Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Jostein Magnussen-Vik

Timeframe-based contiguous file carving in video surveillance systems

Master's thesis in Information Security
Supervisor: Dr. Kyle Porter
Co-supervisor: Dr. Fergus Toolan
June 2023

NTNU
Norwegian University of
Science and Technology

Jostein Magnussen-Vik

Timeframe-based contiguous file carving in video surveillance systems

Master's thesis in Information Security
Supervisor: Dr. Kyle Porter
Co-supervisor: Dr. Fergus Toolan
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Abstract

Surveillance video is often a crucial piece of evidence in criminal investigations. It is time critical to collect the video data before it is made unavailable due to privacy policies or hard drive storage capacities. The video data is often deleted within a week or so, and if it's not collected by then, the evidence might be lost. Additional evidence may surface during the investigation in certain criminal cases, and retrieving deleted surveillance videos is often crucial.

The recovery of deleted data, data from unallocated areas of the hard drives, is commonly known as carving. There are several methods of carving data; searching for a header-footer signature is one of the most basic methods and is suitable for files with a recognizable signature to determine the start and end of the file. Another reliable method is frame-based carving, which involves searching for the smallest parts of a video (frames) and reconstructing the video. This method can effectively handle fragmented and partly overwritten files. However, when it comes to retrieving surveillance video, it may not always be successful due to the unique file formats used by surveillance systems.

The surveillance systems store timestamps along with the video data to help determine when an incident occurred. And while the systems are running, they are constantly storing new video data, and either by privacy policies or to prevent reaching storage capacity, the system deletes older video data.

We propose carving methods for several surveillance systems, including Milestone, Mirasys, Avigilon, and Detec, that recover video data within a time-frame of interest. By identifying patterns surrounding timestamps in the video data, our methods search for those patterns with a regular expression. The regular expression matches a range of timestamps to include a time frame of interest. In the data surrounding the timestamps, we find information to carve out the corresponding frame data. Our results indicate that the methods have very high precision and recall values for retrieving old video data not yet overwritten.

We also present a method and a tool to discover patterns of timestamps, signatures, offsets, and size information within the video data. We use this method in our video data analysis and present these patterns as part of developing the carving algorithms. We hope this method and tool may lay the foundation for recognizing patterns in other systems.

Sammendrag

I straffesaker er ofte overvåkingsfilm en viktig del av bevisbildet. Det er viktig å få innhentet opptakene så tidlig som mulig på grunn av krav om sletting og lagringsplass på overvåkingssystemene. På grunn av personvernregler blir ofte opptak slettet etter en uke, og da kan bevis forsvinne om det ikke er innhentet innen det. I løpet av en etterforsking kan det også dukke opp nye bevis som gjør nye områder eller tidsrom aktuelt for innhenting av overvåkingsmateriale, og da kan gjenoppretting av slettede opptak være avgjørende.

Gjenoppretting av slettede data fra uallokerte områder på en harddisk er kjent som 'carving'. Det finnes flere metoder for å gjenopprette data, og en av de mer grunnleggende er å søke etter kjente start- og slutt-signaturer i en fil. For denne er det viktig at filtypen har en definert start og slutt. En annen utprøvd metode er bilde-basert gjenoppretting. Denne leter etter de miste bestanddelene i en videofil, enkeltbilder, og gjenoppretter videoen ved å sette samme alle enkeltbildene. I tillegg kan denne metoden håndtere filer som er delvis overskrevet, samt filer som er lagret på spredte plasser på harddisken. Men, disse generiske metodene kan ha problemer med å gjenopprette overvåkingsmateriale på grunn av at dette ofte lagres i proprietære filformater.

For all videodata som lagres er det svært viktig å lagre tidspunkt sammen med videoen, slik at det er mulig å tidfeste en hendelse. Og mens overvåkingssystemer kjører lagrer de en konstant strøm med ny videodata, og sletter gammel data fortløpende, enten på grunn av personvernhensyn eller for å unngå at lagringsplassen går fullt.

Vi presenterer metoder som gjenoppretter overvåkingsmateriale fra flere overvåkingssystemer, inkludert Milestone, Mirasys, Avigilon og Detec, der gjenopprettingen fokuserer på et gitt tidsrom. Ved å finne gjentakende mønster omkring tidsstempler i videodataene, lages et søkemønster med mulighet for å definere et aktuelt tidsrom for å søke etter videodata. I de data som ligger omkring tidsstemplene har vi funnet nok informasjon om størrelser og avstander til å gjenopprette de tilhørende videodata. I våre resultater virker metodene til å være lovende med svært høye verdier for presisjon og gjenkallelse.

I tillegg har vi utviklet en metode og et verktøy for å analysere og gjenkjenne mønster i videodata, disse mønstrene kan bestå av tidsstempel, signaturer, størrelse- og avstandsinformasjon. I vår analyse av videodata fra de forskjellige overvåkingssystemene ble denne metoden brukt til å lage gjenopprettingsmetodene for slettet overvåkingsfilm. Vi håper at denne analysemetoden og verktøyet kan danne grunnlaget for å gjenkjenne mønster i andre lignende systemer.

Contents

Figures

Tables

Code Listings

Acronyms

CCTV Closed-Circuit Television. 3, 15

CSV Comma-Separated Values. 16

DVR Digital Video Recorder. 3, 8, 15

EBML Extensible Binary Meta Language. 37

HEVC High Efficiency Video Coding. 7

IDR Instantaneous Decoder Refresh. 6, 7

leb128 Little Endian Base128. 30, 32

NAL Network Abstraction Layer. 6, 7, 10

NVR Network Video Recorder. 8

P/B Predictive or Bi-Directional. 6, 7

PPS Picture Parameter Set. 6, 7, 10

SPS Sequence Parameter Set. 6, 7, 10

STSZ Sample-to-Size. 10

VMS Video Management Software. 8, 36, 39, 51–55

VOP Video Object Plane. 6

Chapter 1

Introduction

Using video surveillance as evidence in criminal cases is highly relevant due to its presence and objectivity, the common reason for installing such systems is to detect abnormal activity. But there is also privacy policies that limits the period of time we can store the surveillance data, a common limit is one or two weeks. This means that after this limit the surveillance video is lost, unless it is possible to recovered the deleted data.

The most basic methods for recovering data are signature-based carving techniques that look for distinct headers and footers of a file. For instance, AVI video files always start with the ASCII characters "RIFF" followed by four bytes that indicate the file's size. MPEG files, on the other hand, begin with four hexadecimal bytes `0x00 00 01 BA` and end with `0x00 00 01 B9`. This information allows for the complete recovery of the files.

Digital video data is stored using various codecs, each with different characteristics. Some codecs offer high compression rates, others prioritize lossless compression, and some are designed for network streaming. IP cameras typically use MPEG-4 and MJPEG codecs, although H.264 has become more prevalent in recent times [1]. The codecs store the video data internally in smaller parts called frames, and each frame has a recognizable header. Frame-based recovery reconstructs video files by identifying the signature of frames and placing them in the correct sequence [2].

Surveillance data is only useful with information about the recording time. A video showing a person walking through the backdoor of a building has no value unless we know when it happened; it might be the suspect we are looking for if we know that it happened in the middle of the night. Therefore, surveillance systems typically store metadata along with the frames, including timestamps, index numbers, frame sizes, and other relevant information for the system's use. The additional data can render traditional recovery tools ineffective as they cannot identify this particular type of data. Our research will analyze these metadata and propose methods to carve for surveillance video. Also, since this metadata contains timestamps and other information, it is possible to specify a relevant timeframe for recovery.

We choose the surveillance systems to analyze by looking at the leading providers[3] and doing an Internet search for popular systems. We will analyze Milestone, Avigilon, Mirasys, Detec, and Agent-DVR systems.

1.1 Research questions

RQ1 - Is it possible to recover surveillance video that is partially overwritten using a timeframe-based approach?

We aim to find methods to recover surveillance data that are deleted and potentially partially overwritten by newer surveillance data and specify which time frame we are recovering data within. By analyzing the video data produced by surveillance systems, we aim to identify structures in the data, focusing on timestamps to narrow down and make the recovery process more effective.

RQ2 - Can existing common carving tools recover surveillance video?

We want to analyze the performance of existing carving tools on surveillance video. Is the surveillance data stored in a manner that allows for deleted data to be retrieved using these tools?

RQ3 - Is it possible to find a generic pattern for stored data of surveillance systems?

Once we have identified the structures within the surveillance data, we aim to identify any common patterns that may exist. Surveillance systems share certain properties, such as timestamped recordings, locating specific recordings by time, and skipping to a particular time when playing back a recording. These similarities may help us recognize generic patterns.

1.2 Thesis structure

This thesis is divided into seven chapters. In chapters 2 and 3, we will present some background information and related work. The fourth chapter presents the experiment of recording video data with different surveillance systems and how we create the data we will try to recover. In chapter five, we present the results of the experiments and later discuss them in chapter six. In the last chapter, we present our conclusion and further work.

1.3 Contributions

Our major contributions in this thesis are carving methods for video data produced by several surveillance systems, including Milestone, Mirasys, Avigilon, and Detec. The carving methods focus on recovering data within a given time frame of interest. They are working on a low level to recover the smallest non-overwritten parts of the video data. We also present a method and a tool to highlight patterns within video data. Using this tool, we could discover the internal structures and create the carving methods.

Chapter 2

Background

2.1 Filesystems

To record videos in CCTV systems a Digital Video Recorder (DVR) is responsible for storing the data, and there must be an underlying filesystem to keep track of the files in the system. DVR systems that are delivered as software are installed on computers or servers and the data storage is done in standard filesystems such as FAT, NTFS, Ext4, ExFat, etc. On the other hand, an all-in-one box DVR system, with a proprietary operating system, will often use a proprietary filesystem for data storage. And the filesystem might store the video data in streams and channels, instead of files and folders [4]. This research will focus on the software systems using standard filesystems, but uses similar methods of reverse engineering the data from proprietary systems.

When describing a filesystem and the possibility of recovering deleted data, the filing cabinet analogy is often used. A card index has information on all the files in the system, and the content of the files is stored in drawers in the filing cabinet. Each index card has information about where the content of the file is located. By knowing the name of the file, it is possible to retrieve the content of a file [5]. The smallest addressable storage unit of a filesystem can be referred to as a cluster, and a cluster is made up of one or more sectors, the smallest allocation unit on a hard drive. In the filing cabinet analogy, a cluster is represented by a drawer, and sectors are compartments in the drawer. If a file is larger than a drawer, the file is stored over multiple drawers, preferably consecutive. A drawer or a cluster can only contain allocated data from one file in the card index.

The file's metadata, including timestamps, access rights, directory hierarchy, and more, is stored in a separate database in the file system, the card index, using a small part at the beginning of the storage space. Without this database, it is almost impossible to recover information or data from a file in the filesystem; similar to a blueprint map of a building, the map contains information on which rooms contain what information and which rooms are unoccupied [5].

2.1.1 Deleting a file

In the filing cabinet analogy, when deleting a file, the index card would first only get a mark as deleted, but the rest of the card and the file content is untouched, and the file will not be visible to the user. In most filesystems, it would be an easy task to recover this file since the index card and file content are intact. After a while, the map of the drawers will update and mark the drawers/sectors as unoccupied and available for other files to use for storage. The entry of the index card is also wiped after a while to make room for new files and their information [6].

2.1.2 FAT

In FAT (File Allocation Table) file systems, the information on which clusters are occupied or not is stored in the 'file allocation table', and the file metadata information is stored in the 'directory table'. The file allocation table contains one entry per cluster in the filesystem, and the directory table contains one entry per file or folder in the filesystem. Each file entry in the directory table points to the starting cluster of the file, and the clusters entry in the file allocation table contains a pointer to the next cluster, which in turn points to the next cluster all to the end of the file where the clusters entry is marked with an EOF (end of file)[7].

Deleting a file will, in the filing cabinet analogy, only put a mark on the index card; in FAT, the file would first only be marked with a special first character in the file name. Then set all entries belonging to the file in the FAT to unoccupied clusters. This removes the pointers to the cluster locations of the file, making the recovery of a file difficult since the file metadata is unattached to the file content, where the pointers from one cluster to another are also missing [8].

2.1.3 NTFS

The blueprint map is a bit simpler in NTFS (New Technology File System), where each cluster is represented by only one bit, called a bitmap. Each bit can only indicate whether a cluster is allocated to a file or unallocated and available [9].

The card index is maintained by MFT (Master File Table), and each file entry has information about all the allocated clusters for the file. As in FAT, the file entry is only marked as deleted, the file content is left untouched for now, and the cluster is marked unallocated in the bitmap [10]. But the cluster pointers are untouched, making recovery far less difficult for files in NTFS file systems.

2.1.4 Ext4

In Ext4, the smallest addressable units are called blocks, similar to drawers in the filing cabinet. And a range of blocks can be addressed in an extent. Similar to the bitmap in NTFS, Ext4 also has a bitmap over the blocks, the Data Block Bitmap [11].

Each card in the card index is known as an inode in Ext4, and the card index is the inode Table and the inode Bitmap. The inode can directly reference up to twelve extents in the inode itself and even more extents through indirect addressing, where blocks are used to store extent references. As in the other file systems, deleting a file will mark the file as deleted, set a timestamp for when the file was deleted, and update the bitmaps for the blocks and inode to make the space unallocated. In addition, the critical inode data is zeroed out: the file size and the extent references, making recovery of a file using the inode information difficult.

2.2 File recovery

2.2.1 Deleted file recovery

As a file is marked as deleted in a FAT or NTFS filesystem, no actual data is altered, and metadata and file content are intact. The operating system is simply not showing the deleted files. Retrieval software could be used to show the files marked as deleted, and the user can choose to recover the file, i.e., removing the deleted marking [5].

2.2.2 Carving signatures

Media formats and other file formats are often stored in a structured manner, resulting in recognizable patterns of data within the file data. These patterns can be used as signatures to identify the start and end of a file. For example, a JPEG file has the pattern `0xFF D8` as the start signature and `0xFF D9` as the end signature. This allows us to search for those patterns to recover JPEG files [12].

This recovery method is useful when the files cluster information or the file table is missing and for files with a known header signature and size information or footer signature. This method is known as signature-based file carving and is used for media files, office files, compound files, databases, etc., with a structured body. Some of the most used tools for this method are Scalpel, Foremost, and PhotoRec [13].

2.2.3 Structure based carving

Knowledge of the internal structure of files can be used to create methods of carving for those structures. Some types of files, such as zip files, document files, and video files, have specific internal structures, which make structure-based carving an effective approach [14]. The internal structures contain information about offsets and the size of blocks of data that can be carved contiguously or individually and put together afterward.

2.2.4 Fragmentation

As mentioned earlier files are preferably stored in consecutive clusters, but this is not always possible. Files that are larger than the biggest space of consecutive clusters must be split over multiple sets of consecutive clusters, resulting in a fragmented file [15]. The file might be scattered across different parts of the hard drive, but the file system keeps track of the location and correct order of the clusters. Fragmentation makes recovery of a file difficult when the cluster information is missing. We will discuss methods for recovery of fragmented files in Chapter 3.

2.3 Codecs

The most used video stream standards used in digital cameras have for several years been MPEG-4, MJPEG, and H.264 [16], and in the latest years, H.265 has also become popular.

2.3.1 MJPEG

The JPEG standard describes a set of compression standards for still images, and compressing each frame in a video into a series of JPEG images is often referred to as Motion-JPEG (MJPEG) [17]. Many of the parameters for the images don't change from frame to frame, such as aspect ratio and color space; these parameters can be described only once in a while for the video stream. But some video-specific parameters such as framerate and interlacing must be defined outside the JPEG standard. The JPEG File Interchange Format (JFIF) used within JPEG files describes these parameters. This would mean that the MJPEG streams don't fully follow the JFIF standard [16].

The Motion JPEG (MJPEG) video protocol consists of a series of JPEG images. And the most interesting markers in JPEG for carving purpose is the start and end of a JPEG image `0xFF D8` and `0xFF D9` [18]. Investigating the MJPEG stream from a camera shows that the JPEG images are streamed back to back without any additional information.

2.3.2 MPEG-4

The ISO/IEC 14496-2:2004 [19] defines the MPEG-4 Visual codec. Where MPEG-4 is a container for different codecs, the MPEG-4 Visual is the codec defined. Frames, or portions of frames, are described in a Video Object Plane (VOP). Intra and predicted VOPs are used to describe the frames; an intra-VOP draws a full frame, and a predicted VOP draws a frame using different prediction methods in reference to intra-VOPs.

The signature of a VOP is a start code of `0x00 00 01`, followed by a byte that indicates the type of data in the VOP. Specifically, the signature for a video frame is `0x00 00 01 B6`, and a type-byte between `0x20` and `0x2F` indicates the decoding header information. The MPEG-4 decoder can decode the frame data and extract its size with this information.

2.3.3 H.264

In H.264, defined in ISO/IEC 14496-10:2009 [20], video data is separated into Network Abstraction Layer (NAL) units. Each NAL unit is identified by a start signature of either `0x00 00 01` or `0x00 00 00 01`. There are three types of NAL units: Sequence Parameter Set (SPS), Picture Parameter Set (PPS), and Slice. SPS and PPS contain important decoding information such as resolution and bit depth, while Slices contain frame data. Slices are comprised of two types of frames: IDR and P/B frames.

Instantaneous Decoder Refresh (IDR) frames draw a full frame, while Predictive or Bi-Directional (P/B) frames only contain changes from the previous frame and require a reference IDR frame to draw a picture. The type of NAL unit is determined by the byte following the start signature, with `0x67` representing SPS, `0x68` representing PPS, `0x65` representing IDR, and `0x61` representing P/B type frame data.

2.3.4 H.265

H.265 codec, also known as High Efficiency Video Coding (HEVC), has an improved compression performance compared to H.264. Being a continuation of H.264, the high-level NAL units are kept, including SPS, PPS, and Slices [21]. The major improvement from H.264 is a 50% bit-rate saving due to improved compression methods [22].

Therefore, the signatures for the different NAL units are equal to those in H.264 in reference to carving video data.

2.4 Binary formats

In our analysis of the binary data from the different surveillance systems, we need to decode different values into valuable data, such as numbers and timestamps.

Numbers are stored as a bit representation of the value, using 0 and 1. A byte is a group of eight bits representing 256 values (0-255). In order to store larger number, multiple bytes are used; two bytes has 65 536 values, four bytes (32 bits) has 2^{32} values, and eight bytes has 2^{64} values.

While interpreting data stored in bytes, we must account for the endianness of the data, in which order the bytes are stored from least to most significant. In Big Endian, the most significant byte is stored first, in the same manner as we write out normal numbers. The number 4660 is stored in big-endian as `0x12 34`. And in little-endian, this value is stored in reversed byte order, with the least significant value first. 4660 in little endian is `0x34 12`.

Timestamps are usually represented as a number of intervals since a start time, epoch, and it's the number of intervals that is stored. Different timestamp formats specify how the timestamp is converted to a number.

2.4.1 Windows Ticks

Windows Ticks is a time format that counts ticks since an epoch. A tick is a 100-nanosecond interval. And the time format represents the number of ticks that have passed since midnight on January 1st, 0001 [23].

The date 2023-03-14 12:34:56 (UTC) equals 638143940960000000 intervals. In hexadecimal, little endian: `0x00 18 7D 84 88 24 DB 08`

2.4.2 Windows Filetime

Similar to Windows Ticks, Filetime also counts the number of 100-nanosecond intervals since an epoch. The epoch for Windows Filetime is January 1, 1601 (UTC) [24].

The date 2023-03-14 12:34:56 (UTC) equals 133232708960000000 intervals. In hexadecimal, little endian: `0x00 18 06 62 71 56 D9 01`

2.4.3 Unix time

Unix Time is defined as the number of seconds elapsed since an epoch, January 1, 1970 [25]. And is often represented in a 32-bit signed integer but is also written in 64-bytes to prevent an overflow problem from occurring in 2038. In addition, the 64-bit representation has a resolution of microseconds.

The date 2023-03-14 12:34:56 (UTC) equals 1678797296 seconds. In hexadecimal, little endian: `0xF0 69 10 64`
And for 64-bit, 2023-03-14 12:34:56 (UTC) equals 1678797296000000 microseconds. In hexadecimal, little endian: `0x00 5C 7A 74 DB F6 05 00`

2.4.4 APFS Filetime

APFS timestamps are a 64-bit unsigned value that counts nanoseconds from January 1, 1970. This is the same epoch as Unix time, only in a greater resolution, seconds vs. nanoseconds. To convert from APFS timestamp to Unix time, the value must be divided by one billion, 10^9 [26].

2.5 DVR vs VMS

In proprietary Digital Video Recorder (DVR) systems the video signal from cameras is converted to digital streams within the DVR and stored on the hard drive in a video file using the same video codec for all video streams [4]. In Video Management Software (VMS) the video signals are digital from the cameras and stored on hard drives through a Network Video Recorder (NVR), this means that the codec in the video streams is determined by the cameras. Unless the NVR re-encodes the stream into another codec, but that would take a lot of processing power. This makes carving in VMS systems a little more difficult since there can be different codecs used in the same system.

Chapter 3

Related work

3.1 Bi-fragment carving

Traditional file carving techniques cannot automatically reassemble fragmented files. Garfinkel [27] surveyed over 300 hard drives and found that most of the fragmented files consisted of only two fragments: bifragmented files. By using the traditional carving techniques, recovering data between a known header and footer signature, the result was files that contained foreign data within the recovered file. This survey was conducted in 2007 and may not reflect today's situation as technology has evolved since then with updated file systems, various file formats and sizes, and different types of usage.

Bifragmented file recovery must carve from the header signature to the end of the first fragment and from the start of the second fragment to the footer signature. One method is to test all possible fragments by trying all gaps between the header and footer signature and selecting a candidate where the file is correctly assembled.

The proposed method from Garfinkel is carving fragmented files with fast object validation. Fast object validation is a validation of the internal structure of the file; for example, if a Microsoft Word file is rendered without errors, the file structure is valid. By knowing the internal structure of a file type, the carver can validate the file while it's being carved. In a Microsoft Office file, the validator can look for a CDH-header in the file's first sector and other structures and signatures within the file. There are also recognizable structures within JPEG image files, but additionally, the image is stored using a Huffman-coded representation, and the validator can check if the carved data is valid Huffman symbols. To carve a bifragmented file, the method starts with the smallest gap and tries to concatenate and validate all sizes of the two fragments, then continues to increase the gap if none is validated. This method will be time-consuming if the gap is large since the algorithm is at best O(n^2) when carving for a known header and footer.

Pal and Memon [12] describes other techniques to recover fragmented files. A technique for predicting characters based on the type of documents is described for text documents. Certain documents contain a higher frequency of certain characters, such as HTML documents containing more "<", ">", "/" and "=" than plain text documents. For images, they describe a system that analyses the pixel differences between a cluster's end and another's beginning to find the most probable combination; this is done by calculating the sum of differences of the last n pixels of the first cluster and the first n pixels in the other clusters, where n is the width of the image. The combination with the lowest difference sum is the most probable.

3.2 Frame-based recovery

Searching for internal fragments of files is a possible method if the separate parts can be put back together in the correct order. Na *et al.* [2] proposed a method that carves for frames of fragmented video files. They showed that the method was successful for MPEG-4 Visual and H.264 encoded video. The method was described in two phases: The extraction phase and the connection phase.

There are different types of video codecs, as they use different methods for compressing and delivering video data. It is important to know what codec was used to encode the video when trying to recover and play video from unallocated areas of the disk. Each codec has different header information, and it is possible to recognize the codec used depending on the structure of the frame header. The extraction phase searches for signatures of frames and interprets codec header information in the frames.

As mentioned in Chapter 2, MPEG-4 Visual frame information starts with the three bytes `0x00 00 01` followed by a byte indicating the data type. And a type-byte between `0x20` and `0x2F` indicates the decoding header information. The method uses this header information to validate the frame data with a MPEG-4 Visual decoder. If the frame data is verified, the decoder returns the frame size, and the frame is extracted.

H.264 frames are extracted similarly; as also mentioned in Chapter 2, H.264 frames have a signature of either `0x00 00 01` or `0x00 00 00 01` and a byte for the data type. Similar to the decoding header information in MPEG-4 Visual, the SPS and PPS NAL units contain decoding information. The method extracts SPS, PPS, and Slice to combine them and verify the frame data with an H.264 decoder.

In the connection phase, the method tries to put the frames back together in the correct order. First, they assumed that if two or more frames were following each other back to back, they were considered contiguous and already in the correct order, and if not, this was a fragmented file. MPEG-4 files have an information block for the size of each frame, stored in an Sample-to-Size (STSZ) box, usually at the end of the file. If a STSZ box is successfully carved in the extraction phase, the method uses this information to put the frames back in the correct order by comparing them with the frame sizes from the NAL units of the frames.

The result of their experiment on damaged or corrupted video showed that the method recovered close to every frame regardless of fragmentation. The results showed a restoration ratio of 40-50% of a video that was 50% overwritten, i.e., almost all the data that was not overwritten. And the amount of fragmentation of the file did not affect the restoration rate.

3.3 Frame-based recovery AVI

Similar to the method proposed by Na *et al.* to carve frames as fragments, a method to carve AVI video format was proposed by Yang *et al.* [15]. The AVI file consists of four parts, file header, stream info list, frames (audio and video), and an index with information and reference to each frame. The index records consist of four parts, frame type id, frame flags, frame offset relative to a starting point, and frame size. Each frame contains a header with a signature of the frame type id and size of the frame. Then by extracting both frames and indexes in the extraction phase and using the frame information from both frames and index in the reordering phase to assemble the fragments back together in the correct order.

A fragment is considered a set of extracted frames stored contiguously on the hard drive. And by comparing the set of frames in the fragment to sets of frames in the index, it is possible to reorder the frames even if there are frames with the same size.

3.4 Time codes in surveillance video

Surveillance equipment and systems are becoming more common for domestic use, and brands such as Hikvision, DAHUA, and Hanbang are the leading brands. The most commonly used video formats among these brands are either common video formats encapsulated with proprietary data or a proprietary video format. Lu *et al.* [28] suggested a method that searches for patterns in the metadata within the video data.

They used recorded data from the same camera to compare and find recurring patterns with metadata such as signatures and time codes. They found that the videos contained repeating flags that appeared thousands of times within the video data, and by analyzing the repeating patterns, they found information about video channel numbers, and close to that, they also found time codes. A four-byte time code that described the time in the first two bytes and the date in the last two was identified. At last, they extracted the video data from the data with the patterns as playable video and validated the video by checking that it was playable.

3.5 Metadata and timestamp carving

Nordvik *et al.* [29] propose a method that focuses on filesystem metadata information. *Generic Metadata Time Carving* is based on the assumption that the structure of the file systems file table allows for timestamps to be stored closely on the hard drive. For files, there are timestamps for when the files are created, modified, deleted, accessed, etc. And these timestamps are often equal to each other.

When a timestamp is located, the method scans for matching timestamps within a designated search window following the initial timestamp. To locate the file and retrieve its data, they used semantic parsers to identify the filesystem, validate the file record, and analyze the metadata. By examining the file record, they could find the file's physical location and try to recover the file data.

Porter *et al.* did additional research and developed the method *Timestamp prefix carving for filesystem metadata extraction* [30]. They improved the method by comparing a prefix part of candidate timestamps instead of only considering equal timestamps. They searched for temporally similar timestamps by only comparing the n most significant bytes. If the prefix length n is decreased, the timestamp equivalency will cover a longer time period. This is because the most significant bytes will represent years, months, and days in the timestamps. The results of their experiments showed that the improved method could massively increase recall for recovering filesystem metadata.

Chapter 4

Methodology

In this section, we describe how we analyzed each surveillance system to find patterns in the video data storage, and we suggest carving algorithms where applicable that allow for searching within a time frame. We analyzed a controlled dataset where we had complete control of all input and the recording period. To create the datasets for each surveillance system, we created a setup with seven IP cameras from different vendors. The cameras supported different video codecs, such as MJPEG, H.264, and H.265. Each surveillance system was installed on a clean operating system, either Windows 10 or Ubuntu 22.04, depending on the requirements of the systems. We used separate hard drives for the video recordings, one for each surveillance system, and the hard drives were zeroed and formatted with an appropriate file system before being allocated. NTFS for Windows and Ext4 for Linux.

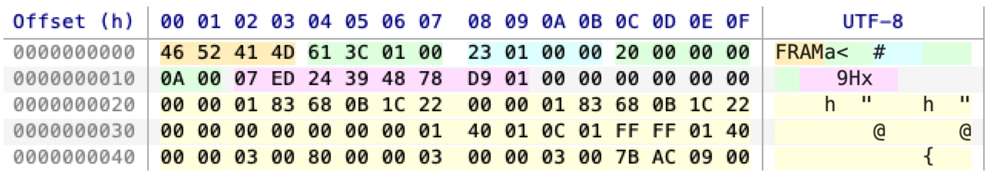
We did two rounds of recording, the first round to fill the hard drive with video data we would later try to recover and the second round a month later to overwrite approximately half the first recording round. There is a criterion in all the surveillance systems that they support deleting video data after a given period due to privacy regulations, and we set the systems in this experiment to delete surveillance data that was more than seven days old. By waiting a month between the two rounds of recording, we ensured that the systems did a deletion procedure before recording the second round. After each round of recording, we created an image file of the entire disk to get a snapshot of the content before and after the data was overwritten.

For each system, we will describe the setup of the system, version number, number of cameras, size of video storage, configuration of the surveillance system, dates for both recording periods, and how much of the allocated storage was used in the second recording.

The first part of analyzing the video data is to run an existing carving tool on the disk image to examine if the tool is able to carve information from the surveillance system. Several tools can carve multimedia files [13], but one tool stands out with a higher percentage of valid recovered files and overall better performance, *PhotoRec* [31]. Fikri *et al.* [32] showed in an experiment with video files as one of the file types that PhotoRec had a slightly lower rate in the number of recovered files but far better results in the validity of the recovered files. The high percentage of correct files is because PhotoRec uses structure-based carving when possible [13].

Then we will analyze the structure of the folders and files to find the most relevant files for the video recording and recovering of deleted video data. We will analyze the files on a binary level for the most relevant files to recognize how video data is stored and how the system keeps track of the timestamps for the video data. And finally, suggest a carving algorithm for the surveillance system if applicable.

To present binary data and highlight different parts, we will present the binary data in a hexadecimal view with an accompanied table to describe the highlighted data with the type of data and a description. In addition, we will use a highlighting color scheme for the hexadecimal view; see Figure 4.1 and Table 4.1 for an example of such highlighting. Signatures are highlighted with an orange color, offsets and sizes with a green color, counting data with a light cyan color, timestamps with a pink color, and video data and other raw data are highlighted with a yellow color.



```
Offset (h)  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F    UTF-8
0000000000  46 52 41 4D 61 3C 01 00  23 01 00 00 20 00 00 00  FRAMa<  #
0000000010  0A 00 07 ED 24 39 48 78  D9 01 00 00 00 00 00 00      9Hx
0000000020  00 00 01 83 68 0B 1C 22  00 00 01 83 68 0B 1C 22     h "     h "
0000000030  00 00 00 00 00 00 00 01  40 01 0C 01 FF FF 01 40         @       @
0000000040  00 00 03 00 80 00 00 03  00 00 03 00 7B AC 09 00              {
```

Figure 4.1: Example of highlighting a binary file with different colors for different content. Signatures are highlighted with an orange color, offsets and sizes with a green color, counting data with a light cyan color, timestamps with a pink color, and video data and other raw data are highlighted with a yellow color. (Offsets described in Table 4.1)

Offset in hex	Field type	Data type	Description
0x0	signature	char[4]	Frame start signature FRAM
0x4	size	uint32	size of the frame
0x8	number	uint32	current frame number, increasing
0xC	offset	uint32	offset to start of video data
0x10	size	uint16	length of header segment
0x12	timestamp	uint64	frame timestamp (Windows Filetime)
0x20	data		video data

Table 4.1: Example of presentation of binary content

4.1 Analyzing filesystems

Some video recording systems store their data inside their own proprietary filesystem and structure. Reverse engineering methods can be applied to analyze and find the correct interpretation for that data. By investigating the data from a CCTV system with an unknown filesystem, Tobin *et al.* [33] presented a pragmatic approach to recovering data from a hard drive. Using an "eavesdrop" approach with a monitoring tool, they found starting points on the hard drive for further analysis. Then by using known values and offsets from the monitoring tool, they decoded the data. Specifically, they were using a known timestamp to find the location and encoding method of that timestamp in the raw data.

Another approach is to compare two images of the same DVR system taken at different times and identify similar patterns and structures. Sandeepa *et al.* [34] proposed this method in a study of a HikVision DVR system and found signatures, offset pointers, size information, tree structures, and timestamps.

4.1.1 Finding patterns

After discovering the files with video recordings and index files, we analyze content on a binary level. First, we look for signatures that are similar across multiple files or appear in a recurring pattern throughout the file. If we find a signature, we extract a chunk of surrounding data from multiple signatures and compare those chunks. The length from one signature to the next can indicate the size of the data area between the signatures, and we search for this value relatively close to the signature. Some values increase from one chunk to the next, and these can be counters that indicate the internal number of the chunk. Another changing value type is timestamps, which are stored as a number-value of time since an epoch. From one chunk to the next, the change in time can be seconds or even fractions of a second. Han *et al.* [35] did a similar analysis of the HIKVISION DVR file system using known values and offsets to identify them in the binary data.

Figure 4.2 shows two types of common patterns. Common for both is video data being prefixed with header information with signatures, size information, and a timestamp of the recording. The difference is where the index is stored. The first example keeps the index in a separate file and contains a list with reference to all of the video data. In the second example, the index is stored within the video data and has reference to only a subset of the video data. Another index will follow the next subset of video data and holds the reference to those parts, and so on. For the surveillance software to search for specific video data, another overview index may reference all the indexes within the video data.

Another common pattern is to group together several video frames into one **'frame-group'** and provide the size and timestamp information in a frame-group header. This reduces the number of index-lines as they only need to reference the frame-group instead of each individual frame.

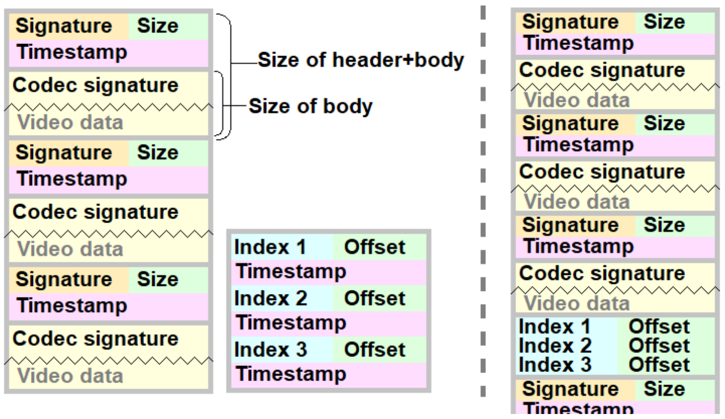


Figure 4.2: Two types of common patterns in dvr-files. The left side shows video data and index stored in separate files, and the right side shows video data and index stored in the same file.

4.1.2 Highlighting software

In order to identify and highlight these types of repetitive patterns, we created a tool that highlights timestamps within a dvr-file with known data. The tool takes a file where we expect to find timestamps within a time range, searches for multiple types of timestamps, and highlights the findings.

The tool takes a date-time value and a precision level and creates regex patterns for the different types of timestamps. The supported timestamps are currently Windows Ticks, Microsoft Filetime, Apple APFS time, and Unix epoch, both seconds and microseconds. The tools also support both little and big endian.

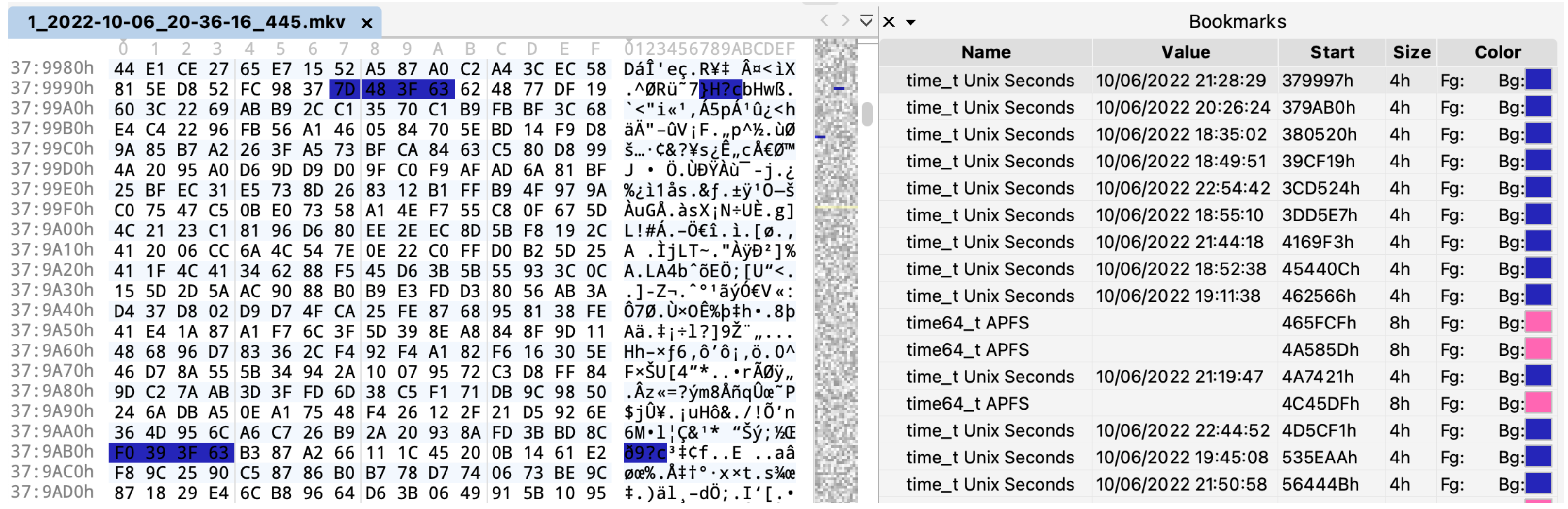
The precision levels are divided into three levels; high = timerange within seconds, medium = timerange within a few hours, and low = timerange about a months. Table 4.2 shows the different regex patterns and time ranges from the same date-time value with varying levels of precision.

The output is a CSV file containing bookmarks that can be imported into the hex-editor *010 Editor* [36]. Such an import is shown in Figure 4.3.

To supplement our tool, we used video codec viewers to identify specific parts of the video data to get an overview of offsets and size information that we could match against an index or header information, such as *h264-bitstream-viewer* [37].

Table 4.2: Regex patterns from different precision levels for date time value "06.10.2022 20:36:16"

	Regex	Time range (UTC)
Windows Ticks		
High	`.{3}\x6A\xDA\xA7\xDA\x08`	20:36:14 - 20:36:16
Medium	`.{4}[\xCA-\xEA]\xA7\xDA\x08`	18:38:45 - 22:34:58
Low	`.{5}[\x9C-\xB2]\xDA\x08`	2022-09-21 - 2022-10-21
Windows Filetime		
High	`.{3}\\x48\\xC3\\xD9\\xD8\x01`	20:36:15 - 20:36:17
Medium	`.{4}[\xB3-\xD3]\xD9\xD8\x01`	18:39:42 - 22:35:56
Low	`.{5}[\xCE-\xE4]\xD8\x01`	2022-09-21 - 2022-10-21
Unix seconds		
High	`[\x3F-\x41]\x3C\x3F\x63`	20:36:15 - 20:36:17
Medium	`.[\x1C-\x5C]\x3F\x63`	18:18:40 - 22:55:59
Low	`.{2}[\x2B-\x53]\x63`	2022-09-21 - 2022-10-22
Unix ms		
High	`.{2}[\x07-\x27]\xA5\x63\xEA\x05\x00`	20:36:14 - 20:36:17
Medium	`.{4}[\x62-\x64]\xEA\x05\x00`	18:38:31 - 22:13:16
Low	`.{5}[\xE9-\xEB]\x05\x00`	2022-09-19 - 2022-10-27
APFS		
High	`.{3}[\x7F-\xFF]\x3C\x95\x1B\x17`	20:36:14 - 20:36:16
Medium	`.{5}[\x8F-\x9B]\x1B\x17`	18:41:57- 22:40:11
Low	`.{6}[\x17-\x1F]\x17`	2022-09-21 - 2022-10-21



1_2022-10-06_20-36-16_445.mkv

```
          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
37:9980h  44 E1 CE 27 65 E7 15 52 A5 87 A0 C2 A4 3C EC 58  DáÎ'eç.R¥‡ Â¤<ìX
37:9990h  81 5E D8 52 FC 98 37 7D 48 3F 63 62 48 77 DF 19  .^ØRü˜7}H?cbHwß.
37:99A0h  60 3C 22 69 AB B9 2C C1 35 70 C1 B9 FB BF 3C 68  `<"i«¹,Á5pÁ¹û¿<h
37:99B0h  E4 C4 22 96 FB 56 A1 46 05 84 70 5E BD 14 F9 D8  äÄ"–ûV¡F.„p^½.ùØ
37:99C0h  9A 85 B7 A2 26 3F A5 73 BF CA 84 63 C5 80 D8 99  š…·¢&?¥s¿Ê„cÅ€Ø™
37:99D0h  4A 20 95 A0 D6 9D D9 D0 9F C0 F9 AF AD 6A 81 BF  J • Ö.ÙÐŸÀù¯-j.¿
37:99E0h  25 BF EC 31 E5 73 8D 26 83 12 B1 FF B9 4F 97 9A  %¿ì1ås.&ƒ.±ÿ¹O—š
37:99F0h  C0 75 47 C5 0B E0 73 58 A1 4E F7 55 C8 0F 67 5D  ÀuGÅ.àsX¡N÷UÈ.g]
37:9A00h  4C 21 23 C1 81 96 D6 80 EE 2E EC 8D 5B F8 19 2C  L!#Á.–Ö€î.ì.[ø.,
37:9A10h  41 20 06 CC 6A 4C 54 7E 0E 22 C0 FF D0 B2 5D 25  A .ÌjLT~."ÀÿÐ²]%
37:9A20h  41 1F 4C 41 34 62 88 F5 45 D6 3B 5B 55 93 3C 0C  A.LA4bˆõEÖ;[U“<.
37:9A30h  15 5D 2D 5A AC 90 88 B0 B9 E3 FD D3 80 56 AB 3A  .]-Z¬.ˆ°¹ãýÓ€V«:
37:9A40h  D4 37 D8 02 D9 D7 4F CA 25 FE 87 68 95 81 38 FE  Ô7Ø.Ù×OÊ%þ‡h•.8þ
37:9A50h  41 E4 1A 87 A1 F7 6C 3F 5D 39 8E A8 84 8F 9D 11  Aä.‡¡÷l?]9Ž¨„...
37:9A60h  48 68 96 D7 83 36 2C F4 92 F4 A1 82 F6 16 30 5E  Hh–×ƒ6,ô’ô¡‚ö.0^
37:9A70h  46 D7 8A 55 5B 34 94 2A 10 07 95 72 C3 D8 FF 84  F×ŠU[4”*..•rÃØÿ„
37:9A80h  9D C2 7A AB 3D 3F FD 6D 38 C5 F1 71 DB 9C 98 50  .Âz«=?ým8ÅñqÛœ˜P
37:9A90h  24 6A DB A5 0E A1 75 48 F4 26 12 2F 21 D5 92 6E  $jÛ¥.¡uHô&./!Õ’n
37:9AA0h  36 4D 95 6C A6 C7 26 B9 2A 20 93 8A FD 3B BD 8C  6M•l¦Ç&¹* “Šý;½Œ
37:9AB0h  F0 39 3F 63 B3 87 A2 66 11 1C 45 20 0B 14 61 E2  ð9?c³‡¢f..E ..aâ
37:9AC0h  F8 9C 25 90 C5 87 86 B0 B7 78 D7 74 06 73 BE 9C  øœ%.Å‡†°·x×t.s¾œ
37:9AD0h  87 18 29 E4 6C B8 96 64 D6 3B 06 49 91 5B 10 95  ‡.)äl¸–dÖ;.I‘[.•
```

Bookmarks

Name	Value	Start	Size	Color	
time_t Unix Seconds	10/06/2022 21:28:29	379997h	4h	Fg:	Bg: ■
time_t Unix Seconds	10/06/2022 20:26:24	379AB0h	4h	Fg:	Bg: ■
time_t Unix Seconds	10/06/2022 18:35:02	380520h	4h	Fg:	Bg: ■
time_t Unix Seconds	10/06/2022 18:49:51	39CF19h	4h	Fg:	Bg: ■
time_t Unix Seconds	10/06/2022 22:54:42	3CD524h	4h	Fg:	Bg: ■
time_t Unix Seconds	10/06/2022 18:55:10	3DD5E7h	4h	Fg:	Bg: ■
time_t Unix Seconds	10/06/2022 21:44:18	4169F3h	4h	Fg:	Bg: ■
time_t Unix Seconds	10/06/2022 18:52:38	45440Ch	4h	Fg:	Bg: ■
time_t Unix Seconds	10/06/2022 19:11:38	462566h	4h	Fg:	Bg: ■
time64_t APFS		465FCFh	8h	Fg:	Bg: ■
time64_t APFS		4A585Dh	8h	Fg:	Bg: ■
time_t Unix Seconds	10/06/2022 21:19:47	4A7421h	4h	Fg:	Bg: ■
time64_t APFS		4C45DFh	8h	Fg:	Bg: ■
time_t Unix Seconds	10/06/2022 22:44:52	4D5CF1h	4h	Fg:	Bg: ■
time_t Unix Seconds	10/06/2022 19:45:08	535EAAh	4h	Fg:	Bg: ■
time_t Unix Seconds	10/06/2022 21:50:58	56444Bh	4h	Fg:	Bg: ■

Figure 4.3: Bookmarks imported into 010 Editor to highlight potential timestamps within a time range.

4.2 Milestone

4.2.1 Setup

Milestone XProtect VMS version 22.2a [38] was installed on a clean Windows 10 operating system, and seven cameras were configured. We set up the system to record continuously and with a deletion policy of 7 days. The first recording period was conducted from 22. Sept 2022 18:50 UTC to 26. Sept 2022 19:25 UTC, and the second period from 27. Oct 2022 to 30. Oct 2022. The system reported at this point that the data storage used 50 percent of the allocated storage. When the second recording period started, the system deleted all the files from the previous recordings.

4.2.2 Filestructure of recordings

The recording disk has a folder 'MediaDatabase' with over 600 folders. The folders have names with the camera name followed by a timestamp, one hour between each folder. Within each folder, there is a config file and three folders, one with the recorded data. In that folder, there are index files describing timestamps and offsets within the recorded data. And the recorded data is stored in files named 'blockNN.blk', where NN is a number from 0 to the number of block files. The size of each of the block files is a maximum of 16MB.

4.2.3 Filecontent of relevant files

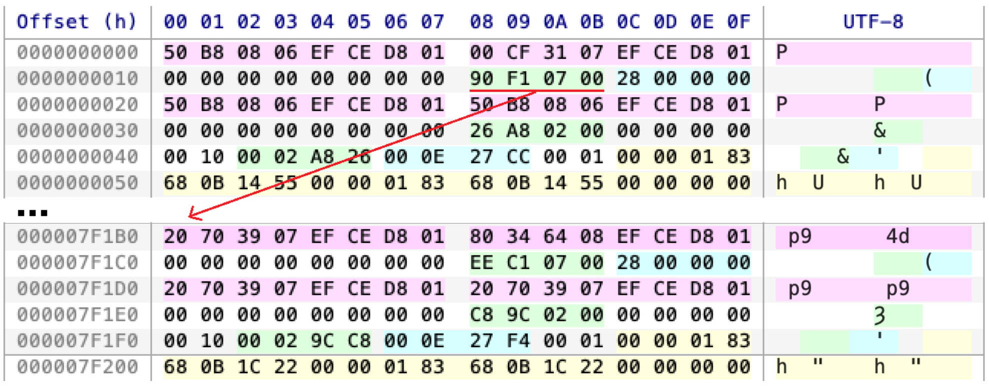
blockNN.blk files contain the video data, separated into frames, and each frame is preceded by timestamps, size and other information. Multiple frames are grouped into a 'frame-group' of approx 40 - 80 frames. See Figure 4.4 and Figure 4.5

cindex.idx contains an index of all the block files. Each block file is described by 24 bytes: position, size, first timestamp, and last timestamp. This is illustrated and described in Figure 4.6 and Table 4.5.

sindexN.idx files (where N is a number) contain an index of all frame-groups within block files. Each frame-group is described with 40 bytes: two timestamps describing the timespan for the frame, the blockNN number, the start and end offset to the frame-group, and the number of frames in the frame-group. This is illustrated and described in Figure 4.7 and Table 4.6.

4.2.4 Proposed carving method

Since the index data is written to separate files from the recorded data, it is difficult to find the connection between the index and block files when the folder structures are missing, for example, in an unallocated section of the hard drive. It could be possible to find the index and match the internal data such as timestamps and size of the blocks to join the index with the block data. But the block header contains enough data to retrieve the video content.

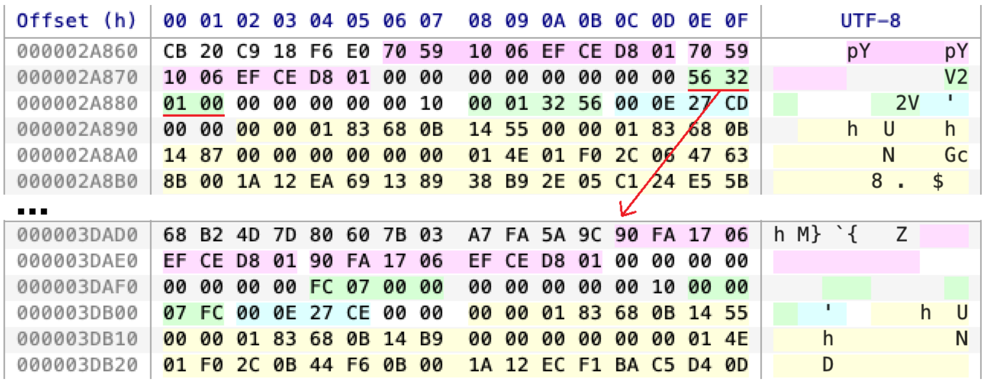


```
Offset (h)  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F      UTF-8
0000000000  50 B8 08 06 EF CE D8 01  00 CF 31 07 EF CE D8 01  P
0000000010  00 00 00 00 00 00 00 00  90 F1 07 00 28 00 00 00              (
0000000020  50 B8 08 06 EF CE D8 01  50 B8 08 06 EF CE D8 01  P       P
0000000030  00 00 00 00 00 00 00 00  26 A8 02 00 00 00 00 00          &
0000000040  00 10 00 02 A8 26 00 0E  27 CC 00 01 00 00 01 83       &  '
0000000050  68 0B 14 55 00 00 01 83  68 0B 14 55 00 00 00 00  h  U    h  U
...
000007F1B0  20 70 39 07 EF CE D8 01  80 34 64 08 EF CE D8 01   p9       4d
000007F1C0  00 00 00 00 00 00 00 00  EE C1 07 00 28 00 00 00              (
000007F1D0  20 70 39 07 EF CE D8 01  20 70 39 07 EF CE D8 01   p9       p9
000007F1E0  00 00 00 00 00 00 00 00  C8 9C 02 00 00 00 00 00          3
000007F1F0  00 10 00 02 9C C8 00 0E  27 F4 00 01 00 00 01 83          '
000007F200  68 0B 1C 22 00 00 01 83  68 0B 1C 22 00 00 00 00  h  "    h  "
```

Figure 4.4: Milestone frame-group headers (Offsets described in Table 4.3) The value in offset `0x18` is the size of the frame-group: `0x07F190`. Adding this size to the first offset after the header, `0x20`, gives us the location of the next frame-group: `0x7F1B0`.

Offset in hex	Field type	Data type	Description
0x0	timestamp	uint64	first timestamp in the frame-group (Windows Filetime)
0x8	timestamp	uint64	last timestamp in the frame-group (Windows Filetime)
0x18	size	uint32	size of the frame-group
0x1C	number	uint32	number of frames in frame-group

Table 4.3: blockNN.blk frame-group header structure

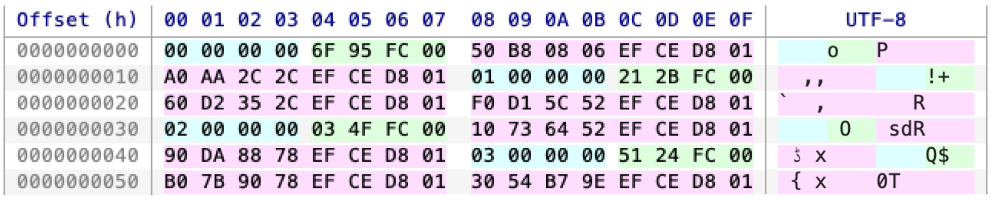


```
Offset (h)  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F   UTF-8
000002A860  CB 20 C9 18 F6 E0 70 59  10 06 EF CE D8 01 70 59        pY      pY
000002A870  10 06 EF CE D8 01 00 00  00 00 00 00 00 00 56 32                V2
000002A880  01 00 00 00 00 00 00 10  00 01 32 56 00 0E 27 CD            2V  '
000002A890  00 00 00 00 01 83 68 0B  14 55 00 00 01 83 68 0B        h  U    h
000002A8A0  14 87 00 00 00 00 00 00  01 4E 01 F0 2C 06 47 63           N    Gc
000002A8B0  8B 00 1A 12 EA 69 13 89  38 B9 2E 05 C1 24 E5 5B          8 .  $
...
000003DAD0  68 B2 4D 7D 80 60 7B 03  A7 FA 5A 9C 90 FA 17 06   h M} `{   Z
000003DAE0  EF CE D8 01 90 FA 17 06  EF CE D8 01 00 00 00 00
000003DAF0  00 00 00 00 FC 07 00 00  00 00 00 00 00 10 00 00
000003DB00  07 FC 00 0E 27 CE 00 00  00 00 01 83 68 0B 14 55       '        h  U
000003DB10  00 00 01 83 68 0B 14 B9  00 00 00 00 00 00 01 4E       h          N
000003DB20  01 F0 2C 0B 44 F6 0B 00  1A 12 EC F1 BA C5 D4 0D       D
```

Figure 4.5: Milestone frame headers. (Offsets described in Table 4.4) The value in offset `0x2A87E` is the size of the frame: `0x013256`. Adding this size to the first offset after the header, `0x2A886`, gives us the location of the next frame: `0x3DADC`.

Offset in hex	Field type	Data type	Description
0x0	timestamp	uint64	first timestamp in the frame (Windows Filetime)
0x8	timestamp	uint64	last timestamp in the frame (Windows Filetime)
0x18	size	uint32	size of the frame
0x22	size	uint32	size of the frame (big endian)
0x26	number	uint32	frame counter

Table 4.4: blockNN.blk frame header structure

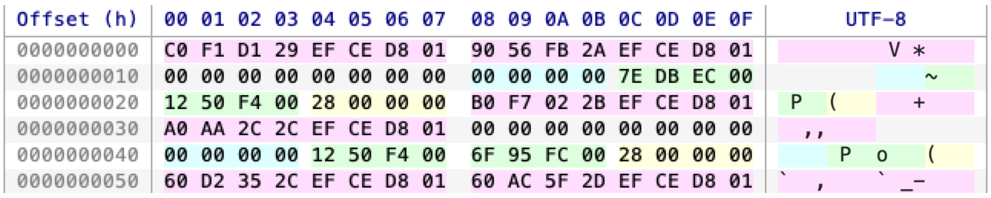


```
Offset (h)  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F       UTF-8
0000000000  00 00 00 00 6F 95 FC 00  50 B8 08 06 EF CE D8 01      o   P
0000000010  A0 AA 2C 2C EF CE D8 01  01 00 00 00 21 2B FC 00     ,,      !+
0000000020  60 D2 35 2C EF CE D8 01  F0 D1 5C 52 EF CE D8 01    `  ,     R
0000000030  02 00 00 00 03 4F FC 00  10 73 64 52 EF CE D8 01       O  sdR
0000000040  90 DA 88 78 EF CE D8 01  03 00 00 00 51 24 FC 00     ڈ x     Q$
0000000050  B0 7B 90 78 EF CE D8 01  30 54 B7 9E EF CE D8 01     { x  0T
```

Figure 4.6: Content of Milestone cindex.idx. (Offsets described in Table 4.5)

Offset in hex	Field type	Data type	Description
0x0	number	uint32	NN value in blockNN.blk filename
0x4	size	uint32	size of the blockNN.blk file
0x8	timestamp	uint64	first timestamp in the blockNN.blk file (Windows Filetime)
0x10	timestamp	uint64	last timestamp in the blockNN.blk file (Windows Filetime)

Table 4.5: cindex.idx index structure



```
Offset (h)  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F       UTF-8
0000000000  C0 F1 D1 29 EF CE D8 01  90 56 FB 2A EF CE D8 01             V *
0000000010  00 00 00 00 00 00 00 00  00 00 00 00 7E DB EC 00              ~
0000000020  12 50 F4 00 28 00 00 00  B0 F7 02 2B EF CE D8 01     P  (      +
0000000030  A0 AA 2C 2C EF CE D8 01  00 00 00 00 00 00 00 00     ,,
0000000040  00 00 00 00 12 50 F4 00  6F 95 FC 00 28 00 00 00        P  o  (
0000000050  60 D2 35 2C EF CE D8 01  60 AC 5F 2D EF CE D8 01    `  ,    ` _-
```

Figure 4.7: Content of Milestone sindexN.idx. (Offsets described in Table 4.6)

Offset in hex	Field type	Data type	Description
0x0	timestamp	uint64	first timestamp in the frame-group (Windows Filetime)
0x8	timestamp	uint64	last timestamp in the frame-group (Windows Filetime)
0x18	number	uint32	NN value in blockNN.blk filename
0x1C	offset	uint32	start offset for frame-group
0x20	offset	uint32	end offset for frame-group
0x24	offset	uint32	number of frames in frame-group

Table 4.6: sindexN.idx index structure

The regex pattern must cover a period from 22.09.2022 18:50 UTC to 26.09.2022 19:25 UTC. In Windows Filetime format, the five bytes with the lowest significance can be any value, the sixth byte must range from `0xCE` to `0xD1`, and the two most significant bytes are `0xD801`. This gives a range from `0x0000000000CED801` (21.09.2022 21:20 UTC) to `0xFFFFFFFFFFD1D801` (26.09.22 23:30 UTC).
And a timeframe regex pattern of `.{5}[\xCE-\xD1]\xD8\x01`

Code listing 4.1: Carving Milestone videodata

```
input: imagefile, regex for timeframe search
output: files of carved data

load imagefile
read chunk of data
  regex-search chunk for frame-group headers
  #(<TIMEFRAME><TIMEFRAME>.{16}<TIMEFRAME><TIMEFRAME>)
  if frame-group header found:
    read frame-group header and body
    if last frame-group is contiguous with this one:
      continue
    else:
      save carved data to file
  read next chunk of data
```

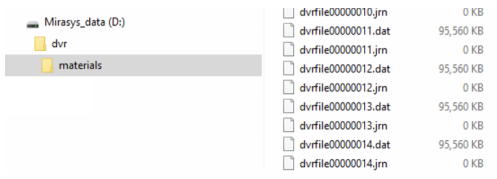
4.3 Mirasys

4.3.1 Setup

Mirasys VMS system version 9.4.0.1 [39] was installed on a clean Windows 10 operating system, and one camera was configured due to demo-license limitations. We set up the system to record continuously on a disk of 230 GB and with a deletion policy of 7 days. The first recording period was conducted from 11. Jan 2023 12:42 UTC to 16. Jan 2023 23:07 UTC, and the second period from 1. Feb 2023 to 6. Feb 2023. The system reported at this point that the data storage used 50 percent of the allocated storage. When the second recording period started, the system did not delete any of the existing files.

4.3.2 Filestructure of recordings

After installation, the system allocates almost the entire media disk with 455 GB of 465 GB. A folder 'dvr' with one subfolder 'materials' is created at the root level, under 'materials', a total of 10001 files were created; 5000 pairs of files, .dat and .jrn, both named 'dvrfile00000001' - 'dvrfile00005000' and an index file 'MaterialFolderIndex.dat' (Figure 4.8)



```
Mirasys_data (D:)        dvrfile00000010.jrn        0 KB
  dvr                    dvrfile00000011.dat   95,560 KB
    materials            dvrfile00000011.jrn        0 KB
                         dvrfile00000012.dat   95,560 KB
                         dvrfile00000012.jrn        0 KB
                         dvrfile00000013.dat   95,560 KB
                         dvrfile00000013.jrn        0 KB
                         dvrfile00000014.dat   95,560 KB
                         dvrfile00000014.jrn        0 KB
```

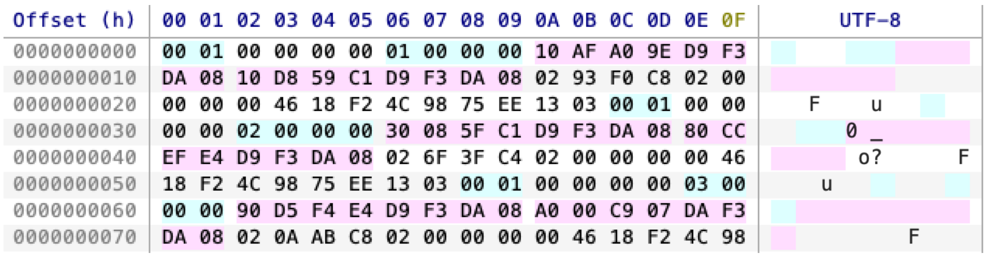
Figure 4.8: Files automaticly generated by Mirasys on the mediadisk.

4.3.3 Filecontent of relevant files

All dvrfile0000NNNN.jrn are initially empty files. All dvrfile000NNNN.dat files are equal in size (94 MB) and initially with no data (all 0x00). MaterialFolderIndex.dat contains a repeating pattern of 44 bytes, describing every dvrfile0000NNNN.dat, whether it is in use or not, and the first and last timestamp in Windows Ticks format. This is illustrated and described in Figure 4.9 and Table 4.7

After recording, the dvrfile000NNNN.dat files contain recognizable patterns. The files contain groups of indexes and frame-groups alternating in the file. The index describes the subsequent frame-group; one file can contain multiple indexes and frame-groups.

The first part is an index-overview for the groups later in the file. Every 24 bytes describe an index and frame-group: the first and last timestamp in the group, the size of the group, and the size of the index. This is illustrated and described in Figure 4.10 and Table 4.8

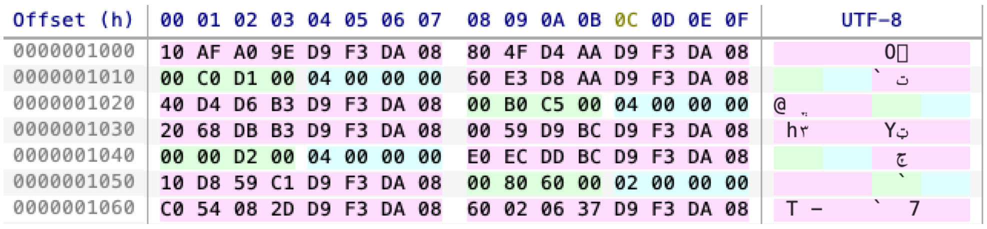


```
Offset (h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   UTF-8
0000000000  00 01 00 00 00 00 01 00 00 00 10 AF A0 9E D9 F3
0000000010  DA 08 10 D8 59 C1 D9 F3 DA 08 02 93 F0 C8 02 00
0000000020  00 00 00 46 18 F2 4C 98 75 EE 13 03 00 01 00 00     F     u
0000000030  00 00 02 00 00 00 30 08 5F C1 D9 F3 DA 08 80 CC           0 _
0000000040  EF E4 D9 F3 DA 08 02 6F 3F C4 02 00 00 00 00 46           o?       F
0000000050  18 F2 4C 98 75 EE 13 03 00 01 00 00 00 00 03 00       u
0000000060  00 00 90 D5 F4 E4 D9 F3 DA 08 A0 00 C9 07 DA F3
0000000070  DA 08 02 0A AB C8 02 00 00 00 00 46 18 F2 4C 98                 F
```

Figure 4.9: Hex view of MaterialFolderIndex.dat in Mirasys. (Offsets described in Table 4.7)

Offset	Field type	Data type	Description
0x0	flag	uint16	is dvrfile000NN.dat allocated to recordings
0x6	number	uint32	NN value in dvrfile000NN.dat filename
0x0A	timestamp	uint64	first timestamp in the dvrfile000NN.dat file (Windows Ticks)
0x12	timestamp	uint64	last timestamp in the dvrfile000NN.dat file (Windows Ticks)

Table 4.7: MaterialFolderIndex.dat index structure



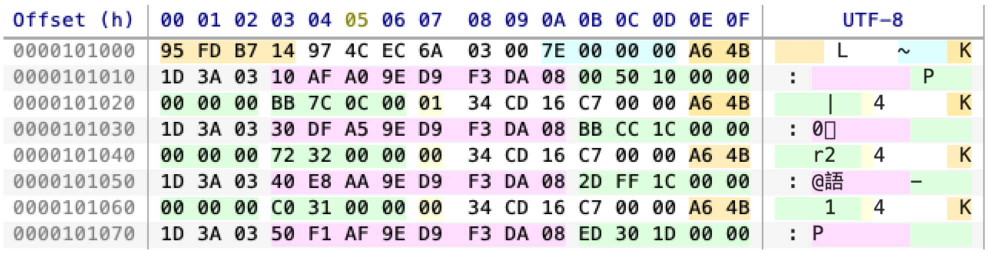
```
Offset (h)  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F   UTF-8
0000001000  10 AF A0 9E D9 F3 DA 08  80 4F D4 AA D9 F3 DA 08          O
0000001010  00 C0 D1 00 04 00 00 00  60 E3 D8 AA D9 F3 DA 08          ` ث
0000001020  40 D4 D6 B3 D9 F3 DA 08  00 B0 C5 00 04 00 00 00   @
0000001030  20 68 DB B3 D9 F3 DA 08  00 59 D9 BC D9 F3 DA 08    h۳       Yټ
0000001040  00 00 D2 00 04 00 00 00  E0 EC DD BC D9 F3 DA 08             ݼ
0000001050  10 D8 59 C1 D9 F3 DA 08  00 80 60 00 02 00 00 00            `
0000001060  C0 54 08 2D D9 F3 DA 08  60 02 06 37 D9 F3 DA 08   T -      `  7
```

Figure 4.10: The index overview in each dvrfile0000NNNN.dat in Mirasys. (Offsets described in Table 4.8)

Offset	Field type	Data type	Description
0x0	timestamp	uint64	first timestamp in group (Windows Ticks)
0x8	timestamp	uint64	last timestamp in group (Windows Ticks)
0x10	size	uint32	size of group
0x14	number	uint32	number of clusters for each group

Table 4.8: The index overview in each dvrfile0000NNNN.dat in Mirasys.

An index starts with the signature `0x95 FD B7 14` and the number of frames in the index at offset 10. From offset 14 it contains information about every subsequent frame: 32 bytes per frame that contain: timestamp, offset to the frame, size, and type of frame.



```
Offset (h)  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F       UTF-8
0000101000  95 FD B7 14 97 4C EC 6A  03 00 7E 00 00 00 A6 4B      L    ~   K
0000101010  1D 3A 03 10 AF A0 9E D9  F3 DA 08 00 50 10 00 00    :         P
0000101020  00 00 00 BB 7C 0C 00 01  34 CD 16 C7 00 00 A6 4B       |  4      K
0000101030  1D 3A 03 30 DF A5 9E D9  F3 DA 08 BB CC 1C 00 00    : 0▯
0000101040  00 00 00 72 32 00 00 00  34 CD 16 C7 00 00 A6 4B      r2  4      K
0000101050  1D 3A 03 40 E8 AA 9E D9  F3 DA 08 2D FF 1C 00 00    : @語     -
0000101060  00 00 00 C0 31 00 00 00  34 CD 16 C7 00 00 A6 4B       1  4      K
0000101070  1D 3A 03 50 F1 AF 9E D9  F3 DA 08 ED 30 1D 00 00    : P
```

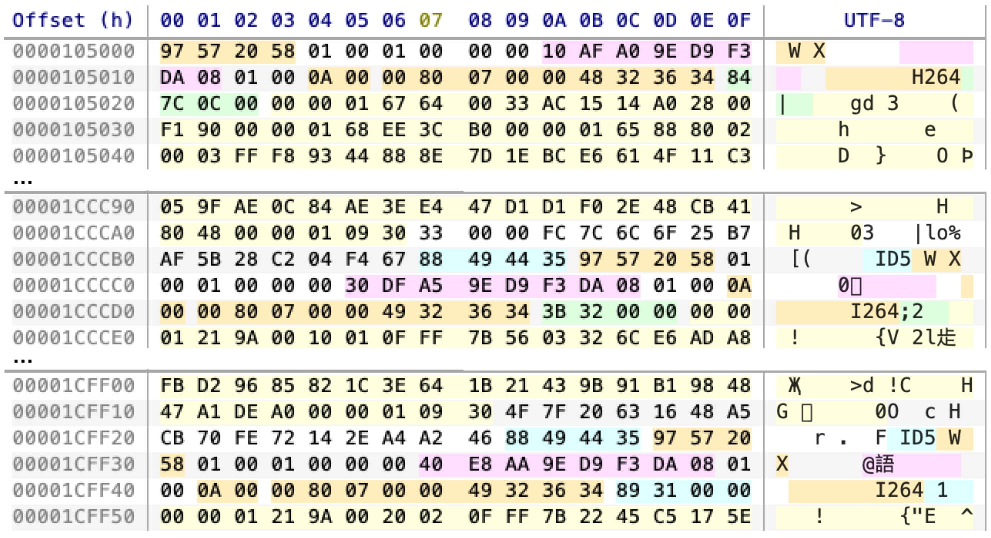
Figure 4.11: One of the indexes in dvrfile0000NNNN.dat in Mirasys. (Offsets described in Table 4.9)

Offset	Field type	Data type	Description
0x0	signature	char[4]	Index signature `0x95 FD B7 14`
0xA	number	uint32	number of indexlines (frames) in index
0xE	indexlines		lines of index rows, each 32bytes
indexlines starting at offset 0xE			
0xE	signature	char[2]	Indexline signature `0xA6 4B`
0x13	timestamp	uint64	frame timestamp (Windows Ticks)
0x1B	number	uint64	offset to frame. From start of file
0x23	number	uint32	size of frame
0x27	number	uint8	type of frame (interframe vs intraframe)

Table 4.9: Structure of indexes in dvrfile0000NNNN.dat in Mirasys.

Each frame contains a header with the signature `0x97 57 20 58`, the timestamp for the frame, and information about the source camera. The header length is 35 bytes, and the video data follows the header.

This makes it possible to recover the entire dvrfile000NNNN.dat file.



```
Offset (h)  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F      UTF-8
0000105000  97 57 20 58 01 00 01 00  00 00 10 AF A0 9E D9 F3   W X
0000105010  DA 08 01 00 0A 00 00 80  07 00 00 48 32 36 34 84             H264
0000105020  7C 0C 00 00 00 01 67 64  00 33 AC 15 14 A0 28 00  |    gd 3     (
0000105030  F1 90 00 00 01 68 EE 3C  B0 00 00 01 65 88 80 02        h        e
0000105040  00 03 FF F8 93 44 88 8E  7D 1E BC E6 61 4F 11 C3        D  }     O Þ
...
00001CCC90  05 9F AE 0C 84 AE 3E E4  47 D1 D1 F0 2E 48 CB 41           >       H
00001CCCA0  80 48 00 00 01 09 30 33  00 00 FC 7C 6C 6F 25 B7   H     03     |lo%
00001CCCB0  AF 5B 28 C2 04 F4 67 88  49 44 35 97 57 20 58 01   [(       ID5 W X
00001CCCC0  00 01 00 00 00 30 DF A5  9E D9 F3 DA 08 01 00 0A          0▯
00001CCCD0  00 00 80 07 00 00 49 32  36 34 3B 32 00 00 00 00          I264;2
00001CCCE0  01 21 9A 00 10 01 0F FF  7B 56 03 32 6C E6 AD A8   !        {V 2l走
...
00001CFF00  FB D2 96 85 82 1C 3E 64  1B 21 43 9B 91 B1 98 48   Ж      >d !C      H
00001CFF10  47 A1 DE A0 00 00 01 09  30 4F 7F 20 63 16 48 A5  G ▯          0O   c H
00001CFF20  CB 70 FE 72 14 2E A4 A2  46 88 49 44 35 97 57 20       r .   F ID5 W
00001CFF30  58 01 00 01 00 00 00 40  E8 AA 9E D9 F3 DA 08 01  X          @語
00001CFF40  00 0A 00 00 80 07 00 00  49 32 36 34 89 31 00 00             I264 1
00001CFF50  00 00 01 21 9A 00 20 02  0F FF 7B 22 45 C5 17 5E        !       {"E   ^
```

Figure 4.12: The first three frame headers in dvrfile0000NNNN.dat in Mirasys. (Offsets described in Table 4.10)

Offset	Field type	Data type	Description
0x0	signature	char[4]	Index signature `0x97 57 20 58`
0xA	timestamp	uint64	frame timestamp (Windows Ticks)
0x14	size	uint16	length of camera info
0x16	signature	char[x]	camera info
0x1F	size	uint32	length of frame
0x23	data		Video data

Table 4.10: Structure of frame header in dvrfile0000NNNN.dat in Mirasys.

4.3.4 Proposed carving method

Both the indexlines and the frame headers contain searchable timestamp information. It is most beneficial to search for the indexlines first and try to recover as much as possible of the file rather than only the individual frames. Each indexline has the offset value from the start of the file to the frame, so it is possible to calculate the beginning of the file by finding and verifying the first frame after the index. Verifying is done by checking for frame signature and comparing the index and frame timestamps. The recovery of the file will continue as long as the indexlines have a corresponding frame, and write out the file as soon as the information doesn't match. This makes it possible to recover the whole file if it hasn't been overwritten. The method also searches for single frames with no corresponding indexlines and recovers the frames.

The regex pattern for the timeframe must cover a period from 11. Jan 2023 12:42 UTC to 16. Jan 2023 23:07 UTC. In Microsoft Ticks timestamp format, the five bytes with the lowest significance can be any value, the sixth byte must range from `0xF3` to `0xF7`, and the two most significant bytes are `0xDA08`. This gives a range from `0x0000000000F3DA08` (10.01.2023 11:44 UTC) to `0xFFFFFFFFFFF7DA08` (16.01.2023 20:26 UTC).
And a timeframe regex pattern of `.{5}[\xF3-\xF7]\xDA\x08`

Code listing 4.2: Carving Mirasys videodata

```
input: imagefile, regex for timeframe search
output: files of carved data

load imagefile
read chunk of data
  regex-search chunk for indexlines #(A6 4B .{3} <TIMEFRAME>)
  if indexline found:
    find index start #(signature "A6 FD B7 14")
    interpret the index into list of frame offsets.

    for each frameoffset:
      if offset contains valid frame: #(signature "97 57 20 58")
        verify frame header and body
        continue
      else:
        # rest of file is overwritten
        save carved data to file

  else:
    regex-search chunk for single frames #(97 57 20 58 .{6} <TIMEFRAME>)
    read frame header and body
    if last frame is contiguous with this one:
      continue
    else:
      save carved data to file
  read next chunk of data
```

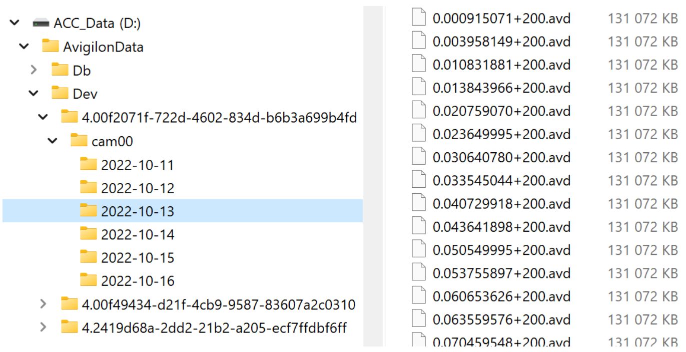
4.4 Avigilon

4.4.1 Setup

Avigilon Control Center version 7.14 [40] was installed on a clean Windows 10 operating system, and seven cameras were configured. We set up the system to record continuously and with a deletion policy of 7 days. The first recording period was conducted from 11. Oct 2022 20:00 UTC to 16. Oct 2022 15:00 UTC, and the second period from 7. Nov 2022 to 9. Nov 2022. The system reported at this point that the data storage used 50 percent of the allocated storage. When the second recording period started, the system moved all video data files back into the FilePool folder.

4.4.2 Filestructure of recordings

After installation, the system allocates almost the entire media disk with 460 GB of 465 GB, and a folder 'AvigilonData' with three subfolders 'Db', 'Dev', and 'FilePool' are created. Almost all of the total capacity of the volume is split into 128 MB files and stored with increasing numbering in the 'FilePool' folder; 3698 files were created for this system. The 'Dev' folder is used for storing recordings from devices, and the recording system allocates files from 'filepool'. The system moves the file into the correct camera folder and subfolder for the date under the 'dev' folder. In the camera and date folders, the files are renamed to a format representing the first recording time in the file [day offset].[hour][minute][seconds][ms][timezone offset].
The file `cam00/2022-10-12/0.211249216+100.avd` equals 12. Oct 2022 21:12:49.216 +01.



```
ACC_Data (D:)
  AvigilonData
    Db
    Dev
      4.00f2071f-722d-4602-834d-b6b3a699b4fd
        cam00
          2022-10-11
          2022-10-12
          2022-10-13
          2022-10-14
          2022-10-15
          2022-10-16
      4.00f49434-d21f-4cb9-9587-83607a2c0310
      4.2419d68a-2dd2-21b2-a205-ecf7ffdbf6ff
```

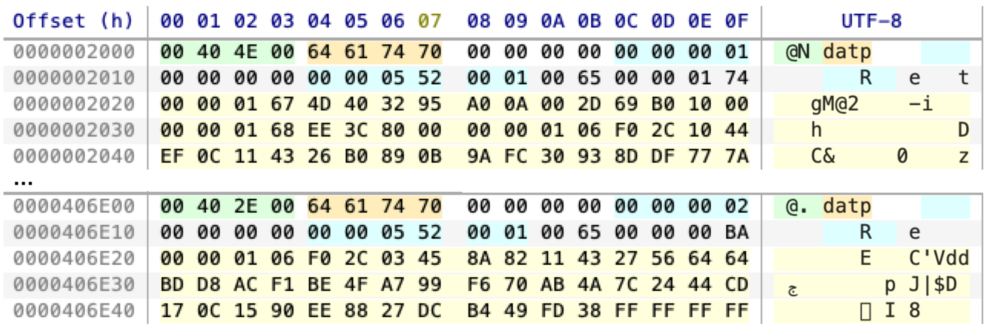
```
0.000915071+200.avd    131 072 KB
0.003958149+200.avd    131 072 KB
0.010831881+200.avd    131 072 KB
0.013843966+200.avd    131 072 KB
0.020759070+200.avd    131 072 KB
0.023649995+200.avd    131 072 KB
0.030640780+200.avd    131 072 KB
0.033545044+200.avd    131 072 KB
0.040729918+200.avd    131 072 KB
0.043641898+200.avd    131 072 KB
0.050549995+200.avd    131 072 KB
0.053755897+200.avd    131 072 KB
0.060653626+200.avd    131 072 KB
0.063559576+200.avd    131 072 KB
0.070459548+200.avd    131 072 KB
```

Figure 4.13: Files automaticly generated by Avigilon on the mediadisk.

4.4.3 Filecontent of relevant files

.avd files contain frame-groups and indexes alternating. Frame-groups contain video data with the signature '`datp`', size info, and counters, as described in Figure 4.14 and Table 4.11. Following a set of frame-groups is metadata describing the preceding frame-groups along with a timestamp, described in Figure 4.15 and Table 4.12. Within the metadata, there is an index containing offsets and the size of the frame-groups, described in Figure 4.16 and Table 4.13.

The timestamp is in APFS time format big-endian, indicating the number of nanoseconds since January 1, 1970. Overall the endianness is mainly big-endian: in the timestamps, size of datp, counters, and other numbers. Except for the frame-group size and offset in the index, denoted in Little Endian Base128 (leb128), a variable length format to store numbers.

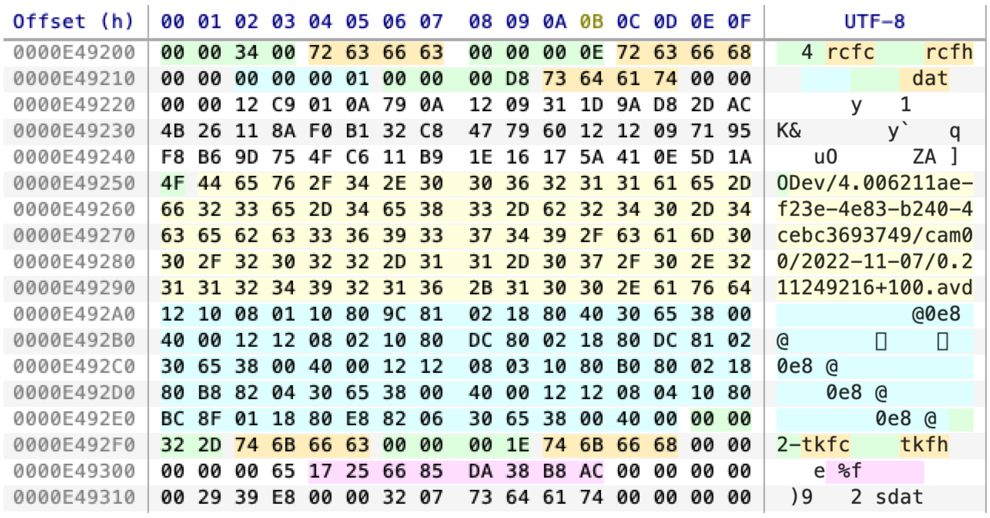


```
Offset (h)  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F      UTF-8
0000002000  00 40 4E 00 64 61 74 70  00 00 00 00 00 00 00 01   @N datp
0000002010  00 00 00 00 00 00 05 52  00 01 00 65 00 00 01 74          R  e   t
0000002020  00 00 01 67 4D 40 32 95  A0 0A 00 2D 69 B0 10 00      gM@2    -i
0000002030  00 00 01 68 EE 3C 80 00  00 00 01 06 F0 2C 10 44      h            D
0000002040  EF 0C 11 43 26 B0 89 0B  9A FC 30 93 8D DF 77 7A      C&      0    z
...
0000406E00  00 40 2E 00 64 61 74 70  00 00 00 00 00 00 00 02   @. datp
0000406E10  00 00 00 00 00 00 05 52  00 01 00 65 00 00 00 BA          R  e
0000406E20  00 00 01 06 F0 2C 03 45  8A 82 11 43 27 56 64 64          E  C'Vdd
0000406E30  BD D8 AC F1 BE 4F A7 99  F6 70 AB 4A 7C 24 44 CD   ޛ           p J|$D
0000406E40  17 0C 15 90 EE 88 27 DC  B4 49 FD 38 FF FF FF FF          ▯ I 8
```

Figure 4.14: Two frame-group headers in Avigilon .avd file. (Offsets described in Table 4.11)

Offset	Field type	Data type	Description
0x0	size	uint32	length of frame-group
0x4	signature	char[4]	Frame-group signature `datp`
0xC	number	uint32	number within index
0x14	number	uint32	counter of frame-group
0x18	number	uint16	decimal of counter (1-10)
0x20	data		Video data

Table 4.11: Structure of frame-group header in Avigilon .avd file

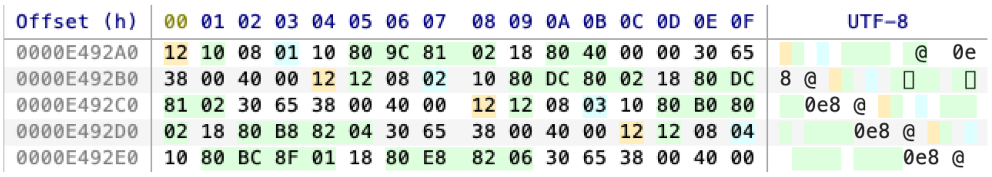


```
Offset (h)  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F      UTF-8
0000E49200  00 00 34 00 72 63 66 63  00 00 00 0E 72 63 66 68    4 rcfc    rcfh
0000E49210  00 00 00 00 00 01 00 00  00 D8 73 64 61 74 00 00              dat
0000E49220  00 00 12 C9 01 0A 79 0A  12 09 31 1D 9A D8 2D AC          y   1
0000E49230  4B 26 11 8A F0 B1 32 C8  47 79 60 12 12 09 71 95    K&        y`   q
0000E49240  F8 B6 9D 75 4F C6 11 B9  1E 16 17 5A 41 0E 5D 1A       uO       ZA ]
0000E49250  4F 44 65 76 2F 34 2E 30  30 36 32 31 31 61 65 2D    ODev/4.006211ae-
0000E49260  66 32 33 65 2D 34 65 38  33 2D 62 32 34 30 2D 34    f23e-4e83-b240-4
0000E49270  63 65 62 63 33 36 39 33  37 34 39 2F 63 61 6D 30    cebc3693749/cam0
0000E49280  30 2F 32 30 32 32 2D 31  31 2D 30 37 2F 30 2E 32    0/2022-11-07/0.2
0000E49290  31 31 32 34 39 32 31 36  2B 31 30 30 2E 61 76 64    11249216+100.avd
0000E492A0  12 10 08 01 10 80 9C 81  02 18 80 40 30 65 38 00                @0e8
0000E492B0  40 00 12 12 08 02 10 80  DC 80 02 18 80 DC 81 02    @        ▯    ▯
0000E492C0  30 65 38 00 40 00 12 12  08 03 10 80 B0 80 02 18    0e8 @
0000E492D0  80 B8 82 04 30 65 38 00  40 00 12 12 08 04 10 80        0e8 @
0000E492E0  BC 8F 01 18 80 E8 82 06  30 65 38 00 40 00 00 00            0e8 @
0000E492F0  32 2D 74 6B 66 63 00 00  00 1E 74 6B 66 68 00 00    2-tkfc    tkfh
0000E49300  00 00 00 65 17 25 66 85  DA 38 B8 AC 00 00 00 00       e %f
0000E49310  00 29 39 E8 00 00 32 07  73 64 61 74 00 00 00 00     )9   2 sdat
```

Figure 4.15: rcfc-metadata in Avigilon .avd file. (Offsets described in Table 4.12)

Offset	Field type	Data type	Description
0x0	size	uint32	length of rcfc field
0x4	signature	char[4]	Index signature `rcfc`
0x8	size	uint32	length of rcfh field
0xC	signature	char[4]	Index signature `rcfh`
0x12	number	uint32	counter for set of frame-group
0x16	size	uint32	length of sdat field
0x1A	signature	char[4]	Index signature `sdat`
0x50	size	uint8	length of filename
0x51	string	char[x]	filename
0xA0	index		index of preceeding datp frame-groups. Described in Table 4.13.
0xEE	size	uint32	length of tkfc field
0xF2	signature	char[4]	Index signature `tkfc`
0xF6	size	uint32	length of tkfh field
0xFA	signature	char[4]	Index signature `tkfh`
0x104	timestamp	uint64	frame-group timestamp (APFS time)

Table 4.12: Structure of rcfc-metadata in Avigilon .avd file



```
Offset (h)  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F     UTF-8
0000E492A0  12 10 08 01 10 80 9C 81  02 18 80 40 00 00 30 65            @  0e
0000E492B0  38 00 40 00 12 12 08 02  10 80 DC 80 02 18 80 DC  8 @          
0000E492C0  81 02 30 65 38 00 40 00  12 12 08 03 10 80 B0 80     0e8 @
0000E492D0  02 18 80 B8 82 04 30 65  38 00 40 00 12 12 08 04          0e8 @
0000E492E0  10 80 BC 8F 01 18 80 E8  82 06 30 65 38 00 40 00            0e8 @
```

Figure 4.16: rcfc-index in Avigilon .avd file, from offset 0xA0 in Figure 4.15. (Offsets described in Table 4.13)

Offset	Field type	Data type	Description
0x0	signature	char[1]	Index signature `0x12`
0x1	size	uint8	length of index-line
0x3	number	uint8	Index-line counter
0x5	number	leb128	size of datp
0xA	number	leb128	offset of datp

Table 4.13: Structure of rcfc-index in Avigilon .avd file

4.4.4 Proposed carving method

The content of rcfc-metadata within .avd files contains searchable timestamp information with a distinct signature. In order to retrieve the frame-group data, we analyze the index lines 100 bytes prior to the timestamp. Each indexline has the datp size and offset from the start of the original .avd file.

The proposed carving method searches all bytes before the rcfc-metadata and creates a list of offsets and sizes of datp frame-groups. When we find rcfc-metadata, the indexlines are compared to the list of offsets and sizes to verify that they are of the same origin. The logical file start is calculated and used as a reference for carving the next datp frame-groups. We consider the frame-groups contiguous if they share the same logical file start, and consecutive frame-groups are carved out as one file.

The regex pattern for the timeframe must cover a period from 11. Oct 20:00 UTC to 16. Oct 15:00 UTC. In APFS timestamp format, the six bytes with the lowest significance can be any value, and the seventh byte must range from `0x1D` to `0x1E`, and the most significant byte is `0x17`. This gives a range from `0x171D000000000000` (11.10.2022 11:23 UTC) to `0x171EFFFFFFFFFFFF` (17.10.2022 23:46 UTC). And a timeframe regex pattern of `.\x17[\x1D-\x1E].{6}`

Code listing 4.3: Carving Avigilon videodata

```
input: imagefile, regex for timeframe search
output: files of carved data

load imagefile
read chunk of data
  search chunk for datp frame-group #(.{4}datp)
  if datp found:
    read datp-header info into list

  regex-search chunk for rcfc-metadata timestamp #(.{4}rcfc.*tkfh.{6}<TIMEFRAME>)
  if metadata found:
    read indexlines.
    if indexlines matches datp-list
      set logical file start
      if carve start not set, set to first datp
      set carve end to last datp

  if new carve start
    save carved data to file
  read next chunk of data
```

4.5 Detec

4.5.1 Setup

Detec Next version 2.1.2207 [41] was installed on a clean Windows 10 operating system, and seven cameras were configured. We set up the system to record continuously and with a deletion policy of 7 days. The first recording period was conducted from 26. Sept 2022 20:00 UTC to 1. Oct 2022 17:38 UTC, and the second period from 30. Oct 2022 to 1. Nov 2022. The system reported at this point that the data storage used 50 percent of the allocated storage. When the second recording period started, the system did not delete any of the existing files.

4.5.2 Filestructure of recordings

After installation, the system allocates almost the entire media disk with 457 GB of 465 GB and creates only files and no folders. Two files for disk information and overview were made, "Disk0000.info" with details of the size of the available space and information on the size and amount of .sector files, and "Disk0000.index" which contains an index of the offsets and sizes of frames within the .sector files. Figure 4.17 shows an excerpt of the file structure.

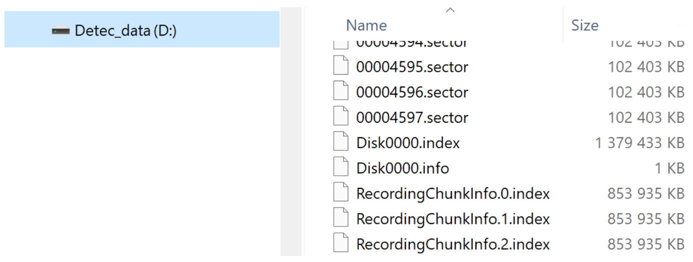
The system split 449 GB into 104,9 MB files and stored them with increasing numbering, named "00000000.sector" to "00004597.sector". A quick content analysis of these files shows that they contain video data. The .sector files are considered one big continuous file hence offsets in "Disk0000.index" are counted from file 0 throughout the last .sector file.

Last, there are a series of index files and database files with similar naming, "RecordingChunkInfo.[0-7].index" and "RecordingInfo.[0-7].database". The index files contain timestamps and references to frames stored in .sector files. The offsets and size of frames are located with a line number in Disk0000.index.

The databases "RecordingInfo.[0-7].database" are SQLite 3 databases which contain one table with five columns: "RecordingId", "RecordingFrom", "RecordingTo", "FirstRecordingChunkId" and "DataChannelId". The database keeps track of which camera the different frame-data chunks belong to. The column "DataChannelId" contains binary data to identify the camera, columns "RecordingFrom", "RecordingTo", and "FirstRecordingChunkId" refer to timestamps and line number in the corresponding "RecordingChunkInfo.[0-7].index".

4.5.3 Filecontent of relevant files

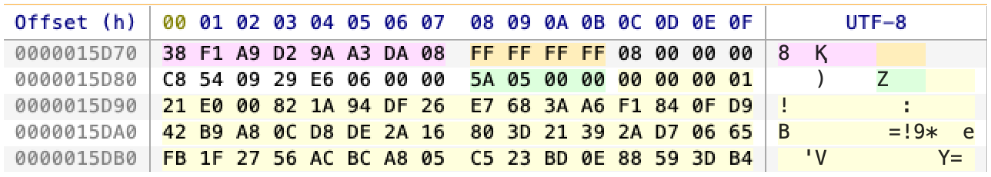
All 00000000.sector files are allocated by the system when configuring the media storage. The files are allocated in a contiguous manner, so when 00000000.sector ends 00000001.sector starts in the next cluster. A small header with a timestamp, signature, and frame size precedes each frame. The timestamp is in Windows Ticks format little-endian, followed by four bytes with `0xFF`, and last in the header is



```
Detec_data (D:)     Name                         Size
                    00004594.sector              102 403 KB
                    00004595.sector              102 403 KB
                    00004596.sector              102 403 KB
                    00004597.sector              102 403 KB
                    Disk0000.index             1 379 433 KB
                    Disk0000.info                      1 KB
                    RecordingChunkInfo.0.index   853 935 KB
                    RecordingChunkInfo.1.index   853 935 KB
                    RecordingChunkInfo.2.index   853 935 KB
```

Figure 4.17: Files automaticly generated by Detec on the mediadisk.

the size of the frame as a uint32 little-endian. The length of the header is 28 bytes, and the frame data follows the header, described in Figure 4.18 and Table 4.14. The next frame header starts immediately after the frame data.



```
Offset (h)  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F   UTF-8
0000015D70  38 F1 A9 D2 9A A3 DA 08  FF FF FF FF 08 00 00 00  8  K
0000015D80  C8 54 09 29 E6 06 00 00  5A 05 00 00 00 00 00 01    )    Z
0000015D90  21 E0 00 82 1A 94 DF 26  E7 68 3A A6 F1 84 0F D9  !        :
0000015DA0  42 B9 A8 0C D8 DE 2A 16  80 3D 21 39 2A D7 06 65  B       =!9*   e
0000015DB0  FB 1F 27 56 AC BC A8 05  C5 23 BD 0E 88 59 3D B4    'V         Y=
```

Figure 4.18: Frame header and content in Detec .sector file. (Offsets described in Table 4.14)

Offset	Field type	Data type	Description
0x0	timestamp	uint64	frame timestamp (Windows Ticks)
0x8	signature	char[4]	`0xFF FF FF FF`
0x18	number	uint32	size of frame
0x1C	data		Video data

Table 4.14: Frame header and content in Detec .sector file.

4.5.4 Proposed carving method

The header for each frame in the .sector files contains a searchable timestamp with a distinct signature and information about the frame size. This makes carving for frames a simple task since the header contains all the necessary information, the start offset is the first byte of the timestamp, and the full length of the carved data is the frame size plus 28 bytes. We consider the frames contiguous if a new frame starts directly after the end of the previous one, and all consecutive frames are carved out as one big file.

The regex pattern for the timeframe must cover a period from 26. Sept 20:00 UTC to 1. Oct 17:38 UTC. In Microsoft Ticks timestamp format, the five bytes with the lowest significance can be any value, the sixth byte must range from `0x9F` to `0xA3`, and the two most significant bytes are `0xDA08`. This gives a range from

`0x00000000009FDA08` (25.09.2022 14:12 UTC) to `0xFFFFFFFFFFA3DA08` (01.10.2022 22:55 UTC).
And a timeframe regex pattern of `.{5}[\x9F-\xA3]\xDA\x08`

Code listing 4.4: Carving Detec videodata

```
input: imagefile, regex for timeframe search
output: files of carved data

load imagefile
read chunk of data
  regex-search chunk for frame headers #(<TIMEFRAME> FF FF FF FF)
  if frame header found:
    read frame header and body
    if last frame is contiguous with this one:
      continue
    else:
      save carved data to file
  read next chunk of data
```

4.6 Agent DVR

Agent DVR is a VMS available for Windows, Linux, MacOS, and Raspberry Pi [42]. There is one installation package for Windows and one for Linux, MacOS, and Raspberry Pi, and we installed both packages to discover potential differences.

4.6.1 Setup Windows

Agent DVR for Windows version 4.2.4 was installed on a clean Windows 10 operating system, and seven cameras were configured. We set up the system to record continuously and with a deletion policy of 168 hours (seven days). The first recording period was conducted from 5. Oct 2022 13:40 UTC to 10. Oct 2022 16:00 UTC, at this time, the recordings had reached the maximum capacity of the storage disk and overwritten the first day of recordings. The earliest recording that was present was 6. Oct 2022 19:21. The second period lasted from 4. Nov 2022 to 6. Nov 2022, and at this time, the system reported that the data storage used 245 GB of 465 GB of the allocated storage (52 percent). When the second recording period started, the system deleted all the files from the previous recordings.

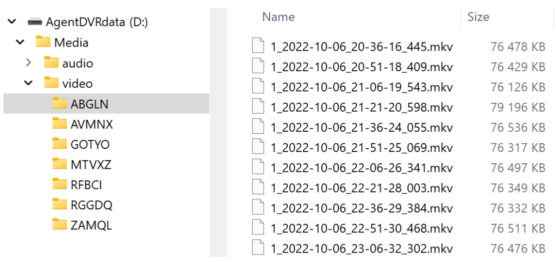
4.6.2 Setup Linux

Agent DVR for Linux version 4.2.6 was installed on a clean Ubuntu 22.04 operating system, and the camera configuration was copied from the Windows installation and implemented in the Linux installation. We set up the system to record continuously and with a deletion policy of 168 hours (seven days). The first recording period was conducted from 5. Oct 2022 20:45 UTC to 10. Oct 2022 07:30 UTC, at this time, the recordings had reached the maximum capacity of the storage disk and overwritten the first day of recordings. The earliest recording that was present was 6. Oct 2022 15:58 UTC. The second period lasted from 6. Nov 2022 to 7. Nov 2022, and at this time, the system reported that the data storage used 247 GB of 465 GB of the allocated storage (53 percent). When the second recording period started, the system deleted all the files from the previous recordings.

4.6.3 Filestructure of recordings

After installation, the system creates a folder called 'Media' with two sub-folders, 'audio' and 'video'. The video folder has separate folders for each camera configured in the system, named with an internal ID reference. The system stores camera video data in the folders with the date and time as their name. Each video file lasts 15 minutes and is stored in a Matroska Multimedia container. This structure is equal in the Windows and Linux setup (Figure 4.19)

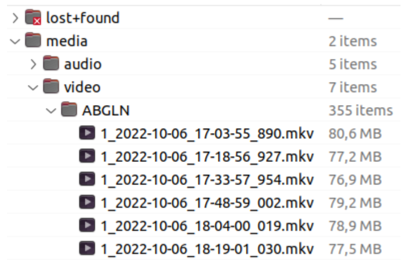
Matroska is a multimedia container that can hold various audio and video streams. It supports all known audio and video compression formats by design, making it an ideal container for video data from different IP cameras. Matroska files store the video stream in the original format, such as MJPEG, H.264, and H.265. Additionally, it is based on Extensible Binary Meta Language (EBML), allowing users to include metadata such as recorded time, duration, and title in the file [43].



```
AgentDVRdata (D:)
  Media
    audio
    video
      ABGLN
      AVMNX
      GOTYO
      MTVXZ
      RFBCI
      RGGDQ
      ZAMQL

Name                                  Size
1_2022-10-06_20-36-16_445.mkv    76 478 KB
1_2022-10-06_20-51-18_409.mkv    76 429 KB
1_2022-10-06_21-06-19_543.mkv    76 126 KB
1_2022-10-06_21-21-20_598.mkv    79 196 KB
1_2022-10-06_21-36-24_055.mkv    76 536 KB
1_2022-10-06_21-51-25_069.mkv    76 317 KB
1_2022-10-06_22-06-26_341.mkv    76 497 KB
1_2022-10-06_22-21-28_003.mkv    76 349 KB
1_2022-10-06_22-36-29_384.mkv    76 332 KB
1_2022-10-06_22-51-30_468.mkv    76 511 KB
1_2022-10-06_23-06-32_302.mkv    76 476 KB
```

(a) AgentDVR - Windows



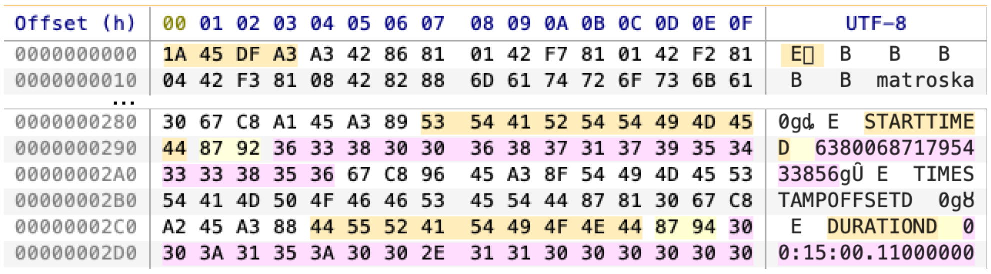
```
lost+found                            —
media                                 2 items
  audio                               5 items
  video                               7 items
    ABGLN                             355 items
      1_2022-10-06_17-03-55_890.mkv   80,6 MB
      1_2022-10-06_17-18-56_927.mkv   77,2 MB
      1_2022-10-06_17-33-57_954.mkv   76,9 MB
      1_2022-10-06_17-48-59_002.mkv   79,2 MB
      1_2022-10-06_18-04-00_019.mkv   78,9 MB
      1_2022-10-06_18-19-01_030.mkv   77,5 MB
```

(b) AgentDVR - Linux

Figure 4.19: File structure generated by AgentDVR - Windows and Linux.

4.6.4 Filecontent of relevant files

The .mkv files contain valid Matroska formatted data but contain no timestamp information binary stored throughout the file. However, there is a timestamp for the entire 15-minute video written in the metadata of the Matroska header. The value is referred to by the attribute name STARTTIME and consists of a number stored as ASCII text. The number is a Windows Ticks timestamp denoting the timestamp for the first frame in the video; as in Figure 4.20, the timestamp 638006871795433856 equals 2022-10-06 21:06:19.



```
Offset (h)  00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F   UTF-8
0000000000  1A 45 DF A3 A3 42 86 81  01 42 F7 81 01 42 F2 81    E▯  B   B   B
0000000010  04 42 F3 81 08 42 82 88  6D 61 74 72 6F 73 6B 61    B   B  matroska
   ...
0000000280  30 67 C8 A1 45 A3 89 53  54 41 52 54 54 49 4D 45   0gď E  STARTTIME
0000000290  44 87 92 36 33 38 30 30  36 38 37 31 37 39 35 34   D  6380068717954
00000002A0  33 33 38 35 36 67 C8 96  45 A3 8F 54 49 4D 45 53   33856gÛ E  TIMES
00000002B0  54 41 4D 50 4F 46 46 53  45 54 44 87 81 30 67 C8   TAMPOFFSETD  0gȣ
00000002C0  A2 45 A3 88 44 55 52 41  54 49 4F 4E 44 87 94 30    E  DURATIOND  0
00000002D0  30 3A 31 35 3A 30 30 2E  31 31 30 30 30 30 30 30   0:15:00.11000000
```

Figure 4.20: File header in AgentDVR .mkv file. (Offsets described in Table 4.15)

Offset	Field type	Data type	Description
0x0	signature	char[4]	EMBL signature (`0x1A 45 DF A3`)
0x287	signature	char[10]	Metadata name: STARTTIMED
0x293	timestamp	char[18]	File timestamp (Windows Ticks)
0x31C	data		Video data

Table 4.15: File header in AgentDVR .mkv file.

4.6.5 Carving, and proposed method for filtering

Since the video data isn't stored with a high timestamp frequency, with timestamps for every frame or frame-group, we are unable to propose a new carving method for this system. However, since the video data is stored as MKV files and are valid multimedia files, using existing tools to carve for multimedia files is most beneficial. It is possible to then highlight the relevant files from the result by inspecting the carved files' metadata and matching the timestamp against a timeframe regex.

The regex pattern must cover a period from 05.10.2022 13:40 UTC to 10.10.2022 16:00 UTC. In Windows Ticks format, the 13 digits with the lowest significance can be any value, the 14th digit must range from 0 to 1, and the four most significant digits are 6380. This gives a range from 638000000000000000 (28.09.2022 22:13 UTC) to 638019999999999999 (22.10.22 01:46 UTC).
And a timeframe regex pattern of `6380[0-1].{13}`

Code listing 4.5: Carving AgentDVR videodata

```
input: imagefile, regex for timeframe search
output: files of carved data

Perform carving with an existing tool for mkv files

load result file from caving
  read header of file
  regex-search header for metadata #(STARTTIME .{3}<TIMEFRAME>)
  if metadata found:
    highlight file result
  else:
    move file to another folder
  read next result file from carving
```

4.7 Summary of methodology

In summary, we found that almost all of the VMS systems we examined store their video data in a proprietary format, and only one uses a standard format. The different proprietary formats include timestamps embedded within the video data, enabling us to propose methods that carve out the video data within a specific time frame, even down to individual frames or frame-groups.

Chapter 5

Results

To get an overview of the content of the discs and an indication of how much data has not been overwritten and can potentially be recovered, we use a method of calculating the entropy and the hash value for each 512-byte sector on the disk. Using blockhashing to compare small pieces of data was described by Garfinkel *et al.* [44] as sub-file forensics and by Hansen [45] as block-hashing. A sector on the disk that could potentially be recovered is a sector with equal hash before and after the overwriting and an entropy above a threshold. A lower entropy threshold could be used to find sectors that are not empty and contains only zeros, and a higher threshold could find sectors that contain compressed data, which coincides with video data. An entropy of zero would indicate that all bytes in the sector are of the same value, typically only zeros. We deemed a sector as potentially non-overwritten if the hash values were equal and the entropy was greater than zero because we wanted to include index, metadata, and video data. In addition to these calculations, we created visualizations for each surveillance system, one of what was equal between the two recordings and another of what was actually recovered.

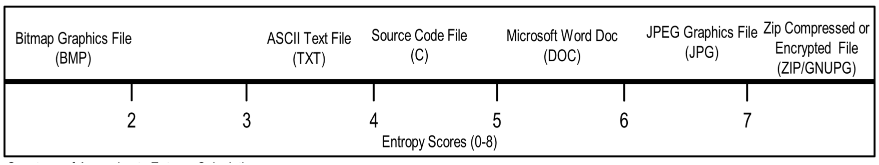


Figure 5.1: Spectrum of Approximate Entropy Calculations [46]. Shows that compressed data, images, and video has a high entropy. And text and uniform data (bitmap) have a lower entropy.

Casey *et al.* [47] defined a vocabulary for file recovery, and we will follow this standard as much as possible. And to define the files or data we are trying to recover in our experiment, we will use the terms overwritten and not-overwritten. We are looking to recover data from the first round of recordings that the surveillance systems deleted, and this data is classified as 'non-allocated' in the file system. For our experiment, this data is either **'overwritten'** with data from round two or **'non-overwritten'** and can potentially be recovered. The **'allocated'** data in the file system is mainly files from the second round of recordings, and files for configuration, logs, etc., and should not be in the recovered data.

For our recovered data, the correct term by the standard of Casey *et al.*[47] is to use 'Content Recovered' since most of our methods focus on only recovering the content of the files or even just fragments of the content, and not filename or metadata. We will use the short name 'Recovered' for Content Recovered since we are not using any other classes.

To measure precision and recall for our methods, we will consider the non-overwritten data as 'Positives' and the overwritten data as Negatives. The positives are data from the first round that is non-allocated and potentially still available, and the negatives are no longer existing. To visualize these quantities and simultaneously show the fragmentation of the data, we created figures for each system that show the entire disk. The first figure shows where there are non-overwritten parts, and the second figure shows the recovered areas of the disk, both marked with a green color.

After we did the recovery of the various systems, we logged all offsets from which the method recovered data. These offsets became the next part of calculating precision and recall.

- The non-overwritten data that was recovered was considered True Positives.
- The allocated data that was recovered was considered False Positives.
- The non-overwritten data that was not recovered was considered False Negative
- The allocated data that was not recovered was considered True Negative.

To calculate the precision and recall, we used the following formulas:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

5.1 Results from carving

In this section, we present the results from the carving experiments. We list the predicted and actual results in both numbers and visualizations for each system, together with the calculated precision and recall. And last, we summarize all the results in Table 5.3.

In the visualization, each pixel is a sum of several sectors (256 or 128 sectors per pixel) and gets a green color based on the results for each sector. For example, for a given pixel that represents 256 sectors, if there are no recovered sectors, the pixel will get a black color. If only half of the sectors are recovered, the color will be dark green. And if all of the sectors are recovered, the pixel will be entirely green.

5.1.1 Milestone

The disk contained a total of 465 GB (976 773 168 sectors) and after the disc was overwritten with 50 % new data in the second recording period there was 220 GB (461 642 936 sectors) of data that was non-overwritten (Figure 5.2a).

Using an existing carving tool resulted in zero files carved from unallocated areas of the disk.

The proposed carving method was implemented as a Python script and ran on the analysis computer for 4 hours and 35 minutes. The result was a total of 216 GB of data, 20831 files; the largest files were 32 MB, the smallest was 24 KB, and the average size was 10.6 MB. The recovered data is visualized in Figure 5.2b.

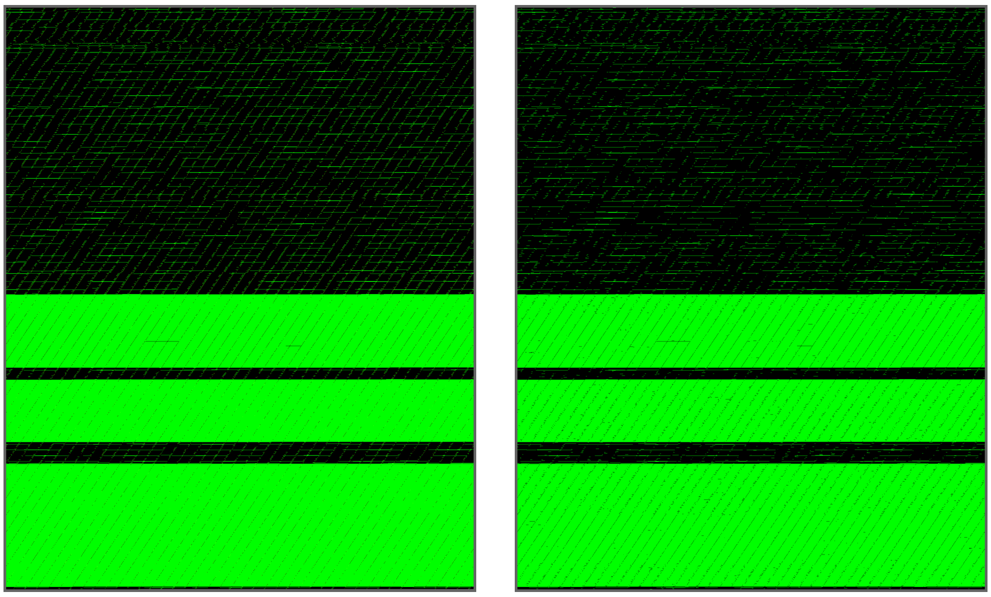

(a) Milestone non-overwritten data **(b)** Milestone recovered data

Figure 5.2: x for Milestone (256 sectors per pixel)

The analysis classified the recovered data as: 214 GB from non-overwritten data and 2 GB from allocated data. However, 6.1 GB of non-overwritten data was not recovered, and 243.6 GB of allocated data was correctly not recovered. This results in a precision of 0.990 and a recall of 0.972.

The timeframe for the carving was from 22 Sept 18:50 to 26 Sept 19:25; the first timestamp recovered was 22 Sept 19:01, and the last was 26 Sept 19:25. This means that we have video recovered from both ends of the recording.

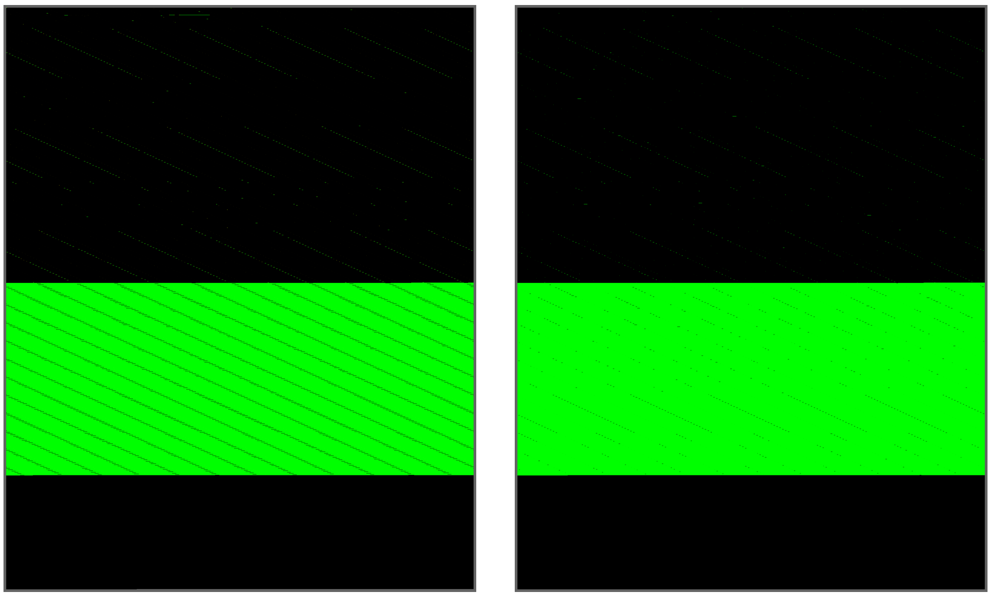
The size of the frame-groups recovered varied from 116 bytes to 1.5 MB (1579319 bytes), and the average size of a frame-group was 238 KB (244015 bytes)

5.1.2 Mirasys

The disk contained a total of 232.9 GB (488 382 464 sectors) and after the disc was overwritten with 50 % new data in the second recording period there was 75.4 GB (158 039 672 sectors) of data that was non-overwritten (Figure 5.3a).

Using an existing carving tool resulted in zero files carved from the entire disk.

The proposed carving method was implemented as a Python script and ran on the analysis computer for 1 hours and 2 minutes. The result was a total of 77 GB of data, 3269 files; the largest files were 46 MB, the smallest was 5.9 KB, and the average size was 24.1 MB. The recovered data is visualized in Figure 5.3b.



(a) Mirasys non-overwritten data **(b)** Mirasys recovered data

Figure 5.3: Disk overview of the data for Mirasys (128 sectors per pixel)

The analysis classified the recovered data as: 75.3 GB from non-overwritten data and 1.7 GB from allocated data. However, 97 MB of non-overwritten data was not recovered, and 155.8 GB of allocated data was correctly not recovered. This results in a precision of 0.998 and a recall of 0.977.

The timeframe for the carving was from 11 Jan 12:42 to 15 Jan 23:07; the first timestamp recovered was 11 Jan 12:43, and the last was 15 Jan 23:07. This means that we have video recovered from both ends of the recording.

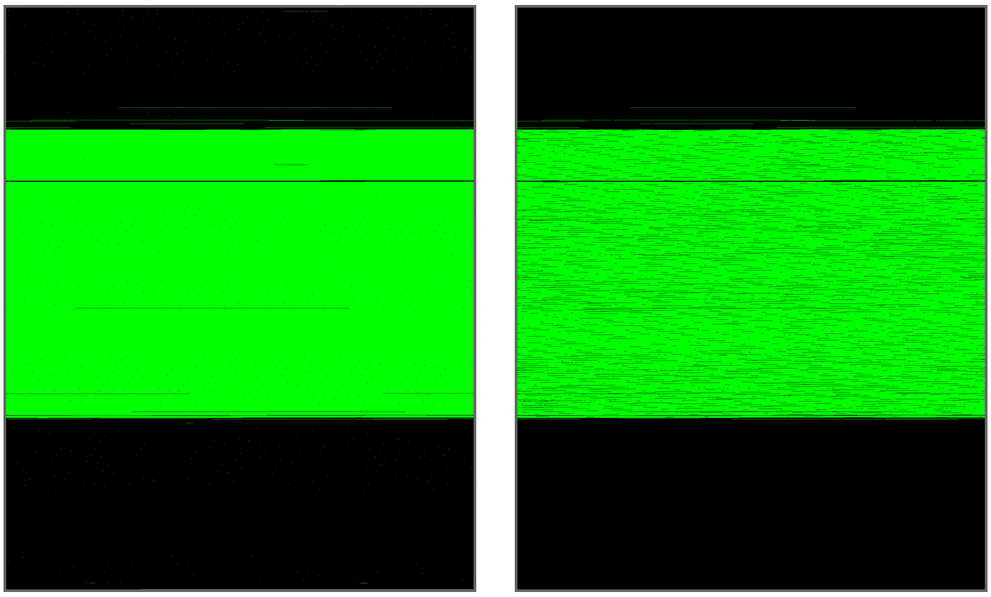
The size of the frame-groups recovered varied from 5954 bytes to 905 KB (926834 bytes), and the average size of a frame-group was 26 KB (27044 bytes)

5.1.3 Avigilon

The disk contained a total of 465 GB (976 769 023 sectors) and after the disc was overwritten with 50 % new data in the second recording period there was 230 GB (482 281 276 sectors) of data that was non-overwritten (Figure 5.4a).

Using an existing carving tool resulted in zero files carved from the entire disk.

The proposed carving method was implemented as a Python script and ran on the analysis computer for 2 hours and 31 minutes. The result was a total of 217.2 GB of data, 7780 files; the largest files were 128 MB, the smallest was 6.5 KB, and the average size was 28.5 MB. The recovered data is visualized in Figure 5.4b.



(a) Avigilon non-overwritten data **(b)** Avigilon recovered data

Figure 5.4: Disk overview of the data for Avigilon (256 sectors per pixel)

The analysis classified the recovered data as: 217.1 GB from non-overwritten data and 37 MB from allocated data. However, 12.8 GB of non-overwritten data was not recovered, and 235.7 GB of allocated data was correctly not recovered. This results in a precision of 0.999 and a recall of 0.944.

The timeframe for the carving was from 11 Oct 20:00 to 16 Oct 15:00; the first timestamp recovered was 12 Oct 12:33, and the last was 15 Oct 22:35. This means there are about 16 hours from both ends of the recording we have no recovered video from.

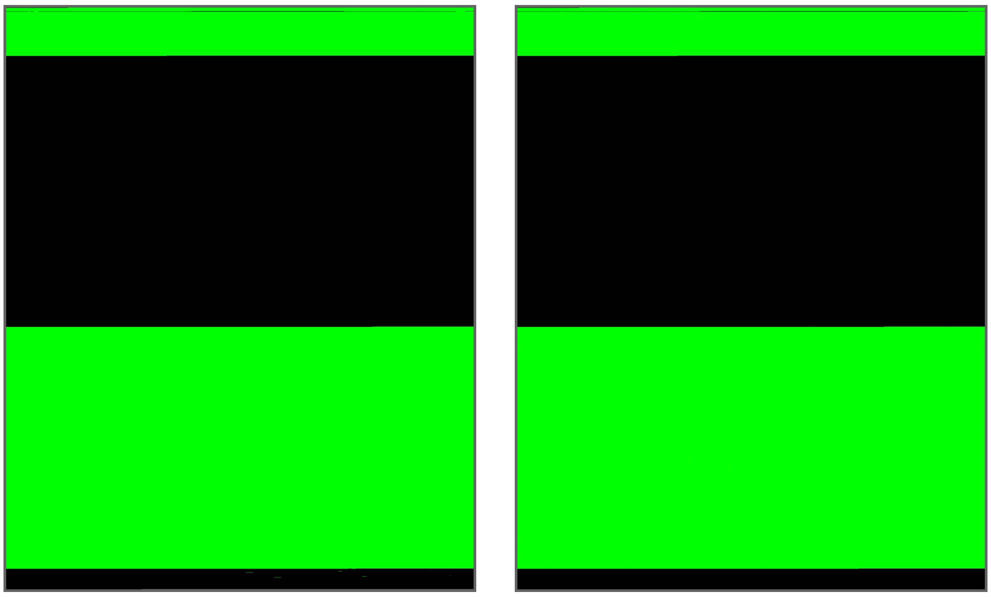
The size of the frame-groups recovered varied from 2218 bytes to 24 MB (25189376 bytes), and the average size of a frame-group was 1,9 MB (1986015 bytes)

5.1.4 Detec

The disk contained a total of 465 GB (976 773 168 sectors) and after the disc was overwritten with 50 % new data in the second recording period there was 232 GB (486 483 603 sectors) of data that was non-overwritten (Figure 5.5a).

Using an existing carving tool resulted in zero files carved from the entire disk.

The proposed carving method was implemented as a Python script and ran on the analysis computer for 5 hours and 14 minutes. The result was a total of 232 GB of data, 2375 files; the largest files were 101 MB, the smallest was 8 MB, and the average size was 99.9 MB. The recovered data is visualized in Figure 5.5b.



(a) Detec non-overwritten data **(b)** Detec recovered data

Figure 5.5: Disk overview of the data for Detec (256 sectors per pixel)

The analysis classified the recovered data as: 231.9 GB from non-overwritten data and 3 MB from allocated data. However, 76 MB of non-overwritten data was not recovered, and 233.8 GB of allocated data was correctly not recovered. This results in a precision of 0.999 and a recall of 0.999.

The timeframe for the carving was from 26 Sept 20:00 to 1 Oct 17:38; the first timestamp recovered was 29 Sept 18:32, and the last was 1 Oct 17:38. This means that we have no recovered video from the first three days of recordings, but all until the end of the recording.

The size of the frame-groups recovered varied from 972 bytes to 13 KB (13699 bytes), and the average size of a frame-group was 9 KB (9449 bytes)

5.1.5 AgentDVR-Windows

The disk contained a total of 465 GB (976 771 073 sectors) and after the disc was overwritten with 50 % new data in the second recording period there was 219.8 GB (461 036 543 sectors) of data that was non-overwritten (Figure 5.6a).

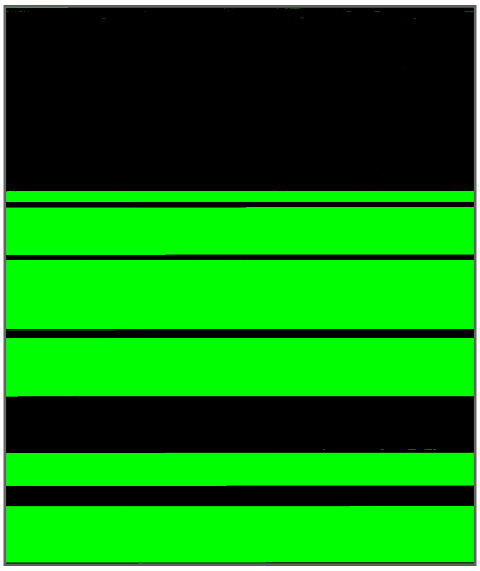
The first part of the carving method was done by loading the image of the disk into PhotoRec with standard settings, and selecting only to carve MKV files. PhotoRec ran for 2 hours and 43 minutes. The result was a total of 287.7 GB, 1850 files. This result was not narrowed down to the wanted time frame.

The second part of the carving was implemented as a Python script and ran on the analysis computer for 3 minutes. This moved the files outside the time frame to another location, and the remaining result was a total of 131.9 GB of data, 844 files; the largest files were 684 MB, the smallest was 11 MB, and the average size was 160 MB. The recovered data is visualized in Figure 5.6b.

The analysis classified the recovered data as: 129.6 GB from non-overwritten data and 2.3 GB from allocated data. However, 90.3 GB of non-overwritten data was not recovered, and 243.6 GB of allocated data was correctly not recovered. This results in a precision of 0.983 and a recall of 0.589.

The timeframe for the carving was from 5 Oct 13:40 to 10 Oct 16:00; the first timestamp recovered was 6 Oct 19:36, and the last was 10 Oct 15:49. This is consistent with the earliest recording after the first round; the storage had reached maximum capacity and started overwriting the first day of recordings. This means that we have video recovered from both ends of the recording.

The size of the files fragments varied from 84 KB (86016 B) to 916 MB (960544768 B), and the average size of a fragment was 98 MB (102759636 B)


(a) AgentDVR-Win non-overwr. data

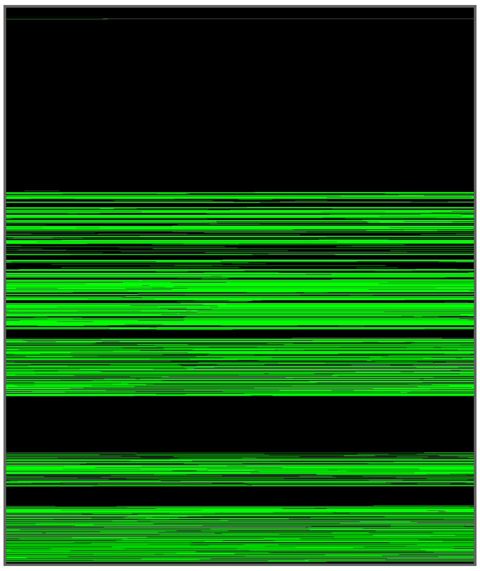

(b) AgentDVR-Win recovered data

Figure 5.6: Disk overview of the data for AgentDVR-Win (256 sectors per pixel)

5.1.6 AgentDVR-Linux

The disk contained a total of 465 GB (976 752 047 sectors) and after the disc was overwritten with 50 % new data in the second recording period there was 209 GB (438 272 446 sectors) of data that was non-overwritten (Figure 5.7a).

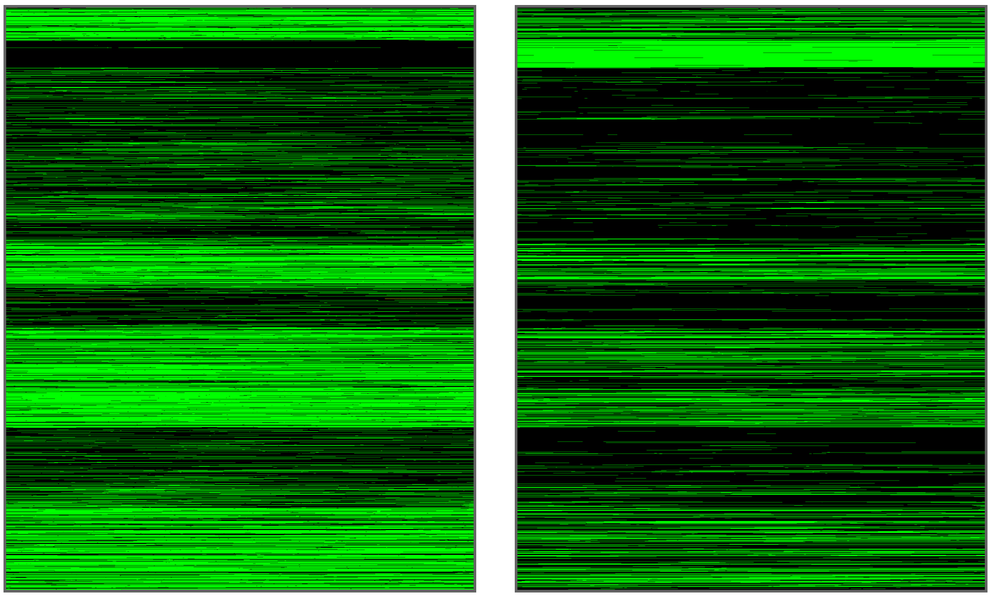
The first part of the carving method was done by loading the image of the disk into PhotoRec with standard settings, and selecting only to carve MKV files. PhotoRec ran for 1 hours and 17 minutes. The result was a total of 125 GB, 737 files. This result was not narrowed down to the wanted time frame.

For the second part we used the same Python script as for the Windows carving and ran on the analysis computer for 2 minutes. None of the files was outside the time frame, so no files was moved to another location. The result was a total of 125 GB of data, 844 files; the largest file was 22 GB, the smallest was 1 MB, and the average size was 173,6 MB. The recovered data is visualized in Figure 5.7b.

The analysis classified the recovered data as: 104 GB from non-overwritten data and 21 GB from allocated data. However, 105 GB of non-overwritten data was not recovered, and 235.7 GB of allocated data was correctly not recovered. This results in a precision of 0.831 and a recall of 0.497.

The timeframe for the carving was from 5 Oct 20:45 to 10 Oct 07:30; the first timestamp recovered was 5 Oct 23:17, and the last was 10 Oct 07:17. This means that we have video recovered from both ends of the recording.

The size of the files fragments varied from 4096 bytes to 1.9 GB (2113732608 B), and the average size of a fragment was 70 MB (73975505 B)



(a) AgentDVR-Linux non-overwritten data **(b)** AgentDVR-Linux recovered data

Figure 5.7: Disk overview of the data for AgentDVR-Linux (256 sectors per pixel)

5.1.7 Result overview

We summarize all the results of the recovered data in Table 5.1 and Table 5.2. The results from AgentDVR are in a separate table because the PhotoRec method isn't comparable to our methods. While PhotoRec operates at a file level, our methods operate at a much lower level; frames or framegroups. Also it is not within scope to compare a generic carver, such as PhotoRec, to our specific carving methods.

We then list the size of the chunks of data recovered. A chunk of data refers to the smallest unit a method recovers, including frames, frame-groups, and file fragments. If the data is recovered contiguously, a chunk of data refers to the individual parts, not the entire contiguous data chunk.

System	P	N	TP	FP	TN	FN	Precision	Recall
Milestone	220.1	245.6	214	2.0	243.6	6.1	0.990	0.972
Mirasys	75.4	157.5	75.3	1.7	155.8	0.1	0.998	0.977
Avigilon	230	235.8	217.1	0.037	235.7	12.8	0,999	0.944
Detec	232	233.8	231.9	0.003	233.8	0.076	0,999	0,999

Table 5.1: Precision and recall for recovered data (rounded to GB)

System	P	N	TP	FP	TN	FN	Precision	Recall
AgentDVR Win	219.8	245.9	129.6	2.3	243.6	90.3	0,983	0,589
AgentDVR Linux	209	256.8	104	21	235.7	105	0,831	0,497

Table 5.2: Precision and recall for carved video with PhotoRec (rounded to GB) After highlighting relevant time frame.

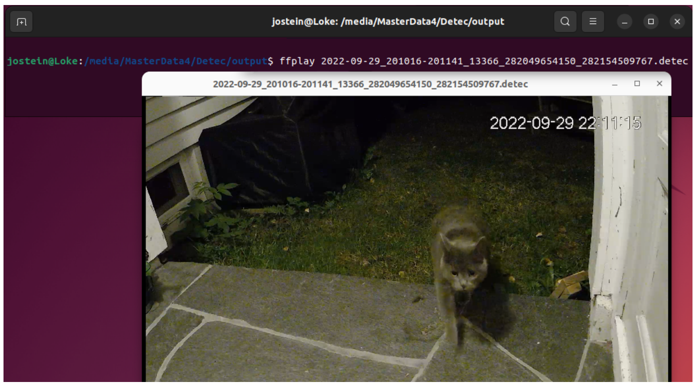
System	Min	Max	Average
Milestone	116 B	1.5 MB	238 KB
Mirasys	5.8 KB	0.9 MB	26 KB
Avigilon	2.2 KB	24 MB	1.9 MB
Detec	972 B	13 KB	9 KB
AgentDVR Win	84 KB	916 MB	98 MB
AgentDVR Linux	4 KB	2015 MB	70 MB

Table 5.3: Sizes of recovered chunks of data, chunks are frame-groups, frames or file fragments depending on the carving method.

5.2 Verification of content

In this section, we will briefly describe how we verified the content of the carved data. After we recovered the video data, we needed to confirm that the video was actually playable, and using a video player was an easy approach. Typically, surveillance systems come with their own players that can manage their unique formats, including headers and video data. But this requires the data to be complete with all surrounding files we are not recovering, such as index files and configuration.

To extract only the video data from the recovered data, we implemented a variant of the carving method. We used the same approach to read header information from the recovered files but concatenated the video data from one file into another. This new file consisted of only the video data, without headers in between. Usually, video players cannot render video data without container information, but *ffplay* can play a video stream without container information. Using *ffplay* in the *FFmpeg tool* [48] in Linux Ubuntu, stream specifiers can be given as arguments to the program if necessary. But in our experiment, we did not need these specifiers, *ffplay* could play all video formats we loaded, an example shown in Figure 5.8.


jostein@Loke: /media/MasterData4/Detec/output
2022-09-29_201016-201141_13366_282049654150_282154509767.detec
2022-09-29 22:11:15

```
jostein@Loke:/media/MasterData4/Detec/output$ ffplay 2022-09-29_201016-201141_13366_282049654150_282154509767.detec
```

Figure 5.8: Example of verifying a recovered file from Detec with *ffplay*. Note that the timeframe of the video file is 20:10:16 - 20:11:41 UTC, and the timestamp imprinted in the video file is 22:11:15 UTC +2 (local time Norway).

For video codecs with frame types that use other frames as a reference, similar to H.264 and its intra- and inter-frames, the carved video must contain an intra-frame that other inter-frames can refer to. The video needs a full frame to know what image to render; if the video consists only of inter-frames, the player can't render a full frame to calculate the changes in the inter-frames.

5.3 Summary of results

We implemented our suggested methods with Python scripts and ran the scripts on our experimental data. The implemented code is made available on GitHub[1].

Before and after overwriting, we carefully analyzed the data to gather information that we could use to calculate the success rate of our methods. We also

[1]`https://github.com/josvik/timeframe-carver`

created a visualization showing how much data we could recover. Lastly, we outlined a method for extracting playable content from the proprietary data and used this to verify the content we recovered.

Chapter 6

Discussion

6.1 Precision and Recall

In our initial analysis of the disks and the potentially recoverable data, we made a hash comparison between the disk after the two recordings. We compared sector by sector and counted the equal ones but not those with no data (zero entropy). But there are still some amounts of data on the disks that are equal but do not contain any video data that would be recovered. This is file system data (file and directory structure, journals, logs, etc.), configuration files, log files, index files, and other files used by the VMS. The values used for calculating precision and recall are, to some degree, affected by these types of data. They are calculated as positives but will never be recovered and considered True Positives. However, this is such a small amount of data that it wouldn't affect the calculations enough.

For the proposed carving methods we implemented (not PhotoRec), there was a recall of 0.944-0.999, and the parts considered false negatives, i.e., not recovered, were at most 0.06 or 6.1 GB out of 220.1 GB of data. The lowest value for false negatives was as low as 0.001 or 76 MB out of 219.8 GB of data. Most of this data is the type that is mistaken for recoverable data, and the actual recall value is even closer to 1.0. These results do not include those that used an existing carving tool (PhotoRec) because the results are not comparable.

Another key component that affects recall is how big the recovered chunks of data are. For the Detec system's method, the average frame-group size is 9 KB, and this method has the highest-scoring recall: 0.999, see Figure 6.1. The recovery of Avigilon data had a chunk size of 1.9 MB and a recall of 0.944. And the AgentDVR systems have average fragment sizes of about 100 MB and the lowest recall value. This means that a larger chunk of data will increase the possibility of corrupted data due to some part being overwritten, and therefore the carving will recover data from the second round of recording instead (overwritten data).

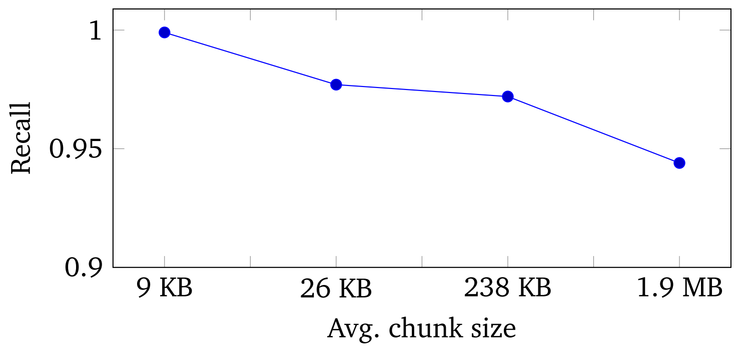


Figure 6.1: Recall value per chunk size.

6.2 Overwriting policy and recover period

In our experiments, we configured all the VMS to have the same policy for how long they kept their video data available for the consumer. We carried out the second round of recordings by a good margin to that policy; the policy was seven days, and the typical period between the two recordings was a month.

There are two significant differences in how the systems store their data; file-system-based storage and preallocated-based storage. These differences affect how the systems handle their deletion policy and overwrite old recordings, illustrated in Figure 6.2.

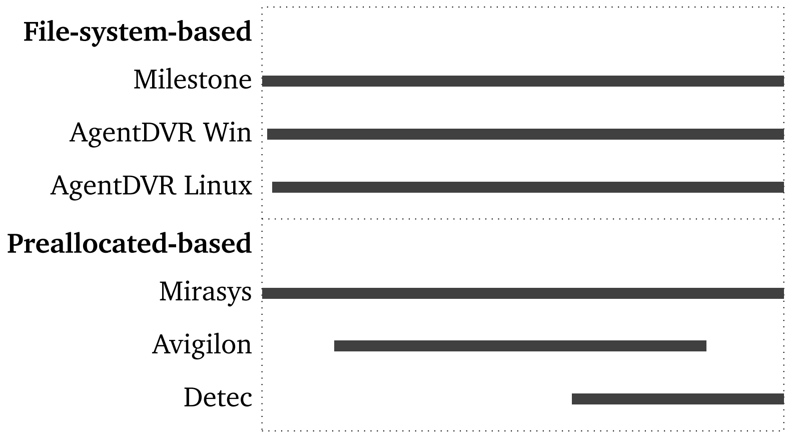


Figure 6.2: Periods of recovered data within the start and end of the first recording, all systems have a first recording period of approximately 100 hours.

The first type is the systems that use a file-system-based storage approach (Milestone and AgentDVR), where files with video data are stored under folders determined by the recording date and time. Each recording is a new file, delimited by size (Milestone) or time (AgentDVR). Milestone splits all files at 16 MB size and a separate storage folder for every hour, while AgentDVR stores the video data into files with a 15-minute duration. When the deletion policy is reached, the systems use the file-system's own deletion method, and the storage area for that video data is made available for future video data. The overwriting policy is then determined by the file-system, not by the VMS.

The other type is preallocated-based storage which allocates the available storage space at initialization (Mirasys, Avigilon, and Detec). The allocated files are not deleted in the file system but reused by the VMS. The allocated files are treated as an internal file system or a pool of available storage area, and the overwriting policy is handled by the VMS itself.

The difference in storage type creates different results in recovery; for file-system-based storage, the recovered data is evenly spread across the first recording period. And for two of the systems with preallocated based storage, big chunks of data are missing, especially from the oldest recordings. This indicates that the systems are overwriting the oldest recordings first.

6.3 Carving time / performance

There are significant variations in how much time the carving methods took, from just over an hour up to almost six hours. The disk sizes were practically identical with 500 GB drives, except for Mirasys, with a size of 230 GB. For this comparison, we multiply the timing with the difference for Mirasys, and we include the timing for both AgentDVR as a reference to an existing carver.

The fastest carving methods were Mirasys at 2 hours and 15 minutes (1:02 x (500/230)) and Avigilon at 2 hours and 31 minutes. Then the two slowest methods were Milestone at 4:35 and Detec at 5:41 shown in Figure 6.3. The main difference in these methods is that the fastest ones had an index structure stored within the video data, and the slowest ones only had frames or frame-groups stored.

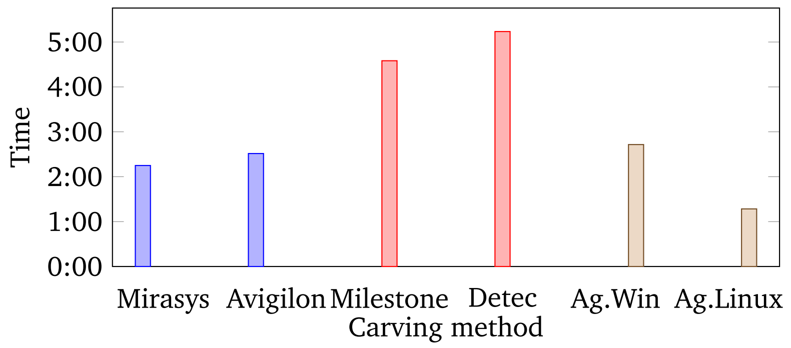


Figure 6.3: Timing of carving methods (in hours:minutes) The first two uses an index-based carving, and the next two a non-index-based carving. The last two is included as a reference to existing carver.

Carving with index information will drastically reduce the carving time and maintain high accuracy and recall values. Together with carving for indexes and the referenced frames, it's possible to carve for individual frames as long as the timestamp is stored in the frame header. By using the information in the index and verifying the offsets found in the index, it is possible to skip ahead while carving, as long as we, for example, find valid frame headers at the given offset. By utilizing this feature, there is no need to search every sector of the disk, therefore reducing the carving time.

6.4 File slack

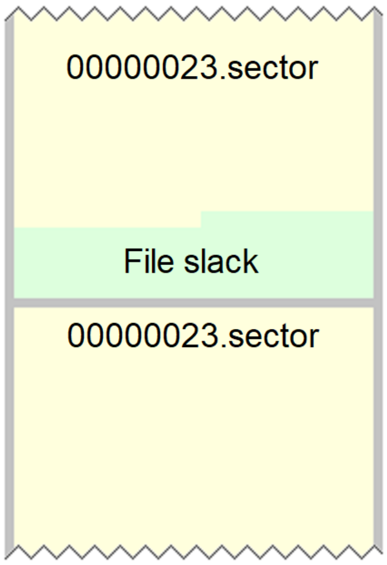
In all file systems, there is a given allocation size that's the smallest bit of addressable data; in NTFS, this is known as clusters [10], and in ext4, it's known as blocks [11]. A typical size is 4096 bytes, which means that if a file only contains 4000 bytes, there will be 96 bytes that are not in use by that file. This is known as file slack [14].

As mentioned, some VMS systems use preallocated-based storage where the files are considered one big filesystem. For example, the file system for Detec has allocated 4598 .sector files, each has a size of 104.9 MB (104 860 099 bytes), and the Disk0000.info file also describes each file as 104 860 099 bytes. But the file system is set up with a cluster size of 4096 bytes, and a .sector file will take up 25 601 clusters leaving 1 597 bytes in file slack that will never be used.

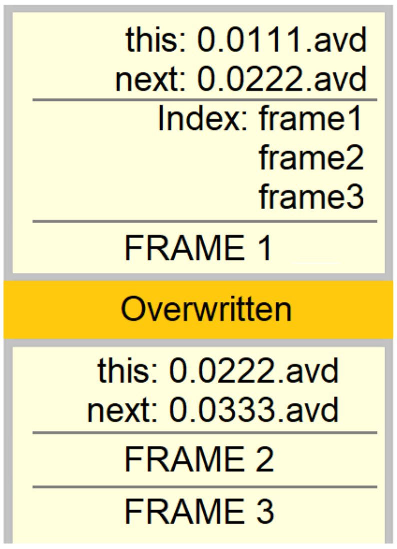
When the system writes video data to the .sector files, it will write until reaching the file size limit and then continue writing at the beginning of the following .sector file. This jump between the two .sector files might occur in the middle of a frame and, therefore, create over 1.5 KB of unrelated data in the middle of the frame (Figure 6.4a). Our method does not account for this file slack and will be unable to do contiguously carving over this jump.

A similar problem occurs for the Avigilon system. A .avd file has information on the following .avd file, but only by its file name. When a file is 'deleted', it is renamed and moved back to the 'FilePool' folder, and the reference between the .avd files is destroyed. The jump between two .avd files can also occur in the middle of a frame, and the Avilgilon system stores the timestamp only in the index. This means that if the index is stored in one .avd file and some of the frames in the following .avd file, we cannot recover the frames stored after the jump (Figure 6.4b).

There is a possibility to recover those frames using the same method as Frame-based recovery [2]. From the index, we have the size of all the missing frames, and by searching for individual frames that do not already belong to an index, we can compare the size of the frame and index information. The result is contiguously carved video data that spans multiple files.



(a) File slack in .sector files for Detec disrupts contiguous carving



(b) Reference between .avd files is dependent on file name and is lost when files are renamed

Figure 6.4: File slack in .sector files and jump in .avd files

6.5 Summary of discussion

In summary, there is a correlation between the size of the chunks we can identify with a timestamp and the recall value. If the chunks are smaller, the recall value is greater. We also found a correlation between the storage system, file-system-based, or preallocated-based storage, and how the VMS overwrites their data. This affects the recovery in such a way that there is a lower chance of recovering older data on preallocated-based storage.

We also found that the methods that utilize index information have a significantly shorter processing time than those that only carve for individual frames or frame-groups. And at last, we discussed the possibility of missing some data during the carving process due to file slack.

Chapter 7

Conclusion and Further Work

In this chapter we will revisit the research questions in Chapter 1 that laid the foundation for this research, and give answers based on the experimental work in chapter 4 and 5.

7.1 Research questions

RQ1 - Is it possible to recover surveillance video that is partially overwritten using a timeframe-based approach??

By analyzing the internal structures of the surveillance data, we found that all the systems we analyzed have timestamps stored within their surveillance data. And together with that information, we also found enough info to create carving methods based on the internal structure in almost all of the systems. We presented methods for carving data from Milestone, Mirasys, Avigilon, and Detec. For AgentDVR, we used an existing carving tool since the system stores the media files in a Matroska multimedia container. The structure-based carving methods we suggested showed very high performance on partly overwritten data, with precision values of 0.990-0.999 and recall values between .944 and .999.

RQ2 - Can existing common carving tools recover surveillance video?

Our experiment on video data from surveillance systems shows that existing carving tools are unable to recover the video data from a majority of the systems. The video data is stored in a proprietary structure which the carving tool is unfamiliar with. Out of all the analyzed systems, only the AgentDVR system stored video data in a commonly used video file format and produced results.

We showed that files from the AgentDVR system have a timestamp embedded in metadata, and we propose a method that highlights the results from an existing carving tool based on a timeframe of interest. Using existing tools to recover data was useful, but the performance was much poorer. Mainly because the video data was stored in big files, and new recordings could easily overwrite parts of the video data.

RQ3 - Is it possible to find a generic pattern for stored data of surveillance systems?

We found that there are patterns in the stored video data of most of the systems we analyzed, while there are some differences in how the data is structured and stored, there are key components in all of the system's data. The most crucial information is the timestamps, offsets, size values, and indexes. We found different types of timestamps; Windows Filetime, Windows Ticks, and APFS time. The timestamps were stored in both little and big endian, in addition to being stored as a number in ASCII text. We presented a method and a tool to highlight the timestamps in order to reveal the patterns in stored video data. We used this method and tool to analyze the systems in this thesis and showed an effective approach to discovering patterns and identifying timestamps and other data values.

7.2 Future work

There are possibilities to do further analysis on the surveillance data to increase the precision rates by recovering configuration and index files stored outside the video data. There is also potential in discovering and accounting for file slack within the carved data, increasing the recall even more. Similarly, by finding a method to reconnect the .avd files in the Avigilon system, there is a potential to recover more of the relevant data. Another potential is to analyze and find information about the video stream to combine individual camera streams from the same video stream.

By implementing a frame-based recovery method, we could enhance the carving tools to patch together individual frames into a more contiguous result. We could also use index information more actively, for example, in the Milestone system, to rearrange fragmented frames or frame-groups in the correct order.

There is also future work to find similarities in the different carving methods and develop software that handles all the methods in one program and even adds support to systems later to be analyzed. And another software development contribution is to enrich the pattern-finding tool with more timestamps and make it more generic by adding other signatures, such as video frame signatures. As noted earlier, the implemented code is made available on GitHub[1].

[1]https://github.com/josvik/timeframe-carver

Bibliography

[1] P.-C. Su, P.-L. Suei, M.-K. Chang and J. Lain, 'Forensic and anti-forensic techniques for video shot editing in H.264/AVC,' *Journal of Visual Communication and Image Representation*, vol. 29, pp. 103–113, 2015.

[2] G.-H. Na, K.-S. Shim, K.-W. Moon, S. G. Kong, E.-S. Kim and J. Lee, 'Frame-based recovery of corrupted video files using video codec specifications,' *IEEE Transactions on Image Processing*, vol. 23, no. 2, pp. 517–526, 2013.

[3] J. McCann, L. Quinn, S. McGrath and C. Flanagan, 'Video surveillance architecture at the edge,' EasyChair, Tech. Rep., 2022.

[4] A. Ariffin, J. Slay and K.-K. Choo, 'Data recovery from proprietary formatted CCTV hard disks,' in *IFIP International Conference on Digital Forensics*, Springer, 2013, pp. 213–223.

[5] B. Carrier, *File system forensic analysis*. Addison-Wesley Professional, 2005, ch. Part III.

[6] P. Nabity and B. Landry, 'Recovering deleted and wiped files: A digital forensic comparison of FAT32 and ntfs file systems using evidence eliminator,' *SWDSI*, 2013.

[7] 'FAT technical reference.' (Feb. 2023), [Online]. Available: `https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc758586(v=ws.10)`.

[8] K. L. Rusbarsky and K. City, 'A forensic comparison of NTFS and FAT32 file systems,' *http://www. marshall. edu/forensics/files/RusbarskyKelsey_Research-Paper-Summer-2012. pdf. Fetched: July*, vol. 6, p. 2017, 2012.

[9] M. Alazab, S. Venkatraman and P. Watters, 'Digital forensic techniques for static analysis of NTFS images,' in *Proceedings of ICIT2009, Fourth International Conference on Information Technology, IEEE Xplore*, 2009.

[10] 'NTFS.' (Feb. 2023), [Online]. Available: `https://www.ntfs.com/`.

[11] 'Ext4 filesystem.' (May 2023), [Online]. Available: `https://www.kernel.org/doc/html/v4.19/filesystems/ext4/`.

[12] A. Pal and N. Memon, 'The evolution of file carving,' *IEEE signal processing magazine*, vol. 26, no. 2, pp. 59–71, 2009.

[13] T. Laurenson, 'Performance analysis of file carving tools,' in *Security and Privacy Protection in Information Processing Systems: 28th IFIP TC 11 International Conference, SEC 2013, Auckland, New Zealand, July 8-10, 2013. Proceedings 28*, Springer, 2013, pp. 419–433.

[14] D. Povar and V. Bhadran, 'Forensic data carving,' in *Digital Forensics and Cyber Crime: Second International ICST Conference, ICDF2C 2010, Abu Dhabi, United Arab Emirates, October 4-6, 2010, Revised Selected Papers 2*, Springer, 2011, pp. 137–148.

[15] Y. Yang, Z. Xu, L. Liu and G. Sun, 'A security carving approach for AVI video based on frame size and index,' *Multimedia Tools and Applications*, vol. 76, no. 3, pp. 3293–3312, 2017.

[16] T. Gloe, A. Fischer and M. Kirchner, 'Forensic analysis of video file formats,' *Digital Investigation*, vol. 11, S68–S76, 2014.

[17] L. Berc, W. Fenner, R. Frederick, S. McCanne and P. Stewart, *RFC2435: RTP payload format for JPEG-compressed video*, 1998.

[18] T. Gloe, 'Forensic analysis of ordered data structures on the example of JPEG files,' in *2012 IEEE International Workshop on Information Forensics and Security (WIFS)*, IEEE, 2012, pp. 139–144.

[19] ISO/IEC 14496-2:2004, *Information technology — Coding of audio-visual objects — Part 2: Visual*. 2004.

[20] ISO/IEC 14496-10:2009, *Information technology — Coding of audio-visual objects — Part 10: Advanced Video Coding*. 2009.

[21] ISO/IEC 23008-2:2013, *Information technology — High efficiency coding and media delivery in heterogeneous environments — Part 2: High efficiency video coding*. 2013.

[22] G. J. Sullivan, J.-R. Ohm, W.-J. Han and T. Wiegand, 'Overview of the high efficiency video coding (HEVC) standard,' *IEEE Transactions on circuits and systems for video technology*, vol. 22, no. 12, pp. 1649–1668, 2012.

[23] 'Datetime.ticks property.' (Jan. 2023), [Online]. Available: `https://learn.microsoft.com/en-us/dotnet/api/system.datetime.ticks?view=net-7.0`.

[24] 'FILETIME structure (minwinbase.h).' (Jan. 2023), [Online]. Available: `https://learn.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-filetime`.

[25] 'The open group base specifications issue 7, 2018 edition.' (Jan. 2023), [Online]. Available: `https://pubs.opengroup.org/onlinepubs/9699919799/`.

[26] K. H. Hansen and F. Toolan, 'Decoding the APFS file system,' *Digital Investigation*, vol. 22, pp. 107–132, 2017.

[27] S. L. Garfinkel, 'Carving contiguous and fragmented files with fast object validation,' *digital investigation*, vol. 4, pp. 2–12, 2007.

[28] Q. Lu, S. Shi, J. Xi, J. Zeng, Y. Li and X. Mao, 'A method of time code retrieval for special format surveillance video based on file header comparison,' in *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, IEEE, 2018, pp. 1–3. DOI: `10.1109/ISDFS.2018.8355348`.

[29] R. Nordvik, K. Porter, F. Toolan, S. Axelsson and K. Franke, 'Generic metadata time carving,' *Forensic Science International: Digital Investigation*, vol. 33, p. 301 005, 2020.

[30] K. Porter, R. Nordvik, F. Toolan and S. Axelsson, 'Timestamp prefix carving for filesystem metadata extraction,' *Forensic Science International: Digital Investigation*, vol. 38, p. 301 266, 2021.

[31] Christophe Grenier, *Photorec*, version 7.1, May 2023. [Online]. Available: `https://www.cgsecurity.org/wiki/PhotoRec`.

[32] N. Fikri *et al.*, 'The analysis of file carving process using photorec and foremost,' in *2017 4th International Conference on Computer Applications and Information Processing Technology (CAIPT)*, IEEE, 2017, pp. 1–6.

[33] L. Tobin, A. Shosha and P. Gladyshev, 'Reverse engineering a CCTV system, a case study,' *Digital Investigation*, vol. 11, no. 3, pp. 179–186, 2014.

[34] S. Sandeepa, A. Reyaz and M. Silpa, 'An efficient approach to recover CCTV video from proprietary dvr file system,' in *2018 International CET Conference on Control, Communication, and Computing (IC4)*, IEEE, 2018, pp. 250–254.

[35] J. Han, D. Jeong and S. Lee, 'Analysis of the HIKVISION DVR file system,' in *Digital Forensics and Cyber Crime: 7th International Conference, ICDF2C 2015, Seoul, South Korea, October 6-8, 2015. Revised Selected Papers 7*, Springer, 2015, pp. 189–199.

[36] S. S. Inc., *010 editor*, version 13.0.1, Oct. 2022. [Online]. Available: `https://www.sweetscape.com/010editor/`.

[37] https://github.com/mradionov, *H264-bitstream-viewer*, version 0.1.0, Jan. 2022. [Online]. Available: `https://github.com/mradionov/h264-bitstream-viewer/`.

[38] Milestone Systems A/S, *Milestone XProtect*, version 22.2a, Oct. 2022. [Online]. Available: `https://www.milestonesys.com`.

[39] Mirasys, *Mirasys VMS*, version 9.4.0.1, Jan. 2023. [Online]. Available: `https://www.mirasys.com`.

[40] Motorola Solutions, *Avigilon Control Center*, version 7.14, Oct. 2022. [Online]. Available: `https://www.avigilon.com`.

[41] Detec AS, *Detec Next VMS*, version 2.1.2207, Oct. 2022. [Online]. Available: `https://detec.no/`.

[42] iSpyConnect, *Agent DVR*, version 4.2.6, Oct. 2022. [Online]. Available: `https://www.ispyconnect.com`.

[43] 'Matroska.' (May 2023), [Online]. Available: `https://matroska.org/`.

[44] S. Garfinkel, A. Nelson, D. White and V. Roussev, 'Using purpose-built functions and block hashes to enable small block and sub-file forensics,' *digital investigation*, vol. 7, S13–S23, 2010.

[45] K.-H. Hansen, 'Blockhashing as a forensic method,' M.S. thesis, University College Dublin, 2016.

[46] M. Shannon, 'Forensic relative strength scoring: ASCII and entropy scoring,' *International Journal of Digital Evidence*, vol. 2, no. 4, pp. 1–19, 2004.

[47] E. Casey, A. Nelson and J. Hyde, 'Standardization of file recovery classification and authentication,' *Digital Investigation*, vol. 31, p. 100 873, 2019.

[48] FFmpeg Developers, *Ffmpeg tool*, version 4.4.2, May 2023. [Online]. Available: `http://ffmpeg.org/`.