[CHK+08] Alexei Czeskis, David J. St Hilaire, Karl Koscher, Steven D. Gribble, Tadayoshi Kohno, and Bruce Schneier. Defeating Encrypted and Deniable File Systems: TrueCrypt v5. 1a and the Case of the Tattling OS and Applications. In *HotSec*, 2008. 10

[Cho13] Gyu-Sang Cho. A computer forensic method for detecting timestamp forgery in NTFS. *Computers & Security*, 34:36–46, 2013. ISBN: 0167-4048 Publisher: Elsevier. 11

[CKNZ11] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30. Springer, 2011. 15

[GL21] Michael Galhuber and Robert Luh. Time for truth: Forensic analysis of ntfs timestamps. In *Proceedings of the 16th International Conference on Availability, Reliability and Security*, pages 1–10, 2021. 11

[KAD19] Martin Karresand, Stefan Axelsson, and Geir Olav Dyrkolbotn. Using ntfs cluster allocation behavior to find the location of user data. *Digital Investigation*, 29:S51–S60, 2019. ISBN: 1742-2876 Publisher: Elsevier. 12

[KAD20] Martin Karresand, Stefan Axelsson, and Geir Olav Dyrkolbotn. Disk Cluster Allocation Behavior in Windows and NTFS. *Mobile Networks and Applications*, 25(1):248–258, February 2020. 12, 42

[KCS17] Michal Kedziora, Yang-Wai Chow, and Willy Susilo. Defeating plausible deniability of VeraCrypt hidden operating systems. In *International Conference on Applications and Techniques in Information Security*, pages 3–13. Springer, 2017. 10

[KDA20] Martin Karresand, Geir Olav Dyrkolbotn, and Stefan Axelsson. An Empirical Study of the NTFS Cluster Allocation Behavior Over Time. *Forensic Science International: Digital Investigation*, 33:301008, July 2020. 14

[Kes07] Gary C. Kessler. Anti-forensics and the digital investigator. 2007. Publisher: School of Computer and Information Science, Edith Cowan University, Perth 11, 51

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955. ISBN: 0005-8580 Publisher: Nokia Bell Labs. 15

[MH03] Mason McDaniel and Mohammad Hossain Heydari. Content based file type detection algorithms. In *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, pages 10–pp. IEEE, 2003. 36, 37, 44, 48

[MK19] Alji Mohamed and Chougdali Khalid. Detection of Timestamps Tampering in NTFS using Machine Learning. *Procedia Computer Science*, 160:778–784, January 2019. 11

[NA22] Rune Nordvik and Stefan Axelsson. It is about time–Do exFAT implementations handle timestamps correctly? *Forensic Science International: Digital Investigation*, 42:301476, 2022. ISBN: 2666-2817 Publisher: Elsevier. 11, 17

[NTA19] Rune Nordvik, Fergus Toolan, and Stefan Axelsson. Using the object ID index as an investigative approach for NTFS file systems. *Digital Investigation*, 28:S30–S39, 2019. ISBN: 1742-2876 Publisher: Elsevier. 11

[PB20] David Palmbach and Frank Breitinger. Artifacts for detecting timestamp manipulation in NTFS on windows and their reliability. *Forensic Science International: Digital Investigation*, 32:300920, 2020. ISBN: 2666-2817 Publisher: Elsevier. 11

[Rog06] M. Rogers. Anti-forensics: the coming wave in digital forensics. *Retrieved September*, 7:2008, 2006. 3

[Tay22] Petroc Taylor. Amount of data created, consumed, and stored 2010-2020, with forecasts to 2025. *Statista. Available online: https://www. statista. com/statistics/871513/worldwide-data-created/(accessed on 24 October 2023)*, 2022. 2

[TM22] Aurélien Thierry and Tilo Müller. A systematic approach to understanding MACB timestamps on Unix-like systems. *Forensic Science International: Digital Investigation*, 40:301338, April 2022. 11

[Val96] Antti Valmari. The state explosion problem. In *Advanced Course on Petri Nets*, pages 429–528. Springer, 1996. 15

[vdMJvdB21] Vincent van der Meer, Hugo Jonker, and Jeroen van den Bos. A contemporary investigation of NTFS file fragmentation. *Forensic Science International: Digital Investigation*, 38:301125, 2021. ISBN: 2666-2817 Publisher: Elsevier. 11